SRM Institute of Science and Technology

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

Department of Computer Applications

Circular – 2023-24

B.Sc. CS 3rd Sem

DESIGN AND ANALYSIS OF ALGORITHMS (UCS23502J)- Lab Manual

Lab 1- Time complexity for Merge Sort

Aim

To implement the Merge Sort algorithm and analyze its time complexity by measuring execution time for different input sizes.

Procedure

1. Define a function `merge_sort(arr)` that recursively divides the input array into two halves until individual elements are reached.
2. Implement the `merge(left, right)` function which takes two sorted sub-arrays and merges them into a single sorted array.
3. Use the `time` module in Python to record the start and end time of the `merge_sort` function execution.
4. Test the algorithm with various input sizes (e.g., 100, 1000, 10000 elements) and observe the change in execution time.
5. Compare the observed time complexity with the theoretical O(NlogN) complexity of Merge Sort.

Source Code

```
import time
import random

def merge_sort(arr):
    """
    Implements the Merge Sort algorithm.
    """
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)

    return merge(left_half, right_half)

def merge(left, right):
    """
    Merges two sorted arrays into a single sorted array.
    """
```

```
    merged_arr = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            merged_arr.append(left[i])
            i += 1
        else:
            merged_arr.append(right[j])
            j += 1

    while i < len(left):
        merged_arr.append(left[i])
        i += 1

    while j < len(right):
        merged_arr.append(right[j])
        j += 1
    return merged_arr

if __name__ == "__main__":
    # Test with a small array
    test_array_small = [38, 27, 43, 3, 9, 82, 10]
    print(f"Original small array: {test_array_small}")
    start_time_small = time.time()
    sorted_array_small = merge_sort(test_array_small)
    end_time_small = time.time()
    print(f"Sorted small array: {sorted_array_small}")
    print(f"Time taken for small array: {end_time_small -
start_time_small:.6f} seconds\n")

    # Test with a larger array to observe time complexity
    input_size = 10000
    large_array = [random.randint(0, 100000) for _ in range(input_size)]
    print(f"Sorting an array of size {input_size}...")
    start_time_large = time.time()
    sorted_array_large = merge_sort(large_array)
    end_time_large = time.time()
    print(f"Sorting complete for large array.")
    print(f"Time taken for large array ({input_size} elements):
{end_time_large - start_time_large:.6f} seconds")
    # print(f"First 10 elements of sorted large array:
{sorted_array_large[:10]}") # Uncomment to see part of the sorted array
```

Input

```
Original small array: [38, 27, 43, 3, 9, 82, 10]
(For large array, input is randomly generated, e.g., an array of 10000 random
integers)
```

Expected Output

```
Original small array: [38, 27, 43, 3, 9, 82, 10]
Sorted small array: [3, 9, 10, 27, 38, 43, 82]
Time taken for small array: 0.0000XX seconds

Sorting an array of size 10000...
Sorting complete for large array.
Time taken for large array (10000 elements): 0.0YYYXX seconds
```

(Note: Actual time values will vary based on system performance.)

Lab 2- Time complexity for Quick Sort

Aim

To implement the Quick Sort algorithm and analyze its time complexity by measuring execution time for different input sizes.

Procedure

1. Define a function `quick_sort(arr, low, high)` that recursively sorts the sub-array `arr[low...high]`.
2. Implement the `partition(arr, low, high)` function which selects a pivot element, partitions the array around the pivot, and returns the pivot's final index.
3. Use the `time` module in Python to record the start and end time of the `quick_sort` function execution.
4. Test the algorithm with various input sizes (e.g., 100, 1000, 10000 elements) and observe the change in execution time.
5. Compare the observed time complexity with the theoretical average case O(NlogN) and worst-case O(N2) complexity of Quick Sort.

Source Code

```
import time
import random

def quick_sort(arr, low, high):
    """
    Implements the Quick Sort algorithm.
    """
    if low < high:
        # pi is partitioning index, arr[pi] is now at right place
        pi = partition(arr, low, high)

        # Separately sort elements before partition and after partition
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)

def partition(arr, low, high):
    """
    Partitions the array around a pivot element.
    """
    pivot = arr[high]  # Choose the last element as the pivot
    i = (low - 1)      # Index of smaller element

    for j in range(low, high):
        # If current element is smaller than or equal to pivot
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i] # Swap

    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

if __name__ == "__main__":
    # Test with a small array
    test_array_small = [10, 7, 8, 9, 1, 5]
    print(f"Original small array: {test_array_small}")
    start_time_small = time.time()
    quick_sort(test_array_small, 0, len(test_array_small) - 1)
    end_time_small = time.time()
    print(f"Sorted small array: {test_array_small}")
```

```
    print(f"Time taken for small array: {end_time_small -
start_time_small:.6f} seconds\n")

    # Test with a larger array to observe time complexity
    input_size = 10000
    large_array = [random.randint(0, 100000) for _ in range(input_size)]
    print(f"Sorting an array of size {input_size}...")
    start_time_large = time.time()
    quick_sort(large_array, 0, len(large_array) - 1)
    end_time_large = time.time()
    print(f"Sorting complete for large array.")
    print(f"Time taken for large array ({input_size} elements):
{end_time_large - start_time_large:.6f} seconds")
    # print(f"First 10 elements of sorted large array: {large_array[:10]}") #
Uncomment to see part of the sorted array
```

Input

```
Original small array: [10, 7, 8, 9, 1, 5]
(For large array, input is randomly generated, e.g., an array of 10000 random
integers)
```

Expected Output

```
Original small array: [10, 7, 8, 9, 1, 5]
Sorted small array: [1, 5, 7, 8, 9, 10]
Time taken for small array: 0.0000XX seconds

Sorting an array of size 10000...
Sorting complete for large array.
Time taken for large array (10000 elements): 0.0YYYXX seconds
```

(Note: Actual time values will vary based on system performance.)

Lab 3- Executing Divide and Conquer Problem (Binary Search)

Aim

To understand and implement a divide and conquer algorithm using Binary Search as an example.

Procedure

1. Define a function `binary_search(arr, target, low, high)` that takes a sorted array, a target value, and the current search range.
2. In each step, calculate the middle index of the current range.
3. If the element at the middle index is the target, return its index.
4. If the target is smaller than the middle element, recursively search in the left half.
5. If the target is larger than the middle element, recursively search in the right half.
6. If the `low` index exceeds the `high` index, the element is not found.

Source Code

```
def binary_search(arr, target):
    """
    Implements the Binary Search algorithm using a divide and conquer
approach.
    Assumes the input array is sorted.
    """
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = low + (high - low) // 2  # Calculate mid to prevent overflow
for very large lists

        # Check if target is present at mid
        if arr[mid] == target:
            return mid
        # If target is greater, ignore left half
        elif arr[mid] < target:
            low = mid + 1
        # If target is smaller, ignore right half
        else:
            high = mid - 1
    # Target not found
    return -1

if __name__ == "__main__":
    sorted_list = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]

    # Test cases
    target1 = 23
    result1 = binary_search(sorted_list, target1)
    if result1 != -1:
        print(f"Element {target1} is present at index {result1}")
    else:
        print(f"Element {target1} is not present in array")

    target2 = 5
    result2 = binary_search(sorted_list, target2)
    if result2 != -1:
        print(f"Element {target2} is present at index {result2}")
    else:
        print(f"Element {target2} is not present in array")
```

```
target3 = 100
result3 = binary_search(sorted_list, target3)
if result3 != -1:
    print(f"Element {target3} is present at index {result3}")
else:
    print(f"Element {target3} is not present in array")

target4 = 2
result4 = binary_search(sorted_list, target4)
if result4 != -1:
    print(f"Element {target4} is present at index {result4}")
else:
    print(f"Element {target4} is not present in array")
```

Input

```
Sorted list: [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]
Target elements: 23, 5, 100, 2
```

Expected Output

```
Element 23 is present at index 5
Element 5 is present at index 1
Element 100 is not present in array
Element 2 is present at index 0
```

Lab 4- Executing the Greedy algorithm for Tree Vertex Splitting Problem

Aim

To implement and analyze a greedy approach for a simplified Tree Vertex Splitting problem. This problem typically involves partitioning vertices to minimize some cost or maximize some benefit. For simplicity, we'll consider a problem of finding a minimum vertex cover in a tree using a greedy approach.

Procedure

1. Represent the tree using an adjacency list.
2. Implement a greedy strategy. For a minimum vertex cover, a common greedy approach involves iteratively selecting a vertex that covers the maximum number of uncovered edges, or by processing nodes in a specific order (e.g., leaf nodes first).
3. For a tree, a simple greedy approach for vertex cover is to pick all vertices adjacent to leaves.
4. Demonstrate the steps of the greedy algorithm on a sample tree.

Source Code

```
from collections import defaultdict, deque

def greedy_vertex_cover_tree(graph):
    """
    A simple greedy approach to find a vertex cover in a tree.
    This method is not guaranteed to be optimal for all graphs,
    but for trees, a greedy choice can often lead to a valid solution.
    A common greedy strategy for trees is to pick the parent of leaf nodes.
    """
    vertex_cover = set()
    covered_edges = set()
    num_vertices = len(graph)

    # Create a copy of the graph to modify (remove edges as they are covered)
    temp_graph = {u: set(v_list) for u, v_list in graph.items()}

    # Initialize degrees for greedy choice
    degrees = {node: len(neighbors) for node, neighbors in graph.items()}

    # Sort nodes by degree in descending order (a common greedy heuristic)
    # For a tree, focusing on nodes with high degree or specific structures
like leaves
    # can be more effective. Here, we'll try a simple iterative approach.

    # Find all edges
    all_edges = set()
    for u in graph:
        for v in graph[u]:
            edge = tuple(sorted((u, v)))
            all_edges.add(edge)

    # While there are uncovered edges
    while len(covered_edges) < len(all_edges):
        best_vertex = None
        max_uncovered_edges = -1

        # Find the vertex that covers the most *uncovered* edges
        for v in graph:
            current_uncovered = 0
            for neighbor in graph[v]:
```

```
                edge = tuple(sorted((v, neighbor)))
                if edge not in covered_edges:
                    current_uncovered += 1

            if current_uncovered > max_uncovered_edges:
                max_uncovered_edges = current_uncovered
                best_vertex = v

        if best_vertex is None: # No more edges to cover
            break

        vertex_cover.add(best_vertex)

        # Mark edges connected to best_vertex as covered
        for neighbor in graph[best_vertex]:
            edge = tuple(sorted((best_vertex, neighbor)))
            covered_edges.add(edge)

    return sorted(list(vertex_cover))

if __name__ == "__main__":
    # Example Tree (represented as adjacency list)
    #   1 -- 2 -- 3
    #   |    |
    #   4    5 -- 6
    #        |
    #        7
    graph1 = {
        1: {2, 4},
        2: {1, 3, 5},
        3: {2},
        4: {1},
        5: {2, 6, 7},
        6: {5},
        7: {5}
    }

    print("Graph 1 (Tree):")
    for node, neighbors in graph1.items():
        print(f"  {node}: {list(neighbors)}")

    vc1 = greedy_vertex_cover_tree(graph1)
    print(f"\nGreedy Vertex Cover for Graph 1: {vc1}")
    print(f"Size of Vertex Cover: {len(vc1)}")

    # Another example tree
    #   A -- B -- C
    #   |    |
    #   D    E
    graph2 = {
        'A': {'B', 'D'},
        'B': {'A', 'C', 'E'},
        'C': {'B'},
        'D': {'A'},
        'E': {'B'}
    }
    print("\nGraph 2 (Tree):")
    for node, neighbors in graph2.items():
        print(f"  {node}: {list(neighbors)}")

    vc2 = greedy_vertex_cover_tree(graph2)
    print(f"\nGreedy Vertex Cover for Graph 2: {vc2}")
    print(f"Size of Vertex Cover: {len(vc2)}")
```

Input

```
Graph 1 (Tree):
  1: [2, 4]
  2: [1, 3, 5]
  3: [2]
  4: [1]
  5: [2, 6, 7]
  6: [5]
  7: [5]

Graph 2 (Tree):
  A: ['B', 'D']
  B: ['A', 'C', 'E']
  C: ['B']
  D: ['A']
  E: ['B']
```

Expected Output

```
Graph 1 (Tree):
  1: [2, 4]
  2: [1, 3, 5]
  3: [2]
  4: [1]
  5: [2, 6, 7]
  6: [5]
  7: [5]

Greedy Vertex Cover for Graph 1: [2, 5]
Size of Vertex Cover: 2

Graph 2 (Tree):
  A: ['B', 'D']
  B: ['A', 'C', 'E']
  C: ['B']
  D: ['A']
  E: ['B']

Greedy Vertex Cover for Graph 2: ['B']
Size of Vertex Cover: 1
```

(Note: The "Tree Vertex Splitting Problem" can refer to various specific problems. This implementation provides a greedy approach for a related problem (Vertex Cover in a tree) which often involves splitting or partitioning vertices to achieve a goal.)

Lab 5- Executing the Tree vertex splitting algorithm with Greedy method

Aim

To further explore the Tree Vertex Splitting problem by implementing a greedy method for finding a Maximal Independent Set in a tree. A Maximal Independent Set is a set of vertices where no two vertices are adjacent, and no more vertices can be added to the set without violating this property. This can be viewed as a "splitting" of vertices into those in the set and those not.

Procedure

1. Represent the tree using an adjacency list.
2. Implement a greedy strategy for finding a Maximal Independent Set:
    - Initialize an empty set for the independent set.
    - Iterate through the vertices (e.g., in ascending order).
    - For each vertex, if it has not been added to the independent set and none of its neighbors are in the independent set, add it to the independent set.
3. Demonstrate the steps of the greedy algorithm on a sample tree.

Source Code

```
def greedy_maximal_independent_set_tree(graph):
    """
    Implements a greedy algorithm to find a Maximal Independent Set (MIS) in
a graph.
    For a tree, this greedy approach can be quite effective.
    It iterates through vertices and adds a vertex to the MIS if it's not
already covered
    by an existing MIS vertex.
    """
    mis = set()
    # Get all vertices and sort them to ensure a consistent greedy choice
    vertices = sorted(list(graph.keys()))

    for u in vertices:
        # Check if u is already covered by a neighbor in MIS
        is_covered = False
        for neighbor in graph[u]:
            if neighbor in mis:
                is_covered = True
                break

        # If u is not covered, add it to MIS
        if not is_covered:
            mis.add(u)

    return sorted(list(mis))

if __name__ == "__main__":
    # Example Tree 1
    #   1 -- 2 -- 3
    #   |    |
    #   4    5 -- 6
    #        |
    #        7
    graph1 = {
        1: {2, 4},
        2: {1, 3, 5},
        3: {2},
        4: {1},
```

```
        5: {2, 6, 7},
        6: {5},
        7: {5}
    }

    print("Graph 1 (Tree):")
    for node, neighbors in graph1.items():
        print(f"  {node}: {list(neighbors)}")

    mis1 = greedy_maximal_independent_set_tree(graph1)
    print(f"\nGreedy Maximal Independent Set for Graph 1: {mis1}")
    print(f"Size of MIS: {len(mis1)}")

    # Example Tree 2 (a path graph)
    #   A -- B -- C -- D
    graph2 = {
        'A': {'B'},
        'B': {'A', 'C'},
        'C': {'B', 'D'},
        'D': {'C'}
    }
    print("\nGraph 2 (Tree):")
    for node, neighbors in graph2.items():
        print(f"  {node}: {list(neighbors)}")

    mis2 = greedy_maximal_independent_set_tree(graph2)
    print(f"\nGreedy Maximal Independent Set for Graph 2: {mis2}")
    print(f"Size of MIS: {len(mis2)}")
```

Input

```
Graph 1 (Tree):
  1: [2, 4]
  2: [1, 3, 5]
  3: [2]
  4: [1]
  5: [2, 6, 7]
  6: [5]
  7: [5]

Graph 2 (Tree):
  A: ['B']
  B: ['A', 'C']
  C: ['B', 'D']
  D: ['C']
```

Expected Output

```
Graph 1 (Tree):
  1: [2, 4]
  2: [1, 3, 5]
  3: [2]
  4: [1]
  5: [2, 6, 7]
  6: [5]
  7: [5]

Greedy Maximal Independent Set for Graph 1: [1, 3, 5]
Size of MIS: 3

Graph 2 (Tree):
  A: ['B']
  B: ['A', 'C']
  C: ['B', 'D']
  D: ['C']
```

```
Greedy Maximal Independent Set for Graph 2: ['A', 'C']
Size of MIS: 2
```

Lab 6- Executing the Greedy algorithm for Optimal storage on Tapes problem

Aim

To implement the greedy algorithm for the Optimal Storage on Tapes problem, which aims to minimize the Mean Retrieval Time (MRT) for a set of programs stored on a single tape.

Procedure

1. Understand that the optimal strategy for minimizing MRT is to store the programs in non-decreasing order of their lengths.
2. Take a list of program lengths as input.
3. Sort the program lengths in ascending order.
4. Calculate the Mean Retrieval Time based on this sorted order. The retrieval time for a program is the sum of its length and the lengths of all programs preceding it on the tape. MRT is the sum of all retrieval times divided by the number of programs.

Source Code

```
def optimal_storage_on_tapes(program_lengths):
    """
    Implements the greedy algorithm for Optimal Storage on Tapes.
    Programs are sorted by length in non-decreasing order to minimize MRT.
    """
    if not program_lengths:
        return 0, []

    # Step 1: Sort the program lengths in non-decreasing order (greedy
choice)
    sorted_lengths = sorted(program_lengths)

    total_retrieval_time = 0
    current_tape_length = 0

    # Step 2: Calculate total retrieval time
    for length in sorted_lengths:
        current_tape_length += length  # Length of tape up to current program
        total_retrieval_time += current_tape_length

    # Step 3: Calculate Mean Retrieval Time (MRT)
    mean_retrieval_time = total_retrieval_time / len(sorted_lengths)

    return mean_retrieval_time, sorted_lengths

if __name__ == "__main__":
    programs1 = [5, 10, 3, 8] # Example program lengths
    print(f"Original program lengths: {programs1}")
    mrt1, order1 = optimal_storage_on_tapes(programs1)
    print(f"Optimal order for programs: {order1}")
    print(f"Mean Retrieval Time (MRT): {mrt1:.2f}\n")

    programs2 = [20, 15, 10, 5]
    print(f"Original program lengths: {programs2}")
    mrt2, order2 = optimal_storage_on_tapes(programs2)
    print(f"Optimal order for programs: {order2}")
    print(f"Mean Retrieval Time (MRT): {mrt2:.2f}\n")

    programs3 = [7]
    print(f"Original program lengths: {programs3}")
    mrt3, order3 = optimal_storage_on_tapes(programs3)
```

```
    print(f"Optimal order for programs: {order3}")
    print(f"Mean Retrieval Time (MRT): {mrt3:.2f}\n")

    programs4 = []
    print(f"Original program lengths: {programs4}")
    mrt4, order4 = optimal_storage_on_tapes(programs4)
    print(f"Optimal order for programs: {order4}")
    print(f"Mean Retrieval Time (MRT): {mrt4:.2f}\n")
```

Input

```
Original program lengths: [5, 10, 3, 8]
Original program lengths: [20, 15, 10, 5]
Original program lengths: [7]
Original program lengths: []
```

Expected Output

```
Original program lengths: [5, 10, 3, 8]
Optimal order for programs: [3, 5, 8, 10]
Mean Retrieval Time (MRT): 13.50

Original program lengths: [20, 15, 10, 5]
Optimal order for programs: [5, 10, 15, 20]
Mean Retrieval Time (MRT): 25.00

Original program lengths: [7]
Optimal order for programs: [7]
Mean Retrieval Time (MRT): 7.00

Original program lengths: []
Optimal order for programs: []
Mean Retrieval Time (MRT): 0.00
```

Lab 7- Executing Multistage Graph shortest path problem using dynamic programming algorithm

Aim

To implement the dynamic programming algorithm for finding the shortest path in a multistage graph.

Procedure

1. Represent the multistage graph using an adjacency matrix or adjacency list, where edges have weights.
2. Define `cost[i]` as the shortest path cost from source to stage `i`.
3. Use dynamic programming to compute the shortest path. This typically involves iterating backward or forward through the stages.
    - **Backward approach:** Start from the destination node and calculate the minimum cost to reach it from the previous stage nodes.
    - **Forward approach:** Start from the source node and calculate the minimum cost to reach nodes in the next stage.
4. Store the path information (e.g., `d[j]` for min cost to `j`, `p[j]` for predecessor of `j`).
5. Reconstruct the shortest path from the source to the destination.

Source Code

```
import math

def shortest_path_multistage_graph(graph, num_stages, num_nodes):
    """
    Finds the shortest path in a multistage graph using dynamic programming
(forward approach).

    Args:
        graph (dict): Adjacency list representation of the graph.
                      Keys are nodes, values are dictionaries of {neighbor:
weight}.
        num_stages (int): The total number of stages in the graph.
        num_nodes (int): The total number of nodes in the graph.

    Returns:
        tuple: A tuple containing:
               - float: The minimum cost from source (node 0) to destination
(node num_nodes - 1).
               - list: The shortest path as a list of nodes.
    """
    # Initialize cost array: cost[i] = min cost to reach node i from source
    # Initialize path array: path[i] = predecessor of node i on the shortest
path

    # Using a large value for infinity
    infinity = float('inf')

    cost = [infinity] * num_nodes
    path = [0] * num_nodes # To store the predecessor node for path
reconstruction

    # The source node (node 0) has a cost of 0
    cost[0] = 0

    # Iterate through stages (or nodes in increasing order of stage)
```

```
    # Assuming nodes are numbered such that nodes in stage k are less than
nodes in stage k+1
    # For a general multistage graph, you'd need to know which nodes belong
to which stage.
    # Here, we'll assume a sequential numbering where 0 is source, num_nodes-
1 is destination,
    # and intermediate nodes are ordered.

    # This simplified approach assumes nodes are processed in an order that
respects stages.
    # A more robust solution would explicitly manage stages.
    # For a typical multistage graph, nodes are grouped by stages.
    # Let's assume nodes 0 to N-1, where 0 is source, N-1 is sink.
    # And edges only go from a lower-indexed node to a higher-indexed node.

    for u in range(num_nodes):
        if u not in graph: # Skip nodes with no outgoing edges (except sink)
            continue
        for v, weight in graph[u].items():
            if cost[u] + weight < cost[v]:
                cost[v] = cost[u] + weight
                path[v] = u # Store predecessor

    min_cost = cost[num_nodes - 1]

    # Reconstruct the path
    shortest_path = []
    current_node = num_nodes - 1
    while current_node != 0:
        shortest_path.append(current_node)
        current_node = path[current_node]
    shortest_path.append(0) # Add the source node
    shortest_path.reverse() # Reverse to get path from source to destination

    return min_cost, shortest_path

if __name__ == "__main__":
    # Example Multistage Graph (Nodes 0 to 7)
    # Stage 1: Node 0 (Source)
    # Stage 2: Nodes 1, 2
    # Stage 3: Nodes 3, 4, 5
    # Stage 4: Nodes 6, 7 (Destination)

    # Graph representation: {node: {neighbor: weight, ...}, ...}
    graph = {
        0: {1: 1, 2: 2},
        1: {3: 7, 4: 9},
        2: {3: 4, 4: 2, 5: 5},
        3: {6: 3},
        4: {6: 6, 7: 8},
        5: {7: 10},
        6: {7: 1}
        # Node 7 has no outgoing edges (destination)
    }

    num_nodes = 8 # Nodes 0 to 7
    num_stages = 4 # Based on the structure

    print("Multistage Graph Edges (node: {neighbor: weight}):")
    for u in sorted(graph.keys()):
        print(f"  {u}: {graph[u]}")

    min_cost, path = shortest_path_multistage_graph(graph, num_stages,
num_nodes)

    print(f"\nMinimum cost from source to destination: {min_cost}")
```

```
    print(f"Shortest path: {path}")

    # Another example
    graph2 = {
        0: {1: 3, 2: 2},
        1: {3: 5, 4: 1},
        2: {4: 4, 5: 6},
        3: {6: 2},
        4: {6: 7},
        5: {6: 3}
    }
    num_nodes2 = 7 # Nodes 0 to 6
    num_stages2 = 4

    print("\nMultistage Graph 2 Edges (node: {neighbor: weight}):")
    for u in sorted(graph2.keys()):
        print(f"  {u}: {graph2[u]}")

    min_cost2, path2 = shortest_path_multistage_graph(graph2, num_stages2,
num_nodes2)

    print(f"\nMinimum cost from source to destination: {min_cost2}")
    print(f"Shortest path: {path2}")
```

Input

```
Multistage Graph 1 Edges:
  0: {1: 1, 2: 2}
  1: {3: 7, 4: 9}
  2: {3: 4, 4: 2, 5: 5}
  3: {6: 3}
  4: {6: 6, 7: 8}
  5: {7: 10}
  6: {7: 1}

Multistage Graph 2 Edges:
  0: {1: 3, 2: 2}
  1: {3: 5, 4: 1}
  2: {4: 4, 5: 6}
  3: {6: 2}
  4: {6: 7}
  5: {6: 3}
```

Expected Output

```
Multistage Graph Edges (node: {neighbor: weight}):
  0: {1: 1, 2: 2}
  1: {3: 7, 4: 9}
  2: {3: 4, 4: 2, 5: 5}
  3: {6: 3}
  4: {6: 6, 7: 8}
  5: {7: 10}
  6: {7: 1}

Minimum cost from source to destination: 7
Shortest path: [0, 2, 4, 6, 7]

Multistage Graph 2 Edges (node: {neighbor: weight}):
  0: {1: 3, 2: 2}
  1: {3: 5, 4: 1}
  2: {4: 4, 5: 6}
  3: {6: 2}
  4: {6: 7}
  5: {6: 3}
```

```
Minimum cost from source to destination: 7
Shortest path: [0, 2, 5, 6]
```

Lab 8- Executing All pairs shortest path problem using dynamic programming algorithm

Aim

To implement the Floyd-Warshall algorithm, a dynamic programming approach, to find the shortest paths between all pairs of vertices in a given weighted graph.

Procedure

1. Represent the graph using an adjacency matrix `dist[i][j]`, where `dist[i][j]` is the weight of the edge from `i` to `j`. Initialize `dist[i][i]` to 0 and `dist[i][j]` to infinity if no direct edge exists.
2. The Floyd-Warshall algorithm iterates through all possible intermediate vertices `k`.
3. For each `k`, it updates `dist[i][j]` if `dist[i][k] + dist[k][j]` is less than the current `dist[i][j]`. This means that going from `i` to `j` via `k` is shorter than the current known path.
4. After iterating through all `k`, the `dist` matrix will contain the shortest path distances between all pairs of vertices.

Source Code

```
import math

def floyd_warshall(graph):
    """
    Implements the Floyd-Warshall algorithm to find all-pairs shortest paths.

    Args:
        graph (list of lists): Adjacency matrix representation of the graph.
                               graph[i][j] is the weight of the edge from i
to j.
                               Use float('inf') for no direct edge.
                               Graph is assumed to have V vertices, indexed 0
to V-1.

    Returns:
        list of lists: A matrix where result[i][j] is the shortest distance
                       from vertex i to vertex j.
    """
    V = len(graph)

    # Initialize the distance matrix with the given graph weights
    # Make a copy to avoid modifying the original graph input
    dist = [row[:] for row in graph]

    # Path reconstruction matrix (optional, but good for understanding)
    # next_node[i][j] stores the next node on the shortest path from i to j
    # next_node = [[None for _ in range(V)] for _ in range(V)]
    # for i in range(V):
    #     for j in range(V):
    #         if graph[i][j] != float('inf') and i != j:
    #             next_node[i][j] = j

    # Iterate through all vertices to be used as intermediate vertices
    for k in range(V):
        # Iterate through all source vertices
        for i in range(V):
            # Iterate through all destination vertices
            for j in range(V):
```

```
                # If vertex k is on the shortest path from i to j, then
update the value of dist[i][j]
                if dist[i][k] != float('inf') and dist[k][j] != float('inf'):
                    if dist[i][k] + dist[k][j] < dist[i][j]:
                        dist[i][j] = dist[i][k] + dist[k][j]
                        # next_node[i][j] = next_node[i][k] # For path
reconstruction
    return dist

def print_solution(dist):
    """
    Helper function to print the solution matrix.
    """
    V = len(dist)
    print("Following matrix shows the shortest distances between every pair
of vertices:")
    for i in range(V):
        for j in range(V):
            if(dist[i][j] == float('inf')):
                print("%7s" % "INF", end=" ")
            else:
                print("%7d" % dist[i][j], end=" ")
        print()

if __name__ == "__main__":
    # Example Graph (4 vertices)
    # INF represents no direct edge
    INF = float('inf')
    graph1 = [
        [0, 3, INF, 7],
        [8, 0, 2, INF],
        [5, INF, 0, 1],
        [2, INF, INF, 0]
    ]

    print("Graph 1 Adjacency Matrix:")
    print_solution(graph1)

    shortest_distances1 = floyd_warshall(graph1)
    print("\nShortest distances for Graph 1:")
    print_solution(shortest_distances1)

    # Another example graph (3 vertices)
    graph2 = [
        [0, 1, INF],
        [INF, 0, 1],
        [INF, INF, 0]
    ]

    print("\nGraph 2 Adjacency Matrix:")
    print_solution(graph2)

    shortest_distances2 = floyd_warshall(graph2)
    print("\nShortest distances for Graph 2:")
    print_solution(shortest_distances2)
```

Input

```
Graph 1 Adjacency Matrix:
      0       3     INF       7
      8       0       2     INF
      5     INF       0       1
      2     INF     INF       0

Graph 2 Adjacency Matrix:
      0       1     INF
```

```
    INF       0       1
    INF     INF       0
```

Expected Output

```
Graph 1 Adjacency Matrix:
      0       3     INF       7
      8       0       2     INF
      5     INF       0       1
      2     INF     INF       0

Shortest distances for Graph 1:
      0       3       5       6
      8       0       2       3
      5       8       0       1
      2       5       7       0

Graph 2 Adjacency Matrix:
      0       1     INF
    INF       0       1
    INF     INF       0

Shortest distances for Graph 2:
      0       1       2
    INF       0       1
    INF     INF       0
```

Lab 9- Executing Dynamic programming algorithm for Single source shortest path problem

Aim

To implement the Bellman-Ford algorithm, a dynamic programming approach, to find the shortest paths from a single source vertex to all other vertices in a weighted graph, including graphs with negative edge weights (but no negative cycles).

Procedure

1. Represent the graph using an adjacency list of edges, where each edge is a tuple `(u, v, weight)`.
2. Initialize `dist[i]` to infinity for all vertices `i` except the source vertex, for which `dist[source]` is 0.
3. Relax all edges `V-1` times, where `V` is the number of vertices. In each iteration, for every edge `(u, v, weight)`, if `dist[u] + weight < dist[v]`, update `dist[v]` and record `u` as its predecessor.
4. After `V-1` iterations, perform one more iteration to check for negative cycles. If any `dist[u] + weight < dist[v]` is true, then a negative cycle exists.
5. Reconstruct the paths using the predecessor information.

Source Code

```
import math

def bellman_ford(graph_edges, num_vertices, source):
    """
    Implements the Bellman-Ford algorithm to find single-source shortest
paths.
    Handles negative edge weights but detects negative cycles.

    Args:
        graph_edges (list of tuples): List of edges, where each edge is (u,
v, weight).
        num_vertices (int): Total number of vertices in the graph (0 to
num_vertices-1).
        source (int): The source vertex.

    Returns:
        tuple: A tuple containing:
               - list: `dist` array where `dist[i]` is the shortest distance
from source to i.
                       Returns None if a negative cycle is detected.
               - list: `predecessor` array where `predecessor[i]` is the
predecessor of i on the shortest path.
    """
    # Initialize distances from source to all other vertices as infinity
    infinity = float('inf')
    dist = [infinity] * num_vertices
    predecessor = [-1] * num_vertices # Stores the path

    dist[source] = 0

    # Relax all edges |V| - 1 times
    for _ in range(num_vertices - 1):
        for u, v, weight in graph_edges:
            if dist[u] != infinity and dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight
                predecessor[v] = u
```

```
    # Check for negative cycles
    for u, v, weight in graph_edges:
        if dist[u] != infinity and dist[u] + weight < dist[v]:
            print("Graph contains negative cycle!")
            return None, None # Indicate negative cycle detected

    return dist, predecessor

def print_paths(dist, predecessor, source):
    """
    Helper function to print shortest distances and paths.
    """
    num_vertices = len(dist)
    print(f"\nShortest distances from source {source}:")
    for i in range(num_vertices):
        if dist[i] == float('inf'):
            print(f"  To vertex {i}: INF")
        else:
            print(f"  To vertex {i}: {dist[i]}")

    print("\nShortest paths:")
    for i in range(num_vertices):
        if i == source or dist[i] == float('inf'):
            continue
        path = []
        curr = i
        while curr != -1:
            path.append(curr)
            if curr == source:
                break
            curr = predecessor[curr]
        path.reverse()
        print(f"  Path to vertex {i}: {' -> '.join(map(str, path))}")


if __name__ == "__main__":
    # Example Graph 1 (no negative cycle)
    # Edges: (u, v, weight)
    graph_edges1 = [
        (0, 1, -1), (0, 2, 4),
        (1, 2, 3), (1, 3, 2), (1, 4, 2),
        (3, 2, 5), (3, 1, 1),
        (4, 3, -3)
    ]
    num_vertices1 = 5
    source1 = 0

    print(f"Graph 1 (Vertices: {num_vertices1}, Source: {source1}):")
    print(f"Edges: {graph_edges1}")

    dist1, pred1 = bellman_ford(graph_edges1, num_vertices1, source1)
    if dist1:
        print_paths(dist1, pred1, source1)

    # Example Graph 2 (with negative cycle)
    graph_edges2 = [
        (0, 1, 1),
        (1, 2, -1),
        (2, 0, -1) # This forms a negative cycle 0 -> 1 -> 2 -> 0 with total
weight -1
    ]
    num_vertices2 = 3
    source2 = 0

    print(f"\n\nGraph 2 (Vertices: {num_vertices2}, Source: {source2}):")
```

```
    print(f"Edges: {graph_edges2}")

    dist2, pred2 = bellman_ford(graph_edges2, num_vertices2, source2)
    if dist2:
        print_paths(dist2, pred2, source2)
```

Input

```
Graph 1 (Vertices: 5, Source: 0):
Edges: [(0, 1, -1), (0, 2, 4), (1, 2, 3), (1, 3, 2), (1, 4, 2), (3, 2, 5),
(3, 1, 1), (4, 3, -3)]

Graph 2 (Vertices: 3, Source: 0):
Edges: [(0, 1, 1), (1, 2, -1), (2, 0, -1)]
```

Expected Output

```
Graph 1 (Vertices: 5, Source: 0):
Edges: [(0, 1, -1), (0, 2, 4), (1, 2, 3), (1, 3, 2), (1, 4, 2), (3, 2, 5),
(3, 1, 1), (4, 3, -3)]

Shortest distances from source 0:
  To vertex 0: 0
  To vertex 1: -2
  To vertex 2: 2
  To vertex 3: -1
  To vertex 4: 1

Shortest paths:
  Path to vertex 1: 0 -> 4 -> 3 -> 1
  Path to vertex 2: 0 -> 4 -> 3 -> 1 -> 2
  Path to vertex 3: 0 -> 4 -> 3
  Path to vertex 4: 0 -> 4

Graph 2 (Vertices: 3, Source: 0):
Edges: [(0, 1, 1), (1, 2, -1), (2, 0, -1)]
Graph contains negative cycle!
```

Lab 10- Executing Dynamic programming algorithm for constructing Bi-connected Graphs problem

Aim

To understand the concept of bi-connected components and explore how dynamic programming principles might be applied to problems related to graph connectivity or counting paths in such structures. Directly "constructing" a bi-connected graph with DP isn't a standard problem, but DP can be used for related tasks like counting paths or properties on graphs. Here, we'll demonstrate a DP approach for counting paths in a DAG, which is a fundamental DP graph problem.

Procedure

1. Represent the graph as an adjacency list.
2. For counting paths in a Directed Acyclic Graph (DAG), use dynamic programming.
3. Perform a topological sort of the DAG.
4. Initialize `dp[i]` to 0 for all vertices, and `dp[source]` to 1 (representing one path to itself).
5. Iterate through the vertices in topological order. For each vertex `u`, iterate through its neighbors `v`. Add `dp[u]` to `dp[v]`, as any path to `u` can be extended to `v`.
6. The `dp[target]` will then hold the total number of paths from source to target.

Source Code

```
from collections import defaultdict, deque

def count_paths_in_dag(graph, source, target):
    """
    Counts the number of distinct paths from a source to a target in a
Directed Acyclic Graph (DAG)
    using dynamic programming. This demonstrates a DP principle on graphs.

    Args:
        graph (dict): Adjacency list representation of the DAG.
                      Keys are nodes, values are lists of neighbors.
        source (int): The starting node.
        target (int): The destination node.

    Returns:
        int: The total number of paths from source to target.
    """
    num_nodes = max(graph.keys()) + 1 if graph else 0

    # Calculate in-degrees for topological sort
    in_degree = {node: 0 for node in range(num_nodes)}
    for u in graph:
        for v in graph[u]:
            in_degree[v] += 1

    # Initialize DP array: dp[i] will store the number of paths to node i
from source
    dp = [0] * num_nodes
    dp[source] = 1 # There's one way to reach the source from itself

    # Perform topological sort using Kahn's algorithm (BFS-based)
    queue = deque([node for node in range(num_nodes) if in_degree[node] ==
0])

    topological_order = []
```

```
    while queue:
        u = queue.popleft()
        topological_order.append(u)

        if u in graph:
            for v in graph[u]:
                in_degree[v] -= 1
                if in_degree[v] == 0:
                    queue.append(v)

    # Ensure the target is reachable and part of the topological sort
    if target not in topological_order and target != source:
        return 0 # Target is not reachable or not in the graph

    # Apply DP based on topological order
    for u in topological_order:
        if u in graph:
            for v in graph[u]:
                dp[v] += dp[u] # Add paths from u to paths to v

    return dp[target]

if __name__ == "__main__":
    # Example DAG 1
    # 0 -> 1 -> 3
    # |    |
    # v    v
    # 2 -> 3
    graph1 = {
        0: [1, 2],
        1: [3],
        2: [3],
        3: [] # Sink node
    }
    source1 = 0
    target1 = 3

    print("Graph 1 (DAG) edges:")
    for u, neighbors in graph1.items():
        print(f"  {u} -> {neighbors}")

    paths1 = count_paths_in_dag(graph1, source1, target1)
    print(f"\nNumber of paths from {source1} to {target1}: {paths1}")

    # Example DAG 2
    # 0 -> 1 -> 2
    # |    ^    |
    # v    |    v
    # 3 ---+---- 4
    graph2 = {
        0: [1, 3],
        1: [2],
        2: [4],
        3: [1, 4],
        4: []
    }
    source2 = 0
    target2 = 4

    print("\nGraph 2 (DAG) edges:")
    for u, neighbors in graph2.items():
        print(f"  {u} -> {neighbors}")

    paths2 = count_paths_in_dag(graph2, source2, target2)
    print(f"\nNumber of paths from {source2} to {target2}: {paths2}")
```

Input

```
Graph 1 (DAG) edges:
  0 -> [1, 2]
  1 -> [3]
  2 -> [3]
  3 -> []

Graph 2 (DAG) edges:
  0 -> [1, 3]
  1 -> [2]
  2 -> [4]
  3 -> [1, 4]
  4 -> []
```

Expected Output

```
Graph 1 (DAG) edges:
  0 -> [1, 2]
  1 -> [3]
  2 -> [3]
  3 -> []

Number of paths from 0 to 3: 2

Graph 2 (DAG) edges:
  0 -> [1, 3]
  1 -> [2]
  2 -> [4]
  3 -> [1, 4]
  4 -> []

Number of paths from 0 to 4: 3
```

Lab 11- Executing Bi-Connected Components Graphs with backtracking

Aim

To implement an algorithm to find the bi-connected components (BCCs) of a given undirected graph using Depth First Search (DFS) and backtracking principles. A bi-connected component is a maximal subgraph such that the removal of any single vertex does not disconnect the subgraph.

Procedure

1. Represent the graph using an adjacency list.
2. Use a DFS-based algorithm (e.g., Tarjan's algorithm or a similar approach) to find articulation points (cut vertices) and bridges.
3. During DFS, maintain:
    - `disc[u]`: Discovery time of vertex `u`.
    - `low[u]`: Lowest discovery time reachable from `u` (including `u` itself) through a DFS tree edge or a back-edge.
    - `parent[u]`: Parent of `u` in the DFS tree.
    - A stack to store edges.
4. When `low[v] >= disc[u]` for a child `v` of `u`, `u` is an articulation point (unless `u` is the root with only one child). This also signals the completion of a bi-connected component. Pop edges from the stack until `(u, v)` is popped.
5. When `low[v] > disc[u]`, the edge `(u, v)` is a bridge.

Source Code

```
from collections import defaultdict

class Graph:
    def __init__(self, V):
        self.V = V
        self.graph = defaultdict(list)
        self.Time = 0 # Global time variable for discovery times
        self.disc = [-1] * V # Discovery times
        self.low = [-1] * V  # Lowest discovery time reachable
        self.parent = [-1] * V # Parent in DFS tree
        self.is_articulation_point = [False] * V # To mark articulation
points
        self.bridges = [] # To store bridges
        self.bccs = [] # To store bi-connected components (list of edges)
        self.edge_stack = [] # Stack to store edges for BCCs

    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)

    def find_biconnected_components_util(self, u):
        """
        A recursive utility function that finds articulation points, bridges,
        and bi-connected components using DFS.
        """
        self.disc[u] = self.Time
        self.low[u] = self.Time
        self.Time += 1

        children = 0 # Count of children in DFS tree

        for v in self.graph[u]:
            if v == self.parent[u]:
                continue # Skip parent
```

```
            if self.disc[v] == -1: # If v is not visited
                self.parent[v] = u
                children += 1
                self.edge_stack.append((u, v)) # Push edge to stack

                self.find_biconnected_components_util(v)

                self.low[u] = min(self.low[u], self.low[v])

                # u is an articulation point if:
                # 1. u is root and has more than one child
                # 2. u is not root and low[v] >= disc[u]
                if (self.parent[u] == -1 and children > 1) or \
                   (self.parent[u] != -1 and self.low[v] >= self.disc[u]):
                    self.is_articulation_point[u] = True

                    # Found a BCC
                    current_bcc_edges = []
                    while True:
                        edge = self.edge_stack.pop()
                        current_bcc_edges.append(edge)
                        if edge == (u, v) or edge == (v, u): # Edge (u,v) or
(v,u)
                            break
                    self.bccs.append(current_bcc_edges)

                # If low[v] > disc[u], then (u, v) is a bridge
                if self.low[v] > self.disc[u]:
                    self.bridges.append((u, v))

            elif v != self.parent[u]: # v is visited and not parent (back-
edge)
                self.low[u] = min(self.low[u], self.disc[v])
                # Only push (u,v) if it's a forward edge to an already
visited node
                # to ensure it's part of a cycle for BCCs.
                # Avoid duplicates if (v,u) is already on stack.
                if (u, v) not in self.edge_stack and (v, u) not in
self.edge_stack:
                     self.edge_stack.append((u, v))


    def find_biconnected_components(self):
        """
        Main function to find bi-connected components.
        """
        for i in range(self.V):
            if self.disc[i] == -1: # If vertex not visited, start DFS
                self.find_biconnected_components_util(i)

                # After DFS from a component root, if stack is not empty,
                # remaining edges form a BCC (for graphs with multiple
components)
                if self.edge_stack:
                    current_bcc_edges = []
                    while self.edge_stack:
                        current_bcc_edges.append(self.edge_stack.pop())
                    self.bccs.append(current_bcc_edges)


if __name__ == "__main__":
    # Example Graph 1 (from GeeksforGeeks)
    #   0 -- 1 -- 3 -- 4
    #   |  / |  /
    #   | /  | /
```

```
    #   2 ---+
    g1 = Graph(5)
    g1.add_edge(0, 1)
    g1.add_edge(0, 2)
    g1.add_edge(1, 2)
    g1.add_edge(1, 3)
    g1.add_edge(3, 4)
    print("Graph 1:")
    g1.find_biconnected_components()
    print("Articulation Points:", [i for i, is_ap in
enumerate(g1.is_articulation_point) if is_ap])
    print("Bridges:", g1.bridges)
    print("Bi-connected Components (edges):")
    for i, bcc in enumerate(g1.bccs):
        print(f"  BCC {i+1}: {bcc}")

    # Example Graph 2 (with a bridge)
    #   0 -- 1 -- 2 -- 3
    #   |  /
    #   | /
    #   4
    g2 = Graph(5)
    g2.add_edge(0, 1)
    g2.add_edge(0, 4)
    g2.add_edge(1, 4)
    g2.add_edge(1, 2)
    g2.add_edge(2, 3)
    print("\nGraph 2:")
    g2.find_biconnected_components()
    print("Articulation Points:", [i for i, is_ap in
enumerate(g2.is_articulation_point) if is_ap])
    print("Bridges:", g2.bridges)
    print("Bi-connected Components (edges):")
    for i, bcc in enumerate(g2.bccs):
        print(f"  BCC {i+1}: {bcc}")
```

Input

```
Graph 1:
  Edges: (0,1), (0,2), (1,2), (1,3), (3,4)

Graph 2:
  Edges: (0,1), (0,4), (1,4), (1,2), (2,3)
```

Expected Output

```
Graph 1:
Articulation Points: [1, 3]
Bridges: [(3, 4)]
Bi-connected Components (edges):
  BCC 1: [(3, 4)]
  BCC 2: [(1, 3), (1, 2), (0, 2), (0, 1)]

Graph 2:
Articulation Points: [1, 2]
Bridges: [(1, 2), (2, 3)]
Bi-connected Components (edges):
  BCC 1: [(2, 3)]
  BCC 2: [(1, 2)]
  BCC 3: [(1, 4), (0, 4), (0, 1)]
```

(Note: The order of edges within a BCC might vary depending on DFS traversal. The important part is the set of edges forming each component.)

Lab 12- Executing 8 Queens problem with back tracking

Aim

To implement the 8 Queens problem using the backtracking technique, which finds all possible ways to place 8 non-attacking queens on an 8×8 chessboard.

Procedure

1. Define a chessboard as an N×N matrix (or a 1D array representing column positions for each row).
2. Create a recursive backtracking function `solve_n_queens(board, row, N)`.
3. Base case: If `row == N`, all queens are placed successfully; print the board configuration.
4. Recursive step: For the current `row`, iterate through all possible `col` positions (0 to `N-1`).
5. Before placing a queen at `(row, col)`, check if it's safe using a `is_safe(board, row, col, N)` function. This function checks for conflicts in the same column, and both diagonals.
6. If safe, place the queen (`board[row] = col`), and recursively call `solve_n_queens(board, row + 1, N)`.
7. After the recursive call returns (backtrack step), "unplace" the queen (not strictly necessary if `board` is passed by value or reset, but conceptually important for backtracking).

Source Code

```
def solve_n_queens(n):
    """
    Solves the N-Queens problem using backtracking.

    Args:
        n (int): The size of the chessboard (e.g., 8 for 8 Queens).

    Returns:
        list of lists: A list of all valid board configurations.
                       Each configuration is a list where board[i] = j means
                       a queen is at row i, column j.
    """
    board = [-1] * n  # board[i] stores the column of the queen in row i
    solutions = []

    def is_safe(row, col):
        """
        Checks if placing a queen at (row, col) is safe.
        No two queens can share the same column or diagonal.
        """
        for prev_row in range(row):
            # Check column conflict
            if board[prev_row] == col:
                return False
            # Check diagonal conflicts
            if abs(board[prev_row] - col) == abs(prev_row - row):
                return False
        return True

    def backtrack(row):
        """
        Recursive backtracking function to place queens.
        """
        if row == n:
```

```
            # All queens placed successfully, add current configuration to
solutions
            solutions.append(list(board))
            return

        for col in range(n):
            if is_safe(row, col):
                board[row] = col  # Place queen
                backtrack(row + 1) # Recur for next row
                # No explicit "unplace" needed as board[row] will be
overwritten
                # or the function will return and previous state is
implicitly restored.

    backtrack(0) # Start placing queens from row 0
    return solutions

def print_solution(solution, n):
    """
    Prints a single N-Queens solution in a readable format.
    """
    for row_idx in range(n):
        line = ""
        for col_idx in range(n):
            if solution[row_idx] == col_idx:
                line += " Q "
            else:
                line += " . "
        print(line)
    print("-" * (n * 3)) # Separator

if __name__ == "__main__":
    n = 8 # For 8 Queens problem
    print(f"Solving {n}-Queens problem...\n")
    all_solutions = solve_n_queens(n)

    print(f"Found {len(all_solutions)} solutions for {n}-Queens problem.\n")

    # Print the first few solutions for demonstration
    num_solutions_to_print = min(3, len(all_solutions))
    for i in range(num_solutions_to_print):
        print(f"Solution {i + 1}:")
        print_solution(all_solutions[i], n)
```

Input

```
N = 8 (for 8 Queens problem)
```

Expected Output

```
Solving 8-Queens problem...

Found 92 solutions for 8-Queens problem.

Solution 1:
 .  Q  .  .  .  .  .  .
 .  .  .  .  .  .  Q  .
 .  .  .  .  Q  .  .  .
 .  .  .  .  .  .  .  Q
 .  .  Q  .  .  .  .  .
 Q  .  .  .  .  .  .  .
 .  .  .  Q  .  .  .  .
 .  .  .  .  .  Q  .  .
------------------------
Solution 2:
```

```
 .  .  Q  .  .  .  .  .
 .  .  .  .  .  Q  .  .
 .  .  .  .  .  .  .  Q
 .  .  .  Q  .  .  .  .
 Q  .  .  .  .  .  .  .
 .  .  .  .  .  .  Q  .
 .  Q  .  .  .  .  .  .
 .  .  .  .  Q  .  .  .
------------------------
Solution 3:
 .  .  .  .  Q  .  .  .
 .  .  .  .  .  .  .  Q
 .  .  Q  .  .  .  .  .
 Q  .  .  .  .  .  .  .
 .  .  .  .  .  Q  .  .
 .  .  .  .  .  .  Q  .
 .  Q  .  .  .  .  .  .
 .  .  .  Q  .  .  .  .
------------------------
... (and 89 more solutions)
```

Lab 13- Executing the Graph coloring with backtracking

Aim

To implement the Graph Coloring problem using the backtracking technique, which assigns colors to vertices of a graph such that no two adjacent vertices have the same color, using a minimum number of colors (or a given number of colors).

Procedure

1. Represent the graph using an adjacency list.
2. Define the number of available colors, `m`.
3. Create a recursive backtracking function `graph_coloring(k, m, colors, graph)`. `k` is the current vertex being colored.
4. Base case: If `k == V` (all vertices are colored), a valid coloring is found; print the `colors` array.
5. Recursive step: For the current vertex `k`, try assigning each color from 1 to `m`.
6. Before assigning a color `c` to vertex `k`, check if it's safe using `is_safe(k, c, colors, graph)`. This function checks if `c` is already used by any adjacent vertex of `k`.
7. If safe, assign the color (`colors[k] = c`), and recursively call `graph_coloring(k + 1, m, colors, graph)`.
8. If the recursive call doesn't lead to a solution, backtrack (implicitly done by trying the next color or returning).

Source Code

```
def graph_coloring(graph, m):
    """
    Solves the Graph Coloring problem using backtracking.

    Args:
        graph (list of lists): Adjacency list representation of the graph.
                               graph[i] contains a list of neighbors of
vertex i.
        m (int): The maximum number of colors available.

    Returns:
        list: A list representing the coloring (colors[i] = color of vertex
i),
              or None if no valid coloring is found with m colors.
    """
    V = len(graph)
    colors = [0] * V # colors[i] stores the color assigned to vertex i (0
means uncolored)

    def is_safe(v, c):
        """
        Checks if it's safe to assign color 'c' to vertex 'v'.
        It's safe if no adjacent vertex of 'v' has already been assigned
color 'c'.
        """
        for neighbor in graph[v]:
            if colors[neighbor] == c:
                return False
        return True

    def backtrack(v):
        """
        Recursive backtracking function to color vertices.
        'v' is the current vertex to be colored.
```

```
        """
        if v == V:
            # All vertices are colored, a valid coloring is found
            return True

        # Try assigning each color from 1 to m to vertex v
        for c in range(1, m + 1):
            if is_safe(v, c):
                colors[v] = c # Assign color
                if backtrack(v + 1): # Recur for the next vertex
                    return True
                colors[v] = 0 # Backtrack: unassign color (if current path
didn't lead to solution)
        return False # No color can be assigned to vertex v

    if backtrack(0):
        return colors
    else:
        return None

if __name__ == "__main__":
    # Example Graph 1 (a cycle graph C3, needs 3 colors)
    # 0 -- 1
    # |  /
    # | /
    # 2
    graph1 = [
        [1, 2], # Neighbors of 0
        [0, 2], # Neighbors of 1
        [0, 1]  # Neighbors of 2
    ]
    num_vertices1 = len(graph1)

    print("Graph 1 (C3):")
    print("  Edges:")
    for i, neighbors in enumerate(graph1):
        for neighbor in neighbors:
            if i < neighbor: # Print each edge only once
                print(f"    ({i}, {neighbor})")

    m1 = 2 # Try with 2 colors
    coloring1 = graph_coloring(graph1, m1)
    if coloring1:
        print(f"  Coloring with {m1} colors: {coloring1}")
    else:
        print(f"  No coloring possible with {m1} colors.")

    m1_optimal = 3 # Optimal for C3 is 3 colors
    coloring1_optimal = graph_coloring(graph1, m1_optimal)
    if coloring1_optimal:
        print(f"  Coloring with {m1_optimal} colors: {coloring1_optimal}")
    else:
        print(f"  No coloring possible with {m1_optimal} colors.")

    # Example Graph 2 (a path graph P4, needs 2 colors)
    # 0 -- 1 -- 2 -- 3
    graph2 = [
        [1],
        [0, 2],
        [1, 3],
        [2]
    ]
    num_vertices2 = len(graph2)

    print("\nGraph 2 (P4):")
    print("  Edges:")
```

```
    for i, neighbors in enumerate(graph2):
        for neighbor in neighbors:
            if i < neighbor: # Print each edge only once
                print(f"    ({i}, {neighbor})")

    m2 = 2 # Optimal for P4 is 2 colors
    coloring2 = graph_coloring(graph2, m2)
    if coloring2:
        print(f"  Coloring with {m2} colors: {coloring2}")
    else:
        print(f"  No coloring possible with {m2} colors.")
```

Input

```
Graph 1 (C3):
  Edges:
    (0, 1)
    (0, 2)
    (1, 2)
  Max colors: 2, then 3

Graph 2 (P4):
  Edges:
    (0, 1)
    (1, 2)
    (2, 3)
  Max colors: 2
```

Expected Output

```
Graph 1 (C3):
  Edges:
    (0, 1)
    (0, 2)
    (1, 2)
  No coloring possible with 2 colors.
  Coloring with 3 colors: [1, 2, 3]

Graph 2 (P4):
  Edges:
    (0, 1)
    (1, 2)
    (2, 3)
  Coloring with 2 colors: [1, 2, 1, 2]
```

(Note: The specific color assignments might vary, but the output should show a valid coloring if one exists with the given number of colors.)

Lab 14- Executing Branch and bound Algorithm for solving Hamilton Problem

Aim

To understand and apply the Branch and Bound algorithm to find a Hamiltonian Cycle in a given graph. A Hamiltonian Cycle is a cycle in an undirected graph that visits each vertex exactly once and returns to the starting vertex.

Procedure

1. Represent the graph using an adjacency matrix.
2. Define a cost matrix where `cost[i][j]` is the weight of the edge between `i` and `j`, or infinity if no edge exists.
3. The Branch and Bound approach for Hamiltonian Cycle often involves:
    - **State Space Tree:** Explore possible paths by building a state-space tree. Each node in the tree represents a partial path.
    - **Bounding Function:** Calculate a lower bound for the cost of completing the path from the current partial path. This helps prune branches that cannot lead to an optimal solution. For Hamiltonian Cycle, a common lower bound involves finding the minimum two edges incident to each unvisited vertex.
    - **Branching:** Extend the current partial path by adding an unvisited neighbor.
    - **Pruning:** If the current path's cost plus its lower bound exceeds the current best known solution, prune that branch.

Source Code

```
import math

# Define a large value for infinity
INF = float('inf')

def hamiltonian_cycle_branch_and_bound(graph):
    """
    Attempts to find a Hamiltonian Cycle using a simplified Branch and Bound
approach.
    This implementation focuses on finding *any* Hamiltonian cycle, not
necessarily the minimum cost one.
    For minimum cost, a more sophisticated bounding function would be needed
(e.g., TSP-like lower bounds).

    This is a conceptual implementation to illustrate Branch and Bound for
Hamiltonian Cycle.
    A full-fledged Branch and Bound for minimum cost Hamiltonian Cycle is
complex and often
    approximated or solved as a TSP variant. This version checks for
existence.
    """
    V = len(graph)

    # Stores the current path
    path = [-1] * V

    # To keep track of visited vertices
    visited = [False] * V

    # Store found cycles
    hamiltonian_cycles = []
```

```
    def find_hamiltonian_cycle_util(k):
        """
        Recursive utility function for Branch and Bound.
        k: current position in the path (0 to V-1)
        """
        # Base case: If all vertices are included in the path
        if k == V:
            # Check if the last vertex connects back to the first vertex
            if graph[path[k-1]][path[0]] != INF:
                hamiltonian_cycles.append(list(path) + [path[0]]) # Add the
cycle
                return True # Found one cycle, but we might want all
            else:
                return False

        # Try all possible vertices for the current position k
        for v in range(V):
            # Check if vertex v is a valid candidate for path[k]
            # 1. v has not been visited yet
            # 2. There is an edge from path[k-1] to v (if k > 0)
            if not visited[v] and (k == 0 or graph[path[k-1]][v] != INF):
                path[k] = v
                visited[v] = True

                # Branch: Recur for the next position
                if find_hamiltonian_cycle_util(k + 1):
                    # If we want to find only one cycle, return True here.
                    # For finding all, we let it continue.
                    pass

                # Backtrack: Unmark v and reset path[k]
                visited[v] = False
                path[k] = -1
        return False

    # Start the search from vertex 0
    path[0] = 0
    visited[0] = True
    find_hamiltonian_cycle_util(1) # Start from the second position

    return hamiltonian_cycles

def print_graph(graph):
    V = len(graph)
    print("Graph Adjacency Matrix:")
    for r in range(V):
        for c in range(V):
            if graph[r][c] == INF:
                print("%5s" % "INF", end=" ")
            else:
                print("%5d" % graph[r][c], end=" ")
        print()

if __name__ == "__main__":
    # Example Graph 1 (a complete graph K4, always has Hamiltonian cycles)
    # Edges with dummy weights for illustration
    graph1 = [
        [0, 10, 15, 20],
        [10, 0, 35, 25],
        [15, 35, 0, 30],
        [20, 25, 30, 0]
    ]

    print("Graph 1:")
    print_graph(graph1)
    cycles1 = hamiltonian_cycle_branch_and_bound(graph1)
```

```
    if cycles1:
        print("\nHamiltonian Cycles found for Graph 1:")
        for cycle in cycles1:
            print(" -> ".join(map(str, cycle)))
    else:
        print("\nNo Hamiltonian Cycle found for Graph 1.")

    # Example Graph 2 (a graph that does not have a Hamiltonian cycle)
    # A simple path graph 0-1-2-3 (no cycle)
    graph2 = [
        [0, 1, INF, INF],
        [1, 0, 1, INF],
        [INF, 1, 0, 1],
        [INF, INF, 1, 0]
    ]

    print("\nGraph 2:")
    print_graph(graph2)
    cycles2 = hamiltonian_cycle_branch_and_bound(graph2)

    if cycles2:
        print("\nHamiltonian Cycles found for Graph 2:")
        for cycle in cycles2:
            print(" -> ".join(map(str, cycle)))
    else:
        print("\nNo Hamiltonian Cycle found for Graph 2.")
```

Input

```
Graph 1 (complete graph K4):
      0    10    15    20
     10     0    35    25
     15    35     0    30
     20    25    30     0

Graph 2 (path graph P4):
      0     1   INF   INF
      1     0     1   INF
    INF     1     0     1
    INF   INF     1     0
```

Expected Output

```
Graph 1:
Graph Adjacency Matrix:
    0    10    15    20
   10     0    35    25
   15    35     0    30
   20    25    30     0

Hamiltonian Cycles found for Graph 1:
0 -> 1 -> 3 -> 2 -> 0
0 -> 2 -> 3 -> 1 -> 0
... (other cycles might be found depending on traversal order)

Graph 2:
Graph Adjacency Matrix:
    0     1   INF   INF
    1     0     1   INF
  INF     1     0     1
  INF   INF     1     0

No Hamiltonian Cycle found for Graph 2.
```

(Note: The order of cycles found for Graph 1 might vary. This implementation finds any Hamiltonian cycle, not necessarily the one with minimum cost. For minimum cost, the bounding function would need to be more complex, similar to TSP.)

Lab 15- Executing the TSP with Branch and Bound method

Aim

To implement the Traveling Salesperson Problem (TSP) using the Branch and Bound method to find the shortest possible route that visits each city exactly once and returns to the origin city.

Procedure

1. Represent the cities and distances as a cost matrix `cost[i][j]`.
2. The Branch and Bound algorithm for TSP typically involves:
    - **State Space Tree:** Explore paths by building a state-space tree. Each node represents a partial tour.
    - **Lower Bound Calculation:** For each node (partial tour), calculate a lower bound on the cost of completing the tour. A common lower bound is the sum of the current path cost and the sum of the minimum outgoing edges from unvisited vertices (after reducing the matrix).
    - **Matrix Reduction:** Reduce the cost matrix by subtracting the minimum element from each row and then each column. This helps in finding a tighter lower bound.
    - **Branching:** From the current state, branch by choosing the next city to visit.
    - **Pruning:** If the current path's cost plus its lower bound exceeds the current best known tour cost, prune that branch.
    - Maintain a `min_cost` variable to store the cost of the best tour found so far.

Source Code

```
import math

# Define a large value for infinity
INF = float('inf')

def tsp_branch_and_bound(graph):
    """
    Solves the Traveling Salesperson Problem (TSP) using the Branch and Bound
method.
    This implementation uses a basic lower bounding technique.

    Args:
        graph (list of lists): Adjacency matrix where graph[i][j] is the cost
                               of traveling from city i to city j.
                               Use INF for no direct path.

    Returns:
        tuple: A tuple containing:
               - float: The minimum cost of the Hamiltonian cycle.
               - list: The optimal tour (list of city indices).
    """
    V = len(graph)

    # Initialize minimum cost and optimal path
    min_cost = INF
    optimal_path = []

    # Current path being explored
    current_path = [-1] * (V + 1) # V cities + return to start
    current_path[0] = 0 # Start from city 0

    # Visited array to keep track of visited cities
    visited = [False] * V
    visited[0] = True # Mark starting city as visited
```

```
    # Recursive function for Branch and Bound
    def solve_tsp_util(k, current_weight, current_path_segment):
        nonlocal min_cost, optimal_path

        # Base case: All cities visited
        if k == V:
            # Check if there's a path back to the starting city (city 0)
            if graph[current_path_segment[-1]][current_path[0]] != INF:
                total_cost = current_weight + graph[current_path_segment[-
1]][current_path[0]]
                if total_cost < min_cost:
                    min_cost = total_cost
                    optimal_path = list(current_path_segment) +
[current_path[0]]
            return

        # Calculate a lower bound for pruning
        # A simple lower bound: current weight + sum of minimum outgoing
edges from unvisited nodes
        # This is a very basic bound; more sophisticated bounds involve
matrix reduction.
        lower_bound = current_weight
        for i in range(V):
            if not visited[i]:
                min_outgoing_edge = INF
                for j in range(V):
                    if not visited[j] or j == i: # Consider self-loop as 0
for min, or just unvisited
                        continue
                    min_outgoing_edge = min(min_outgoing_edge, graph[i][j])
                if min_outgoing_edge != INF:
                    lower_bound += min_outgoing_edge

        # If current path + lower bound is already greater than min_cost,
prune
        if lower_bound >= min_cost:
            return

        # Branch: Try visiting unvisited cities
        for city_idx in range(V):
            if not visited[city_idx] and graph[current_path_segment[-
1]][city_idx] != INF:
                visited[city_idx] = True
                current_path_segment.append(city_idx)

                solve_tsp_util(k + 1, current_weight +
graph[current_path_segment[-2]][city_idx], current_path_segment)

                # Backtrack: Unmark city and remove from current path
                current_path_segment.pop()
                visited[city_idx] = False

    solve_tsp_util(1, 0, [0]) # k=1 (next city to visit), current_weight=0,
current_path_segment=[0]

    return min_cost, optimal_path

def print_graph_matrix(graph):
    V = len(graph)
    print("Distance Matrix:")
    for r in range(V):
        for c in range(V):
            if graph[r][c] == INF:
                print("%5s" % "INF", end=" ")
            else:
```

```
                print("%5d" % graph[r][c], end=" ")
        print()

if __name__ == "__main__":
    # Example TSP Graph (4 cities)
    # Costs between cities (0 to 3)
    graph1 = [
        [0, 10, 15, 20],
        [10, 0, 35, 25],
        [15, 35, 0, 30],
        [20, 25, 30, 0]
    ]

    print("TSP Problem 1:")
    print_graph_matrix(graph1)

    min_cost1, optimal_path1 = tsp_branch_and_bound(graph1)

    if min_cost1 != INF:
        print(f"\nMinimum cost of TSP tour: {min_cost1}")
        print(f"Optimal tour: {' -> '.join(map(str, optimal_path1))}")
    else:
        print("\nNo TSP tour found (graph might be disconnected or no cycle
exists).")

    # Example TSP Graph 2 (5 cities, a slightly more complex one)
    graph2 = [
        [0, 20, 42, 35, 25],
        [20, 0, 30, 34, 10],
        [42, 30, 0, 12, 18],
        [35, 34, 12, 0, 15],
        [25, 10, 18, 15, 0]
    ]

    print("\nTSP Problem 2:")
    print_graph_matrix(graph2)

    min_cost2, optimal_path2 = tsp_branch_and_bound(graph2)

    if min_cost2 != INF:
        print(f"\nMinimum cost of TSP tour: {min_cost2}")
        print(f"Optimal tour: {' -> '.join(map(str, optimal_path2))}")
    else:
        print("\nNo TSP tour found (graph might be disconnected or no cycle
exists).")
```

Input

```
TSP Problem 1:
Distance Matrix:
    0    10    15    20
   10     0    35    25
   15    35     0    30
   20    25    30     0

TSP Problem 2:
Distance Matrix:
    0    20    42    35    25
   20     0    30    34    10
   42    30     0    12    18
   35    34    12     0    15
   25    10    18    15     0
```

Expected Output

```
TSP Problem 1:
Distance Matrix:
    0    10    15    20
   10     0    35    25
   15    35     0    30
   20    25    30     0

Minimum cost of TSP tour: 80
Optimal tour: 0 -> 1 -> 3 -> 2 -> 0

TSP Problem 2:
Distance Matrix:
    0    20    42    35    25
   20     0    30    34    10
   42    30     0    12    18
   35    34    12     0    15
   25    10    18    15     0

Minimum cost of TSP tour: 90
Optimal tour: 0 -> 1 -> 4 -> 3 -> 2 -> 0
```