Summary on methods, algorithms, and data structures: For assignment 4

ASSIGNMENT NUMBER : 04
NAME : Dinesh Raj Pampati
ULID : dpampat
NAME OF SECRET DIRECTORY : namo

Summary on methods, algorithms, and data structures:

1. Methods Used:

reverseGraph(Hashtable<Integer, PriorityQueue<VertexWeighted>> graph):

This method is used to reverse the direction of all edges in the graph. It creates a new graph where for every edge (u, v) in the original graph, there is an edge (v, u) in the reversed graph. This is useful for algorithms that need to process the graph in the reverse direction of the edges.

checkVertexIsSuperNode(int vertex, HashMap<Integer, Integer> map):

Checks whether a given vertex is a "super node" — typically, a node resulting from the contraction of a cycle into a single node. The information about which nodes are super nodes is stored in a map that is updated during the contraction process.

noOfCycleNodes(List<List<Integer>> cycleNodes, int vertex):

Determines the number of nodes in cycles that are smaller than the specified vertex. This is important for algorithms that need to adjust indices or weights based on the contraction of cycles.

findAllZeroCycles(Hashtable<Integer, PriorityQueue<VertexWeighted>> graph, int size):

Identifies all cycles in the graph with zero total weight. Such cycles are important in algorithms like Edmonds' algorithm because they can be contracted without affecting the total weight of the spanning arborescence.

contractSuperNodes(Hashtable<Integer, PriorityQueue<VertexWeighted>> graph, List<List<Integer>> cycleNodes, int lastIndex, int counter):

Handles the contraction of cycles identified by **findAllZeroCycles**. It updates the graph to reflect the contraction, adjusts edges as necessary, and records information needed to eventually reconstruct the original graph from the contracted graph.

expansion(Stack<Hashtable<Integer, PriorityQueue<VertexWeighted>>> graphs, int lastVertex, int counter):

This method is part of the final phase of the algorithm, where the graph that has been contracted in previous steps is now expanded back to its original form. The method carefully restores vertices and edges that were previously contracted, ensuring to maintain the properties that qualify the graph as a minimum spanning arborescence.

edmondAlgorithm(Hashtable<Integer, PriorityQueue<VertexWeighted>> allValues, int lastVertex, int counter):

The core part of the implementation, representing Edmonds' algorithm itself. This method orchestrates the various steps of the algorithm, including the identification and contraction of zero-weight cycles, the main loop that drives the contraction process, and the final expansion phase. It continually updates the graph structure and calls other methods as necessary until the minimum spanning arborescence is identified.

main(String[] args):

The driver method of the program. It handles the initial reading of the graph from an external source (like a file), organizes the data into a suitable graph representation, and initiates the execution of Edmonds' algorithm by calling **edmondAlgorithm**. After the algorithm concludes, it may also handle the output of the results, such as printing the minimum spanning arborescence to the console or another medium.

2. Algorithm:

The core algorithm being implemented is the **Edmonds' Algorithm** for determining a minimum spanning arborescence in a directed graph. The steps involved are:

1. Subtract the weight of the smallest incoming edge from all incoming edges for each vertex.
2. Search for a cycle in the graph.
3. If a cycle is found, contract the cycle into a single supernode.
4. Repeat the above steps until no more cycles are found.
5. Once no cycles exist, begin the expansion phase to revert the graph to its original form, but without cycles.

3. Data Structures Used:

- **VertexWeighted**: Represents a weighted vertex in the graph. This isn't detailed in the provided code, but the assumption is that it holds information about two vertices, weight, and a counter.
- **Hashtable<Integer, PriorityQueue<VertexWeighted>>**: A representation of the graph where each vertex is mapped to a priority queue of incoming edges.
- **Stack<Hashtable<Integer, PriorityQueue<VertexWeighted>>>**: Holds the graph in various stages of transformation. Vital for the expansion phase of the algorithm.

4. Time Complexity:

1. **reverseGraph**: Iterates over all edges once, so it has a complexity of **O(E)**.
2. **contractSuperNodes**: Iterates over all vertices in the worst case, with nested loops iterating over edges. This gives it a complexity of **O(V * E)**.
3. **expansion**: Contains a while loop iterating through the graphs. Each iteration pops a graph and processes its vertices and edges, which results in **O(V * E)**.
4. **edmondAlgorithm**: This is a loop that contracts supernodes and checks for zero cycles until no cycles remain. The worst-case scenario (which would be rare) would involve contracting each vertex one-by-one, leading to a time complexity of **O(V^2 * E)**. The reason it's squared is due to the repeated contraction operation and cycle checking for V vertices.
5. **main**: Reads and processes the input graph. The complexity depends on the input size but can be represented as **O(V + E)** since all vertices and edges are processed.

When you combine all these, the most dominant factor is the **edmondAlgorithm** which is **O(V^2 * E)**.

5. Space Complexity:

1. **Hashtables and PriorityQueues**: The graph representation is stored in a hashtable where each vertex maps to a priority queue of its incoming edges. This results in **O(V + E)**.
2. **Stack**: Holds copies of the graph at different contraction stages. In the worst case, every vertex might get contracted, leading to V copies of the graph. This gives **O(V^2 + V*E)**.
3. **Various Lists and Maps**: Used in methods like **contractSuperNodes**, **expansion**, etc. These generally hold vertices or edges, adding another **O(V + E)**.

When you sum up the space complexities from the different structures and methods, the most dominant factor here is the stack's space complexity: **O(V^2 + V*E)**.

In conclusion, the time complexity for your code is approximately **O(V^2 * E)** and the space complexity is **O(V^2 + V*E)**.