Report

**Summary:**

**Edmonds's algorithm:**

Edmonds's algorithm is a tool for finding an arborescence in a directed weighted graph. An arborescence is essentially a directed tree-like subgraph rooted at a specific node, and the primary objective is to create one that minimizes the total weight of its edges.

The algorithm's approach involves a series of well-defined steps. It begins by attempting to find a maximum branching within the graph. This is done by starting from the root node and, in a somewhat greedy manner, adding the incoming edge with the least weight for each node. Importantly, it carefully avoids forming cycles in the process. If, at any point, the algorithm detects a cycle, it takes action (forms a super node with those vertices ) by marking the edges that make up that cycle. Subsequently, it proceeds to adjust the weights of these marked edges. This adjustment includes reducing the weight of the marked edges while increasing the weight of unmarked edges by the same amount. This effectively removes the cycle from consideration. The algorithm repeats these steps until it successfully constructs the desired arborescence, which represents an optimal tree-like subgraph in the context of the original directed graph.

**Data structures used:**

1. HashMap: Several instances of “HashMap” are used to represent different aspects of the graph:

**- “HashMap<String, List<List<String>>> graph”:** This “HashMap” represents the directed graph, where keys are vertices represented as strings, and values are lists of edges. Each edge is represented as a list of strings, containing the neighboring vertex, edge weight, and other information.

**- “HashMap<String, List<String>> FStar”:** This “HashMap” represents a part of the graph and is used to store certain information derived from the original graph.

- **“HashMap<String, List<String>> FStarReversed”:** Similar to “FStar”, this “HashMap” stores information related to the reversed graph.

2. **Set (HashSet and LinkedHashSet):**

- “Set<String> visited”: A “HashSet” is used to keep track of visited nodes during a depth-first search (DFS) traversal.

- **“Set<List<String>> cycles”:** A “LinkedHashSet” is used to store cycles found in the graph, ensuring uniqueness and preserving the order of insertion.

3. **List (ArrayList and PriorityQueue):**

- **“List<String> path”**: An “ArrayList” is used to keep track of the path during DFS traversal. It's used to detect cycles in the graph.

- **“List<List<String>> edges”:** Lists of edges are used to represent the graph, and a “PriorityQueue” (“priorityQueue”) is used to store these edges based on their weights, allowing for efficient retrieval of the edge with the minimum weight.

- **“List<Object> values”:** A list is used to return multiple values from a function. In this case, it's used to return “graph”, “FStar”, and “FStarReversed”.

- **“List<List<String>> finalGraph”:** This list is used to store the final representation of the graph after processing. It contains lists representing edges with their corresponding weights.

4. **Other Data Structures:**

- **“HashMap<List<String>, List<String>> edgeMapping”:** This “HashMap” is used to map edges in the contracted graph to their corresponding edges in the original graph.

- **“HashMap<String, String> vertexMap”:** This “HashMap” is used to map vertices in the contracted graph to their corresponding super-nodes in the original graph.

**6. Other Data Structures and Classes:**

- **“PriorityQueue”:** A priority queue is used to keep track of edges and retrieve the one with the minimum weight during certain operations.

- **“Set<List<String>>“:** Sets are used to represent sets of cycles and unique paths in the graph.

**Pseudo Code:**

1. Class: Edmond

2. Define class GraphUtils inside Edmond:

- a. **findUniqueCycles(fStar):**

- Initialize an empty set 'visited' to keep track of visited nodes.

- Initialize an empty set 'cycles' to store unique cycles.

- For each key in 'fStar':

- Create an empty list 'path' to represent the current path.

- Call the dfs function with the current key, path, visited, fStar, and cycles.

- b**. dfs(node, path, visited, fStar, cycles):**

- If 'node' is already in 'path':

- Extract the cycle from 'path' and add it to 'cycles'.

- If 'node' is in 'visited', return.

- Add 'node' to 'visited'.

- For each neighbor node in fStar[node]:

- Create a new list 'newPath' by copying 'path' and adding 'node'.

- Recursively call dfs with 'neighborNode', 'newPath', visited, fStar, and cycles.

- Remove 'node' from 'visited'.

- c. **isArborescence(fStar, fStarReversed):**

- Check for cycles in the graph by calling findUniqueCycles.

- Check if any vertex has more than one in-degree. If so, return the cycles.

- Check if the total number of edges is greater than or equal to the number of vertices.

- Return null if there are no cycles and the conditions for arborescence are met.

3. Define class-level variables and methods:

- a. **readGraphFile(fileName):**

- Initialize an empty list 'graphs' to store multiple graphs.

- Initialize 'graph' as null to represent the current graph being read.

- Open and read the file 'fileName' line by line:

- Create a BufferedReader to read the file 'fileName'.

- Initialize 'br' as the BufferedReader.

- Parse the lines to construct graphs and edges:

- While there are lines left in the file:

- Read the next line using the BufferedReader 'br'.

- Trim the line to remove leading and trailing whitespace using line.trim().

- Replace all spaces in the trimmed line to remove any spaces using line.replace(" ", "").

- Check the content of the line:

- If the line starts with , it represents a new graph.

- Parse the number of vertices from the line and create a new 'graph' HashMap with vertex placeholders.

- If the line starts with --, it indicates the end of the current graph.

- If 'graph' is not null, append it to the 'graphs' list.

- 'graph' is stored in a reversed adjacency list format.

- If the line starts with (u, it is a special case and can be skipped.

- If the line starts with (, it represents a new edge.

- Extract the vertices, weight, and other information from the line.

- Create a new edge representation and add it to the current 'graph'.

- Update the vertex-to-edge mappings in the 'graph'.

- After reading the entire file:

- 'graphs' contains all the graphs constructed from the file.

- Close the BufferedReader 'br' to free up system resources.

- Return 'graphs', which is a list of graphs.

- b. **getMinEdge(listOfLists):**

- Create a priority queue 'priorityQueue' with a custom comparator based on edge weights.

- Add all edges from 'listOfLists' to 'priorityQueue'.

- Retrieve and return the edge with the minimum weight from 'priorityQueue'.

- c. **appendReducedWeight(graph):**

- Iterate through the graph's edges and calculate the minimum edge weight for each node.

- Update the edge weights by subtracting the minimum weight.

- Return the updated graph.

- d. **getFStar(graph):**

- Initialize two empty HashMaps 'FStar' and 'FStarReversed':

- 'FStar': Represents the forward star graph since it's easier to find cycles in dfs with this representation. It will store outgoing edges from each vertex as key-value pairs.

- 'FStarReversed': Represents the reversed star graph. It will store incoming edges to each vertex and their weights.

- For each vertex in the 'graph':

- Initialize 'FStar' and 'FStarReversed' entries for the current vertex.

- Iterate through the vertices and their associated edges in the 'graph':

- For each vertex, find the outgoing edges to other vertices.

- For each edge, determine if the weight is equal to zero.

- If the weight is zero, it implies that this edge is part of the forward star ('FStar').

- Update 'FStar':

- If the target vertex of the edge is not already a key in 'FStar', create an entry with an empty list.

- Append the source vertex to the list of the target vertex's outgoing edges.

- Update 'FStarReversed':

- If the source vertex is not already a key in 'FStarReversed', create an entry with an empty list.

- Append the target vertex, weight, and the edge's weight to the source vertex's incoming edges.

- Add the label "fStar" to the edge to indicate it is part of the forward star.

- After iterating through all vertices and edges:

- 'FStar' contains information about outgoing edges.

- 'FStarReversed' contains information about incoming edges with their weights.

- Return a list containing 'graph', 'FStar', and 'FStarReversed' for further processing.

- e. getCycleNode(cycles, node):

- cycles is the list of all the cycles in fStar

- Check if 'node' belongs to any cycle in 'cycles'.

- If yes, return the super node; otherwise, return 'node'.

- f. **contractCycles(graph, cycles, edgeMapping):**

- Create a new graph 'newGraph' after contracting the cycles.

- Create 'edgeMapping' to store mappings between old and new edges.

- Update edge weights based on the minimum weight within cycles.

- Return 'newGraph' and 'edgeMapping'.

- g. **mainRecursion(graph, verbose):**

- This is the main recursive function for finding the minimum arborescence of a graph.

- It takes two parameters: 'graph,' which represents the current state of the graph, and 'verbose,' a boolean indicating whether to display debugging information.

- If 'verbose' is true, print debugging information to track the progress of the algorithm.

- Calculate the initial 'graph' by appending reduced weights to the edges using the 'appendReducedWeight' function. This step prepares the graph for further processing by reducing its weights.

- Call the 'getFStar' function to extract the forward star graph ('FStar') and its reverse ('FStarReversed'). 'FStar' is used for exploring the graph, and 'FStarReversed' is used for cycle detection.

- Check for the presence of cycles in the forward star graph ('FStar') and its reverse ('FStarReversed') by calling the 'isArborescence' function. The 'isArborescence' function checks for cycles, and if cycles are present, it returns the cycle information.

- If cycles are detected:

- Create an 'edgeMapping' to map edges within cycles to their original edges.

- Create a 'vertexMap' to map vertices to their corresponding super nodes in cycles.

- Contract the cycles in the graph using the 'contractCycles' function. This step involves merging nodes within cycles and adjusting edge weights.

- Recursively call 'mainRecursion' with the contracted graph. This step continues the exploration of the contracted graph to handle nested cycles.

- If 'verbose' is true, print debugging information regarding expansion, including the actual, contracted, and returned graphs.

- Create a 'cycleMap' to store information about the vertices of cycles and edges within cycles for the expanded graph.

- Build a 'newFinalGraph' that represents the final minimum arborescence.

- Find the old edge from edgeMapping and update all the edges in the newFinalGraph.

- Add the edges within each cycle and exclude the edge that goes to the starting vertex of the cycle.

- Return 'newFinalGraph' as the result.

- If no cycles are detected:

- Handle the case where the graph is already an arborescence.

- Create a temporary graph 'tmpGraph' with edges that have single in-degrees.

- Return 'tmpGraph' as the result.

- The function concludes with the final result, which represents the minimum arborescence of the input graph.

- h. cleanFinalGraph(finalGraph, nVertices, count, runtime):

- Clean and print the final arborescence, including the edges and their types.

- Calculate the total weight of the arborescence.

- Print the result with graph information.

4. **Main Function:**

- Read the input file and get a list of graphs.

- For each graph, perform the main recursion to find the minimum arborescence.

- Clean and print the final arborescence for each graph, including total weight and runtime.

**Time and space complexity analysis:**

**Time Complexity analysis:**

The overall time complexity of the code for finding the Minimum Cost Arborescence using Edmonds' algorithm can be determined by summing up the time complexities of the individual components. Based on the provided information, here's the breakdown:

1. Finding Minimum Cost Arborescence:

- Time Complexity: O(V \* (E + V log V)), where V is the number of vertices and E is the number of edges in the graph. This is the time complexity of the main algorithm.

2. Finding Cycles:

- Time Complexity: O(V + E) for DFS traversal. This step is part of the main algorithm and contributes to the overall time complexity.

3. Creating Compressed Graph:

- Time Complexity: O(E) for compressing each cycle. This operation occurs during the algorithm's execution.

4. Expanding Compressed Graph:

- Time Complexity: O(E) for expanding the compressed graph. This step is also part of the algorithm's execution.

So, the overall time complexity is the sum of these time complexities:

Overall Time Complexity = O(V \* (E + V log V)) + O(V + E) + O(E) + O(E)

In big O notation, when we sum these terms, we focus on the dominant factors:

Overall Time Complexity = O(V \* (E + V log V)) + O(V + E + E)

When simplifying, the most significant term is V \* E, and log V is typically smaller than E:

Overall Time Complexity ≈ O(V \* E)

- The overall time complexity of the algorithm depends on the number of iterations of the mainRecursion function. In the worst case, it may be O(V \* E).

**2. Space Complexity:**

The space complexity of the provided code can be analyzed as follows:

1. Graph Storage:

- The original graph is represented using an adjacency list data structure. It uses a HashMap<String, List<List<String>>> to store the graph, where each vertex (represented as a string) is associated with a list of edges. The space complexity for storing the original graph is O(V + E), where V is the number of vertices, and E is the number of edges. Each vertex is stored once, and each edge is stored once.

3. Recursive Calls:

- The algorithm employs a recursive approach to contract cycles in the graph. Recursive function calls add to the call stack. The maximum depth of the call stack is related to the number of iterations required to contract all cycles in the graph. In the worst case, where the algorithm performs a full contraction for each vertex and edge, the space complexity of the call stack is O(V + E).

Overall, the space complexity of the provided code is dominated by the space required to store the original graph, intermediate data structures, and the call stack during recursion. In the worst case, the space complexity is O(V + E), where V is the number of vertices and E is the number of edges. Please note that this analysis assumes that the input graph is the primary factor affecting space complexity, and other factors may contribute to space usage during execution.