Summary on methods, algorithms, and data structures: For assignment 8

ASSIGNMENT NUMBER : 08
NAME : Dinesh Raj Pampati
ULID : dpampat
NAME OF SECRET DIRECTORY : namo

1. Methods Used

Bipartite Matching Program

1. **main(String[] args)**:
    - **Purpose**: Serves as the entry point of the program. Reads data from a file and initiates the bipartite matching computation.
    - **Process**: Opens the specified file and reads data to construct a bipartite graph represented as an adjacency list. After constructing the graph and verifying its bipartite nature, it calls **constructFlowNetwork** to build the flow network and **fordFulkerson** to find the maximum matching.
2. **constructFlowNetwork(ArrayList<ArrayList<Edges>>, int, List<Integer>[])**:
    - **Purpose**: Constructs the flow network from the bipartite graph, including a source and sink node.
    - **Process**: Initializes a new flow network, adds edges from the bipartite graph to the flow network (without reverse edges), and connects the source to one set and the sink to the other set of the bipartite graph.
3. **findBipartiteParts(ArrayList<ArrayList<Edges>>, int)**:
    - **Purpose**: Determines if the given graph is bipartite and identifies the two disjoint sets.
    - **Process**: Uses BFS to check bipartiteness. If the graph is bipartite, it returns two lists representing the two disjoint sets; otherwise, returns null.
4. **bfsCheckBipartite(ArrayList<ArrayList<Edges>>, int, int[], List<Integer>, List<Integer>)**:
    - **Purpose**: Performs BFS to check if a graph is bipartite and assigns vertices to two parts.
    - **Process**: Standard BFS approach, coloring vertices to identify the two different sets. If a conflict in coloring is found, the graph is not bipartite.
5. **fordFulkerson(ArrayList<ArrayList<Edges>>, int, int)**:
    - **Purpose**: Implements the Ford-Fulkerson algorithm to calculate the maximum flow (and thus maximum matching) in a flow network.
    - **Process**: Repeatedly finds augmenting paths using BFS and updates flows along these paths. The process repeats until no augmenting path can be found.
6. **bfs(ArrayList<ArrayList<Edges>>, int, int, int[])**:
    - **Purpose**: Implements BFS to find an augmenting path from the source to the sink in the flow network.
    - **Process**: Traverses the graph from the source, tracking visited nodes and their parents to reconstruct the path. Returns true if a path is found, false otherwise.

2. Data Structures Used

- **ArrayList<ArrayList<Edges>> (Graph)**:
    - An adjacency list representation of the graph.
    - Each list within the main list corresponds to a vertex and contains a list of **Edges** objects representing the edges emanating from that vertex.
- **Edges Class**:
    - Custom class to represent an edge in the graph.
    - Stores properties like source, destination, capacity, and flow of each edge.
- **List<Integer>[] (for Bipartite Parts)**:
    - Used to store the two disjoint sets of vertices in a bipartite graph.
- **int[] (colors, parent Array)**:
    - **colors**: Used in checking bipartiteness to mark vertices of different sets.
    - **parent**: Used in BFS to keep track of the parent nodes for reconstructing paths.
- **Queue<Integer> (in BFS Methods)**:
    - A queue used for BFS traversal in both bipartite checking and finding augmenting paths.
- **File and Scanner**:
    - Used for reading input data from a file.

Algorithm Explanation

1. **Bipartite Check (findBipartiteParts & bfsCheckBipartite)**:
    - The code first checks if the given graph is bipartite. This is crucial because the maximum matching problem is being solved in the context of bipartite graphs.
    - **Process**:
        - It uses a breadth-first search (BFS) approach.
        - Each vertex is either uncolored (initial state), colored as part of set 1, or colored as part of set 2.
        - The algorithm traverses the graph, trying to assign alternate colors to adjacent vertices.
        - If it encounters a vertex that should be colored but has already been colored with a different color, the graph is not bipartite.
2. **Construct Flow Network (constructFlowNetwork)**:

- Once the graph is confirmed as bipartite, it is converted into a flow network.
- **Process**:
    - Two additional nodes are added: a source (s) and a sink (t).
    - Directed edges are added from the source to all vertices in one part of the bipartite graph and from all vertices in the other part to the sink. Each of these edges has a capacity of 1.
    - The existing edges between the two parts of the bipartite graph are directed from the first part to the second part.

3. **Ford-Fulkerson Algorithm (fordFulkerson & bfs)**:
    - This algorithm is used to find the maximum flow in the constructed flow network, which corresponds to the maximum matching in the bipartite graph.
    - **Process**:
        - It repeatedly searches for augmenting paths: paths from the source to the sink where the capacity exceeds the flow.
        - BFS is used to find these paths efficiently.
        - Once an augmenting path is found, the flow along this path is increased by the minimum residual capacity on the path.
        - The algorithm continues until no more augmenting paths can be found.

Time Complexity

1. **Bipartite Check**:
    - The BFS in a graph has a time complexity of O(V + E), where V is the number of vertices and E is the number of edges. Since every vertex and edge is visited at most once, this holds for the bipartite check.
2. **Construct Flow Network**:
    - The construction of the flow network is done in linear time relative to the number of vertices and edges in the original graph. Hence, the complexity is O(V + E).
3. **Ford-Fulkerson Algorithm**:
    - The worst-case time complexity is O(E * maxFlow), where E is the number of edges in the flow network and maxFlow is the maximum flow found.
    - The BFS used in finding augmenting paths has a complexity of O(V + E), and it is repeated potentially maxFlow times.

Space Complexity

1. **Graph Storage**:

- The space complexity for storing the graph using an adjacency list is O(V + E).
- Each vertex and edge contributes to the space used.

2. **Auxiliary Arrays**:
    - The **parent** array in BFS, and the **colors** array in the bipartite check, each require O(V) space.
    - The space used for these arrays is linear in the number of vertices.
3. **Flow Network**:
    - The flow network is an extended version of the original graph with two extra nodes and additional edges. Its space complexity is also O(V + E).