SE2205: Algorithms and Data Structures for Object-Oriented Design
Lab Assignment 3
Assigned: Mar 20, 2019; Due: April 8, 2019 @ 10:00 a.m.

If you are working in a group of two, then indicate the associated student IDs and numbers in the `Assignment3.java` file as a comment in the header.

1 Objectives

In this assignment, you will implement an algorithm that computes the maximum flow possible in a flow network digraph. A flow network can represent a network of pipelines having different flow rates. Please refer to the diagram in Figure 1 for an example of a flow network. In a flow network, there are two special nodes called the **source** and **destination** nodes (marked as `S` and `D` in Figure 1). Material such as oil fluids can propagate through the network starting from the source `S` and flow into the destination node `D`. Intermediate nodes (i.e. 1 to 7 in the example) can represent pumping stations.

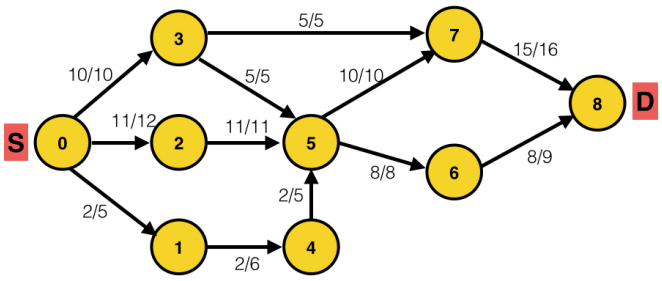

Figure 1: A sample flow network

1.1 Constraints

Since a flow network is a special kind of network subject to physical limitations, flow and capacity constraints do apply. Every edge $e_{i,j}$ or pipeline connected by vertices v_i and v_j has a maximum capacity of $c_{i,j}$. Hence, due to this physical **capacity limitation**, any substance flowing through pipeline $e_{i,j}$ must have a flow of $f_{i,j}$ that is between 0 and $c_{i,j}$ (i.e. $0 \leq f_{i,j} \leq c_{i,j}$). Referring back to Figure 1, $e_{0,2}$ has a flow of $f_{0,2} = 11$ and a capacity of 12. Since $f_{0,2} \leq c_{0,2}$, this is a valid flow.

Next, suppose three pipelines or edges meet at a single node and two pipelines leave that node (consider $e_{2,5}$, $e_{3,5}$, $e_{4,5}$, $e_{5,6}$, $e_{5,7}$). Any substance flowing into a node must be equal to the substance flowing out of the node and this is called **flow conservation**. In Figure 1, $f_{2,5} + f_{3,5} + f_{4,5} = 18$ and $f_{5,6} + f_{5,7} = 18$ and more generally this can be expressed as $\sum_{i \in V} f_{i,j} = \sum_{i \in V} f_{j,i}$ for all v_j with the exception of S and D as these are source and sink nodes (generate or consume substance).

Finally, there is the notion of **back flow**. For instance, if there exists positive flow $f_{i,j}$ on edge $e_{i,j}$, then it is possible for substance to flow in the opposite direction $e_{j,i}$ where $0 \leq f_{j,i} \leq f_{i,j}$. If there is no substance in the pipeline then nothing can propagate in the opposite direction. However, if there is some material flowing through the pipe in one direction, then it is entirely possible for this material to flow in the opposite direction.

1.2 General Algorithm for Finding Maximum Flow in the Network

If you attempt to increase the flows on edges of the flow network provided in Figure 1, you will notice that this is not possible. Suppose you are increasing flow on $e_{0,2}$ by 1 to 12. In order to satisfy flow conservation, the

flow on $e_{2,5}$ must also increase by 1. This is impossible as this will violate the maximum capacity restriction on $e_{2,5}$ (i.e. maximum flow possible through this edge is 11 not 12). Hence, if you attempt to increase the flow like this on every edge, you will realize that this is not feasible. For this reason, all flows listed in Figure 1 are the maximum possible flows throughout the flow network. The amount of substance leaving from node 0 which is the source node is 23 and similarly the amount of substance entering the destination node 8 is also 23. For this reason, the maximum flow throughout this flow network is 23! In this assignment, you will implement a well known algorithm to compute the maximum flow throughout a flow network. In general, this algorithm is composed of the following steps:

- Find a path p from `S` to `D` that consists of no edges with flow that is equal to the full capacity of the edge (i.e. $f_{i,j} \neq c_{i,j}$)
- Find the maximum flow that can be added to the path so that none of the edges in the path violate their flow capacity constraints
- Add this flow to every edge in the path p
- Repeat the above until no more paths exist in the graph from `S` to `D` to which more flow can be added

This assignment is divided into two parts. In the first part, you will implement a **breadth-first** search algorithm to discover a path from S to D to which more flow can be added to all edges in this path. These paths will be referred to as **augmenting** paths. In the second part, you will implement code that calls on this path discovering algorithm repeatedly to add flow to the flow network until no more flow can be added.

You can assume that you are provided with the definitions to the `Graph` ADT and `Edge` class (you can assume that all the functions provided in the "Graphs" lecture slides are available). An `Edge` is composed of the following members:

- `public int flow;`
- `public int flowCap;`

`Edge` represents an edge and stores the current flow in that edge and the maximum capacity of that edge in the `flow` and `flowCap` members respectively.

You will implement the function `public int breadthFirstPathSearch(Graph FN, int s, int d)` function. This function uses breadth-first search to find an augmenting path connecting vertices `s` and `d` to which flow can be added without infringing on capacity constraints of the network. Since this is a breadth-first search algorithm, you will have to use a queue in your implementation. You can assume that all the interface functions associated with the `LinkedListQueue` are provided.

In order to understand the general breadth-first path-finding algorithm, consider Figures 2, 3, 4. Figure 2 illustrates the initial flow network which has no flow yet on any edges. The path-finding algorithm should find a path connecting `s` and `d` which are source and destination nodes that consists of no edges with flow being equal to the maximum edge capacity. The breadth-first search algorithm attempts to find such a path that has the shortest distance (i.e. smallest number of edges) from `s` to `d`. A breadth-first search algorithm will start from `s` and examine all children/adjacent nodes of `s` (by enqueueing these nodes into a queue) one by one. For the network in Figure 2, node 1 will be examined first and checked to see if it has been visited or not and whether the current flow in the edge (v_0, v_1) has not reached maximum capacity. If these conditions are met, node 1 is set to be visited and in the index corresponding to node 1 in the `parent` array is set to node 0 as node 0 is the parent of node 1 forming the edge (v_0, v_1) and then node 1 is enqueued into the queue. All the children of node 0 are examined in a similar manner and enqueued into the queue if the checking conditions are met.

After examining all adjacent nodes of node 0, `parent={-1, 0, 0, 0, -1, -1, -1, -1, -1}`. The next node in the queue will be node 1 and its child node 4 is examined. As 4 is unvisited and has no flow, the parent of node 4 is set to node 1, marked as visited and enqueued into the queue. Proceeding in this manner, parent of 5 is set to 2 and node 5 is enqueued into the queue. The next node examined is 3 and its children are 5 and 7. Since 5 has already been visited, its parent will not change and it will not be enqueued into the queue. However, since 7 is not visited, its parent is set to 3, marked as visited and enqueued into the queue. This will proceed until all nodes in the graph are examined (i.e. the queue is empty). At this point, the parent array for the flow network in Figure 2 will be `parent={-1, 0, 0, 0,`

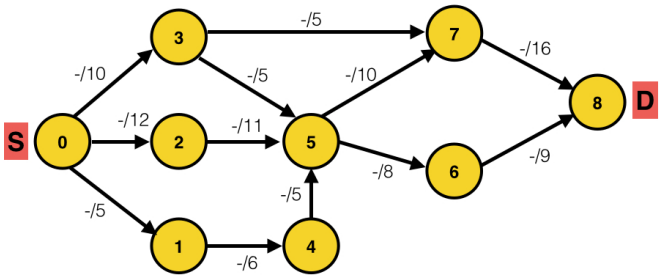

Figure 2: Initial network with no flow

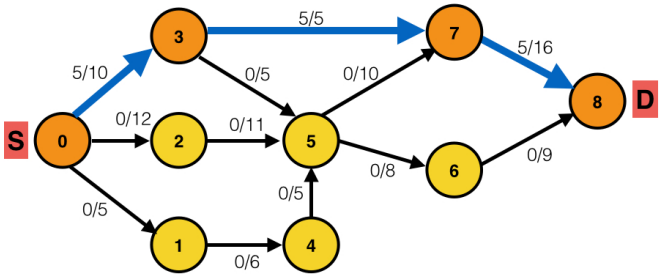

Figure 3: First Augmenting Path from S-D

`1, 2, 5, 3, 7`}. Starting from the last index corresponding to node 8 which is the destination node, it is clear that the parent of 8 is 7, the parent of 7 is 3, the parent of 3 is 0. Hence, the augmenting path from `s` to `d` in which flow can be increased is 0-3-7-8 and this is the shortest available path from node 0 to 8 as illustrated in Figure 3.

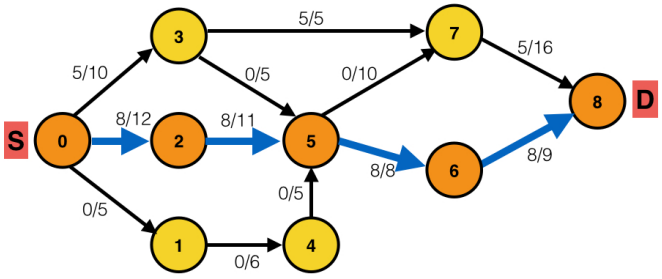

Figure 4: Second Augmenting Path from S-D

Suppose that a flow of 5 has been added to the path 0-3-7-8. If the breadth-first search algorithm is called again to operate on this flow network, the parent of 7 will not be set to be 3 anymore as the flow on the edge (v_3, v_7) is 5 which is the maximum capacity of that edge (no more flow can be added without infringing on the capacity constraint of that edge). The parent of 7 in this case will be set to 5. The augmented path resulting from this graph will then be 0-2-5-6-8 as illustrated in Figure 4. Hence, the path-finding algorithm will successfully avoid the edge that is full. At the conclusion of the path search, the function will return 1 if the destination node has been visited and 0 otherwise. If the return value is 0, then an augmenting path

does not exist that connects `s` to `d`. This breadth-first searching algorithm is called the **Edmonds-Karp** algorithm. The implementation of this algorithm is summarized in the following:

- First set all elements in the `visitedNodes` array to 0 (to mark all nodes as unvisited)
- Initialize a queue and enqueue the starting node
- Begin a while-loop in which:
  - The queue is dequeued (say the dequeued node is v_i)
  - For every node v_j that is adjacent to the dequeued node v_i, check if this adjacent node is unvisited and whether the current flow in the edge (v_i, v_j) is such that more than 0 flow added to the edge (i.e. $c_{i,j} - f_{i,j} > 0$)
  - If these conditions are met, then set the parent of this adjacent node v_j to be the dequeued node v_i and enqueue the adjacent node into the queue
  - Repeat the above until the queue is empty
  - Before the function terminates return 1 if `d` is visited and 0 otherwise

Next, you will implement the function `public void maximizeFlowNetwork(Graph fN, int s, int t)`. This is the function that computes the maximum flow in the flow network `fN`. Following are the three main components of this function:

- The `breadthFirstPathSearch` function is called to find an augmented path
- If such a path exists then the return value of this function is 1.(i.e. if node `d` is visited by the path searching algorithm). In this case, the maximum possible flow that can be added on all edges that form the augmenting path is computed (this flow must heed the capacity constraints of the edge)
- The flow on all edges forming the augmenting path is incremented to the value computed in the previous step. For every edge in which flow is added, ensure that you account for the back flow in the opposite direction
- All three steps above are repeated until no more flow can be added to the flow network

For instance, suppose that the augmented path computed by the path searching algorithm is 0-2-5-7-8. From Figure 3, it is clear that the maximum flow that can be added/augmented to all edges in this path is 10 due to edge (v_5, v_7) resulting in the flow network illustrated in Figure 4. This algorithm is called the **Ford-Fulkerson** algorithm.

Available Functions (Clarifications)

If you have already completed the assignment, do not worry about this section. If you need more guidance on the return types of the Graph ADT, then you can utilize the following function declarations (this can be different):

- `public int numVertices();`
- `public Iterable<Vertex<V>> vertices();`
- `public int numEdges();`
- `public Iterable<Edges<E>> edges();`
- `public Edge<E> getEdge(Vertex<V> u,Vertex<V> v);`
- `public Iterable<Vertex<V>> endVertices(Edge<E> e);`
- `public Vertex<V> opposite(Vertex<V> v, Edge<E> e);`
- `public int outDegree(Vertex<V> v);`

- `public int inDegree(Vertex<V> v);`
- `public Iterable<Edges<E>> outgoingEdges(Vertex<V> v);`
- `public Iterable<Edges<E>> incomingEdges(Vertex<V> v);`
- `public void insertVertex(Vertex<V> v);`
- `public void insertEdge(Vertex<V> u, Vertex<V> v, E x);`
- `public void removeVertex(Vertex<V> v);`
- `public void removeEdge(Edge<E> e);`
- `public Vertex<V> getVertex(int label);`

For the `Edge<E>` ADT, the value that you are storing can be flow and flow capacity (combined into `E`). You can assume that the following `Edge<E>` interface functions are available (if you have already made your assumptions, then don't worry about this).

- `public int flowCapacity();`
- `public int flow();`

Similarly, for the `Vertex<V>` ADT, you can assume that the following interface functions are available:

- `public int getLabel();`

For the functions: `public void maximizeFlowNetwork(Graph fN, int s, int t)` and `public int breadthFirstPathSearch(Graph FN, int s, int d)`, the last two arguments indicate the labels/numbers assigned to the vertices (start and end nodes). You can utilize the function `getVertex` to obtain the reference to the vertex corresponding to the label. Or you can assume that the two arguments are of `Vertex<V>` type (change the input arguments type).

Grading

We will evaluate the logic behind your function implementations (`breadthFirstPathSearch` and `maximizeFlowNetwork`) (i.e. we will not run it). Marks will be deducted for syntax errors and logical errors.

Code Submission

You will use `git` to submit this assignment. Create a new repository for `LabAssignment3`:

- Create a new folder `SE2205B-LabAssignment3` and add the file `Assignment3.java` in which you will implement the two afore-mentioned functions.
- Open GitHub Desktop.
- Run `File` → `Add local repository`.
- Click `Choose` and browse for the folder `SE2205B-LabAssignment3`.
- You will get the following warning: `This directory does not appear to be a Git repository. Would you like to create a repository here instead?`
- Click on `Create a Repository` and then click on `Create Repository`.
- In the GitHub Desktop tool bar, click `Publish` repository, check the option `Keep this code private`, choose your GitHub account, and then click `Publish` repository.
- Now a new repository called "SE2205B-LabAssignment3" should appear in your GitHub account.
- You will need to add the instructor and the TAs to this repository. For this,

- In the GitHub account click the repository `SE2205B-LabAssignment3` and then click `Setting`
- Click `Collaborators & teams` option.
- At the bottom of the page and in the Collaborators section, enter the account id of the GitHub user you would like to add and then click `Add collaborator`.
- You need to add the following accounts to your repo:
  * **psrikan**
  * **LiYangHart**
  * **jliu2325**
  * **mithun03**

ENSURE that your work satisfies the following checklist:

- You submit before the deadline