HW8: Network Flows (Coding)

Objectives:
To practice solving problems by modeling them as network flow problems.

Due date: Friday, March 2nd by 11:00pm.

Materials: starter code (hw8-network-flows.tar.gz)

What to turn in:

- Your final version of `TableRounder.java` and any other Java files that you create or change.
- A PDF file containing your answers to the question from problem 3.

How to submit:
Use the canvas page (https://canvas.uw.edu/courses/1124392/assignments/4107372) to submit your solution.

Overview

In this assignment, you will write code to solve a table rounding problem by modelling it as a network flow problem (maximum flow). You are given a library that solves maximum flow, so your job is to create an appropriate instance of the maximum flow problem and then use the solution of that problem (produced by the library) to round the table properly.

Your Task

The input to the table rounding problem is a table of numbers that may not be integers. The problem asks you to round each of these numbers to an integer (either up or down) in such a manner that the sum of all the numbers in each row and column is also rounded up or down.

As an example, consider the following table. Each of the row sums are shown along the right side and the column sums along the bottom.

3.1	**6.8**	**7.3**	17.2
9.6	**2.4**	**0.7**	12.7
3.6	**1.2**	**6.5**	11.3
16.3	10.4	14.5	

The (1,1) entry contains 3.1, which we can round to either 3 or 4, the (1,2) entry contains 6.8, which we can round to either 6 or 7, and the (1,3) entry contains 7.3, which we can round to 7 or 8. However, we cannot round all of these entries down (to 3, 6, and 7) because the first row would then sum to 16, which is not a rounding of the initial row sum of 17.2. Hence, we need to round at least one of the numbers up. (Likewise, though, we cannot round them all up because the resulting row sum would be 19, which is too large to be a rounding of 17.2).

We can similarly determine how to round each row to so that its row sum is rounded properly, but the problem is made harder by the fact that the column sums also need to be properly rounded, so we cannot choose how to round each of the rows independently of the others.

As we will see in the assignment, this problem is easily solved by modeling it as a network flow problem.

Provided Code

Much of the code needed for the program we are making in this assignment is provided for you. In particular, you are already given code for reading and writing the data files, parsing the command-line arguments, and computing maximum flows. You will only need to fill in the bodies of a few methods that implement the core logic of the algorithm.

You will need to familiarize yourself with two files in the provided code: `TableRounder.java` and `GraphUtils.java`. While the other code will be used by your program, you will not need to use it directly.

`GraphUtils.``java` is a helper class that solves path and flow problems on graphs. The following methods will be likely be useful to you:

- **`maxFlow`**`(source, target, nodes, capacities)`
  Returns a maximum flow, where the flow on each edge is no more than the given capacity and every node is balanced except source and target.
- **`imbalanceAt`**`(node, nodes, flow)`
  Returns the outgoing flow minus the incoming flow at the given node (what we called the negative excess in lectures).

`TableRounder.``java` contains the entry point for the program. The `main` method parses the command-line arguments and loads the passed-in table. It then calls the `roundTable` method, which you will implement, to round the table, and then writes the result to the output.

`TableRounder` contains three method bodies that you will fill in as part of the assignment. `roundTable` is problem 4 below. The other two methods, which you will fill in for problems 1-2, will be used to implement `roundTable`.

Weighted Edges

The libraries above, as well as the methods that you will write, represent per-edge information like flows and capacities in a way that may seem strange at first.

The more obvious way to store this information would be to have a map from edges to doubles. Each edge is a pair of nodes, so if the nodes are, say, represented by integers, you could store a flow in a map of the form `Map<Pair<Integer>,Double>`. (In fact, there is a `Pair` class included in `GraphUtils.java`, and you are free to make this public so that you can use it in your code if you wish to.)

Providing a map like this allows you to retrieve the flow on a given edge using its `get` method. However, a more general way to do this is to just provide a **function** that takes two nodes and returns the flow on the edge between them.

Java provides a type ToDoubleBiFunction (https://docs.oracle.com/javase/8/docs/api/java/util/function/ToDoubleBiFunction.html) that represents a function that takes two arguments (in this case, nodes) and returns a double. If `f` is such a function, then `f.applyAsDouble(u, v)`, would return the flow on the edge (u, v).

The advantage of using a function like this is that you do not always have to construct the map. For example, if you want to create a zero flow, you can create a function that returns 0 for any inputs with one line of code, using Java's lambda syntax:

```
(u, v) -> 0
```

Once they are familiar, using lambda syntax should reduce the amount of code you have to write. The unit tests mentioned below are an example where the code would be much longer using maps.

However, if you want to work with maps as described, you are free to do so. In order to call a method that that wants a function, you can create one that looks up the value in your map using lambda syntax like this:

```
(u, v) -> !u.equals(v) && map.containsKey(Pair.of(u, v)) ? map.get(Pair.of(u, v)
) : 0.0
```

assuming that your map uses pairs of nodes as keys.

Note: the network flow library assumes that all graphs are dense. That is, it assumes there is an edge (u,v) and an edge (v,u) between every nodes u != v. This is done without any loss of generality since you can assign a capacity of 0 to any edge that you do not want to receive flow. In fact, it should reduce the amount of code you need to write because you do not need to construct both a map of capacities and a list of edges. (The edges are essentially inferred from those capacities that are greater than zero.)

Problems

1. Implement the method `findFeasibleDemandFlow`, which solves a flow problem with arbitrary demands on the nodes. You can reduce this to a normal maximum flow problem using the construction described in lecture.

   (A simple test for your method is provided in `TableRounderTest.java`.)

2. Implement the method `findFeasibleBoundedFlow`, which solves a flow problem with both lower and upper bounds on the edge capacities. You can reduce this to a feasible flow problem with demands using the construction described next. Once you have done so, you can call your method from problem 1 to get the answer.

   Most of the construction for this case is given was also given in lecture. The one difference is that, there, we assumed that we knew what flow we were looking for between the source and sink. In particular, we used that amount to set the desired demand at the source and the sink. In this case, though, we do not know how much flow we will need, so it is unclear how to set the demands at those two nodes.

We can get around this problem by adding an infinite capacity edge from the sink to the source. Now, whatever amount of flow goes from the source to the sink can now go back from the sink to the source along this edge. Hence, both the source and sink will be *balanced* like all the other nodes in the graph.

With that extra edge added, you can continue the construction as described in lecture. The source and sink are now just like any other nodes.

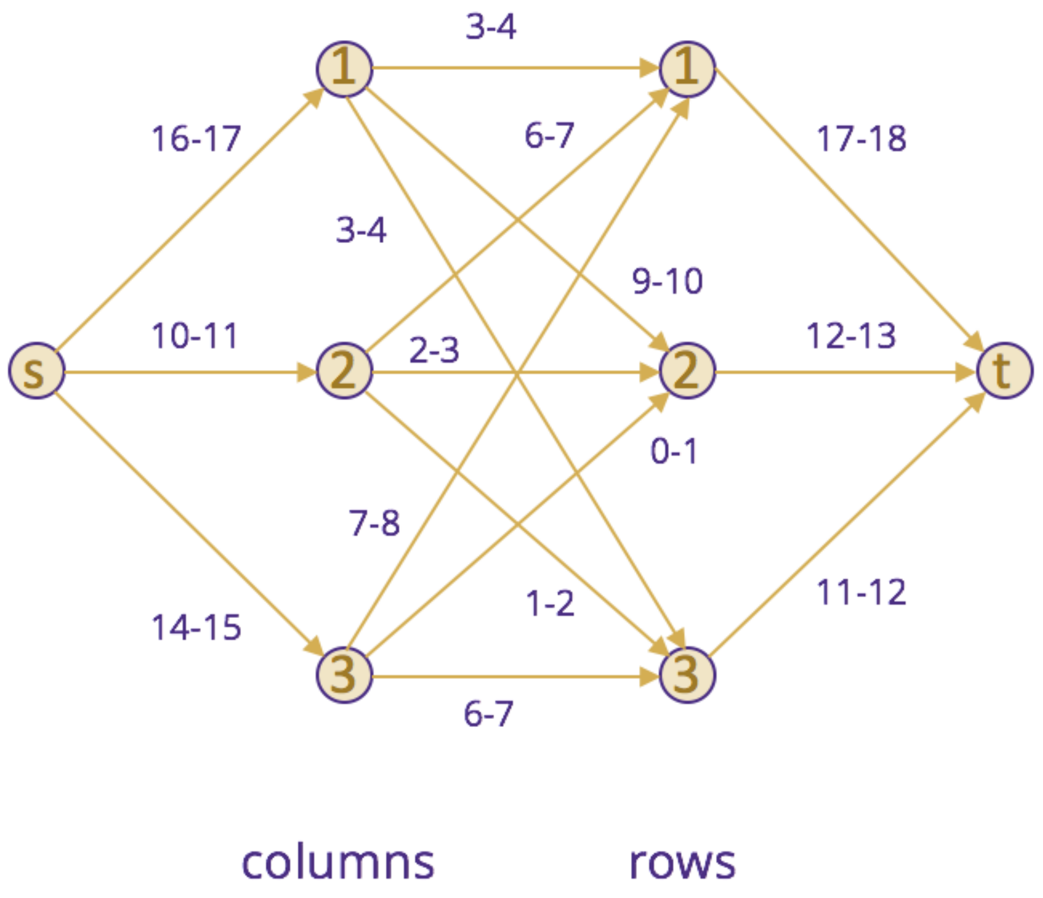
(A simple test for your method is provided in `TableRounderTest.java`.)

3. We will solve the table rounding problem by reducing it to a feasible flow problem with lower bounds, as described next.

   To round the given table, start by creating a bipartite graph with one set of nodes for columns and another set of nodes for rows. The flow on the edge (col, row) will indicate the rounded value for that entry in the table. To ensure that this is a valid rounding, set the lower and upper bounds for the flow on this edge to the floor and ceiling of the table entry, respectively.

   Next, for each column, create an edge (source, col), and set its lower and upper bounds to the floor and ceiling of that column sum. Similarly, for each row, create an edge (row, sink), and set its lower and upper bounds to the floor and ceilling of that row sum.

   For the table given above, the construction should look like this, where the capacity A-B means a lower bound of A and an upper bound of B:



   Answer the following questions:

   a. Briefly, describe why a feasible flow in this graph gives a valid rounding of the table. In particular, why is the flow on (col, row) a valid rounding of that table entry, and, more importantly, why is each row and column sum (which is a sum of the flows on some of those edges) also rounded properly?

   b. The library you were given in this problem solved maximum flow using Ford-Fulkerson. While many other algorithms exist for solving this problem (some of which would also work for this assignment), what additional property does Ford-Fulkerson provide that is necessary for this reduction to work?

4. Implement the method `roundTable`, which modifies the table entries to store rounded values in such a manner that row and column sums are also rounded. You can solve this by reducing to a feasible flow problem with lower bounds. First, construct the graph as described above. Then, use your `findFeasibleBoundedFlow` method to find a feasible flow. Then, replace each entry of the table with a rounded version, where the new value for (col, row) is given by the flow on the (col, row) edge.

Testing

You are provided with simple unit tests for the methods you will write in problems 1 and 2. You are encouraged to add to these unit tests to test more of behaviors of your methods.

You are also provided with three sample tables in the `data` subdirectory. Solutions are not provided because there is not necessarily a unique solution for how to round them. However, it is not difficult to check by hand whether your output is a correct rounding. Just make sure that each of the entries and each row and column sum is either the floor or ceiling of the same quantity from the input table.