Algorithms and Data structures
Algorithmen und Datenstrukturen

June 14, 2018

Lab 5 - Max Flow

Version 1.0 - english

Submission Deadline: July 1, 2018 @ 23:59

Submission System:
`https://aud.cdc.informatik.tu-darmstadt.de/cgi-bin/index.cgi`

1 Introduction

The city of Iksburg is a small, old town located next to a big highway. An unfortunate location since the highway needs to be temporarily closed for repairs and part of the traffic will have to be redirected through the narrow streets of Iksburg. The company responsible for the highway repairs has to find out how many cars they can redirect through Iksburg per hour, so they use a map of the old town's streets to make an estimate.

The map of the town is represented by a directed graph: vertices are crossings and edges are the streets. Each street has a number associated with it, representing the number of cars per hour that can travel on that street in one hour. We assume that the crossings can handle any number of cars.

The manager of the project (who took no computer science classes) claims that the map is useless and wants to simply redirect the incoming cars to the town to find out how many cars pass through in one hour. You, as a young programmer, consider that a more elegant way would be to write a program that finds the maximum flow of cars, using the given map as an input.

2 Task

Your task is to implement a Java class (`MaxFlow`) that calculates the maximum flow of cars (in one hour) that can travel between a given set of starting points (sources) and a given set of end points (destinations). The constructor of the class takes as input the name of a file containing the input map. The class must provide methods for calculating the maximum flow in terms of number of cars, and to give the output either as an integer (say 25 cars/hour) or as a new map in which the streets with unused capacity are indicated.

The input map is given as a directed graph with vertices and edges. Each vertex represents a crossing, and every edge represents a street. Every vertex has a name associated with it. Every edge has an associated value representing the maximum number of cars able to pass the street in one hour. Your class shall output a new map where the streets whose maximum capacity is not reached are marked by making the edges on the route bold (basically thicker). For this lab again the `dot` language is used for representing the map, both for input and output.[1] See section 5 for examples on how maps are represented.

You are to use the **Ford-Fulkerson** algorithm to find the maximum number of cars that can travel the route from A to B, in one hour.

[1] For sure, it is allowed to reuse code from previous labs, as long as it is your own code.

3 The Code

We provide the following Java classes and test files within the zip archive **Lab5.zip** which contains the Java project with the building blocks. The classes are split in two packages– the 'frame' and the 'lab' package. You should implement your solution starting with the skeleton code provided in the zip archive.

3.1 The 'frame' package

The submission system will use its own copy of these classes when testing your source code. Changes made by you within this package will not be considered by the submission system.

3.1.1 `AllTests.java`

This is the JUnit test case class. The test cases defined by this class are used to test the correctness of your solution. This class is also responsible for writing the output maps into the project directory.

3.2 The 'lab' package

This package contains the files you are allowed to modify. You are free to add additional classes (in new files within the lab packages, no subpackages), as well as to add any additional methods or variables to the provided classes, as long as you **do not change the signature (*name, parameter type and number*) of any given method**, as they will be used by the JUnit tests. Any source code changes that you make outside the 'lab' package will be ignored when you upload your source code to the submission system.

3.2.1 `MaxFlow.java`

For the `MaxFlow` class the following methods are to be implemented:

- `public MaxFlow(String filename)`

  The constructor of the class takes the name of the file containing the map as input. You can assume that the file is located in the same directory as the executable program.

- `public int findMaxFlow(String[] sources, String[] destinations)`

  This method returns the maximum number of cars able to travel the path between the crossings specified by the array `sources` and the crossings specified by the array `destinations`. The first parameter, `sources`, is an array containing the names of the source crossings. The second parameter, `destinations`, represents the crossings where the town ends. For instance, imagine the situation where Iksburg, a citadel town, has three entry gates and five exit gates. The maximum flow of cars will be calculated for the subsets of sources and destinations actually present on the map. If no name in `sources` is on the map it returns -1, if no name from the `destinations` array is on the map it returns -2, if no name from both arrays is on the map it returns -3. If no path is found it returns -4. If `sources` is identical to `destinations` it should return `Integer.MAX_VALUE`.

- `public ArrayList<String> findResidualNetwork (String[] sources, String[] destinations)`

  This method returns the streets on which the capacity was not completely used, between the sources and the destinations specified by the `sources` array and the `destinations` array respectively. It returns an `ArrayList<String>` containing the output map, where each edge representing a street whose full capacity was not completely used should be marked bold as described in Section 4. The labels of all edges will be changed to reflect the used capacity and sources and destination nodes will be marked accordingly (see Section 4). Each item in the `ArrayList` corresponds to one line of the output map. The output should be directly interpretable `dot` code. The crossings are identified by their names as given in the input file.

  If none of the sources or destination crossings are on the map or if no path is found, the returned map should be the original map, without marked edges. In any case, the contents of the input file is not to be changed.

3.3 Test Files

We provide four input files for testing as follows:

- `Iksburg1`
- `Iksburg2`
- `Iksburg3`
- `Iksburg4`

You are encouraged to test your solution using additional input files as well. Note that you need to write your own JUnit test cases in order to run with your customized input graphs. Think about the assumptions made on the input. To make sure that your solution works, do test it with all the test cases provided. Apart from the given input files, the submission system will test your solution with several additional input files, to confirm the correctness of your program. Those additional input files are made available to you but are not included in the zip file.

3.4 Additional Hints

- You may reuse code from previos labs, as long as it is your own code.

- You may start with implementing a class to represent edges and one to represent vertices, in order to finally represent the graph as a list or edges and a list of vertices.
- Use the constructor of the `MaxFlow` class to read the test file into your graph representation.

4 Edges, Sources and Destinations in `dot`

As input and output format for this lab we will use a subset of the `dot` language for directed graphs. This section will only describe this subset. Please refer to the online documentation for getting more information and downloading tools:

`https://www.graphviz.org`

Let's assume that in the input map of the town, we have the following line describing an edge that has a capacity of 10 cars per hour:[2]

```
A -> B [label="10"];
```

In the graph all sources will be represented using bold double circles and all destinations using bold circles. The corresponding markup in dot code is described below. Let A be the source and B the destination. Depending on how much of this street's capacity was actually used, the edge will be represented by a bold line if there is some capacity left and with a normal line if 100% was used. The label of the edge will change from only displaying the original capacity to a format as `<original capacity> - <used capacity>`. For example:

1. if full capacity was used:

   ```
   A -> B [label="10-10"];
   ```

2. if only a part was used (for instance, 8 cars out of the maximum of 10):

   ```
   A -> B [label="10-8"][style=bold];
   ```

Marking the sources and destinations is done as below. If there are multiple sources and multiple destinations, all of them should be marked:

```
A [shape=doublecircle][style=bold];
B [shape=circle][style=bold];
```

For more examples, see Section 5. The names of the vertices can also be actual names of places, containing small and big caps and "_". For instance "Taunus_Platz". No blanks are used within names.

[2] Since we are counting cars, you can assume all numbers to be integers

5 Examples

Figure 1 and 2 show an example of an input map.

```
digraph {
A -> B [label="10"];
A -> C [label="10"];
A -> D [label="8"];
B -> C [label="3"];
B -> E [label="5"];
C -> E [label="15"];
D -> E [label="3"];
}
```

Figure 1: The input file

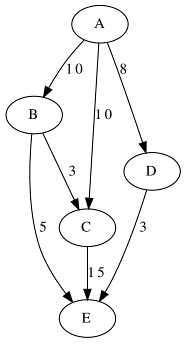

Figure 2: The graph described by the `dot` code in Figure 1

Calling the method `findMaxFlow ("A", "E")` on this graph should return `21`. Calling `findResidualNetwork ("A", "E")` will return an `ArrayList` object, containing as elements the lines of the `dot` code in Figure 3.

```
digraph {
A -> B [label="10-8"][style=bold];
A -> C [label="10-10"];
A -> D [label="8-3"][style=bold];
B -> C [label="3-3"];
B -> E [label="5-5"];
C -> E [label="15-13"][style=bold];
D -> E [label="3-3"];
A [shape=doublecircle][style=bold];
E [shape=circle][style=bold];
}
```

Figure 3: The code for Figure 4

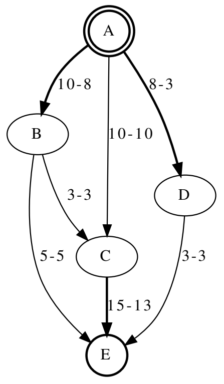

Figure 4: The graph resulted after calling `findResidualNetwork ("A", "E")`

Calling `findResidualNetwork (sources, destinations)` (with `sources = "A", "B"` and `destinations = "D", "E"`) will return an `ArrayList` object, containing as elements the lines of the `dot` code in Figure 5. In this case, the solution is not unique, depending on the algorithm for finding the augmenting path.*

```
digraph {
A -> B [label="10-0"][style=bold];
A -> C [label="10-10"];
A -> D [label="8-8"];
B -> C [label="3-3"];
B -> E [label="5-5"];
C -> E [label="15-13"][style=bold];
D -> E [label="3-0"][style=bold];
A [shape=doublecircle][style=bold];
B [shape=doublecircle][style=bold];
D [shape=circle][style=bold];
E [shape=circle][style=bold];
}
```

Figure 5: The code for Figure 6

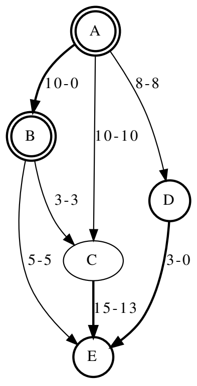

Figure 6: The graph resulted after calling `findResidualNetwork (("A", "B"), ("D", "E"))`

* Actually, the edge from A to B might be ”10-8”, depending on how the augmenting path was chosen.