Algorithms H Assessed Coursework Implementation Report

Iva Babukova
2030458B

Task 1

1. Status Report

The status of all methods implemented for this task is working. It was tested after implementing task 2 with the input file supplied as well as additional test file made by me.

2. Justification of Operation

`isFlow()`

To check whether the assigned flow in the network is valid, I used the definitions of a valid flow studied during the lectures, namely:

- the capacity constraint: for every edge in the network, the flow through that edge is always non-negative and less or equal to the edge capacity.
- the flow conservation constraint: for every vertex in the network except the source and the sink, the total incoming flow is equal to the total outgoing flow.

For the implementation of these properties, I construct a matrix, called flowMatrix, where flowMatrix[i][j] is equal to the flow of the edge (i,j), where i and j are vertices. If such an edge does not exist, flowMatrix[i][j] = 0. While constructing the flow matrix, I check the capacity constraint for the particular flow. If it violated, the method terminates and false is returned. The flow matrix initialization step is of O(numVertices^2) complexity when the flow capacity constraint is preserved.
Using the constructed flow matrix, I build two arrays of integers, called inflow and outflow, where inflow[i] contains the total incoming flow for vertex with label i and outflow[i] contains the total outgoing flow for the vertex labeled as i. The total outgoing flow for a vertex labeled as i is then the sum of all flows in the i-th row of the flow matrix. Consequently, the total incoming flow for a vertex labeled as i is the i-th column in the flow matrix.
Table 1 below shows an illustrated example of calculating the inflow and outflow.

	inflow[i]	**0**	**2**	**2**	**3**	**3**	**0**
outflow[i]	i	**0**	**1**	**2**	**3**	**4**	**5**
5	**0**	0	2	0	3	0	0
2	**1**	0	0	0	0	2	0
2	**2**	0	0	0	0	0	2
3	**3**	0	0	2	0	1	0
3	**4**	0	0	0	0	0	3
0	**5**	0	0	0	0	0	0

Table 1: The flowMatrix (in gray) and the sum of the entries of each row (in red), and the sum of the entries of each column (in blue) that form the inflow and outflow respectively.

`getValue()`
Here, I use the network property stated above to compute the total outgoing flow from the source, which is the total flow of the network. The complexity of this method is O(numVertices)

`printFlow()`
I used the print format specified in the assignment instructions. For every vertex U, I find every existing edge (U, Vi), where i is between 1 and numVertices inclusive, and print its properties. The complexity of this method is O(numVertices^2).

Task 2

1. Status Report

The status of this task is working.

2. Justification of Operation

`ResidualGraph(Network net)`

The residual graph *RG* of a network *N* has the same vertex set as *N*, but the direction and the properties of the edges that are in *N* are different. In particular, an edge *(u, v)* is in *RG* if and only if:

- *(u,v)* is in *N*, also called a forward edge and its capacity in RG is equal to the capacity of *(u,v)* in N - the flow of *(u, v)* in *N*
- *(v,u)* is in N, also called a backward edge. Its capacity in RG is the same as the flow of (v, u) in N

The main idea behind creating a *RG* of *N* is to be able to find an augmenting path (if it exists), during the execution of the Ford-Fulkerson algorithm and consequently increase the flow of the network. An augmenting path is a path from the source to the sink using the intermediate vertices in the residual graph. There may be more than one such paths, only one or none (the latter is an indication that the current flow can not be increased).
As the residual graph of a network has different edges and has only capacity on the edges (according to the lecture slides), I create a new *Edge[ ][ ]* adjacency matrix, called *residualAdjMatrix* to record the edges of the graph. I also create a new adjacency list of type *ArrayList<LinkedList<Vertex>>* that contains the adjacency lists of the residual graph as an array, with an element for each vertex *v* giving a linked list corresponding to *v*'s adjacency list.
For every *v* in the network *N* and for every neighbor *u* it has, I take the edge *(v, u)* and create the corresponding edge(s) in the residual graph, following the principles explained in the previous paragraphs. The edges are created in a separate function, called

setResidualEdges (the edges can be created directly in the constructor, but I decided to do it in a separate function for core readability purposes). More details about this function can be seen in the code.

`findAugmentingPath()`
I use a breadth-first search starting from the source node to find the shortest augmenting path in the graph. An augmenting path that is not necessarily the shortest path can be still used, but I decided to use the shortest one to remove the possibility of the worst-case scenario explained in the lectures.
I have implemented a function called *bfs*- a modified form of breadth-first search that returns an integer array *pred*[ ], where the value of *pred[i]* is equal to the label of the predecessor vertex of the vertex labeled as *i*. A predecessor of a vertex *u* is a vertex *v* such that *u* is visited from *v* when performing breadth first search.
The array returned from *bfs* is used to find the shortest path from the source to the sink. The algorithm starts from the sink and backtracks until reaching the source vertex label. The backtracking step (going one node closer to the source) is achieved by updating the current label to be equal to its predecessor: *pred[current label]*. The result from these steps is the list of the vertices that take part in the shortest augmenting path in the graph, referred to in the code as *verticesPath*. Using the *verticesPath* list, all edges that form the augmenting path are found and returned.

`fordFulkerson()`
This method is the implementation of the Ford-Fulkerson algorithm studied in the lectures. The algorithm starts with initializing step (lines 83-92) of complexity *O(numVertices^2)* that sets the flow to the network to 0. After that the algorithm repeatedly tries to find an augmenting path and increase the flow of the network with the minimum edge capacity of the augmenting path. The infinite for loop terminated only when no such path is found, i.e. the network flow can not be increased and the current network flow is the maximum flow. The claim that a flow is maximum if and only if it admits no augmenting path is also known as the augmenting path theorem and it was proved during the lectures.

Task 3

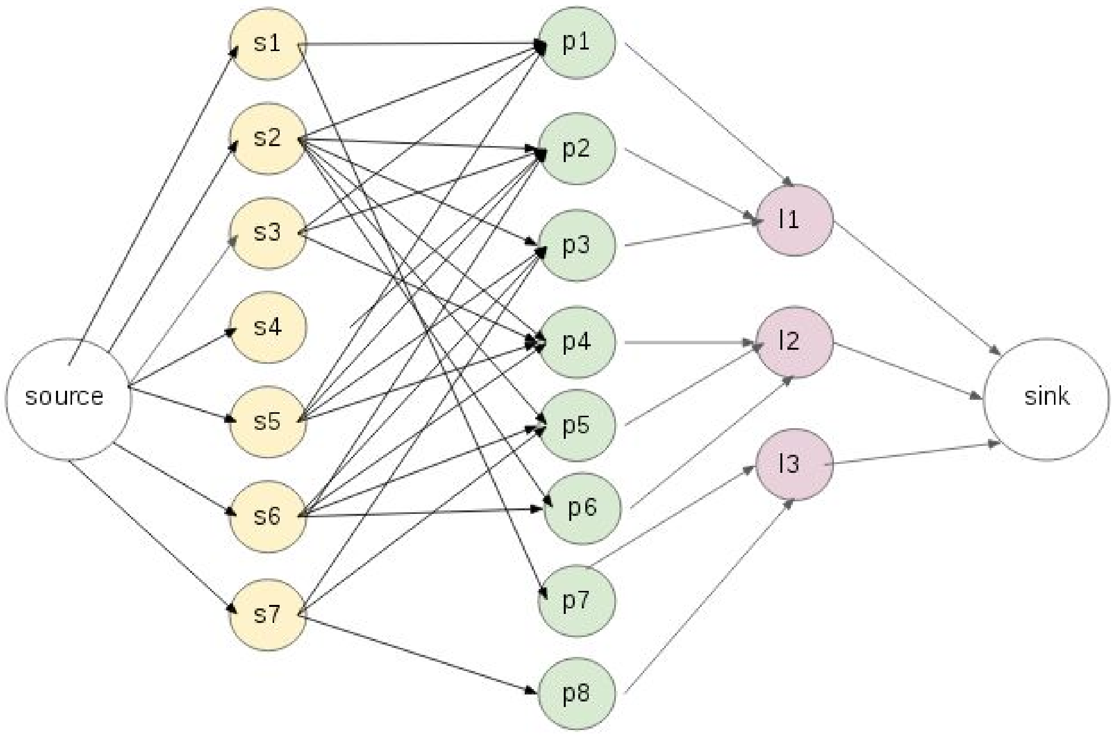
1. Status Report

The status of this task is working.

2. Justification of Operation

Figure 1 shows the model I chose for representing the Student Allocation Problem(SAP) as finding a maximum flow in a network. The instance of the problem on the figure has 7 students, 8 projects and 3 lecturers. In this model, students are the sources and the lecturers are the sinks. As a network can have only one source and one sink, I create two additional vertices: the source, connected to all students with an edge directed to the corresponding student (with capacity 1, as we want to assign no more than 1 project to a student), and the sink: connected to all lecturers, where each edge is directed towards the sink with capacity the same as the lecturer upper quota(we can’t assign more students to a lecturer than he is able to supervise). When a student Si ranks a project Pj, we draw an edge Eij from Si directed to Pj that has capacity 1(as we want to assign no more than 1 project to a student). If a project Pi is proposed by a lecturer Lj, we draw an edge Eij directed to the lecturer with capacity equal to the project’s capacity (we do not want to assign too many students to a project and exceed its capacity). Finding a solution that satisfies as many students as possible is now a maximum flow in a network problem and I use Ford-Fulkerson algorithm to solve it.

Figure 1: Student Allocation Problem Model



In order to implement this model, I created three new classes: Student, Project and Lecturer.

Class Student has a boolean isSE that is true if the corresponding object instance is a software engineer or false otherwise. Student also has a list of choices which are the labels of all Projects that the student finds acceptable.

Class Project has a capacity (int), isSE boolean that is true if the project is suitable for software engineering students or false otherwise, and an int called “proposedBy” that is equal to the label of the Lecturer who proposed the project.

Class Lecturer has only one field: the lecturer’s capacity.

All three classes mentioned above extend the class Vertex. In order to have unique label for each vertex, labels are assigned in the following way. If |S| denotes the number of students,

|P| is the number of projects and |L| - the number of lecturers, then the assigned labels for each vertex are as follows. The label of the source is 0 and the label of the sink is (|S| + |P| + |L|) (as previously). The labels assigned for the students vertices are from 1 to |S|, the project labels are from (|S| + 1) to (|S| + |P|) and the labels for the lecturer vertices are numbered starting from (|S| + |P| + 1) to (|S| + |P| + |L|).

In class DirectedGraph one method called "addToVertices" is added which takes a Vertex as an argument and adds it to the list of vertices. In class Network the method "printFlow" is re-written, as the requirements for the results printing are changed. Apart from these, nothing else in the networkFlow package is changed.

Method "readNetworkFromFile" is changed completely to support the new input format. Here, I add all edges between the vertices explained previously after enough information is read in from the file. More information and comments on the implementation of this method is available in the source code.

Task 4

The status of this task is implemented and working. The solution is as follows. One needs to run the Ford-Fulkerson twice:

- the first time, the capacity of each lecturer should be equal to their lower quota. If the maximum flow after running the algorithm is less than the sum of the lower quotas of every lecturer, then there is no solution, the program terminates and returns false.
- if the first run of Ford-Fulkerson is successful, the second Ford-Fulkerson has to be run with setting the capacities of the teachers to their upper quota, but not changing the flow values of each edge that were set up after the initial run of Ford Fulkerson.

To implement this, several alterations were done. First in the Lecturer class I have introduced another field for lower quota of type int. In FordFulk class, I have added additional line in the "readNetworkFromFile" method to read in the lower quota and a line that sets up the lecturer lower quota as the capacity of the edge from this lecturer to the sink (which is used for the first run of the fordFulk algorithm).
In Ford-Fulkerson algorithm is modified in the following way. It takes a parameter initial flow. If the initial flow is 0, that means that the algorithm is run for first time and we set up flow 0 for every edge. If the initial flow is -1, this is an indicator that fordFulk is run for second time. Therefore, I set up the capacity of each (lecturer i, sink) edge to be equal to the upper quota of lecturer i. There is an additional modification of the return type of the fordFulkerson method: it is of type boolean. This is added for the first time when the algorithm is run in order to check whether there is a solution. The algorithm returns false if the network maximum flow is less than the sum of the lower quotas of all lecturers or true otherwise.
In Main.java, I call the fordFulk algorithm twice, as explained previously. If the first run returns false, I print that there is no solution and the program exits. Otherwise, I call fordFulkerson twice to check whether I can further increase the current maximum flow.