Flow Report

Kasper Hjort Berthelsen, Jesper Kato Jensen, Frederik Schou Madsen, Heidi Regina Schröder, Julie Schou Sørensen and Jon Voigt Tøttrup

November 4, 2018

Results

Our implementation successfully computes a flow of 163 on the input file, confirming the analysis of the American enemy.

We have analysed the possibilities of decreasing the capacities near Minsk. Our analysis is summaries in the following table:

Case	4W–48	4W–49	Effect on flow
1	30	20	no change
2	20	30	no change
3	20	20	no change
4	20	10	-10
5	10	20	-10
6	10	10	-20

Implementation details

The MaximumFlow-implementation is build up by the .java files:

- `Main.java`: The main method
- `FileReader.java`: Reads the input file
- `FlowNetwork.java`: The graph data type
- `FlowEdge.java`: The edge data type
- `FordFulkerson.java`: The algorithm solving the maximum flow problem

We use the implementation of Ford-Fulkerson's Maximum-Flow algorithm described in 7.1 of Kleinberg and Tardos, *Algorithm Design*, Pearson Education 2006 as well as page 886-902 of Sedgewick and Wayne, *Algorithms*, Fourth edition, Pearson Education 2011.

The algorithm is based on the assumption that the flow network is a directed graph, but in the assignment the flow network is undirected. To overcome this, we implement it as a directed graph where all edges are added as both a forward and a backward edge. Also,

the names of the vertices in the input file aren't unique, so we name the vertices with indexes from 0 (the source) to $V - 1$ (the sink), where V is the number of vertices.

The data structures used are as follows: The flow network is implemented as an array of linked lists of edges (`FlowEdges`). The array indexes denote the vertices and the linked list for each vertex is an adjacency list of which edges are adjacent from that vertex.

The data type for `FlowEdge` is defines by the following API:

public	class	FlowEdge	
		`FlowEdge(int v, int w, int capacity)`	
	int	`capacity()`	capacity of this edge
	int	`either()`	either of this edge's vertices
	int	`other()`	the other vertex
	int	`residualCapacityTo(int v)`	residual capacity toward v
	int	`updateResidualCapacityTo(int v, int b)`	add b flow toward v

Each edge goes from a vertex v to a vertex w and has a capacity. Also, each edge has a residual capacity, `residualCap`, that is initiated with 0. The original capacity is final and only the residual capacity can be updated. In this way the residual graph is implemented in the `FlowEdge`-object instead of declaring a second graph. The residual graph is maintained via the methods `residualCapacityTo(int v)` and `updateResidualCapacityTo(int v, int b)`. The method `residualCapacityTo(int v)` returns the residual capacity toward v and the method `updateResidualCapacityTo(int v, int b)` updates `residualCap` with the bottleneck of the augmenting path.

To find an augmenting path we use Breadth-first search (the private method `hasAugmentingPath` in class `FordFulkerson`). This method checks whether there is a path from the source s to the sink t. It returns `true` if there is such a path and `false` if there is not. The path is stored in an array of `FlowEdges`, called `edgeTo`. It is now easy to trace back the path by the loop:

```
for(int v = t; v != s; v = edgeTo[v].other(v))
```

starting from the sink and updating with the other vertex of the edge on the augmenting path.

If no more possible path's are available the overall loop in `FordFulkerson` stops and calls the method `minCut`, that prints the minimum cut. The method `minCut` loop through all the vertices that the source s can reach (with `true` in the array `visited`). For each such vertex the method loops through the adjacent edges and checks whether the other vertex of the edge is reachable from the source. If not, it is part of the minimum cut and the method prints the value.

The overall running time of our implementation is $O(C(V + E))$, where

- $C = \sum_{e \text{ out of } s} c_e$ and c_e is the capacity of edge e.
- V is the number of vertices.
- E is the number of edges.