# A graph theoretic solvability check for biproportional multiplier methods

Diplomarbeit
im
Studiengang
Diplom-Wirtschaftsmathematik
von
Bianca Joas

eingereicht
beim Institut für Mathematik der
Mathematisch-Naturwissenschaftlichen Fakultät der
Universität Augsburg
Juli 2005

Erstgutachter: Prof. Dr. Friedrich Pukelsheim
Zweitgutachter: Prof. Antony R. Unwin

**Author**

Bianca Joas

**Title**

A graph theoretic solvability check for biproportional multiplier methods.

**Abstract**

This work is concerned with the existence problem for biproportional multi- plier methods. Given a problem and its underlying biproportional multiplier method, we determine a necessary and sufficient condition for a feasible apportionment to exist. In the case of nonexistence of a solution, we establish a procedure to define a set of parties and a set of districts, denying a feasible allotment. Imposing further restrictions on the problems, we derive some information about the properties of these sets. Based on graph theoretic considerations, we formulate an efficient algorithm to establish existence and comment on its implementation in Java.

# Contents

# List of Figures

# Chapter 1

# Introduction

Democratic elections are either based on majority voting, or on proportional representation. In the case of majority voting, the applicant with the relative or absolute majority of votes is elected. Proportional representation aims to portray the political views of the electorate within the parliament, such that each applicant is allotted the number of seats proportional to its share of votes.[1] Diverse variations of electoral systems emerge, based on one of the above mentioned ideas, or representing a combination of both. In this thesis, we focus on multiplier methods for proportional representation systems.

A biproportional multiplier method, as given for example within the New Zurich apportionment method[2], is subject to the apportionment of a certain number of seats among both parties and districts, such that every seat is allotted to one party and to one district. The assigned number of seats, and thus the apportionment, is always integer valued. The apportionment underlies both weights, given as the result of an election, and prespecified marginal constraints. The weights are represented in a weight matrix, where the entry in row $i$ and column $j$ specifies the weight of party $j$ in district $i$. The apportionment is to reflect the weights of each party per district in a proportional way. At the same time, each district and each party is assigned a certain number of seats, denoted as the marginal constraints, that have to be met with the allotment. The district magnitude often is prescribed by law, whereas the number of seats for each party is the result of a superapportionment due to the votes. The following work is concerned with a graph theoretic approach to establish existence for biproportional multiplier methods.

Checking upon the customary condition for a feasible apportionment as given by Bacharach[3] is inefficient, especially for big problems. Hence, we take an alternative approach as suggested by Balinski and Demange[4], by transforming the problem into a network and solving a maximum flow problem. The maximum flow value establishes existence or nonexistence. In the case of nonexistence, we

[1] See Schmidt and Seidel [13] on page 34 and 35.
[2] For details see Pukelsheim and Schuhmacher [11].
[3] See [3] on page 51.
[4] See [5] on page 707.

use cuts to determine a set of parties and a set of districts, that are responsible for the failure.
In the end, we summarize this procedure and formulate it as an algorithm, which is implemented in Java[5].

[5]The program is a subroutine for the BAZI program [14] by Prof. Dr. Friedrich Pukelsheim and the BAZI team, university Augsburg, which calculates, if possible, the apportionment for various apportionment methods.

# Chapter 2

# Biproportional apportionment problems

Section 1 and 2 provide definitions of biproportional problems and multiplier methods. In section 3 and 4 we establish a necessary and sufficient condition for the existence of a feasible apportionment to a biproportional problem, distinguishing between different kinds of multiplier methods.

## 2.1 Biproportional problems

We assume that there are $k$ districts, where the electors can choose between $l$ parties of their favor. The number of seats, which are to be assigned, is some positive integer valued number $h$, as for 'house size'. The weight for each party $j$ in district $i$ is denoted by $w_{ij}$ and represented in the weight matrix $W \in \mathbb{R}^{k\times l}$. Denote the marginal district magnitude for district $i = 1, \ldots, k$ with $r_i \in \mathbb{N}_0$ and the number of seats for party $j = 1, \ldots, l$ with $c_j \in \mathbb{N}_0$. This is conformed with the matrix terminology, where rows represent districts and columns stand for parties. The marginal constraints both for parties and districts must sum up to $h$. Further on, we assume, that all entries of $W$ are nonnegative and there are no rows or columns of zeros in $W$.
For notational reasons, denote the row sum over all entries in matrix $W$ in row $i$ by

$$w_{i+} := \sum_{j=1}^{l} w_{ij}.$$

Similarly $w_{+j}$ denotes the sum over all entries in column $j$. For any subset $J \subset \{1, \ldots, l\}$ we define the sum over all entries in matrix $W$ in row $i$ and column $j$, with $j \in J$, as follows

$$w_{iJ} := \sum_{j\epsilon J} w_{ij}.$$

The same notation will be used for any subset $I \subset \{1, \ldots, k\}$, to get the sum over all entries in column $j$ and row $i$, with $i \in I$. Summation over an empty set is defined to be zero.

**Definition 2.1.1** *A pair $(W, \sigma)$, consisting of a weight matrix $W = (w_{ij}) \geq \mathbf{0}$, $i \leq k$, $j \leq l$, with nonzero columns and rows and a nonnegative vector $\sigma = (\mathbf{r}, \mathbf{c})$ with $\mathbf{r} = (r_i) \in \mathbb{N}_0^k$, $i \leq k$ , $\mathbf{c} = (c_j) \in \mathbb{N}_0^l$, $j \leq l$, where $\mathbf{c}_+ = \mathbf{r}_+$, is called* **problem**.

**Definition 2.1.2** *Let $\sigma = (\mathbf{r}, \mathbf{c})$ with $\mathbf{r} = (r_i) \in \mathbb{N}_0^k$, $i \leq k$ and $\mathbf{c} = (c_j) \in \mathbb{N}_0^l$, $j \leq l$. The* **region of allocation** *$R(\sigma)$ is defined as*

$$R(\sigma) := \{a = (a_{ij}) \geq \mathbf{0} : a_{i+} = r_i,\ i \leq k,\ a_{+j} = c_j,\ j \leq l,\ a_{ij} \in \mathbb{N}_0\}.$$

$R(\sigma)$ contains all matrices with nonnegative integer valued entries, that fulfill the marginal constraints, given by the vector $\sigma$. If any feasible apportionment exists, it will be a nonempty subset of $R(\sigma)$.[1]

A central role within the apportionment play the scaling methods, which are tightly connected to rounding functions. Thus, we first take a closer look on the last ones, before turning to the multiplier methods.

## 2.2 Scaling methods

As a prerequisite for scaling methods, we turn our attention to the rounding of numbers. For this reason, take $[\cdot]_s$ to be a rounding function, where a positive number $x \in [n, n+1]$, $n \in \mathbb{N}_0$, is mapped due to a dividing point $s(n)$ in the interval $[n, n+1]$ either to the smallest integer larger than $x$, if $x > s(n)$, or to the largest integer smaller than $x$, if $x < s(n)$. In the case of a tie, $x = s(n)$, $x$ can be either rounded up or down[2]. The set of dividing points $s(n)$ is called signpost sequence as proposed by Balinski and Young[3] and used for the definition of the s-rounding as given in Balinski and Demange[4].

**Definition 2.2.1** *Define the* **signpost sequence** *$s$ as a strictly increasing function over the nonnegative integers, such that $s(n) \in [n, n+1]$ for all $n \in \mathbb{N}_0$ and there exist no integer valued numbers $a \geq 1$ and $b \geq 0$, such that $s(a) = a$ and $s(b) = b + 1$.*

We define s(-1):=0 to state the following definition.

[1] See Balinski and Demange [5] on page 701.
[2] See Pukelsheim [12].
[3] See [6] on page 62.
[4] See [5] on page 711.

**Definition 2.2.2** *Let $s$ be a signpost sequence. Define the* **s-rounding** $[\cdot]_s$ *according to $s$ as an integer-valued function, such that for any $x \geq 0$ and $n \in \mathbb{N}_0$:*

$$[x]_s = \begin{cases} n, & \text{if } s(n-1) < x < s(n), \\ n \text{ or } n+1, & \text{if } x = s(n), \\ 0, & \text{if } x{=}0\ . \end{cases}$$

Often, simple rounding of the given data does not suffice to get a feasible apportionment. Thus, the data must be scaled beforehand. For an example see the vector $z = (1.6, 2.1, 4.3, 5.3)$, each component representing the share of one party. To apportion six seats among the four parties in a proportional way, using standard rounding, the vector has to be multiplied with a number $\mu \in [0.35, 0.47]$, so that the scaled data, when rounded, add up to six. Notice that for notational reason the given interval was shortened and does not specify the whole scope of possible multipliers. Take $\mu = 0.4$, to get the scaled vector $\mu z = (0.64, 0.84, 1.72, 2.12)$ and its associated rounded version $(1, 1, 2, 2)$ with $s(n) = n + 1/2$. Now, the components of the rounded version add up to six and thus represent a feasible apportionment. This leads to the definition of a multiplier method of rounding vectors, as presented by Balinski and Rachev[5], given $h$ as the number of seats to be apportioned and $z$ as the weight vector.

**Definition 2.2.3** *Let $h$ be a positive integer and $z \in \mathbb{R}^k$, with $z_i \geq 0$, $i \leq k$. The* **multiplier method** *of rounding vectors, based on the signpost sequence $s$ for $(z, h)$ is defined as*

$$A^s(z,h) = \{a = (a_i),\ i \leq k :\ a_i = [\mu z_i]_s,\ \mu > 0 \text{ and } a_+ = h\}.$$

In the biproportional case, given a weight matrix $W \in \mathbb{R}^{k \times l}$, two positive multipliers $\lambda \in \mathbb{R}^k$ and $\rho \in \mathbb{R}^l$ have to be specified, which scale the weights row and column-wise, before rounding.[6]

**Definition 2.2.4** *Let the pair $(W, \sigma)$ be a problem. The* **biproportional multiplier method** *of rounding matrices based on the signpost sequence $s$ for $(W, \sigma)$ is defined as*

$$A^s(W,\sigma) = \{a = (a_{ij}) \in R(\sigma),\ i \leq k,\ j \leq l :\ a_{ij} = [\lambda_i w_{ij} \rho_j]_s \text{ for } \lambda, \rho > \mathbf{0}\}.$$

A multiplier method with its associated signpost sequence $s$, is called **impervious**, if $s(0) = 0$ and **pervious**, if $s(0) > 0$.[7] In the first case every positive weight, as small as it may be, is rewarded at least one seat, whereas in the second case positive weight of a party does not guarantee a positive apportionment. This is the case, if the multiplier is small enough, so that the scaled weight

[5] See [7] on page 4.
[6] See Balinski and Rachev [7] on page 19.
[7] See Pukelsheim [12].

gets smaller than $s(0)$. Henceforth, we assume for impervious scaling methods the number of positive weights within the weight matrix $W$ to be less than or equal to $h$. To establish existence of a solution to a biproportional apportionment problem, it is important, wether the used scaling method is pervious or impervious.

## 2.3 Existence for pervious multiplier methods

With the existence of a solution of an apportionment to the problem $(W, \sigma)$ based on $s$, we mean the nonemptyness of the set $A^s(W, \sigma)$. Thus, we have to find a necessary and sufficient condition, for $A^s(W, \sigma)$ to be nonempty. For this reason define $R^0(W, \sigma)$ as follows

$$R^0(W, \sigma) := \{\ a\ \epsilon\ R(\sigma) :\ w_{ij} = 0\ \Rightarrow\ a_{ij} = 0\}$$

The set $R^0(W, \sigma)$ consists of all matrices, that both fulfill the marginal row and column constraints and are compatible with the weight matrix $W$, such that all zeros of $W$ are preserved.[8]

**Theorem 2.3.1** *Let $(W, \sigma)$ be a problem and $s$ a pervious signpost sequence. $A^s(W, \sigma)$ is nonempty if and only if $R^0(W, \sigma)$ is nonempty.*

**Proof.** The theorem is proofed constructively by an algorithm and presented in the appendix A.

Now we can derive a condition for the existence of a solution of $(W, \sigma)$, dependent on the set $R^0(W, \sigma)$.

**Theorem 2.3.2** *The set $R^0(W, \sigma)$ is nonempty if and only if*

$$\begin{cases} c_J \geq r_I \\ \text{for any } I \subset \{1, \ldots, k\} \text{ and } J \subset \{1, \ldots, l\} \text{ with} \quad w_{I\overline{J}} = 0. \end{cases} \tag{2.3.1}$$

**Proof.** The proof will be established in section 5.2.

Assume that all entries of the weight matrix are positive. In this case condition (2.3.1) holds. Set $I = \emptyset$ or $\overline{J} = \emptyset$ to get $w_{I\overline{J}} = 0$. Thus, we either have $c_J \geq r_I = 0$ or $h = c_J \geq r_I$. This leads to the following proposition.

**Proposition 2.3.1** *Let $(W, \sigma)$ be a problem and the underlying multiplier method be pervious. If $W$ is strictly positive, then $A^s(W, \sigma)$ is nonempty.*

[8] See Balinski and Demange [5] on page 707.

## 2.4 Existence for impervious multiplier methods

To establish existence in the case of impervious multiplier methods we define the support matrix $e = (e_{ij})$, $i \leq k$ and $j \leq l$, to the corresponding weight matrix $W$ as

$$e_{ij} = \begin{cases} 0, & \text{if } w_{ij} = 0, \\ 1, & \text{otherwise.} \end{cases}$$

Summation over row $i$ of matrix $e$ leads to the number of parties with positive weight in district $i$ of the corresponding weight matrix $W$ and thus to a lower bound for the seats, which are to be apportioned to district $i$. An apportionment can only be found, if the marginal constraint for district $i$, denoted by $r_i$, is larger than or equal to $e_{i+}$. Thus, we only regard weight matrices, which satisfy this necessary condition. Similarly, we assume that $e_{+j} \leq c_j$.
Given a problem $(W, \sigma)$, we define the set

$$R^1(W,\sigma) := \{\ a\ \in\ R^0(W,\sigma):\ w_{ij} > 0\ \Rightarrow\ a_{ij} \geq\ 1\}$$

as a subset of $R^0(\sigma)$, such that the number of apportioned seats $a_{ij}$ is larger than or equal to one if and only if party $j$ has positive weight in district $i$.[9]

**Theorem 2.4.1** *Let $(W, \sigma)$ be a problem and the given multiplier method be impervious. $A^s(W, \sigma)$ is nonempty if and only if $R^1(W, \sigma)$ is nonempty.*

**Proof.** The theorem is proofed constructively by an algorithm and presented in the appendix A.

Similar to the case of pervious multiplier methods, we can find a necessary and sufficient condition for $R^1(W, \sigma)$ to be nonempty.

**Theorem 2.4.2** *The set $R^1(W, \sigma)$ is nonempty if and only if*

$$\begin{cases} c_J \geq r_I + e_{\overline{I}J} \\ \text{for any } I \subset \{1,\ldots,k\} \text{ and } J \subset \{1,\ldots,l\} \text{ with} \quad w_{I\overline{J}} = 0. \end{cases} \tag{2.4.1}$$

**Proof.** The proof will be established in section 5.2.

Again, assume all weights in the weight matrix $W$ to be strictly positive. For $I = \emptyset$ or $\overline{J} = \emptyset$, with $w_{I\overline{J}} = 0$, condition (2.4.1) becomes either $c_J \geq r_I + e_{\overline{I}J} \geq 0 + c_J$ or $h = c_J \geq r_I + e_{\overline{I}J} \geq r_I + r_{\overline{I}} = h$, since we assumed that $e_{i+} \leq r_i$ , for all $i \leq k$ and $e_{+j} \leq c_j$ , for all $j \leq l$.

[9] See Balinski and Demange [5] on page 712.

**Proposition 2.4.1** *Let $(W, \sigma)$ be a problem and the underlying multiplier method be impervious. If $W$ is strictly positive, then $A^s(W, \sigma)$ is nonempty.*

Instead of verifying condition (2.3.1) and (2.4.1), we take an alternative graph theoretic approach. The strategy is as follows: The given problem $(W, \sigma)$ is transformed into a network due to its underlying multiplier method. After that, a maximum flow problem has to be solved to establish existence. Thus, the following two chapters provide graph theoretic basics, which are needed for further considerations.

# Chapter 3

# Flows and cuts in networks

Section 1 is concerned with the introduction of basic notations and definitions of network theory. The concept of maximum flow problems is discussed in detail. In section 2 the fundamental max-flow min-cut theorem is established and the connections to primal and dual problem drawn. The last section provides information about the propoerties of cuts.

## 3.1 Maximum flow problem

Let $G = (V, A)$ be a network, consisting of a set $V$ of $n$ nodes and a set $A$ of $m$ arcs, with $m, n \in \mathbb{N}$. Each arc $(i, j)$ in the network is directed and thus carrying a specific orientation. Flow on arc $(i, j)$ can only be sent from node $i$ to node $j$. Furthermore, the nonnegative flow on each arc $(i, j)$ is restricted by some finite arc capacity $u_{ij}$. Henceforth, assume that $u_{ij}$ is integer valued. Two nodes $s$ and $t \in V$ are denoted as source and sink, respectively. Without loss of generality, no parallel arcs are admitted. Figure 3.1 presents the directed and capacitated network $G = (V, A)$ with $V = \{s, 1, 2, 3, 4, 5, t\}$, $A = \{(s, 1), (s, 2), (s, 3), (1, 4), (1, 5), (2, 5), (3, 5), (4, t), (5, t)\}$, and the capacities $u = (u_{s,1}, u_{s,2}, u_{s,3}, \ldots, u_{5,t}) = (3, 3, 1, 7, 7, 7, 7, 4, 3)$. All capacities are written above their affiliated arcs. This network will be used for further examples in this work.

The **maximum flow problem** seeks for the maximal flow from the source node $s$ to the sink node $t$ through the given network $G$. Any flow $x = (x_{ij})$, with $(i, j) \in A$, must satisfy both the flow bound and the mass balance constraint. That is:

1. $0 \leq x_{ij} \leq u_{ij}, \quad \forall\ (i, j) \in A$ (flow bound constraint)
2. $\sum_{(i,j)\in\delta^+(i)} x_{ij} - \sum_{(j,i)\in\delta^-(i)} x_{ji} = 0, \quad \forall\ i \in V \setminus \{s, t\}$ (mass balance constraint)

where $\delta^+(i)$ denotes the set of arcs emanating from node $i$. Similarly, $\delta^-(i)$ is

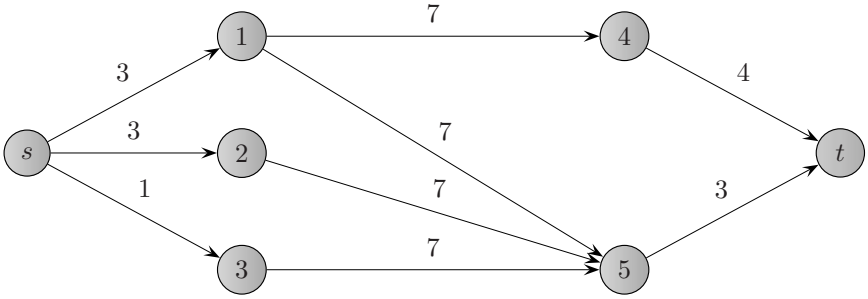


**Figure 3.1:**

Directed and capacitated network $G$.

the set of all arcs ending in node $i$[1]. The flow value $v(x)$ is given by

$$v(x) = \sum_{(i,j)\in\delta^+(s)} x_{ij} - \sum_{(i,j)\in\delta^-(s)} x_{ij}.$$

The **residual network** is a central point within most algorithm to solve maximum flow problems. It represents the network with its residual capacities, given a feasible flow $x$. Since many maximum flow algorithms establish the solution, if possible, by increasing the flow value incrementally, the residual network is favored at intermediate steps.
The following considerations about residual networks are based on the work by Ahuja, Magnanti and Orlin[2]. Given a feasible flow $x = (x_{ij})$ in the original network $G$, each arc $(i, j)$ is carrying $x_{ij}$ units of flow. Further flow on the arc $(i, j)$ is restricted by the residual capacity $u_{ij}$ - $x_{ij}$. On the other hand, given the flow $x_{ij}$ on arc $(i, j)$, it can be decreased, or in other words, can be sent back from node $j$ to node $i$ along arc $(i, j)$. Based on this observation the residual network $G(x)$, given the feasible flow $x$, is defined as follows.
For any arc $(i, j)$ in the original network with positive flow, $x_{ij} > 0$, insert in the residual network $G(x)$ two arcs $(i, j)$ and $(j, i)$, having capacity $u_{ij} - x_{ij}$ and $x_{ij}$, respectively. Observe that only arcs with nonnegative capacity appear, since the flow $x$ was assumed to be feasible, that is, nonnegative and satisfying the flow bound constraints. Ambiguity could arise by this construction as to parallel arcs. Without loss of generality, those two arcs are to be merged into one, while adding their capacities. The updated capacities in the residual network are denoted by $r$ as for 'residual'.
To see that there is a one-to-one correspondence between flows in the original network $G$ and flows in the residual network $G(x^*)$, let $x$ be a feasible flow in $G$. It has to be shown that $x$ corresponds to a feasible flow $x'$ in the residual network $G(x^*)$. Thus, define $x' \geq 0$ as

[1] See Borgwarth [8] on page 411.
[2] See [1] on page 44 to 46.

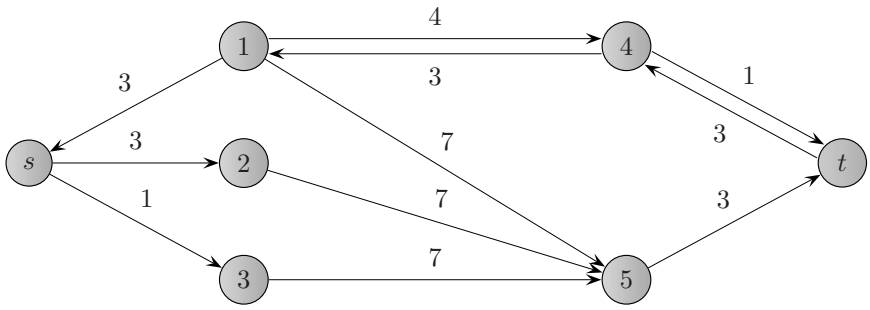


**Figure 3.2:**

The residual network $G(x^*)$, with $G$ as in figure 3.1 and $x^* = (3, 0, 0, 3, 0, 0, 0, 3, 0)$.

$$\begin{aligned} x'_{ij} - x'_{ji} &= x_{ij} - x^*_{ij} \qquad \text{and} \\ x'_{ij} x'_{ji} &= 0. \end{aligned}$$

This is,
$x'_{ij} = x_{ij} - x^*_{ij}$ and $x'_{ji} = 0$, if $x_{ij} \geq x^*_{ij}$,
$x'_{ji} = x^*_{ij} - x_{ij}$ and $x'_{ij} = 0$, otherwise.
In both cases the nonnegative flow $x'$ satisfies the flow bound constraint of the residual network $G(x^*)$. Since $x$ is a feasible flow in $G$, $0 \leq x_{ij} \leq u_{ij}$:
$0 \leq x'_{ij} \leq u_{ij} - x^*_{ij} = r_{ij}$, if $x_{ij} \geq x^*_{ij}$
$0 \leq x'_{ji} \leq x^*_{ij} = r_{ji}$, otherwise.
Similarly, any feasible flow $x'$ of the residual network $G(x^*)$ corresponds to a feasible flow $x$ given by $x_{ij} = (x'_{ij} - x'_{ji}) + x^*_{ij}$ in the original network $G$. Due to the feasibility of $x'$, we have $x'_{ji} \leq x^*_{ij}$ and $x'_{ij} \leq u_{ij} - x^*_{ij}$. Thus,
$0 \leq x_{ij} = x^*_{ij} - x'_{ji} \leq x^*_{ij} \leq u_{ij}$, if $x'_{ij} = 0$ and
$0 \leq x_{ij} = x'_{ij} + x^*_{ij} \leq u_{ij} - x^*_{ij} + x^*_{ij} = u_{ij}$, if $x'_{ji} = 0$.
This established the following theorem.

**Theorem 3.1.1** *A flow $x$ is a feasible flow in the network $G$ if and only if its corresponding flow $x'$, defined by $x'_{ij} - x'_{ji} = x_{ij} - x^*_{ij}$ and $x'_{ij} x'_{ji} = 0$, is feasible in the residual network $G(x^*)$.*

Figure 3.2 shows the residual network $G(x^*)$, given the network $G$ of figure 3.1 and the feasible flow $x^* = (x^*_{s,1}, x^*_{s,2}, x^*_{s,3}, \ldots, x^*_{5,t}) = (3, 0, 0, 3, 0, 0, 0, 3, 0)$ with flow of three units on the arcs $(s, 1)$, $(1, 4)$ and $(4, t)$. The capacities of the above mentioned arcs are reduced by their flow value of three units. At the same time, arcs with opposite orientation $(1, s)$, $(4, 1)$ and $(t, 4)$ are inserted, carrying the capacity of the previously withdrawn flow value. Notice that arc $(s, 1)$ vanishes due to its new upper capacity of zero.

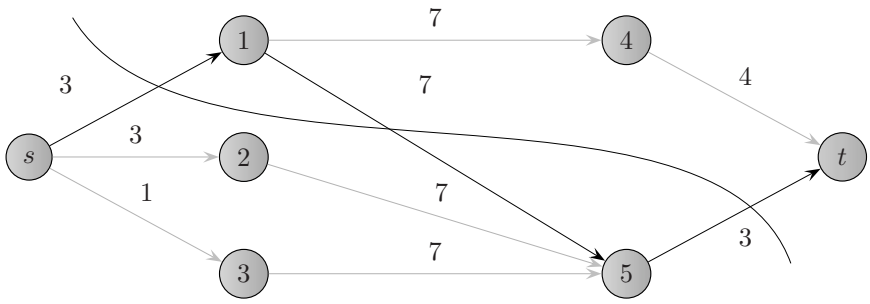


**Figure 3.3:**

Cut $\delta(S) = \{(s,1),(1,5),(5,t)\}$, consisting of the sets $\delta^+(S) = \{(s,1),(5,t)\}$ and $\delta^-(S) = \{(1,5)\}$ for $S = \{s,2,3,5\}$ .

Maximum flows are tightly connected with a special class of cuts. The next section provides some basics of the concept of cuts and establishes the fundamental max-flow min-cut theorem.

## 3.2 Max-flow min-cut theorem

If we turn back to the existence problem, this section is important in consideration of determining a set of parties and a set of districts, violating the relevant conditions in the case of nonexistence of a solution.

**Definition 3.2.1** *Let $S$ be a subset of $V$ in the network $G = (V, A)$ and $\overline{S}$ its complement. The set of arcs between $S$ and $\overline{S}$ is called* **cut** *and denoted by $\delta(S)$. It contains the set of arcs leading out of $S$, $\delta^+(S) := \{(i,j) \in A \mid i \in S, j \in \overline{S}\}$, and the set of arcs leading into $S$, $\delta^-(S) := \{(i,j) \in A \mid j \in S, i \in \overline{S}\}$.*

See in figure 3.3 an example of the cut $\delta(S)$ in the network $G$ of figure 3.1, with $S = \{s,2,3,5\}$.

**Definition 3.2.2** *A cut $\delta(S)$ is called* **s-t separating cut** *if the source node $s$ is contained in $S$ and the sink node $t$ is an element of $\overline{S}$.*

The cut of figure 3.3 illustrates an $s$-$t$ separating cut since $s \in S = \{s,2,3,5\}$ and $t \in \overline{S} = \{1,4,t\}$.

**Definition 3.2.3** *The* **cut capacity** *$u(S)$ of an $s$-$t$ separating cut $\delta(S)$ is defined as the sum over all upper capacities of arcs in $\delta^+(S)$,*

$$u(S) := \sum_{(i,j)\in\delta^+(S)} u_{ij}.$$

In the case of the example in figure 3.3, the cut capacity of $\delta(S)$ equals $u_{s,1} + u_{5,t} = 3 + 3 = 6$.

**Definition 3.2.4** *An s-t separating cut with minimum cut capacity among all s-t cuts is called* **minimum cut**.

Figure 3.3 presents the minimum cut $\delta(S)$, since for any other set $S' \subseteq V \setminus \{t\}$, $u(S) \leq u(S')$.

The interplay between flow values and cut capacities states the following theorem.

**Theorem 3.2.1** *For any flow $x$ in a capacitated network from the source node $s$ to the sink node $t$ and any s-t separating cut $\delta(S)$ the following holds:*

$$v(x) \leq u(S).$$

**Proof.** Let $x$ be a feasible flow from $s$ to $t$ with flow value $v(x)$ and $\delta(S)$ an $s$-$t$ separating cut, with $S \subset V$. Flow $x$ satisfies the following equations:

$$\begin{aligned} &\textstyle\sum_{(i,j)\in\delta^+(s)} x_{ij} - \sum_{(i,j)\in\delta^-(s)} x_{ij} = v(x) \\ &\textstyle\sum_{(i,j)\in\delta^+(v)} x_{ij} - \sum_{(i,j)\in\delta^-(v)} x_{ij} = 0, \qquad \forall\ v \in V \setminus \{s,t\} \\ &\textstyle\sum_{(i,j)\in\delta^+(t)} x_{ij} - \sum_{(i,j)\in\delta^-(t)} x_{ij} = -v(x) \end{aligned}$$

Sum over all $n \in S$ to get:

$$v(x) = \sum_{(i,j)\in\delta^+(s)} x_{ij} - \sum_{(i,j)\in\delta^-(s)} x_{ij} = \sum_{n\in S} \Big( \sum_{(i,j)\in\delta^+(n)} x_{ij} - \sum_{(i,j)\in\delta^-(n)} x_{ij} \Big)$$

Since any arc $(i,j)$ with $i,j \in S$ occurs twice in this equation, once negative and once positive, it can be reduced to summing up over all arcs $(i,j)$ with either $i \in S$ and $j \in \overline{S}$ or $i \in \overline{S}$ and $j \in S$. Recall that this is the set of arcs in $\delta^+(S)$ and $\delta^-(S)$. In addition, $x_{ij} \leq u_{ij}$, since the flow $x$ was assumed to be feasible. Thus,

$$v(x) = \sum_{(i,j)\in\delta^+(S)} x_{ij} - \sum_{(i,j)\in\delta^-(S)} x_{ij} \leq \sum_{(i,j)\in\delta^+(S)} u_{ij} - \sum_{(i,j)\in\delta^-(S)} u_{ij} \leq u(S).^3$$

A fundamental result in network theory yields that even equality holds for the maximum flow value and the minimum cut capacity.

**Theorem 3.2.2** (Max-flow min-cut theorem) *For any capacitated network the maximum flow value from the source node $s$ to the sink node $t$ is equal to the minimum capacity of all s-t separating cuts.*

[3] See Ford and Fulkerson [9] on page 10 and 11.

**Proof.** With the result of theorem 3.2.1, it suffices to establish the existence of a flow $x$ and an $s$-$t$ separating cut $\delta(S)$, such that $v(x) = u(S)$. A maximum flow exists, since the set of all feasible flows is compact, the capacities are bounded and there is at least one feasible flow, the null flow.
Let $x$ be the maximum flow with

$v(x) = \sum_{(i,j)\in\delta^+(s)} x_{ij} - \sum_{(i,j)\in\delta^-(s)} x_{ij}$

and define the set $S$, based on the flow $x$, recursively as follows:

1. $s \in S$
2. if $i \in S$ and $x_{ij} < u_{ij}$, then add $j$ to $S$,
   if $i \in S$ and $x_{ji} > 0$, then add $j$ to $S$.

Observe, that node $t$ must be in $\overline{S}$, since $x$ is defined maximum. Assume $t$ is contained in $S$, we can find a directed path $P$ from $s$ to $t$, such that for any forward arc $(i,j)$ on path $P$, the flow value does not exhaust the upper capacity, $x_{ij} < u_{ij}$, and for any backward arc $(i,j)$ on $P$, the flow value is positive, $x_{ij} > 0$. Take $\epsilon := \min\{r_{ij} : (i,j) \in P\}$ to be the residual capacity of path $P$. $\epsilon$ is positive, so that $x$ cannot be a maximum flow. Consequently, $t \in \overline{S}$ and $\delta(S)$ defines an $s$-$t$ separating cut. By construction, it follows, that every arc $(i,j)$, leaving $S$, is saturated, whereas arcs $(i,j)$, leading into $S$, do not carry any flow:

$\sum_{(i,j)\in\delta^+(S)} x_{ij} = u(S)$ and
$\sum_{(i,j)\in\delta^-(S)} x_{ij} = 0$

Now we have established the existence of a flow $x$ and an $s$-$t$ separating cut $\delta(S)$, for which equality of the flow value and the cut capacity hold. With the fact, that the flow value is bounded from above by any cut capacity, the proof is finished.[4]

The max-flow min-cut theorem treats a special case of the duality theorem[5], which states the interplay between the solutions of the primal maximization and its associated dual minimization problem or vice versa.
Since the maximum flow problem is about the maximization of a linear function under given linear constraints, we formulate it as linear programming problem.

(LP) maximize $v(x) = \sum_{(i,j)\in\delta^+(s)} x_{ij} - \sum_{(i,j)\in\delta^-(s)} x_{ij}$ under the linear constraints,

$\sum_{(i,j)\in\delta^+(i)} x_{ij} - \sum_{(j,i)\in\delta^-(i)} x_{ji}, \qquad \forall\, i \in V \setminus \{s,t\}$
$0 \le x_{ij} \le u_{ij}, \qquad \forall\, (i,j) \in A$

Herewith we can associate its dual problem, which can be generated by transforming the primal one. For the transformation rules see Borgwarth[6]. We get the dual problem:

[4] See Ford and Fulkerson [9] on page 11 and 12.
[5] For details see Borgwarth [8] on page 69.
[6] See [8] on page 459.

(DP) minimize $\sum_{(i,j)\in A} u_{ij}y_{ij}$ under the linear constraints,

$$y_{ij} \geq 0, \qquad \forall\ (i,j) \in A$$
$$y_{ij} + z_i - z_j \geq 0, \qquad \forall\ (i,j) \in A$$
$$z_s = -1$$
$$z_t = 0$$

The maximum flow and thus the linear programming problem always has an optimal solution. Due to the duality theorem the dual problem has an optimal solution, where the minimum value is equal to the maximum value of the primal one. This is exactly what the max-flow min-cut theorem provides, with the dual problem corresponding to the minimum cut problem. Indeed, the optimal solution $\begin{pmatrix} y \\ z \end{pmatrix}$ of the dual problem can be converted into a minimum *s-t* separating cut $\delta(S)$ with cut capacity $u(S) = \sum_{(i,j)\in A} u_{ij}y_{ij}$.[7]

## 3.3 Properties of cuts

For the following considerations see Ford and Fulkerson[8].

**Theorem 3.3.1** *Let $\delta(S')$ be a minimum cut in the network $G$ and let $\delta(S)$ be the minimum cut, based on the maximum flow $x$, as defined in the proof of theorem 3.2.2. Then $S \subseteq S'$.*

To proof theorem 3.3.1 we need the following corollaries.

The minimality of cuts can also be formulated regarding selected sets of arcs and the realized flow upon.

**Corollary 3.3.1** *A cut $\delta(S)$ is minimum if and only if every maximum flow $x$ saturates all arcs of $\delta^+(S)$ and has zero flow on all arcs of $\delta^-(S)$.*

**Proof.** Let $\delta(S)$ be a minimum cut and assume the flow $x$ is maximum, such that $v(x) = u(S)$. If there is any arc $(i,j) \in \delta^+(S)$, with $x_{ij} < u_{ij}$, or $(i,j) \in \delta^-(S)$, with $x_{ij} > 0$, then $v(x) < u(S)$. Thus, $x$ cannot be maximum due to theorem 3.2.2 on page 17. On the other hand, take $S \subset V$, such that $\delta(S)$ defines an *s-t* separating cut. If $x$ is a maximum flow, with $x_{ij} = u_{ij}$ for all arcs $(i,j) \in \delta^+(S)$ and $x_{ij} = 0$ for all arcs $(i,j) \in \delta^-(S)$, then $v(x) = u(S)$ and thus $\delta(S)$ must be a minimum cut.

**Corollary 3.3.2** *Let $\delta(S)$ and $\delta(C)$ be minimum cuts. Then $\delta(S \cap C)$ is also a minimum cut.*

[7] See Ford and Fulkerson [9] on page 26 to 30 and Borgwarth [8] on page 458 to 460.
[8] See [9] on page 13 and 14.

**Proof.** We have to show that for every maximum flow $x$ any arc emanating $\delta(S \cap C)$ is saturated and any arc leading into it has zero flow. Take arc $(i,j) \in \delta^+(S \cap C)$, such that $i \in (S \cap C)$ and $j \in (\overline{S} \cup \overline{C})$. We find, $i \in S$ and $i \in C$ and $j \in \overline{S}$ and/or $j \in \overline{C}$. If $j \in \overline{S}$, then $x_{ij} = u_{ij}$, since $\delta(S)$ was defined to be minimum. If $j \in \overline{C}$, then $x_{ij} = u_{ij}$, since $\delta(C)$ was defined to be minimum. Use the same argumentation to show that any arc leading out of $(\overline{S} \cup \overline{C})$ has zero flow.

**Proof of theorem 3.3.1.** Assume that $S \not\subseteq S'$, then there is some $i \in S$, with $i \notin (S \cap S')$. Notice that $\delta(S \cap S')$ defines a minimum cut. Since $i \neq s$, we can find a path $P$ from $s$ to $i$, such that, by definition, every forward arc on $P$ is not saturated and every backward arc on $P$ has positive flow. We define arc $(i,j) \in P$ as forward arc, if it is used via its true orientation, otherwise it is defined as backward arc. Since $s \in (S \cap S')$ and $i \in (\overline{S \cap S'})$, there are two following nodes $i_k \in (S \cap S')$ and $i_{k+1} \in (\overline{S \cap S'})$ on $P$. If $(i_k, i_{k+1})$ is emanating $(S \cap S')$, then flow on this arc is less than its upper capacity, contradicting corollary 3.3.1 on page 19. If $(i_k, i_{k+1})$ is leading into $(S \cap S')$, we have positive flow, again contradicting corollary 3.3.1. Thus, $S \subset S'$. This proofs theorem 3.3.1 on page 19.

Let $x$ be a maximum flow. Similar to the defined set $S$ as in the proof of theorem 3.2.2 on page 18, define the set $\overline{S'}$, and thus the minimum cut $\delta(S')$, based on $x$ as follows:

1. $t \in \overline{S'}$
2. if $j \in \overline{S'}$ and $x_{ij} < u_{ij}$, then add $i$ to $\overline{S'}$,
   if $j \in \overline{S'}$ and $x_{ji} > 0$, then add $i$ to $\overline{S'}$.

**Theorem 3.3.2** *Let $S$ be defined as in the proof of theorem 3.2.2 and $S'$ as the complement of the previously defined set $\overline{S'}$. If the upper capacities are strictly positive, then the minimum cut $\delta(S)$ is unique if and only if $\delta(S) = \delta(S')$.*

**Proof.** Let $\delta(C)$ be a minimum cut. We have to show that, if $\delta(S) = \delta(S')$, then $\delta(S) = \delta(C)$. If $\delta(S) = \delta(S')$, then both equal the set $A_{S,S'} := \{(i,j) : i \in S \text{ and } j \in \overline{S'}\}$, since, due to theorem 3.3.1 on page 19, $S \subseteq S'$ and thus $A_{S,S'} \subseteq \delta(S')$ and for any arc $(i,j) \in \delta(S) = \delta(S')$, $i \in S$ and $j \in S'$, such that $(i,j) \in A_{S,S'}$.
For $\delta(C)$, we get by theorem 3.3.1 on page 19, $S \subseteq C$ and $\overline{S'} \subseteq \overline{C}$. Thus, $\delta(S) = A_{S,S'} \subseteq \{(i,j) : i \in C \text{ and } j \in \overline{S'}\} \subseteq \{(i,j) : i \in C \text{ and } j \in \overline{C}\} = \delta(C)$ and therefore $u(S) \leq u(C)$. If $\delta(S)$ is a real subset of $\delta(C)$, then either some arcs in $\delta(C)$ have zero capacity, contradicting our assumption of strictly positive upper capacity bounds, or $u(S) < u(C)$, contradicting the assumption of the minimality of $\delta(C)$. Thus, $\delta(S) = \delta(C)$.

# Chapter 4

# Solving maximum flow problems

Section 1 presents the basic idea of the generic augmenting path algorithm to solve maximum flow problems. Section 2 describes the Labelling algorithm, a special implementation of the generic augmenting path algorithm, followed by the pseudo-code, the proof of correctness and the complexity, in section 3 to 5.[1] Section 6 provides an example.

## 4.1 Generic augmenting path algorithm

A methodology to solve maximum flow problems is provided by the generic augmenting path algorithm. Throughout the algorithm the concept of directed paths from the source node $s$ to the sink node $t$ in the residual network $G(x)$, the so-called augmenting paths, is of great importance. $P$ is called an **augmenting path** in the residual network $G(x)$, for a feasible flow $x$, if for all arcs $(i, j) \in P$ the residual capacity $r_{ij}$ is positive. The **residual capacity** $r(P)$ of an augmenting path $P$ is defined as the minimal residual capacity $r_{ij}$ of any arc $(i, j)$ on the path $P$

$$r(P) = \min\{r_{ij} : (i, j) \in P\}.$$

For an example of an augmenting path and its residual capacity in the network $G$ of the previous chapter see figure 4.1.
Notice that the residual capacity $r(P)$ is always positive. On every augmenting path $P$ further flow of $r(P)$ units can be sent from the source to the sink, so that the absolute flow value increases by $r(P)$ units. Based on this observation, the algorithm can be partitioned into two subroutines, which are executed alternately:

1. identifying an augmenting path $P$ from $s$ to $t$ and
2. sending flow along $P$ and updating the residual network (augmentation).

[1] For section 1 to 5 see Ahuja, Magnanti and Orlin [1] on page 180 to 186.

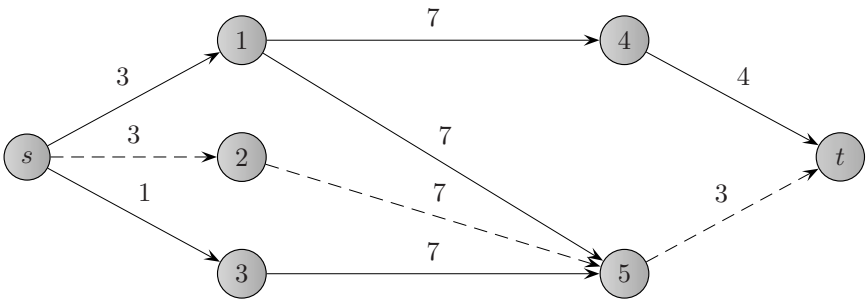


**Figure 4.1:**

The augmenting path $P : s \rightarrow 2 \rightarrow 5 \rightarrow t$ in the network $G$ . The residual capacity $r(P) = min\{r_{s,2}, r_{2,5}, r_{5,t}\} = 3$.

The algorithm terminates, if no further augmenting path is contained in the residual network and the actual flow value is returned. However, it does not specify how to identify such a path. A special implementation of the generic augmenting path algorithm is given by the Labelling algorithm and is described in detail below.

## 4.2 Labelling algorithm

Given a network $G = (V, A)$ with two nodes $s$, $t \in V$, defined as source and sink, the algorithm uses a labelling process to identify any node in the residual network, that can be reached from the source along a directed path. Starting at the source node $s$, the algorithm labels all direct reachable neighbor nodes in the residual network. This scanning process is done for every labelled node, until either the sink node is labelled, too, or all labelled nodes are scanned and the sink remains unlabelled. Is $t$ labelled, an augmenting path was found. After augmentation along that path, the labelling process starts again with unlabelling all nodes. Otherwise the sink is not connected to the source and no such path exists. The algorithm terminates and returns the actual flow, which is maximum.

## 4.3 Pseudo-code

**algorithm** Labelling
  **begin**
    label node $t$
    **while** $t$ is labelled **do**
    **begin**
      unlabel all nodes;
      set pred($i$) := 0 $\forall\ i \in V$;

```
        label s and set LIST := {s};
        while LIST≠ ∅ and t unlabelled do
        begin
            remove a node i from LIST;
            for each arc (i, j) in the residual
                network emanating node i do
            begin
                if node j is unlabelled
                then
                    set pred(j) := i;
                    label j and add j to LIST;
            end;
        end;
        if t is labelled then
            subroutine augment;
    end;
end;

subroutine augment
    begin
        use the predecessor pred to trace back from the sink to the source
        to obtain an augmenting path P from node s to t;
        δ := min{r_ij : (i, j) ∈ P};
        augment δ units of flow along P and update the residual
        capacities;
    end;
```

## 4.4 Correctness

To establish correctness, we have to show that the algorithm terminates within finite steps and that the returned flow value at the end is maximum.

**Theorem 4.4.1** (Augmenting path theorem) *A flow $x^*$ is a maximum flow in $G(x^*)$ if and only if the residual network $G(x^*)$ contains no augmenting path.*

**Proof.** If the residual network $G(x^*)$ contains an augmenting path, the flow value can be increased by the residual capacity $r(P)$ of path $P$ and $x^*$ cannot be maximum. Conversely, if the residual network contains no augmenting path, two sets $S$ and $\overline{S}$ exist (actually they are determined by the algorithm), which define an $s$-$t$ cut $\delta(S)$, whose capacity equals the flow value of flow $x^*$. Since the flow value of any flow is smaller than or equal to the capacity of any $s$-$t$ cut, $x^*$ must be maximum.

Since the algorithm terminates if no augmenting path can be found, the actual flow must be maximum, according to the augmenting path theorem.

To establish termination within finite steps, recall that all arc capacities were assumed to be integer and finite. Therefore any residual capacity of an augmenting path is larger than or equal to one and each augmentation will increase the absolute flow value by at least one. In each iteration the algorithm augments along a directed path or in the last one, terminates if no such path was found. Thus, the finite maximum flow value can be reached within a finite number of iterations.

## 4.5 Complexity

**Theorem 4.5.1** *The Labelling algorithm solves the maximum flow problem within $O(nmU)$ time, where $U$ denotes the maximum value among all arc capacities of the original network.*

**Proof.** Recall that the network contains $n$ nodes and $m$ arcs. The Labelling algorithm performs at each iteration either an augmentation or terminates, if the source and sink are disconnected in the residual network $G(x)$. However, each iteration starts with the search of a directed path from $s$ to $t$. In the worst case every arc in $G(x)$ has to be checked, so that in the end the search takes $O(m)$ time. Similarly, augmentation involves at most every arc in the network, leading to the worst case complexity of $O(m)$. Assuming that the integer valued capacities are bounded from above by some finite value $U$, the maximum flow value is at most $nU$. Since any augmentation carries at least one unit, the worst case bound on the number of iterations is $O(nU)$. This leads to the overall worst case complexity of $O(nmU)$.

## 4.6 Example

We take network $G$ of the previous chapter and establish a maximum flow by applying the Labelling algorithm. See figure 4.2 for the input network.

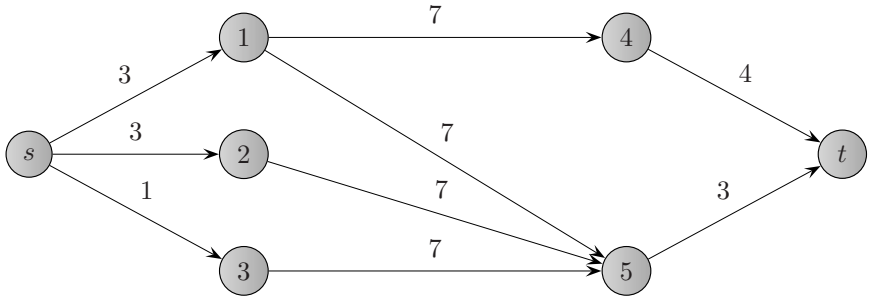


**Figure 4.2:**

Network $G$, for which the maximum flow is determined.

**1.Iteration:**

pred$(i) = 0 \quad \forall\, i \in V$
Labelled= $\{s\}$;
List=$\{s\}$;
**take node s:**
List= $\emptyset$;
**take arc (s,1):**
pred$(1) = s$;
Labelled=$\{$s,1$\}$;
List= $\{1\}$;
**take arc (s,2):**
pred$(2) = s$;
Labelled= $\{s, 1, 2\}$;
List= $\{1, 2\}$;
**take arc (s,3):**
pred$(3) = s$;
Labelled=$\{$s,1,2,3$\}$;
List= $\{1, 2, 3\}$;
**take node 1:**
List= $\{2, 3\}$;
**take arc (1,4):**
pred$(4) = 1$;
Labelled= $\{s, 1, 2, 3, 4\}$;
List= $\{2, 3, 4\}$;
**take arc (1,5):**
pred$(5) = 1$;
Labelled= $\{s, 1, 2, 3, 4, 5\}$;
List= $\{2, 3, 4, 5\}$;
**take node 2:**
List= $\{3, 4, 5\}$;
**take arc (2,5):** –, since node 5 is labelled
**take node 3:**
List= $\{4, 5\}$;
**take arc (3,5):** –, since node 5 is labelled
**take node 4:**
List= $\{5\}$;
**take arc (4,t):**
pred$(t) = 4$;
Labelled= $\{s, 1, 2, 3, 4, 5, t\}$;
List= $\{5, t\}$;

node t is labelled:

**augment:**

$P \;:\; t \to 4 \to 1 \to s$;
$\delta(P) = \min\{3, 7, 4\} = 3$;

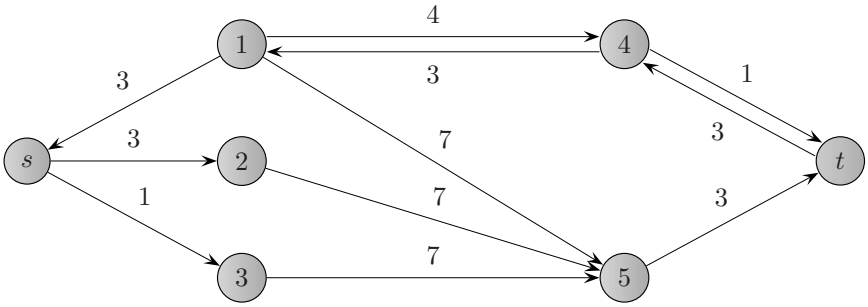


**Figure 4.3:**

Residual network $G(x)$, with $x = (3, 0, 0, 3, 0, 0, 0, 3, 0)$.

realize augmentation of 3 units along $P$ to get the residual network in figure 4.3

**2.Iteration:**

pred$(i) = 0 \quad \forall\, i \in V$
Labelled= $\{s\}$;
List= $\{s\}$;
**take node** $s$**:**
List= $\emptyset$;
**take arc** $(s, 2)$**:**
pred$(2) = s$;
Labelled= $\{s, 2\}$;
List= $\{2\}$;
**take arc** $(s, 3)$**:**
pred$(3) = s$;
Labelled= $\{s, 2, 3\}$;
List= $\{2, 3\}$;
**take node 2:**
List= $\{3\}$;
**take arc** $(2, 5)$**:**
pred$(5) = 2$;
Labelled= $\{s, 2, 3, 5\}$;
List= $\{3, 5\}$;
**take node 3:**
List= $\{5\}$;
**take arc (3,5):**–, since node 5 is labelled;
**take node 5:**
List= $\emptyset$;
**take arc (5,**$t$**):**
pred$(t)$=5;
Labelled=$\{s, 2, 3, 5, t\}$;

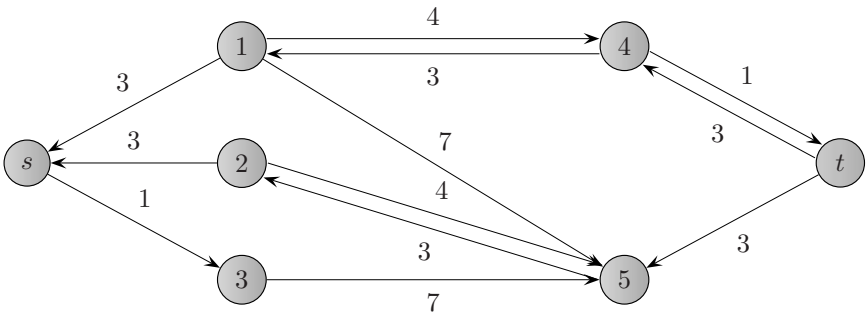


**Figure 4.4:**

Residual network $G(x)$, with $x = (3, 3, 0, 3, 0, 3, 0, 3, 3)$.

List=$\{t\}$;

node $t$ is labelled:

**augment:**

$P = t \to 5 \to 2 \to s$;

$\delta(P) = \min\{3, 7, 3\} = 3$;

realize augmentation of 3 units along $P$ to get the residual network in figure 4.4

**3.Iteration:**

pred$(i) = 0 \quad \forall\, i \in V$

Labelled= $\{s\}$;

List= $\{s\}$;

**take node $s$:**

List=$\emptyset$;

**take arc** $(s, 3)$**:**

pred$(3) = s$;

Labelled=$\{s$,3$\}$;

List=$\{3\}$;

**take node 3:** –

List= $\emptyset$;

**take arc (3,5):**

pred$(5)$=3;

Labelled=$\{s$,3,5$\}$;

List=$\{5\}$;

**take node 5:** –

List= $\emptyset$;

**take arc (5,2):**

pred$(2)$=5;

Labelled=$\{s$,3,5,2$\}$;

List=$\{2\}$;

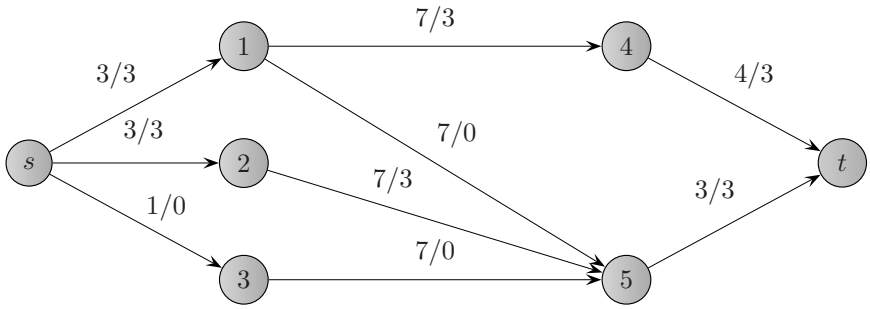


**Figure 4.5:**

Network $G$, with the determined maximum flow $x = (3, 3, 0, 3, 0, 3, 0, 3, 3)$ and flow value $v(x) = 6$.

**take node 2:**

List= $\emptyset$;

**take arc (2,$s$):**–, since node s is labelled

List is empty and node $t$ is unlabelled:
The algorithm terminates with the maximum flow
$x = (3, 3, 0, 3, 0, 3, 0, 3, 3)$ and flow value $v(x) = 6$.
See figure 4.5 for the realized flow, where the tuple y/z
above the arcs corresponds to upper capacity/flow value.

# Chapter 5

# Establishing existence by means of network theory

To establish existence one might try to verify condition (2.3.1) or (2.4.1) on page 10 and 11. This becomes inefficient, especially for big problems. As an alternative approach to a solution of the existence problem, the biproportional apportionment problem can be transformed into a capacitated network, which is presented in section 1. Section 2 establishes the equivalence of feasibility of a circulation within the constructed network to the existence of a feasible apportionment and illustrates how to solve the feasibility problem. Section 3 derives a simple and efficient strategy to gain a statement about the existence of a solution, using the results of section 2. In section 4 the complexity of the procedure is determined.

## 5.1 Transformation of the problem into a network

Let $W$ denote the weight matrix and $r$ and $c$ the marginal constraint vectors for the districts and parties. For a transformation due to Balinski and Demange[1], introduce a node for each party and each district and insert the directed arc $(i, j)$, starting in district node $i$ and ending in party-node $j$, if and only if the weight for party $j$ in district $i$ is strictly positive, that is, $w_{ij} > 0$. Define the upper capacity bound $u_{ij}$ of arc $(i, j)$ to be $h$, the house size, and set the lower capacity $l_{ij} = 0$, if $s(0) > 0$, and $l_{ij} = 1$, if $s(0) = 0$. The lower capacity reflects the quality of the given multiplier method with its associated signpost sequence $s$. If it is impervious, $s(0) = 0$, then each party with positive weight in some district $j$ is guaranteed at least one seat there.
Continue to add a source node $s$ and arcs $(s, i)$ emanating from node $s$ to each district node $i = 1, \ldots, k$ with lower and upper capacity equal to $r_i$. Additionally, adjunct a sink node $t$ and arcs $(j, t)$ from each party node $j = 1, \ldots, l$ to

[1] See [5] on page 707 and 713.

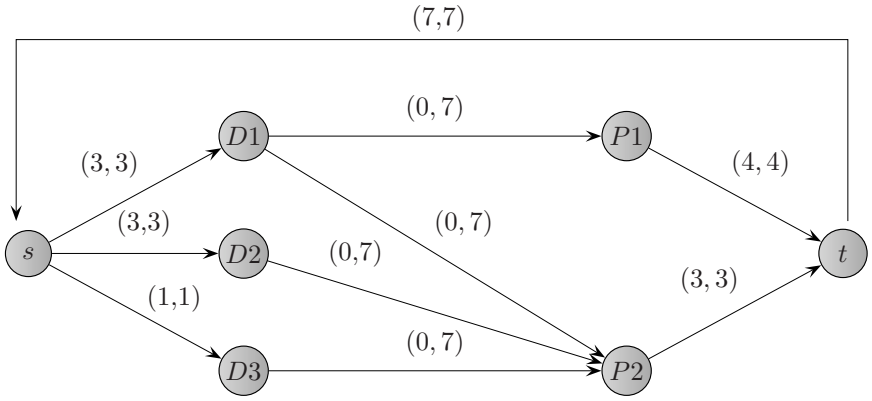


**Figure 5.1:**

Generated network for the problem $(W^*, \sigma^*)$ with three districts, two parties and an underlying pervious multiplier method.

the sink $t$ with lower and upper capacity $c_j$. In the end, insert arc $(t, s)$ with lower and upper capacity equal to the house size $h$.
See in figure 5.1 an example of this transformation according to the problem $(W^*, \sigma^*)$, with $W^* = \begin{pmatrix} \times & \times \\ 0 & \times \\ 0 & \times \end{pmatrix}$, where $\times$ denotes some positive number, $\sigma^* = ((3, 3, 1), (4, 3))$ and house size $h = 7$, for an underlying pervious multiplier method. The tuple $(y, z)$ above any arc is to interpret as (lower capacity bound, upper capacity bound). In the case of $s(0) = 0$, the lower capacity bounds on arcs between district and party nodes are set to 1.

Flow on arcs between the source and a district node $i$ or a party node $j$ and the sink corresponds to the total number of seats, allotted to district $i$ or party $j$, respectively. Flow on arcs between district node $i$ and party node $j$ represent the number of allotted seats to party $j$ in district $i$. The feasibility of a circulation in the above constructed networks is equivalent to the nonemptyness of the sets $R^0(W, \sigma)$ and $R^1(W, \sigma)$, respectively, since a feasible circulation represents an element of the relevant set. As we will see in the next section, condition (2.3.1) and (2.3.2) correspond to the necessary and sufficient condition for a feasible circulation to exist. Moreover we can derive another existence criterion, which can be checked more efficiently.

## 5.2 Feasibility of circulation

Let the given directed and capacitated network $G = (V, A)$ be originated from a problem $(W, \sigma)$ and its underlying multiplier method. Let the lower and upper capacity bound of arc $(i, j)$ be denoted by $l_{ij}$ and $u_{ij}$, respectively, where $0 \leq l_{ij} \leq u_{ij}$. A feasible circulation $x$ in $G$ satisfies both the flow bound and

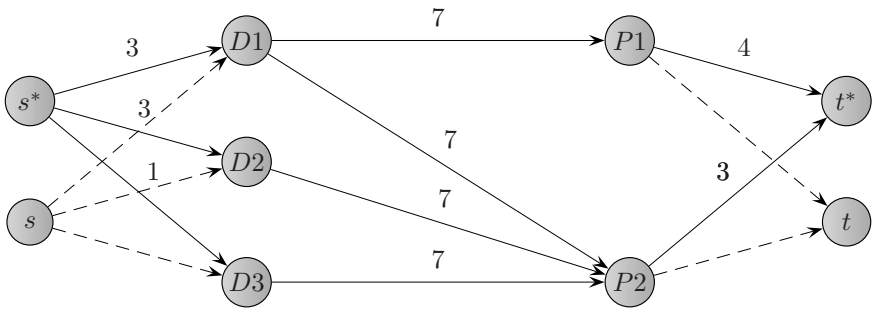


**Figure 5.2:**

Transformation of network $G$ into $G^*$.

mass balance constraint, such that

1. $l_{ij} \leq x_{ij} \leq u_{ij}, \quad \forall\ (i,j) \in A$ (flow bound constraint)
2. $\sum_{(i,j)\in\delta^+(i)} x_{ij} - \sum_{(j,i)\in\delta^-(i)} x_{ji} = 0, \quad \forall\ i \in V$(mass balance constraint)

To solve the circulation problem, Ford and Fulkerson[2] suggest the following procedure. Adjunct two nodes $s^*$ and $t^*$ to $V$ and arcs as follows to $A$ to get the extended network $G^* = (V^*, A^*)$.
Add arc $(s^*, i)$ for all nodes $i$ in $V$, if $b(i) > 0$ and arc $(i, t^*)$ for all nodes $i$ in $V$, if $b(i) < 0$ with

$$b(i) = \sum_{j:(j,i)\in A} l_{ji} - \sum_{j:(i,j)\in A} l_{ij}, \qquad \forall\ i \in V.$$

Put the lower capacity at any arc in $G^*$ to zero and the upper capacity to

$$\begin{aligned} u^*_{ij} &= u_{ij} - l_{ij}, \qquad \text{for} \quad (i,j) \in A, \\ u^*_{s^*i} &= b(i), \qquad \text{for} \quad i \in V \quad \text{and} \quad (s^*,i) \in A^*, \\ u^*_{it^*} &= -b(i), \qquad \text{for} \quad i \in V \quad \text{and} \quad (s^*,i) \in A^*. \end{aligned}$$

See figure 5.2 for the transformation of the original network of figure 5.1 with an underlying pervious multiplier method, where the dashed arcs have zero capacity.

A feasible circulation x in $G$ corresponds to a flow $x^*$ from $s^*$ to $t^*$ in $G^*$ by putting

$$\begin{aligned} x^*_{ij} &= x_{ij} - l_{ij}, \qquad \text{for} \quad (i,j) \in A, \\ x^*_{s^*i} &= b(i), \qquad \text{for} \quad (s^*,i) \in A^*, \\ x^*_{it^*} &= -b(i), \qquad \text{for} \quad (i,t^*) \in A^*. \end{aligned}$$

[2]See [9] on page 50.

Whenever there is a flow $x^*$ in $G^*$, saturating all arcs emanating $s^*$ and therefore all arcs ending in $t^*$, a circulation in $G$ exists. We can establish a necessary and sufficient condition for a flow $x^*$ in $G^*$ to exist, with flow value $v(x^*)$ larger than or equal to $h$, if $s(0) > 0$, or $h - e_{++}$, if $s(0) = 0$. By theorem 3.2.1 on page 17 any cut capacity must exceed or equal the demanded flow value. Let $S^* \subseteq V^*$ be a set of nodes, defining an $s$-$t$ separating cut $\delta(S^*)$. Let $S = S^* \setminus \{s^*\}$ and $\overline{S} = \overline{S^*} \setminus \{t^*\}$. For the cut capacity of $\delta(S^*)$ see the following:

$$\begin{aligned} u(S^*) &= u(S \cup s^*) = \sum_{(i,j)\in A^*, i\in S\cup s^*, j\in\overline{S}\cup t^*} u^*_{ij} = \\ &= \sum_{(i,j)\in A,\ i\in S,\ j\in\overline{S}} u^*_{ij} + \sum_{(s^*,i)\in A^*, i\in\overline{S}} u^*_{s^*i} + \sum_{(i,t^*)\in A^*, i\in S} u^*_{it^*}. \end{aligned}$$

Further on, the nodes $s$ and $t$ are dropped from consideration, since they do not contribute to any cut capacity due to their isolation.
In the pervious case, for $S = \emptyset$ the first and the last sum vanish, leaving the middle term, which sums up to $h$. The same holds for $S = V$, the last term sums up to $h$, whereas the rest becomes zero. In any other case, except that of $S = I \cup J$ and $\overline{S} = \overline{I} \cup \overline{J}$ where $I \subset \{1, \ldots, k\}$, $J \subset \{1, \ldots, l\}$ and $w_{I\overline{J}} = 0$, the cut capacity gets larger than $h$, since all arcs between a district and a party node carry upper capacity of $h$. Take $S$ to be the union of any $I$ and $J$ with $w_{I\overline{J}} = 0$. The cut capacity gets

$$\begin{aligned} u(S^*) &= \sum_{(i,j)\in A,\ i\in S,\ j\in\overline{S}} u^*_{ij} + \sum_{(s^*,i)\in A^*, i\in\overline{S}} u^*_{s^*i} + \sum_{(i,t^*)\in A^*, i\in S} u^*_{it^*} \\ &= 0 + (h - r_I) + c_J. \end{aligned}$$

Hence, a necessary and sufficient condition for the cut capacity to be larger than or equal to $h$ is, that for any $I \subset \{1, \ldots, k\}$ and $J \subset \{1, \ldots, l\}$ with $w_{I\overline{J}} = 0$,

$$(h - r_I + c_J \geq h \quad \Leftrightarrow \quad c_J \geq r_I)$$

This proofs theorem 2.3.2 on page 10.
In the impervious case, see that arcs $(s^*, i)$ carry upper capacity of $r_i - e_{i+}$, such that, for a feasible flow in $G$ to exist, all cut capacities in $G^*$ must exceed $h - e_{++}$. Again, if $S = s^*$ or $S = V^* \setminus \{t^*\}$, the cut capacity $u(S)$ is equal to $h - e_{++}$. In any other case, except that of $S = s^* \cup I \cup J$ and $\overline{S} = \overline{I} \cup \overline{J} \cup t^*$ where $w_{I\overline{J}} = 0$, the cut capacity gets larger than $h-1$ and thus $h-e_{++}$. Take $S$ to be the union of $\{s^*\}$ and any $I \subset \{1, \ldots, k\}$ and $J \subset \{1, \ldots, l\}$ with $w_{I\overline{J}} = 0$ to get the necessary and sufficient condition for a feasible circulation in $G$ to exist.

$$\begin{aligned} u(S^*) &= \sum_{(i,j)\in A,\ i\in S,\ j\in\overline{S}} u^*_{ij} + \sum_{(s^*,i)\in A^*, i\in\overline{S}} u^*_{s^*i} + \sum_{(it^*)\in A^*, i\in S} u^*_{it^*} = \\ &= 0 + h - e_{++} - (r_I - e_{I+}) + (c_J - e_{+J}) \end{aligned}$$

and therefore $u^*(S) \geq h - e_{++}$ if and only if

$$c_J \geq r_I - e_{I+} + e_{+J} = r_I + e_{\overline{I}J} - e_{I\overline{J}} = r_I + e_{\overline{I}J}$$

for any $I \subset \{1, \ldots, k\}$ and $J \subset \{1, \ldots, l\}$ with $w_{I\overline{J}} = 0$.
This proofs theorem 2.4.2 on page 11.

Based on these observations and on the fact, that the cut $\delta(\{s^*\})$ has cut capacity $u(\{s^*\}) = h$ in the pervious and $u(\{s^*\}) = h - e_{++}$ in the impervious case, we can formulate an alternative condition for a feasible apportionment to exist: Solve the maximum flow problem on $G^*$. If its corresponding maximum flow value is equal to $h$ or $h - e_{++}$, respectively, all concerned arcs are saturated and thus, the sets $R^0(W, \sigma)$ and $R^1(W, \sigma)$ are nonempty. The given result and the similarity of the original and transformed network leads to a shortened procedure presented in the next section.

## 5.3 Algorithmic approach

Since in the transformed network $G^*$ the upper capacities are reduced by the value of their corresponding lower capacities in $G$, all arcs $(i, j)$ with $l_{ij} = u_{ij}$ and thus all arcs emanating from and ending in the nodes $s$ and $t$, vanish. Both nodes $s$ and $t$ stay unconnected, since $b(s) = b(t) = 0$. Due to their isolation, they do not play any further role in the network. At the same time, two new nodes $s^*$ and $t^*$ are inserted. Recall, that we assumed the number of arcs emanating from any district node $i$ to be less than or equal to the required number of seats in district $i$. This implies that $b(i)$, with node $i$ corresponding to a district $i$, is nonnegative for both the pervious and impervious case. The equivalent assumption for any party node $j$ urges its associated $b(j)$ to be limited from above by zero.
For the pervious case, $s(0) > 0$, arc $(s^*, i)$ is inserted if and only if $i$ corresponds to a district node, whereas arc $(i, t^*)$ is added if and only if node $i$ corresponds to a party node. Thus, $s^*$ and $t^*$ take over the role of the isolated $s$ and $t$, respectively, and we define the network $G_{(W,\sigma),s}$ of the problem $(W, \sigma)$ for a pervious multiplier method $A^s$ as follows:

1. insert a node for each party and each district,
2. add a source node $s$ and a sink node $t$,
3. insert arc $(i, j)$ with upper capacity $u_{ij} = h$ and lower capacity $l_{ij} = 0$ if and only if $w_{ij} > 0$,
4. insert arc $(s, i)$ with upper capacity $u_{si} = r_i$ and lower capacity $l_{si} = 0$ for all nodes $i = 1, \ldots, k$ corresponding to a district node and
5. insert arc $(j, t)$ with upper capacity $u_{jt} = c_j$ and lower capacity $l_{jt} = 0$ for all nodes $j = 1, \ldots, l$ corresponding to a party node.

If and only if the maximum flow value of network $G_{(W,\sigma),s}$ reaches $h$, the needed saturation is given and an apportionment exists, so that we can formulate the following steps:

6. solve the maximum flow problem on $G_{(W,\sigma),s}$ to get a maximum flow $x$,
7. if $v(x) = h$, then a feasible apportionment exists.
   if $v(x) < h$, then no feasible apportionment exists.

**Theorem 5.3.1** *Let $(W, \sigma)$ be a problem and $A^s$ a pervious multiplier method. Condition (2.3.1) holds if and only if the maximum flow value of the network $G_{(W,\sigma),s}$ equals $h$.*

In the case of an impervious multiplier method, $s^*$ and $t^*$ take over the role of $s$ and $t$, again. The network $G_{(W,\sigma),s}$ based on the problem $(W, \sigma)$ for an impervious multiplier method $A^s$ is defined as follows:

1. insert a node for each party and each district,
2. add a source node s and a sink node t,
3. insert arc $(i, j)$ with upper capacity $u_{ij} = h$ and lower capacity $l_{ij} = 0$ if and only if $w_{ij} > 0$,
4. insert arc $(s, i)$ with upper capacity $u_{si} = r_i - e_{i+}$ and lower capacity $l_{si} = 0$ for all nodes $i = 1, \ldots, k$ corresponding to a district node and
5. insert arc $(j, t)$ with upper capacity $u_{jt} = c_j - e_{+j}$ and lower capacity $l_{jt} = 0$ for all nodes $j = 1, \ldots, l$ corresponding to a party node.

The reduction of the upper capacity bounds on arcs between district and party nodes is due to the needed reduced maximum flow value irrelevant and the upper capacity therefore put to $h$. If and only if the maximum flow value reaches $h - e_{++}$ the concerned arcs are saturated and thus existence is established. Thus,

6. solve the maximum flow problem on $G_{(W,\sigma),s}$ to get a maximum flow $x$,
7. if $v(x) = h - e_{++}$, then a feasible apportionment exists.
   if $v(x) < h - e_{++}$, then no feasible apportionment exists.

**Theorem 5.3.2** *Let $(W, \sigma)$ be a problem and $A^s$ an impervious multiplier method. Condition (2.4.1) holds if and only if the maximum flow value of the network $G_{(W,\sigma),s}$ equals $h - e_{++}$.*

## 5.4 Complexity

Given a network originated from a biproportional apportionment problem, the worst case complexity for the Labelling algortihm can be specified more precisely. The number of arcs is bounded by $(l + k + lk)$ and the worst case bound on the number of iterations is $O(h)$, since each augmentation increases the flow value of at least one and the maximum flow value is bounded by $h$. This leads to a overall worst case complexity of $O(hlk)$.

**Proposition 5.4.1** *Let $G$ be a network originated from the problem $(W, \sigma)$. The Labelling algorithm solves the maximum flow problem within $O(hlk)$ time.*

Together with the worst case bound $O(lk)$ for the transformation process we get the overall worst case complexity of $O(hlk)$.

# Chapter 6

# Violating sets

In the case of nonexistence of a feasible apportionment, it might be interesting, which constellation of parties and districts violate condition (2.3.1) and (2.4.1) on page 10 and 11, respectively. Section 1 shows how to determine such sets, which we denote by **violating sets**. Section 2 examines the complement sets, both in consideration of violating the relevant condition and their role within the failure of finding a feasible apportionment. Section 3 provides some information about the quality of violating sets.

## 6.1 Identification of violating sets

With a given maximum flow, a dual solution can be fixed, representing a minimal $s$-$t$ separating cut, which itself determines the sets $I$ and $J$, for which the above mentioned conditions are violated.
Let $x$ be such a maximum flow in the network $G$, originated from a problem. Define $L$ as the set of nodes, that are directly reachable from the source node $s$ in the residual network $G(x)$ as defined in the proof of theorem 3.2.2 on page 18. This set can be partitioned into the source node $\{s\}$ and two sets $I$ and $J$, where $I$ contains all nodes corresponding to a district node and $J$ consists of all nodes corresponding to a party node. Thus,

$$L = \{s\} \cup I \cup J.$$

Notice that $L$ equals the set of labelled nodes in the last iteration of the Labelling algorithm and defines an $s$-$t$ separating minimal cut, since by definition of $L$, any arc $(i, j)$ emanating $L$ is saturated and any arc $(i, j)$ leading into $L$ does not carry positive flow. Hence, the cut capacity is equal to the maximum flow and $\delta(L)$ is minimum.
Notice, that there is no arc $(i, j)$, with $i \in I$ and $j \in \overline{J}$. If so, node $j$ would be labelled ($j \in J$), since $(i, j)$ cannot be saturated. This contradicts our assumption. Hence, $w_{ij} = 0$ for all $(i, j)$, with $i \in I$ and $j \in \overline{J}$. Additionally, we know that the maximum flow and thus the minimum cut capacity is strictly

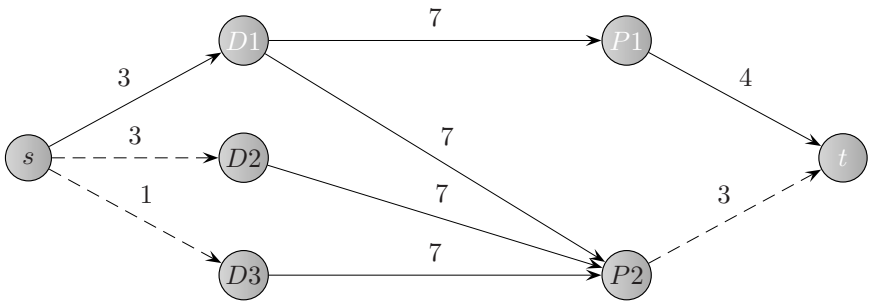


**Figure 6.1:**

Network $G_{(W^*,\sigma^*),s}$ with the labelled nodes $L = \{s, D2, D3, P2\}$. The dashed arcs illustrate the unfavorable constellation.

less than $h$ in the pervious case and strictly less than $h - e_{++}$ in the impervious case. We get

$$h > r_{\overline{I}} + c_J = h - r_I + c_J \qquad \Leftrightarrow \qquad r_I > c_J, \qquad \text{and}$$

$$h - e_{++} > r_{\overline{I}} - e_{\overline{I}+} + c_J - e_{+J} = h - r_I - e_{\overline{I}+} + c_J - e_{+J} \qquad \Leftrightarrow \qquad r_I + e_{\overline{I}J} > c_J,$$

respectively. This leads to the following proposition.

**Proposition 6.1.1** *Let $G_{(W,\sigma),s}$ be a network, originated from a problem $(W, \sigma)$ and its underlying multiplier method $A^s$, as defined in section 5.3. If no feasible apportionment exists, the set $L$ of all labelled nodes of $G_{(W,\sigma),s}$ in the last iteration of the Labelling algorithm determines two sets $I \subset \{1, \ldots, k\}$ and $J \subset \{1, \ldots, l\}$, such that $I$ consists of all labelled nodes corresponding to a district and $J$ consists of all labelled nodes corresponding to a party, for which condition (2.3.1) and (2.4.1), respectively, is violated.*

Network $G$ in figure 4.2 on page 24 corresponds to the network $G_{(W^*,\sigma^*),s}$ in figure 6.1 with $W^* = \begin{pmatrix} \times & \times \\ 0 & \times \\ 0 & \times \end{pmatrix}$, where $\times$ denotes some positive number and $\sigma^* = ((3, 3, 1), (4, 3))$, with an underlying pervious multiplier method. Nodes 1 to 3 stand for district nodes $D1$, $D2$, $D3$ and nodes 4 and 5 correspond to the party nodes $P1$ and $P2$. Take the labelled nodes $L$ of this network in the last iteration of the Labelling algorithm on page 27 to get $L = \{s, D2, D3, P2\}$, $I = \{D2, D3\}$ and $J = \{P2\}$. We find, that $w_{I\overline{J}} = 0$ and $3 = c_J < r_I = 4$. In other words, the total number of seats of district $D2$ and $D3$ (4) cannot be saturated by $P2$, the only party running in these districts, which is only entitled to get a total of 3 seats.

## 6.2 Complement sets

In consideration of the following implementation, we wish to find some sets of parties and districts, such that the number of required seats by those parties exceed the total number of seats within the districts, where they run. Given $I$ and $J$, as defined in the previous section, we now concentrate on their complement sets $\overline{I}$ and $\overline{J}$ and get

$$c_J < r_I \quad \Leftrightarrow \quad h - c_{\overline{J}} < h - r_{\overline{I}} \quad \Leftrightarrow \quad c_{\overline{J}} > r_{\overline{I}}$$

in the pervious and

$$r_I + e_{\overline{I}J} > c_J \quad \Leftrightarrow \quad h - r_{\overline{I}} + e_{\overline{I}J} > h - c_{\overline{J}} \quad \Leftrightarrow \quad c_{\overline{J}} > r_{\overline{I}} - e_{\overline{I}J}$$

in the impervious case.

The relevant condition is only checked for the complement sets $\overline{I}$ and $\overline{J}$, if $w_{\overline{I}J} = 0$. In this case, we see that for pervious multiplier methods, condition (2.3.1) on page 10 holds. For impervious multiplier methods we do not get a usable statement, since no further information is provided upon $e_{I\overline{J}}$.

Nevertheless, the constellation within the complement sets is as unfavorable as for $I$ and $J$ itself. This is based on the fact, that we deal with equality constraint problems. The inequality for the complement sets is to interpret as follows:
The overall demand of seats of all parties $\{j : j \in \overline{J}\}$ cannot be satisfied by the available seats of districts $\{i : i \in I\}$. Notice that in the impervious case, 'available' denotes the total number of seats reduced by the number of parties $\{j : j \in J\}$, which run in district $\{i : i \in I\}$.
Take the previous example of $G_{(W^*,\sigma^*),s}$, to get $\overline{I} = \{D1\}$ and $\overline{J} = \{P1\}$, such that the required number of seats by party $P1$ (4), does exceed the number of available seats in district $D1$ (3), where $P1$ gained positive weight. Or reformulated, as given in the implementation, the number of total seats for district $D1$ (3) is less than what party $P1$ would need there (4).

In the impervious case, it is possible, that we can find some district $d \in \overline{I}$, where there is some party $j \in J$, that has positive weight, but still no party $u \in \overline{J}$ is running there, such that $w_{dJ} > 0$ and $w_{d\overline{J}} = 0$. For the allocation of seats of parties in $\overline{J}$, this means that the seats of district $d$ are not at their disposal. Hence, the above formulated explanation can even be tightened, since $r_i \geq e_{i+}$, $\forall\ i \in \{1, \ldots, k\}$. We define

$$\begin{aligned} I' &:= I \cup D \quad \text{and} \\ J' &:= J, \end{aligned}$$

where $D := \{d : d \in \overline{I}, w_{dJ} > 0 \text{ and } w_{d\overline{J}} = 0\}$, to get

$$c_{\overline{J}} > r_{\overline{I}} - r_D - e_{\overline{I}J} + e_{DJ} = r_{\overline{I'}} - e_{\overline{I'}J}.$$

Within the implementation, we have used the adjusted complement sets $\overline{I'}$ and $\overline{J'}$, to explain the reason of failure.

## 6.3 Properties of violating sets

As we have seen in the previous section, there are still other sets, next to $I$ and $J$, denying a feasible apportionment. Thus, minimality in terms of theorem 3.3.1 on page 19 can only be formulated with additional restrictions. The demand of violating condition (2.3.1) and (2.4.1) on page 10 and 11 follows from above.

Define the weight matrix $W$ to be **disconnected**, if we can find some nonempty sets $E \subset \{1, \ldots, k\}$ and $F \subset \{1, \ldots, l\}$, such that $w_{E\overline{F}} = w_{\overline{E}F} = 0$. In this case we can permute rows and columns of $W$ to get a matrix of the following form:

$$W = \left(\begin{array}{c|c} W1 & 0 \\ \hline 0 & W2 \end{array}\right)$$

$W$ is called **connected**, if it is not disconnected.[1]

Given the problem $(W, \sigma)$, with $W$ disconnected, we define two subproblems $(W1, \sigma1)$ and $(W2, \sigma2)$, where $\sigma1$ contains all necessary row and column constraints for the weight matrix $W1$. $\sigma2$ is defined analogously. If there is no feasible apportionment for one of the subproblems, there is none for $(W, \sigma)$. Henceforth, we restrict on connected weight matrices.

Moreover, we find that for any set $I \subset \{1, \ldots, k\}$ and $J \subset \{1, \ldots, l\}$, which violate the relevant condition, we cannot imply that the $s$-$t$ separating cut $\delta(L)$, with $L := \{s\} \cup I \cup J$ is minimal.
For an example, see figure 6.2 and 6.3. We take the already well-known weight matrix $W^*$, the constraint vector $\sigma^{**} := ((2, 3, 1), (4, 2))$ and an underlying pervious multiplier method to get the network illustrated in figure 6.2. A maximum flow $x = (x_{s,D1}, x_{s,D2}, \ldots, x_{P2,t}) = (2, 2, 0, 2, 0, 2, 0, 2, 2)$ with flow value $v(x) = 4$ is already established. Based on $x$ we determine $L = \{s, D2, D3, P2\}$, $I = \{D2, D3\}$ and $J = \{P2\}$, such that $w_{I\overline{J}} = 0$ and $2 = c_J < r_I + e_{\overline{I}J} = 5$.

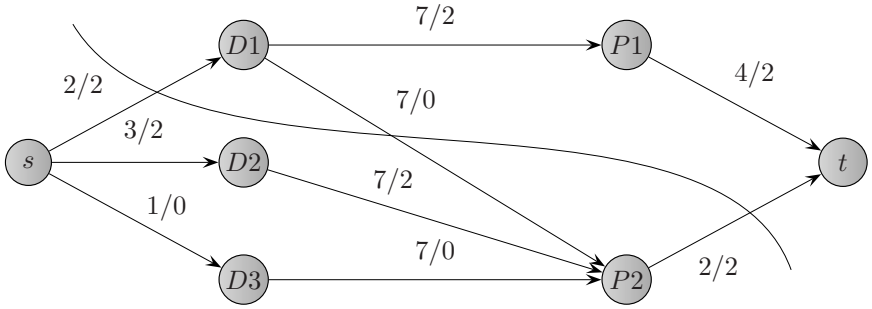


**Figure 6.2:**

$G_{(W^*,\sigma^{**}),s}$ with the realized maximum flow $x = (x_{s,D1}, x_{s,D2}, \ldots, x_{P2,t}) = (2, 2, 0, 2, 0, 2, 0, 2, 2)$ and $v(x) = 4$. $\delta(L) = \{(s, D1), (D1, P2), (P2, t)\}$ with $u(L) = 4$.

[1] See Bacharach [3] on page 47.

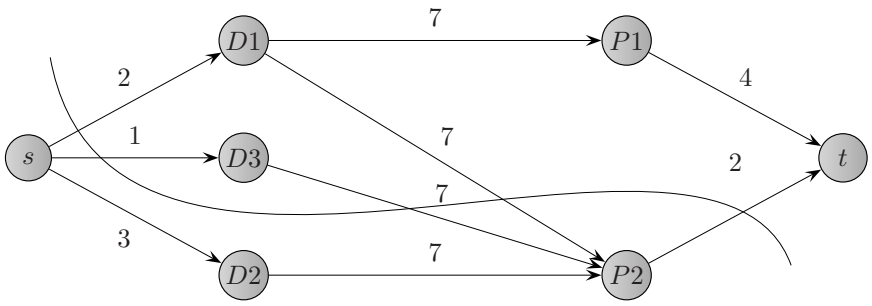


**Figure 6.3:**

Network $G_{(W^*,\sigma^{**}),s}$ with the cut $\delta(\{s, D2, P2\}) = \{(s, D1), (s, D3), (D1, P2), (D3, P2), (P2, t)\}$ and cut capacity $u(\{s, D2, P2\}) = 5$.

Take $I^\circ = \{D2\}$ and $J^\circ = \{P2\}$ to get $w_{I^\circ \overline{J^\circ}} = 0$ and $2 = c^\circ_J < r^\circ_I = 3$. Although $I^\circ$ and $J^\circ$ violate condition (2.3.1) on page 10, they do not define a minimal cut, since $u(s \cup I^\circ \cup J^\circ) = u_{s,D1} + u_{s,D3} + u_{P2,t} = 5 > 4 = v(x)$. See figure 6.3 for the cut $\delta(\{s, D2, P2\})$.

Hence, we further restrict on problems for which the maximum flow value in the underlying network is $h - 1$ in the pervious and $h - e_{++} - 1$ in the impervious case. This forces any cut, defined by some sets $I$ and $J$, violating the relevant condition, to be minimum, since $u(\{s\} \cup I \cup J) < h$ in the pervious, $u(\{s\} \cup I \cup J) < h - e_{\overline{I}J}$ in the impervious case and every maximum flow value is integer valued.
We can formulate the following proposition.

**Proposition 6.3.1** *Let $(W, \sigma)$ be a problem, with $W$ connected, $A^s$ a multiplier method and $G_{(W,\sigma),s}$ its corresponding network. If the minimum cut capacity equals $h - 1$ in the pervious and $h + e_{++} - 1$ in the impervious case, the sets $I$ and $J$, determined as in proposition 6.1.1, denote a minimal set of both parties and districts, for which condition (2.3.1) and (2.4.1), respectively, is violated.*

Upholding the restrictions from above, we can now reformulate the result of theorem 3.3.2 on page 20.

**Proposition 6.3.2** *Let $(W, \sigma)$ be a problem, with $W$ connected, $A^s$ a multiplier method and $G_{(W,\sigma),s}$ its corresponding network. If the minimum cut capacity equals $h - 1$ in the pervious and $h + e_{++} - 1$ in the impervious case, the sets $I$ and $J$, determined as in proposition 6.1.1, are unique if and only if the minimum cut $\delta(L')$, based on the complement set of $\overline{L'}$ as defined in section 3.3, is equal to the minimum cut $\delta(\{s\} \cup I \cup J)$.*

In the case of network $G_{(W^*,\sigma^*),s}$, with $W^*$ connected, and an underlying pervious multiplier method, we get the minimum cut capacity $u(L) = 6 = h - 1$. Hence we determine $\overline{L'} = \{t, P1, D1\}$ and find $\delta(L) = \delta(L')$, such that the sets $I = \{D2, D3\}$ and $J = \{P2\}$ are the only sets within this network, which violate condition (2.3.1) on page 10.

# Chapter 7

# Example

Let $(W^*, \sigma^*)$ be the given problem, with $W^* = \begin{pmatrix} \times & \times \\ 0 & \times \\ 0 & \times \end{pmatrix}$, where $\times$ denotes some positive number and $\sigma^* = ((3,3,1),(4,3))$. $A^s$ denotes the underlying impervious multiplier method. We want to check, wether there is a feasible solution to this problem or not, using the results of the previous chapters.

## 7.1 Transformation

We start with transforming the problem into the network $G_{(W^*,\sigma^*),s}$ as defined in section 5.3 on page 34 and illustrated in figure 7.1. For the upper capacities we get:

$$\begin{aligned} u_{s,D1} &= r_{D1} - e_{D1,+} = 3 - 2 = 1, \\ u_{s,D2} &= r_{D2} - e_{D2,+} = 3 - 1 = 2, \\ u_{s,D3} &= r_{D3} - e_{D3,+} = 3 - 3 = 0, \\ u_{s,P1} &= c_{P1} - e_{+,P1} = 4 - 1 = 3, \\ u_{s,P2} &= c_{P2} - e_{+,P2} = 3 - 3 = 0. \end{aligned}$$

## 7.2 Solving the maximum flow problem

We continue with the second step and go on solving the maximum flow problem on $G_{(W^*,\sigma^*),s}$ by applying the Labelling algorithm. In the first iteration we determine path $P : s \to D1 \to P1 \to t$ with its associated residual capacity $r(P) = \min\{r_{s,D1}, r_{D1,P1}, r_{P1,t}\} = \min\{1,7,3\} = 1$. In the next iteration no further directed path can be found, so that the algorithm stops with the maximum flow $x = (x_{s,D1}, x_{s,D2}, \ldots, x_{P1,t}) = (1,0,1,0,0,0,1)$ and its flow value $v(x) = 1$. The determined flow and the finally labelled nodes $L = \{s, D2, P2\}$

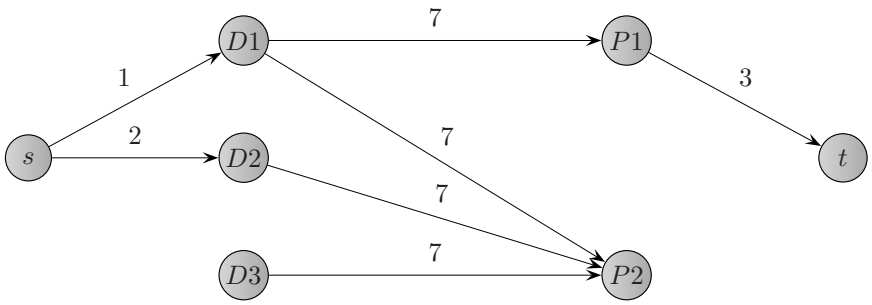


**Figure 7.1:**

Transformation of the problem $(W^*, \sigma^*)$ into the network $G_{(W^*,\sigma^*),s}$.

are presented in figure 7.2, where $z/y$ above any arc is to interpret as upper capacity bound/actual flow.

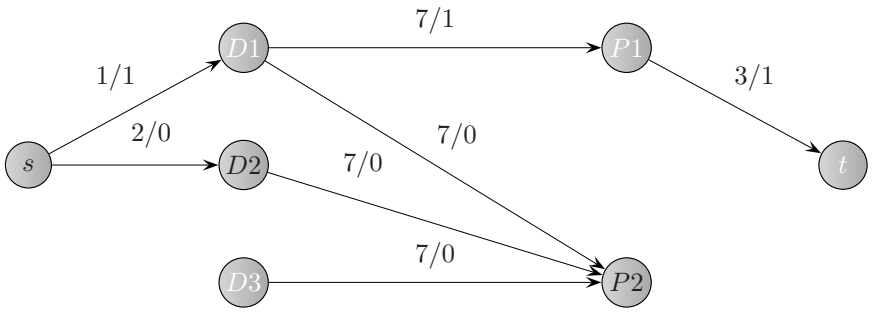


**Figure 7.2:**

The maximum flow $x = (x_{s,D1}, x_{s,D2}, \ldots, x_{P1,t}) = (1, 0, 1, 0, 0, 0, 1)$ in $G_{(W^*,\sigma^*),s}$ with $v(x) = 1$ and $L = \{s, D2, P2\}$.

With $e = \begin{pmatrix} 1 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix}$, we get $e_{++} = 4$ and thus no feasible solution exists, since $h - e_{++} = 7 - 4 = 3 > 1 = v(x)$.

## 7.3 Identifying violating sets

Since there is no feasible apportionment, we go on determining the sets $I$ and $J$ as defined in proposition 6.1.1 on page 37 and get

$L = \{s, D2, P2\} = \{s\} \cup \{D2\} \cup \{P2\}$,
$I = \{D2\}$, $\overline{I} = \{D1, D3\}$ and
$J = \{P2\}$, $\overline{J} = \{P1\}$ with

$$r_I + e_{\overline{I}J} = 3 + 2 > 3 = c_J.$$

This means, party $P2$ totals three seats, which is not enough to satisfy the total demand of seats of district $D2$, where it is the only party running in, plus two extra seats in district $D1$ and $D2$, where it has positive weight. In figure 7.3 the corresponding arcs are dashed and illustrate the unfavorable constellation. $(y, z)$ above any arc denotes the lower and upper capacity bound.

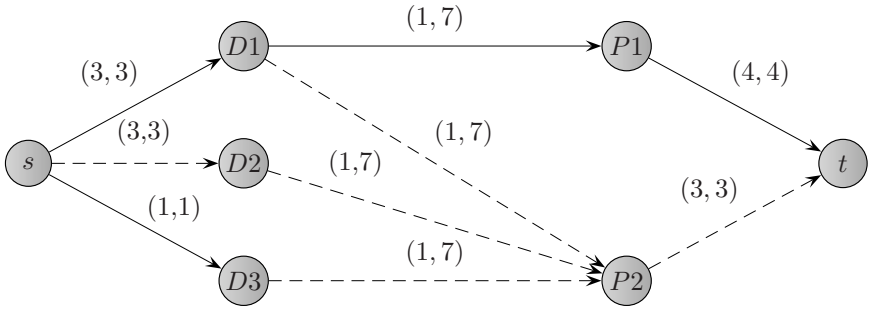


**Figure 7.3:**

The dashed arcs demonstrate the unfavorable constellation.

The returned string, explaining the reason of failure within the implementation, uses the adjusted complement sets $\overline{I'}$, $\overline{J'}$ as defined in section 6.2 on page 38. Consequently,

$$\overline{I'} := \overline{I} \setminus \{D3\} = \{D1\},$$

since $w_{D3,J} = 2$ and $w_{D3,\overline{J}} = 0$. We get

$$r_{\overline{I'}} - e_{\overline{I'}J} = 3 - 2 < 3 = c_J.$$

Parties $P1$ and $P2$ would total 5 seats, but district $D1$, where they run just commands 3 seats. Again, we have illustrated the unfavorable constellation in figure 7.4.

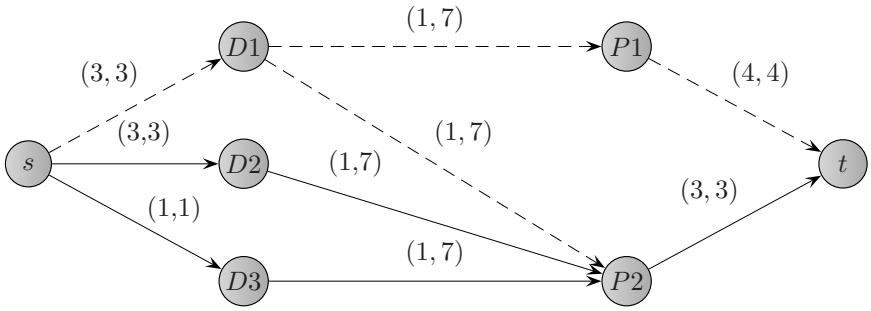


**Figure 7.4:**

The dashed arcs illustrate the unfavorable constellation.

## 7.4 Properties of the violating sets

Since the minimum cut capacity is less than $v(x) - 1 = 2$, we can neither apply proposition 6.3.1 on page 40 nor proposition 6.3.2 on page 40.

# Chapter 8

# Pseudo-code

The pseudo-code is conformed with the notation of the Augsburg BAZI Pseudo-Code[1].

As throughout the work, the weight matrix is denoted by $W = (w_{ij})$, $i \leq k$, $j \leq l$; marginal row and column constraints are given by the variables $row = (row_i)_{i \leq k}$ and $col = (col_j)_{j \leq l}$. Notation as to the generated network agrees with that introduced in chapter 2. The pseudo-code for the Labelling algorithm is adopted from chapter 4.

**Input**

$w_{ij} \geq 0$, real, $i = 1, \ldots, k$, $j = 1, \ldots, l$;
$row_i \geq 0$, integer, $i = 1, \ldots, k$;
$col_j \geq 0$, integer, $j = 1, \ldots, l$;
$s(0)$, integer, $s$ signpost sequence

**Output**

The algorithm returns a null-string, if there is a feasible apportionment, otherwise the returned string contains an explanation why a feasible apportionment must fail.

[1]The Augsburg BAZI Pseudo-Code [15] by Friedrich Pukelsheim and the BAZI team, university Augsburg.

**subroutine** generateNetwork
**begin**
insert $l + k + 2$ nodes, which either stand for a party, a district, the sink $t$ or the source $s$;
**if** $s(0) > 0$ **then**
adjunct arc $(s, i)$ for all district nodes $i = 1, \ldots, k$ and arc $(j, t)$ for all party nodes $j = 1, \ldots, l$, with $l_{si} := 0$, $u_{si} := row_i$ and $l_{jt} := 0$, $u_{jt} := col_j$;
add arc $(i, j)$, with $i$ corresponding to a district node and $j$ corresponding to a party node, if and only if $w_{ij} > 0$, with $l_{ij} := 0$, $u_{ij} := \{\text{sum}_j\ col_j\} = h$;
**else**
adjunct arc $(s, i)$ for all district nodes $i = 1, \ldots, k$ and arc $(j, t)$ for all party nodes $j = 1, \ldots, l$, with $l_{si} := 0$, $u_{si} := row_i - e_{i+}$ and $l_{jt} := 0$, $u_{jt} := col_j - e_{+j}$;
add arc $(i, j)$, with $i$ corresponding to a district node and $j$ corresponding to a party node, if and only if $w_{ij} > 0$, with $l_{ij} := 0$, $u_{ij} := \{\text{sum}_j\ col_j\} = h$;
**end;**

**subroutine** augment
**begin**
use the predecessor *pred* to trace back from the sink to the source to obtain an augmenting path $P$ from node $s$ to $t$ ;
$\delta := \min\{r_{ij} : (i, j)\ \epsilon\ P\}$;
augment $\delta$ units of flow along $P$ and update the residual capacities;
**end;**

**algorithm** existence test
**begin**
**subroutine** generateNetwork;
x:=0,e:=0,d:=1,p:=1;
label node $t$;
**while** $t$ is labelled **do**
**begin**
unlabel all nodes;
set $pred(i) := 0 \quad \forall\ i \in V$;
label $s$ and set LIST:= $\{s\}$;
**while** LIST$\neq \emptyset$ and $t$ unlabelled **do**
**begin**
remove a node $i$ from LIST;
**for** each arc $(i, j)$ in the residual network emanating node $i$ **do**
**begin**

```
                if node j is unlabelled then
                    set pred(j) := i, label j and add j to LIST;
            end;
        end;
        if t is labelled then
            subroutine augment;
    end;
    if ( s(0) > 0 and v(x) < h ) or (s(0) = 0 and v(x) < h − e++) then
    begin
        for each labelled node i do
        begin
            if node i corresponds to party t then p_t = 0;
            if node i corresponds to district z and s(0) > 0 then d_z = 0;
        end;
        if s(0) = 0 then
        begin
            d:=0;
            for i = 1, ..., k and j = 1, ..., l do
            begin
                if p_j = 1 and w_ij > 0 then d_i = 1;
            end;
            for i = 1, ..., k and j = 1, ..., l do
            begin
                if d_i = 1 and p_j = 0 and w_ij > 0 then e = e + 1;
            end;
        end;
        prompt
            "NA: Not Available: Parties {j : p_j ≠ 0} would total
            [sum_{j:p_j=1} p_j col_j]+e seats, but the districts {i : d_i = 1}
            where they run command just [sum_i d_i row_i] seats.";
        return;
    end;
prompt transient message "Iteration [Step] ...";
return;
end;
```

# Chapter 9

# Implementation

The implementation was done in Java, the code is given in the appendix. The programm consists of two classes, NetworkChange and MaxFlow. As a subroutine for the BAZI algorithm, it specifies wether there is a solution to a biproportional problem or not. The programm is based on the results of the previous chapters and has to be started within the Class BipropMethod by generating an object of the class MaxFlow. The classes themselves, contain following methods:

1. **MaxFlow**: String existSolution(Weight [ ][ ] WeightMatrix), boolean containsPath(), String getCut(Weight [ ][ ] WeightMatrix), boolean contains(int [ ] set, int node);
2. **NetworkChange**: Vector getAdjalists(), Vector getRAdjalists(), int getM(), int getN(), int getE() .

Additionally, each class has its constructor NetworkChange(Weight [ ][ ] WeightMatrix, int [ ] Districtres, int [ ] Partyres, int numberDistricts, int numberParties, int Divisormethod) and MaxFlow(Weight [ ][ ] WeightMatrix, int [ ] Districtres, int [ ] Partyres, int Divisormethod), respectively.

The programm can essentially be partitioned into four subroutines:

1. conversion of the problem $(W, \sigma)$ into a network ($\rightarrow$ class Network- Change)
2. establishing the maximum flow ($\rightarrow$ class MaxFlow $\rightarrow$ method boolean containsPath())
3. comparing the maximum flow to the relevant number: $h$ in the pervious case and $h - e_{++}$ in the impervious case ($\rightarrow$ class MaxFlow $\rightarrow$ method boolean existSolution())
4. determination and return of a set of parties and districts explaining the reason of failure ($\rightarrow$ class MaxFlow $\rightarrow$ method String getCut())

The last subroutine can be dropped from consideration, if existence had been established before.

## 9.1 Class NetworkChange

Within the class MaxFlow an object of the class NetworkChange is generated upon calling its constructor. Handing over the weight matrix of concern, its marginal constraint arrays, the number of parties and districts and an integer value to specify wether the multiplier method is pervious or impervious, this constructor will determine the corresponding network. The network is stored due to its adjacency list of both its forward and backward arcs, for reasons of space sufficiency and the efficient manipulation.[1] Two Vectors Adjalists and RAdjalists are initialized in this process, denoting the adjacency list of the forward arcs and the backward arcs, respectively. The entries of each Vector are specified as LinkedList, such that for each node in the network, there is one LinkedList consisting of all adjacent nodes. The entries of each LinkedList themselves are organized again as Vectors, where the first component denotes the adjacent node itself. The second component tells about the capacity of the arc linking those two nodes and is followed by the actual flow, which is set to zero for all arcs in the beginning. Further components are for intern reasons and so far not important. Notice, that every arc $(i, j)$ is stored twice. Firstly, as a forward arc of node $i$ in the Vector Adjalists. Secondly as a backward arc of node $j$ in the Vector RAdjalists. At the same time the constructor determines the number of nodes, the number of arcs and the number of positive weights in the weight matrix, stored in the private variables $n$, $m$ and $E$. The methods Vector getAdjalists(), Vector getRAdjalists, int getN(), int getM() and int getE() return the forward and backward adjacency lists, the number of nodes and arcs and the number of positive weights within the weight matrix. This information will be the basis for the implemented algorithm to solve the maximum flow problem.

## 9.2 Class MaxFlow

To start the algorithm, an object of the class MaxFlow has to be generated, handing over the weight matrix, the marginal constraint arrays and an integer $i$, specifying the multiplier method. If it is impervious, then $i = 0$, otherwise $i = 1$. To start the calculation, the adjacency lists as well as the number of nodes and arcs are needed. As illustrated before, after generating an object NetworkChange, this information is provided upon calling the appropriate methods.
The actual calculation phase starts now. The method boolean containsPath() is executed until no further path can be found and the actual flow is maximum. Within this method, the Labelling algorithm is put into effect, such that in every iteration the search of an augmenting path is realized, followed by the augmentation, except in the last iteration, where the actual flow is already maximum. Based on the maximum flow, subroutine three starts and the flow value is compared to the relevant numbers, $h$ and $h - e_{++}$. This is realized by calling the method String existSolution() within the class BipropMethod. If equality holds,

[1] See Ahuja, Magnanti and Orlin [1] on page 46.

the algorithm terminates after returning the Null-String, which is interpreted by the BAZI program to go on with the calculation of an apportionment.
If, however, in the case of nonexistence, the maximum flow is too small, the last subroutine is executed by calling the method String getCut(), to determine a minimal cut and thus, a set of parties and districts, which explain the reason of failure. Since the variable Labelled still contains all labelled nodes of the last iteration of the Labelling algorithm, the minimum cut and thus the sets $I$ and $J$, violating the relevant condition, are easily determined. Notice, that for didactic reason, we take the complement sets $\overline{I'}$ and $\overline{J'}$ to generate the output string, explaining the reason of failure. Recall, that $\overline{I'}$ does only contain districts, where at least one of the parties in $\overline{J'}$ has positive weight. The total seats of all districts contained in $\overline{I'}$ and the required seats of all parties in $\overline{J'}$ are calculated. Again, the two cases of impervious and pervious multiplier method are treated differently, since in the first case the number of required seats has to be enlarged by the number $e_{\overline{I}J}$, such that the required seats represent a lower bound, whereas in the case of a pervious multiplier method, the number of required seats specifies the absolute number. The string is returned both for the method String getCut() and String existSolution() hereafter. The algorithm terminates. The BAZI program, realizing, that the returned string is not Null and thus no solution exists, stops after some further steps, too.

# Chapter 10

# Summary

By means of network theory we have found a necessary and sufficient condition for the existence of a feasible apportionment for biproportional multiplier methods. This condition is efficiently checked by transforming the problem into a network and establishing the maximum flow value. In the case of nonexistence, we have determined a procedure to get a set of parties and a set of districts, denying a feasible apportionment. Again, these results were based on graph theoretic considerations. With further restrictions on the problems and the determined maximum flow value we have provided some properties as to the minimality and uniqueness of the violating sets. Finally, we have formulated the procedure of establishing existence and determining the sets in question as pseudo-code. We provided some information about the implementation in Java, the code is given in the appendix.

However, an open questions is, how to proceed in the case of nonexistence of a feasible allotment. Since the apportionment method is used in political areas, the failure of an allotment is not acceptable.
Anthonisse [2] proposes to admit only those superapportionments and thus marginal constraints for the parties, for which a feasible solution exists. If we want to solve the problem within the given prerequisites, several political questions arise, which have to be answered. Are districts or parties with a deficit to be rewarded with extra seats due to a high electoral participation or the successful electioneering? Or are the districts and parties with the surplus to be punished for just the opposite reasons? Clearly, more work is needed to get a better understanding.

# Appendix A

# Algorithm - Balinski/Demange

**Theorem 2.3.1** *Let $(W, \sigma)$ be a problem and $s$ a pervious signpost sequence. $A^s(W, \sigma)$ is nonempty if and only if $R^0(W, \sigma)$ is nonempty.*

**Theorem 2.4.1** *Let $(W, \sigma)$ be a problem and the given multiplier method be impervious. $A^s(W, \sigma)$ is nonempty if and only if $R^1(W, \sigma)$ is nonempty.*

Existence is proofed constructively by an algorithm given by Balinski and Demange[1] and Gier[2], for the special case of equality constraint problems, which either determines a feasible apportionment or terminates without one in the case of nonexistence. Again, we only regard problems $(W, \sigma)$, for which the number of strictly positive entries in $W$ is less than or equal to $h$ in the impervious case.

The given problem $(W, \sigma)$ is transformed into a directed and capacitated network:

1. insert a node for each party and each district,
2. add a source node s and a sink node t,
3. insert arc $(i, j)$ with upper capacity $u_{ij} = h$ and lower capacity $l_{ij} = 0$ if and only if $w_{ij} > 0$,
4. insert arc $(s, i)$ with upper and lower capacity $u_{si} = l_{si} = r_i$ for all nodes $i = 1, \ldots, k$ corresponding to a district node and
5. insert arc $(j, t)$ with upper and lower capacity $u_{jt} = l_{jt} = c_j$ for all nodes $j = 1, \ldots, l$ corresponding to a party node,
6. insert arc $(t, s)$ with upper and lower capacity $u_{ts} = l_{ts} = h$.

[1] See [4] on page 205.
[2] See [10].

The algorithm starts with a trial solution $a = (a_{ij}), i \leq k, j \leq l$, where $a_{ij} = [\lambda_i w_{ij} \rho_j]_s$ and $a_{++} = h$, typically a nonfeasible circulation in the given network. The main idea of this algorithm is to determine circles within the given network on which a transfer of flow and thus seats decreases the total error

$$\triangle(a) := \sum_{i \in \{1,\ldots,k\}} |r_i - a_{i+}| + \sum_{j \in \{1,\ldots,l\}} |c_j - a_{+j}|$$

of at least one.
Define $I^< := \{i \in \{1,\ldots,k\} : a_{i+} < r_i\}$, $I^> := \{i \in \{1,\ldots,k\} : a_{i+} > r_i\}$ and analogously $J^<$, $J^>$. Obviously all marginal constraints are fulfilled, if $I^< = J^< = \emptyset$. The search for a possible circle is realized with a labelling procedure. Starting with the set $I^<$ (if $I^< \neq \emptyset$, we take $J^<$ and its corresponding adjusted procedure), we wish to label nodes, such that they determine a circle through node $s$. In this case, a transfer of flow on the circle would reduce the total error by at least one. Is no further labelling possible and $s$ is unlabelled, the nonnegative multipliers $\lambda$ and $\rho$ have to be adjusted, and we get a new scaled problem $\overline{w}_{ij} = (\lambda_i w_{ij} \rho_j)$. Is this scaling process successful, any choice of the new multipliers does not change the apportionment and further labelling is possible. Hence, after a finite number of successful scaling phases node $s$ is labelled and the total error can be reduced. Thus, a reduction of the total error to zero is possible within finite steps and the algorithm terminates with a feasible apportionment, if every scaling phase is successful. If it is unsuccessful, such that no possible scaling could be found, no feasible apportionment can exist, the sets $R^0(W,\sigma)$ and $R^1(W,\sigma)$, respectively, are empty and the algorithm terminates.

## A.1 Pseudo-code

The pseudo-code below, as provided by Gier[3], is conformed with the terminology hitherto.

Input: Network due to the problem $(W,\sigma)$ and its underlying multiplier method $A^s$, as defined above.
Output: An apportionment $a \in R^0(W,\sigma)$ or $a \in R^1(W,\sigma)$ or termination with the result that $R^0(W,\sigma)$ or $R^1(W,\sigma)$, respectively, is empty and no feasible apportionment exists.

**algorithm** Balinski/Demange
  **begin**
    determine $I^<$, $I^>$, $J^<$ and $J^>$;
    choose two nonnegative multipliers $\lambda \in R^k$ and $\rho \in R^l$,
    such that $\sum_{ij} [\lambda_i w_{ij} \rho_j]_s = h$;
    **while** $(I^< \neq \emptyset)$ **do**
    **begin**
      **while** s is unlabelled **do**

[3]See [10].

```
        begin
            if further labelling is possible then subroutine label;
            else
                if scaling is possible then subroutine scale;
                else EXIT: no feasible apportionment exists
        end;
    end;
    if s is labelled then subroutine augment
end;
```

Notice, that only the case of $I^< \neq \emptyset$ is treated. If $I^< = \emptyset \neq J^<$ then we continue with the equivalent procedure for $J^<$, which is formulated analogously.

```
subroutine label
    begin
        label all nodes i ∈ I^< and set pred(i) := s;
        define (s, i) to be a forward arc;
        for each arc (i, j), with i labelled, j unlabelled and (λ_i w_ij ρ_j) = s(a_ij) do
        begin
            label node j;
            set pred(j) := i;
            define arc (i, j) to be a forward arc;
        end;
        for each labelled node i with r_i < a_i+ do
        begin
            label node s;
            set pred(s) := i;
            define arc (s, i) to be a backward arc;
        end;
        for each arc (i, j), with i unlabelled, j labelled and
            (λ_i w_ij ρ_j) = s(a_ij − 1), with a_ij ≥ 1 do
        begin
            label node i;
            set pred(i) := j;
            define arc (i, j) to be a backward arc;
        end;
        for each labelled node j with c_j > a_+j do
        begin
            label node t;
            set pred(t) := j;
            define arc (i, j) to be a forward arc;
        end;
        if t is labelled then
            for each unlabelled node j with c_j < a_+j do
            begin
                label node j;
```

```
            set pred(j) := t;
            define arc (j,t) to be a backward arc;
        end;
    end;

subroutine scale
    begin
        δ1 := min{ s(a_ij) / (λ_i w_ij ρ_j) :  i labelled, j unlabelled};
        δ2 := min{ λ_i w_ij ρ_j / (s(a_ij) − 1) : i unlabelled, j labelled};
        δ := min{δ1, δ2};
        set λ := { δλ_i   for all labelled  i,
                 { λ_i    otherwise;
        set ρ := { (1/δ)ρ_i   for all labelled  j,
                 { ρ_i        otherwise;
    end;

subroutine augment
    begin
        use the predecessor pred to trace the circle C passing through
        node s;
        for each arc (i,j) on C, with (i,j) forward arc do
        begin
            set a_ij := a_ij + 1;
        end;
        for each arc (i,j) on C, with (i,j) backward arc do
        begin
            set a_ij := a_ij − 1;
        end;
    end;
```

## A.2 Correctness

The algorithm terminates, if the total error $\triangle(a) = 0$ or no further scaling is possible. Thus, we have to show that the algorithm terminates within a finite number of steps and returns a feasible apportionment, if every scaling phase is successful or no feasible apportionment exists and hence the sets $R^1(W\sigma)$ and $R^0(W, \sigma)$ are empty, if there is an unsuccessful scaling phase.

If no unsuccessful scaling phase occurs then the algorithm can reduce the total error by at least one after at most k+l scaling phases, since after every successful scaling phase further labelling is possible. With $\triangle(a) < \infty$, the algorithm terminates within a finite number of steps.

According to Gier[4], an appropriate scaling of $\lambda$ and $\rho$ only takes place, if no further labelling is possible and node $s$ stays unlabelled. Hence, there is no

[4]See [10].

labelled node $i$, for which $r_i < a_{i+}$. Moreover $(\lambda_i * w_{ij} * \rho_j) < d(a_{ij})$, with $i$ labelled and $j$ unlabelled and $s(a_{ij} - 1) < (\lambda_i w_{ij} \rho_j)$, with $i$ unlabelled and $j$ labelled.
We define the scalar $\delta$ as follows:

$$\begin{aligned}
\delta &:= \min\{\delta_1, \delta_2\}, \quad \text{with} \\
\delta_1 &:= \min\{\frac{s(a_{ij})}{\lambda_i w_{ij} \rho_j} : i \text{ labelled, } j \text{ unlabelled}\} \quad \text{and} \\
\delta_2 &:= \min\{\frac{\lambda_i w_{ij} \rho_j}{s(a_{ij} - 1)} : i \text{ unlabelled, } j \text{ labelled}\}.
\end{aligned}$$

No further scaling is possible, if $\delta$ and thus both $\delta_1$ and $\delta_2$ stay undefined. Define $I$ to consist of all labelled district nodes and $J$ to consist of all labelled party nodes.
$\delta_1$ is undefined, if all party nodes are labelled, $\overline{J} = \emptyset$, or $(\lambda_i w_{ij} \rho_j) = w_{ij} = 0$, for all $(i, j)$, with $i \in I$, $j \in \overline{J}$.
In the second case, $\overline{I} = \emptyset$, all district nodes are labelled, or $s(a_{ij} - 1) = 0$, for all $i \in \overline{I}$, $j \in J$ forces $\delta_2$ to be undefined.

Impervious multiplier methods ($s(0) > 0$):

If $(\lambda_i w_{ij} \rho_j) = w_{ij} = 0$, for all $(i, j)$, with $i \in I$, $j \in \overline{J}$ and $s(a_{ij} - 1) = 0$, for all $i \in \overline{I}$, $j \in J$, we get $a_{I\overline{J}} = 0 = a_{\overline{I}J}$. Together with the fact, that there is no labelled district for which $r_i < a_{i+}$, we get $a_{IJ} < r_I$. Notice that $c_J \leq a_{IJ}$, since $J \cap J^< = \emptyset$, if $t$ is unlabelled and $J^> \subseteq J$, if $t$ is labelled. We get

$$c_J \leq a_{IJ} < r_I,$$

what contradicts the demand that we should have $r_I = c_J$, since $a_{\overline{I}J} = 0 = a_{I\overline{J}}$.
$R^0(W, \sigma)$ is empty.
For any other combination with $I = \emptyset$ and $J = \emptyset$, we get the same result.

Impervious multiplier method ($s(0) = 0$):

If $(\lambda_i w_{ij} \rho_j) = w_{ij} = 0$, for all $(i, j)$, with $i \in I$, $j \in \overline{J}$ and $s(a_{ij} - 1) = 0$, for all $i \in \overline{I}$, $j \in J$, we get $w_{I\overline{J}} = 0$ and $w_{\overline{I}J} = e_{\overline{I}J}$. With $c_J \leq a_{+J}$ and $r_I < a_{I+}$ we get

$$c_J \leq a_{+J} = a_{IJ} + a_{\overline{I}J} = a_{IJ} + a_{I\overline{J}} + e_{\overline{I}J} = a_{I+} + e_{\overline{I}J} < r_I + e_{\overline{I}J}.$$

This contradicts the demand that we should have $c_J \geq r_I + e_{\overline{I}J}$. $R^1(W, \sigma)$ is empty. For any combination with $I = \emptyset$ and $J = \emptyset$ follows the same.

# Appendix B

# Java-code

## B.1 Class MaxFlow

```
package bazi.lib;
import java.util.*;
import bazi.*;


public class MaxFlow
{
   private int E, h, n, m, maxfluss = 0, gesamtfluss, s = 0, DivMethode;
   private int Anzahl_Parteien, Anzahl_Distrikte;
   private int [ ] rows, cols , Labelled;
   private double x;
   private char a;
   private String b;
   private Vector Adjalisten = new Vector();
   private Vector RAdjalisten = new Vector();
   private NetworkChange nu;
   private String [ ] districtNames;

   //constructor
   public MaxFlow(Weight[ ][ ] Gewicht, int[ ] Zeilen , int [ ] Spalten,
                  int Divisormethode, String [ ] dNames)
    {
      districtNames = dNames;
      nu = new NetworkChange(Gewicht, Zeilen, Spalten, Zeilen.length, Spalten.
           length, Divisormethode);
      Adjalisten = nu.getAdjalists () ;
      RAdjalisten = nu.getRAdjalists();
      gesamtfluss = 0;
      rows = Zeilen;
      cols = Spalten;
      x = 0;
```

```
        m = nu.getM();
        n = nu.getN();
        E=nu.getE();
        Anzahl_Parteien = Spalten.length;
        Anzahl_Distrikte = Zeilen.length;
        DivMethode = Divisormethode;
        h=0;

        for (int i=0; i<Zeilen.length; i++)
        {
            h = h + Zeilen[i];
        }
    }

    //method existSolution to establish the maximum flow/existence
    public String existSolution(Weight [ ][ ] Gewicht)
    {
        while ( containsPath() ) {}

        if ( ( h==gesamtfluss && DivMethode==1 )||
             ( gesamtfluss==h && DivMethode==0 ))
        {
                return null;
        }
        else
        {
            return getCut(Gewicht);
        }
     }

    //method containsPath to determine augmenting paths
    public boolean containsPath()
    {
        Labelled = new int[n + 1];
         int [ ] Kontrolliert = new int[n + 1];
         int [ ] Epsilon = new int[n + 1];
         int [ ][ ] Vorgaenger = new int[n + 1][2];

         for (int i = 0; i < n + 1; i++)
         {
                Labelled[i] = 0;
                Kontrolliert [i] = 0;
         }

         for (int i = 0; i < n + 1; i++)
         {
                for (int j = 0; j < 2; j++)
                {
                Vorgaenger[i][j] = 0;
```

```
80              }
                Epsilon[i] = 10000000;
        }

        int help1 = 0;
85      int help2 = 0;
        int help3 = 0;

        Labelled[help1] = 1;
        help1++;
90      Kontrolliert [help2] = 1;
        help2++;

        while (!contains(Labelled, n))
        {
95          if (! ( help2 == 0))
            {
              help3 = Kontrolliert[help2 - 1];
              Kontrolliert [help2 - 1] = 0;
              help2 = help2 - 1;
100
              Iterator it = ( (LinkedList) Adjalisten.elementAt(help3)).
                            listIterator ();

              int position1 = 0;
105           while (it.hasNext())
              {
                  Vector vhelp = (Vector) it.next();

                  if ( ! contains ( Labelled, ( (Integer) vhelp.get(0)).
110                     intValue()))
                  {
                     if ( ( ( Integer) vhelp.get(2)).intValue() <
                           ( (Integer) vhelp.get(1)).intValue())
                      {
115                     Vorgaenger[ ( (Integer) vhelp.get(0)).
                           intValue() ][0] = help3;
                        Vorgaenger[ ( (Integer) vhelp.get(0)).
                           intValue() ][1] = position1;
                        Epsilon [ ( ( Integer) vhelp.get(0)).intValue()] =
120                        java.lang.Math.min( ( (Integer) vhelp.get(1)).
                           intValue() - ( (Integer) vhelp.get(2)).
                           intValue(), Epsilon[help3] );

                        Labelled[help1] = ( (Integer) vhelp.get(0)).
125                                          intValue();

                        help1++;
```

```
                    if (!contains(Kontrolliert , ((Integer) vhelp.get(0)).
130                     intValue()))
                    {
                        Kontrolliert[help2] = ((Integer) vhelp.get(0)).
                            intValue();
                        help2++;
135
                    }
                }
            }
            position1++;
140     }

        Iterator ite = ((LinkedList) RAdjalisten.elementAt(help3)).
            listIterator();

145     int position2 = 0;
        while (ite.hasNext())
        {
            Vector vhelp = (Vector) ite.next();
            if (!contains(Labelled,
150                 ((Integer) vhelp.get(0)).intValue()) )
            {
            if ((((Integer) vhelp.get(2)).intValue()>0)
            {
                Vorgaenger[((Integer) vhelp.get(0)).
155                 intValue()][0] = -1 * help3;
                Vorgaenger[((Integer) vhelp.get(0)).
                    intValue()][1] = position2;
                Epsilon[((Integer) vhelp.get(0)).intValue()] =
                    java.lang.Math.min((Integer) vhelp.get(2)).
160                 intValue(), Epsilon[help3]);

                Labelled[help1] = ((Integer) vhelp.get(0)).intValue();

                help1++;
165             if (!contains(Kontrolliert , ((Integer) vhelp.get(0)).
                      intValue()))
                {
                    Kontrolliert[help2] = ((Integer) vhelp.get(0)).
                        intValue();
                    help2++;
                }
170         }

            }
            position2++;
        }
175
```

```
            }
            else
            {
               return false;
            }

        }

        int help4 = n;
        while (! (help4 == 1))
        {
            if (Vorgaenger[help4][0] >= 0)
            {

            int a = ((Integer) ((Vector) ((LinkedList) Adjalisten.
                get(Vorgaenger[help4][0])).get(Vorgaenger[help4][1])).
                elementAt(2)).intValue();

            ((Vector) ((LinkedList) Adjalisten.get
                (Vorgaenger[help4][0])).get(Vorgaenger[help4][1])).
                 removeElementAt(2);
            ((Vector) ((LinkedList) Adjalisten.get
                (Vorgaenger[help4][0])).get(Vorgaenger[help4][1])).
                 add(2, new Integer(a + Epsilon[n]));

            LinkedList b = (LinkedList) RAdjalisten.get(help4);
            Iterator lit = b.listIterator();
            int zahl = 0;
            while (lit.hasNext() && zahl >= 0)
            {
               Vector c = (Vector) lit.next();
                  if ((((Integer) c.get(0)).intValue() ==
                                        Vorgaenger[help4][0])
                  {
                     int ar = ((Integer) ((Vector) ((LinkedList)RAdjalisten.
                        get(help4)).get(zahl)).get(2)).intValue();
                     ((Vector) ((LinkedList) RAdjalisten.get(help4)).
                        get(zahl)).removeElementAt(2);
                     ((Vector) ((LinkedList) RAdjalisten.get(help4)).
                        get(zahl)).add(2,new Integer(ar + Epsilon[n]));
                     zahl = -2;
                  }
                  zahl++;
            }

            help4 = Vorgaenger[help4][0];
            }
            else
            {
```

```
225             int a = ((Integer) ((Vector) ((LinkedList)RAdjalisten.
                    get(-1 * Vorgaenger[help4][0])).get(Vorgaenger[help4][1])).
                    elementAt(2)).intValue();
                ((Vector) ((LinkedList) RAdjalisten.get(-1 * Vorgaenger
                    [help4][0])).get(Vorgaenger[help4][1])).
230                 removeElementAt(2);
                ((Vector) ((LinkedList) RAdjalisten.get(-1 * Vorgaenger
                    [help4][0])).get(Vorgaenger[help4][1])).
                    add(2, new Integer(a - Epsilon[n]));

235             LinkedList b1 = (LinkedList) Adjalisten.get(help4);
                Iterator lit1 = b1.listIterator();
                int zahl1 = 0;
                while (lit1.hasNext() && zahl1 >= 0)
                {
240                 Vector c = (Vector) lit1.next();
                        if ((((Integer) c.get(0)).intValue() ==
                                        -1 * Vorgaenger[help4][0])
                        {
                            ((Vector) ((LinkedList) Adjalisten.get(help4)).
245                             get(zahl1)).removeElementAt(2);
                            ((Vector) ((LinkedList) Adjalisten.get(help4)).
                                get(zahl1)).add(2,new Integer(a - Epsilon[n]));
                            zahl1 = -1;
                        }
250                     zahl1++;
                }

                help4 = -1 * Vorgaenger[help4][0];
                }
255

        }
        maxfluss = Epsilon[n];
        gesamtfluss = gesamtfluss + maxfluss;

260     return true;

    }


265 //method contains
    public boolean contains(int[] menge, int Knoten)
    {
       for (int i = 0; i < menge.length; i++)
       {
270        if (menge[i] == Knoten)
           {
              return true;
              }
```

```
275         }
            return false;
        }

        //method getCut to generate the output String
280     public String getCut(Weight [ ][ ] Gewicht)
        {
            String sDistricts = "";
            String sParties = "";
            String seatsRec = "";
285         String seatsEx = "";

            //for pervious multiplier methods
            if (DivMethode == 1)
            {
290             System.out.println("MF-IF");
                int SummeParteien = 0;
                int SummeDistrikte = 0;
                int [ ] D = new int[rows.length + 1];
                int [ ] P = new int[cols.length + 1];
295             for (int u = 0; u < rows.length + 1; u++)
                {
                    D[u] = 1;
                }
                D[0] = 0;
300             for (int h = 0; h < cols.length + 1; h++)
                {
                    P[h] = 1;
                }
                P[0] = 0;
305

                for (int i = 0; i < Labelled.length; i++)
                {
                    if (Labelled[i] > 1 + Anzahl_Distrikte && Labelled[i] < n)
                    {
310                     P[Labelled[i] - 1 - Anzahl_Distrikte] = 0;
                }
                }
                int h = 0;
                for (int f = 1; f < cols.length + 1; f++)
315             {
                    if (P[f] == 1)
                    {
                        if (h == 0)
                        {
320                         sParties += "\"" + Gewicht[0][f - 1].name + "\"";
                        }
                        else
```

```
                {
                    sParties += ", \"" + Gewicht[0][f - 1].name + "\"";
325             }
                SummeParteien += cols[f - 1];
                h++;
            }
            }

330
            seatsRec += SummeParteien;

            for (int j = 0; j < Labelled.length; j++)
            {
335         if (Labelled[j] <= 1 + Anzahl_Distrikte && Labelled[j] > 1)
               {
                  D[Labelled[j] - 1] = 0;
            }
            }
            int z = 0;
            for (int r = 1; r < rows.length + 1; r++)
            {
               if (D[r] == 1)
               {
345               if (z == 0)
                  {
                     sDistricts += "\"" + districtNames[r - 1] + "\"";
                     z++;
                  }
350               else
                  {
                     sDistricts += ", \"" + districtNames[r - 1] + "\"";
                   }
                  SummeDistrikte += rows[r - 1];
355               z++;
            }
            }

            seatsEx += SummeDistrikte;

360
        }

        //for impervious multiplier methods
        else
365     {
           System.out.println("MF-ELSE");
            int SummeParteien = 0;
            int SummeDistrikte = 0;
            int [ ] D = new int[rows.length + 1];
370         int [ ] P = new int[cols.length + 1];
            for (int u = 0; u < rows.length + 1; u++)
```

```
{
    D[u] = 0;
}
D[0] = 1;
for (int h = 0; h < cols.length + 1; h++)
{
    P[h] = 1;
}
P[0] = 0;

for (int i = 0; i < Labelled.length; i++)
{
    if (Labelled[i] > 1 + Anzahl_Distrikte && Labelled[i] < n)
    {
        P[Labelled[i] - 1 - Anzahl_Distrikte] = 0;
}
}

int h = 0;
for (int f = 1; f < cols.length + 1; f++)
{
    if (P[f] == 1)
    {
        System.out.println(f);
        if (h == 0)
        {
            sParties += "\"" + Gewicht[0][f - 1].name + "\"";
        }
        else
        {
            sParties += ", \"" + Gewicht[0][f - 1].name + "\"";
        }
        SummeParteien += cols[f - 1];
        h++;
}
}

h = 0;
for (int g = 1; g < rows.length + 1; g++)
{
    for (int gg = 0; gg < cols.length; gg++)
    {
        if (P[gg+1] == 1)
        {
            if (Gewicht[ (g - 1)][gg].weight > 0)
            {
                D[g] = 1;

            }
```

```
                }
            }
        }

425     int z = 0;
        for (int r = 1; r < rows.length + 1; r++)
        {
            if (D[r] == 1)
            {
430             if (z == 0)
                {
                    sDistricts += " \"" + districtNames[r - 1] + "\"";
                    z++;
                }
435             else
                {
                    sDistricts += ", \"" + districtNames[r - 1] + "\"";
                }
                SummeDistrikte += rows[r - 1];
440         }
        }

        for (int g = 1; g < rows.length + 1; g++)
        {
445         if (D[g]==1)
            {
                for (int gg = 0; gg < cols.length; gg++)
                {
                    if (P[gg+1] == 0)
                    {
450                     if (Gewicht[ (g - 1)][gg].weight > 0)
                        {
                            sParties += ", \"" + Gewicht[0][gg].name
                                    + "\"";
                            SummeParteien += 1;

                        }
                    }
                }
460         }
        }

        seatsRec += SummeParteien;
        seatsEx += SummeDistrikte;
465
    }

     String returnString =
         Resource.getString("bazi.gui.dpp.maxflow1") + " "
```

```
470             + sDistricts
                + " " + Resource.getString("bazi.gui.dpp.maxflow2")
                + seatsEx
                + Resource.getString("bazi.gui.dpp.maxflow3") + " "
                + sParties
475             + " " + Resource.getString("bazi.gui.dpp.maxflow4")
                + seatsRec
                + Resource.getString("bazi.gui.dpp.maxflow5");
            return returnString;
        }
480

    }
```

## B.2 Class NetworkChange

```
package bazi.lib;
import java.util.*;

 class NetworkChange
{

    private int E, n, m = 0;
    private Vector Adjazenzlisten;
    private Vector RAdjazenzlisten;
    private int Kantenanzahl = 0;

    //constructor to determine the network in question
    public NetworkChange(Weight[ ][ ] Gewicht, int[ ] Distriktres,
            int [ ] Parteires, int AnzahlDistrikte, int AnzahlParteien,
            int Divisormethode)
      {
        n = AnzahlParteien + AnzahlDistrikte + 2;
        Adjazenzlisten = new Vector();
        RAdjazenzlisten = new Vector();
        E=0;

        int helpstart;
        int helpende;
        int Kantenkap;

        for (int j = 0; j < n + 1; j++)
        {
          LinkedList c = new LinkedList();
            Adjazenzlisten.add(j, c);
            LinkedList d = new LinkedList();
            RAdjazenzlisten.add(j, d);

       }

        // node s
        for (int i = 2; i < AnzahlDistrikte + 2; i++)
        {
            Vector Kante = new Vector();
            Vector RKante = new Vector();
            helpstart = 1;
            helpende = i;
            Kantenkap = Distriktres[i - 2];
            Kante.add(new Integer(helpende));
            Kante.add(new Integer(Kantenkap));
            Kante.add(new Integer(0));
            Kante.add(new Integer(m + 1));
            RKante.add(new Integer(helpstart));
```

```
            RKante.add(new Integer(Kantenkap));
            RKante.add(new Integer(0));

            ((LinkedList) Adjazenzlisten.elementAt(helpstart)).
                addLast((Object) Kante);
            ((LinkedList) RAdjazenzlisten.elementAt(helpende)).
                addLast((Object) RKante);
            m++;
        }

        // party and district nodes
        for (int k = 2; k < AnzahlDistrikte + 2; k++)
        {
            for (int i = AnzahlDistrikte + 2; i < n; i++)
            {
            if ((Gewicht[k - 2][i - AnzahlDistrikte - 2]).weight > 0)
                Vector Kante = new Vector();
                  Vector RKante = new Vector();
                  helpstart = k;
                  helpende = i;
                  Kantenkap = 1000000;
                  Kante.add(new Integer(helpende));
                  Kante.add(new Integer(Kantenkap));
                  Kante.add(new Integer(0));
                  Kante.add(new Integer(m + 1));
                  RKante.add(new Integer(helpstart));
                  RKante.add(new Integer(Kantenkap));
                  RKante.add(new Integer(0));


                  ((LinkedList) Adjazenzlisten.elementAt(helpstart)).
                      addLast((Object) Kante);
                  ((LinkedList) RAdjazenzlisten.elementAt(helpende)).
                      addLast((Object) RKante);
                  m++;
                  E++;
            }
            }
        }

        // node t
        for (int i = AnzahlDistrikte + 2; i < n; i++)
        {
            Vector Kante = new Vector();
            Vector RKante = new Vector();
            helpstart = i;
            helpende = n;
            Kantenkap = Parteires[i - AnzahlDistrikte - 2];
```

```
                Kante.add(new Integer(helpende));
                Kante.add(new Integer(Kantenkap));
                Kante.add(new Integer(0));
                Kante.add(new Integer(m + 1));
                RKante.add(new Integer(helpstart));
                RKante.add(new Integer(Kantenkap));
                RKante.add(new Integer(0));


                ((LinkedList) Adjazenzlisten.elementAt(helpstart)).
                        addLast((Object) Kante);
                ((LinkedList) RAdjazenzlisten.elementAt(helpende)).
                        addLast((Object) RKante);
                m++;
            }

            // corrections for impervious multiplier methods
            if (Divisormethode == 0)
            {
              LinkedList help = ((LinkedList) Adjazenzlisten.elementAt(1));
               Iterator it = help.listIterator();
               Vector vec = new Vector();

               for (int i = 2; i < AnzahlDistrikte + 2; i++)
               {
                 vec = (Vector) it.next();
                   int neuesr = ((Integer) vec.elementAt(1)).intValue() -
                        ((LinkedList) Adjazenzlisten.elementAt(i)).size();

                   vec.removeElementAt(1);
                   vec.add(1, new Integer(neuesr));
                   ((Vector) ((LinkedList) RAdjazenzlisten.elementAt(i)).
                        get(0)).removeElementAt(1);
                   ((Vector) ((LinkedList) RAdjazenzlisten.elementAt(i)).
                        get(0)).add(1, new Integer(neuesr));
              }

               for (int i = AnzahlDistrikte + 2; i < n; i++)
               {
                   vec = (Vector) ((LinkedList) Adjazenzlisten.elementAt(i)).
                    get(0);
                   int neuesc = ((Integer) vec.elementAt(1)).intValue() -
                        ((LinkedList) RAdjazenzlisten.elementAt(i)).size();

                   vec.removeElementAt(1);
                   vec.add(1, new Integer(neuesc));
                   ((Vector) ((LinkedList) RAdjazenzlisten.elementAt(n)).
                    get(i - AnzahlDistrikte - 2)).removeElementAt(1);
                   ((Vector) ((LinkedList) RAdjazenzlisten.elementAt(n)).
```

```
                    get(i − AnzahlDistrikte − 2)).add(1, new Integer(neuesc));
                }
            }
        }

    //method getAdjalists
    public Vector getAdjalists()
    {
        return Adjazenzlisten;
    }

    //method getRAdjalists
    public Vector getRAdjalists()
    {
        return RAdjazenzlisten;
    }

    //method getM
    public int getM()
    {
        return m;
    }

    //method getN
    public int getN()
    {
        return n;
    }

    //method getE
        public int getE()
        {
        return E;
    }

}
```

# Bibliography


[1] R.K. Ahuja, T.L. Magnanti and J.B. Orlin. *Network Flows*. Prentice Hall, New Yersey, 1993. Cited on page 14, 21, 50

[2] J.M. Anthonisse. Proportional representation in a regional council. *Centrum voor Wiskunde en Informatica (CWI) Newsletter*, December 1984, 22-29. Cited on page 52

[3] M. Bacharach. *Biproportional Matrices & Input-Output Change*. Cambridge University Press, Cambridge, 1970. Cited on page 5, 39

[4] M.L. Balinski and G. Demange. Algorithms for proportional matrices in reals and integers. *Mathematical Programming*, 45:193-210, 1989. Cited on page 53

[5] M.L. Balinski and G. Demange. An axiomatic approach to proportionality between matrices. *Mathematics of Operations Research*, 14:700-719, 1989. Cited on page 5, 8, 10, 11, 29

[6] M.L. Balinski and H.P. Young. *Fair representation - Meeting the Ideal of One Man, One Vote*. Brookings Institution Press, Washington, D.C., Second Edition, 2001. Cited on page 8

[7] M.L. Balinski and S.T. Rachev. Rounding proportions: Methods of rounding. *Mathematical Scientist*, 22:1-26, 1997. Cited on page 9

[8] K.H. Borgwarth. *Optimierung Operations Research Spieltheorie*. Birkhäuser Verlag, Basel, 2001. Cited on page 14, 18, 19

[9] L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, 1962. Cited on page 17, 18, 19, 31

[10] S. Gier. *Der Balinski-Algorithmus: Biproportionale Zuteilungen im Gleichheitsfall*. Unpublished, 2003. Cited on page 53, 54, 56

[11] F. Pukelsheim and C. Schuhmacher. Das neue Zürcher Zuteilungsverfahren für Parlamentswahlen. *Aktuelle Juristische Praxis*, 5:505-522, 2004. Cited on page 5

[12] F. Pukelsheim. *New Looks at Biproportional Apportionment, Iterative Proportional Fitting, Alternating Scaling, and Cyclic Projections*. Unpublished working paper. September 2004. Cited on page 8, 9

[13] R. Schmidt and S. Seidel. *Staatsorganisationsrecht*. Rolf Schmidt Verlag, Grasberg, 2000. Cited on page 5

# Internet sites:

[14] BAZI homepage and download site of version 2005.07
http://www.uni-augsburg.de/bazi
Cited on page 6

[15] BAZI pseudo-code
http://www.uni-augsburg.de/bazi/pseudoCode.html
Cited on page 46

Erklärung zur Diplomarbeit

Hiermit versichere ich, dass die vorgelegte Diplomarbeit selbständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt wurde. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Augsburg, den _____

_____

Bianca Joas