Algorithmic Aspects of Divisor-Based Biproportional Rounding

Martin Zachariasen

Technical Report no. 06/05
ISSN: 0107-8283

Dept. of Computer Science
University of Copenhagen • Universitetsparken 1
DK-2100 Copenhagen • Denmark

Algorithmic Aspects of Divisor-Based Biproportional Rounding

M. Zachariasen[*]

April 20, 2006

Abstract

Biproportional rounding of a matrix is the problem of assigning values to the elements of a matrix that are proportional to a given input matrix. The assignment should be integral and fulfill a set of row- and column-sum requirements. In a divisor-based method the problem is solved by computing appropriate row- and column-divisors, and by rounding the quotients. The only known divisor-based method that provably solves the problem is the tie-and-transfer algorithm by Balinski, Demange and Rachev. We analyze the complexity of this algorithm, and show that it is pseudo-polynomial. Two different approaches for reducing the complexity to (weakly) polynomial are presented. Finally, we give efficient algorithms for identifying ties, quotient intervals and divisor intervals.

1 Introduction and notation

The classical vector apportionment problem is the following: Given an n-vector $\mathbf{p} = (p_i)$ of non-negative integers (vote numbers) and a positive integer h (house size), compute an *apportionment* vector $\mathbf{x} = (x_i)$ (seat numbers) such that $\sum_i x_i = h$, and such that

$$\frac{x_i}{\sum_j x_j} \approx \frac{p_i}{\sum_j p_j}, \ \ \forall i$$

that is, the vector $\mathbf{x}$ is a proportional rounding of vector $\mathbf{p}$. This problem has a history that goes centuries back — in particular wrt. distributing seats in the House of Representatives according to census data. The historical background and mathematics of the problem is outlined in [6].

[*]Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark (`martinz@diku.dk`).

One class of methods for solving this problem are the *divisor*-based methods[1]. The apportionment problem is solved by choosing a multiplier $\lambda > 0$ and setting $x_i = [\lambda p_i]$, where $[\cdot]$ is the so-called *rounding function*. The multiplier λ should be chosen in such a way that $\sum_i x_i = h$. The rounding function $[\cdot]$ rounds a rational number q to either $\lfloor q \rfloor$ or $\lfloor q \rfloor + 1$. More specifically, the rounding function is based on a *signpost function* $d(x)$ that maps an integer x to a rational number in the interval $[x, x+1]$. If $q < d(\lfloor q \rfloor)$ then $[q] = \lfloor q \rfloor$, and if $q > d(\lfloor q \rfloor)$ then $[q] = \lfloor q \rfloor + 1$. When $q = d(\lfloor q \rfloor)$, then $[q]$ can either be $\lfloor q \rfloor$ or $\lfloor q \rfloor + 1$. Choosing $d(x) = x + 0.5$ results in standard rounding of fractions, where numbers with fractional parts greater than 0.5 are rounded up while numbers with fractional parts less than 0.5 are rounded down. In addition to the above, the signpost function should fulfill some technical conditions [6].

A generalization of the vector apportionment problem is the *matrix* apportionment problem — or *biproportional* rounding problem. Here we are given an $n \times m$ matrix $\mathbf{P} = (p_{ij})$ of nonnegative integers (vote numbers), an n-vector $\mathbf{r} = (r_i)$ of positive integers (row-sum requirements) and an m-vector $\mathbf{c} = (c_j)$ of positive integers (column-sum requirements) such that

$$\sum_i r_i = \sum_j c_j = h$$

We would like to "round" the matrix $\mathbf{P}$ in such a way that we obtain a $n \times m$ matrix $\mathbf{X} = (x_{ij})$ of nonnegative integers (seat numbers) where

$$x_{i*} := \sum_j x_{ij} = r_i, \quad \forall i$$

and

$$x_{*j} := \sum_i x_{ij} = c_j, \quad \forall j$$

Note that the total sum of all elements in $\mathbf{X}$ is h (house size). Similarly to the vector problem, the matrix apportionment problem can be solved by using a divisor-based method. The task is now to compute row multipliers λ_i and column multipliers μ_j, both rational numbers, such that

$$x_{ij} = [\lambda_i p_{ij} \mu_j], \quad \forall (i,j)$$

and such that the row- and column-sum requirements are fulfilled.

Theoretical properties of divisor-based matrix apportionment methods were given by Balinski and Demange [3]. It was shown that these methods have several important and unique properties. Variants of an algorithm, denoted the tie-and-transfer algorithm, for computing the divisors were given by Balinski, Demange and Rachev in [2, 4].

The Zürich City Parliament recently chose to use divisor-based biproportional rounding for the distribution of its seats [16]. In this application, each row i in the vote matrix $\mathbf{P}$ represents a

[1]The more convenient and equivalent *multiplier*-based methods are used throughout this paper.

District Party	WK1+2	WK3	WK4+5	WK6	WK7+8	WK9	WK10	WK11	WK12	Total (r_i)
SP	28518	45541	26673	24092	61738	42044	35259	56547	13215	44
SVP	15305	22060	8174	9676	27906	31559	19557	40144	10248	24
FDP	21833	10450	4536	10919	51252	12060	15267	19744	3066	19
Grüne	12401	17319	10221	8420	25486	9154	9689	12559	2187	14
CVP	7318	8661	4099	4399	14223	11333	8347	14762	4941	10
AL	2413	7418	9086	2304	5483	2465	2539	3623	429	5
EVP	2829	2816	1029	3422	10508	9841	4690	11998	0	6
SD	1651	3173	1406	1106	2454	5333	1490	6226	2078	3
Total (c_j)	12	16	13	10	17	16	12	19	10	125

Table 1: Zürich City Parliament election on February 12, 2006. Vote numbers $\mathbf{P}$, row-sum requirements **r** (parties) and column-sum requirements **c** (districts).

party whose number of seats r_i should be proportional to its vote share. Similarly, each column j represents a district whose number of seats c_j should be proportional to its population. Computing the number of seats for each party and district is now equivalent to biproportional rounding. In Tables 1 and 2 we present vote numbers and resulting seat numbers for the Zürich City Parliament election on February 12, 2006; this election was the first time ever that a divisor-based biproportional method was used for distributing parliament seats. Applications to other parliaments, including comparisons to alternative apportionment methods, are studied in [1, 5, 7, 13, 15, 17]. A software package named BAZI is available for computing vector and matrix apportionments using divisor-based methods [14].

The main focus of this paper is to analyze and improve the complexity of the tie-and-transfer algorithm by Balinski, Demange and Rachev [2, 4]. Our computational model is that arithmetic operations (comparisons, additions, subtractions, divisions and multiplications) take one unit of time. Our interest in the tie-and-transfer algorithm is due to the fact that this is currently the only algorithm that *provably* solves the biproportional rounding problem using a divisor-based method.

As a warm-up, and in order to illustrate some general algorithmic ideas, in Section 2 we first describe an efficient algorithm to compute a divisor-based *vector* apportionment; we show that the running time of the algorithm given by Dorfleitner and Klein [9] can be improved from $O(n^2)$ to $O(n \log n)$ by a clever implementation of the algorithm. We then move on to the matrix problem: In Section 3 we give a detailed description of the tie-and-transfer algorithm; some new algorithmic details and an analysis of the running time is given. In Section 4 we present techniques to improve the running time of the tie-and-transfer algorithm from pseudo-polynomial to (weakly) polynomial running time. Section 5 presents new algorithms to identify ties in a computed apportionment, and to compute quotient and multiplier/divisor intervals. Concluding remarks are given in Section 6.

District Party	WK1+2	WK3	WK4+5	WK6	WK7+8	WK9	WK10	WK11	WK12	Divisor $(1/\lambda_i)$
SP	4	7	5	4	5	6	4	6	3	1.006
SVP	2	3	2	1	2	4	3	4	3	1.002
FDP	3	1	1	2	5	2	2	2	1	1.010
Grüne	2	3	2	1	2	1	1	1	1	0.970
CVP	1	1	1	1	1	1	1	2	1	1.000
AL	0	1	2	0	1	0	0	1	0	0.800
EVP	0	0	0	1	1	1	1	2	0	0.880
SD	0	0	0	0	0	1	0	1	1	1.000
Divisor $(1/\mu_j)$	7000	6900	5000	6600	11200	7580	7800	9000	4000	

Table 2: Zürich City Parliament election on February 12, 2006. Seat numbers $\mathbf{X}$, row-divisors and column-divisors. Standard rounding is used, and the divisors are those that were published by the Zürich City administration (`http://daten.wahlen.stzh.ch/`). As an example, the number of seats for party SP in district WK6 is computed as $[24092/(1.006 \cdot 6600)] = 4$.

2 Improved algorithm for proportional rounding

Recall the divisor-based proportional rounding problem: Given an n-vector $\mathbf{p} = (p_i)$ of non-negative integers and a positive integer h, find a multiplier $\lambda > 0$ such $\sum_i x_i = h$, where $x_i = [\lambda p_i]$ for $i = 1, \ldots, n$. For illustrative purposes, we will refer to p_i (resp. x_i) as the number of votes (resp. seats) given to party i.

2.1 Preprocessing

Let $d(\cdot)$ be the signpost function associated with the rounding function $[\cdot]$. For any rounding function we have that $[0] = 0$, so $p_i = 0$ implies that $x_i = 0$. Another special case occurs when the signpost function fulfills $d(0) = 0$. This means that any party i where $p_i > 0$ should have at least one seat ($x_i \geq 1$). Let κ be the number of parties where $p_i > 0$. If $\kappa > h$, then the problem is clearly infeasible. In the following we therefore assume that $p_i > 0$ for each of the n parties (and $n \leq h$ if $d(0) = 0$).

2.2 Quadratic time algorithm

The algorithm by Dorfleitner and Klein [9] proceeds (implicitly) by iteratively updating the multiplier λ until the seat numbers add up correctly. The initial guess for the multiplier is $\lambda = h/\sum_i p_i$. For this so-called *canonical* multiplier we have $\sum_i \lambda p_i = h$, that is, the *quotients* add up correctly. The initial apportionment is $x_i = [\lambda p_i]$ for all parties i. If $\sum_i x_i = h$, then a valid multiplier has been found and the algorithm terminates. So assume that $\sum_i x_i \neq h$. By definition $|x_i - \lambda p_i| \leq 1$, so $|\sum_i x_i - \sum_i \lambda p_i| = |\sum_i x_i - h| \leq n$. The initial error (or discrepancy) $\Delta = |\sum_i x_i - h|$ for the canonical multiplier is therefore at most n. The expected error of the canonical and other multipliers was studied by Happacher and Pukelsheim [10, 11]; the canonical multiplier has zero expected error if standard rounding is used.

Assume first that $\sum_i x_i < h$. This means that we should increase the multiplier λ. We simulate the process of increasing λ by iteratively updating the apportionment. Recall that we have $d(x_i - 1) \leq \lambda p_i \leq d(x_i)$ for all parties i. So in the initial apportionment we have $\lambda \leq d(x_i)/p_i$ for all parties i. Consider a party k for which $d(x_k)/p_k$ is *minimum*. This party should be the first to receive an additional seat as we increase λ, so we set $x_k = x_k + 1$ (note that as a consequence of this update, $d(x_k)/p_k$ increases). By repeatedly picking a party k where $d(x_k)/p_k$ is minimum and setting $x_k = x_k + 1$, after at most Δ updates, we have $\sum_i x_i = h$.

A similar algorithm is used when $\sum_i x_i > h$. Here we pick a party k where $d(x_k - 1)/p_k$ is *maximum*, and set $x_k = x_k - 1$. Finding the minimum/maximum in each iteration takes $O(n)$ time, resulting in a total running time of $O(n + \Delta n)$ — which is $O(n^2)$ since $\Delta \leq n$. Handling of ties is described in [9].

2.3 A faster implementation

The running time of the quadratic time algorithm can be improved to $O(n \log n)$ as follows. After the initial multiplier guess, assume that $\sum_i x_i < h$. Create a *priority queue* Q, and append all parties i with priority $d(x_i)/p_i$ to this queue. Finding a party k with minimum $d(x_k)/p_k$ is equivalent to extracting the minimum element from Q, and this can be accomplished in $O(\log n)$ time since Q contains n elements [8]. After the update $x_k = x_k + 1$, the party k is reinserted into Q with the new priority $d(x_k)/p_k$, an operation that also can be performed in $O(\log n)$ time. Thus, in total each iteration takes $O(\log n)$ time.

When $\sum_i x_i > h$ after the initial multiplier guess, we use a priority queue with priorities $-d(x_k - 1)/p_k$, such that a minimum element in the queue corresponds to a maximum value of $d(x_k - 1)/p_k$. In conclusion, we have the following theorem:

Theorem 2.1 *Computing a divisor-based apportionment for an n-party problem can be accomplished in time $O(n + \Delta \log n)$, where Δ is the error (or discrepancy) of the canonical multiplier. Since $\Delta \leq n$, the running time is $O(n \log n)$.*

3 Tie-and-transfer algorithm

In the biproportional rounding problem the task is to compute an $n \times m$ matrix $\mathbf{X} = (x_{ij})$ that fulfills the given row-sum requirements $x_{i*} = r_i$ and column-sum requirements $x_{*j} = c_j$. In a divisor-based algorithm for solving the problem, we should compute row multipliers $\lambda_i > 0$ and column multipliers $\mu_j > 0$, such that $x_{ij} = [\lambda_i p_{ij} \mu_j]$, where $\mathbf{P} = (p_{ij})$ is the given (vote) matrix. If no solution exists, the algorithm should report this.

The tie-and-transfer algorithm [2, 4] is currently the only known algorithm that correctly solves the problem, that is, for *every* input it provably computes a set of valid multipliers or reports that no solution exists. Heuristics that work well in practice (but that can fail) are discussed in Section 4.

The purpose of this section is to give the first comprehensive running time analysis of the tie-and-transfer algorithm. In order to do so, some additional algorithmic details, not present in the original papers, are given. As a side-effect, our detailed presentation of the algorithm makes its implementation relatively straightforward.

The tie-and-transfer algorithm iteratively updates the multipliers λ_i and μ_j until the rows and columns add up correctly. More precisely, the error function

$$\Delta(\mathbf{X}) = \frac{1}{2}\sum_i |x_{i*} - r_i| + \frac{1}{2}\sum_j |x_{*j} - c_j|$$

is iteratively minimized by making so-called *transfers*. Each transfer decreases the error function by 1. Between each transfer the algorithm creates so-called *ties* by updating the multipliers in a synchronized manner. These ties eventually make a new transfer possible, and the algorithm iterates. When the error has dropped to zero, a valid set of multipliers has been found, and the algorithm terminates.

3.1 Preprocessing

In order to simplify the description of the algorithm, we assume that the problem is not trivially infeasible for the case where $d(0) = 0$. Let κ_i be the number of positive elements of $\mathbf{P}$ in row i. If $\kappa_i > r_i$ for any row i, then the problem is clearly infeasible. Similarly, let ν_j be the number of positive elements of $\mathbf{P}$ in column j. If $\nu_j > c_j$ for any column j, then the problem is infeasible.

3.2 Initialization

The multipliers are initially chosen by setting all row multipliers $\lambda_i = 1$, and by choosing the column multipliers such that the columns add up correctly, i.e. $x_{*j} = c_j$. That is, for each column j in $\mathbf{P} = (p_{ij})$ we solve the vector apportionment problem with vote numbers $p_{1j}, p_{2j}, \ldots, p_{nj}$ and required total c_j. All subsequent changes to the multipliers and to the apportionment are made in such a way that the columns continuously add up correctly. After initialization, the error function therefore reduces to

$$\Delta(\mathbf{X}) = \frac{1}{2}\sum_i |x_{i*} - r_i|$$

3.3 Row/column graph

Let I be the set of rows and J the set of columns in $\mathbf{P}$. For a given set of multipliers and (not necessarily feasible) apportionment we define the so-called *row/column graph* $G = (V, E)$ as follows: The graph G is a directed bipartite graph with vertex set $V = I \cup J$. The set of edges E is the union of two sets:

1) For each $i \in I$ and $j \in J$ where $p_{ij} > 0$ and $\lambda_i p_{ij} \mu_j = d(x_{ij})$ we have an edge $(i, j) \in E$. This means that x_{ij} may be *rounded up* without changing the multipliers. (Recall that we have $d(x_{ij} - 1) \leq \lambda_i p_{ij} \mu_j \leq d(x_{ij})$ for all i, j.)

2) For each $i \in I$ and $j \in J$ where $p_{ij} > 0$ and $\lambda_i p_{ij} \mu_j = d(x_{ij} - 1)$ we have an edge $(j, i) \in E$. This means that x_{ij} may be *rounded down* without changing the multipliers.

Informally, we have an edge (i, j) out of row i for each column j where x_{ij} can be rounded up. Similarly, we have an edge (j, i) out of column j for each row i where x_{ij} can be rounded down. Note that for any pair of vertices i, j, there is at most one directed edge that connects them.

3.4 Transfers

Assume that G contains a simple path p consisting of the edges $(i_1, j_1), (j_1, i_2), (i_2, j_2) \ldots, (j_{k-1}, i_k)$. The path p defines a *transfer* by rounding up $x_{i_1 j_1}$, rounding down $x_{i_2 j_1}$, rounding up $x_{i_2 j_2}$ etc. and finally rounding down $x_{i_k j_{k-1}}$. The net effect of this transfer is that it increases the i_1'th row sum by 1 and decreases the i_k'th row sum by 1. No other row sums and no column sums are affected by the transfer. If $i_1 = i_k$ the result is a *cyclic* transfer, where no row or column sums are changed.

Let I^- be the set of row vertices i where $x_{i*} < r_i$. Similarly, let I^+ be the set of row vertices i where $x_{i*} > r_i$. The main idea of the algorithm is to search for a path in G from a row vertex $i^- \in I^-$ to a row vertex $i^+ \in I^+$ (such a vertex i^+ is *reachable* from i^-). The corresponding transfer would clearly decrease the error function by exactly 1.

3.5 Searching for transfers

Algorithm 1 finds all vertices in G that are reachable from the vertices in a given set $L \subseteq V$. The algorithm performs a breadth-first-search (BFS) [8] in G from the set L. The advantage of using BFS is that its running time is bounded by $O(nm)$ and that it in addition finds shortest possible paths (in the number of edges) from the labeled vertices to the reachable vertices. Therefore, all computed paths are simple, that is, contain no repeated vertices.

We run Algorithm 1 with input $L = I^-$, and the result is that all reachable vertices are added to L. Upon return, if $L \cap I^+ \neq \emptyset$ then a transfer exists that reduces the error function; the transfer

Algorithm 1 Breadth-first search (BFS) in row/column graph

```
Require: $L$ is the set of initially labeled vertices
1. $Q = L$   // Queue of labeled but not yet processed vertices
2. while $Q \neq \emptyset$ do
3.   DEQUEUE$(Q, u)$   // Extract next vertex from $Q$
4.   if $u$ is a row vertex then
5.     $i$ = ROW$(u)$
6.     for $j = 1$ to $m$ do
7.       if $j \notin L$ and $p_{ij} > 0$ and $\lambda_i p_{ij} \mu_j = d(x_{ij})$ then
8.         // Edge $(i, j)$ where $j$ is not labeled. Append $j$ to $Q$ and $L$
9.         ENQUEUE$(Q, j)$   // Append to end of $Q$
10.        $L = L \cup \{j\}$
11.        $\pi(j) = i$   // Predecessor of $j$ is $i$
12.  else
13.    $j$ = COLUMN$(u)$
14.    for $i = 1$ to $n$ do
15.      if $i \notin L$ and $p_{ij} > 0$ and $\lambda_i p_{ij} \mu_j = d(x_{ij} - 1)$ then
16.        // Edge $(j, i)$ where $i$ is not labeled. Append $i$ to $Q$ and $L$
17.        ENQUEUE$(Q, i)$   // Append to end of $Q$
18.        $L = L \cup \{i\}$
19.        $\pi(i) = j$   // Predecessor of $i$ is $j$
20. return $L$   // Return final set of labeled vertices
```

can be identified by following the predecessor pointers π from any vertex $i^+ \in L \cap I^+$ back to a vertex in I^-. If $L \cap I^+ = \emptyset$ then no such transfer exists, and we need to update the multipliers.

3.6 Updating the multipliers

Let I_L and J_L, where $L = I_L \cup J_L$, be the set of labeled rows and columns, respectively, after running Algorithm 1. Also, let $\overline{I_L} = I \setminus I_L$ and $\overline{J_L} = J \setminus J_L$. If no vertices in I^+ are reachable from the vertices in I^-, we may depict the situation as in Figure 1.

Our goal is now to compute a factor $\delta > 1$, such that we can multiply λ_i by δ for all $i \in I_L$ and divide μ_j by δ for all $j \in J_L$. This should be done in such a way that the current apportionment remains feasible. (Note that $\lambda_i p_{ij} \mu_j$ is unchanged for all elements where $(i, j) \in I_L \times J_L$, that is, when *both* the row and column is labeled.) Algorithm 2 computes the largest possible such δ by iterating over all edges from a labeled to an unlabeled vertex in G. Assume first that after running

	J_L	$\overline{J_L}$
$I^- \subseteq I_L$	Rows and columns are labeled	Only rows are labeled $\lambda_i p_{ij} \mu_j < d(x_{ij})$
$I^+ \subseteq \overline{I_L}$	Only columns are labeled $d(x_{ij} - 1) < \lambda_i p_{ij} \mu_j$	Neither rows nor columns are labeled

Figure 1: Illustration of the situation when no transfer can be found by Algorithm 1. The figure depicts the (reordered) apportionment matrix where labeled rows and columns appear first. The idea of the algorithm is to multiply all row multipliers in the "upper" half of the matrix by δ and at the same time divide all column multipliers in the "left" half by the same multiplier. This will keep the quotients in the upper left corner unchanged, that is, where both rows and columns are labeled.

Algorithm 2 Compute δ for updating multipliers

```
Require: $L = I_L \cup J_L$ is the set of labeled rows and columns
1.  $\delta = \infty$
2.  for $i = 1$ to $n$ do
3.    for $j = 1$ to $m$ do
4.      if $i \in I_L$ and $j \notin J_L$ and $p_{ij} > 0$ then
5.        // For this element we must have $\lambda_i p_{ij} \mu_j < d(x_{ij})$
6.        $\delta = \min(\delta, \frac{d(x_{ij})}{\lambda_i p_{ij} \mu_j})$
7.      if $i \notin I_L$ and $j \in J_L$ and $p_{ij} > 0$ then
8.        // For this element we must have $d(x_{ij} - 1) < \lambda_i p_{ij} \mu_j$
9.        if $d(x_{ij} - 1) > 0$ then
10.         $\delta = \min(\delta, \frac{\lambda_i p_{ij} \mu_j}{d(x_{ij} - 1)})$
11. return $\delta$   // Return final $\delta$ (which may be equal to $\infty$)
```

Algorithm 2 we have that $\delta \neq \infty$. Then either

$$(\delta \lambda_i) p_{ij} \mu_j = d(x_{ij}) \text{ and } p_{ij} > 0 \quad \text{for some} \quad (i, j) \in I_L \times \overline{J_L}$$

or

$$\lambda_i p_{ij} (\mu_j / \delta) = d(x_{ij} - 1) \text{ and } p_{ij} > 0 \quad \text{for some} \quad (i, j) \in \overline{I_L} \times J_L$$

In other words, after updating the multipliers, at least one more row or column can be labeled by the BFS algorithm. (All previously labeled rows and columns remain labeled.)

If Algorithm 2 returns $\delta = \infty$, then the problem instance is *infeasible*. The reason is that in this case we must have (when d(0) > 0) that

$$p_{ij} = 0 \quad (\text{which implies that } x_{ij} = 0) \quad \text{for all} \quad (i, j) \in I_L \times \overline{J_L}$$

and

$$d(x_{ij} - 1) = 0 \quad (\text{which implies that } x_{ij} = 0) \quad \text{for all} \quad (i, j) \in \overline{I_L} \times J_L$$

Therefore,

$$\sum_{j \in J_L} c_j = \sum_{(i,j) \in I_L \times J_L} x_{ij} < \sum_{i \in I_L} r_i$$

and thus there is no way that the labeled rows can achieve the required sum. The labeled rows and columns form a certificate of the infeasibility of the problem instance. A similar argument applies for the case when $d(0) = 0$ [4].

3.7 Running time

What is the running time of the Balinski and Rachev algorithm? The initialization takes $O(nm \log n)$ time since we solve m vector problems, each of size n. Searching the row/column graph takes $O(nm)$ time. Computing δ and updating the multipliers also takes $O(nm)$ time. Each time the multipliers are updated, at least one more row or column can be labeled. Thus after at most $n + m$ multiplier-updates, a transfer is found — resulting in a worst-case running time of $O(nm(n + m))$ to find a transfer.

The total row-sum error in the first iteration of the algorithm is at most h. Thus at most h transfers are made throughout the execution of the algorithm, giving a total running time of $O(nm(n + m)h)$. Combining this analysis with the correctness arguments from [4], we get:

Theorem 3.1 *The tie-and-transfer algorithm correctly computes a feasible apportionment* $\mathbf{X}$ *and corresponding multipliers* λ *and* μ *in time* $O(nm(n+m)h)$*, where* h *is the total sum of the rounded* $n \times m$ *matrix. When* $m = O(n)$*, the running time is* $O(n^3h)$*.*

4 Improved algorithm for biproportional rounding

In this section we present two different techniques to improve the running time of the tie-and-transfer algorithm. In fact, we do not improve the running time directly, but instead we use another algorithm to find a provably good guess for the multipliers *before* invoking the tie-and-transfer algorithm. The guess guarantees that the initial error $\Delta(\mathbf{X})$, as defined in Section 3, in the tie-and-transfer algorithm is sufficiently small.

We present two different approaches. The first one is based on solving the *continuous* version of the matrix apportionment problem (Section 4.1), while the other is based on using the tie-and-transfer algorithm recursively with scaled row- and column-sum requirements (Section 4.2). In both cases we reduce the running time dependency of h from $O(h)$ to $O(\log h)$. Basically this means that we improve the running time from pseudo-polynomial to (weakly) polynomial running time. The idea of using a so-called *front-end* for the tie-and-transfer algorithm was already discussed in Balinski and Demange [2]. However, the impact on the worst-case running time has not been studied before. In Section 4.3 we discuss various practical considerations concerning combinations of front- and back-ends.

4.1 Continuous front-end

The continuous version of the problem is the following: Given an $n \times m$ matrix $\mathbf{P} = (p_{ij})$, row-sum requirements r_i and column-sum requirements c_j, compute row multipliers λ_i and column multipliers μ_j, such that the matrix $\mathbf{X} = (x_{ij})$ given by $x_{ij} = \lambda_i p_{ij} \mu_j$ fulfills the given row- and column-sum requirements. The only difference compared to the biproportional rounding problem — or the discrete version of the problem — is the definition of x_{ij}, where no rounding of the value $\lambda_i p_{ij} \mu_j$ takes place in the continuous case.

One distinctive difference between the discrete and continuous versions of the problem is that the continuous problem can be scaled in a preprocessing phase without changing the solution to the problem: Multiplying the entries in $\mathbf{P}$ with any factor $\gamma > 0$, and similarly multiplying the row- and column-sum requirements with the same factor, results in a problem that has the same solution. Therefore, one may assume that, say, $\sum_i r_i = \sum_j c_j = n$. This is something that is *not* possible in the discrete case. Furthermore, while in the discrete case the row- and column-sum requirements should be met exactly, this is not practically feasible nor necessary in the continuous case. As a consequence, in the continuous problem we are also given a parameter $\epsilon > 0$ that specifies the maximum error that can be accepted, that is, we seek an apportionment $\mathbf{X}$ such that $\Delta(\mathbf{X}) \leq \epsilon$.

The continuous problem is usually denoted *matrix scaling*, and it has numerous applications in e.g. statistics, numerical analysis and operations research. When solved inexactly, the problem is denoted *ϵ-scaling*. From a mathematical point of view, the problem is well-studied, but the literature on algorithms for solving the problem is relatively sparse. The only strongly polynomial algorithm that exists for solving the ϵ-scaling problem was given by Linial et al [12]. Here strongly polynomial means that the running time is independent of the (relative) sizes of the numbers in the matrix $\mathbf{P}$:

Lemma 4.1 *The ϵ-scaling problem for an $n \times m$ matrix where $m = O(n)$ and $\sum_i r_i = \sum_j c_j = h$ can be solved in time $O(n^7 \log(n) \log(h/\epsilon))$.*

We will now use this algorithm to solve the biproportional rounding problem. First we solve the corresponding ϵ-scaling problem with $\epsilon = 1$. Let λ, μ and $\mathbf{X} = (x_{ij})$ be the resulting multipliers and apportionment. We construct a new apportionment $\mathbf{X}' = (x'_{ij})$ by setting $x'_{ij} = [x_{ij}] = [\lambda_i p_{ij} \mu_j]$. Clearly $|\Delta(\mathbf{X}') - \Delta(\mathbf{X})| \leq nm$. Therefore, by using λ and μ as guesses on the multipliers, the initial error in the tie-and-transfer algorithm is $O(nm)$, and the total running time of the tie-and-transfer algorithm therefore becomes $O((nm)^2(n+m))$ — which is $O(n^5)$ when $m = O(n)$. Since the running time for solving the ϵ-scaling problem dominates the running time of the tie-and-transfer algorithm, we have the following:

Theorem 4.2 *The biproportional rounding problem for an $n \times m$ matrix where $m = O(n)$ and with total sum h can be solved in time $O(n^7 \log(n) \log(h))$.*

4.2 Tie-and-transfer front-end

In this section we show that we can change the number of tie-and-transfer iterations in the worst-case from h to $O(nm \log h)$. The idea is to obtain good guesses on the multipliers by solving a series of problems with smaller marginal requirements. The algorithm works *recursively* as follows.

Let $\mathcal{I} = (\mathbf{P}, \mathbf{r}, \mathbf{c})$ be the given problem instance. First we pick a small integer constant $b \geq 2$, say, $b = 10$. If $h < (n+1)(m+1)b$ then we solve $\mathcal{I}$ directly by using the tie-and-transfer algorithm — performing at most $(n+1)(m+1)b$ transfers. Otherwise, we construct a new $n \times m$ problem instance $\mathcal{I}' = (\mathbf{P}', \mathbf{r}', \mathbf{c}')$ with total sum $h' = [h/b]$. The required row sum for row i in $\mathcal{I}'$ is either $r'_i = \lfloor r_i/b \rfloor$ or $r'_i = \lceil r_i/b \rceil$ such that $\sum_i r'_i = h'$. Any quota apportionment method can be used to compute the row requirements. The required column sums for $\mathcal{I}'$ are computed in a similar fashion. Finally, we have $\mathbf{P}' = \mathbf{P}$.

Now assume that we have solved problem instance $\mathcal{I}'$ recursively. Let λ' and μ' be the computed feasible multipliers for $\mathcal{I}'$. Since the marginal sums in $\mathcal{I}$ are approximately a factor b larger than in $\mathcal{I}'$, we compute the following guesses on the multipliers for problem instance $\mathcal{I}$:

$$\lambda_i = b\,\lambda'_i, \quad \forall i$$

$$\mu_j = \mu'_j, \quad \forall j$$

Lemma 4.3 *The initial error in problem instance $\mathcal{I}$ using multipliers λ and μ, as computed above, is at most $(n+1)(m+1)b$.*

Proof. Let $x'_{ij} = [\lambda'_i\, p_{ij}\, \mu'_j]$ be the apportionment obtained for problem instance $\mathcal{I}'$. Clearly we can multiply this apportionment and the required row and column sums for $\mathcal{I}'$ by a factor b and obtain a scaled apportionment that satisfies the scaled marginal sums:

$$\sum_j b\,x'_{ij} = b\,r'_i, \ \forall i$$

$$\sum_i b\,x'_{ij} = b\,c'_j, \ \forall j$$

Now consider the initial solution to problem instance $\mathcal{I}$. We have

$$[\lambda_i\, p_{ij} \mu_j] = [b\,\lambda'_i\, p_{ij}\, \mu'_j]$$

which is at most b units away from $b\,x'_{ij} = b\,[\lambda'_i\, p_{ij}\, \mu'_j]$. (This holds since for any integer b and rational number q we have that both $b[q]$ and $[bq]$ are in the interval from $b\lfloor q \rfloor$ to $b\lceil q \rceil$.) Similarly we have that r_i is at most b units away from $b\,r'_i$. In total the error made in row i is bounded by $(m+1)b$. The total error of all rows is therefore at most $n(m+1)b$. Similarly, the total error of all columns is bounded by $(n+1)mb$. Adding these two numbers and dividing by 2, we obtain an upper bound on the error function which is bounded by $(n+1)(m+1)b$. ∎

The number of problem instances solved in total is $O(\log h)$, since the marginal sums are made a factor b smaller each time. Therefore, the total number of transfers performed when solving the problem instances is at most $(n+1)(m+1)b \cdot O(\log h)$ which is $O(nm \log h)$. Combining this with the running time to perform a single transfer, we get:

Theorem 4.4 *The biproportional rounding problem for an $n \times m$ matrix with total sum h can be solved in time $O((nm)^2(n+m)\log h)$. When $m = O(n)$, the running time is $O(n^5 \log h)$.*

4.3 Practical combinations of front- and back-ends

Although the algorithms presented in the previous sections have the best known running time bounds, other algorithms may work better in practice. One approach that is fast is practice is the so-called *iterative proportional fitting (IPF)* algorithm. This algorithm solves the continuous problem by alternating between scaling the rows and columns to meet the required row- and column-sums. The discrete version of the IPF algorithm is named *alternating scaling (AS)* [15].

An experimental study of various combinations of front- and back-ends was recently undertaken by Maier [13]. Using either AS or IPF as front-ends to the tie-and-transfer algorithm gives a significant speed-up in practice. Using IPF and a switching barrier of 10 (i.e., switching to tie-and-transfer when the error drops below 10) gave the best results on both realistic and artificial problem instances. It is still an open question whether the simple recursive algorithm presented in Section 4.2 is competitive with these approaches in practice.

5 Sensitivity analysis

A central result in the theory of divisor-based biproportional rounding is that the apportionment $\mathbf{X}$ (if it exists) is *unique up to ties* [4]:

Lemma 5.1 *Suppose $\mathbf{X}$ is a feasible apportionment with multipliers λ and μ. Let $\mathbf{X}' \neq \mathbf{X}$ be another apportionment for the same problem. Then multipliers λ and μ can also be used for $\mathbf{X}'$, that is,*

$$x'_{ij} = [\lambda_i p_{ij} \mu_j], \ \ \forall (i,j)$$

As a consequence, if $x'_{ij} \neq x_{ij}$ for some element (i,j), then either $x'_{ij} = x_{ij} - 1$ or $x'_{ij} = x_{ij} + 1$, and this is a result of a tie in the rounding function.

In this section we first present an algorithm that identifies all ties in a computed apportionment (Section 5.1). Then we extend this algorithm to compute feasible intervals for quotients (Section 5.2) and multipliers (Section 5.3).

5.1 Identifying ties

Recall that we have

$$d(x_{ij}-1) \le \lambda_i p_{ij} \mu_j \le d(x_{ij}), \ \ \forall (i,j)$$

for a feasible apportionment $\mathbf{X}$ with multipliers λ and μ. Furthermore, if

$$d(x_{ij}-1) < \lambda_i p_{ij} \mu_j < d(x_{ij})$$

for some element (i, j) then Lemma 5.1 shows that x_{ij} is in fact the *unique* solution to the problem for element (i, j). However, $d(x_{ij}-1) = \lambda_i p_{ij} \mu_j$ or $\lambda_i p_{ij} \mu_j = d(x_{ij})$ does *not* imply that element (i, j) necessarily is a tie, that is, can be rounded up or down. (Note that an element (i, j) can only be a tie if $p_{ij} > 0$, since $p_{ij} = 0$ implies $x_{ij} = 0$.)

Consider an element (i, j) where $d(x_{ij}-1) = \lambda_i p_{ij} \mu_j$ (and $p_{ij} > 0$). Since $\lambda_i p_{ij} \mu_j > 0$ we have $d(x_{ij}-1) > 0$ which means that $x_{ij} > 0$ since $d(-1) = 0$. We would like to decide if element (i, j) is a tie or not. This is done by searching for a *cyclic transfer* in the row/column graph (see Section 3.3) that includes element (i, j). If such a transfer is found, then element (i, j) clearly *is* a tie. If no cyclic transfer can be found we will show how the multipliers λ and μ can be updated to a set of new multipliers λ' and μ' in such a way that

1. the apportionment $\mathbf{X}$ remains feasible for the new multipliers λ' and μ'.
2. $d(x_{ij}-1) < \lambda'_i p_{ij} \mu'_j$ which proves that element (i, j) is *not* a tie,
3. no new equalities of the form $d(x_{i'j'}-1) = \lambda'_{i'} p_{i'j'} \mu'_{j'}$ or $\lambda'_{i'} p_{i'j'} \mu'_{j'} = d(x_{i'j'})$ are introduced for any element (i', j').

Algorithm 3 decides whether an element (i, j) where $d(x_{ij}-1) = \lambda_i p_{ij} \mu_j$ is a tie or not. Upon return from the algorithm we either still have $d(x_{ij}-1) = \lambda'_i p_{ij} \mu'_j$ which means that element (i, j) is a tie, or we have $d(x_{ij}-1) < \lambda'_i p_{ij} \mu'_j$, proving that element (i, j) is not a tie.

Lemma 5.2 *Algorithm 3 correctly decides whether an element (i, j), where $d(x_{ij}-1) = \lambda_i p_{ij} \mu_j$, is a tie or not.*

Proof. The main idea of the algorithm is to search for a path in the row/column graph G from row i to column j (lines 1–3). If such a path $(i, j_1), (j_1, i_1), (i_1, j_2), \ldots, (i_k, j)$ exists (lines 5–8), then we may round down (i, j), round up (i, j_1), round down (i_1, j_1) etc. until we finally round up (i_k, j). Thus there is a cyclic transfer in the row/column graph, and (i, j) is a tie.

On the other hand, if no such path exists (lines 10–16), we use Algorithm 2 from Section 3.6 to compute a largest possible $\delta > 1$ that can be used to update the multipliers — as in the tie-and-transfer algorithm. We show that δ is well-defined and in fact $1 < \delta < \infty$. That $\delta > 1$ follows from the arguments given for the original tie-and-transfer algorithm (Section 3.6). Furthermore, row i is labeled, but column j is not, so clearly $\delta \leq d(x_{ij})/(\lambda_i p_{ij} \mu_j) < \infty$.

The computed δ value is not used directly to update the multipliers, since this would introduce new equalities of the the form $d(x_{i'j'} - 1) = \lambda'_{i'} p_{i'j'} \mu'_{j'}$ or $\lambda'_{i'} p_{i'j'} \mu'_{j'} = d(x_{i'j'})$ for some element (i', j'). Instead we compute the average of 1 and δ, i.e., set $\delta = (\delta + 1)/2$ (line 13). When we make the update (lines 14–16), no new equalities are therefore introduced by the new multipliers λ' and μ'. Since $\delta > 1$, we have $d(x_{ij} - 1) < \lambda'_i p_{ij} \mu'_j$, so we have proved that element (i, j) is *not* a tie. ∎

Algorithm 4 similarly handles the case where $\lambda_i p_{ij} \mu_j = d(x_{ij})$. In this case we search for a cyclic transfer starting in *column* j. If row i can be reached from column j in the row/column graph G, then the element (i, j) is a tie — otherwise we update the multipliers proving that (i, j) is not a tie. Again we note that δ is well-defined, but in this case we may have that $\delta = \infty$; here we may choose any value between 1 and ∞, and in the algorithm we choose $\delta = 2$ (line 16). Therefore, we get the following:

Lemma 5.3 *Algorithm 4 correctly decides whether an element (i, j), where $\lambda_i p_{ij} \mu_j = d(x_{ij})$, is a tie or not.*

Since Algorithms 3 and 4 run in $O(nm)$ time, and we have nm elements in total, we obtain:

Theorem 5.4 *Given a feasible apportionment $\mathbf{X}$ and a set of multipliers λ and μ, by running Algorithm 3 for all elements where $d(x_{ij} - 1) = \lambda_i p_{ij} \mu_j$, and Algorithm 4 for all elements where $\lambda_i p_{ij} \mu_j = d(x_{ij})$, we obtain a set of multipliers λ' and μ', such that*

$$d(x_{ij} - 1) < \lambda'_i p_{ij} \mu'_j < d(x_{ij})$$

if and only if element (i, j) is not *a tie. The running time to compute λ' and μ' is $O((nm)^2)$.*

5.2 Computing feasible intervals for quotients

Although the apportionment for a given problem instance essentially is unique, there is an infinite number of feasible multipliers. For any $\Delta > 0$ we have that $\lambda_i p_{ij} \mu_j = (\Delta \lambda_i) p_{ij} (\mu_j / \Delta)$ so we may perform any simultaneous scaling of the row and column multipliers. In addition, we may often change one or more multipliers without affecting the apportionment. One useful approach is to compute lower bounds L_{ij} and upper bounds U_{ij} on the quotients $\lambda_i p_{ij} \mu_j$, such that for any feasible apportionment $\mathbf{X}$ and multipliers λ and μ we have that

$$L_{ij} \leq \lambda_i p_{ij} \mu_j \leq U_{ij}$$

Algorithm 3 Test if an element (i, j) can be rounded *down*.

```
Require: Element $(i, j)$ where $d(x_{ij} - 1) = \lambda_i p_{ij} \mu_j$ and $p_{ij} > 0$
 1. $L = \{i\}$
 2. Run BFS with input $L$ on row/column graph $G$ (Algorithm 1)
 3. // Let $L = I_L \cup J_L$ be the output of BFS
 4. if $j \in J_L$ then
 5.     // Element $(i, j)$ is part of a cyclic transfer and therefore a tie!
 6.     // Compute new (unchanged) multipliers
 7.     $\lambda'_i = \lambda_i, \;\; i \in I$
 8.     $\mu'_j = \mu_j, \;\; j \in J$
 9. else
10.     // Update multipliers to prove that element $(i, j)$ is not a tie
11.     Compute $\delta$ for $G$ and $L$ (Algorithm 2)
12.     // We now have that $1 < \delta < \infty$
13.     $\delta = (\delta + 1)/2$   // Change to average between 1 and $\delta$
14.     // Compute new (updated) multipliers
15.     $\lambda'_i = \begin{cases} \delta\lambda_i & i \in I_L \\ \lambda_i & i \notin I_L \end{cases}$
16.     $\mu'_j = \begin{cases} \mu_j/\delta & j \in J_L \\ \mu_j & j \notin J_L \end{cases}$
17. return $(\lambda', \mu')$ // Updated multipliers
```

Algorithm 4 Test if an element (i, j) can be rounded *up*.

```
Require: Element $(i, j)$ where $\lambda_i p_{ij} \mu_j = d(x_{ij})$ and $p_{ij} > 0$
 1. $L = \{j\}$
 2. Run BFS with input $L$ on row/column graph $G$ (Algorithm 1)
 3. // Let $L = I_L \cup J_L$ be the output of BFS
 4. if $i \in I_L$ then
 5.     // Element $(i, j)$ is part of a cyclic transfer and therefore a tie!
 6.     // Compute new (unchanged) multipliers
 7.     $\lambda'_i = \lambda_i,\ \ i \in I$
 8.     $\mu'_j = \mu_j,\ \ j \in J$
 9. else
10.     // Update multipliers to prove that element $(i, j)$ is not a tie
11.     Compute $\delta$ for $G$ and $L$ (Algorithm 2)
12.     // We now have that $1 < \delta \leq \infty$
13.     if $\delta < \infty$ then
14.         $\delta = (\delta + 1)/2$   // Change to average between 1 and $\delta$
15.     else
16.         $\delta = 2$   // We can choose any value greater than 1
17.     // Compute new (updated) multipliers
18.     $\lambda'_i = \begin{cases} \delta\lambda_i & i \in I_L \\ \lambda_i & i \notin I_L \end{cases}$
19.     $\mu'_j = \begin{cases} \mu_j/\delta & j \in J_L \\ \mu_j & j \notin J_L \end{cases}$
20. return $(\lambda', \mu')$ // Updated multipliers
```

The first observation is that for any apportionment $\mathbf{X}$ and multipliers λ and μ we must have

$$d(x_{ij}-1) \leq L_{ij} \leq \lambda_i p_{ij} \mu_j \leq U_{ij} \leq d(x_{ij})$$

since by Lemma 5.1 quotients cannot “cross” signpost boundaries.

We sketch an algorithm to compute U_{ij} for a given element (i,j) where $p_{ij} > 0$. (If $p_{ij} = 0$ then $L_{ij} = U_{ij} = 0$.) Assume that we have an apportionment $\mathbf{X}$ and multipliers λ and μ. If $\lambda_i p_{ij} \mu_j = d(x_{ij})$ we set $U_{ij} = d(x_{ij})$ and are done. Otherwise we attempt to increase the row multiplier λ_i while keeping the column multiplier μ_j fixed. This can be done by searching in the row/column graph for a path from row i to column j as in the tie-deciding algorithm. If no path is found, we may compute a $\delta > 1$ and update the multipliers. Note that this update increases the quotient from $\lambda_i p_{ij} \mu_j$ to $\delta \lambda_i p_{ij} \mu_j$. Then we again search in the (modified) row/column graph for a path from row i to column j. We iterate until one of the following events happens:

1. $\lambda_i p_{ij} \mu_j = d(x_{ij})$. In this case we set $U_{ij} = d(x_{ij})$ and are done.

2. There is a path from row i to column j, i.e., both row i and column j is labeled. In this case updating the multipliers has no effect on the quotient, since the row multiplier is increased while the column multiplier is decreased with the same factor. In Lemma 5.5 we prove that in this case we must have $U_{ij} = \lambda_i p_{ij} \mu_j$, and are therefore done.

Clearly, this algorithm runs in $O(nm(n+m))$ time, and its correctness follows from:

Lemma 5.5 *For any apportionment $\mathbf{X}$ and multipliers λ and μ, if there exists a path in the row/column graph from row i to column j, then $U_{ij} = \lambda_i p_{ij} \mu_j$, that is, the quotient is at its upper bound.*

Proof. We prove this by contradiction. Assume that there exists another set of multipliers λ' and μ', where $\lambda'_i p_{ij} \mu'_j > \lambda_i p_{ij} \mu_j$. Assume w.l.o.g. that $\mu'_j = \mu_j$ and $\lambda'_i > \lambda_i$. By Lemma 5.1 multipliers λ' and μ' are valid for the apportionment $\mathbf{X}$. Consider a path $(i, j_1), (j_1, i_1), (i_1, j_2), \ldots, (i_k, j)$ in the row/column graph. By construction, the edge (i, j_1) corresponds to $\lambda_i p_{ij_1} \mu_{j_1} = d(x_{ij_1})$. Clearly, $\lambda'_i p_{ij_1} \mu'_{j_1} \leq d(x_{ij_1}) = \lambda_i p_{ij_1} \mu_{j_1}$ which implies that $\lambda'_i \mu'_{j_1} \leq \lambda_i \mu_{j_1}$. Consequently, since $\lambda'_i > \lambda_i$, we have that $\mu'_{j_1} < \mu_{j_1}$.

Now consider the edge (j_1, i_1), which corresponds to $\lambda_{i_1} p_{i_1 j_1} \mu_{j_1} = d(x_{i_1 j_1} - 1)$. Clearly, $\lambda'_{i_1} p_{i_1 j_1} \mu'_{j_1} \geq d(x_{i_1 j_1} - 1) = \lambda_{i_1} p_{i_1 j_1} \mu_{j_1}$ which implies that $\lambda'_{i_1} \mu'_{j_1} \geq \lambda_{i_1} \mu_{j_1}$. Consequently, since $\mu'_{j_1} < \mu_{j_1}$, we have that $\lambda'_{i_1} > \lambda_{i_1}$.

By induction over the edges of the path we therefore get that $\mu'_j < \mu_j$, which is a contradiction to our assumption that $\mu'_j = \mu_j$. ∎

Lower bounds L_{ij} on the quotients may be computed in a similar way. The quotient intervals $[L_{ij}, U_{ij}]$ have some interesting applications. Firstly, if we have $L_{ij} = U_{ij} = d(x_{ij})$ or $L_{ij} = U_{ij} =$

$d(x_{ij} - 1)$ this means that element (i, j) is a tie. Thus computing quotient intervals immediately identifies ties. Secondly, quotient intervals give the widest possible ranges for changes of elements in $\mathbf{P} = (p_{ij})$ that do not change the apportionment. More specifically, assume that we have an element (i, j) where $p_{ij} > 0$ and $L_{ij} < U_{ij}$. Then p_{ij} may be increased to $(d(x_{ij})/L_{ij})p_{ij}$ without affecting the feasibility of the apportionment. Similarly, p_{ij} may be decreased to $(d(x_{ij}-1)/U_{ij})p_{ij}$ without affecting the apportionment. These ranges are in general larger than those obtained by considering an arbitrary set of multipliers.

5.3 Computing feasible intervals for multipliers

The technique to compute quotient intervals may be extended to computing multiplier (or divisor) intervals — and especially "nice" multipliers/divisors. A nice multiplier/divisor is for example one that can be written using a small number of (decimal) digits. One greedy approach is to fix one of the multipliers (say to 1), and then to compute the widest possible intervals for each of the other multipliers. The multiplier that has the smallest feasible interval is then fixed at some nice value in its feasible interval. Then the intervals for the other multipliers are updated, another multiplier is fixed etc. Note that as soon as we fix some multiplier, the feasible intervals for the other multipliers may change — therefore the intervals need to be recomputed.

Assume that we have fixed some subset of the multipliers, say $F = I_F \cup J_F$, where I_F is the set of fixed row multipliers and J_F is the set of fixed column multipliers, respectively. Let us say that we would like to compute an upper bound on the row multiplier λ_i, where $i \notin I_F$. As in the algorithm to compute quotient upper bounds, we attempt to find a $\delta > 1$ such that $\delta\lambda_i$ is a valid multiplier for row i. The idea is again to search the row/column graph from row i. If no row or column in F is reached (or labeled), we compute $\delta > 1$ and update the multipliers as in the tie-and-transfer algorithm. (Note that since no fixed rows or columns are labeled, the corresponding multipliers are not changed.) Then we again search the (modified) row/column graph, update the multipliers etc. until some row or column in F is labeled. Using arguments similar to those in the proof of Lemma 5.5, it can be shown that the multiplier for row i must be at its upper bound.

Lower bounds for the multipliers can be computed in a similar way. The running time to compute an upper or lower bound for a multiplier is $O(nm(n + m))$.

6 Conclusion

In this paper we presented the first (weakly) polynomial algorithms for solving the divisor-based biproportional rounding problem. Polynomial algorithms for analyzing ties, quotients and multipliers/divisors were given. A prototype implementation of the algorithm presented in Section 4.2

has shown that the algorithm is very fast in practice. [2] Future work includes an experimental study of the practical running times, including comparisons to the algorithms presented in Maier [13], and practical methods for finding nice divisors. From a theoretical point of view, it remains an open question whether there exists a *strongly* polynomial algorithm for divisor-based biproportional rounding of matrices.

Acknowledgments

The author would like to thank Friedrich Pukelsheim, Sebastian Maier and Petur Zachariassen for many fruitful comments and suggestions.

References

[1] M. L. Balinski. Wahlen in Mexico - Verhältniswahlrecht häppchenweise. *Spektrum der Wissenschaft*, Oktober:72–74, 2002.

[2] M. L. Balinski and G. Demange. Algorithms for Proportional Matrices in Reals and Integers. *Mathematical Programming*, 45:193–210, 1989.

[3] M. L. Balinski and G. Demange. An Axiomatic Approach to Proportionality Between Matrices. *Mathematics of Operations Research*, 14:700–719, 1989.

[4] M. L. Balinski and S. T. Rachev. Rounding Proportions: Methods of Rounding. *Mathematical Scientist*, 22:1–26, 1997.

[5] M. L. Balinski and V. Ramírez. Mexico's 1997 Apportionment Defies its Electoral Law. *Electoral Studies*, 18:117–124, 1999.

[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.

[7] D. Bochsler. Biproportionale Wahlverfahren für den Schweizer Nationalrat. `www.opus-bayern.de/uni-augsburg/volltexte/2005/160`, 2005.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, 2001.

[9] G. Dorfleitner and T. Klein. Rounding with Multiplier Methods: An Efficient Algorithm and Applications in Statistics. *Statistical Papers*, 40:143–158, 1999.

[2] Java-code is available from `http://www.diku.dk/users/martinz/biprop/`.

[10] M. Happacher. The Discrepancy Distribution of Stationary Multiplier Rules for Rounding Probabilities. *Metrika*, 53:171–181, 2001.

[11] M. Happacher and F. Pukelsheim. Rounding Probabilities: Maximum Probability and Minimum Complexity Multipliers. *Journal of Statistical Planning and Inference*, 85:145–158, 2000.

[12] N. Linial, A. Samorodnitsky, and A. Wigderson. A Deterministic Strongly Polynomial Algorithm for Matrix Scaling and Approximate Permanents. *Combinatorica*, 20:545–568, 2000.

[13] S. Maier. Algorithms for Biproportional Apportionment Methods. In *Mathematics and Democracy. Recent Advances in Voting Systems and Collective Choice*, In Press, New York, 2006.

[14] F. Pukelsheim. BAZI – A Java Program for Proportional Representation. In *Oberwolfach Reports*, volume 1, pages 735–737, 2004.

[15] F. Pukelsheim. Matrices and Politics. In *15th International Workshop on Matrices and Statistics*, In Press, 2006.

[16] F. Pukelsheim and C. Schuhmacher. Das neue Zürcher Zuteilungsverfahren für Parlamentswahlen. *Aktuelle Juristische Praxis - Pratique Juridique Actuelle*, 5:505–522, 2004.

[17] P. Zachariasen and M. Zachariasen. A Comparison of Electoral Formulae for the Faroese Parliament (The Løgting). In *Mathematics and Democracy. Recent Advances in Voting Systems and Collective Choice*, In Press, New York, 2006.