Algorithmen & mathematische Modellierung

**Forschungsschwerpunkt**

Algorithmen und mathematische Modellierung

TU Graz
Graz University of Technology

# Oracle-guided search in sorted matrices improving balanced flow compuatation

Matthias W. Altenhöfer, Elisabeth Gassner, Riko Jacob, and Sven O. Krumke

*Project Area(s):*
Kombinatorische Optimierung komplexer Systeme

Institut für Optimierung und Diskrete Mathematik (Math B)

Report 2008-13, July 2008

# ORACLE-GUIDED SEARCH IN SORTED MATRICES IMPROVING BALANCED FLOW COMPUTATION

MATTHIAS W. ALTENHÖFER [1], ELISABETH GASSNER [2], RIKO JACOB [3], AND SVEN O. KRUMKE [1]

ABSTRACT. In a successor search we are given a key $x$ and a set $A$ from a totally ordered universe and search for the smallest element of $A$ that is larger than or equal to $x$. It is well known that the number of comparisons with $x$ needed for this task changes from $\Theta(|A|)$ to $\Theta(\log |A|)$ if $A$ is stored in sorted order. Here, we consider a related situation where the elements of $A$ are organised as a so called sorted matrix. In such a matrix every column and every row is sorted. Further, $x$ is given implicitly by a "monotone oracle". Given a test value $t$, the oracle answers the question whether $t \geq x$. We give a search algorithm for a sorted $n \times n$-matrix performing $O(\log n)$ calls to the oracle and $O(n)$ comparisons between matrix elements which we prove to be optimal. We extend this result to the case of non-square matrices and the situation where only columns are sorted.

Our search techniques can be applied as the key tool to give an improved algorithm for the uniform balanced network flow problem (UBNFP). The UBNFP consists of finding a feasible $s$-$t$-flow of given value $F$ in a graph $G = (V, A)$ which minimizes the difference of the maximum and the minimum flow on an arc. We show that our search techniques can be applied to obtain an $O(\log^2 m \cdot T^2_{\mathrm{MF}}(n, m))$ time algorithm for solving the UBNFP, where $T_{\mathrm{MF}}(n, m)$ is the time required for a maximum flow computation in a network with $n$ vertices and $m$ arcs. This improves upon the previous best time bound of $O(n^2 \cdot T^2_{\mathrm{MF}}(n, m))$.

[1]Email: {altenhoe,krumke}@mathematik.uni-kl.de Dept. of Mathematics, University of Kaiserslautern, Paul-Ehrlich-Str. 14, 67663 Kaiserslautern, Germany.

[2]Email: gassner@opt.math.tu-graz.ac.at Institut fuer Mathematik, Technische Universitaet Graz Steyrergasse 30/II, 8010 Graz, Austria. This research is partially supported by the Austrian Science Fund Project P18918-N18

[3]Email: jacob@in.tum.de Department of Computer Science, Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany

## 1. Introduction

In the uniform balanced network flow problem (UBNFP) one is given a directed graph $G = (V, A)$ and searches for an $s$-$t$-flow $f$ of prescribed value $F$ such that the difference between the maximum and the minimum flow on an arc, i.e., $\max_{a \in A} f(a) - \min_{a \in A} f(a)$ is minimized. This problem falls into the class of balanced optimization problems which (to some extent) model "equitable distribution" of resources [1, 4, 10].

It has been known for a while that UBNFP can be solved in strongly polynomial time even in a more general setting [9, 16, 17]. However, in the special case of $s$-$t$-flows it turns out that the search for an optimal solution (value) can be narrowed down to a small candidate set which can be represented as the entries of a matrix $M$, whose columns and rows are sorted. Essentially, one then only needs to find the smallest entry in $M$ such that a feasible flow exists, in other words the smallest entry in $M$ such that a "flow oracle" answers "yes".

This problem is a special case of an oracle guided search in a sorted matrix ("$f$,$\ell$ matrix oracle"). We develop general techniques for searching in sorted matrices which depend on how much information about the entries is available. Our techniques lead to an $O(\log m \cdot T_{\mathrm{MF}}(n, m))$ time algorithm for a variant of the parametric flow problem and $O(\log^2 m \cdot T^2_{\mathrm{MF}}(n, m))$ time for the UBNFP. Here, $T_{\mathrm{MF}}(n, m)$ is the time used by a maximum flow algorithm. The latter improves upon the previously best known time bound of $O(n^2 \cdot T^2_{\mathrm{MF}}(n, m))$ from [9].

Our paper is organized as follows. We start with the exposition of our search techniques. Section 2 introduces notation and the general framework and gives a brief review of related work. In Section 3 we give lower bounds for searching in sorted matrices, while Section 4 presents our search algorithms which match these bounds. In Section 5 we review previous work on the UBNFP and show how to obtain an $O(\log^2 m \cdot T^2_{\mathrm{MF}}(n, m))$ time algorithm by using an oracle guided search.

## 2. Searching in Ordered Two-Dimensional Arrays

It is well known that the complexity of searching for the predecessor of some key in a set of size $n$ heavily depends on how the set is organized: If it is sorted, it can be done in $\log n$ steps, if nothing is known about the set, it will take $n$ steps. Here, we consider a setting where the set is organized in a two dimensional array – from now on simply called "matrix" – such that each column and each row is sorted.

A similar problem has been considered by Frederickson and Johnson [6]. There, the complexity of selecting the $k$-th smallest element in a sorted matrix and of finding the number of elements in a sorted matrix that are smaller than a key (ranking) is analyzed. Ranking is closely related to successor search, the main difference is the oracle: In [6] there can be as many questions to the oracle as there are steps of the algorithm.

Here, we assume that the entries of the matrix stem from an ordered universe, and that the only operation allowed on such elements is the comparison. The indices necessary to describe positions of the matrix are natural

numbers. We assume that such numbers (up to $n \cdot m$, the size of the matrix) can be added and compared at unit cost.

We count the indexed accesses to the matrix separately from general computation time. Further, comparisons involving the element for which we search the predecessor are counted as oracle questions.

**Definition 2.1.** Let $A$ be an $m \times n$ matrix. We say that $A$ is (*column*) *semi-sorted matrix* if $i \leq i'$ implies that $a_{ij} \leq a_{i'j}$. If $i \leq i'$ and $j \leq j'$ imply that $a_{ij} \leq a_{i'j'}$, $A$ is called *sorted matrix*.

Slightly deviating from what is usually considered a matrix, it is meaningful to consider semi-sorted matrices where not all columns are equally long. Then the vector $(m_1, \ldots, m_n)$ gives the size (length) of the different columns (technically the number of non-infinity entries per column).

**Definition 2.2.** A *monotone oracle* $\mathfrak{B}$ contains the number $b \in \mathbb{R}$. Upon question $a \in \mathbb{R}$, the oracle returns the yes/no answer to whether $a \geq b$.

**Definition 2.3.** Given a (semi-)sorted matrix $A$, a *(Semi-)Sorted Matrix Oracle* returns a smallest entry $a_{ij}$ of $A$ such that $\mathfrak{B}$ answers yes.

One algorithm to solve a Sorted Matrix Oracle is the following: Sort the $n \cdot m$ entries of the matrix $A$ in time $O(nm \log nm)$, then use the oracle $\mathfrak{B}$ to perform a binary search. This algorithm accesses all $n \cdot m$ elements of the matrix, and uses the oracle $\lceil \log n \cdot m \rceil$ times. In the following, we want to reduce the number of accesses to the matrix without significantly increasing the number of questions to the oracle.

**Definition 2.4.** An $f, \ell$ *Matrix Oracle* is given by an $n \times n$ matrix given by $a_{ij} = (f + i\ell)/j$. After mirroring around a vertical axis, this matrix is a (strictly) sorted matrix in the above sense.

**Weighted Median.** As a subroutine, we need to compute the weighted median: For $n$ pairs $(a_i, w_i)$ we search for an index $a$ with $\sum_{i:a_i<a} w_i \leq N/2$ and $\sum_{i:a_i>a} w_i \leq N/2$, where $N = \sum_i w_i$. This can be done in $O(n)$ time [8], where it is assumed that summing two numbers take constant time.

## 3. Lower Bounds for Sorted Matrix Oracle

We start our discussion by observing some simple lower bounds on the number of accesses to the oracle and the matrix.

Because there are $n \cdot m$ possible answers for Sorted Matrix Oracle, the oracle must be queried at least $\log n + \log m$ many times. Further, in an $n \times n$ Sorted Matrix Oracle $A$ two elements on the secondary diagonal do not have an ordering prescribed. Hence, it is easy to construct an $A$ that requires any correct algorithm to access at least $n$ entries.

Further, for a non-square semi-sorted matrices with chain-lengths vector $m_1, \ldots, m_n$, any deterministic algorithm can be forced to access $\sum_{i=1}^{n}(1+\log m_i)$ elements of the matrix. By Yao's Minimax-principle [13, 19, p. 35] it is clear that the worst case number of matrix accesses or oracle questions cannot be improved by using randomization, and some standard further considerations show that the given lower bounds remain valid for the expected number of accesses as well.

Combining the ideas above, we get that an $n \times m$ Sorted Matrix Oracle with $m \geq n$ requires $n \log \lfloor m/n \rfloor$ accesses to the matrix: to this end, consider a situation where $n$ divides $m$. Define an $n \times n$ block matrix $B$, whose entries $b_{ij}$ are $m/n \times 1$ matrices. Embed the columns of an arbitrary $m \times m/n$ Semi-Sorted Matrix Oracle $A$ into the blocks $b_{ij}$ with $i + j = n$ and fill the remaining entries of the matrix with $\pm\infty$. Now solving the Sorted Matrix Oracle $B$ solves $A$.

## 4. Sorted Matrix Oracle Algorithms

Our exposition starts with an algorithm that uses the oracle very frequently. It can be understood as a variation of an algorithm in [6]. Because it mainly serves as an introduction and reference for the other algorithms, we discuss it in detail.

4.1. **Algorithm 0 / Optimizing Matrix Accesses only.** Consider the situation of a square matrix $A$ and $n = 2^k - 1$. Hence, there are $k$ natural submatrices $A_i$, given by entries with both indices being divisible by $2^{k-i}$ for $i \in \{1, \ldots, k\}$. We generalize Sorted Matrix Oracle in the following way: Instead of asking only for the smallest element for which the oracle answers "yes", we keep track of such an element for every row and every column this element (in case of ties, the one with smallest index). Actually, this is the same as finding the correct position of $b$ in each row and each column. Now assume this knowledge is already available for the submatrix of level $i$, and we want to create it for level $i + 1$. To this end, we only need to access elements that could still be this boundary. Observe that every new element either is in an old row or column and has two old horizontal and vertical neighbors, or has two old diagonal neighbors. Because old rows, columns and diagonals are ordered and the boundary is known, there can be at most one new element per old row, column, and diagonal. Hence there are at most $2^i + 2^i + 2 \cdot 2^i = 4 \cdot 2^i$ new elements to be accessed and tested in this round. Hence, this algorithm accesses in total $\sum_{i=1}^{k} 4 \cdot 2^i = 4 \cdot (2^{k+1} - 1) \leq 8n$ out of the $n^2$ elements of the matrix.

The number of questions to the oracle can be kept at $k^2 = \log^2 n$ by not asking every element individually. Instead, we collect all new elements of the matrix of one round, sort them, and perform a binary search on this list. Then, all necessary answers of the oracle can be deduced. The $O(n \log n)$ computation time of this step can be improved to $O(n)$ by replacing the sorting with repeated median finding. In general, our algorithms will in addition to the oracle questions take computational time linear in the number of oracle questions.

4.2. **Semi-Sorted Matrix Oracle.** Considering the columns of a Semi-Sorted Matrix Oracle as individual tasks, it can certainly be solved with $n \log m$ oracle questions and matrix accesses. A first improvement is to proceed in rounds, where each column contributes with the current median of the remaining possible answers. Again, instead of asking all $n$ oracle questions, the $n$ values are sorted and a binary search with the oracle is performed, yielding all $n$ answers. The total number of questions to the oracle is now $\log n \cdot \log m$.

The following algorithm reduces this to $O(\log n + \log m)$ questions to the oracle. The main idea is to let go the synchronization between the searches in the different columns. Then, in every column, at all times, the search can be characterized by 3 entries, namely the smallest one known to be larger than $b$, the largest one known to be smaller than $b$, and the middle position between the two, the element $x_i$ for which the comparison with $b$ (the question to the oracle) is pending (unless the column is finished). This identifies the number $m_i$ of elements in this column that are still potentially the right answer. Now the median $x$ of the $x_i$ weighed with $m_i$ is computed and is used to question the oracle. If the oracle answers "yes" ($b \leq x$) this makes progress for all searches with $x_i \geq x$, if the answer is "no" it makes progress for all searches with $x_i \leq x$. If a search in column $i$ makes progress, the number $m_i$ is halved. Because $x$ is the weighted median, this progress happens for one half of the potential outcomes, and $\sum_i m_i$ is reduced by a factor 3/4. Hence, the number of rounds and oracle questions is $O(\log(n \cdot m)) = O(\log n + \log m)$, and the number of accessed matrix elements is $O(n \log m)$, which is optimal.

4.3. **Square Sorted Matrix Oracle.** Now the ideas of Section 4.1 and 4.2 are combined. The resulting algorithm for square Sorted Matrix Oracles performs $O(n)$ accesses to the matrix and $O(\log n)$ oracle questions.

Observe that we can identify new elements to be accessed in the matrix (i.e., scheduled to be compared with the oracle) as soon as its neighbors one level higher are identified to be on the boundary. Define for every element that is accessed in Algorithm 0 at level $i$ its weight to be $(2^{k-i})^2$. Similarly to Section 4.2, we keep a list of active nodes (for which the comparison with the oracle is pending), compute their weighted median, and query the oracle with this median. This resolves either all active nodes whose values are at least as large as the median, or at most as large as the median.

If an element $a$ of level $i$ is resolved (implicitly compared to the oracle), it gives rise to at most three new neighboring elements on level $i+1$, namely one for the row, one for the column, and one for the diagonal. Because the weight of one such element is one quarter of the weight of $a$, exchanging $a$ with its three replacements (follow-ups) still reduces the weight of active nodes by one quarter of the weight of $a$. Hence, every question to the oracle reduces half the weight of the active nodes by a factor 3/4. Because initially there is one active node of weight $n^2$, the total number of queries to the oracle is $\log_{8/7} n^2 = O(\log n)$.

4.4. **General Sorted Matrix Oracle.** Assume w.l.o.g. $n > m$. Select $m$ columns of $A$ at distance roughly $n/m$, leading to the square $m \times m$ submatrix $A'$. Solve Sorted Matrix Oracle on $A'$, remembering the dividing line. Now, solve the remaining pieces of the rows as columns of a (transposed) $m \times n/m$ Semi-Sorted Matrix Oracle. In total, this gives $O(\log nm)$ questions to the oracle and $O(n\lceil\log(n/m)\rceil$ accesses to the matrix (both terms dominated by the second part).

4.5. $f, \ell$ **Matrix Oracle.** For the particularly structured case of a $f, \ell$ Matrix Oracle, a more direct algorithm achieves $O(n)$ matrix accesses and

$O(\log n)$ oracle queries, asymptotically matching the Sorted Matrix Oracle case.

Define $g : \mathbb{R}^2 \to \mathbb{R} : (x, y) \mapsto \frac{f+\ell x}{y}$. Then the $m \times m$-matrix $A$ is given as $a_{ij} = g(i, j)$. We will narrow down the set of interest for our search by considering level sets of $g$. Given level $\alpha \in \mathbb{R}$, all points $(x, y) \in \mathbb{R}^2$ with function value $g(x, y) = \alpha$ lie on the line $\Gamma_\alpha$ given by $y = \frac{f+x\ell}{\alpha}$. These lines intersect at $(-f/L, 0)$ – where $g$ itself is not defined. We will first find $b_1, b_2$ in

$$\begin{aligned} S_1 &:= g\big(\{(1,m),\ldots,(m,m),\ldots,(m,1)\}\big) \\ &= \left\{\tfrac{f+1\cdot\ell}{m} < \ldots < \tfrac{f+m\cdot\ell}{m} < \ldots < \tfrac{f+m\ell}{1}\right\}, \end{aligned} \tag{1}$$

such that, when asked $b_1$ and $b_2$ the oracle answers "no" and "yes", respectively and such that there is no $s \in S_1$ satisfying $b_1 < s < b_2$, hence, $b_1 < b \leq b_2$.[1] The values $b_1$ and $b_2$ can be found by a binary search using $\log |S_1| = \log(2m - 1)$ calls to the oracle. Denote by $p_1, p_2 \in \{(1, m), \ldots, (m, m), \ldots, (m, 1)\}$ the coordinates satisfying $g(p_j) = b_j$. Then either $p_1 = (m, i)$ and $p_2 = (m, i - 1)$ or $p_1 = (j, m)$ and $p_2 = (j + 1, m)$. The first case is illustrated in Figure 1.

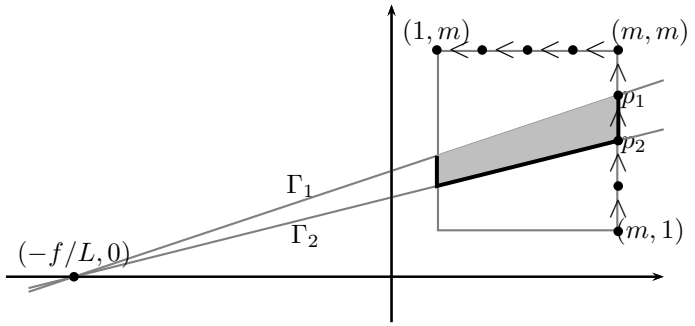


FIGURE 1. Set $S_1$ (see (1)) embedded in $\mathbb{R}^2$. The operators $<$ refer to the relations of the values of $g$. The dots represent $S_1$ and the shaded area between the level sets $\Gamma_{b_1}$ and $\Gamma_{b_2}$ depicts $S_2$.

Thenceforth, we will assume the first case, the second case can be lead to a solution in a symmetric way. Using $b_1 < b \leq b_2$, we know that the solution is a point with integral coordinates in

$$S_2 := \left\{(x, y) \in \mathbb{R}^2 \middle| 1 \leq x \leq m, u_1 < \frac{f + x\ell}{y} \leq u_2\right\}.$$

We can limit the "height" of $S_2$ by a geometric argument or, with $x < m$, by

$$\left|\frac{f + x\ell}{u_1} - \frac{f + x\ell}{u_2}\right| = \left|\frac{f + x\ell}{g(p_1)} - \frac{f + x\ell}{g(p_2)}\right| = \left|\frac{f + x\ell}{f + m\ell}\right| < 1.$$

Consequently, no two integral points in $S_2$ exist with the same first coordinate and we get $|S_2 \cap \mathbb{N}^2| \leq m$. Starting with $(k, l) = p_2$, one can

[1] If such $b_1, b_2$ do not exist, we know that either $a < b$ or $a \geq b$ for all $a \in A$.

identify these integral points by walking $S_2$ in a staircase scheme: if $g(k,l) < u_1$ set $k \leftarrow k-1$, else set $l \leftarrow l-1$. Starting with the candidate list $\{u_2 = g(p_2)\}$ add $g(k,l)$ if $g(k,l) \in [u_2, u_1)$ after each step.

Finally, an iterative median search is applied to determine $U_L$ among these candidates, taking $O(m)$ time to identify the medians and $\log m$ oracle questions for the binary search. Note that by choosing $L > 0$ sufficiently small we can achieve a situation with $m$ candidates.

**Remark 4.1.** The following improvement applies to a setting that is slightly more general than the application in Section 5. Consider non-square $f$-$\ell$-matrices of size $m \times n$. Now it is possible that the cone identified with the binary search still intersects some row or column at more than one position, i.e., has an intersection of length $\geq 1$ with a horizontal or vertical line (intersecting the positive quadrant) at distance to the origin at most $n$ or $m$. In that case another binary search, orthogonal to the boundary of $p_1$ or $p_2$, is performed, shrinking the cone such that all relevant horizontal and vertical intersections have length $< 1$. Hence, there is at most one entry to be checked within any column, and also within any row. Hence, after the binary searches, only $\min\{n, m\}$ entries of the matrix are accessed.

## 5. Uniform Parametric Flow and Balanced Flow

In this section we illustrate how to apply the matrix search techniques to obtain a fast algorithm for a class of parametric flow problems (UPFP) with running time $O(\log m \cdot T_{\mathrm{MF}}(n,m))$. Here, $T_{\mathrm{MF}}(n,m)$ is the time used for a maximum flow computation in a network with $n$ vertices and $m$ arcs. This algorithm is a building block for an $O(\log^2 m \cdot T^2_{\mathrm{MF}}(n,m))$ time algorithm for the uniform balanced flow problem (UBNFP).

5.1. **Problem Definitions.** Let $G = (V, A)$ be a network with $n = |V|$ vertices, $m = |A|$ arcs and two distinguished vertices $s, t \in V$. An $s$-$t$-flow is a function $f\colon A \to \mathbb{R}_{\geq 0}$ satisfying the flow-conservation constraints at each vertex with the exception of the source $s$ and the sink $t$. Given a flow value $F$, the uniform parametric flow problem with fixed lower bound $L$ ($\mathrm{UPFP}_L$) consists of finding an $s$-$t$-flow of value $F$ such that $L \leq f(a)$ for each arc $a \in A$ and the maximum flow on an arc, i.e., $\max_{a \in A} f(a)$ is minimized. The $\mathrm{UPFP}_L$ can be given as a Linear Program as follows

$$
\begin{aligned}
(\mathrm{UPFP}_L) \qquad \min \quad & U \\
& \sum_{a \in \delta^-(v)} f(a) - \sum_{a \in \delta^+(v)} f(a) = \begin{cases} -F, & \text{for } v = s \\ F, & \text{for } v = t \\ 0, & \text{otherwise} \end{cases} \\
& L \leq f(a) \leq U \text{ for all } a \in A
\end{aligned}
$$

Here, as is usual, $\delta^-(v)$ and $\delta^+(v)$ denote the set of arcs entering and leaving node $v$, respectively. The uniform balanced $s$-$t$-flow problem (UBNFP) consists of finding an $s$-$t$-flow of value $F$ with minimum difference between the maximum and the minimum flow on an arc. Formally, this problem can

be stated as the following Linear Program:

$$\text{(UBNFP)} \qquad \min \quad U - L$$

$$\sum_{a \in \delta^-(v)} f(a) - \sum_{a \in \delta^+(v)} f(a) = \begin{cases} -F, & \text{for } v = s \\ F, & \text{for } v = t \\ 0, & \text{otherwise} \end{cases}$$

$$0 \leq L \leq f(a) \leq U \text{ for all } a \in A$$

5.2. **Previous Work.** Both the UPFP$_L$ and the UBNFP can be stated as Linear Programs with a $\{0, \pm 1\}$-constraint matrix and hence can be solved in strongly polynomial time by Tardos algorithm [17].

There are various variants of the parametric flow problem in the literature. A general form allows for individual supplies/demands $b(v)$ for each vertex $v \in V$ and a real valued capacity function $c\colon \mathbb{R} \times A \to \mathbb{R}_{\geq 0}$. Frequently a lower bound of 0 for the flow on each arc is assumed. The UPFP$_L$ considered in this paper fits in this scheme by setting $c(p, a) = p$ for parameter $p$ and every arc $a$ and adding the constraint $f(a) \geq L$ for each arc $a \in A$. If one is not restricted to $s$-$t$-flows, fixed lower bounds can be removed by pushing the required flow and adjusting the supply/demand value accordingly, thereby, UPFP$_L$ can be matched into this general classification.

Radzik [15] developed an algorithm for the case $c(p, a) = \lambda_a p$ for a fixed set of capacity multipliers $\lambda_a$. He proved the complexity of his approach to be $O(mT_{\mathrm{MF}}(n, m))$ in the general case and $O(nT_{\mathrm{MF}}(n, m))$, when setting $\lambda_a = 1$ for all $a \in A$. We improve the latter, by taking advantage of the fact that for all $s$-$t$-cuts, the amount of flow that is required to pass through the cut is the same in $s$-$t$-networks. This achieves a running time of $O(\log m \cdot T_{\mathrm{MF}}(n, m))$. Gallo, Grigoriadis and Tarjan [7] show that certain versions of the push-relabel algorithm for the maximum flow problem can be extended to solve the parametric flow problem whilst only increasing the run time by a constant factor, attaining a time bound of $O(mn \log(n^2/m))$. This approach consists of extending the maximum flow algorithm such that it evaluates maximal flows in the network for $O(n)$ ordered values of the parameter, without "loosing time significantly". Babenko et al. [3] compare the GGT algorithm to a balancing algorithm designed for bipartite networks developed by Zhang et al. [18, 20].

Scutellà [16] presented a combinatorial algorithm for the UBNFP which runs in time $O(n^2 \log^3 n \cdot T_{\mathrm{MF}}(n, m))$. Her approach relies on an extension of Radzik's analysis of Newton's method for linear fractional combinatorial optimization problems [14, 15]. The algorithm from [16] does not only work for the uniform balanced $s$-$t$-flow problem but for the more general case where a supply/demand $b(i)$ is given at each node $i$. Klinz and Scutellà [9] improved the algorithm from [16] by additionally employing Megiddo's parametric search technique [11,12]. They obtain an $O(n^2 \cdot T^2_{\mathrm{MF}}(n, m))$ algorithm for minimizing the weighted difference $\max_{a \in A} c(a)f(a) - \min_{a \in A} c(a)f(a)$. We improve the asymptotic running time by factor of $n^2/\log^2 m$.

Our approach uses the same subproblem structure as in [9]. In particular, our algorithm relies on the function $L \mapsto U_L$. Given the lower bound $L$ to the flow values of all arcs, this function is defined as $U_L$, the

best corresponding upper bound permitting a feasible flow. In other words, $U_L$ is the optimal solution of UPFP$_L$. However, in order to solve the occurring parametric flow problem (see Section 5.3) instead of using Radzik's method [14,15] we identify a "sufficiently small" candidate set containing $U_L$ which can be represented as a sorted matrix and apply our search techniques. This structural property is no longer valid in the general case considered in [9].

5.3. **Uniform Parametric Flow with Fixed Lower Bound.** A pair $(S,T)$ is called $s$-$t$-cut if $S \dot\cup T = V$, $s \in S$, and $t \in T$. In the following, we will refer to an $s$-$t$-cut simply by cut, and write $\delta^+(S) = \{(v,w) \in A \,|\, v \in S, w \in T\}$ and $\delta^-(S) = \{(v,w) \in A \,|\, v \in T, w \in S\}$. Since for any cut $(S,T)$, we need to send flow of value $F$ from $S$ to $T$, and on every arc in $\delta^-(S)$ a flow of value at least $L$ is sent in the other direction, we get by weak duality

$$F \leq U_L \cdot |\delta^+(S)| - L \cdot |\delta^-(S)|$$

or equivalently

$$U_L \geq \frac{F + L \cdot |\delta^-(S)|}{|\delta^+(S)|}. \tag{2}$$

(Here we have assumed that $\delta^+(S) \neq \emptyset$ for each cut $S$ which can be done without loss of generality, since otherwise only for $L = F = 0$ there is a feasible flow).

On the other hand, by the generalized max-flow-min-cut-theorem (see e.g. [2, Section 6.7]), we also know that a feasible flow exists if (2) is fulfilled for all cuts, and since $U_L$ is minimum satisfying this condition, there must be a cut $(S,T)$ for which the inequality holds with equality. Thus, we have

$$U_L = \max_{(S,T)} \frac{F + L \cdot |\delta^-(S)|}{|\delta^+(S)|}. \tag{3}$$

By definition $|\delta^+(S)| \in \{1,\dots,m\}$ and $|\delta^-(S)| \in \{0,\dots,m-1\}$ for any cut $(S,T)$, and thus we know

$$U_L \in \mathfrak{U}_L := \left\{ \frac{F + Li}{j} : i = 0,\dots,m-1, j = 1,\dots,m \right\}.$$

The entries of $\mathfrak{U}_L$ are ordered by

$$i \leq i', j \geq j' \Longrightarrow \frac{f + Li}{j} \leq \frac{f + Li'}{j'},$$

Hence, given a monotone oracle which for a fixed $U \in \mathbb{R}$ answers whether $U \geq U_L$, we can apply our search technique from Section 4.5 to the set $\mathfrak{U}_L$ in order to find $U_L$. To this end, we define the oracle as follows. When the oracle is queried with $U$, we compute a maximum $s$-$t$-flow in the network $G$ subject to the constraint that $L \leq f(a) \leq U$ for all arcs $a \in A$. If this flow exists and has value greater than or equal to $F$, we know that $U \geq U_L$. This gives the following result and justifies the special form of the entries considered in Section 4.5:

**Theorem 5.1.** UPFP *can be solved in* $O(\log m \cdot T_{MF}(n,m))$ *time.* □

5.4. **Uniform Balanced Network $s$-$t$-Flow.** We now illustrate the application of the technique introduced in the previous section to solve the balanced network flow problem (UBNFP). Define the function $h(L) = U_L - L$, where $U_L$ is the optimal solution value to UPFP$_L$, i.e., the smallest value of $U$ for which a feasible flow $f$ exists, satisfying $L \leq f(a) \leq U$ for all arcs $a \in A$. Denote by $L^*$ and $U^* = U_{L^*}$ the values of $L$ and $U$ in an optimal solution of UBNFP, i.e. $L^* = \arg\min h(L)$. From (3) we know that

$$h(L) = U_L - L = \max_{(S,T)} \left\{ \frac{F}{|\delta^+(S)|} + L \cdot \frac{|\delta^-(S)| - |\delta^+(S)|}{|\delta^+(S)|} \right\}, \tag{4}$$

establishing the following lemma:

**Lemma 5.2.** *$h(L)\colon \mathbb{R} \to \mathbb{R}$ is a piecewise linear, convex function.* □

This result is also established in [9] for the more general case of flows with prescribed supplies/demands. However, our approach considers an $s$-$t$-flow with uniform lower bounds $L$ in the network $G$, whereas Klinz and Scutellà [9, 16] treat a flow in the residual network after pushing $L$ units of flow on each arc of the network. While being more involved, their proof reveals a dual problem (the maximum mean weight cut problem) and also establishes the connection to Radzik's fractional programming method [14].

From (4) we see the important fact that $h$ is the maximum of affine linear functions, each of which relates to a cut in $G$. For a given $L \geq 0$ the function $h$ can be evaluated at $L$ by solving UPFP$_L$ as in Section 5.3 (cf. (3) and (4)) with the help of our matrix search in $\mathfrak{U}_L$. Once $U_L$ is identified in $\mathfrak{U}_L$ by means of its coordinates $(i, j)$, we know that there is a cut $(S, T)$, whose associated affine linear function attains the value $h(L)$ at $L$ (this is the maximum value among the affine linear functions at this point) and $i = |\delta^-(S)|$ and $j = |\delta^+(S)|$. Then $h(L) = (F + iL)/j - L$ and (if the maximum is unique) $h'(L) = i/j - 1 = (i - j)/j$.

In the following we wish to use the slope at $L$ only to determine whether $L^* \geq L$ or $L^* \leq L$. To this end, if the maximum is not unique, it suffices to know the slope of one of the affine functions attaining the maximum. Thus we somewhat abuse the definition of the derivative such that for points $x$ of non-differentiability $h'(x) \geq 0$ and $h'(x) \leq 0$ will merely mean that one of the occurring slopes is non-negative and non-positive, respectively. This yields the following result:

**Lemma 5.3.** *$h(L)$ and $h'(L)$ can be computed simultaneously by means of a single* UPFP$_L$ *computation for given $L \geq 0$.* □

With the former discussion, one can solve UBNFP in the style of Meggido's parametric search [11, 12] by solving UPFP$_L$, using $L^*$ as a symbol for a not (yet) known minimizer of $h$. Below, we discuss how to decide comparisons that appear in the algorithm from Section 5.3 involving symbolic variable $L^*$. Observe that our algorithm only uses comparisons and additions (provided, of course, that the maximum flow algorithm used does so, too).

Given $L^*$, the unknown (symbolic) minimizer of $h$, our aim is to find the minimum value $U_{L^*} \in \mathfrak{U}_{L^*}$ such that a feasible flow exists in the network,

obeying $L^* \leq f(a) \leq U_{L^*}$ for each arc $a \in A$. As in Sections 4.5 and 5.3, we start by lower- and upper-bounding this value with elements from the set

$$S_1 = \left\{ \frac{F + iL^*}{j} \middle| (i,j) \in \{(0,m), \ldots, (m-1,m), \ldots, (m-1,1)\} \right\} \subsetneq \mathfrak{U}_{L^*}.$$

Observe that independent of the value of $L^*$, this set is ordered, hence one can apply a binary search to find two elements $p_1, p_2 \in S_1$ as in Section 4.5. This requires $O(\log m)$ answers to questions of the form

$$(F + iL^*)/j \stackrel{?}{\geq} U_{L^*}. \tag{5}$$

The answer to such a question can again be found by a maximum flow computation with the constraints $L^* \leq f(a) \leq (F + iL^*)/j$, however the peculiarity of a maximum flow computation of this type is that the lower and upper capacities the flow values, and thus also the residual capacities, of arcs are affine linear functions in the unknown (symbolic) variable $L^*$.

Most relevant to the execution of this max-flow algorithm is the comparison of two "values" that are affine functions of $L^*$, say $a_1L^* + b_1$ and $a_2L^* + b_2$. We know that $L^*$ is non-negative, and hence if the lines given by $y = a_1x + b_1$ and $y = a_2x + b_2$ do not intersect in the positive reals, we know the outcome right away. Otherwise, the lines intersect at $\overline{x} \geq 0$ and we can evaluate $h'(\overline{x})$, which gives information about the relative position of $L^*$ to $\overline{x}$. In conjunction with the slopes $a_1$ and $a_2$, we can then decide whether $a_1L^* + b_1 \leq a_2L^* + b_2$. By Lemma 5.3, the cost of evaluating $h'(\overline{x})$ equals the time to solve $\text{UPFP}_L$. Thus, the total time for a question of the form (5) is $O(T_{\text{MF}}(n,m))$ times the effort for solving $\text{UPFP}_L$. Since by Section 5.3 the latter can be carried out in time $O(\log m \cdot T_{\text{MF}}(n,m))$, this yields a total time of $O(\log m \cdot T^2_{\text{MF}}(n,m))$ to answer a question (5).

As in Section 4.5 we now determine $U_{L^*}$ among the $O(m)$ candidates in the set $S_2$, which is no longer ordered independently from $L^*$. However, we can sort the elements of $S_2$ using ideas from [12]. Observe that comparing two values of $S_2$ requires solving one instance of $\text{UPFP}_L$, i.e., time $O(\log m \cdot T_{\text{MF}}(n,m))$. Using an adaptation of a sequentialized parallel sorting algorithm such as Cole's scheme [5] which uses $m$ processors to sort an array of $m$ elements in parallel time $O(\log m)$, we can sort $S_2$ in time $O(m \log m + \log^3 m \cdot T_{\text{MF}}(n,m))$.

Upon sorting $S_2$ we can use a binary search to locate $U_{L^*}$ with additional $O(\log m)$ parametrized maximum flow computations (each of which involves the symbolical value $L^*$). Hence, the total time for the algorithm needed so far is $O(\log^2 m \cdot T^2_{\text{MF}}(n,m))$. Standard arguments for parametric search [11, 12] allow us to maintain an interval $I^*$ containing the optimal solution $L^*$ which with every comparison is updated (unless the considered "critical value" is already outside the current interval). At termination, $I^*$ consists of optimal solutions to the UBNFP.

## References


1. R. K. Ahuja, *The balanced linear programming problem*, European Journal of Operational Research **101** (1997), 29–38.
2. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Networks flows*, Prentice Hall, Englewood Cliffs, New Jersey, 1993.
3. M. A. Babenko, J. D., A. V. Goldberg, R. E. Tarjan, and Y. Zhou, *Experimental evaluation of parametric max-flow algorithms*, WEA, 2007, pp. 256–269.
4. P. M. Camerini, F. Maffioli, S. Martello, and P. Toth, *Most and least uniform spanning trees*, Discrete Applied Mathematics **15** (1986), 181–197.
5. R. Cole, *Parallel merge sort*, SIAM Journal on Computing **17** (1988), no. 4, 770–785.
6. G. N. Frederickson and D. B. Johnson, *Generalized selection and ranking: Sorted matrices*, SIAM Journal on Computing **13** (1984), no. 1, 14–30.
7. G. Gallo, M. D. Grigoriadis, and R. E. Tarjan, *A fast parametric maximum flow algorithm and applications*, SIAM Journal on Computing **18** (1989), no. 1, 30–55.
8. D. B. Johnson and T. Mizoguchi, *Selecting the kth element in $x + y$ and $x_1 + x_2 + \cdots + x_m$*, SIAM Journal on Computing **7** (1978), no. 2, 147–153.
9. B. Klinz and M. G. Scutellà, *A strongly polynomial algorithm for the balanced network flow problem*, Technical Report TR-97-17, Università di Pisa, October 1999.
10. S. Martello, W. R. Pulleyblank, P. Toth, and D. de Werra, *Balanced optimization problems*, Operations Research Letters **5** (1984), no. 3, 275–278.
11. N. Megiddo, *Combinatorial optimization with rational objective functions*, Mathematics of Operations Research **4** (1979), no. 4, 414–424.
12. ———, *Applying parallel computation algorithms in the design of serial algorithms*, Journal of the ACM **30** (1983), no. 4, 852–865.
13. R. Motwani and P. Raghavan, *Randomized algorithms*, Cambridge University Press, 1995.
14. T. Radzik, *Newton's method for fractional combinatorial optimization*, Proceedings of the 33rd Annual IEEE Symposium on the Foundations of Computer Science, 1992, pp. 659–669.
15. ———, *Parametric flows, weighted means of cuts, and fractional combinatorial optimization*, Complexity in Numerical Optimization (P. M. Pardalos, ed.), World Scientific Publishing Co., 1993, pp. 351–386.
16. M. G. Scutellà, *A strongly polynomial algorithm for the uniform balanced network flow problem*, Discrete Applied Mathematics **81** (1998), 123–131.
17. É. Tardos, *A strongly polynomial algorithm for solving combinatorial linear programs*, Operations Research **34** (1986), 250–256.
18. R. E. Tarjan, J. Ward, B. Zhang, Y. Zhou, and J. Mao, *Balancing applied to maximum network flow problems*, ESA'06: Proceedings of the 14th conference on Annual European Symposium (London, UK), Springer-Verlag, 2006, pp. 612–623.
19. A. C. C. Yao, *Probabilistic computations: Towards a unified measure of complexity*, Proceedings of the 18th Annual IEEE Symposium on the Foundations of Computer Science, 1977, pp. 222–227.
20. B. Zhang, J. Ward, and Q. Feng, *Simultaneous parametric maximum flow algorithm with vertex balancing*, Tech. report, HP Labs, 2005.