Optimization Methods and Software

ISSN: 1055-6788 (Print) 1029-4937 (Online) Journal homepage: https://www.tandfonline.com/loi/goms20

Minimum-cost flow algorithms: an experimental evaluation

Péter Kovács

To cite this article: Péter Kovács (2015) Minimum-cost flow algorithms: an experimental evaluation, Optimization Methods and Software, 30:1, 94-127, DOI: 10.1080/10556788.2014.895828

To link to this article: https://doi.org/10.1080/10556788.2014.895828

Optimization Methods & Software, 2015
Vol. 30, No. 1, 94–127, http://dx.doi.org/10.1080/10556788.2014.895828

Taylor & Francis
Taylor & Francis Group

Minimum-cost flow algorithms: an experimental evaluation

Péter Kovács[a,b*]

[a]*Department of Algorithms and Applications and MTA-ELTE Egerváry Research Group, Eötvös Loránd University, Pázmány P. s. 1/c, H-1117 Budapest, Hungary;* [b]*ChemAxon Ltd., Záhony u. 7, H-1031 Budapest, Hungary*

(*Received 28 May 2013; accepted 13 February 2014*)

An extensive computational analysis of several algorithms for solving the minimum-cost network flow problem is conducted. Some of the considered implementations were developed by the author and are available as part of an open-source C++ optimization library called LEMON (http://lemon.cs.elte.hu/). These codes are compared to other publicly available solvers: CS2, MCF, RelaxIV, PDNET, MCFSimplex, as well as the corresponding components of the IBM ILOG CPLEX Optimization Studio and the LEDA C++ library. This evaluation, to the author's knowledge, is more comprehensive than earlier studies in terms of the range of considered implementations as well as the diversity and size of problem instances. The presented results demonstrate that the primal network simplex and cost-scaling algorithms are the most efficient and robust in general. Other methods, however, can outperform them in particular cases. The network simplex code of the author turned out to be far superior to the other implementations of this method, and it is the most efficient solver on the majority of the considered problem instances. In contrast, the cost-scaling algorithms tend to show better asymptotic behaviour, especially on sparse graphs, and hence they are typically faster than simplex-based methods on huge networks.

Keywords: network flows; minimum-cost flows; efficient algorithms; experimental study; LEMON

AMS Subject Classifications: G.2.2; G.4

1. Introduction

The *minimum-cost flow* (MCF) problem plays a fundamental role within the area of network optimization. It is to find a minimum-cost transportation of a specified amount of flow from a set of supply nodes to a set of demand nodes in a directed network with capacity constraints and linear cost functions defined on the arcs. This problem has a remarkably wide range of applications in various fields, for instance, telecommunication, network design, transportation, routing, scheduling, resource planning, and manufacturing. Furthermore, it often arises as a subtask of more complex optimization problems. Comprehensive studies of network flow theory and algorithms can be found in, for example [3,73,96]. A recent survey related to minimum-cost network flows is presented in [98].

Over the last five decades, numerous algorithms have been devised for solving the MCF problem, and they have been thoroughly studied both from theoretical and from practical aspects. It has become evident that the asymptotic worst-case time complexity of an algorithm does not determine its actual performance on real-life problem instances. Therefore, many researchers have

*Email: kpeter@inf.elte.hu

worked on efficient implementation and experimental analysis of MCF algorithms, for example, [6,14–18,20,22,39,48,49,51,59,61,77,89,91,93]. Dominant contributions in this area were attained during the First DIMACS Implementation Challenge [32]. Furthermore, Brunsch *et al.* [21] recently applied the smoothed analysis model to a well-known MCF algorithm to obtain better explanation of its practical behaviour.

A previous paper [70] presents our implementations of several MCF algorithms along with an experimental evaluation. This study improves on that work by providing a more comprehensive computational analysis including additional MCF solvers and more test instances. The contribution of the presented results is twofold. First, a great number of implementations are compared within the same benchmark framework using a wide variety of test instances. Second, we consider larger networks than that in previous studies, which turned out to be essential to draw appropriate conclusions. The author is not aware of other recent works that provide an empirical study of this extent.

Fifteen MCF solvers are evaluated in this paper. Seven implementations are due to the author: three cycle-cancelling algorithms; implementations of the successive shortest path and capacity-scaling methods; a cost-scaling algorithm; and a primal network simplex code. All of them are available with full source code as part of the LEMON [76] optimization library under permissive license terms. These codes are compared to eight other publicly available solvers: CS2 code of Goldberg and Cherkassky (see [49,64]); the corresponding function of the LEDA library [4]; MCF solver of Löbel [77,78]; the NETOPT component of the IBM ILOG CPLEX Optimization Studio [63]; the primal and dual versions of MCFSimplex solver due to Bertolini *et al.* [10]; RelaxIV [10], a C + + translation of the original Fortran code due to Bertsekas and Tseng [11,15]; and PDNET solver of Portugal *et al.* [90,91].

The benchmark tests were conducted on an extensive collection of large-scale networks (up to millions of nodes and arcs). These instances were created either using standard generators NETGEN, GRIDGEN, GOTO, and GRIDGRAPH (see [32]), or based on networks arising in real-life problems.

The presented results provide a profound and reliable survey of the efficiency and robustness of the considered codes. Implementations of the cost-scaling and primal network simplex algorithms turned out to have the best overall performance. On special problem instances, however, other methods (for instance, the augmenting path algorithms and the relaxation method) can outperform them. Among the four primal network simplex codes evaluated in this study, our implementation turned out to be the most efficient and robust. Moreover, it is consistently superior to all other solvers on relatively small network instances (up to many thousands of nodes). However, the cost-scaling codes are substantially faster on the largest sparse networks due to their better asymptotic behaviour in terms of the number of nodes. The most robust implementations of this algorithm are CS2 and the one developed by the author. The latter implementation, to the author's knowledge, is the first to apply Goldberg's partial augment-relabel technique [50] in the MCF context, which turned out to be a considerable improvement in accordance with Goldberg's results related to the maximum flow problem.

The rest of this paper is organized as follows. Section 2 provides a brief overview of the MCF problem and its solution methods. Section 3 describes the implementations used in our experiments. Section 4 presents the computational results. Finally, the conclusions are drawn in Section 5.

2. The MCF problem

We assume familiarity with network flow theory and standard notations (see, e.g. [3,73,96]). Here we discuss only the most important definitions and results.

Let $G = (V, A)$ be a weakly connected directed graph consisting of $n = |V|$ nodes and $m = |A|$ arcs. We associate with each arc $ij \in A$ a *capacity* (upper bound) $u_{ij} \geq 0$ and a *cost* c_{ij}. Each node $i \in V$ has a signed *supply* value b_i. The single-commodity, linear minimum-cost network flow (MCF) problem is defined as

$$\min \sum_{ij \in A} c_{ij} x_{ij}, \tag{1a}$$

subject to

$$\sum_{j:ij \in A} x_{ij} - \sum_{j:ji \in A} x_{ji} = b_i \quad \forall i \in V, \tag{1b}$$

$$0 \leq x_{ij} \leq u_{ij} \quad \forall ij \in A. \tag{1c}$$

We refer to (1b) as *flow conservation constraints* and (1c) as *capacity constraints*. We assume that all data are integer and wish to find an integer-valued optimal solution. Without loss of generality, we may further assume that all arc capacities are finite, all arc costs are nonnegative, and the problem has a feasible solution (see [3]). It also implies that $\sum_{i \in V} b_i = 0$.

A *pseudoflow* is a function x defined on the arcs that satisfies the capacity constraints (1c), but may violate (1b). A feasible flow is also a pseudoflow. The *excess* value of a node i with respect to a pseudoflow x is defined as

$$e_i = b_i + \sum_{j:ji \in A} x_{ji} - \sum_{j:ij \in A} x_{ij}. \tag{2}$$

Node i is referred to as an *excess node* if $e_i > 0$ and as a *deficit node* if $e_i < 0$. Note that $\sum_{i \in V} e_i = \sum_{i \in V} b_i = 0$.

Given a pseudoflow x, the corresponding *residual network* $G_x = (V, A_x)$ is defined as follows. For each original arc $ij \in A$, corresponds a *forward arc* $ij \in A_x$ with residual capacity $r_{ij} = u_{ij} - x_{ij}$ and cost c_{ij} if $r_{ij} > 0$; and a *backward arc* $ji \in A_x$ with residual capacity $r_{ji} = x_{ij}$ and cost $-c_{ij}$ if $r_{ji} > 0$.

The linear programming (LP) dual solution of the MCF problem is represented by *node potentials* π_i ($i \in V$). The *reduced cost* of an arc ij, with respect to a potential function π, is defined as $c^{\pi}_{ij} = c_{ij} + \pi_i - \pi_j$.

Optimality criteria can be defined for the MCF problem as follows. These results are discussed in [3,38,57,73,96].

THEOREM 2.1 Negative cycle optimality conditions *A feasible solution x of the MCF problem is optimal if and only if the residual network G_x contains no directed cycle of negative total cost.*

An equivalent formulation can be stated using node potentials and reduced costs.

THEOREM 2.2 Reduced cost optimality conditions *A feasible solution x of the MCF problem is optimal if and only if for some node potential function π, $c^{\pi}_{ij} \geq 0$ holds for each arc ij in the residual network G_x.*

The concept of *approximate optimality* is also essential for some algorithms. For a given $\epsilon \geq 0$, a pseudoflow x is called *ϵ-optimal* if for some node potential function π, $c^{\pi}_{ij} \geq -\epsilon$ holds for each arc ij in the residual network G_x. If the arc costs are integer and $\epsilon < 1/n$, then an ϵ-optimal feasible flow is indeed optimal. Furthermore, for a non-optimal feasible solution x, the smallest $\epsilon \geq 0$ for which x is ϵ-optimal equals to the negative of the minimum-mean cost of a directed cycle in the residual network G_x (see, e.g. [3,40]).

The MCF problem and its solution methods have been the object of intensive research for more than 50 years. Dantzig was the first to solve a special case of the problem, the so-called transportation problem, by specializing his well-known simplex method. Later, he also applied this approach directly to the MCF problem and devised the network simplex algorithm (see [28]). Ford and Fulkerson [38] developed the first combinatorial algorithms by generalizing Kuhn's remarkable Hungarian method [74].

Various algorithms were proposed in the next few years, but they do not run in polynomial time. Edmonds and Karp [35] introduced the scaling technique and developed the first weakly polynomial-time algorithm. The first strongly polynomial method is due to Tardos [102]. These results were followed by many other algorithms of improved running time bounds, most of which rely on the scaling technique.

Efficient implementation and profound computational evaluation of MCF algorithms have also been of high interest (see, e.g. [6,14–18,20,22,39,48,49,51,59,61,77,89,91,93]). The network simplex algorithm became quite popular when spanning tree labelling techniques were developed to improve its practical performance. Later, implementations of relaxation and cost-scaling algorithms also turned out to be highly efficient.

Table 1 provides a complexity survey of MCF algorithms, which was made according to [3,96], and the papers referenced in the table.

Table 1. Complexity survey of MCF algorithms.

$O(n^4CU)$	Ford and Fulkerson [37]
$O(m^3U)$	Yakovleva [109], Minty [81], Fulkerson [43], *out-of-kilter*
$O(n^2mU)$	Jewell [66], Busacker and Gowen [23], Iri [65], *successive shortest path*
$O(nm^2CU)$	Klein [71], *cycle-canceling*
$O(nU \cdot \mathrm{SP}_+(n,m,nC))$[a]	Edmonds and Karp [34,35], Tomizawa [105], *successive shortest path (using potentials)*
$O(m \log U \cdot \mathrm{SP}_+(n,m,nC))$[a]	Edmonds and Karp [35], *capacity-scaling*
$O(n \log C \cdot \mathrm{MF}(n,m,U))$	Röck [95], Bland and Jensen [16,17], *cost-scaling*
$O(m^2 \log n \cdot \mathrm{MF}(n,m,U))$	Tardos [102]
$O(m^2 \log n \cdot \mathrm{SP}_+(n,m))$	Orlin [83], Fujishige [42]
$O(n^2 \log n \cdot \mathrm{SP}_+(n,m))$	Galil and Tardos [45,46]
$O(n^2m \log(nC))$	Goldberg and Tarjan [52,56], *cost-scaling (generic version)*
$O(n^3 \log(nC))$	Goldberg and Tarjan [52,56], Bertsekas and Eckstein [12], *cost-scaling*
$O(n^{5/3}m^{2/3} \log(nC))$	Goldberg and Tarjan [52,56], *cost-scaling*
$O(nm \log n \log(nC))$	Goldberg and Tarjan [52,56], *cost-scaling (with dynamic trees)*
$O(n^2m^2 \min\{\log(nC), m \log n\})$	Goldberg and Tarjan [53,55], *minimum-mean cycle-canceling*
$O(n^2m \min\{\log(nC), m \log n\})$	Goldberg and Tarjan [53,55], *cancel-and-tighten*
	Orlin [86], *primal network simplex*
$O(nm \log n \min\{\log(nC), m \log n\})$	Goldberg and Tarjan [53,55], *cancel-and-tighten (with dynamic trees)*
	Tarjan [104], *primal network simplex (with dynamic trees)*
$O(m \log n \cdot \mathrm{SP}_+(n,m))$[a]	Orlin [84,85], *enhanced capacity-scaling*
	Vygen [106,107]
$O(nm \log\log U \log(nC))$[a]	Ahuja *et al.* [1,2], *double scaling*
$O((m^{3/2}U^{1/2} + mU \log(mU)) \log(nC))$[a]	Gabow and Tarjan [44]
$O((nm + mU \log(mU)) \log(nC))$[a]	Gabow and Tarjan [44]
$O(n^3 \min\{\log(nC), m \log n\})$	Goldberg and Tarjan [56], *generalized cost-scaling*
$O(nm \log(n^2/m) \min\{\log(nC), m \log n\})$[a]	Goldberg and Tarjan [56], *generalized cost-scaling (with dynamic trees)*
$O(m(m + n \log n) \min\{\log(nU), m \log n\})$	Orlin *et al.* [87], *dual network simplex*
	Sokkalingam *et al.* [100], *scaling cycle-cancelling*
$O(nm \log n(m + n \log n))$	Armstrong and Jin [5], *dual network simplex*

Notes: n and m denote the number of nodes and arcs, respectively; U denotes the maximum of supply values and arc capacities; C denotes the largest arc cost. $\mathrm{SP}_+(n,m,C)$ denotes the running time of any algorithm solving the single-source shortest path problem in a directed graph with n nodes, m arcs, and nonnegative integer arc lengths, each at most C. $\mathrm{SP}_+(n,m)$ denotes the strongly polynomial complexity for solving the single-source shortest path problem with nonnegative arc lengths. Dijkstra's algorithm with Fibonacci heaps provides an $O(m + n \log n)$ bound for $\mathrm{SP}_+(n,m)$ (see [41]). $\mathrm{MF}(n,m,U)$ denotes the running time of any algorithm solving the maximum flow problem in a directed graph with n nodes, m arcs, and integer capacities, each at most U.

[a]An asymptotically best complexity bound.

3. Implementations of MCF algorithms

In this section, we first introduce the MCF algorithms implemented by the author in the LEMON library, then we briefly discuss all other solvers used in our experiments. For a comprehensive study of MCF algorithms, see the book of Ahuja *et al.* [3].

3.1 *MCF algorithms in the LEMON library*

LEMON [76] is an open-source C++ template library with focus on combinatorial optimization tasks related mainly to graphs and networks. Its name is an abbreviation of *Library for Efficient Modeling and Optimization in Networks*. LEMON provides highly efficient implementations of graph algorithms and related data structures, which can be combined easily to solve complex real-life optimization problems. The LEMON project is maintained by the MTA–ELTE Egerváry Research Group on Combinatorial Optimization (EGRES) [36] at the Department of Operations Research, Eötvös Loránd University, Budapest, Hungary. The library is also a member of the Computational Infrastructure for Operations Research (COIN–OR) project [25], a collection of open-source projects related to operations research. LEMON can be used in both commercial and non-commercial software development under very permissive license terms. For more information about LEMON, the readers are referred to the introductory paper [31] and to the web site of the library: http://lemon.cs.elte.hu/.

LEMON provides implementations of several MCF algorithms, which are presented in this section along with the applied data structures and heuristics. For a more detailed discussion, see [70].

3.1.1 *Simple cycle-cancelling algorithm*

Cycle-cancelling is a general primal method for solving the MCF problem, originally proposed by Klein [71]. A feasible solution x is first established, which can be carried out by a maximum flow computation. After that, the algorithm throughout maintains feasibility and gradually decreases the total cost of the solution. At each iteration, a directed cycle of negative cost is found in the current residual network G_x, and it is cancelled by augmenting flow along it. When G_x contains no negative-cost cycle, the algorithm terminates with an optimal solution found (see Theorem 2.1).

Many variants of this approach have been developed by applying different rules for cycle selection (see, e.g. [7,55,100,108]). These algorithms have quite different theoretical and practical behaviour; some of them run in polynomial or even strongly polynomial time. Moreover, the primal network simplex algorithm (see Section 3.1.7) can also be viewed as a particular variant of the cycle-cancelling method.

LEMON provides three cycle-cancelling algorithms. One of them, which we call *simple cycle-cancelling* (SCC), identifies negative cycles using the Bellman–Ford algorithm with successively increased limit on the number of its iterations. The worst-case time complexity of this method is $O(nm^2CU)$ (using the notations defined in Section 2).

3.1.2 *Minimum-mean cycle-cancelling algorithm*

Goldberg and Tarjan [53,55] devised this famous special variant of the cycle-cancelling method, which selects a negative cycle of minimum mean cost at each iteration. It is the simplest strongly polynomial-time MCF algorithm; it performs $O(nm^2 \log n)$ iterations for arbitrary real-valued arc costs and $O(nm \log(nC))$ iterations for integer arc costs (see, e.g. [3,55,57,73,96]).

The minimum-mean cycle-cancelling (MMCC) algorithm relies on finding minimum-mean directed cycles in a graph. This optimization problem has also been studied for a long time,

and various algorithms have been developed for solving it. Comprehensive experimental studies can be found in [29,30,47]. In the MMCC implementation, we use a combination of Howard's policy-iteration algorithm (see [29,62]) and the algorithm of Hartmann and Orlin [60]. This combined method is quite efficient in practice and also ensures the best strongly polynomial time bound $O(nm)$. Therefore, the overall complexity of the MMCC algorithm is $O(n^2m^2 \min\{\log(nC), m \log n\})$ for the MCF problem with integer data.

3.1.3 *Cancel-and-tighten algorithm*

This algorithm is also due to Goldberg and Tarjan [53,55]. It improves on the MMCC method by maintaining node potentials to make the detection of negative residual cycles easier and faster. The cancel-and-tighten (CAT) algorithm proceeds by ensuring ϵ-optimality of the current solution for successively smaller values of $\epsilon \geq 0$ until it becomes optimal. At every iteration, two main steps are performed. In the *cancel* step, directed cycles consisting entirely of arcs with negative reduced cost are successively cancelled until no such cycle exists. In the *tighten* step, the node potentials are modified in order to introduce new arcs of negative reduced cost and to decrease ϵ to at most $(1 - 1/n)$ times its former value.

We implemented the cancel step using a straightforward method, which is based on a depth-first search and has a worst-case complexity of $O(nm)$. Although it could be improved to run in $O(m \log n)$ time using dynamic tree data structures (see [55] and [99]), we did not implement this variant since the cycle-cancelling algorithms turned out to be slow in practice.

The tighten step can be carried out in $O(m)$ time using a topological ordering of the nodes. However, this method does not ensure strongly polynomial bound on the number of iterations. Therefore, after every k iterations, the parameter ϵ is set to its smallest possible value using a minimum-mean cycle computation. Goldberg and Tarjan [55] suggest to use $k = n$ (to preserve the amortized running time $O(m)$ of the tighten step), but we use $k = \lfloor \sqrt{n} \rfloor$ since it turned out to be more efficient in practice. The minimum-mean cycle computations are carried out using the same combined method that was applied in the MMCC algorithm.

The CAT algorithm performs $O(n \min\{\log(nC), m \log n\})$ iterations, and thus our implementation runs in $O(n^2m \min\{\log(nC), m \log n\})$ time for the MCF problem with integer data.

3.1.4 *Successive shortest path algorithm*

This is another fundamental MCF algorithm, which solves the problem by a sequence of shortest path computations. The initial version of this method was devised independently by Jewell [66], Iri [65], and Busacker and Gowen [23]. Later, Edmonds and Karp [34,35] and Tomizawa [105] independently suggested the utilization of node potentials to ensure nonnegative arc costs throughout the algorithm, which greatly improves its performance both in theory and in practice.

The successive shortest path (SSP) algorithm applies a dual approach. It maintains an optimal pseudoflow along with corresponding node potentials and attempts to achieve feasibility. At each iteration, it selects a node v with $e_v > 0$ and augments flow from node v to a node w with $e_w < 0$ along a shortest path in the residual network with respect to the reduced arc costs. The node potentials are then increased by the computed shortest path distances to preserve the reduced cost optimality conditions (see Theorem 2.2). The algorithm terminates after $O(nU)$ iterations.

The practical performance of the SSP algorithm mainly depends on the shortest path computations. Our implementation uses Dijkstra's algorithm with binary heap data structure (other heap structures were also tested, but they did not turn out to be more robust in practice). Therefore, an iteration is performed in $O(m \log n)$ time and the overall worst-case complexity of our SSP code is

$O(nmU \log n)$. An essential improvement of the algorithm is also applied as it is discussed in [3]. At every iteration, Dijkstra's algorithm is terminated once it permanently labels the deficit node w, and the potentials are modified accordingly. Furthermore, we use an efficient representation of the residual network that allows fast traversal of the outgoing arcs of a node, which is crucial for the shortest path computations.

3.1.5 *Capacity-scaling algorithm*

This is an improved version of the SSP method, which was developed by Edmonds and Karp [35] as the first weakly polynomial-time MCF algorithm. This method performs capacity-scaling phases to ensure that the path augmentations carry sufficiently large amounts of flow, which often reduces the number of iterations. In a Δ-scaling phase, each path augmentation delivers Δ units of flow from a node v with $e_v \geq \Delta$ to a node w with $e_w \leq -\Delta$. When no such augmenting path is found, the value of Δ is halved, and the algorithm proceeds with the next phase. At the end of the phase with $\Delta = 1$, an optimal solution is found.

The capacity-scaling (CAS) algorithm performs $O(m \log U)$ iterations under the additional assumption that a directed path of sufficiently large capacity exists between each pair of nodes. Although this condition can easily be achieved by a simple extension of the underlying network, we decided to avoid this transformation. As a result, more units of excess may remain at the end of a Δ-scaling phase, but we allow the path augmentations of subsequent phases to deliver more units of flow than the current value of Δ. The polynomial running time is thereby not guaranteed, but our experiments showed that this version does not perform more path augmentations and runs significantly faster in practice.

Our implementation uses a scaling factor α other than two, that is, Δ is initially set to $\alpha^{\lfloor \log_\alpha U \rfloor}$ and divided by α at the end of each phase. We use $\alpha = 4$ by default, as it turned out to provide the best overall performance. The CAS algorithm has much in common with the SSP method, hence similar data structures and algorithmic improvements are applied in its implementation.

3.1.6 *Cost-scaling algorithm*

The cost-scaling (COS) technique was first proposed independently by Röck [95] and Bland and Jensen [16,17]. Goldberg and Tarjan [52,56] improved on these methods by also utilizing the concept of ϵ-optimality and developed an algorithm that has turned out to be among most efficient ones, both in theory and in practice.

The COS algorithm performs ϵ-phases for successively smaller values of ϵ until $\epsilon < 1/n$. In each phase, a *refine* procedure is applied to transform an ϵ-optimal solution into an (ϵ/α)-optimal solution for a given scaling factor $\alpha > 1$.

The refine procedure first converts the ϵ-optimal flow x to an optimal pseudoflow. The value of ϵ is then divided by α and the pseudoflow x is gradually transformed into a feasible flow again, but in a way that preserves ϵ-optimality for the new value of ϵ. This is achieved by performing a sequence of local *push* and *relabel* operations similarly to the well-known push-relabel algorithm for the maximum flow problem, which is also due to Goldberg and Tarjan [54].

Given a pseudoflow x and a potential function π, a node i is called *active* if $e_i > 0$ and a residual arc ij is called *admissible* if $c_{ij}^{\pi} < 0$. A basic operation selects an active node v and either *pushes* flow on an admissible residual arc vw or if no such arc exists, *relabels* node v. It means that the potential of node v is decreased by the largest possible amount that does not violate the ϵ-optimality conditions. This operation introduces new admissible outgoing arcs at node v and thereby allows subsequent push operations to carry the remaining excess of this node. The refine

procedure terminates when no active node remains in the network, and hence an ϵ-optimal feasible solution is obtained (since the basic operations preserve ϵ-optimality).

The generic version of the refine procedure takes $O(n^2m)$ time. As the number of ϵ-scaling phases is $O(\log(nC))$, the COS algorithm runs in weakly polynomial time $O(n^2m\log(nC))$. Goldberg and Tarjan [56] also developed special versions of improved running time bounds by applying particular selection rules of the basic operations and using complex data structures. The best variant runs in strongly polynomial time $O(nm\log(n^2/m)\min\{\log(nC), m\log n\})$.

We implemented three variants of the COS algorithm that perform the refine procedure rather differently. We first discuss the standard push-relabel implementation, and only the differences are presented for the other two variants.

3.1.6.1 *Push-relabel method.* This implementation is based on the generic version of the COS algorithm and hence performs local push and relabel operations. Its practical performance highly depends on the application of various effective heuristics and other improvements. Such ideas are discussed in [22,49,51]. Most of these improvements are analogous to similar techniques devised for the push-relabel maximum flow algorithm (see, e.g. [24]). In the followings, we briefly present the improvements we applied.

The bottleneck of the COS algorithm corresponds to searching for admissible arcs for the basic operations. Therefore, we apply the same representation of the residual network as for the SSP and CAS algorithms. Furthermore, we maintain a current arc pointer for each node, which is a standard technique for making these arc searches even faster.

We also apply a well-known technique to overcome the issue that the generic COS algorithm performs internal computations with non-integer values of ϵ and non-integer node potentials. All arc costs are multiplied by αn for a given integer scaling factor $\alpha \geq 2$, and ϵ is also scaled accordingly. Initially, ϵ is set to $\alpha^{\lceil \log_\alpha(\alpha nC)\rceil}$ and is divided by α in each phase until $\epsilon = 1$. Thereby, it is ensured that all computations operate solely on integer numbers, which improves the performance of the operations and avoids potential numerical issues.

The optimal value of α depends on the heuristics applied in the algorithm and on the problem instance, as well. Our experiments showed that its optimal value is usually between 8 and 24, and the differences are typically moderate, thus we use $\alpha = 16$ by default.

The strategy for selecting an active node for the next basic operation is essential. We experimented with a few variants, including the so-called wave implementation, but the simple FIFO selection rule turned out to be the most robust (as it is also observed in [22,49]).

In addition to these improvements, we also applied three complex heuristics out of the four proposed in [49,51].

The *potential refinement* (or *price refinement*) heuristic introduces an additional step at the beginning of each phase to check if the current solution is already (ϵ/α)-optimal. This step attempts to adjust the potentials to satisfy the (ϵ/α)-optimality conditions, but without modifying the flow. If it succeeds, the refine procedure is skipped and the next phase begins. Our results also verified that this heuristic substantially improves the overall performance of the algorithm in most cases.

Another variant of this heuristic performs a minimum-mean cycle computation at the beginning of each phase to determine the smallest ϵ for which the current flow is ϵ-optimal and computes corresponding node potentials. However, our experiments showed that an efficient implementation of the former variant of the heuristic is clearly superior to this one, which contradicts the results of [22].

The *global update* heuristic performs relabel operations on several nodes at once by iteratively applying the following operation. Let $S \subset V$ denote a set of nodes such that all deficit nodes are in S, but at least one active node is in $V \setminus S$. If no admissible arc enters S, then the potential of every node in S can be increased by ϵ without violating the ϵ-optimality conditions. This

heuristic turned out to greatly improve the efficiency of the COS algorithm on some problem classes, although it does not help to much or even slightly worsens the performance on other instances.

The *push-look-ahead* heuristic attempts to avoid pushing flow from node i to node j when a subsequent push operation is likely to send this amount of flow back to node i. To achieve this, only a limited amount of flow is allowed to be pushed into a node j, and the relabel operation is extended to those nodes at which this limitation is applied regardless of their current excess values. In most cases, this heuristic significantly decreases the number of push operations.

Another heuristic, the *speculative arc fixing*, is also proposed in [49,51]. Although this heuristic would most likely improve the performance of our implementation, as well, we did not implement it until now, mainly because it seems to be rather involved and sensitive to parameter settings.

3.1.6.2 *Augment-relabel method.* This variant of the COS algorithm performs path augmentations instead of local push operations, but relabelling is heavily used to find augmenting paths. At each step of the refine procedure, this method selects an active node v and performs a depth-first search on the admissible arcs to find an augmenting path to a deficit node w. Whenever the search process has to step back from a node i, then this node is relabelled.

The flow augmentation on the admissible paths can be performed in two natural ways. We can either push the same amount of flow on each arc of the path or push the maximum possible amount of flow on each arc independently. According to our experiments, the latter variant performs slightly better, thus it is applied in our implementation.

Note that these path search and flow augmentation procedures correspond to particular sequences of local push and relabel operations. However, the actual push operations are carried out in a delayed and more guided manner, in aware of an admissible path to a deficit node. This helps to avoid such problems for which the push-look-ahead heuristic is devised (see above), but a lot of work may be required to find augmenting paths, especially if they are long.

In this implementation, we used the same data representation, improvements, and heuristics as for the push-relabel variant except for the push-look-ahead heuristic.

3.1.6.3 *Partial augment-relabel method.* The third implementation of the COS algorithm can be viewed as an intermediate approach between the other two variants. It is based on the partial augment-relabel algorithm recently proposed by Goldberg [50] as a novel variant of the push-relabel algorithm for solving the maximum flow problem. As this method turned out to be highly efficient and robust in practice, Goldberg also suggests the utilization of the idea in the MCF context, but he does not investigate it. Our implementation of the COS algorithm, to the author's knowledge, is the first to incorporate this technique.

The partial augment-relabel implementation of the COS algorithm is quite similar to the augment-relabel variant, but it limits the length of the augmenting paths. The path search process is stopped either if a deficit node is reached or if the length of the path reaches a given parameter $k \geq 1$. In fact, the push-relabel and augment-relabel variants are special cases of this approach with $k = 1$ and $k = n - 1$, respectively. Our code uses $k = 4$ by default, just like Goldberg's maximum flow implementation, as it results in a quite robust algorithm in the MCF context, as well.

Apart from the length limitation for the augmenting paths, this variant is exactly the same as the augment-relabel method. (Actually, they have a common implementation but with different values of the parameter k.) However, the partial augment-relabel technique attains a good compromise between the former two approaches and turned out to be superior to them. We refer to this implementation as COS in the followings.

3.1.7 *Network simplex algorithm*

The primal network simplex (NS) algorithm is one of the most popular solution methods for the MCF problem. It is a specialized version of the well-known LP simplex method that exploits the network structure of the MCF problem and performs the basic operations directly on the graph representation. The LP variables correspond to the arcs of the graph, and the LP bases are represented by spanning trees.

The NS algorithm was devised by Dantzig [27,28], the inventor of the LP simplex method. Although the generic version of the algorithm does not run in polynomial time, it turned out to be rather effective in practice. Therefore, subsequent research has focused on efficient implementation of the NS algorithm (see [9,20,48,59,69,77,101]). Furthermore, researchers also developed particular variants of both the primal and the dual network simplex methods that run in polynomial time (see [5,58,83,86–88,103,104]). Detailed discussion of the NS method can be found in, for example, [3] and [68].

The NS algorithm is based on the concept of *spanning tree solutions*. Such a solution is represented by a partitioning of the arc set A into three subsets (T, L, U) such that each arc in L has flow fixed at zero, each arc in U has flow fixed at the capacity of the arc, and the arcs in T form an (undirected) spanning tree of the network with flow values not restricted to the lower or upper bound. If an instance of the MCF problem has an optimal solution, then it also has an optimal spanning tree solution, which can be found by successively transforming a spanning tree solution to another one. These solutions actually correspond to the LP basic feasible solutions of the problem.

The LP simplex method maintains a basic feasible solution and gradually improves its objective function value by small transformations, known as pivots. Accordingly, the primal NS algorithm throughout maintains a spanning tree solution of the MCF problem and successively decreases the total cost of the flow until it becomes optimal. Node potentials are also maintained such that the reduced cost of each arc in the spanning tree is zero. At each iteration, a non-tree arc violating the optimality conditions is added to the current spanning tree, which uniquely determines a negative-cost residual cycle. This cycle is cancelled by augmenting flow on it and a tree arc corresponding to a saturated residual arc is removed from the tree. The tree structure is then updated, and the node potentials are adjusted accordingly. This whole operation transforming a spanning tree solution to another one is called pivot. If no suitable entering arc can be found, the current flow is optimal and the algorithm terminates.

In fact, the NS algorithm can also be viewed as a particular variant of the cycle-cancelling algorithm. Due to the sophisticated method of maintaining spanning tree solutions, however, a negative cycle can be found and cancelled much faster (in $O(m)$ time).

Similarly to the LP simplex method, degeneracy is also a crucial issue for the NS algorithm. In many cases, a cycle of zero residual capacity is found in a pivot step, and hence the flow cost cannot be decreased, only the spanning tree is modified. Degeneracy not only worsens the performance of the algorithm, but its finiteness is even not guaranteed. A simple and popular technique to overcome these issues is to ensure that the spanning tree solution is strongly feasible at each iteration. It means that a positive amount of flow can be sent from each node to a designated root node of the spanning tree along the tree path. For the details, see [3,8,26].

3.1.7.1 *Spanning tree data structures.* The representation of spanning tree solutions is essential to implement the NS algorithm efficiently. Several storage schemes have been developed for this purpose along with efficient methods for updating them during the pivot operations (see [3,9,48,67,69,101]).

We implemented the primal NS algorithm with two spanning tree storage techniques. The first one is usually referred to as the *Augmented Threaded Index* (ATI) method and represents a

spanning tree as follows. The tree has a designated root node, and three indices are stored for each node: the depth of the node in the tree, the index of its parent node, and a so-called thread index that is used to define a depth-first traversal of the spanning tree. It is a quite popular method, which is discussed in detail in [68] and [3].

The ATI technique has an improved version, which is due to Barr *et al.* [9] and is usually referred to as the *eXtended Threaded Index* (XTI) method. The XTI scheme replaces the depth index by two indices for each node: the number of successors of the node in the tree and the last successor of the node according to the traversal defined by the thread indices. This modification allows faster update process since a tree alteration of a single pivot usually modifies the depth of several nodes in subtrees that are moved from a position to another one, while the set of successors is typically modified only for much fewer nodes.

We implemented the XTI scheme with an additional improvement that a reverse thread index is also stored for each node to represent the depth-first traversal as a doubly linked list. This modification turned out to substantially improve the performance of the update process. In fact, the inventors of the XTI technique also discussed this extension [9], but they did not apply it in order to reduce the memory requirements of the representation.

Although the XTI labelling method is not as widely known and popular as the simpler ATI method, our experiments showed that it is much more efficient on all problem instances, so the final version of our code only implements the XTI technique.

3.1.7.2 *Initialization.* The NS algorithm also requires an initial spanning tree solution to start with. It is possible to transform any feasible solution to a spanning tree solution, and the required tree indices can also be computed by a depth-first traversal of the tree arcs. However, artificial initialization is more common in practice. It means that the underlying network is extended with an artificial root node and additional arcs connecting this node and the original nodes in a way that a strongly feasible spanning tree solution can easily be constructed. We apply this method as it is much easier and usually provides better overall performance than starting with a non-artificial solution.

3.1.7.3 *Pivot rules.* One of the most crucial aspects of the NS algorithm is the selection of entering arcs for the pivot operations. The applied method affects the 'goodness' of the entering arcs and thereby the number of iterations as well as the average time required for selecting an entering arc, which is a dominant part of each iteration.

We implemented five pivot rules, which are briefly discussed in the followings. A non-tree arc ij is called *eligible* if it violates the optimality conditions, that is, either $x_{ij} = 0$ and $c^{\pi}_{ij} < 0$ or $x_{ij} = u_{ij}$ and $c^{\pi}_{ij} > 0$ with respect to the current potential function π. We refer to $|c^{\pi}_{ij}|$ as the *violation* of arc ij.

The two simplest pivot rules are the *best eligible* and the *first eligible* strategies. The former one always selects an eligible arc of maximum violation to enter the tree, while the latter one examines the arcs cyclically and selects the first eligible arc at each iteration. Neither of them provides good overall performance in practice.

The *block search pivot rule* was proposed by Grigoriadis [59]. This method cyclically examines certain subsets (blocks) of the arcs and selects the best eligible candidate among these arcs at each iteration. The block size B is an important parameter of this method. In fact, the previous two rules are special cases of this one with $B = m$ and $B = 1$, respectively. Several sources suggest to set B proportionally to the number of arcs, for example, between 1% and 10% [59,68]. However, our experiments showed that a much robust implementation can be achieved using $B = \lfloor \sqrt{m} \rfloor$.

The *candidate list pivot rule* is another well-known approach, which was proposed by Mulvey [82]. In a so-called major iteration, this method examines the arcs cyclically to build a list containing at most L eligible arcs. This list is then used by at most K subsequent iterations to select

an arc of maximum violation among the candidates. If an arc becomes non-eligible, it is removed from the list. We obtained the best average running time using $L = \lfloor\sqrt{m}/4\rfloor$ and $K = \lfloor L/10\rfloor$.

We also developed an improved version of the previous method, which we call *altering candidate list pivot rule*. It also maintains a candidate list, but it attempts to extend this list at each iteration and keeps only the several best candidates for the next one. At least one arc block of size B is examined at every iteration to extend the list with new eligible arcs. After that, an arc of maximum violation is selected to enter the basis, and the list is truncated to contain at most H of the best candidate arcs. According to our measurements, $B = \lfloor\sqrt{m}\rfloor$ and $H = \lfloor B/100\rfloor$ result in a rather robust implementation.

Our experimental results showed that the block search and the altering candidate list pivot rules are the most efficient for all classes of problem instances. Since the block search rule is simpler and turned out to be slightly more robust, it is our default pivot strategy, and we refer to this variant as NS in the followings.

3.2 *Other MCF solvers*

In the followings, we briefly introduce the MCF solvers that are compared to the implementations provided by the LEMON library. The MCFClass project [10] was of great help for us in these experiments. This project features a common and flexible C++ interface for several MCF solvers, which are all considered in this paper. However, MCFClass does not support the latest versions of CS2 and MCFZIB, thus we used these two solvers directly.

3.2.1 *CS2*

CS2 is a highly robust, authoritative implementation of the cost-scaling push-relabel algorithm. It was written in C language by A.V. Goldberg and B. Cherkassky applying the improvements and heuristics described in [49,51]. CS2 has been widely used as a benchmark for solving the MCF problem for a long time (see, e.g. [39,49,77]). We used the latest version, CS2 4.6, which is available from the IG Systems, Inc. [64]. CS2 software can be used for academic research and evaluation purposes free of charge, but commercial use requires a license.

3.2.2 *LEDA*

LEDA is a comprehensive commercial C++ library providing efficient implementations of various data types and algorithms [4,80]. It is widely used in many application areas, such as telecommunication, scheduling, geographic information science (GIS), traffic planning, very-large-scale integration (VLSI) design, and computational biology. The `MIN_COST_FLOW()` procedure of the LEDA library implements the cost-scaling push-relabel algorithm similarly to CS2. We used LEDA 5.0 in our experiments. In fact, LEDA 5.1.1 was also tested, but its MCF method turned out to be slower than that of version 5.0.

3.2.3 *MCFZIB*

MCF is a network simplex code written in C language by Löbel [77] at the Zuse Institute Berlin (ZIB). We denote this solver as MCFZIB in order to differentiate it from the problem itself (similarly to the MCFClass project [10]). This solver features both a primal and a dual network simplex implementation, from which the former one is used by default as it tends to be more efficient. It applies a usual data structure for representing the spanning tree solutions along with an improved version of the candidate list pivot rule, which is called *multiple partial pricing* (see [77]

for the details). This implementation has been shown to be rather efficient (see, e.g. [39,77]). We used the latest version, MCF 1.3, which is available for academic use free of charge, see [78].

3.2.4 *CPLEX*

IBM ILOG CPLEX Optimization Studio [63], usually referred to simply as CPLEX, is a well-known and powerful software package aimed at large-scale optimization problems. It provides efficient methods for solving different kinds of mathematical programming problems, including linear, mixed integer, and convex quadratic programming. The NETOPT module of CPLEX implements the primal network simplex algorithm for solving the MCF problem. We used this component through a 'wrapper class' called MCFCplex, which is provided by the MCFClass project [10]. MCFCplex implements the common interface defined in this project using the CPLEX Callable Library. Although CPLEX is a comprehensive commercial software, it is available free of charge for academic use and for non-commercial research. We used its latest version, 12.4.

3.2.5 *MCFSimplex*

MCFSimplex is a recent, open-source software written by A. Bertolini and A. Frangioni in C + + language. It is available as part of the MCFClass project [10] and can be used under the flexible LGPL license. This code implements both the primal and the dual network simplex algorithms, from which the former one is the default, as it is usually faster and more robust. MCFSimplex is reported to be reasonably efficient compared to other network simplex implementations, moreover, it can solve separable quadratic MCF problems, as well. According to our experiments, the dual version of MCFSimplex turned out to be superior to the dual version of MCFZIB on the majority of test instances, thus only the former one is considered in the rest of this paper. We abbreviate the primal and dual versions of MCFSimplex as MSim and MSim-D, respectively.

3.2.6 *RelaxIV*

RelaxIV is an efficient, authoritative implementation of the relaxation algorithm [14]. The original FORTRAN code was written by Bertsekas and Tseng [13,15] and is available at [11]. We used a C + + translation of this code, which was made by A. Frangioni and C. Gentile and is available as part of the MCFClass project [10]. Similarly to CS2, the RelaxIV solver and its previous versions have been used in experimental studies for a long time (see, e.g. [13,15,39,49,77]).

3.2.7 *PDNET*

PDNET implements the truncated primal-infeasible dual-feasible interior-point algorithm for solving linear network flow problems. It was written in Fortran and C language by Portugal *et al.* [89,91]. This solver can be used free of charge, its source code is available at [90]. For more information about interior-point network flow algorithms, see, for example, [92–94].

3.3 *Integrality and numerical stability*

This subsection briefly discusses the numerical aspects of MCF algorithms. In this study, we throughout assume that all input data are integer, which is a typical and particularly important case of the MCF problem. Actually, this assumption is not really restrictive in most applications, since computers work with only rational numbers (up to a limited precision), which can always be converted to integers by multiplying them by a suitably large constant.

It is proved that in case of integer input data, if an instance of the MCF problem has optimal solution, then it always has integer optimal primal and dual solutions, as well (see, e.g. [3]). Moreover, most algorithms can further exploit this integrality assumption by working solely with integer numbers. Thereby, they can compute exact optimal solutions and can avoid potential numerical problems.

On the other hand, some solution methods are inherently continuous even if all input data are integer. For example, interior-point methods perform most of their internal computations involving floating-point numbers and thereby cannot fully exploit the integrality of the input. These methods typically apply numerical tolerances for inexact computations. However, other techniques may also be feasible in some algorithms. A particular example is the cost-scaling algorithm, in which the integrality of all internal data can be ensured by multiplying the arc costs by a suitable constant, as it is described in Section 3.1.6.

In general, algorithms that can exploit integrality are numerically more stable than those that rely on floating-point arithmetic. In addition, operations on integers are usually faster than on floating-point numbers.

All MCF algorithms provided by the LEMON library assume integer input data and work solely with integer numbers. For some applications, however, it would be practical to enhance these implementations with direct support of real-valued input.

CS2 and LEDA also operate on integer numbers by default. MCFZIB, MCFSimplex, and RelaxIV support both integer and floating-point input, and the number types they use can be customized. In order to ensure fair comparison, these codes were also compiled using integer types. In contrast, PDNET inherently uses floating-point numbers and tolerances. Finally, CPLEX most likely operate on floating-point numbers, as well, but it does not provide options to change this.

4. Experimental study

This section presents an extensive computational analysis of all considered MCF solvers. First, we describe the problem families used for the experiments. In Section 4.2, we present a comparison of the algorithms implemented by the author in the LEMON library (see Section 3.1). After that, we focus on the most efficient ones among these codes and compare them to other publicly available solvers presented in Section 3.2.

This study improves on [70] by providing a more comprehensive evaluation of the fastest implementations, including four additional solvers: CPLEX, both versions of MCFSimplex, and PDNET. Although empirical analysis of MCF algorithms has always been of high interest (see [6,14–18,20,22,39,48,49,51,59,61,77,93]), the author is not aware of any recent study that is comparably extensive and considers networks up to millions of nodes and arcs. Benchmark results on such large graphs, however, turned out to be essential to draw appropriate conclusions.

4.1 *Test setup*

Our test suite comprises numerous networks of various size and characteristics. It can be accessed at http://lemon.cs.elte.hu/trac/lemon/wiki/MinCostFlowData.

Most of the problem instances were generated with standard random generators NETGEN, GRIDGEN, GOTO, and GRIDGRAPH, which are available as source codes at the FTP site of the First DIMACS Implementation Challenge [32]. We used these generators with similar parameter settings as previous works (e.g. [15,18,22,39,49,51,77]), but we created larger networks, as well. Other problem families were also generated based on either real-life road networks or maximum flow problems arising in computer vision applications. All instances involve solely integer data.

We used LEMON 1.3 for this study, which is the latest release of the library. The versions of the other solvers we considered can be found in Section 3.2.

The presented experiments were conducted on a machine with AMD Opteron Dual Core 2.2 GHz CPU and 16 GB RAM (1 MB cache), running openSUSE 11.4 operating system. All codes were compiled with GCC 4.5.3 using -O3 optimization flag.

4.1.1 *NETGEN instances*

NETGEN is a classic and popular generator developed by Klingman *et al.* [32,72]. It produces random instances of the MCF problem and other network optimization problems. In fact, NETGEN is known to create relatively easy MCF instances.

We generated NETGEN networks as follows. Arc capacities and costs are selected uniformly at random from the ranges $[1 \dots 1000]$ and $[1 \dots 10,000]$, respectively. The number of supply nodes and the number of demand nodes are both set to $\sqrt{n}$ (rounded to integer). The five problem families we used differ in the density of the networks and the average amount of supply per supply node, which is denoted by S.

- *NETGEN–8.* Sparse networks; $m = 8n$, $S = 1000$.
- *NETGEN–SR.* Dense networks; $m \approx n\sqrt{n}$, $S = 1000$.
- *NETGEN–LO–8.* Sparse networks with low supplies; $m = 8n$, $S = 10$.
- *NETGEN–LO–SR.* Dense networks with low supplies; $m \approx n\sqrt{n}$, $S = 10$.
- *NETGEN–DEG.* Networks of increasing density; $n = 4096$ and m ranges from $2n$ to n^2, $S = 1000$.

4.1.2 *GRIDGEN instances*

GRIDGEN is a random generator that produces grid-like networks. It was written by Lee and Orlin [32,75].

We used the same parameters for GRIDGEN families as for the corresponding NETGEN families with the additional setting that the width of the underlying grid is set to $\sqrt{n}$ for each GRIDGEN network. Actually, the shape of the grid did not turn out to be a significant parameter for GRIDGEN, since this generator selects the supply and demand nodes uniformly at random.

- *GRIDGEN–8.* Sparse networks; $m = 8n$, $S = 1000$.
- *GRIDGEN–SR.* Dense networks; $m \approx n\sqrt{n}$, $S = 1000$.
- *GRIDGEN–DEG.* Networks of increasing density; $n = 4096$ and m ranges from $4n$ to n^2, $S = 1000$.

4.1.3 *GOTO instances*

GOTO is another well-known generator for the MCF problem, which was developed by Goldberg [32,51] and intended to produce hard instances. The name of the generator stands for *Grid On Torus*, which reflects to the basic structure of the generated networks. Each GOTO instance has one supply node and one demand node and the supply value is adjusted according to the arc capacities.

As for the previous problem families, the maximum arc capacity is set to 1000 and the maximum arc cost is set to $10,000$.

- *GOTO–8.* Sparse networks; $m = 8n$.
- *GOTO–SR.* Dense networks; $m \approx n\sqrt{n}$.

4.1.4 *GRIDGRAPH instances*

GRIDGRAPH generator was written by Resende and Veiga [32,93]. It produces grid networks similarly to GRIDGEN and GOTO, but applying a stricter scheme. A GRIDGRAPH network consists of transshipment nodes forming a grid of W rows and L columns as well as a single source node s and a single sink node t. Arcs go from s to the nodes of the first column; from the nodes of the last column to t; and from each transshipment node (w, l) to nodes $(w + 1, l)$ and $(w, l + 1)$, except for the last row and column, respectively. The arc capacities and costs are set uniformly at random within a specified range.

We used three GRIDGRAPH families based on grids of different shape as it turned out to be a crucial parameter. The maximum arc capacity and cost are set to 1000 and 10, 000, respectively, just like for the previous generators.

- *GRID–WIDE*. $L = 16$ and W increases.
- *GRID–LONG*. $W = 16$ and L increases.
- *GRID–SQUARE*. $W = L$.

4.1.5 *ROAD instances*

Special MCF problems based on real-world road networks were also considered. To generate such instances, we used the TIGER/Line road network files of several states of the USA. These data files are available at the web site of the Ninth DIMACS Implementation Challenge [33].

We selected seven states having road networks of increasing size (namely, DC, DE, NH, NV, WI, FL, and TX) and generated MCF problem instances as follows. The original undirected graphs are converted to directed graphs by replacing each edge with two oppositely directed arcs. The cost of an arc is set to the travel time on the corresponding road section. The number of supply and demand nodes are both set to $\lfloor \sqrt{n}/10 \rfloor$. These nodes are selected randomly, and the supply-demand values are determined by a maximum flow computation that maximizes the total supply with respect to the fixed set of supply and demand nodes and fixed arc capacities.

We generated two problem families with different arc capacity settings.

- *ROAD–PATHS*. The arc capacities are uniformly set to one. This means that we are actually looking for a specified number of arc-disjoint directed paths from supply nodes to demand nodes having minimum total cost.
- *ROAD–FLOW*. The capacity of an arc is set 40, 60, 80, or 100 according to the category of the corresponding road section.

4.1.6 *VISION instances*

Our test suite also includes MCF instances based on large-scale maximum flow problems arising in computer vision applications. The corresponding data files were made available by the Computer Vision Research Group at the University of Western Ontario [79] for benchmarking maximum flow algorithms (see, e.g. [50]).

We used some of the segmentation instances related to medical image analysis, which are defined on three-dimensional grid networks. Those variants were selected in which the underlying graphs are 6-connected and the maximum arc capacity is 100 (the `bone_sub*_n6c100` files). These networks were converted to minimum-cost maximum flow problem instances applying different arc cost functions. The original networks also contain arcs of zero capacity, but we skipped these arcs during the transformation and thereby did not preserve the exact 6-connectivity.

- *VISION–RND*. The arc costs are selected uniformly at random from the range $[1 \ldots 100]$.

- *VISION–PROP.* The cost of an arc is approximately proportional to its capacity: $c_{ij} = \lfloor \alpha_{ij} u_{ij} \rfloor$, where α_{ij} is a random factor selected uniformly from the range $[0.9, 1.1)$.
- *VISION–INV.* The cost of an arc is approximately inversely proportional to its capacity: $c_{ij} = \lfloor \alpha_{ij} K / u_{ij} \rfloor$, where $K = 1000$ and α_{ij} is a random factor selected uniformly from the range $[0.9, 1.1)$.

We also experimented with other parameter settings both in case of the generators and in case of the ROAD and VISION instances, but the presented collection turned out to be a representative benchmark suite of reasonable size.

For all problem families, we generated five instances of each network size using different random seeds. To achieve reliable comparison, we report the average running time over such five instances in all cases.

4.2 *Comparison of algorithms in the LEMON library*

In what follows, we present benchmark results obtained for the MCF algorithm implementations of the LEMON library, which are discussed in Section 3.1.

Table 2 and Figure 1 show the results on NETGEN problem families, while Table 3 and Figure 2 show the results on GOTO instances. The first part of each table presents running time in seconds, while the second part reports normalized time results. The best running time is highlighted for each problem size, and a '−' sign denotes the cases when the explicit time limit of 1 h was exceeded. The charts display running time in seconds as a function of the number of nodes in the network. Logarithmic scale is used for both axes. Each time result is the average running time on five different problem instances, which were generated with exactly the same settings but with different random seeds.

In accordance with previous studies, we found that GOTO instances are substantially harder than NETGEN problems, and the relative performance of the various solution methods also turned out to be rather different on them. Furthermore, as one would expect, NETGEN–LO–8 and NETGEN–LO–SR instances are considerably easier to solve than NETGEN–8 and NETGEN–SR instances of the same size, respectively. However, the results on GRIDGEN networks are quite similar to those obtained on the corresponding NETGEN families, thus we omit GRIDGEN results here.

According to these experiments, the simple cycle-canceling methods, SCC and MMCC, are orders of magnitude slower than all other algorithms. CAT, which is an advanced cycle-cancelling algorithm, is much faster and usually performs similarly to the dual algorithms SSP and CAS. The relative performance of these three methods greatly depends on the characteristics of the problem instance. The COS and NS algorithms are generally the most efficient. COS shows better asymptotic behaviour than NS, thus it is typically faster on the largest networks, while NS clearly outperforms COS on the smaller ones.

Table 4 compares the efficiency of LEMON implementations on GRIDGRAPH families. The shape of the underlying grid is an essential parameter for these networks as it determines the length of augmenting paths or cycles an algorithm should find. This phenomenon has consistently been observed before, for example, see [15]. On GRID–WIDE networks, NS is by far the fastest and COS is the second, while on GRID–LONG instances, the augmenting path algorithms, SSP and CAS, greatly outperform all other solution methods. The GRID–SQUARE results are just similar to the experiments on NETGEN and GOTO families.

The benchmark results on ROAD networks are presented in Table 5. As one would expect, the SSP algorithm is the fastest on these special instances. On the ROAD–PATHS family, the CAS algorithm works exactly the same as SSP since arc capacities are uniformly set to one, but it is slower than SSP on ROAD–FLOW instances. The COS and NS algorithms perform

Table 2. Comparison of LEMON implementations on NETGEN networks.

Problem family	n	m/n	SCC	MMCC	CAT	SSP	CAS	COS	NS
Running time (s)									
NETGEN–8	2^{10}	8	4.82	11.78	0.19	0.12	0.20	0.02	**0.01**
	2^{13}	8	571.14	2032.79	6.99	6.38	20.82	0.42	**0.15**
	2^{16}	8	–	–	349.54	286.58	2316.07	**4.30**	6.71
	2^{19}	8	–	–	–	–	–	**45.57**	345.16
	2^{22}	8	–	–	–	–	–	**569.68**	–
NETGEN–SR	2^{10}	32	33.18	82.47	0.71	0.31	1.51	0.06	**0.02**
	2^{12}	64	1442.21	–	15.85	6.75	76.22	0.80	**0.21**
	2^{14}	128	–	–	320.55	132.74	–	8.00	**3.47**
	2^{16}	256	–	–	–	2895.84	–	103.72	**63.22**
NETGEN–LO–8	2^{10}	8	0.84	2.14	0.12	0.02	**0.01**	**0.01**	**0.01**
	2^{13}	8	25.71	191.79	3.82	0.41	0.21	0.19	**0.06**
	2^{16}	8	657.61	–	147.75	16.39	6.34	2.51	**2.19**
	2^{19}	8	–	–	–	486.72	182.17	**29.54**	98.51
	2^{22}	8	–	–	–	–	–	**470.68**	2702.33
NETGEN–LO–SR	2^{10}	32	6.37	39.60	0.71	0.03	0.02	0.04	**0.01**
	2^{12}	64	167.79	1520.41	9.48	0.52	0.32	0.51	**0.10**
	2^{14}	128	–	–	190.59	8.65	4.40	5.18	**1.38**
	2^{16}	256	–	–	–	208.38	70.63	75.87	**21.92**
NETGEN–DEG	2^{12}	32	653.08	2633.90	7.30	4.25	28.47	0.42	**0.13**
	2^{12}	128	3441.75	–	32.35	12.22	240.58	1.94	**0.39**
	2^{12}	512	–	–	139.31	30.81	2694.02	7.72	**1.33**
	2^{12}	2048	–	–	576.70	103.16	–	40.26	**4.99**
Normalized time									
NETGEN–8	2^{10}	8	482.00	1178.00	19.00	12.00	20.00	2.00	**1.00**
	2^{13}	8	3807.60	13551.93	46.60	42.53	138.80	2.80	**1.00**
	2^{16}	8	–	–	81.29	66.65	538.62	**1.00**	1.56
	2^{19}	8	–	–	–	–	–	**1.00**	7.57
	2^{22}	8	–	–	–	–	–	**1.00**	–
NETGEN–SR	2^{10}	32	1659.00	4123.50	35.50	15.50	75.50	3.00	**1.00**
	2^{12}	64	6867.67	–	75.48	32.14	362.95	3.81	**1.00**
	2^{14}	128	–	–	92.38	38.25	–	2.31	**1.00**
	2^{16}	256	–	–	–	45.81	–	1.64	**1.00**
NETGEN–LO–8	2^{10}	8	84.00	214.00	12.00	2.00	**1.00**	**1.00**	**1.00**
	2^{13}	8	428.50	3196.50	63.67	6.83	3.50	3.17	**1.00**
	2^{16}	8	300.28	–	67.47	7.48	2.89	1.15	**1.00**
	2^{19}	8	–	–	–	16.48	6.17	**1.00**	3.33
	2^{22}	8	–	–	–	–	–	**1.00**	5.74
NETGEN–LO–SR	2^{10}	32	637.00	3960.00	71.00	3.00	2.00	4.00	**1.00**
	2^{12}	64	1677.90	15204.10	94.80	5.20	3.20	5.10	**1.00**
	2^{14}	128	–	–	138.11	6.27	3.19	3.75	**1.00**
	2^{16}	256	–	–	–	9.51	3.22	3.46	**1.00**
NETGEN–DEG	2^{12}	32	5023.69	20260.77	56.15	32.69	219.00	3.23	**1.00**
	2^{12}	128	8825.00	–	82.95	31.33	616.87	4.97	**1.00**
	2^{12}	512	–	–	104.74	23.17	2025.58	5.80	**1.00**
	2^{12}	2048	–	–	115.57	20.67	–	8.07	**1.00**

significantly worse than SSP and CAS, especially in case of the ROAD–PATHS family, while the cycle-cancelling algorithms are drastically slower than the other methods.

Finally, Table 6 and Figure 3 summarize the benchmark results on VISION networks. These MCF instances turned out to be rather hard. The slowest algorithms, SCC and MMCC, were unable to solve any of them within 1 h, hence these methods are omitted in the table and the

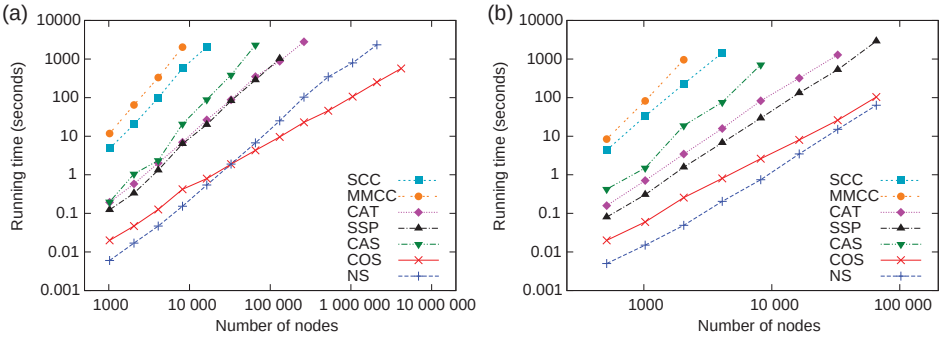

Figure 1. Comparison of LEMON implementations on NETGEN networks. (a) NETGEN–8 family and (b) NETGEN–SR family.

Table 3. Comparison of LEMON implementations on GOTO networks.

Problem family	n	m/n	SCC	MMCC	CAT	SSP	CAS	COS	NS
Running time (s)									
GOTO–8	2^{10}	8	92.57	32.03	0.63	2.37	0.14	0.06	**0.01**
	2^{13}	8	–	–	48.65	237.60	4.53	1.45	**0.81**
	2^{16}	8	–	–	–	–	157.15	**37.87**	208.61
	2^{19}	8	–	–	–	–	–	**1413.75**	–
GOTO–SR	2^{10}	32	2389.67	588.14	2.35	35.92	2.77	0.30	**0.08**
	2^{12}	64	–	–	89.97	–	142.50	5.17	**2.88**
	2^{14}	128	–	–	3024.20	–	–	**72.98**	163.26
	2^{16}	256	–	–	–	–	–	**1260.33**	–
Normalized time									
GOTO–8	2^{10}	8	9257.00	3203.00	63.00	237.00	14.00	6.00	**1.00**
	2^{13}	8	–	–	60.06	293.33	5.59	1.79	**1.00**
	2^{16}	8	–	–	–	–	4.15	**1.00**	5.51
	2^{19}	8	–	–	–	–	–	**1.00**	–
GOTO–SR	2^{10}	32	29870.88	7351.75	29.38	449.00	34.63	3.75	**1.00**
	2^{12}	64	–	–	31.24	–	49.48	1.80	**1.00**
	2^{14}	128	–	–	41.44	–	–	**1.00**	2.24
	2^{16}	256	–	–	–	–	–	**1.00**	–

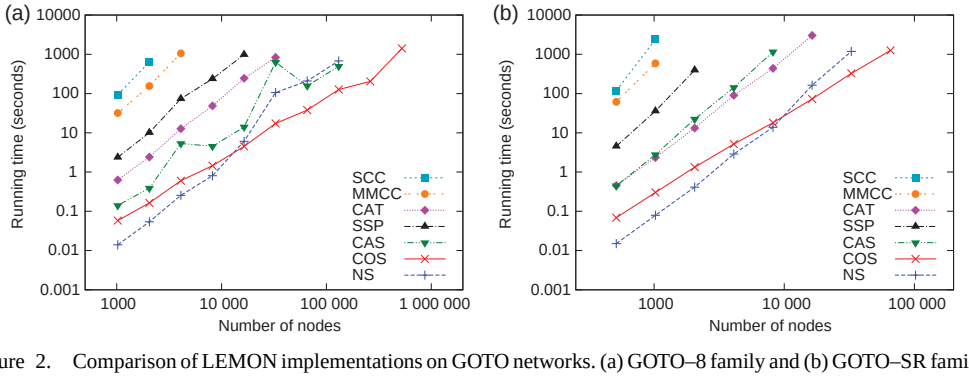

Figure 2. Comparison of LEMON implementations on GOTO networks. (a) GOTO–8 family and (b) GOTO–SR family.

Table 4. Comparison of LEMON implementations on GRIDGRAPH networks.

Problem family	n	m/n	SCC	MMCC	CAT	SSP	CAS	COS	NS
Running time (s)									
GRID–WIDE	2^{10}	2.04	0.84	1.35	0.11	0.05	**0.01**	0.02	**0.01**
	2^{13}	2.06	72.62	150.15	2.44	4.19	0.97	0.32	**0.03**
	2^{16}	2.06	–	–	54.27	392.94	100.52	9.96	**0.66**
	2^{19}	2.06	–	–	1027.71	–	–	463.73	**12.83**
GRID–LONG	2^{10}	1.95	0.59	0.93	0.13	0.02	**0.01**	**0.01**	**0.01**
	2^{13}	1.94	66.32	41.77	7.36	0.11	**0.05**	0.21	0.08
	2^{16}	1.94	2857.30	1520.42	818.85	0.46	**0.36**	2.38	5.53
	2^{19}	1.94	–	–	–	**2.93**	3.05	22.39	504.82
GRID–SQUARE	2^{10}	2	0.72	1.38	0.12	0.04	**0.01**	0.02	**0.01**
	2^{13}	2	118.44	214.00	5.44	2.49	0.30	0.32	**0.07**
	2^{16}	2	–	–	297.16	204.53	11.11	7.05	**4.26**
	2^{19}	2	–	–	–	–	699.42	**152.58**	477.84
Normalized time									
GRID–WIDE	2^{10}	2.04	84.00	135.00	11.00	5.00	**1.00**	2.00	**1.00**
	2^{13}	2.06	2420.67	5005.00	81.33	139.67	32.33	10.67	**1.00**
	2^{16}	2.06	–	–	82.23	595.36	152.30	15.09	**1.00**
	2^{19}	2.06	–	–	80.10	–	–	36.14	**1.00**
GRID–LONG	2^{10}	1.95	59.00	93.00	13.00	2.00	**1.00**	**1.00**	**1.00**
	2^{13}	1.94	1326.40	835.40	147.20	2.20	**1.00**	4.20	1.60
	2^{16}	1.94	7936.94	4223.39	2274.58	1.28	**1.00**	6.61	15.36
	2^{19}	1.94	–	–	–	**1.00**	1.04	7.64	172.29
GRID–SQUARE	2^{10}	2	72.00	138.00	12.00	4.00	**1.00**	2.00	**1.00**
	2^{13}	2	1692.00	3057.14	77.71	35.57	4.29	4.57	**1.00**
	2^{16}	2	–	–	69.76	48.01	2.61	1.65	**1.00**
	2^{19}	2	–	–	–	–	4.58	**1.00**	3.13

Table 5. Comparison of LEMON implementations on ROAD networks.

Problem family	n	m/n	SCC	MMCC	CAT	SSP	CAS	COS	NS
Running time (s)									
ROAD–PATHS	9559	3.11	2.62	396.79	2.77	**0.01**	**0.01**	0.16	0.06
	116,920	2.27	203.66	–	247.86	**0.30**	**0.30**	4.69	3.15
	519,157	2.44	–	–	3318.63	**3.46**	**3.46**	36.13	42.58
	2,073,870	2.49	–	–	–	**19.06**	**19.06**	248.32	506.40
ROAD–FLOW	9559	3.11	7.71	649.40	3.35	0.04	**0.03**	0.21	0.06
	116,920	2.27	742.47	–	399.39	**1.10**	1.50	8.43	4.65
	519,157	2.44	–	–	–	**11.06**	19.05	54.38	47.70
	2,073,870	2.49	–	–	–	**88.74**	336.75	471.64	1106.19
Normalized time									
ROAD–PATHS	9559	3.11	262.00	39679.00	277.00	**1.00**	**1.00**	16.00	6.00
	116,920	2.27	678.87	–	826.20	**1.00**	**1.00**	15.63	10.50
	519,157	2.44	–	–	959.14	**1.00**	**1.00**	10.44	12.31
	2,073,870	2.49	–	–	–	**1.00**	**1.00**	13.03	26.57
ROAD–FLOW	9559	3.11	257.00	21646.67	111.67	1.33	**1.00**	7.00	2.00
	116,920	2.27	674.97	–	363.08	**1.00**	1.36	7.66	4.23
	519,157	2.44	–	–	–	**1.00**	1.72	4.92	4.31
	2,073,870	2.49	–	–	–	**1.00**	3.79	5.31	12.47

chart. COS clearly turned out to be the most robust algorithm on VISION families, the asymptotic trend of its running time is much better than that of NS. However, CAT, SSP, and CAS are even substantially slower.

Table 6. Comparison of LEMON implementations on VISION networks.

Problem family	n	m/n	CAT	SSP	CAS	COS	NS
Running time (s)							
VISION–RND	245,762	5.82	633.99	267.18	129.16	24.43	**20.92**
	491,522	5.85	1684.60	979.09	603.44	**63.27**	115.48
	983,042	5.88	–	–	2027.55	**187.69**	554.55
	1,949,698	5.91	–	–	–	**494.24**	3510.13
	3,899,394	5.92	–	–	–	**1341.04**	–
VISION–PROP	245,762	5.82	611.30	500.43	225.14	49.75	**21.17**
	491,522	5.85	1644.80	2049.59	800.09	**111.03**	112.52
	983,042	5.88	–	–	3086.13	**406.98**	458.90
	1,949,698	5.91	–	–	–	**842.25**	2830.07
	3,899,394	5.92	–	–	–	**2426.39**	–
VISION–INV	245,762	5.82	406.30	116.26	41.53	41.29	**14.26**
	491,522	5.85	1137.09	457.24	181.83	97.07	**81.93**
	983,042	5.88	–	2922.38	745.12	**298.76**	304.45
	1,949,698	5.91	–	–	2683.08	**820.11**	2168.45
	3,899,394	5.92	–	–	–	**2201.84**	–
Normalized time							
VISION–RND	245,762	5.82	30.31	12.77	6.17	1.17	**1.00**
	491,522	5.85	26.63	15.47	9.54	**1.00**	1.83
	983,042	5.88	–	–	10.80	**1.00**	2.95
	1,949,698	5.91	–	–	–	**1.00**	7.10
	3,899,394	5.92	–	–	–	**1.00**	–
VISION–PROP	245,762	5.82	28.88	23.64	10.63	2.35	**1.00**
	491,522	5.85	14.81	18.46	7.21	**1.00**	1.01
	983,042	5.88	–	–	7.58	**1.00**	1.13
	1,949,698	5.91	–	–	–	**1.00**	3.36
	3,899,394	5.92	–	–	–	**1.00**	–
VISION–INV	245,762	5.82	28.49	8.15	2.91	2.90	**1.00**
	491,522	5.85	13.88	5.58	2.22	1.18	**1.00**
	983,042	5.88	–	9.78	2.49	**1.00**	1.02
	1,949,698	5.91	–	–	3.27	**1.00**	2.64
	3,899,394	5.92	–	–	–	**1.00**	–

According to these results and many additional experiments, we can conclude that COS and NS are generally the most efficient algorithms provided by the LEMON library. On particular problem instances, however, the dual ascent augmenting path algorithms, SSP and CAS, could be significantly faster (e.g. GRID–LONG and ROAD networks).

4.3 *Comparison of various solvers*

In this section, we compare the best MCF solution methods of LEMON, mainly COS and NS, to other publicly available MCF solvers described in Section 3.2. CS2, MCFZIB, CPLEX, and RelaxIV are widely known and have served as benchmarks for a long time (see, e.g. [15,39,49,77]), while the LEDA library and PDNET are rarely included in experimental studies, and MCFSimplex is a recent implementation.

All of these codes were compiled with the same compiler and optimization settings as we used for the LEMON implementations (GCC 4.5.3 with -O3 optimization), and they were also executed applying a time limit of 1 h. We used all solvers with their default options as we were interested in evaluating their robustness without exploiting the flexibility they provide in parameter settings. However, flexibility may also be important in many applications.

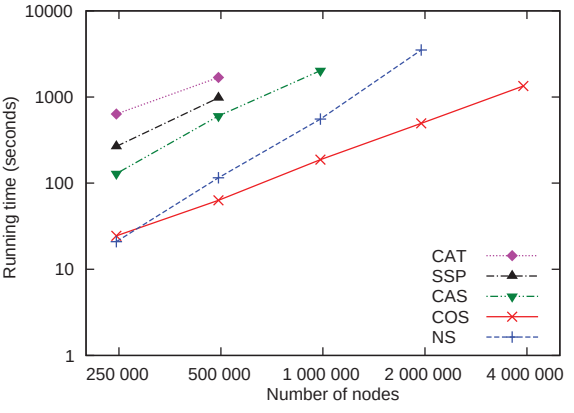

Figure 3. Comparison of LEMON implementations on VISION–RND networks.

As in the previous section, we always report average running time over five different problem instances of the same size. To avoid overwhelmed charts, the results of LEDA and the primal version of MCFSimplex are not depicted in them. LEDA performed worse than or similarly to the other two cost-scaling methods, COS in LEMON and CS2; while MCFSimplex has much in common with MCFZIB, and thereby their performance is also similar. However, the tables report the results for these two solvers, as well. For technical reasons, the primal and dual versions of MCFSimplex are abbreviated as MSim and MSim-D, respectively.

Table 7 and Figure 4 compare the implementations on NETGEN instances, while Table 8 compares them on GRIDGEN families. The NS code of the LEMON library turned out to be the most efficient on most of these networks, but it is consistently outperformed by the cost-scaling algorithms and RelaxIV on the largest sparse graphs. MCFZIB and MSim are typically slower than NS by a factor between 2 and 5, while CPLEX turned out to be even less efficient. MSim-D is competitive with the primal network simplex codes on these families, especially in case of large networks.

Our COS code performs similarly to or slightly slower than CS2 on these instances. However, the third cost-scaling implementation, LEDA, is at least 1.5–2 times slower than them. Furthermore, it failed to solve the largest instances due to number overflow errors, which is denoted as '*error*' in the tables. Since the LEDA library has closed source, we could not replace the internally used number types to eliminate this issue. RelaxIV turned out to be highly efficient on these families, it is competitive with or even faster than the other codes. Finally, PDNET runs much slower than all other algorithms and failed to solve the NETGEN–LO–8 and NETGEN–LO–SR instances. It terminated with an error message: 'STOP disconnected network', even on the smallest instances, which may be due to inappropriate setting of tolerances for floating-point computations. These cases are denoted as '*error*' in Table 7.

In the 'LO' versions of NETGEN instances, the total supply to be delivered is one hundred times lower than that of other NETGEN and GRIDGEN instances of the same size. Thereby, the arc capacities incorporate only 'loose' bounds for the feasible solutions. The impact of the tightness of MCF instances was observed and evaluated before, for example, in [18]. Instances with loose or infinite capacity bounds are easier to solve, especially for network simplex methods, while in case of tight capacities, the cost-scaling algorithms are more robust according to our experiments.

The performance results for the GOTO families are presented in Table 9 and Figure 5. In these tests, COS and CS2 also perform similarly, and they are the most efficient on the largest instances of both families. LEDA is also similarly efficient on GOTO–8 networks, but it is 2–3 times slower than CS2 and COS on the GOTO–SR family. NS turned out to be an order of magnitude faster than

Table 7. Comparison of MCF solvers on NETGEN networks.

			LEMON		Other solvers							
Problem family	n	m/n	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	MSim-D	RelaxIV	PDNET
Running time (s)												
NETGEN–8	2^{10}	8	0.02	**0.01**	0.02	0.03	0.02	0.02	**0.01**	0.04	**0.01**	0.74
	2^{13}	8	0.42	**0.15**	0.31	0.47	0.57	1.07	0.43	1.83	0.26	13.51
	2^{16}	8	4.30	6.71	4.28	8.34	18.29	116.11	23.04	19.45	**3.54**	336.62
	2^{19}	8	45.57	345.16	48.08	*error*	740.96	–	1040.51	304.64	**41.85**	–
	2^{22}	8	569.68	–	547.95	*error*	–	–	–	–	**431.21**	–
NETGEN–SR	2^{10}	32	0.06	**0.02**	0.05	0.08	0.05	0.09	0.05	0.27	0.03	3.82
	2^{12}	64	0.80	**0.21**	0.58	1.42	0.68	1.46	0.70	3.71	1.09	62.73
	2^{14}	128	8.00	**3.47**	8.01	19.84	21.11	30.45	15.50	32.25	4.54	1250.90
	2^{16}	256	103.72	63.22	99.44	266.39	637.93	967.10	815.55	261.94	**41.40**	–
NETGEN–LO–8	2^{10}	8	**0.01**	**0.01**	**0.01**	0.02	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**	*error*
	2^{13}	8	0.19	**0.06**	0.18	0.34	0.17	0.41	0.15	0.40	0.62	*error*
	2^{16}	8	2.51	**2.19**	2.23	6.30	3.98	16.56	4.28	4.91	2.62	*error*
	2^{19}	8	29.54	98.51	**24.00**	*error*	78.43	650.33	92.67	62.46	25.94	*error*
	2^{22}	8	470.68	2702.33	291.35	*error*	2315.19	–	2860.57	541.93	**223.62**	*error*
NETGEN–LO–SR	2^{10}	32	0.04	**0.01**	0.04	0.08	0.03	0.05	0.03	0.07	0.04	*error*
	2^{12}	64	0.51	**0.10**	0.46	1.25	0.34	0.66	0.39	1.29	0.57	*error*
	2^{14}	128	5.18	**1.38**	6.02	15.64	5.40	9.78	5.94	10.78	4.05	*error*
	2^{16}	256	75.87	**21.92**	77.53	220.36	84.55	299.05	105.89	116.83	26.06	*error*
NETGEN–DEG	2^{12}	32	0.42	**0.13**	0.32	0.69	0.40	1.11	0.38	2.06	0.38	24.03
	2^{12}	128	1.94	**0.39**	1.13	2.94	1.29	2.31	1.37	7.02	1.34	176.71
	2^{12}	512	7.72	**1.33**	5.00	12.54	4.69	8.41	5.19	29.90	3.16	1288.59
	2^{12}	2048	40.26	**4.99**	29.43	66.54	16.81	45.49	15.22	107.51	10.93	–
Normalized time												
NETGEN–8	2^{10}	8	2.00	**1.00**	2.00	3.00	2.00	2.00	**1.00**	4.00	**1.00**	74.00
	2^{13}	8	2.80	**1.00**	2.07	3.13	3.80	7.13	2.87	12.20	1.73	90.07
	2^{16}	8	1.21	1.90	1.21	2.36	5.17	32.80	6.51	5.49	**1.00**	95.09
	2^{19}	8	1.09	8.25	1.15	*error*	17.71	–	24.86	7.28	**1.00**	–
	2^{22}	8	1.32	–	1.27	*error*	–	–	–	–	**1.00**	–
NETGEN–SR	2^{10}	32	3.00	**1.00**	2.50	4.00	2.50	4.50	2.50	13.50	1.50	191.00
	2^{12}	64	3.81	**1.00**	2.76	6.76	3.24	6.95	3.33	17.67	5.19	298.71
	2^{14}	128	2.31	**1.00**	2.31	5.72	6.08	8.78	4.47	9.29	1.31	360.49
	2^{16}	256	2.51	1.53	2.40	6.43	15.41	23.36	19.70	6.33	**1.00**	–
NETGEN–LO–8	2^{10}	8	**1.00**	**1.00**	**1.00**	2.00	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	*error*
	2^{13}	8	3.17	**1.00**	3.00	5.67	2.83	6.83	2.50	6.67	10.33	*error*
	2^{16}	8	1.15	**1.00**	1.02	2.88	1.82	7.56	1.95	2.24	1.20	*error*
	2^{19}	8	1.23	4.10	**1.00**	*error*	3.27	27.10	3.86	2.60	1.08	*error*
	2^{22}	8	2.10	12.08	1.30	*error*	10.35	–	12.79	2.42	**1.00**	*error*
NETGEN–LO–SR	2^{10}	32	4.00	**1.00**	4.00	8.00	3.00	5.00	3.00	7.00	4.00	*error*
	2^{12}	64	5.10	**1.00**	4.60	12.50	3.40	6.60	3.90	12.90	5.70	*error*
	2^{14}	128	3.75	**1.00**	4.36	11.33	3.91	7.09	4.30	7.81	2.93	*error*
	2^{16}	256	3.46	**1.00**	3.54	10.05	3.86	13.64	4.83	5.33	1.19	*error*
NETGEN–DEG	2^{12}	32	3.23	**1.00**	2.46	5.31	3.08	8.54	2.92	15.85	2.92	184.85
	2^{12}	128	4.97	**1.00**	2.90	7.54	3.31	5.92	3.51	18.00	3.44	453.10
	2^{12}	512	5.80	**1.00**	3.76	9.43	3.53	6.32	3.90	22.48	2.38	968.86
	2^{12}	2048	8.07	**1.00**	5.90	13.33	3.37	9.12	3.05	21.55	2.19	–

the other primal network simplex codes, MCFZIB, CPLEX, and MSim. Similarly to the earlier results, NS is the most efficient on relatively small graphs, but it is substantially slower than the cost-scaling codes on the large networks, especially on the sparse ones. Despite its relatively good performance on the easier NETGEN and GRIDGEN networks, MSim-D is far not competitive on the GOTO instances. RelaxIV also turned out to be very slow on these hard instances, which is in

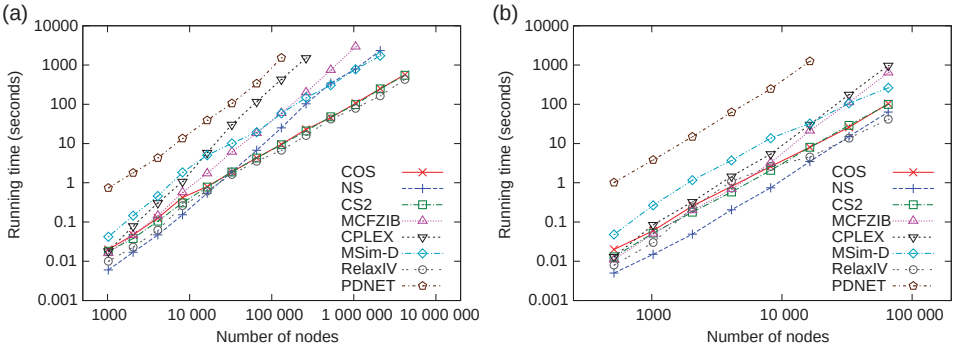

Figure 4. Comparison of MCF solvers on NETGEN networks. (a) NETGEN–8 family and (b) NETGEN–SR family.

Table 8. Comparison of MCF solvers on GRIDGEN networks.

			LEMON		Other solvers							
Problem family	n	m/n	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	MSim-D	RelaxIV	PDNET
Running time (s)												
GRIDGEN–8	2^{10}	8	0.02	**0.01**	0.02	0.03	0.02	0.02	**0.01**	0.05	**0.01**	0.81
	2^{13}	8	0.40	**0.14**	0.34	0.52	0.53	1.39	0.43	1.33	0.34	13.07
	2^{16}	8	4.24	4.27	4.31	8.29	11.59	109.57	12.68	37.11	**3.77**	361.02
	2^{19}	8	**48.51**	174.48	56.15	*error*	553.96	–	569.76	437.70	63.75	–
	2^{22}	8	532.30	–	**465.14**	*error*	–	–	–	–	470.27	–
GRIDGEN–SR	2^{10}	32	0.08	**0.02**	0.06	0.11	0.05	0.13	0.06	0.55	0.07	3.37
	2^{12}	64	0.95	**0.19**	0.70	1.69	0.59	1.94	0.59	10.17	1.15	40.27
	2^{14}	128	9.72	**2.94**	9.02	21.33	15.13	64.37	16.49	70.48	11.71	846.06
	2^{16}	256	153.09	**73.43**	135.30	368.18	376.05	2537.32	783.91	591.63	113.79	–
GRIDGEN–DEG	2^{12}	32	0.47	**0.11**	0.37	0.82	0.34	1.25	0.32	2.99	0.62	19.64
	2^{12}	128	1.93	**0.35**	1.34	3.61	1.05	3.06	1.09	15.31	3.09	85.60
	2^{12}	512	8.50	**1.22**	5.49	14.22	3.64	9.35	3.80	30.80	5.86	369.47
	2^{12}	2048	42.70	**4.82**	30.50	80.76	14.06	41.32	14.64	143.66	15.25	1800.28
Normalized time												
GRIDGEN–8	2^{10}	8	2.00	**1.00**	2.00	3.00	2.00	2.00	**1.00**	5.00	**1.00**	81.00
	2^{13}	8	2.86	**1.00**	2.43	3.71	3.79	9.93	3.07	9.50	2.43	93.36
	2^{16}	8	1.12	1.13	1.14	2.20	3.07	29.06	3.36	9.84	**1.00**	95.76
	2^{19}	8	**1.00**	3.60	1.16	*error*	11.42	–	11.75	9.02	1.31	–
	2^{22}	8	1.14	–	**1.00**	*error*	–	–	–	–	1.01	–
GRIDGEN–SR	2^{10}	32	4.00	**1.00**	3.00	5.50	2.50	6.50	3.00	27.50	3.50	168.50
	2^{12}	64	5.00	**1.00**	3.68	8.89	3.11	10.21	3.11	53.53	6.05	211.95
	2^{14}	128	3.31	**1.00**	3.07	7.26	5.15	21.89	5.61	23.97	3.98	287.78
	2^{16}	256	2.08	**1.00**	1.84	5.01	5.12	34.55	10.68	8.06	1.55	–
GRIDGEN–DEG	2^{12}	32	4.27	**1.00**	3.36	7.45	3.09	11.36	2.91	27.18	5.64	178.55
	2^{12}	128	5.51	**1.00**	3.83	10.31	3.00	8.74	3.11	43.74	8.83	244.57
	2^{12}	512	6.97	**1.00**	4.50	11.66	2.98	7.66	3.11	25.25	4.80	302.84
	2^{12}	2048	8.86	**1.00**	6.33	16.76	2.92	8.57	3.04	29.80	3.16	373.50

sharp contrast to its efficiency on the NETGEN and GRIDGEN families. On the other hand, the relative performance of PDNET compared to the other solvers is much better in case of GOTO networks than on easier instances. It outperforms the network simplex codes on the large GOTO–8 instances, with the only exception of our implementation.

Table 10 and Figure 6 show the results for the GRIDGRAPH families, while Table 11 and Figure 7 present the results for the ROAD families. PDNET failed on most of these special MCF instances with an error message: 'STOP disconnected network', just like in case of

Table 9. Comparison of MCF solvers on GOTO networks.

			LEMON		Other solvers							
Problem family	n	m/n	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	MSim-D	RelaxIV	PDNET
Running time (s)												
GOTO–8	2^{10}	8	0.06	**0.01**	0.06	0.08	0.22	0.21	0.17	0.67	0.85	0.93
	2^{13}	8	1.45	**0.81**	1.94	1.68	33.60	30.41	29.20	162.14	104.71	17.22
	2^{16}	8	**37.87**	208.61	49.67	53.46	–	3379.22	–	–	–	811.74
	2^{19}	8	1413.75	–	**1398.30**	1659.08	–	–	–	–	–	–
GOTO–SR	2^{10}	32	0.30	**0.08**	0.28	0.37	0.72	0.82	0.58	34.34	8.84	6.45
	2^{12}	64	5.17	**2.88**	4.77	9.62	47.02	36.13	42.12	–	479.58	143.41
	2^{14}	128	**72.98**	163.26	82.72	190.09	2395.41	1407.59	1601.92	–	–	2534.18
	2^{16}	256	1260.33	–	**1203.92**	3095.03	–	–	–	–	–	–
Normalized time												
GOTO–8	2^{10}	8	6.00	**1.00**	6.00	8.00	22.00	21.00	17.00	67.00	85.00	93.00
	2^{13}	8	1.79	**1.00**	2.40	2.07	41.48	37.54	36.05	200.17	129.27	21.26
	2^{16}	8	**1.00**	5.51	1.31	1.41	–	89.23	–	–	–	21.43
	2^{19}	8	1.01	–	**1.00**	1.19	–	–	–	–	–	–
GOTO–SR	2^{10}	32	3.75	**1.00**	3.50	4.63	9.00	10.25	7.25	429.25	110.50	80.63
	2^{12}	64	1.80	**1.00**	1.66	3.34	16.33	12.55	14.63	–	166.52	49.80
	2^{14}	128	**1.00**	2.24	1.13	2.60	32.82	19.29	21.95	–	–	34.72
	2^{16}	256	1.05	–	**1.00**	2.57	–	–	–	–	–	–

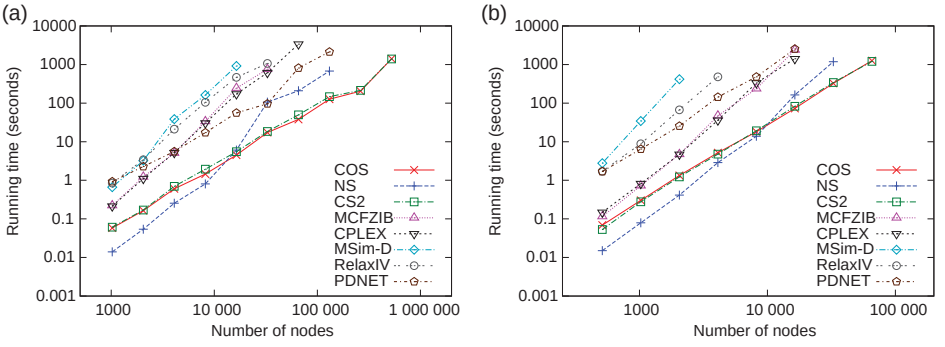

Figure 5. Comparison of MCF solvers on GOTO networks. (a) GOTO–8 family and (b) GOTO–SR family.

NETGEN–LO–8 and NETGEN–LO–SR networks. Therefore, its running time is not reported. However, the CAS algorithm of LEMON is included instead, because the augmenting path methods turned out to be interesting in case of these particular networks.

On the GRID–WIDE instances (see Figure 6(a)), the primal network simplex algorithms are the fastest, probably because they are capable of cancelling a lot of short negative cycles efficiently due to the limited depth of the underlying spanning tree data structure. The cost-scaling methods perform clearly worse on these wide grids (even on the largest ones), while CAS, RelaxIV, and MSim-D are even much slower. On the other hand, in case of GRID–LONG networks (see Figure 6(b)), the augmenting paths or cycles are much longer, hence CAS, the capacity-scaling augmenting path algorithm, turned out to be the fastest. The cost-scaling codes are significantly slower, but they clearly outperform the primal network simplex codes and RelaxIV. MSim-D is the slowest solver, just like on the wide grids. The experiments on the GRID–SQUARE family are more similar to the previous results on NETGEN, GRIDGEN, and GOTO networks: NS is the fastest on the smaller graphs, while the cost-scaling algorithms are superior to it on the largest ones.

Table 10. Comparison of MCF solvers on GRIDGRAPH networks.

			LEMON			Other solvers						
Problem family	n	m/n	CAS	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	MSim-D	RelaxIV
Running time (s)												
GRID–WIDE	2^{10}	2.04	**0.01**	0.02	**0.01**	**0.01**	0.02	**0.01**	**0.01**	**0.01**	0.04	0.03
	2^{13}	2.06	0.97	0.32	**0.03**	0.29	0.54	0.10	0.11	0.07	11.23	2.84
	2^{16}	2.06	100.52	9.96	**0.66**	6.30	49.45	2.07	2.10	1.77	1184.36	743.83
	2^{19}	2.06	–	463.73	**12.83**	108.82	*error*	19.19	24.21	16.31	–	–
GRID–LONG	2^{10}	1.95	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**	0.02	0.02
	2^{13}	1.94	**0.05**	0.21	0.08	0.13	0.16	0.43	0.73	0.44	2.85	0.81
	2^{16}	1.94	**0.36**	2.38	5.53	2.82	2.20	47.80	51.06	37.97	264.50	73.75
	2^{19}	1.94	**3.05**	22.39	504.82	24.85	*error*	–	3483.86	3319.92	–	–
GRID–SQUARE	2^{10}	2	**0.01**	0.02	**0.01**	**0.01**	0.02	**0.01**	**0.01**	**0.01**	0.03	0.03
	2^{13}	2	0.30	0.32	**0.07**	0.22	0.26	0.27	0.20	0.19	5.33	2.08
	2^{16}	2	11.11	7.05	**4.26**	4.69	4.90	25.61	13.64	16.85	849.16	205.38
	2^{19}	2	699.42	152.58	477.84	**77.01**	*error*	1829.95	958.73	1591.71	–	–
Normalized time												
GRID–WIDE	2^{10}	2.04	**1.00**	2.00	**1.00**	**1.00**	2.00	**1.00**	**1.00**	**1.00**	4.00	3.00
	2^{13}	2.06	32.33	10.67	**1.00**	9.67	18.00	3.33	3.67	2.33	374.33	94.67
	2^{16}	2.06	152.30	15.09	**1.00**	9.55	74.92	3.14	3.18	2.68	1794.48	1127.02
	2^{19}	2.06	–	36.14	**1.00**	8.48	*error*	1.50	1.89	1.27	–	–
GRID–LONG	2^{10}	1.95	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	2.00	2.00
	2^{13}	1.94	**1.00**	4.20	1.60	2.60	3.20	8.60	14.60	8.80	57.00	16.20
	2^{16}	1.94	**1.00**	6.61	15.36	7.83	6.11	132.78	141.83	105.47	734.72	204.86
	2^{19}	1.94	**1.00**	7.34	165.51	8.15	*error*	–	1142.25	1088.50	–	–
GRID–SQUARE	2^{10}	2	**1.00**	2.00	**1.00**	**1.00**	2.00	**1.00**	**1.00**	**1.00**	3.00	3.00
	2^{13}	2	4.29	4.57	**1.00**	3.14	3.71	3.86	2.86	2.71	76.14	29.71
	2^{16}	2	2.61	1.65	**1.00**	1.10	1.15	6.01	3.20	3.96	199.33	48.21
	2^{19}	2	9.08	1.98	6.20	**1.00**	*error*	23.76	12.45	20.67	–	–

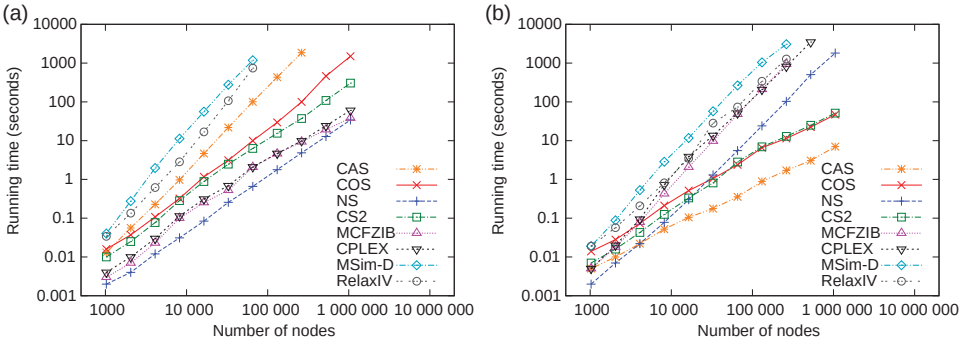

Figure 6. Comparison of MCF solvers on GRIDGRAPH networks. (a) GRID–WIDE family and (b) GRID–LONG family.

On the ROAD–PATHS family (see Figure 7(a)), CAS is much faster than the primal network simplex and cost-scaling solution methods, while MSim-D and RelaxIV are even orders of magnitude slower. The results are similar on the ROAD–FLOW networks (see Figure 7(b)), with the only exception that CAS has smaller advantage due to the larger capacity values. CS2 is superior to CAS on the largest ROAD-FLOW instances, but note that the SSP algorithm of LEMON is even faster (cf. Tables 5 and 11).

Table 12 and Figure 8 present the benchmark results for the VISION families. CS2 is the fastest to solve these hard instances, while COS is about 1.5–2 times slower than it. All other codes

Table 11. Comparison of MCF solvers on ROAD networks.

			LEMON			Other solvers						
Problem family	n	m/n	CAS	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	MSim-D	RelaxIV
Running time (s)												
ROAD–PATHS	9559	3.11	**0.01**	0.16	0.06	0.13	0.15	0.15	0.22	0.12	1.35	0.67
	116,920	2.27	**0.30**	4.69	3.15	2.65	3.30	6.22	6.52	5.70	188.71	219.60
	519,157	2.44	**3.46**	36.13	42.58	19.38	*error*	83.00	80.64	75.95	–	–
	2,073,870	2.49	**19.06**	248.32	506.40	101.02	*error*	948.34	968.58	831.71	–	–
ROAD–FLOW	9559	3.11	**0.03**	0.21	0.06	0.14	0.19	0.15	0.18	0.12	1.48	0.72
	116,920	2.27	**1.50**	8.43	4.65	4.37	5.51	11.21	11.13	10.09	310.20	376.14
	519,157	2.44	**19.05**	54.38	47.70	23.59	*error*	105.36	91.22	97.09	–	–
	2,073,870	2.49	336.75	471.64	1106.19	**157.24**	*error*	1288.25	2200.65	1352.18	–	–
Normalized time												
ROAD–PATHS	9559	3.11	**1.00**	16.00	6.00	13.00	15.00	15.00	22.00	12.00	135.00	67.00
	116,920	2.27	**1.00**	15.63	10.50	8.83	11.00	20.73	21.73	19.00	629.03	732.00
	519,157	2.44	**1.00**	10.44	12.31	5.60	*error*	23.99	23.31	21.95	–	–
	2,073,870	2.49	**1.00**	13.03	26.57	5.30	*error*	49.76	50.82	43.64	–	–
ROAD–FLOW	9559	3.11	**1.00**	7.00	2.00	4.67	6.33	5.00	6.00	4.00	49.33	24.00
	116,920	2.27	**1.00**	5.62	3.10	2.91	3.67	7.47	7.42	6.73	206.80	250.76
	519,157	2.44	**1.00**	2.85	2.50	1.24	*error*	5.53	4.79	5.10	–	–
	2,073,870	2.49	2.14	3.00	7.04	**1.00**	*error*	8.19	14.00	8.60	–	–

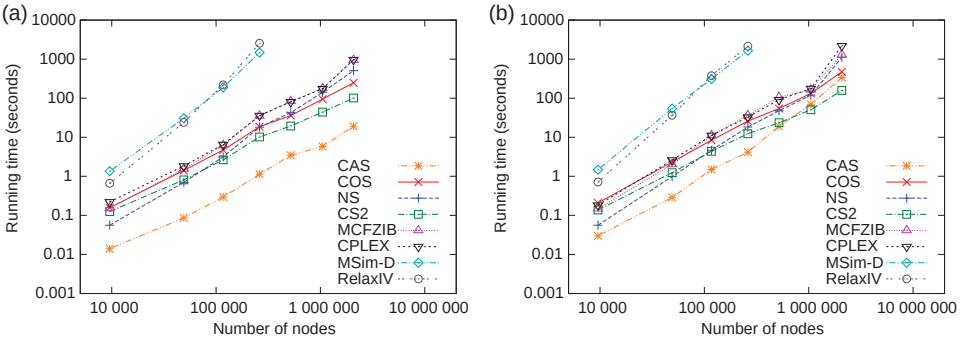

Figure 7. Comparison of MCF solvers on ROAD networks. (a) ROAD–PATHS family and (b) ROAD–FLOW family.

perform much worse, including the third cost-scaling implementation, LEDA. It even failed to solve all VISION–PROP instances due to number overflow errors. NS turned out to be much faster than the other three primal network simplex codes. RelaxIV is very slow on these instances, but MSim-D and PDNET perform even much worse, they could not solve any VISION instance within the 1-h time limit. Therefore, these two solvers are omitted in Table 12 and Figure 8.

The choice of the cost function has only modest impact on the performance of the algorithms on VISION networks. Surprisingly, the cost-scaling codes tend to be slightly slower on both VISION–PROP and VISION–INV instances than in case of random costs, while other methods are faster on these networks, especially on VISION–INV instances.

Finally, in order to provide a compact summary of our experiments, Tables 13 and 14 compare most of the considered MCF solvers on networks of various families having approximately the same number of arcs, namely, 2^{18} and 2^{21}, respectively. Apart from the average running time, these tables also report the standard deviation (measured on five instances that were generated using the same parameters but different random seeds).

Table 12. Comparison of MCF solvers on VISION networks.

			LEMON		Other solvers					
Problem family	n	m/n	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	RelaxIV
Running time (s)										
VISION–RND	245,762	5.82	24.43	20.92	**19.46**	57.48	129.25	97.11	163.30	641.78
	491,522	5.85	63.27	115.48	**44.50**	218.61	320.91	345.68	409.43	2987.68
	983,042	5.88	187.69	554.55	**141.07**	1132.98	–	–	–	–
	1,949,698	5.91	494.24	3510.13	**344.35**	–	–	–	–	–
	3,899,394	5.92	1341.04	–	**865.86**	–	–	–	–	–
VISION–PROP	245,762	5.82	49.75	**21.17**	25.33	*error*	130.91	100.84	140.90	730.04
	491,522	5.85	111.03	112.52	**61.70**	*error*	228.54	333.83	243.70	2112.62
	983,042	5.88	406.98	458.90	**202.76**	*error*	3356.82	2387.97	–	–
	1,949,698	5.91	842.25	2830.07	**459.16**	*error*	–	–	–	–
	3,899,394	5.92	2426.39	–	**1127.42**	*error*	–	–	–	–
VISION–INV	245,762	5.82	41.29	**14.26**	25.11	50.25	27.95	60.27	31.36	367.71
	491,522	5.85	97.07	81.93	58.10	205.03	**57.15**	280.85	63.09	1451.90
	983,042	5.88	298.76	304.45	**177.39**	599.32	2040.11	2632.67	2213.28	–
	1,949,698	5.91	820.11	2168.45	**434.57**	2314.01	–	–	–	–
	3,899,394	5.92	2201.84	–	**1040.63**	*error*	–	–	–	–
Normalized time										
VISION–RND	245,762	5.82	1.26	1.08	**1.00**	2.95	6.64	4.99	8.39	32.98
	491,522	5.85	1.42	2.60	**1.00**	4.91	7.21	7.77	9.20	67.14
	983,042	5.88	1.33	3.93	**1.00**	8.03	–	–	–	–
	1,949,698	5.91	1.44	10.19	**1.00**	–	–	–	–	–
	3,899,394	5.92	1.55	–	**1.00**	–	–	–	–	–
VISION–PROP	245,762	5.82	2.35	**1.00**	1.20	*error*	6.18	4.76	6.66	34.48
	491,522	5.85	1.80	1.82	**1.00**	*error*	3.70	5.41	3.95	34.24
	983,042	5.88	2.01	2.26	**1.00**	*error*	16.56	11.78	–	–
	1,949,698	5.91	1.83	6.16	**1.00**	*error*	–	–	–	–
	3,899,394	5.92	2.15	–	**1.00**	*error*	–	–	–	–
VISION–INV	245,762	5.82	2.90	**1.00**	1.76	3.52	1.96	4.23	2.20	25.79
	491,522	5.85	1.70	1.43	1.02	3.59	**1.00**	4.91	1.10	25.41
	983,042	5.88	1.68	1.72	**1.00**	3.38	11.50	14.84	12.48	–
	1,949,698	5.91	1.89	4.99	**1.00**	5.32	–	–	–	–
	3,899,394	5.92	2.12	–	**1.00**	*error*	–	–	–	–

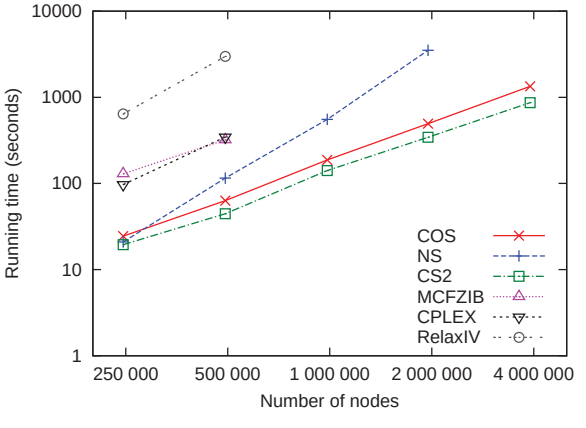

Figure 8. Comparison of MCF solvers on VISION–RND networks.

Table 13. Comparison of MCF solvers on various networks with approximately 2^{18} arcs.

		LEMON		Other solvers							
Problem family	m	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	MSim-D	RelaxIV	PDNET
Average running time (s) and standard deviation											
NETGEN–8	262,144	1.88	1.90	1.88	3.66	6.10	30.44	6.58	10.11	**1.63**	105.22
		0.09	0.09	0.05	0.12	0.40	2.04	1.21	3.98	0.39	6.77
NETGEN–SR	262,144	0.80	**0.21**	0.58	1.42	0.68	1.46	0.70	3.71	1.09	62.73
		0.06	0.01	0.01	0.05	0.02	0.05	0.02	1.12	0.63	1.76
NETGEN–LO–8	262,144	1.05	**0.67**	0.99	2.86	1.61	5.32	1.54	2.10	2.36	*error*
		0.08	0.04	0.04	0.18	0.06	0.35	0.08	0.19	2.14	
NETGEN–LO–SR	262,144	0.51	**0.10**	0.46	1.25	0.34	0.66	0.39	1.29	0.57	*error*
		0.01	0.01	0.05	0.14	0.05	0.05	0.05	0.21	0.16	
GRIDGEN–8	262,096	1.96	**1.67**	1.88	3.30	3.96	26.68	3.49	8.45	1.89	127.28
		0.06	0.12	0.11	0.08	0.49	2.20	0.46	1.29	0.29	28.39
GRIDGEN–SR	262,208	0.95	**0.19**	0.70	1.69	0.59	1.94	0.59	10.17	1.15	40.27
		0.05	0.01	0.02	0.05	0.05	0.02	0.04	3.59	0.33	2.02
GOTO–8	262,144	**17.30**	106.37	18.45	17.94	756.04	613.39	763.73	–	1070.03	96.37
		0.91	4.74	1.49	1.28	88.83	81.26	147.10		76.45	4.35
GOTO–SR	262,144	5.17	**2.88**	4.77	9.62	47.02	36.13	42.12	–	479.58	143.41
		0.22	0.06	0.07	0.97	5.89	2.93	2.08		22.53	6.54
GRID–WIDE	270,320	29.56	**1.79**	15.55	347.94	4.60	4.67	4.00	–	–	*error*
		5.07	0.04	0.37	36.00	0.33	0.24	0.27			
GRID–LONG	253,968	**6.55**	24.02	6.93	7.50	209.99	208.11	169.43	1041.44	338.70	*error*
		1.32	0.84	1.82	1.62	5.95	3.09	6.35	67.53	44.75	
GRID–SQUARE	262,088	23.70	20.55	**12.08**	13.41	117.77	66.58	84.53	–	1266.86	*error*
		1.38	0.59	0.30	0.57	3.45	2.09	2.80		82.87	
ROAD–PATHS	265,402	4.69	3.15	**2.65**	3.30	6.22	6.52	5.70	188.71	219.60	*error*
		0.31	0.39	0.13	0.10	0.75	1.08	0.66	59.62	81.73	
ROAD–FLOW	265,402	8.43	4.65	**4.37**	5.51	11.21	11.13	10.09	310.20	376.14	*error*
		2.14	1.50	1.35	1.05	4.50	4.22	3.93	164.46	215.84	

These tables also demonstrate that the COS and NS codes of the LEMON library and the CS2 solver provide the best overall performance. On relatively small networks, NS is usually superior to the other implementations (see Table 13), but on large sparse networks, COS and CS2 are the fastest and most robust (see Table 14). The other implementations of the cost-scaling and primal network simplex methods (i.e. LEDA, MCFZIB, CPLEX, and MSim) turned out to be less efficient in general, and LEDA also has numerical issues in case of large problem instances. MSim-D, RelaxIV, and PDNET are not robust at all, although they perform reasonably well on some particular networks. The deviation of running time turned out to be moderate in most cases, but MSim-D and RelaxIV seem to be less stable in this aspect, as well.

Apart from the presented results, many other experiments have also been conducted varying several parameters of the problem instances. A notable observation is that the magnitudes of the supply, capacity, and cost values hardly affect the running time of the algorithms, but the relation between supply and capacity values is obviously important (cf. different NETGEN families).

We also compared the number of basic operations performed by the cost-scaling and primal network simplex implementations whose source codes we had access to. We found that our COS code tends to perform significantly less push operations than CS2 due to the partial augment method (see Section 3.1.6), but it performs approximately the same number of relabel operations in most cases. On the other hand, CS2 may find admissible arcs faster due to its speculative arc fixing heuristic (see [49,51]). The COS implementation could be further improved by applying this technique.

As for the network simplex algorithms, MCFZIB and MSim perform exactly the same number of iterations as they apply the same pivot strategy. Our NS code, however, tends to perform

Table 14. Comparison of MCF solvers on various networks with approximately 2^{21} arcs.

		LEMON		Other solvers							
Problem family	*m*	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	MSim-D	RelaxIV	PDNET
Average running time (s) and standard deviation											
NETGEN–8	2,097,152	22.82	102.60	20.87	*error*	198.84	1511.92	286.87	146.76	**16.01**	–
		2.03	3.91	1.09		14.79	46.15	22.09	50.81	5.59	
NETGEN–SR	2,097,152	8.00	**3.47**	8.01	19.84	21.11	30.45	15.50	32.25	4.54	1250.90
		0.33	0.13	0.27	0.40	0.73	0.35	0.48	4.00	0.26	59.38
NETGEN–LO–8	2,097,152	13.29	29.62	**11.04**	*error*	27.86	186.88	34.03	30.91	16.28	*error*
		0.80	1.77	0.76		1.28	5.12	1.55	3.80	9.75	
NETGEN–LO–SR	2,097,152	5.18	**1.38**	6.02	15.64	5.40	9.78	5.94	10.78	4.05	*error*
		0.15	0.07	0.22	0.29	0.21	0.48	0.56	1.24	0.38	
NETGEN–DEG	2,097,152	7.72	**1.33**	5.00	12.54	4.69	8.41	5.19	29.90	3.16	1288.59
		0.33	0.05	0.24	0.46	0.15	0.72	0.23	5.51	0.08	4.90
GRIDGEN–8	2,097,160	**22.92**	53.08	26.03	*error*	150.53	1381.98	162.02	130.17	24.76	–
		4.45	3.23	6.41		50.91	85.60	21.54	54.41	9.61	
GRIDGEN–SR	2,097,280	9.72	**2.94**	9.02	21.33	15.13	64.37	16.49	70.48	11.71	846.06
		0.56	0.27	0.41	0.20	0.75	7.52	2.37	21.29	5.57	30.78
GRIDGEN–DEG	2,097,664	8.50	**1.22**	5.49	14.22	3.64	9.35	3.80	30.80	5.86	369.47
		0.68	0.04	0.16	0.27	0.28	0.16	0.19	4.91	0.41	12.25
GOTO–8	2,097,152	**204.43**	–	214.20	236.63	–	–	–	–	–	–
		14.18		12.09	22.74						
GOTO–SR	2,097,152	**72.98**	163.26	82.72	190.09	2395.41	1407.59	1601.92	–	–	2534.18
		4.75	0.64	12.80	7.38	94.82	178.40	475.33			85.64
GRID–WIDE	2,162,672	1499.82	34.21	305.21	*error*	38.33	60.21	**31.92**	–	–	*error*
		196.15	1.05	73.16		1.18	1.46	1.12			
GRID–LONG	2,031,632	**47.13**	1804.92	51.38	*error*	–	–	–	–	–	*error*
		5.38	101.48	16.34							
GRID–SQUARE	2,097,152	420.45	2715.49	**217.73**	*error*	–	3523.53	–	–	–	*error*
		42.15	86.87	22.39			93.43				
ROAD–PATHS	2,653,624	94.37	141.91	**43.99**	*error*	178.05	174.28	164.20	–	–	*error*
		9.80	24.90	0.72		40.38	36.33	34.93			
ROAD–FLOW	2,653,624	131.93	119.44	**50.74**	*error*	164.37	176.40	153.19	–	–	*error*
		8.45	7.88	1.25		21.05	14.57	18.59			
VISION–RND	2,877,382	63.27	115.48	**44.50**	218.61	320.91	345.68	409.43	–	2987.68	–
		5.51	8.39	1.10	33.35	24.24	104.82	137.19		1070.66	
VISION–PROP	2,877,382	111.03	112.52	**61.70**	*error*	228.54	333.83	243.70	–	2112.62	–
		6.76	8.51	0.80		20.13	101.64	21.43		926.37	
VISION–INV	2,877,382	97.07	81.93	58.10	205.03	**57.15**	280.85	63.09	–	1451.90	–
		4.47	6.76	1.72	17.36	2.85	83.22	19.33		24.01	

substantially less iterations than them, although it performs either less or more arc pricing steps depending on the characteristics of the problem instance.

5. Conclusions

We have considered several efficient implementations of algorithms for solving the minimum-cost network flow problem, which is essential in a wide range of applications. The performance of these codes has been evaluated on numerous problem instances of various size and characteristics. This work provides a comprehensive experimental survey of currently available MCF solvers and thereby also gives guidelines for selecting a solution method for a certain application.

According to the presented results, we can conclude that implementations of the cost-scaling and primal network simplex algorithms are usually the most efficient and robust. In certain cases,

if an optimal flow need not be split into many paths, however, the augmenting path algorithms are superior to these methods.

The primal network simplex code developed by the author (NS) turned out to be significantly faster, often by an order of magnitude, than the other implementations of this method: MCF, MCFSimplex, and the NETOPT component of CPLEX. Furthermore, NS is generally the most efficient among all solvers considered in this study on relatively small networks (up to many thousands of nodes). However, the cost-scaling algorithms tend to be superior to simplex-based methods on huge sparse networks due to their better asymptotic behaviour in terms of the number of nodes. The cost-scaling code of the author (COS) performs similarly to or slightly slower than CS2, which is a highly efficient authoritative implementation of this algorithm. The LEDA library also implements this method, but it is slower and numerically less stable than COS and CS2. We also considered three other solvers, the dual version of MCFSimplex, RelaxIV, and PDNET, but they did not turn out to be robust. They are orders of magnitude slower than cost-scaling and primal simplex codes in many cases, although RelaxIV is very fast on certain kinds of easy problem instances.

Our implementations are not standalone solvers, but they are included in a versatile C++ network optimization library called LEMON [31,76]. This makes it easy to combine them with various other algorithms for solving complex network optimization problems. LEMON is an open-source library that can be used in both commercial and non-commercial software development under a permissive license. It is also involved in the COIN–OR project [25]. The author believes that this library with its great variety of efficient algorithms and data structures is a viable alternative to other graph libraries, such as the Boost Graph Library [19,97] and LEDA [4,80], as well as to the MCFClass project [10].

Acknowledgements

The author would like to thank Zoltán Király for his essential contribution to this work, as well as Alpár Jüttner and Balázs Dezső for many useful suggestions related to the implementations. Special thanks are due to the developers of CS2, MCF, RelaxIV, and PDNET for making these codes available for academic use: Andrew V. Goldberg and Boris V. Cherkassky; Andreas Löbel; Dimitri P. Bertsekas and Paul Tseng; Luis F. Portugal, Mauricio G.C. Resende, Geraldo Veiga, João Patrício, and Joaquim J. Júdice; respectively. The author also thanks Antonio Frangioni, Claudio Gentile, and Alessandro Bertolini for developing the MCFClass project including the MCFSimplex solver and the C++ translation of the Fortran code of RelaxIV. Finally, the author is grateful to Antonio Frangioni for his help with the MCFClass project.

Funding

This work was partially supported by grants (nos. CNK 77780 and CK 80124) from the National Development Agency of Hungary, based on a source from the Research and Technology Innovation Fund.

References

[1] R. Ahuja, A. Goldberg, J. Orlin, and R. Tarjan, *Finding minimum-cost flows by double scaling*, Tech. Rep. Sloan W.P. No. 2047-88, Stanford University, Stanford, CA, 1988.
[2] R. Ahuja, A. Goldberg, J. Orlin, and R. Tarjan, *Finding minimum-cost flows by double scaling*, Math. Program. 53 (1992), pp. 243–266 (a preliminary version appeared in [1]).
[3] R. Ahuja, T. Magnanti, and J. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1993.
[4] Algorithmic Solutions Software GmbH, *The LEDA Library, Version 5.0*, 2004. Available at http://www.algorithmic-solutions.com/leda/
[5] R. Armstrong and Z. Jin, *A new strongly polynomial dual network simplex algorithm*, Math. Program. 78 (1997), pp. 131–148.
[6] R. Armstrong, D. Klingman, and D. Whitman, *Implementation and analysis of a variant of the dual method for the capacitated transshipment problem*, Eur. J. Oper. Res. 4 (1980), pp. 403–420.

[7] F. Barahona and E. Tardos, *Note on Weintraub's minimum-cost circulation algorithm*, SIAM J. Comput. 18 (1989), pp. 579–583.
[8] R. Barr, F. Glover, and D. Klingman, *The alternating basis algorithm for assignment problems*, Math. Program. 13 (1977), pp. 1–13.
[9] R. Barr, F. Glover, and D. Klingman, *Enhancements to spanning tree labelling procedures for network optimization*, INFOR 17 (1979), pp. 16–34.
[10] A. Bertolini, A. Frangioni, and C. Gentile, *The MCFClass Project*, 2011. Available at http://www.di.unipi.it/di/groups/optimize/Software/MCF.html
[11] D. Bertsekas, *Network Optimization Codes*, 1994. Available at http://web.mit.edu/dimitrib/www/noc.htm
[12] D. Bertsekas and J. Eckstein, *Dual coordinate step methods for linear network flow problems*, Math. Program. 42 (1988), pp. 203–243.
[13] D. Bertsekas and P. Tseng, *The relax codes for linear minimum cost network flow problems*, Ann. Oper. Res. 13 (1988), pp. 125–190.
[14] D. Bertsekas and P. Tseng, *Relaxation methods for minimum cost ordinary and generalized network flow problems*, Oper. Res. 36 (1988), pp. 93–114.
[15] D. Bertsekas and P. Tseng, *RELAX-IV: A faster version of the relax code for solving minimum cost flow problems*, Tech. Rep. LIDS-P-2276, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, 1994.
[16] R. Bland and D. Jensen, *On the computational behavior of a polynomial-time network flow algorithm*, Tech. Rep. 661, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, NY, 1985.
[17] R. Bland and D. Jensen, *On the computational behavior of a polynomial-time network flow algorithm*, Math. Program. 54 (1992), pp. 1–39 (a preliminary version appeared in [16]).
[18] R. Bland, J. Cheriyan, D. Jensen, and L. Ladányi, *An empirical study of min cost flow algorithms*, in *Network Flows and Matching: First DIMACS Implementation Challenge*, D.S. Johnson and C.C. McGeoch, eds., DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 12, American Mathematical Society, Providence, RI, 1993, pp. 119–156.
[19] *Boost C++ Libraries*. Available at http://www.boost.org/
[20] G. Bradley, G. Brown, and G. Graves, *Design and implementation of large scale primal transshipment algorithms*, Manage. Sci. 24 (1977), pp. 1–34.
[21] T. Brunsch, K. Cornelissen, B. Manthey, and H. Röglin, *Smoothed Analysis of the Successive Shortest Path Algorithm*, Proceedings of the 24th ACM-SIAM Symposium on Discrete Algorithms, SODA'13, SIAM, Philadelphia, PA, 2013, pp. 1180–1189.
[22] U. Bünnagel, B. Korte, and J. Vygen, *Efficient implementation of the Goldberg–Tarjan minimum-cost flow algorithm*, Opt. Methods Softw. 10 (1998), pp. 157–174.
[23] R. Busacker and P. Gowen, *A procedure for determining a family of minimum-cost network flow patterns*, Tech. Rep. ORO-TP-15, Operations Research Office, The Johns Hopkins University, Bethesda, MD, 1960.
[24] B. Cherkassky and A. Goldberg, *On implementing the push-relabel method for the maximum flow problem*, Algorithmica 19 (1997), pp. 390–410.
[25] *COIN-OR—Computational Infrastructure for Operations Research*, 2012. Available at http://www.coin-or.org/
[26] W. Cunningham, *A network simplex method*, Math. Program. 11 (1976), pp. 105–116.
[27] G. Dantzig, *Application of the simplex method to a transportation problem*, in *Activity Analysis of Production and Allocation*, T. Koopmans, ed., John Wiley & Sons, Inc., New York, 1951, pp. 359–373.
[28] G. Dantzig, *Linear Programming and Extensions*, Princeton University Press, Princeton, 1963.
[29] A. Dasdan, *Experimental analysis of the fastest optimum cycle ratio and mean algorithms*, ACM Trans. Des. Autom. Electron. Syst. 9 (2004), pp. 385–418.
[30] A. Dasdan and R. Gupta, *Faster maximum and minimum mean cycle algorithms for system performance analysis*, IEEE Trans. Comput. Aided Des. 17 (1998), pp. 889–899.
[31] B. Dezső, A. Jüttner, and P. Kovács, *LEMON—An open source C++ graph template library*, Electron. Notes Theor. Comput. Sci. 264 (2011), pp. 23–45.
[32] DIMACS, *Network Flows and Matching: First DIMACS Implementation Challenge*, 1990–1991. Available at ftp://dimacs.rutgers.edu/pub/netflow/
[33] DIMACS, *Shortest Paths: Ninth DIMACS Implementation Challenge*, 2005–2006. Available at http://www.dis.uniroma1.it/~challenge9/
[34] J. Edmonds and R. Karp, *Theoretical improvements in algorithmic efficiency for network flow problems*, in *Combinatorial Structures and Their Applications*, R.K. Guy, ed., Gordon and Breach, New York, 1970, pp. 93–96.
[35] J. Edmonds and R. Karp, *Theoretical improvements in algorithmic efficiency for network flow problems*, J. ACM 19 (1972), pp. 248–264 (a preliminary version appeared in [34]).
[36] *EGRES—MTA-ELTE Egerváry Research Group on Combinatorial Optimization*, 2012. Available at http://www.cs.elte.hu/egres/
[37] L. Ford and D. Fulkerson, *Constructing maximal dynamic flows from static flows*, Oper. Res. 6 (1958), pp. 419–433.
[38] L. Ford and D. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, 1962.
[39] A. Frangioni and A. Manca, *A computational study of cost reoptimization for min-cost flow problems*, INFORMS J. Comput. 18 (2006), pp. 61–70.
[40] A. Frank, *Connections in Combinatorial Optimization*, Oxford University Press, Oxford, 2011.
[41] M. Fredman and R. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, J. ACM 34 (1987), pp. 596–615.

[42] S. Fujishige, *A capacity-rounding algorithm for the minimum-cost circulation problem: A dual framework of the Tardos algorithm*, Math. Program. 35 (1986), pp. 298–308.
[43] D. Fulkerson, *An out-of-kilter method for minimal-cost flow problems*, J. Soc. Ind. Appl. Math. 9 (1961), pp. 18–27.
[44] H. Gabow and R. Tarjan, *Faster scaling algorithms for network problems*, SIAM J. Comput. 18 (1989), pp. 1013–1036.
[45] Z. Galil and E. Tardos, *An* $O(n^2(m + n \log n) \log n)$ *Min-Cost Flow Algorithm*, Proceedings of the 27th Symposium on Foundations of Computer Science, FOCS'86, IEEE Computer Society Press, Washington, DC, 1986, pp. 1–9.
[46] Z. Galil and E. Tardos, *An* $O(n^2(m + n \log n) \log n)$ *min-cost flow algorithm*, J. ACM 35 (1988), pp. 374–386 (a preliminary version appeared in [45]).
[47] L. Georgiadis, A. Goldberg, R. Tarjan, and R. Werneck, *An Experimental Study of Minimum Mean Cycle Algorithms*, Proceedings of the 6th International Workshop on Algorithm Engineering and Experiments, ALENEX 2009, SIAM, New York, 2009, pp. 1–13.
[48] F. Glover, D. Karney, D. Klingman, and A. Napier, *A computation study on start procedures, basis change criteria, and solution algorithms for transportation problems*, Manage. Sci. 20 (1974), pp. 793–813.
[49] A. Goldberg, *An efficient implementation of a scaling minimum-cost flow algorithm*, J. Algorithms 22 (1997), pp. 1–29.
[50] A. Goldberg, *The Partial Augment-Relabel Algorithm for the Maximum Flow Problem*, Proceedings of the 16th Annual European Symposium on Algorithms, ESA'08, Springer-Verlag, Heidelberg, 2008, pp. 466–477.
[51] A. Goldberg and M. Kharitonov, *On implementing scaling push-relabel algorithms for the minimum-cost flow problem*, in *Network Flows and Matching: First DIMACS Implementation Challenge*, D. Johnson and C. McGeoch, eds., DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 12, American Mathematical Society, Providence, RI, 1993, pp. 157–198.
[52] A. Goldberg and R. Tarjan, *Solving Minimum-Cost Flow Problems by Successive Approximation*, Proceedings of the 19th ACM Symposium on Theory of Computing, STOC'87, ACM Press, New York, 1987, pp. 7–18.
[53] A. Goldberg and R. Tarjan, *Finding Minimum-Cost Circulations by Canceling Negative Cycles*, Proceedings of the 20th ACM Symposium on Theory of Computing, STOC'88, ACM Press, New York, 1988, pp. 388–397.
[54] A. Goldberg and R. Tarjan, *A new approach to the maximum-flow problem*, J. ACM 35 (1988), pp. 921–940.
[55] A. Goldberg and R. Tarjan, *Finding minimum-cost circulations by canceling negative cycles*, J. ACM 36 (1989), pp. 873–886 (a preliminary version appeared in [53]).
[56] A. Goldberg and R. Tarjan, *Finding minimum-cost circulations by successive approximation*, Math. Oper. Res. 15 (1990), pp. 430–466 (a preliminary version appeared in [52]).
[57] A. Goldberg, E. Tardos, and R. Tarjan, *Network flow algorithms*, in *Paths, Flows and VLSI-Layout*, B. Korte et al., eds., Springer-Verlag, Berlin, 1990, pp. 101–164.
[58] D. Goldfarb and J. Hao, *Polynomial-time primal simplex algorithms for the minimum cost network flow problem*, Algorithmica 8 (1992), pp. 145–160.
[59] M. Grigoriadis, *An efficient implementation of the network simplex method*, Math. Program. Stud. 26 (1986), pp. 83–111.
[60] M. Hartmann and J. Orlin, *Finding minimum cost to time ratio cycles with small integral transit times*, Networks 23 (1993), pp. 567–574.
[61] R. Helgason and J. Kennington, *An efficient procedure for implementing a dual simplex network flow algorithm*, AIEE Trans. 9 (1977), pp. 63–68.
[62] R. Howard, *Dynamic Programming and Markov Processes*, The MIT Press, Cambridge, MA, 1960.
[63] *IBM ILOG CPLEX Optimization Studio, V12.4*, 2011. Available at http://www.ibm.com/software/integration/optimization/cplex-optimization-studio/
[64] IG Systems, Inc., *CS2 Software, Version 4.6*, 2009. Available at http://www.igsystems.com/cs2/
[65] M. Iri, *A new method for solving transportation-network problems*, J. Oper. Res. Soc. Japan 3 (1960), pp. 27–87.
[66] W. Jewell, *Optimal flow through networks*, Tech. Rep. No. 8, Operations Research Center, MIT, Cambridge, MA, 1958.
[67] E. Johnson, *Networks and basic solutions*, Oper. Res. 14 (1966), pp. 619–623.
[68] D. Kelly and G. O'Neill, *The minimum cost flow problem and the network simplex solution method*, Ph.D. diss., University College, Dublin, 1991.
[69] J. Kennington and R. Helgason, *Algorithms for Network Programming*, John Wiley & Sons, Inc., New York, 1980.
[70] Z. Király and P. Kovács, *Efficient implementations of minimum-cost flow algorithms*, Acta Universitatis Sapientiae, Informatica 4 (2012), pp. 67–118.
[71] M. Klein, *A primal method for minimal cost flows with applications to the assignment and transportation problems*, Manage. Sci. 14 (1967), pp. 205–220.
[72] D. Klingman, A. Napier, and J. Stutz, *NETGEN: A program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems*, Manage. Sci. 20 (1974), pp. 814–821.
[73] B. Korte and J. Vygen, *Combinatorial Optimization: Theory and Algorithms*, 5th ed., Springer-Verlag, Berlin, 2012.
[74] H. Kuhn, *The Hungarian method for the assignment problem*, Naval Res. Logist. Quart. 2 (1955), pp. 83–97.
[75] Y. Lee, *Computational analysis of network optimization algorithms*, Ph.D. diss., Department of Civil and Environmental Engineering, (Supervisor: J.B. Orlin) MIT, Cambridge, MA, 1993.
[76] *LEMON—Library for Efficient Modeling and Optimization in Networks*, 2012. Available at http://lemon.cs.elte.hu/
[77] A. Löbel, *Solving large-scale real-world minimum-cost flow problems by a network simplex method*, Tech. Rep. SC 96-7, Zuse Institute Berlin (ZIB), Berlin, 1996.
[78] A. Löbel, *MCF Version 1.3—A Network Simplex Implementation*, 2004. Available at http://www.zib.de/

[79] *Max-Flow Problem Instances in Vision*, Computer Vision Research Group, University of Western Ontario, 2008. Available at http://vision.csd.uwo.ca/data/maxflow/
[80] K. Mehlhorn and S. Näher, *LEDA: A Platform for Combinatorial and Geometric Computing*, Cambridge University Press, New York, 1999.
[81] G. Minty, *Monotone networks*, Proc. R. Soc. Lond. Ser. A 257 (1960), pp. 194–212.
[82] J. Mulvey, *Pivot strategies for primal-simplex network codes*, J. ACM 25 (1978), pp. 266–270.
[83] J. Orlin, *Genuinely polynomial simplex and non-simplex algorithms for the minimum cost flow problem*, Tech. Rep. 1615-84, Sloan School of Management, MIT, Cambridge, MA, 1984.
[84] J. Orlin, *A Faster Strongly Polynomial Minimum Cost Flow Algorithm*, Proceedings of the 20th ACM Symposium on Theory of Computing, STOC'88, ACM Press, New York, 1988, pp. 377–387.
[85] J. Orlin, *A faster strongly polynomial minimum cost flow algorithm*, Oper. Res. 41 (1993), pp. 338–350 (a preliminary version appeared in [84]).
[86] J. Orlin, *A polynomial time primal network simplex algorithm for minimum cost flows*, Math. Program. 78 (1997), pp. 109–129.
[87] J. Orlin, S. Plotkin, and E. Tardos, *Polynomial dual network simplex algorithms*, Math. Program. 60 (1993), pp. 255–276.
[88] S. Plotkin and E. Tardos, *Improved Dual Network Simplex*, Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms, SODA'90, SIAM, Philadelphia, PA, 1990, pp. 367–376.
[89] L. Portugal, M. Resende, G. Veiga, J. Patrício, and J. Júdice, *Fortran subroutines for network flow optimization using an interior point algorithm*, Tech. Rep. TD-5X2SLN, AT&T Labs Research, 2004.
[90] L. Portugal, M. Resende, G. Veiga, J. Patrício, and J. Júdice, *PDNET—Software for Network Flow Optimization Using an Interior Point Method*, 2004. Available at http://www.research.att.com/ mgcr/pdnet/
[91] L. Portugal, M. Resende, G. Veiga, J. Patrício, and J. Júdice, *Fortran subroutines for network flow optimization using an interior point algorithm*, Pesquisa Operacional 28 (2008), pp. 243–261 (a preliminary version appeared in [89]).
[92] M. Resende and P. Pardalos, *Interior point algorithms for network flow problems*, in *Advances in Linear and Integer Programming*, J.E. Beasley, ed., Oxford University Press, Oxford, 1996, pp. 147–187.
[93] M. Resende and G. Veiga, *An efficient implementation of a network interior point method*, in *Network Flows and Matching: First DIMACS Implementation Challenge*, D.S. Johnson and C.C. McGeoch, eds., DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 12, American Mathematical Society, Providence, RI, 1993, pp. 299–348.
[94] M. Resende and G. Veiga, *An annotated bibliography of network interior point methods*, Networks 42 (2003), pp. 114–121.
[95] H. Röck, *Scaling techniques for minimal cost network flows*, in *Discrete Structures and Algorithms*, U. Pape, ed., Carl Hanser, München, 1980, pp. 181–191.
[96] A. Schrijver, *Combinatorial Optimization: Polyhedra and Efficiency*, Springer-Verlag, Berlin, 2003.
[97] J. Siek, L.Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*, Addison-Wesley, Reading, PA, 2002.
[98] A. Sifaleras, *Minimum cost network flows: Problems, algorithms, and software*, Yugosl. J. Oper. Res. 23 (2013), pp. 3–17.
[99] D. Sleator and R. Tarjan, *A data structure for dynamic trees*, J. Comput. Syst. Sci. 26 (1983), pp. 362–391.
[100] P. Sokkalingam, R. Ahuja, and J. Orlin, *New polynomial-time cycle-canceling algorithms for minimum-cost flows*, Networks 36 (2000), pp. 53–63.
[101] V. Srinivasan and G. Thompson, *Benefit–cost analysis of coding techniques for the primal transportation algorithm*, J. ACM 20 (1973), pp. 194–213.
[102] E. Tardos, *A strongly polynomial minimum cost circulation algorithm*, Combinatorica 5 (1985), pp. 247–255.
[103] R. Tarjan, *Efficiency of the primal network simplex algorithm for the minimum-cost circulation problem*, Math. Oper. Res. 16 (1991), pp. 272–291.
[104] R. Tarjan, *Dynamic trees as search trees via Euler tours, applied to the network simplex algorithm*, Math. Program. 78 (1997), pp. 169–177.
[105] N. Tomizawa, *On some techniques useful for solution of transportation network problems*, Networks 1 (1971), pp. 173–194.
[106] J. Vygen, *On Dual Minimum Cost Flow Algorithms*, Proceedings of the 32nd ACM Symposium on Theory of Computing, STOC'00, ACM Press, New York, 2000, pp. 117–125.
[107] J. Vygen, *On dual minimum cost flow algorithms*, Math. Methods Oper. Res. 56 (2002), pp. 101–126 (a preliminary version appeared in [106]).
[108] A. Weintraub, *A primal algorithm to solve network flow problems with convex costs*, Manage. Sci. 21 (1974), pp. 87–97.
[109] M. Yakovleva, *A problem on minimum transportation cost*, in *Applications of Mathematics in Economic Research*, V.S. Nemchinov, ed., Izdat. Social'no-Ekon. Lit., Moscow, 1959, pp. 390–399.