Proceedings of the 8th International Conference on Applied Informatics
Eger, Hungary, January 27–30, 2010. Vol. 2. pp. 227–235.

An Experimental Study of Minimum Cost Flow Algorithms*

Zoltán Király[a], Péter Kovács[b]

[a] Department of Computer Science and CNL
[b] Department of Algorithms and Their Applications and CNL

Eötvös Loránd University, Budapest
e-mail: kiraly@cs.elte.hu, kpeter@inf.elte.hu

Abstract

This paper presents an experimental study of efficient algorithms for the minimum cost flow problem. It is more comprehensive than earlier surveys both in terms of the range of considered implementations and the size of the test instances. In the cost scaling algorithm, Goldberg's partial augment-relabel method was also applied, which is a novel result.

The studied algorithms were implemented as part of the LEMON C++ optimization library (`http://lemon.cs.elte.hu`). They were compared to widely known efficient solvers, namely, the corresponding method of the LEDA library and three public codes: CS2, RelaxIV, and MCF. Our implementations turned out to be comparable and often superior to them.

Keywords: graph, network, flow, minimum cost flow, algorithm, implementation, optimization, performance

MSC: 05C21

1. Introduction

The single commodity minimum cost flow problem is one of the most fundamental models in network flow theory. It is to find a feasible flow of minimum total cost from a set of supply nodes to a set of demand nodes in a network with capacity constraints and arc costs. This model can be used directly in various real world applications, which arise in transportation, logistics, telecommunication, network design, resource planning, scheduling, and many other industries. Moreover, it

*Research is supported by EGRES group (MTA-ELTE) and OTKA grants CNK 77780, K 60802, CK 80124 and NK 67867.

often arises as a subproblem in more complex optimization models, such as multicommodity flow problems. A comprehensive survey of the theory and applications of this problem can be found in the book of Ahuja, Magnanti, and Orlin [1].

In the last decades, several algorithms were developed for the minimum cost flow problem. They have been studied and compared both from theoretical and practical aspects, and various implementations are available under different license terms. Our main contribution is the highly efficient implementation of several algorithms with some new heuristics and the comparative analysis of their performance in practice. Furthermore, the application of Goldberg's recent partial augment-relabel idea [7] in the cost scaling algorithm is an essential novel result. Our implementations are available with full source code as part of the LEMON graph library [12]. LEMON is an abbreviation of *Library for Efficient Modeling and Optimization in Networks*. It is an open source C++ template library with focus on combinatorial optimization tasks related to graphs and networks.

Numerous benchmark tests were performed on many kinds of large-scale random networks (containing up to millions of nodes and arcs), as well as some real-life problem instances. In most cases, the network simplex and the cost scaling implementations were the fastest. The latter one typically performs better on large sparse networks. Compared to various public solvers, our implementations proved to be competitive or even faster.

The rest of this paper is organized as follows. Section 2 summarizes the used definitions and notations. Section 3 describes the implemented algorithms and their variants. Section 4 presents the main experimental results. Finally, the conclusions are drawn in Section 5.

2. Definitions and Notations

Let $G = (V, A)$ be a directed graph consisting of $n = |V|$ nodes and $m = |A|$ arcs. We associate with each arc $(i, j) \in A$ a *lower bound* $l_{ij} \geq 0$, an *upper bound* $u_{ij} \geq l_{ij}$, and a *cost* c_{ij}, which denotes the cost per unit flow on the arc. Each node $i \in V$ has a signed *supply* value b_i. We assume that all data are integral and we wish to find an integral feasible flow of minimum total cost satisfying the supply/demand constraints at each node. Therefore, the minimum cost flow problem can be stated as

$$\min \sum_{(i,j)\in A} c_{ij} x_{ij} \tag{2.1}$$

subject to

$$\sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = b_i \qquad \forall i \in V, \tag{2.2}$$

$$l_{ij} \leq x_{ij} \leq u_{ij} \qquad \forall (i,j) \in A, \tag{2.3}$$

where $\sum_{i\in V} b_i = 0$. Without loss of generality, we may assume that all lower bounds are zero and all arc costs are non-negative [1].

For a solution $x \geq 0$ of the problem, we can define the *residual network* $G_x = (V, A_x)$, which contains *forward* and *backward* arcs. A forward arc $(i,j) \in A_x$ corresponds to each $(i,j) \in A$ for which $r_{ij} = u_{ij} - x_{ij}$ is positive. A backward arc $(j,i) \in A_x$ corresponds to each $(i,j) \in A$ for which $r_{ji} = x_{ij}$ is positive. These r_{ij} and r_{ji} values are called the residual capacities of the arcs in G_x. The cost of a forward arc (i,j) is c_{ij}, while the cost of a backward arc (j,i) is $-c_{ij}$. The optimality conditions and most of the solution methods of this problem can be stated more conveniently in terms of this residual network.

Theorem 2.1 (Negative cycle optimality condition). *A feasible solution x of the minimum cost flow problem is optimal if and only if the residual network G_x contains no directed cycle of negative total cost.*

The linear programming dual solution of this problem is represented by signed *node potentials* π_i. For a given potential function π, the reduced cost of an arc (i,j) is defined as $c^{\pi}_{ij} = c_{ij} + \pi_i - \pi_j$. The following optimality condition is an equivalent reformulation of Theorem 2.1 using node potentials and reduced costs.

Theorem 2.2 (Reduced cost optimality condition). *A feasible solution x of the minimum cost flow problem is optimal if and only if for some node potential function π, the* reduced cost *of each arc in the residual network G_x is non-negative.*

3. Algorithms

This section provides an overview of the implemented algorithms and their variants. The detailed explanation of these methods are omitted here due to page limit, but they can be found in several books and papers. For more information about our implementations, the readers are referred to the documentation and source code of the LEMON library [12].

3.1. Cycle-Canceling Algorithms

Cycle-canceling is the simplest solution method for the minimum cost flow problem. It applies a primal approach based on Theorem 2.1. First, a feasible solution is found, which can be performed by computing a maximum flow. After that, the algorithm successively finds directed cycles of negative total cost in the current residual network and augments flow along them. When no negative cycle can be found, the solution is optimal.

This method has many variants of quite different efficiency. In LEMON, two strongly polynomial algorithms were implemented, both of which is due to Goldberg and Tarjan [8]. The first one is the *minimum mean cycle-canceling* algorithm (MMCC), which runs in $O(n^2 m^3 \log n)$ time, the other one is its improved variant, the *cancel and tighten* algorithm (CAT), which runs in $O(n^2 m^2 \log n)$. Apart from these methods, a *simple cycle-canceling* variant (SCC) was also implemented, in

which the negative cycles are found using the Bellman-Ford algorithm with successively increased limit for the number of iterations. Our experiments show that the SCC implementation greatly outperforms MMCC, but CAT is orders of magnitude faster than both of them. Therefore, we only consider CAT in the following comparisons.

3.2. Augmenting Path Algorithms

We also implemented two dual methods, the *successive shortest path* algorithm (SSP) and the *capacity scaling* algorithm (CAS) of Edmonds and Karp [4]. Contrary to the primal approach of the cycle-canceling methods, the SSP algorithm maintains a flow that is not necessarily feasible (it may violate constraint (2.2)) and node potentials so that they satisfy the conditions of Theorem 2.2. At each step, the algorithm sends flow from an excess node to a deficit node along a shortest path in the residual network with respect to the reduced costs and modifies the potentials using the computed node distances. Therefore, it successively decreases the total excess of the nodes until the solution becomes feasible.

The general SSP method performs $O(nU)$ iterations, where U denotes the maximum supply value. Each iteration runs Dijkstra's algorithm to find a shortest path from a selected excess node to any demand node. The CAS algorithm is an improved version of SSP that uses a capacity scaling scheme to reduce the number of iterations to $O(m \log U)$. Both of these algorithms are relatively efficient in practice, but CAS is typically faster than SSP except for the simpler test cases in which the optimal solution consists of only a few paths.

3.3. Cost Scaling Algorithm

The *cost scaling* method (COS) is another widely used approach for solving the minimum cost flow problem based on a primal-dual scheme [9]. At each phase, an ϵ-optimal primal-dual solution pair is computed for a suitable $\epsilon > 0$ value. It means that $c^{\pi}_{ij} \geq -\epsilon$ holds for each arc (i, j) in the residual network. After that, ϵ is divided by a factor $\alpha > 1$ and another phase is performed. When ϵ decreases below $1/n$, then optimal primal-dual solutions are found. This algorithm is one of the fastest solution methods, however, its practical performance highly depends on the proper application of several complicated heuristics [6].

We implemented three variants of this method using different operations within the scaling phases. The standard cost scaling algorithm performs local push and relabel operations, thus it can be viewed as a generalization of the well-known push-relabel algorithm for the maximum flow problem. Another variant replaces the local push operations with augmentations along admissible paths between excess and deficit nodes. The third approach performs partial augmentations based on Goldberg's recent algorithm for the maximum flow problem [7], which attains a good compromise between the former two methods. As far as we know, our implementation is the first application of this idea to the minimum cost flow problem.

According to our tests, it turned out to be significantly faster than the other two variants on virtually all problem instances.

3.4. Network Simplex Algorithm

The *network simplex* algorithm (NS) is the specialized version of the linear programming (LP) simplex method directly for the minimum cost flow problem [10]. The LP variables correspond to the arcs of the graph and the LP bases are represented by spanning trees along with suitable flow values and node potentials. At each iteration, we attempt to reduce the objective function value by moving from the current spanning tree solution to another. For this, a non-tree arc violating the optimality condition is added to the spanning tree, which determines a negative cycle. This cycle is canceled by augmenting flow along it and one of the saturated arcs is removed from the tree. This whole operation is called pivot. If no suitable incomming arc can be selected, then the flow is optimal.

The efficient implementation of the network simplex method requires a complex data structure for storing and updating the spanning trees. We applied the XTI method for this purpose, which is one of the most efficient schemes [2]. Another crucial part of the implementation is the strategy of selecting the entering arcs for the pivots. We implemented five different methods: *first eligible*, *best eligible*, *candidate list*, *altering list*, and *block search*. The latter two strategies performed the best in our benchmark tests, but block search turned out to be more robust, so it was selected to be the default pivot rule for NS.

4. Experimental Results

4.1. Test Framework

The benchmark tests of the implemented algorithms were performed on an AMD Opteron Dual Core 2.2 GHz machine with 16GB RAM and 1MB cache, running openSUSE 10.1 operating system. The codes were compiled with GCC 4.1 using -O3 optimization flag. Our test instances include both random and real-life networks containing up to millions of nodes and arcs. The random problems were created using well-known standard generators NETGEN and GOTO. Since the density of the network is an essential parameter, we studied both sparse networks, for which $m = 8n$ and relatively dense networks, for which m is about $n\sqrt{n}$. The real-world instances are based on segmentation problems of medical image processing, which are available at `http://vision.csd.uwo.ca/`.

4.2. Benchmark Results

Table 1 and Figure 1 compare our implementations on problem instances generated with NETGEN. The cycle-canceling algorithms (SCC, MMCC, CAT) are clearly

the slowest methods, but CAT is much faster than SCC and MMCC. The augmenting path methods are considerably faster, CAS typically outperforms the basic SSP algorithm. Cost scaling (COS) and network simplex (NS) are obviously the most efficient solution methods. For large sparse graphs, COS is significantly faster than NS, otherwise NS typically performed better.

n	m/n	SCC	MMCC	CAT	SSP	CAS	COS	NS
2^{12}	8	55.504s	407.65s	2.0093s	2.1059s	0.5982s	0.1471s	0.0451s
2^{16}	8	–	–	240.51s	130.42s	87.742s	5.6940s	5.3486s
2^{20}	8	–	–	–	–	–	150.11s	665.98s
2^{12}	64	1185.5s	6855.7s	20.252s	8.3316s	8.7219s	0.5906s	0.2209s
2^{16}	256	–	–	–	–	–	114.10s	58.127s

Table 1: Comparison of our implementations on NETGEN instances

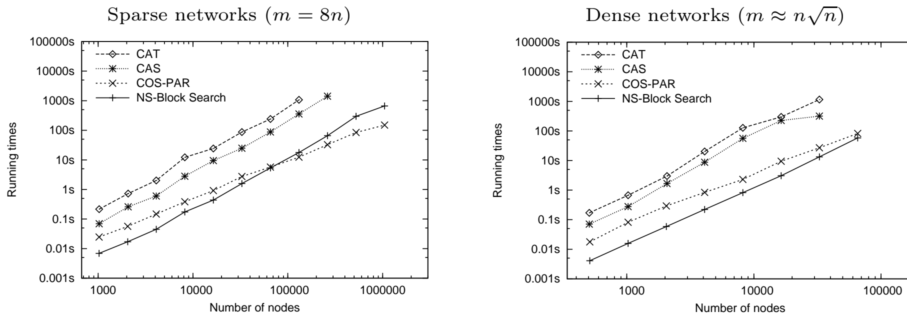

Figure 1: Comparison of our implementations on NETGEN instances

The following tables and charts compare our fastest algorithms (COS and NS) to efficient public solvers for the minimum cost flow problem. One of these solvers is the MIN_COST_FLOW method of the LEDA library [11], which implements a cost scaling algorithm. The other three solvers are available under the MCFClass project [5]: (1) CS2, which is based on the cost scaling version 3.7 C code of Goldberg and Cherkassky [6], (2) RelaxIV, which is based on the original FORTRAN code of Bertsekas and Tseng [3], (3) MCF, which is based on the version 1.1 C code of Löbel [13].

Tables 2 and 3 contain benchmark results on random networks generated with NETGEN and GOTO, respectively. GOTO is known to create much harder instances than NETGEN, which is also verified by our results. Therefore, the studied GOTO networks were smaller than the NETGEN graphs.

COS, CS2 and LEDA are three different implementations of the cost scaling algorithm. The fastest one is CS2, which is developed by the author of this method. Our COS implementation usually turned out to be faster than LEDA (except for the sparse GOTO networks). Moreover, LEDA failed on the largest sparse network

with *cost overflow* error. MCF is a network simplex implementation, but it proved to be much slower than the NS algorithm of LEMON. RelaxIV is an interesting relaxation algorithm. It was obviously the fastest on the NETGEN networks, however it performs rather poorly on harder instances, such as GOTO networks.

Figure 2 shows the benchmark results for the real-world networks. LEDA failed to solve these huge problems, but the other solvers and our algorithms could solve them. COS and CS2 turned out to be the most efficient implementations on these networks. NS also performed rather good, but MCF and RelaxIV were much slower.

		LEMON		Other solvers			
n	m/n	COS	NS	LEDA	CS2	MCF	RelaxIV
2^{12}	8	0.1471s	0.0451s	0.1539s	0.1094s	0.1254s	0.0613s
2^{16}	8	5.6940s	5.3486s	9.0663s	4.8089s	15.800s	5.0782s
2^{20}	8	150.11s	665.98s	*error*	87.344s	1916.3s	77.833s
2^{12}	64	0.8378s	0.2209s	1.7748s	0.5906s	0.7098s	0.6887s
2^{16}	256	82.166s	58.127s	464.09s	114.10s	594.96s	41.502s

Table 2: Comparison of our implementations and other solvers on NETGEN instances

		LEMON		Other solvers			
n	m/n	COS	NS	LEDA	CS2	MCF	RelaxIV
2^{12}	8	1.4900s	0.3109s	0.6177s	1.5007s	4.8567s	21.273s
2^{14}	8	15.134s	6.0875s	5.3180s	6.7113s	232.63s	501.95s
2^{16}	8	192.17s	182.08s	91.291s	56.276s	–	–
2^{12}	64	10.031s	2.3758s	10.711s	5.8248s	43.155s	514.53s
2^{14}	128	117.37s	136.99s	295.89s	82.825s	2142.1s	–

Table 3: Comparison of our implementations and other solvers on GOTO instances

5. Conclusions

We implemented various algorithms for the minimum cost flow problem, they are available as part of the LEMON open source C++ optimization library. The network simplex and the cost scaling algorithms proved to be the fastest solution methods. The cost scaling implementation was usually faster on large and relatively sparse networks. Our implementations turned out to be competitive or often faster than widely used efficient solvers.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice-Hall, Inc., 1993.

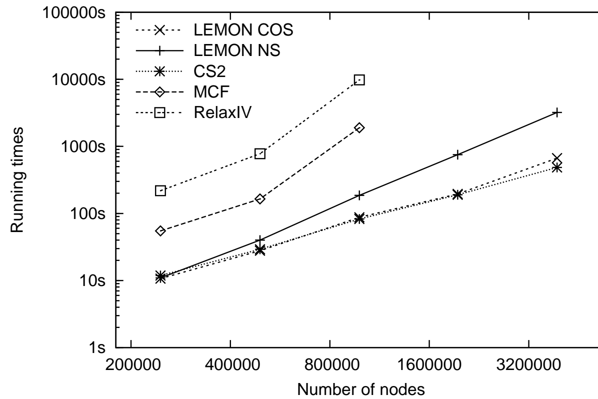

Figure 2: Comparison of our implementations and other solvers on real-life networks

[2] R. Barr, F. Glover, and D. Klingman. Enhancements to spanning tree labelling procedures for network optimization. *INFOR*, 17(1):16–34, 1979.

[3] D. P. Bertsekas and P. Tseng. RELAX-IV: A faster version of the relax code for solving minimum cost flow problems. Technical Report LIDS-P-2276, Dept. of Electrical Engineering and Computer Science, M.I.T., Cambridge, MA, 1994.

[4] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[5] A. Frangioni and A. Manca. A computational study of cost reoptimization for min-cost flow problems. *INFORMS J. On Computing*, 18(1):61–70, 2006.

[6] A. V. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *J. Algorithms*, 22(1):1–29, 1997.

[7] A. V. Goldberg. The partial augment-relabel algorithm for the maximum flow problem. *16th Annual European Symp. on Algorithms*, pages 466–477, 2008.

[8] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. *J. ACM*, 36(4):873–886, 1989.

[9] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by successive approximation. *Mathematics of Op. Res.*, 15(3):430–466, 1990.

[10] D. J. Kelly and G. M. O'Neill. The minimum cost flow problem and the network simplex method. Master's thesis, University College, Dublin, 1991.

[11] LEDA – Library of Efficient Data Types and Algorithms. `http://www.algorithmic-solutions.com/`, 2010.

[12] LEMON – Library for Efficient Modeling and Optimization in Networks. `http://lemon.cs.elte.hu/`, 2010.

[13] A. Löbel. Solving large-scale real-world minimum-cost flow problems by a network simplex method. Technical Report SC 96-7, Zuse Institute Berlin (ZIB), Berlin, Germany, 1996.

Zoltán Király, Péter Kovács
Pázmány Péter sétány 1/C, H-1117 Budapest