A Computational Study of Cost Reoptimization for Min-Cost Flow Problems

Antonio Frangioni
Dipartimento di Informatica, Università di Pisa, Largo B. Pontecorvo, 1, 56127 Pisa, Italy, frangio@di.unipi.it

Antonio Manca
Dipartimento di Ingegneria del Territorio, Università di Cagliari, Piazza d'Armi, 09123 Cagliari, Italy, amanca@unica.it

In the last two decades, a number of algorithms for the linear single-commodity min-cost flow (MCF) problem have been proposed, and several efficient codes are available that implement different variants of the algorithms. The practical significance of the algorithms has been tested by comparing the time required by their implementations for solving "from-scratch" instances of MCF, of different classes, as the size of the problem (number of nodes and arcs) increases. However, in many applications several closely related instances of MCF have to be sequentially solved, so that reoptimization techniques can be used to speed up computations, and the most attractive algorithm is the one that minimizes the total time required to solve all the instances in the sequence. In this paper we compare the performances of four different efficient implementations of algorithms for MCF under cost reoptimization in the context of decomposition algorithms for the multicommodity min-cost flow (MMCF) problem, showing that for some classes of instances the relative performances of the codes doing "from-scratch" optimization do not accurately predict the relative performances when reoptimization is used. Since the best solver depends both on the class and on the size of the instance, this work also shows the usefulness of a standard interface for MCF problem solvers that we have proposed and implemented.

Key words: analysis of algorithms; computers-computer science: software; networks-graphs: flow algorithms; networks-graphs: multicommodity
History: Accepted by William Cook, Area Editor for Design and Analysis of Algorithms; received March 2003; revised December 2003; accepted February 2004.

1. Introduction

The linear single-commodity min-cost flow (MCF) problem is a very interesting problem sitting "on the edge" between linear and combinatorial optimization. Given a directed graph $G = (N, A)$, with $n = |N|$ nodes and $m = |A|$ arcs, an m-vector c of arc costs, an m-vector u of arc upper capacities, and an n-vector b of node surpluses, the problem is defined as

$$\begin{array}{lll} \min & \sum_{(i,j)\in A} c_{ij}x_{ij} & \\ & \sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = b_i & \forall i \in N \\ & 0 \leq x_{ij} \leq u_{ij} & \forall (i,j) \in A \end{array} \tag{1}$$

In other words, a *flow* x of minimal cost has to be found that satisfies both node-balancing constraints (for each node, the flow leaving the node minus the flow entering must be equal to the flow produced by the node) and arc-capacity constraints. This problem has a huge set of applications, either in itself (Ahuja et. al. 1993) or — more often — as a submodel of more complex and demanding problems, as in Bertsekas (1998), Frangioni and Gallo (1999), Löbel (1999), Gendron et. al (2001) and, again, Ahuja et. al. (1993). This is testified by the enormous amount of research that has been invested in developing efficient solution algorithms for MCF problems (Ahuja et. al. 1993, Bertsekas 1991) either by specializing LP algorithms — such as the simplex method (Ahuja et. al. 1993, Helgason and Kennington 1995, Löbel 1996) or the interior point method (Resende and Pardalos 1996) — to the network case, or by developing ad-hoc approaches such as those of Bertsekas and Eckstein (1988), Bertsekas and Tseng (1988a), and Goldberg and Tarjan (1990).

It is therefore extremely interesting, both for practitioners and for algorithm developers, to evaluate which algorithm is the most efficient in practice to solve MCF. This is usually done as follows: a large set of — usually randomly generated — instances of different classes is collected, and the running time required by some implementation of different algorithms for solving these instances is computed. The algorithm whose implementation is, on average, faster for most classes of instances as the size of the problem (number of nodes and arcs) increases is usually regarded as the most efficient, or at least the most promising.

However, in many applications several closely related instances of MCF have to be sequentially solved, each instance differing from the previous one only for a possibly small fraction of the data, so that reoptimization techniques can be used to speed up computations; we will not discuss all these applications here, the interested reader being referred to Ahuja et. al. (1993) and the discussion in Amini and Barr (1993). In this setting, the most attractive algorithm is the one that minimizes the total time required to solve all the instances, as opposed to the one that just solves the first instance "from scratch" more efficiently. Also, in most of these applications, the size of the instances to be solved is not extremely large, either because each MCF problem captures only a part of a very large-scale problem (as e.g. in Frangioni and Gallo 1999), or because the MCF computation is used within approaches for difficult combinatorial problems (as e.g. in Gendron et al. 2001), which typically are not of very large scale.

Thus, a guideline for choosing the correct MCF algorithm in this setting would be valuable for practitioners and researchers. Some effort for analyzing this issue has been done by several researchers, typically after having implemented some specific MCF solver, but the results were limited and hardly conclusive, as clearly discussed in Amini and Barr (1993). A much more focused effort, using sophisticated statistical analysis, was performed in Amini and Barr (1993) for three `FORTRAN` implementations of MCF algorithms: a primal simplex algorithm, a dual simplex algorithm, and an out-of-kilter approach. Although the results of that paper were quite general and showed some trends that presumably still hold true, we believe that part of the analysis has to be updated, for the following reasons:

- The field of practical implementations of MCF algorithms has significantly progressed since the publication of Amini and Barr (1993); for instance, efficient implementations of primal-dual algorithms have been shown to outperform simplex-based approaches on several classes of instances.

- The size of the instances tested in Amini and Barr (1993), with no more than 1500 nodes and 8000 arcs, can no longer be considered relevant; furthermore, the instances were all generated by the Netgen generator, which is known to produce "easy" MCF instances with a "very random" graph structure, and experience has shown that the relative performance of codes on these kind of instances may be very different from that on instances produced by other generators.

- In Amini and Barr (1993), random changes of the data were performed for the sake of the statistical analysis, but the data changes in "real" applications are not necessarily random.

Therefore, we decided to perform a new experimental analysis of the efficiency of re-optimization procedures for "modern" MCF solvers. Due to the need to test much larger instances while keeping the computational times low, we decided to focus on *cost reoptimization*, i.e., the case where only (a subset of) the flow costs c_{ij} are allowed to change from one instance to the following one. Cost reoptimization is required in many applications; see e.g. Amini and Barr (1993), Ahuja et. al. (1993), Frangioni and Gallo (1999), and Gendron et al. (2001), and the references therein; in other classes of applications, different kinds of changes on the data of the problem are required, and we plan to investigate some of these applications in the future.

In the paper, we compare the performance of four different efficient implementations of algorithms for MCF under cost reoptimization in the context of a "real" application, i.e., the solution of multicommodity min-cost flow (MMCF) problems through a decomposition approach. The results clearly show that, for some classes of instances, the relative performances of the codes doing "from-scratch" optimization do not accurately predict the relative performances when reoptimization is used, and therefore confirm the need for a more accurate evaluation of the MCF algorithm to be used if performance is an issue. Incidentally, the experiments also show that the good results obtained in Frangioni and Gallo (1999) for solving MMCF by means of a bundle-based decomposition approach could probably be substantially improved, since the specific solver used there seems to be the worst suited to cost reoptimization among the four tested.

The structure of the paper is the following: in Section 2 we briefly describe the relevant details four MCF solvers tested and of our testing environment; in Section 3 we present and discuss the computational results and, finally, in Section 4 we draw some conclusions and outline some possible future research issues.

2. The MCF Solvers

In the following, we give a brief description of the four MCF algorithms, focusing on the details of the cost-reoptimization phase; for a deeper description of the algorithms see Ahuja et al. (1993) and the original references for each one. We need to introduce some notation.

For a given m-vector x, the surplus $g_i(x)$ of node $i \in N$ w.r.t. x is

$$g_i(x) = b_i - \sum_{j:(i,j)\in A} x_{ij} + \sum_{j:(j,i)\in A} x_{ji} \ ,$$

i.e., the violation of the *flow-conservation constraints* in (1). The surplus of a subset S of nodes w.r.t. x is the sum of the surpluses of the nodes in S, and the total surplus of G (w.r.t.

x) is the sum of the *positive* surpluses of the nodes in G; x satisfies the flow-conservation constraints if and only if the corresponding total surplus of G is zero. Any vector x such that $0 \leq x_{ij} \leq u_{ij}$ for each $(i, j) \in A$ is a *pseudoflow*; a pseudoflow with zero total surplus is a flow. The dual solution of the MCF problem is denoted by π, the vector of *node potentials*. Given a scalar $\xi \geq 0$, a primal-dual pair (x, π) satisfies the ξ-complementary slackness conditions (ξ-CS for short) if x is a pseudoflow and the following hold:

$$x_{ij} < u_{ij} \quad \Rightarrow \quad -\xi \leq c_{ij} - \pi_i + \pi_j \quad \forall (i,j) \in A \; ,$$

$$0 < x_{ij} \quad \Rightarrow \quad c_{ij} - \pi_i + \pi_j \leq \xi \quad \forall (i,j) \in A \; .$$

The number $c^{\pi}_{ij} = c_{ij} - \pi_i + \pi_j$ is the *reduced cost* of the arc (i, j). It is well-known that 0-CS are necessary and sufficient conditions for optimality of a primal-dual pair (x, π) where x is feasible; analogously, a primal-dual pair satisfying ξ-CS is said ξ-optimal.

2.1 The RelaxIV Solver

The `RelaxIV` solver is based on the `Relax` version 4 `FORTRAN` code (Bertsekas and Tseng 1988b, Bertsekas and Tseng 1994). `RelaxIV` implements a primal-dual algorithm, which can be succintly described as follows. At each iteration, a primal-dual pair (x, π), where x is a pseudoflow, which satisfies 0-CS is available; if x is a flow then an optimal solution has been found and the algorithm stops. Otherwise, the algorithm tries to convert x in a feasible flow that still satisfies 0-CS with π by attempting to construct augmenting paths — all made of arcs with zero reduced cost — from one node with positive surplus to one node with negative surplus. This corresponds to running a max-flow algorithm, of the augmenting path type, on the subgraph of G comprising only the arcs with zero reduced costs. If the path is found, the total surplus of the solution is decreased; otherwise a set of nodes with positive or negative surplus is found such that all arcs in the corresponding cutset are either saturated or empty. Hence, the potentials of all the nodes in the set can be increased or decreased without violating 0-CS with the current pseudoflow x, creating new arcs with zero reduced costs (or finding that the problem has no feasible solution). `RelaxIV` implements checks for early termination of the primal phase (the max-flow computation), in order to avoid performing flow operations when it is clear that no feasible flow exists that satisfies 0-CS with the current vector of potentials π.

Cost reoptimization in this code is very easy: given the new cost vector $\bar{c}$ and the optimal primal-dual pair (x, π), x can be turned into a pseudoflow x' satisfying 0-CS with π by just saturating ($x_{ij} = u_{ij}$) all arcs with negative reduced costs and emptying ($x_{ij} = 0$) all arcs with positive reduced costs. Then, the algorithm can be restarted with (x', π).

2.2 The CS2 Solver

The `CS2` solver is based on the `cs2` version 3.7 `C` code, developed by Andrew Goldberg and Boris Cherkassky. `CS2` is based on a cost-scaling push-relabel method (Goldberg and Tarjan 1990, Goldberg 1997), which is a primal-dual approach similar to that of `RelaxIV`, except that a cost-scaling phase allows us to operate on arcs of nonzero reduced cost, and a push-relabel algorithm is used for the max-flow computation instead of an augmenting path one.

The algorithm starts with a "scaling" variable $\xi > 0$, a vector of potentials (e.g., $\pi = 0$) and any pseudoflow. The main loop of the algorithm begins by converting x into a ξ-optimal pseudoflow x'; this is simply done by saturating or emptying every arc that does not satisfy ξ-CS. Then, x' is converted into a ξ-optimal flow by applying a push-relabel algorithm for maximum flow, i.e., a sequence of *push* and *relabel* operations, each of which preserves ξ-optimality, that moves the flow from nodes with positive surplus to nodes with negative surplus until the total surplus is zero. A push operation is applied to a residual arc (i, j) (such that $x_{ij} < u_{ij}$) with negative reduced cost whose tail node i has positive surplus: it consists of pushing $\delta = \min(g_i(x), u_{ij} - x_{ij})$ units of flow from i to j, thereby decreasing $g_i(x)$ by δ and increasing $g_j(x)$ and x_{ij} by δ ("reverse-push" operations are also performed on nonempty arcs with positive reduced cost to decrease their flow: in the code, a "sister-arc" implementation of the graph is used that makes the two operations indistinguishable). A relabel operation is applied to a node i with positive surplus that has no exiting residual arc with negative reduced cost. It consists of changing its potential π_i to the largest possible degree allowed by the $\xi - \mathit{optimality}$ constraints, thereby making some other arc admissible for the push operation (or detecting that the problem has no feasible solution).

When a ξ-optimal primal-dual pair has been obtained, the algorithm is stopped if ξ is small enough; otherwise ξ is decreased (a new scaling phase is started) and the process is repeated.

Cost reoptimization in this code is also very easy: given the new cost vector $\bar{c}$, the algorithm can simply be restarted with the previous primal-dual optimal pair, provided that

ξ is chosen large enough:

$$\xi := -\min\{\bar{c}^{\pi}_{ij} : \ x_{ij} < u_{ij} \ , \ \bar{c}^{\pi}_{ij} < 0, (i,j) \in A\} \ .$$

Due to the "sister-arc" implementation of CS2, this also takes into account nonempty arcs with positive reduced cost.

2.3 The MCFZIB Solver

The MCFZIB solver is based on the mcf version 1.1 C code, developed by Andreas Löbel (Löbel 1996). MCFZIB is a network simplex algorithm, i.e., a specialized version of the simplex algorithm that performs the fundamental simplex operations (computation of the primal and dual basic solutions) directly on the network itself. Although MCFZIB implements both the primal and dual network simplex algorithm, we found the primal network simplex to be consistently more efficient than the dual; also, it is generally accepted (Amini and Barr 1993) that the primal simplex is more suited to cost reoptimization than is the dual simplex because it can more easily exploit the previous optimal base.

At each iteration of the primal simplex algorithm, a primal feasible base (B, L, U) is available; that is, the arc set A is partitioned into the set B, corresponding to the basic variables, that describes a spanning tree for the graph, and the sets L and U, corresponding to the non-basic variables whose values are set to the lower and upper bound, respectively. The corresponding basic primal (feasible) solution x and dual solution π are computed; if the 0-CS are not satisfied, then an arc belonging to either L or U is chosen that violates 0-CS and it is put into B (pivoting phase). This amounts at sending flow along an augmenting cycle with negative reduced cost. The strategy used to select the entering arc has a dramatic influence on the solver efficiency, especially on large networks. In our experiment, the *multiple partial pricing strategy*, where pricing is preferably restricted to a set of candidate arcs that is revised only if necessary, has always been the most efficient one.

Cost reoptimization for a network simplex code depends on whether the primal or dual simplex is to be used. For the primal case, given the previous optimal base (B, L, U) and the new cost vector $\bar{c}$, the dual optimal solution π has to be recomputed with a top-down visit of the tree B; then, all arcs $(i,j) \notin B$ have to be scanned to recompute the reduced costs (meanwhile, the nontrivial data structures for the pricing procedure are updated). Reoptimization for the dual simplex case is more complex because the previous optimal base may no longer be dual-feasible with the new cost vector $\bar{c}$. This can be faced by inserting at most $n - 1$ artificial arcs with very large cost in the network, exactly as it is done in "from-scratch" optimization when a (primal or dual) feasible initial base is not available.

2.4 The MCFCplex Solver

CPLEX (CPLEX Inc. 2002) is a commercial package specifically designed to facilitate the development of applications to solve, modify, and interpret the results of linear, mixed integer, and convex quadratic programming programs. Among the other algorithms, the CPLEX callable libraries offer a primal and dual network simplex implementation for MCF, called NETOPT; the `MCFCplex` solver is nothing but a "wrapper class" that implements the `MCFClass` interface using calls to the CPLEX callable libraries API. The CPLEX NETOPT code appears to be very efficient, and seems to offer full reoptimization capabilities; as with `MCFZIB`, the primal network simplex with the default "multiple-partial-pricing- with-sorting" rule appears to be the most efficient option available.

2.5 The MCFClass Interface

All the above four solvers have been implemented as derived classes of the `MCFClass` class. `MCFClass` is an abstract (pure virtual) `C++` base class that defines the interface between a generic MCF problem solver and the application programs. The interface tackles basically all needs that an application might have, and provides an abstract layer that makes applications independent from the details of the particular solver that is used. The public methods of `MCFClass`, properly redefined in the derived classes, are used for loading the instance, reading the parameters, solving the problems, collecting the results, changing cost/deficits/upper-bound values, and modifying the network structure by adding or deleting arcs or nodes. A set of virtualized data types is used to the largest flexibility in choosing the type (integer or floating-point) and the precision of the numbers (costs, flows, indexes, etc.), making it possible to tailor the code to the specific machine and application. A set of compile-time switches is also provided to allow control on some important features without having to bother about their actual implementation.

We ported the four solvers under the `MCFClass` interface in order to facilitate our own research projects, and in the hope of providing to practitioners a standard and complete interface for developing applications that require the solution of MCF problems without having to bother with the details of the different solvers. Many researchers from all over the world have obtained copies of at least one of the solvers, freely available for research purposes.

While the porting of `RelaxIV` required an almost full rewriting of the original `FORTRAN` code, `CS2` and `MCFZIB` required significantly less effort; however, since the `MCFClass` interface is general with respect to data types, stopping conditions, and types of reoptimizations allowed, a nontrivial porting and validating effort was needed. For instance, for all codes, full reoptimization capabilities were implemented for changes in all the data of the problem (costs, capacities, deficits, and graph topology), which were not present in the original codes.

In all our tests, the `C` and `C++` versions have behaved almost identically, with only a few percent maximum performance difference; the `C++` implementations were even slightly faster than the `C` implementations in many cases, despite sometimes requiring some extra effort to conform to the `MCFClass` interface. For `CS2`, for instance, a data structure is kept updated that allows us to recover the optimal flow solution in the same order as the original arc set, which is not done in the original `cs2 C` code. Also, for all codes, both integer and floating-point data are independently supported for both flows (capacities, deficits) and potentials (costs), which was not the case for all the original solvers.

The codes are currently available at

```
http://www.di.unipi.it/di/groups/optimize/Software/MCF.html
```

together with a fifth solver, `SPTree`, that implements the `MCFClass` interface for the special case of MCF problems that can be solved using only one shortest-path tree computation.

3. Computational Experiments

In order to compare the performances of the four solvers under cost reoptimization in a realistic application, we used the bundle-based cost-decomposition algorithm of Frangioni and Gallo (1999) to solve four different families of large-scale MMCF problems. The (linear) multicommodity min-cost flow problem is the following linear program:

$$\begin{array}{lll} \min & \sum_{h\in K}\sum_{(i,j)\in A} c_{ij}^h x_{ij}^h & \\ & \sum_{j:(i,j)\in A} x_{ij}^h - \sum_{j:(j,i)\in A} x_{ji}^h = b_i^h & \forall i \in N, \forall h \in K \\ & 0 \le x_{ij}^h \le u_{ij}^h & \forall (i,j) \in A, \forall h \in K \\ & \sum_{h\in K} x_{ij}^h \le u_{ij} & \forall (i,j) \in A \end{array} \tag{2}$$

That is, on the same directed network $G = (N, A)$, a set K of different "kinds" of flow (commodities) co-exist. The commodities don't "mix," but they compete for the shared resource of arc *mutual* capacities u_{ij}. It is beyond the scope of this paper to provide a description of the algorithm, for which we direct the interested reader to the original paper; suffice it here to say that, like many other cost-decomposition approaches, the algorithm in Frangioni and Gallo (1999) relaxes the *mutual capacity constraints* and solves, at each iteration, $k = |K|$ independent MCF problems, one for each commodity of the original MMCF problem. The optimal solutions of the MCF problems are then used to modify the cost vectors of the subsequent iteration. The optimization process can require several hundreds of iterations, depending on the problem size and difficulty, and, qualitatively speaking, the costs in the final iterations "change less" than in the initial iterations, so that reoptimization has a different impact in different stages of the algorithm. It may be worth remarking that, in our implementation, k separate MCF solver objects are built, one for each commodity of the MMCF instance. Since each solver has its own independent internal status variables, it is possible (indeed, automatic) to exploit the optimal solution of the previous call for any commodity h when solving the MCF again corresponding to that commodity, even though other $k-1$ flow problems have been solved in between. This would have not been possible if the original `C` or `FORTRAN` codes had been used, since they all used global variables. Note that it could even be possible to use different MCF solvers for different commodities in the same MMCF instance; this is in fact allowed in our code, although up to now only in a limited way (see Frangioni and Gallo 1999 for more details).

In the rest of this section we will describe the instances used, discuss some important details of the test setup, and finally report and discuss the results of our experiments.

3.1 The Test Instances

We generated four classes of test instances. The first three classes were generated using the `dimacs2pprn` "meta" generator, first introduced in Castro and Nabona (1996) and afterwards used in other papers such as Frangioni and Gallo (1999). `dimacs2pprn` is dubbed a "meta" generator because the basic characteristics of the MMCF instances it produces (topology of the network, distribution of costs/capacities/deficits) are not hard-wired in the generator code, but rather are taken as input. More specifically, the generator inputs the description of a single-commodity MCF in DIMACS standard format, and three parameters k, r, and f. The generator produces an MMCF instance with k commodities on the same topological graph. The deficits and capacities of each individual commodity are obtained by scaling those of the original MCF by a pseudo-random number drawn uniformly in the interval $[1, r]$, while each arc costs c^h_{ij} for commodity h is independently and uniformly drawn at random in $[0, c_{ij}]$. Finally, the mutual capacity for each arc is initially fixed to fu_{ij}, and subsequently adjusted to ensure that a feasible multicommodity flow exists. Note that, due to the scaling, all data of the MMCF instance are in general not integer even if the data of the original MCF instance are. The `dimacs2pprn` generator allows us to produce instances with integral capacities/surpluses or costs; however, the costs would rapidly become fractional anyway as the cost-decomposition approach proceeds. It may be worth remarking again that the original `cs2` and `Relax` solvers work only with integral data, while our versions have worked perfectly with non-integral data. Of course, working with non-integral data types requires setting some tolerances on both optimality and feasibility conditions; we have set them both to very high values (at least `1e-8` relative) in order to match the level of accuracy natively provided by the `MCFCplex` solver.

We used three different, well-known random generators for producing the initial MCF instances: `GOTO`, `NETGEN` and `GRIDGEN`. For each generator "GEN," we produced one instance "GEN_q_p" for some values of $q \in \{8, 9, 10, 12, 14\}$ and $p \in \{8, 16, 32, 64\}$, such that the instance has 2^q nodes and density p, i.e., (roughly) $p2^q$ arcs. For each generator we selected only a subset of the possible p and q values, both because of the structure of the generator and to keep the overall running time at an acceptable level. Note that the largest MMCF instances that we tested correspond to linear programs with over 2 million variables and 200000 constraints.

For each of the original MCF instances we generated one MMCF instance with $k = 10$, $r = 1$, and $f = 4$. Since each MMCF instance contains ten different MCF instances that are independently solved, dividing the total running time required by all the MCF computations for one MMCF instance by ten, we obtain the average computational cost among ten different MCF instances. Finally, we also used a set of well-known MMCF instances, the PDS (patient distribution system) problems; these instances originate by transportation problem of patients from a place of military conflict. The PDS instances all have $k = 11$ (and therefore each one provides average results for 11 different MCFs), and differ for one parameter t indicating the number of days covered by the model: each instance PDSt has roughly $126t$ nodes and a density between three and four (growing slowly as t grows).

All the above generators, both MCF and MMCF, and instances are available at

`http://www.di.unipi.it/di/groups/optimize/Data/MMCF.html`.

3.2 Test Setup

We tested our instances on a PC with an Intel Pentium 4 CPU at 1.70 GHz, running RedHat Linux version 7.2. The code was compiled using the GNU `g++` compiler version 2.96, using aggressive optimization option "-O3." We used CPLEX version 7.5. As discussed in Sections 2.3 and 2.4, all the results for both simplex-based codes were obtained using the primal simplex algorithm with a multiple partial pricing rule.

While setting up the tests we were confronted with two problems: how to test all the solvers on exactly the same instances, and how to measure the running time exactly.

The first problem arises because, in the dual-ascent approach of Frangioni and Gallo (1999), the flow solutions of each MCF at a given iteration are part of the data of an optimization problem whose solution gives the flow costs at the subsequent iteration. Different MCF solvers may provide different (ξ-)optimal solutions of the same instance, and this is very likely to happen during the course of the dual-ascent algorithm. Hence, the sequences of the flow costs generated during two runs of the algorithm on the same MMCF instance, but with two different MCF solvers are very likely to be different. Furthermore, dual-ascent methods are known to be very "unstable" in that the trajectory followed by the methods can vary significantly with only tiny variations in the results; hence, the sequences of arc costs corresponding to two different MCF solvers cannot be expected to be even "close." Thus, ensuring that each solver was required to solve exactly the same sequence of MCF problem was not straightforward.

The second problem arises because the MCF solvers are used within a complex application, which uses several different methods of the `MCFClass` interface to load and modify the MCF problems, solve them, and collect the results. Some of these methods are likely to terminate too quickly to be reliably timed with the standard timing procedures, especially on small MCF instances, while possibly having a non-negligible impact on the overall running time. Thus, it was not straightforward to ensure that the total running time spent by each MCF solver was correctly measured.

Fortunately, the object-oriented design of `MCFClass` provided us with an easy and effective way to solve both problems at once, in the form of a "tester class." That is, we could easily implement an `MCFTest` class, derived from `MCFClass`, that just holds two pointers to two different MCF solvers, the "master solver" and the "slave solver." For each call to any method of `MCFClass`, the `MCFTest` class calls the corresponding method of both solvers, but, for methods that output results, the output of the "slave" solver is discarded (usually, a tester class would perform cross checks to ensure that the output of the two solvers is correct, but this was not our aim here).

Our experiments were therefore performed as follows: for each class of instances we chose a solver among the four available ones, and then solved each instance in the class five times. The first time we just used that particular solver as the MCF solver. For the other four runs we used as MCF solver the four possible `MCFTest` objects obtained by chosing as "master solver" the selected one, and as "slave solver" one of the four MCF solvers, comprising the one that was being used as "master." Finally, for each instance we subtracted the recorded running time for the run with just one solver from all the recorded running times for all the other runs corresponding to the same instance; note that we never counted the time for creating the object and loading the instance. In this way, we could be sure that the sequence of MCF problems solved by each different MCF solver was exactly the same, and we obtained a very precise estimate of the time that was spent in the "slave solver" alone (the overhead caused by the `MCFTest` layer being negligible in this case).

3.3 Computational Results

Preliminary results clearly showed that reoptimization usually gave, for the same solver, better results than "from-scratch" optimization. Thus, we avoided testing the solvers without reoptimization. Also, we found that the time required to solve the first MCF was often close to the average time required to solve any other MCF — of the same commodity — in the sequence if cost reoptimization was not used, i.e., that the costs generated at all steps of the dual-ascent approach did not substantially change the "difficulty" of solving the MCF with respect to the initial ones. Therefore, for each class of instances, we decided to report the following three data:

- T_1, the (average) time required to solve the first MCF;
- T_{tot}, the (average) time required to solve all the MCFs;
- The *reoptimization index*

$$RI = \frac{T_{tot} - T_1}{T_1(\text{ number of iterations } -1 \text{ })},$$

Table 1: Results for PDS Instances

	MCFCplex			MCFZIB			RelaxIV			CS2		
network	T_1	T_{tot}	RI	T_1	T_{tot}	RI	T_1	T_{tot}	RI	T_1	T_{tot}	RI
pds10	0.12	24.93	0.41	0.34	23.67	0.14	0.07	60.01	1.70	0.66	116.16	0.35
pds11	0.16	27.09	0.34	0.27	33.85	0.25	0.14	104.93	1.50	0.79	190.01	0.48
pds12	0.19	28.86	0.30	0.54	50.63	0.20	0.11	60.28	1.10	0.81	220.89	0.54
pds13	0.24	45.03	0.37	0.43	53.54	0.25	0.14	167.65	2.40	1.09	273.58	0.50
pds14	0.24	38.59	0.27	0.79	69.57	0.17	0.18	180.86	2.00	1.18	297.24	0.50
pds15	0.37	49.98	0.27	0.61	56.65	0.18	0.21	251.37	2.40	1.43	342.24	0.47
pds18	0.57	81.64	0.28	1.41	114.70	0.16	0.32	491.95	3.07	1.84	786.63	0.85
pds30	1.93	214.18	0.22	3.40	323.61	0.18	1.92	2082.43	2.10	4.28	2060.86	0.96
pds40	5.82	1973.25	0.67	7.18	2449.66	0.68	5.75	4063.55	1.40	9.34	6690.75	1.43

i.e., the ratio between the average running time for all iterations after the first one (where reoptimization is used), and the running time for the first iteration (where "from-scratch" optimization is used).

The reoptimization index can be taken as an indication of how efficiently each MCF solver reoptimizes, i.e., of the relative importance of using reoptimization — as opposed as not using it — for that code. Of course, the most interesting datum from a practitioner's viewpoint is T_{tot}, in that it directly measures the impact of using each solver in the application.

Table 1 reports results obtained for PDS instances. For these instances, `RelaxIV` is the fastest code "from-scratch," closely followed by `MCFCplex`; however, the latter reoptimizes much more efficiently, as testified by its much smaller T_{tot} times. `MCFZIB` obtains even better RI values than does `MCFCplex`, but it is considerably slower "from scratch", so that it ends up being outperformed, albeit not too substantially, by `MCFCplex`. The same happens for `CS2` and `RelaxIV`: while the former obtains better RI values than the latter, it is slower in "from-scratch" optimization and it ends up being outperformed. Remarkably, the RI values obtained for pds40 are much worse than those obtained on all the other instances for `MCFCplex`, `MCFZIB` and `CS2`, but not so for `RelaxIV`. Also, note that `RelaxIV` often obtains RI values (significantly) larger than one. This probably depends on the special structure of the underlying network in PDS problems. This is a time-space network (with ten to 40 time periods) that contains very "long" paths, that are well-known to affect the performances of `RelaxIV` adversely. These paths contain "return" arcs that are initially "inactive" (do not carry flow), but may become "active" later in the optimization process. Thus, the MCF problems solved at the first iteration of the cost-decomposition method are much easier for `RelaxIV` than some of those solved at later stages, and the difference is large enough to offset any advantage due to reoptimization.

Table 2 reports results obtained for `GOTO` instances. The two simplex-based codes reop-

Table 2: Results using GOTO Instances

	`MCFCplex`			`MCFZIB`			`RelaxIV`			`CS2`		
network	T_1	T_{tot}	RI	T_1	T_{tot}	RI	T_1	T_{tot}	RI	T_1	T_{tot}	RI
GOTO8_8	0.33	5.69	0.03	0.52	12.04	0.04	42.42	54.93	0.02	0.45	122.86	0.54
GOTO8_16	0.63	13.22	0.04	0.85	28.26	0.06	28.21	211.60	0.01	0.75	122.86	0.33
GOTO8_32	1.40	22.11	0.03	1.35	42.85	0.04	62.87	1576.20	0.05	0.45	122.86	0.55
GOTO9_16	2.45	61.97	0.04	3.73	113.69	0.05	263.50	2007.53	0.01	2.41	713.94	0.59
GOTO9_32	5.44	117.61	0.04	9.59	217.24	0.04	2002.56	2546.20	0.0005	9.50	686.61	0.14
GOTO9_64	10.10	241.11	0.04	14.22	438.90	0.05	2078.13	3883.01	0.001	6.08	863.70	0.28
GOTO10_8	5.13	67.33	0.02	8.22	122.00	0.02	58.75	788.37	0.02	3.60	734.35	0.40
GOTO10_16	15.48	124.01	0.01	18.64	1193.31	0.13	656.40	2249.69	0.004	11.35	209.66	0.04
GOTO10_32	17.81	696.38	0.07	22.50	900.31	0.08	1700.00	4923.19	0.002	19.29	1994.66	0.21
GOTO12_8	99.54	3462.90	0.06	142.33	4544.30	0.06	—	—	—	28.81	8814.16	0.60
GOTO12_16	321.50	3305.70	0.02	842.50	4995.90	0.01	—	—	—	45.79	14276.00	0.60

Table 3: Results using NETGEN Instances

	`MCFCplex`			`MCFZIB`			`RelaxIV`			`CS2`		
network	T_1	T_{tot}	RI	T_1	T_{tot}	RI	T_1	T_{tot}	RI	T_1	T_{tot}	RI
NET8_8	0.06	4.14	0.13	0.05	4.78	0.19	0.08	13.46	0.33	0.12	33.29	0.55
NET8_16	0.11	7.58	0.13	0.13	12.11	0.18	0.17	28.76	0.33	0.26	43.41	0.33
NET8_32	0.28	22.66	0.16	0.38	34.05	0.17	0.37	98.36	0.53	0.47	275.88	1.17
NET8_64	0.24	20.57	0.16	0.35	49.71	0.28	0.50	126.42	0.50	0.79	251.09	0.63
NET10_8	0.49	4.12	0.01	0.46	4.79	0.02	0.64	13.47	0.04	0.93	68.87	0.14
NET10_16	2.09	210.95	0.2	1.86	308.4	0.33	2.05	656.96	0.64	2.03	945.84	0.93
NET10_32	1.70	80.10	0.09	1.76	155.37	0.17	4.46	421.43	0.19	2.88	656.69	0.45
NET10_64	7.33	418.53	0.11	10.15	484.22	0.16	11.64	3133.25	0.53	6.36	3297.25	1.03
NET12_8	14.30	308.71	0.04	7.78	402.76	0.10	11.30	1158.64	0.20	6.89	2260.97	0.65
NET12_16	22.16	560.29	0.24	18.59	622.61	0.32	21.15	1308.41	0.60	16.80	1514.81	0.89
NET14_8	627.23	6217.36	0.09	231.08	5737.99	0.24	232.90	8472.30	0.35	167.28	8435.76	0.49
NET14_16	897.22	10128.60	0.10	359.77	9547.49	0.26	307.85	14223.40	0.45	243.15	15080.50	0.61

timize very efficiently for these instances, obtaining RI values always smaller than 0.08 and as small as 0.01 in one case. Once again, `MCFCplex` is faster in “from-scratch” optimization, and, as the RI values are comparable, is also faster when the total running time is considered. `CS2` is often faster than `MCFCplex` in “from-scratch” optimization, up to almost an order of magnitude in the largest instances, but it reoptimizes far less efficiently and the total running time is consistently larger than that of `MCFCplex`. Finally, the results obtained by `RelaxIV` appear to be very erratic — the largest instances could not even be solved in reasonable time — and no clear conclusion can be reached.

Table 3 reports results obtained for `NETGEN` instances. The results are similar to those of the `GOTO` instances, with the simplex-based codes being faster when the total running time is considered due to more efficient reoptimization, although being clearly outperformed by the primal-dual codes in “from-scratch” optimization when the sizes of the MCF instances grow. For these instances `MCFCplex` is most often faster than `MCFZIB` on small instances, while the opposite is true for large instances due to much better “from-scratch” optimization times. `RelaxIV` is usually slightly faster than `CS2`, but not than the simplex-based codes, when the total running time is considered, although `CS2` is the fastest code among the four in “from-scratch” optimization for the largest instances.

Finally, Table 4 reports results obtained for `GRIDGEN` instances. For these instances `MCFZIB` is always faster than `MCFCplex` in “from-scratch” optimization, but the latter reop-

Table 4: Results using GRIDGEN Instances

	`MCFCplex`			`MCFZIB`			`RelaxIV`			`CS2`		
network	T_1	T_{tot}	RI	T_1	T_{tot}	RI	T_1	T_{tot}	RI	T_1	T_{tot}	RI
GRID8_8	0.07	5.37	0.15	0.06	11.45	0.38	0.08	16.31	0.40	0.11	45.38	0.82
GRID8_16	0.11	8.23	0.15	0.11	11.45	0.21	0.15	31.20	0.41	0.26	111.27	0.63
GRID8_32	0.38	25.76	0.13	0.38	36.16	0.19	0.54	11.97	0.43	0.57	224.49	0.78
GRID8_64	1.35	58.19	0.10	1.35	142.16	0.20	1.87	517.72	0.55	1.20	554.46	0.92
GRID10_8	0.55	21.06	0.07	0.38	26.90	0.14	0.65	104.54	0.32	0.97	223.26	0.46
GRID10_16	2.77	196.36	0.14	2.36	266.20	0.22	2.79	790.69	0.56	2.50	1004.79	0.80
GRID10_32	3.79	134.80	0.07	2.96	303.16	0.20	7.94	1743.85	0.44	3.87	1349.83	0.70
GRID10_64	10.57	538.24	0.10	8.45	991.11	0.23	7.50	3660.42	0.97	18.96	5292.58	0.55
GRID12_8	16.29	5236.34	0.05	9.58	346.50	0.35	18.22	4771.50	0.52	14.50	6477.5	0.90
GRID12_16	70.91	1587.48	0.20	32.29	1672.21	0.50	43.54	2678.49	0.60	20.48	2375.70	1.15
GRID12_32	111.22	6254.10	0.55	55.79	8211.50	1.40	96.01	6689.50	0.68	33.90	3570.50	1.04
GRID12_64	275.90	8043.00	0.28	119.88	11683.00	0.96	144.36	13850.00	0.94	57.42	4943.00	0.85

timizes more efficiently, and it is always faster when the total running time is considered. `RelaxIV` is most often worse, sometimes considerably, than the simplex-based codes, and so is `CS2`, both "from scratch" and in reoptimization, for the smaller instances; however, for the largest instances `CS2` is much faster "from-scratch" than all the other codes, and, despite having somewhat large RI values, it manages to keep the lead even when the total running time is considered, solving the largest instances about a factor of two faster than the fastest among the other codes. It is particularly interesting to contrast `MCFCplex` and `CS2` on the GRID12_64 instances: while the former reoptimizes much more efficiently (but still not nearly as efficiently as, for instance, in the `GOTO` instances), the latter is more than five times faster in "from scratch" optimization, thus overbalancing the factor of roughly three between the respective RI values.

4. Conclusions

We have experimented to compare the relative efficiency of four MCF codes under cost reoptimization in the context of a "realistic" application, that is, the solution of MMCF problems with a cost-decomposition algorithm. We were able to test different classes of MCF instances of varying size, showing how the running time for "from scratch" optimization of an instance may not always be a good guide for selecting the algorithm to be used in a cost-reoptimization setting. As a general guideline, our findings confirm those of Amini and Barr (1993), i.e., codes implementing the primal simplex algorithm tend to be more efficient at reoptimizing after a change of costs than those based on a primal-dual approach, and this often overbalances any advantage in "from-scratch" optimization that primal-dual codes may have, especially if the size of the instance is not very large. However, for some large instances, primal-dual codes may become competitive even when cost reoptimization is taken into account. It should also be remarked, as correctly pointed out by one referee, that there may be scope for improving the rules for reoptimizing in `RelaxIV` and `CS2`, which may result in better reoptimization performance especially if only relatively few arc costs change or most arc costs change of only a relatively small amount. In our opinion, one interesting finding of the experiments is that, although some general indications can be drawn, no algorithm is always the most efficient: the best choice for the MCF solver depends on both the class and on the size of the instance. Hence, our experience shows that being able to test several different MCF solvers easily and efficiently within a complex optimization code for the solution of a network-structured problem may be very important if an overall efficient approach has to be obtained. This was precisely the rationale under our proposal of the `MCFClass` project for a standard interface for MCF problem solvers. As previously discussed, having the four MCF solvers ported under the `MCFClass` interface was instrumental for being able to perform the comparisons quickly and fairly. Therefore, we believe that more effort should be undertaken to provide standard optimization components that can be easily exchanged and used for construction of sophisticated approaches. This is also confirmed by the existence of the similar — but with a much broader-range — `OSI` project for a standard interface for mixed-integer linear and quadratic problems solvers, that is gaining momentum in the community. The `OSI` project is available at

`http://oss.software.ibm.com/developerworks/opensource/coin`.

Although with a much more limited scope, we hope that the `MCFClass` project can provide a valuable tool for researchers and practitioners alike.

We plan to run similar comparisons in other kinds of reoptimization settings, e.g., capacity reoptimization or deficit reoptimization. With the better understanding of the reoptimization phenomenon thus acquired, it could become possible to design and implement MCF algorithms particularly well-suited for one of the possible different types of reoptimization. Finally, we plan to bring forward the `MCFClass` project, developing or porting other MCF codes, and to design standard interfaces for other common network problems, like the shortest-path problem. We believe that a growing set of standard optimization components that can be easily exchanged and used for constructing sophisticated approaches for difficult optimization problems would be useful for researchers and practitioners.

Acknowledgments

This work would not have been possible without the fundamental contribution of D.P. Bertsekas, A. Goldberg, and A. Löbel, who developed the original versions of the `RelaxIV`, `CS2`, and `MCFZIB` solvers, respectively, made them available to the academic community, and especially allowed us to modify and redistribute them within the `MCFClass` project.

This research has been partly funded by project CNR 5.12.6.2 CTB 02 "SORSA - Simulazione e Ottimizzazione per Reti: Software e Applicazioni" of the Italian Ministry of Education, University and Research.

References

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

Amini, M.M., R.S. Barr. 1993. Network reoptimization algorithms: a statistically designed comparison, *ORSA J. Comput.* **5** 395–409.

Bertsekas, D.P. 1991 *Linear Network Optimization: Algorithms and Codes.* MIT Press, Cambridge, MA.

Bertsekas, D.P. 1998. *Network Optimization: Continuous and Discrete Models.* Athena Scientific, Belmont, MA.

Bertsekas, D.P., J. Eckstein. 1988. Dual coordinate step methods for linear network flow problems. *Math. Prog.* **42** 203–243.

Bertsekas, D.P., P. Tseng. 1988a. Relaxation methods for minimum cost ordinary and generalized network flow problems. *Oper. Res.* **36** 93–114.

Bertsekas, D.P., P. Tseng, 1988b. RELAX: a computer code for minimum cost network flow problems. *Annals of Oper. Res.* **13** 127–190.

Bertsekas, D.P., P. Tseng. 1994. 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, Massachusetts Institute of Technology, Cambridge, MA.

Castro, J., N. Nabona. 1996. An implementation of linear and nonlinear multicommodity network flows. *European J. Oper. Res.* **92** 37–53.

CPLEX, Inc. 2002. Using the CPLEX callable library. Incline Village, NV.

Frangioni, A., G. Gallo. 1999. A bundle type dual-ascent approach to linear multicommodity min cost flow problems. *INFORMS J. Comput.* **11** 370–393.

Gendron, B., T.G. Crainic, A. Frangioni. 2001. Bundle-based relaxation methods for multicommodity capacitated fixed charge network design problems. *Discrete Appl. Math.* **112** 73–99.

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

Goldberg, A.V., R. Tarjan. 1990. Finding minimum cost circulation by successive approximation. *Math. of Oper. Res.* **15** 430–466.

Helgason, R.V., J.L. Kennington. 1995. Primal simplex algorithms for minimum cost network. M.O. Ball, T.L .Magnanti, C.L. Monma, G.L. Nemhauser, eds. *Handbooks in Operations Research and Management Science*, Volume 7, North-Holland, Amsterdam, 85–113.

Löbel, A. 1996. Solving large-scale real-world minimum-cost flow problems by a network simplex method. *Technical Report* **SC 96-7**, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Germany.

Löbel, A. 1999. Vehicle scheduling in public transit and lagrangean pricing. *Management Sci.* **44** 1637–1649.

Resende, M.G.C., P.M. Pardalos. 1996. Interior point algorithms for network flow problems. J.E. Beasley, ed. *Advances in linear and integer programming.* Oxford University Press, Oxford, UK. 147–187.