Technical Report: TR-96-01, December 15, 1996

A Bundle type Dual-ascent Approach to Linear Multi-Commodity Min Cost Flow Problems

Frangioni Antonio Gallo Giorgio

Dipartimento di Informatica, Università di Pisa
corso Italia 40, 56125 Pisa, Italy

Abstract

We present a cost decomposition approach to Linear Multicommodity Min Cost Flow problems, based on dualizing the mutual capacity constraints: the resulting Lagrangean Dual is solved by means of a new, specialized Bundle type-dual ascent algorithm. Although decomposition approaches are generally believed not to be competitive, even for the solution of large-scale network structured problems, we present evidence based on extensive computational comparisons that a careful implementation of a decomposition algorithm can outperform several other approaches, especially on problems where the number of commodities is "large" w.r.t. the size of the graph. Our specialized Bundle algorithm is characterised by a new heuristic for the trust region parameter handling, and embeds a custom, fast Quadratic Programming solver that permits the implementation of a Lagrangean variables generation strategy. The Lagrangean Relaxation solver is capable of exploiting the structural properties of the single-commodity Min Cost Flow subproblems to avoid using a "generic" MCF solver whenever it is possible. The proposed approach can be easily extended to handle extra constraints or variants such as the Nonsimultaneous Multicommodity problem. In our computational experience, we also studied the impact on the relative efficiencies of the different approaches tested of some characteristics, such as the number of commodities w.r.t. the total size of the problem.

ACM–CR Subject Classification:

Keyword and phrases: Multicommodity Flows, Cost Decomposition, Bundle methods

0. Introduction

The Multicommodity Min Cost Flow problem (MMCF) is a generalization of the ordinary single-commodity Min Cost Flow problem, in which flows of different nature (*commodities*) must be shipped at minimal cost on a network, competing for the resources represented by the arc capacities.
The interest of MMCF is twofold: from a practical point of wiev, the Linear MMCF (to which we will restrict our attention) and its varaints are (sub)models of a wide variety of hard transportation and scheduling problems [GG71] [PS83] [ZTT84] [AKS88] [AKL93] [Ba93] [BS93] [GC95], where often very large instances of the "simple" linear problem have to be repeatedly solved in order to approach the solution of the integer version; although MMCF is in fact just a structured Linear Program, the usual LP techniques fail to be efficient enough on practical problems, and several specialized algorithms have been proposed for its solution during the last two decades. Moreover, MMCF seems to have a particular role on the "edge" between hard and simple graph problems: in fact, some variants of MMCF seem to be intimately related with approximation algorithms for several hard graph problems of both theoretical and practical interest [BL84] [KAR90] while ε-approximation combinatorial approaches have been developed for the problem [GVY93] [PT93] [Ta94], making MMCF one of few LPs for which an approximation algorithm of practical interest is known [LPS93].
In this work, we present some computational experience with a cost decomposition approach to large-scale MMCF problems: the idea of applying decomposition methods to MMCF is not new (it seems that the original Dantzig-Wolfe algorithm had been motivated by this problem), and several proposals for the implementation of the "coordination phase" [FPL93] [LGM93] [Me94] [PZ94] [DP95] can be found in the literature. Our aim was to explore the overall efficiency of a cost decomposition approach based on solution of the Lagrangean Dual (w.r.t. the mutual capacity constraints) by a Nondifferentiable Optimization (NDO) algorithm belonging to the class of the "Bundle methods" [Ki89] [Le89] [SZ92] [HL93] [CFN95]: to the best of our knowledge, only one attempt [Me94] had already been done to use Bundle methods in this contest, and just making use of pre-existent NDO software. Rather, as a part of the development of a new Bundle approach tailored for maximization of polyhedral functions [CFN95], we devised a specialized Bundle-like code for MMCF, that has shown to be very effective: the main characteristics of our implementation are

- a simple "main" logic, based on a new heuristic for the trust region parameter handling
- use of an efficient Quadratic Programming solver [Fr95], that allows a new Lagrangean variables generation strategy
- a Lagrangean Relaxation solver that is capable of exploiting the structural properties of the single-commodity Min Cost Flow subproblems to avoid using a "generic" MCF solver whenever it is possible

The structure of the work is the following: in §1 the MMCF problem is introduced, and some of its characteristics discussed, while in §2 Bundle methods are brief reviewed, and §3 describes our particular Bundle implementation in detail. The following paragraphs describes the extensive set of computational experiences performed with the code: in §4 the approaches that we tested against our implementation are described, and the relevant optimization parameters of each code are discussed, in

§5 the different sets of test problems (of various size and nature) are introduced, and finally in §6 the computational results are presented and conclusions are drawn.

1. Problem formulation

We consider the basic linear Multicommodity Min Cost Flow Problem (MMCF): given a directed graph $G(N, E)$, where $|N| = n$ and $|E| = m$, we want to route a set of k commodities on G at minimal total cost. Flow x_{ij}^h of the h-th commodity on arc (i, j) has individual lower and upper bounds (respectively 0 and u_{ij}^h) and a linear cost function $c_{ij}^h x_{ij}^h$, while *mutual* (or *aggregate) capacity* constraints $\Sigma_h x_{ij}^h \le u_{ij}$ limit the total quantity of flow on arc (i, j), regardless of the commodity.
A formulation of the problem is

$$\text{(MMCF)} \qquad \begin{cases} \min \Sigma_h \Sigma_{ij} c_{ij}^h x_{ij}^h & \\ \Sigma_j x_{ij}^h - \Sigma_j x_{ji}^h = b_i^h & \forall\, i, h \quad \text{(a)} \\ 0 \le x_{ij}^h \le u_{ij}^h & \forall\, i, j, h \quad \text{(b)} \\ \Sigma_h x_{ij}^h \le u_{ij} & \forall\, i, j \quad \text{(c)} \end{cases}$$

where (a) are the (usual) independent flow conservation constraints for each commodity and (b), (c) are respectively the "individual" and mutual capacity constraints: when restated in matrix notation, using the node-arc incidence matrix $\boldsymbol{E}$ of G, this (node-arc) formulation highlight the block-structured nature of MMCF

$$\text{(MMCF)} \qquad \begin{cases} \min \Sigma_h \boldsymbol{c}^h \boldsymbol{x}^h & \\ \begin{bmatrix} \boldsymbol{E} & \cdots & \boldsymbol{0} \\ \vdots & \ddots & \vdots \\ \boldsymbol{0} & \cdots & \boldsymbol{E} \\ \boldsymbol{I} & \cdots & \boldsymbol{I} \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{x}^1 \\ \vdots \\ \boldsymbol{x}^k \end{bmatrix} = \begin{bmatrix} \boldsymbol{b}^1 \\ \vdots \\ \boldsymbol{b}^k \\ \boldsymbol{u} \end{bmatrix} & \le \\ 0 \le \boldsymbol{x}^h \le \boldsymbol{u}^h & \forall h \end{cases}$$

A special case of MMCF is the one in which no single-commodity upper bounds exist (i.e. $u_{ij}^h = +\infty$ $\forall i, j, k$): in this case, the k independent MCF subproblems that are obtained relaxing the mutual capacity constraints (c) can become either Origin-Destination Shortest Path or Shortest Path Tree problems, depending on the structure of the demand vectors (one source and one sink or one source and many sinks respectively). From the point of view of number k of different commodities, real life MMCFs range from the case in which the commodities are just a few (as in many distribution problems) to the case in which k approaches the number of all the possible O/D pairs (as in traffic circulation models): of course, the capability of exploiting the special structure of the problem to be solved is crucial to the design of efficient algorithms. Interesting generalisations of MMCF are the Generalised Multicommodity (MCF) Problem, where mutual capacity constraints (c) are replaced by a block of general linear constraints $\boldsymbol{A}[\boldsymbol{x}^1 \ \ldots \ \boldsymbol{x}^k] = \boldsymbol{d}$, the Nonsimultaneous Multicommodity (MCF) Problem, where the commodities are partitioned in p "blocks" $B(1) \ \ldots \ B(p)$ and mutual capacity constraints are "restricted" within a block, i.e.

$$\sum_{h \in B(q)} x_{ij}^h \le u_{ij}^q \qquad q = 1 \ldots p$$

and the Equal Flow Problem, where there is just one commodity but constraints $x_a = x_b$ are imposed on certain pairs of the arcs.

Several algorithmic approaches have been proposed in the past for MMCF, as the surveys [As78] [Ke78] Chap. 17 of [AMO93] show: among them, we just cite *column generation* approaches [CD74] [Po92] [BHJ95], *resource directive decomposition* [GG71] [KS77], *primal partitioning* methods [FPL93] [CN95] and *specialized interior point* algorithms [SM91] [KKR94]; our approach belong to the class of *cost decomposition* approaches [LGM93] [Me94] [PZ94] [DP95], more or less directly related (see [CFN95]) with the well-known Dantzig-Wolfe decomposition method [FPL93].
To address the solution of MMCF, *cost decomposition* methods consider its Lagrangean Relaxation w.r.t. the "complicating" constraints (c), i.e.

$$(\mathrm{RMG}_\lambda) \qquad \min\left\{ \Sigma_h \Sigma_{ij} c_{ij}^h x_{ij}^h + \Sigma_{ij} \lambda_{ij} (\Sigma_h x_{ij}^h - u_{ij}) : \boldsymbol{E}\boldsymbol{x}^h = \boldsymbol{b}^h , \mathbf{0} \le \boldsymbol{x}^h \le \boldsymbol{u}^h \ \forall h \right\}$$

and the corresponding Lagrangean Dual

$$(\mathrm{DMMCF}) \qquad \max_{\lambda \ge \mathbf{0}} \left\{ \min\left\{ \Sigma_h (\boldsymbol{c}^h + \lambda)\boldsymbol{x}^h - \lambda \boldsymbol{u} : \boldsymbol{E}\boldsymbol{x}^h = \boldsymbol{b}^h , \mathbf{0} \le \boldsymbol{x}^h \le \boldsymbol{u}^h \ \forall h \right\} \right\}$$

that requires the (constrained) maximization of the polyhedral (concave nondifferentiable) function

$$\varphi(\lambda) = \Sigma_h \ \min\left\{ (\boldsymbol{c}^h + \lambda)\boldsymbol{x}^h : \boldsymbol{E}\boldsymbol{x}^h = \boldsymbol{b}^h , \mathbf{0} \le \boldsymbol{x}^h \le \boldsymbol{u}^h \right\} - \lambda \boldsymbol{u}$$

The advantage of this approach is that, for any fixed vector of dual multipliers λ, the calculation of $\varphi(\lambda)$ can be performed via the solution of k independent (single-commodity) MCFs, for which several efficient algorithms exist: this is especially true in the case of no single-commodity upper bounds, where the computation of $\varphi(\lambda)$ may just require the solution of k Shortest Path subproblems w.r.t. the modified costs $(\boldsymbol{c}^h + \lambda)$. Obviously, the drawback of this approach is that maximization of the nondifferentiable function $\varphi()$ is required, and here's where the various approaches differs: in a first class of methods, such as *smooth Penalty Function* [PZ94] and *Augmented Lagrangean* algorithms [DP95], $\varphi()$ is approximated with a "more tractable" (smooth) function that is then maximized with some appropriate method. Alternatively, NDO algorithms as the *Subgradient* method (used in most application-specific MMCF codes), the *Cutting Plane* approach (that is the dual form of the DW decomposition) [FPL93] and various form of *Bundle* methods [LGM93] [Me94] can be used: in all such methods, all that is required at each step is a *subgradient* $\boldsymbol{g}(\lambda)$ of $\varphi()$ in λ, that can be cheaply obtained, after having calculated $\varphi(\lambda)$, as $\boldsymbol{u} - \Sigma_h \boldsymbol{x}^h(\lambda)$, being $(\boldsymbol{x}^1(\lambda) \ ... \ \boldsymbol{x}^k(\lambda))$ any optimal solution of (RMG_λ); our algorithm belongs to the latter class, as described in the next paragraph.

2. Overview on Bundle Methods

Bundle methods are a well-known class of NDO algorithms that are mainly characterised by keeping the first-order information about $\varphi()$ in a *disaggregated* form, in contrast with *aggragated subgradient* algorithms: visiting a (finite) sequence of points $\{\lambda_i\}$, the whole set (Bundle) of informations $\beta = \{ (\lambda_i , \varphi(\lambda_i) , \boldsymbol{g}(\lambda_i)) \}$ that represents the "history" of the computations performed so far is used to compute the next trial point (actually, a tentative ascent direction $\boldsymbol{d}$ along which λ_{i+1} is choosen). Note that, although $\boldsymbol{g}(\lambda_i)$ can be any of the subgradients of $\varphi()$ at λ_i, here it is that particular subgradient that is fund by the actual algorithm used to evaluate the functions: in this sense it can be denoted as a point-to-point function.

When applied to maximization of polyhedral functions, that can always be described as (large) Linear Programs, Bundle methods can be interpreted as variants of the Dantzig-Wolfe Decomposition approach [CFN95]: the *tentative ascent direction* $\boldsymbol{d}$ w.r.t. the *current point* $\bar{\lambda}$ is found by solving the Quadratic Problem

$$(\Delta_{\beta t}) \qquad \min_{\theta}\{\ (1/2)t\| \Sigma_{i\in\beta}\, \boldsymbol{g}_i\theta_i \|^2 + \Sigma_{i\in\beta}\, \alpha_i\theta_i : \Sigma_{i\in\beta}\, \theta_i = 1\ ,\ \theta \geq \boldsymbol{0}\ \}$$

where the $\boldsymbol{g}_i = \boldsymbol{g}(\lambda_i)$ are (in principle) all the previously obtained subgradients, $\alpha_i = \alpha_i(\bar{\lambda}) = \varphi(\lambda_i) + \boldsymbol{g}(\lambda_i)(\bar{\lambda} - \lambda_i) - \varphi(\bar{\lambda})$ are the current *linearization errors* and $t > 0$ is the so-called *trust region parameter*. It is important to note that, for practical efficiency, the Bundle can be implemented as $\beta = \{\ (\boldsymbol{g}_i\ ,\ \alpha_i)\ \}$, since such informations, together with $\bar{\lambda}$, suffice to carry on the computations. As shown in [CFN95], $(\Delta_{\beta t})$ can be seen to be equivalent to a (Lagrangean relaxation of a) "least-square version" of the classical Master Problem of Dantzig-Wolfe Decomposition, avoiding the typical problems that DW approaches encounter when dealing with unbounded MPs (i.e. the need for a Phase 1), but also has interesting interpretations in the usual terminology of nondifferentiable optimization, since it can be described as (a Lagrangean relaxation of) the problem of finding the minimum norm vector in some inner approximation of the $\max_{i\in\beta}\{\alpha_i\}$-subdifferential of $\varphi()$ in $\bar{\lambda}$.

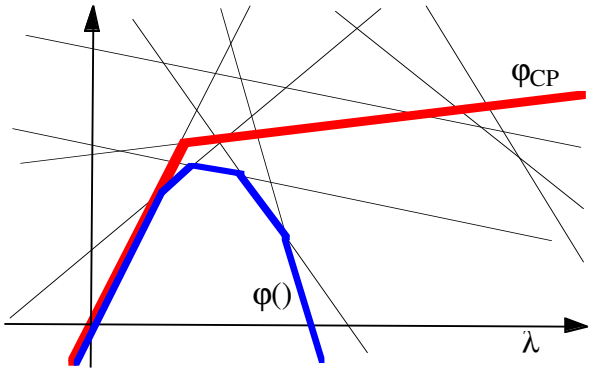

Figure 1: $\varphi_{CP}()$ is an upper approximation of $\varphi()$

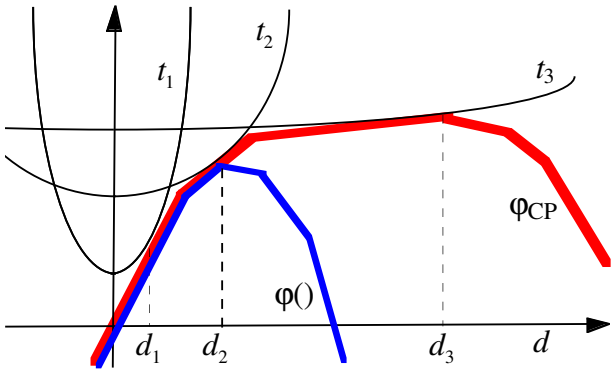

Figure 2: d_1, d_2 and d_3 are opt. sol. of $(\Pi_{\beta t})$ for $t_1 < t_2 < t_3$

Another interesting property of $(\Delta_{\beta t})$ is that its Quadratic Dual is

$$(\Pi_{\beta t}) \qquad \max_{\boldsymbol{d}}\{\ \varphi_{CP}(\bar{\lambda} + \boldsymbol{d}) - (1/2)t\| \boldsymbol{d} \|^2\ \}$$

where $\varphi_{CP}(\lambda) = \min_{i\in\beta}\{\ \varphi(\lambda_i) + \boldsymbol{g}_i(\lambda - \lambda_i)\ \} - \varphi(\bar{\lambda})$: this gives a "dual" interpretation of the Master Problem, in which the bundle is essentially a representation of the *Cutting Plane model* $\varphi_{CP}()$, the polyhedral upper approximation of $\varphi()$ built up with the first order information found so far. Note that, when $\varphi()$ itself is polyhedral, $\varphi_{CP}()$ is just the approximation of $\varphi()$ containing only that (small) subset of the faces (Figure 1), and that $\varphi_{CP}(\boldsymbol{0}) = 0$, i.e. the function is properly translated in such a way a simple quadratic *penalty term* (sometimes called *stabilising term*), weighted with t, can be used to discourage directions leading to "far away" points, in which $\varphi_{CP}()$ is probably a "bad" approximation of $\varphi()$ (Figure 2): the parameter t somehow measures our "trust" in the $\varphi_{CP}()$ model as we move farther from the current point, playing here the role played by the *trust radius* in the Trust Region methods for smooth optimization. For sufficiently large values of t (Figure 2), $(\Pi_{\beta t})$ is equivalent to the "naive" maximization of $\varphi_{CP}()$ alone

$$(\Pi) \qquad \max_{\boldsymbol{d}}\{\ \varphi_{CP}(\bar{\lambda} + \boldsymbol{d})\ \}$$

provided that the latter problem has finite maximum, which is very often *not* the case, expecially in earlier stages of the algorithm's operations: since the Cutting Plane algorithm for maximization of $\varphi()$ is exactly based on using $\bar{\lambda} + \boldsymbol{d}^*$ (being $\boldsymbol{d}^*$ the solution of (Π)) as the next trial point, the analogies between the two approaches, as well as the potential advantages of the Bundle method, are clear.
Once the tentative ascent direction has been found, there are several ways of exploiting it: basically, either a line search is started along $\boldsymbol{d}$ [HL93], in the style of Line Search methods for smooth optimization, or the value of $\varphi(\bar{\lambda} + \boldsymbol{d}^*)$ and the relative subgradient are used to adjust the parameter t [Ki89] [SZ92], in the style of Trust Region methods for smooth optimization. Common to both approaches, is the possibility not to "accept" the step, i.e. to remain in $\bar{\lambda}$, if an insufficient increase in the value of the objective function is attained: in this case, the newly obtained first-order information is used to update the Cutting Plane model $\varphi_{CP}()$, and evenctually t is reduced to be "more conservative" in the following steps. Actually, for solving MMCF a *constrained* Bundle algorithm is required, since nonnegativity of the Lagrangean variables has to be ensured: this can be tackled by just letting the QP subproblem "aware" of the constraints, i.e. by solving the extended problem

$$(\Pi'_{\beta t}) \qquad \max_{\boldsymbol{d}} \{ \varphi_{CP}(\bar{\lambda} + \boldsymbol{d}) - (1/2)t \| \boldsymbol{d} \|^2 : \boldsymbol{d} \geq -\bar{\lambda} \}$$

that guarantees feasibility of the tentative point $\lambda = \bar{\lambda} + \boldsymbol{d}$ w.r.t. the $\lambda \geq \mathbf{0}$ constraints, and such a problem can still be efficiently solved [Fr95]. $(\Pi'_{\beta t})$ also has a interesting theoretical interpretation, since it(s dual) can be seen as an approximation of the *steepest feasible ascent direction* finding problem for $\varphi()$ in $\bar{\lambda}$, but we cannot detail it any further: the interest reader is referred to [CFN95] for a complete discussion of the theory of Bundle algorithms when applied to polyhedral functions.
As a final remark, we want to stress the fact that, unlike most other dual algorithms, upon termination a Bundle algorithm not only givs the dual optimal solution $\bar{\lambda}$, but also the corresponding primal optimal solution $\boldsymbol{x}$, that is of obvious importance for practical applications. In our case, the optimal solution can be obtained as

$$(\bar{\boldsymbol{x}}^1 \ldots \bar{\boldsymbol{x}}^k) = \Sigma_{i \in \beta} (\boldsymbol{x}_i^1 \ldots \boldsymbol{x}_i^k)\theta_i$$

where $\boldsymbol{x}_i^h$ is the solution of the h-th MCF with costs $(\boldsymbol{c}^h + \lambda_i)$ and θ are the solutions of the latest $(\Delta_{\beta t})$.

3. A Bundle-type algorithm.

Although based on a common scheme [CFN95], Bundle algorithms can have dramatically different performances in practice, depending on how the various strategies are implemented, and on the choice of the parameters: the critical part is addmittedly the *trust region parameter updating strategies* (respectively the ε updating sdtrategies in [HL93]), for which no convincingly theory is known, and that are always implemented by means of heuristics whose value can only be judged by practical computational experiences. In our experience, methods and parameters that are claimed to work well on non-polyhedral functions (that are usually characterized by a limited number of variables, but a "strongly" nonlinear nature) are often not efficient on polyhedral functions, that seem to have different qualitative characteristics: this is not surprising, just thinking to the fact that general nondifferentiable functions often still have some sort of "second order" information that can be exploited, while polyhedral functions inherently lack it.

Our algorithm follows the general scheme for Bundle algorithms presented in [CFN95], using a simple implementation that can be concisely described as follows:

```
⟨ choose m1 ∈ [0, 1] , 0 < m3 , t0 > 0 , λ0 ≥ 0 , t* > 0 , ε > 0 ⟩;
⟨ let λ̄ = λ0 and t = t0; calculate φ(λ̄), g(λ̄) and let β = { (g(λ̄), 0) } ⟩;
repeat
  ⟨ let θ, d be the optimal solutions of (Δ'β,t), (Π'β,t) respectively ⟩;
  if( t*·|| d ||² + Σ_{i∈β} αi θi ≤ ε·φ(ȳ) )
        then STOP;
        else  λ = λ̄ + d;
              ⟨ calculate φ(λ), g(λ) and the corresponding α ⟩;
              ⟨ add (g, α) to the current "bundle" β ⟩;
              if( φ(λ) - φ(λ̄) ≥ m1 φCP(d) )
                  then λ̄ = λ; t = IncreaseT();
                  else  if( α > m3 Σ_{i∈β} αi θi )
                              then t = DecreaseT();
until( STOP );
```

The IncreaseT() and DecreaseT() functions are implemented as follows:

```
⟨ let M > 1 , m < 1 and tM ≥ t0 ≥ tm be fixed ⟩;
IncreaseT() =   max{ t , min{ tM , M·t , 2tv( v - (φ(λ) - φ(λ̄))) } }
DecreaseT() =   min{ t , max{ tm , m·t , (α + (φ(λ) - φ(λ̄))) / 2α } }
```

At each step, the *predicted increase* $\varphi_{CP}(\boldsymbol{d})$ is compared with the *obtained increase* $\varphi(\lambda) - \varphi(\bar{\lambda})$, and the step to λ is done only if a sufficient (relative) increase has been obtained. In this case, t is also increased with a (new) heuristic which uses a quadratic interpolation of the restriction of $\varphi()$ along $\boldsymbol{d}$: the new t is chosen as the maximizer of the quadratic function passing through $(\lambda, \varphi(\lambda))$ and $(\bar{\lambda}, \varphi(\bar{\lambda}))$ and having derivative $\boldsymbol{g}\cdot\boldsymbol{d}$ (where $\boldsymbol{g} = \boldsymbol{g}(\lambda)$) in λ, provided that it is larger than the current t but smaller than some (absolute and relative) safeguards. If the step is not accepted, the value $\sigma = \Sigma_{i\in\beta}\, \alpha_i\theta_i$ (that can be seen as an "average linearization error") is used to test the "reliability" of the newly obtained subgradient: if α is not too large than σ, $\boldsymbol{g}$ is believed to be significant and its addition to the bundle is expected to be sufficient to consistently improve the "accuracy" of $\varphi_{CP}()$, otherwise t is reduced, forcing the next tentative point to be searched "closer" to $\bar{\lambda}$, where $\varphi_{CP}()$ is hopefully accurate enough. To reduce t, a different heuristic [Ki89] is used, which differs from the previous in that it fixes the derivative of the interpolation in $\bar{\lambda}$ rather than in λ (Figure 3): the rationale for using two different heuristics is that it is reasonable to exploit the information given by the derivative in λ only if we are interested in what happens there, while it may make a little sense when we have already decided that our $\varphi_{CP}()$ model is a poor approximation of $\varphi()$ near λ. Furthermore, the first heuristic guarantee that the new t will be strictly larger than the old one if and only if $\boldsymbol{g}\cdot\boldsymbol{d} > 0$, while the second don't. Even though using two different heuristics instead of just one is not uniformly better on any instance, our experience seem to indicate that in the average it decreases the iterations count of about 5%.

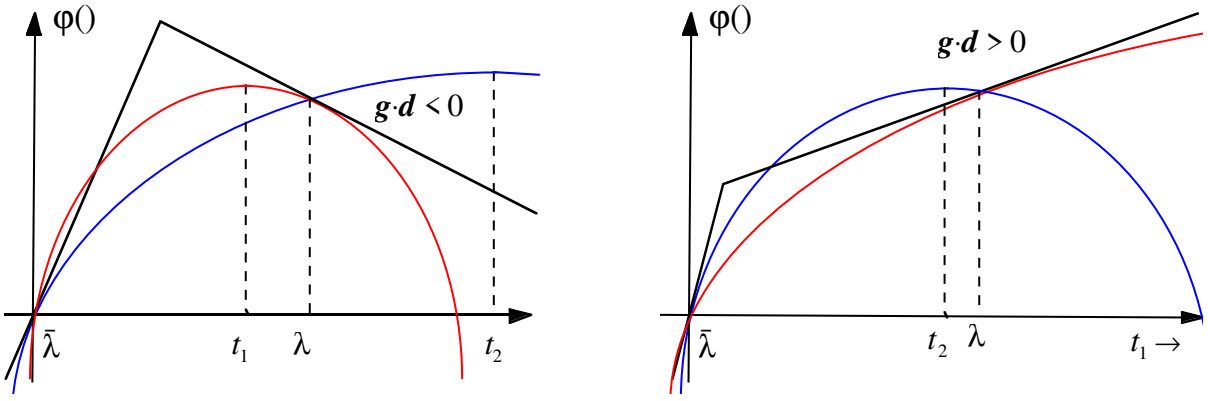

Figure 3: different quadratic interpolations of φ() give different new values for t

It is possible to show [CFN95] that such an algorithm finitely terminates: then, if t^* had been chosen sufficiently large the current point $\bar{\lambda}$ is guaranteed to be an ε-optimal solution of the problem; although selecting a feasible value for t^* is in principle difficult, in practice it can be easily guessed by looking at few test executions of the code (it has to be a bit larger than the average value of t), and it is fairly stable among instances of the same class.

For this algorithm to work efficiently, it needs a "large" value of the m_3 parameter, say something around 3: this is a relevant difference with the suggested value of the analogous parameters in [Ki89] and [SZ92], say 0.9, that is intuitively justifiable with the qualitative difference between polyhedral and general nondifferentiable functions alluded to in the previous paragraph, i.e. the extremely "`kinky`" nature of the former (whose epigraphs have an enormous number of facets) compared to the relatively "smooth" behaviour of the latter. The practical result of such a setting is that, for problems requiring a large number of iterations to terminate, the value of t is fixed in the early stages of the algorithm's operations and almost never changed thereafter: using smaller values, as suggested in the literature, causes t to rapidly get to a very small value, ending in a very long sequence of small steps that dramatically increase the iterations count. The choice of the other parameters is far less critical: m_1 seem to be properly fixed to 0.1 and the (absolute and relative) safeguards on t almost never enter to play.

An innovative characteristic of our implementation is the *Lagrangean variables generation strategy*: rather than having a fixed vector of Lagrangean multipliers, we only use the variables that are really necessary, i.e. those that at some iteration have the corresponding entry of the direction $\boldsymbol{d}$ strictly positive. Such an approach can be seen from a primal viewpoint as a row generation strategy, in which a complicating constraint is added only when it become tight: this is useful because, especially for "easy" problems, few of Lagrangean variables ever become nonzero at any point of the algorithm, and the set of active Lagrangean variables is very stable after the first few iterations, so that the (relatively costly) "pricing" operation of determining whether new variables have to be added can be performed with a low frequency. This has been allowed by the use of a fast, specialized QP solver [Fr95] that uses a *two-level active set* strategy, efficiently supporting the online *creation and destruction of variables* of the QP subproblem ($\Pi_{\beta t}'$).

Some other details of the Bundle-specific part of the implementation might be of interest: we use a standard "reset rule" by eliminating subgradients that have zero θ_i multiplier for more than a fixed number (≈20) of consecutive steps, and never try to reduce the size of the Bundle by aggregation; we do pricing for all the first p_1 (≈30) iterations, and then once every p_2 (≈10) iterations; we never try to eliminate "active" variables λ_i that stay at zero value for a long time, for it seems that the overhead to be paid it is not worth the savings that can be obtained. Finally, we remark that our code is a "complete" MMCF solver, in that it can give the primal optimal solution $\boldsymbol{x}$: since this requires keeping the solutions of (in principle) all the MCFs, the cost may be considerable in term of memory, but not so much in terms of time (degradation of performances is always well under the 5%). However, it can be easily seen that the *aggregate* optimal solution $\bar{\boldsymbol{x}}_A = \Sigma_h \bar{\boldsymbol{x}}^h$ can be recovered at essentially no cost from the solution $\boldsymbol{d}$ of the last ($\Pi'_{\beta t}$) solved, and in some cases this is all that is needed, so that calculation of the full optimal solution is performed only on request.

In most cases, and especially if the number of commodities is large, most of the time is spent in solving the Min Cost Flow subproblems: hence, it is of obvious importance to have a fast MCF solver, that is capable to efficiently reoptimize after a cost change. However, in many classes of instances the MCF subproblems are in fact just Shortest Path problems, that can be solved much faster than ordinary MCFs: therefore, any general purpose Multicommodity solver should have a set of efficient, specialized solvers that may be invoked according to the properties of each subproblem.
In our C++ code, this is solved by means of an *abstract base class*, `McfClass` defining a standard interface for MCF solvers: within the MMCF solver, *k objects* of proper *derived* classes are constructed according to the properties of any single subproblem, making it easy to deal even with instances with nonuniform subproblems, and the *public methods* of such objects are used for changing the costs, solving the problems and collecting the results. The advantage of this approach is that it is easy to extend the MMCF solver to deal with special classes of instances, by essentially just developing the appropriate subproblem solver: moreover, the MMCF solver is completely independent from upgrades in the MCF solver, or even changes of algorithm. Up to now, two *derived classes* of McfClass have been constructed, the first one implementing a general-purpose MCF solver based on the RelaxIV algorithm [Be91] (that is known to be fast and to reoptimize efficiently), and the second one implementing four classical Shortest Path algorithms (queue, dequeue, Dykstra and [c-]heap).
An analogous structure has been used for the MMCF solver: an *abstract base class* `MMCFClass` has been defined to isolate applications from the details of the actual algorithm used, and not only our solver, but also all the codes that we have used in the experimentations has been inserted into *derived classes*, making it possible for an application to use any of the algorithms indifferently. Moreover, a service class `Grafo` provides an unified mean to read description of MMCF problems (from files, in several different formats, or from memory) and handle them: preprocessing, changes or queries of the instance's data and memory management are all implemented within that class, making it really easy to develop new solvers, or to use an existing algorithm as MMCF solver by writing only a small amount of proper interface code.

We want to stress the fact that, although apparently minor, all such implementative details are critical for obtaining an overall efficient code: as a first example, consider the following tables where our custom BTT code [Fr94] for ($\Delta'_{\beta t}$) is compared against two standard QP codes when used within a

Bundle algorithm, i.e. considering as execution times the sum of all the execution times for all the calls to the QP solver within a single execution of the Bundle algorithm.

BTT	*QPP*	***Ratio***
.011	0.33	28.88
.013	0.96	71.88
.021	1.94	92.14
.040	2.98	74.38
.048	5.09	106.08
.149	16.09	108.07

Table 1: BTT and an Interior Point QP solver

TT	*QL*	***Ratio***
0.03	0.36	12.68
0.62	9.89	16.00
1.38	27.59	19.98
12.72	277.80	21.84
9.50	295.29	31.08
10.12	751.90	74.32

Table 2: TT and a standard QP solver

It can be easily seen that our custom code is up to two orders of magnitude faster than non-specialized codes (for a discussion on these results, the interested reader is referred to [Fr95]: we only remark that one of such QP codes is exactly the one used within the only previous Bundle approach to MMCF [Me94]): since the relative computational cost of the QP solution w.r.t. the total time spent by the algorithm can range from much less than 1% to 20%, depending on some factors such as the number of commodities and the "difficulty" of the instance, any Bundle algorithm using a non-specialized QP solver would rapidly become impractical for the excessive cost of the solution of the ***d*** finding subproblem. Actually, the above results are in fact prudential, in that they do not take into account the extra advantages of the BTT algorithm, such as the ability to efficiently cope with variables creation/deletion, that is necessary to implement the Lagrangean Variables Generation (LVG) strategy. As shown in Table 3, the latter in turn allows consistent savings: the Ratio between the *total* execution time of a version of the code not using the LVG strategy (N-LVG column) and a version using it (LVG column) is over 1.5, but the factor becomes of 5 or more (Rel. Ratio column) when the cost of the corresponding *synchronisation phases*, i.e. time that is not spent in the calculation of $\varphi(\lambda)$, is compared instead. This is especially important for "easy" problems with relatively few commodities, where the fraction of the time spent outside the $\varphi()$ calculation is more relevant: conversely, for large problems with many commodities the computation cost of $\varphi(\lambda)$ is by far the computational bottleneck of the algorithm, but it can in turn be dramatically decreased by exploiting the structure of the subproblem.

k	*n*	*m*	*LVG*	*N-LVG*	*Ratio*	*Rel. Ratio*
10	50	400	1.56	2.12	1.47	4.88
10	50	625	2.47	3.21	1.51	2.72
10	100	1600	7.65	10.22	1.38	4.22
10	100	2500	6.49	10.29	1.68	5.18
100	50	535	1.69	2.65	1.52	2.67
100	30	518	1.02	1.79	1.72	3.08
100	30	682	0.89	1.64	1.68	3.02
100	100	1600	1.47	2.34	1.57	3.73
400	30	520	3.23	5.42	1.48	2.45
400	30	686	2.69	4.21	1.50	2.56

Table 3: effect of the variables generation strategy

k	*n*	*m*	*SPT*	*MCF*	*Ratio*
3	47	98	0.02	0.04	2.00
6	85	204	0.09	0.18	1.98
5	34	216	2.42	6.35	2.38
15	43	285	13.36	33.87	2.31
10	190	502	1.23	3.83	3.51
15	285	771	3.38	10.68	3.83

Table 4: using a SPT algorithm instead of a MCF whenever possible allows consistent savings

This is shown in Table 4, where the total execution time of a version of the code using a (relatively naive) SPT code (SPT column) is compared with that of a version using the (sophisticated) MCF solver (MCF column) instead: the Ratio column shows that the specialized solver can reduce the *total* running time of the algorithm of up to a factor of four, even on very small problems with small graphs and few commodities.

4. The other codes

It has been stated in the past [HL81] [HL83], and it seems to be currently agreed among many researchers, that cost decomposition methods for the solution of large scale block-angular Linear Programs are unlikely to achieve substantial performance improvements w.r.t. more "traditional" LP approaches, since "... the folklore is that generally such schemes take a long time to converge so that they're slower than just solving the model as a whole, although research continues. For now my advice, unless [...] your model is so huge that a good solver can't fit it in memory, is to not bother decomposing it. It's probably more cost effective to upgrade your solver, if the algorithm is limiting you, than to invest your time ..."[◊]. The veracity of such observation, based on several computational experiences about the Dantzig-Wolfe Decomposition approach and on sensible considerations about the programming cost that a state-of-the-art decomposition approach would require, has not to be underestimated: however, if cost decomposition approaches have to be of any value, MMCF problems are probably the application where such a value could show up, and our aim was to compare recent alternative MMCF solvers, in order to understand where exactly a modern decomposition approach stand, in terms of efficiency, among them.
In principle, any general-purpose LP solver can be used to solve MMCF problems: actually, our experience shows that (according to above statement) an efficient general-purpose LP solver can be the best alternative even for solving even quite large Multicommodity problems. This is not surprising, considering the dramatic performance improvements that simplex based LP solvers have been obtained in the last years by means of both theoretical (refined matrix-factorisation algorithms, efficient pricing rules) and practical (sophisticated implementation of data structures, preprocessing, state-of-the art programming) refinements: such codes are often the result of an impressive amount of work and experience obtained during many years of development.

For our experiments, we used one of the best commercial LP solvers on the market, Cplex[‡] 3.0 [Cp94]: the code offers choice between Primal and Dual simplex algorithm, with possibility of exploiting embedded network structure of the problem to perform "advanced starts", and a Barrier (Interior Point) solver. Like for the other codes, all the efforts have been made to use Cplex in its full power: some significant subsets of the test problems have been repeatedly solved with various combinations of the available optimization parameters, in order to find a good one. We claim this process being very important in order to obtain a fair computational comparison: to this purpose, we report our experience in tuning two Cplex optimization parameters, namely the solution method and the pricing rule. As we said, Cplex offers both "Pure" Primal and Dual (PP and PD) simplex: even

[◊] John W. Gregory, in the "Linear Programming FAQ" monthly posted to the UseNet newsgroup sci.op-research.

[‡] Cplex is a registered trademark of Cplex Inc.

more important, it embeds a network identification/extraction phase that can identify the "hidden" network structure of a Linear Program, and a network simplex algorithm that can be used to efficiently solve the "restricted" LP in order to obtain a good starting base for the general simplex. The performances of the resulting "Hybrid" Primal and Dual (HP and HD), obtained by starting the primal and dual simplex algorithms respectively from such a base, are reported in Table 5.

Size	PP	PD	HP	HD
5125	9.02	2.72	4.73	1.36
16000	50.90	23.47	21.37	6.07
25000	81.05	14.15	22.65	5.41
47850	504.04	124.33	25.48	10.06
68225	1393.63	42.24	18.96	8.10
61275	1646.35	254.13	185.04	78.94
207733	16707.66	492.33	244.89	44.67
257700	13301.88	556.39	165.56	48.55

Table 5: different simplex algorithms in Cplex

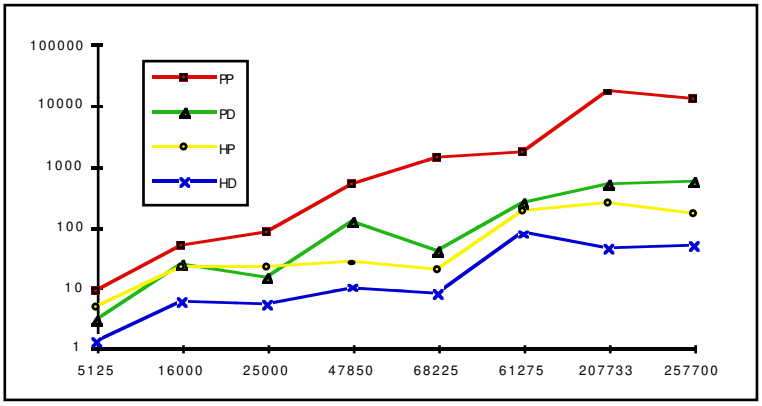

Chart 1: data in Table 5

Here, and later on, *Size* = *m*×*k* is the number of variables of the LP problem, times are in seconds on a HP9000/712-80 workstation (≈ 80 SPECint92 and 120 SPECfp92) with 64Mb of physical RAM and figures are averaged from groups of several problems: since the Hybrid versions are faster than their Pure counterparts of up to a factor of 70, and that the Dual approach can be faster than the Primal one of up to a factor of 30, the difference between Pure Primal and Hybrid Dual algorithms can be a factor of almost 400 on large problems, so that it would not be not unfair to regard Cplex as a *semi-specialized* algorithm for network-structured problems:

Even within the same algorithm, details such as the pricing strategy can have a dramatic impact on the number of iterations, hence on the performances: in Table 6, a comparison between the four available pricing strategies for Dual simplex (the standard dual pricing HD1 and three variants of the steepest-edge pricing HD2-4) is presented, showing that performance improvements of up to a factor of 15 can be obtained by just selecting the appropriate rule - even more, since the two entries marked with " * " in the standard dual pricing column are for instances that had taken (much) longer than 1000 seconds to terminate.

Size	HD1	HD2	HD3	HD4
5125	4.08	1.36	2.11	1.34
16000	19.65	5.99	11.00	6.08
25000	18.24	6.16	19.71	5.39
47850	13.35	11.95	51.03	10.00
68225	8.05	10.76	71.59	8.06
61275	1000*	54.07	152.36	79.16
207733	1000*	88.38	681.27	44.88
257700	57.53	92.58	846.65	48.48

Table 6: different pricing rules in Cplex

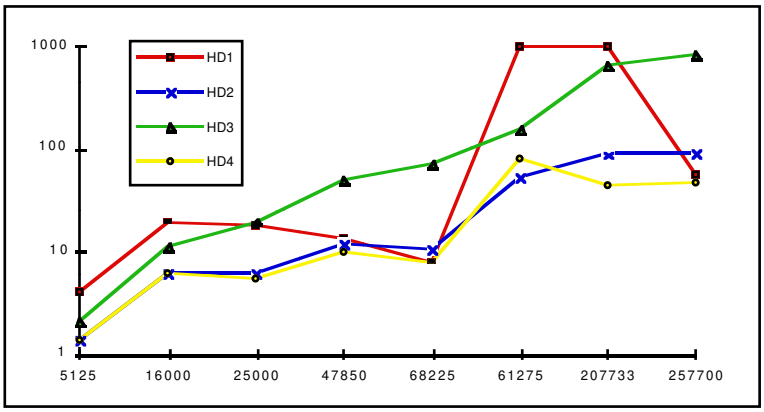

Chart 2: data in Table 6

Hence, MMCF problems can be very efficiently dealt with by means of a "standard" Dual simplex algorithm, provided that careful implementations of some critical details are available, and the network structure is exploited: as we already said, comparing a specialized MMCF algorithm against such a semi-specialized solver is therefore not unfair, provided that enough efforts are done to properly use the existing technology.

Among the LP techniques, Interior Point (IP) algorithms have grown a favourable reputation for the solution of very large, sparse Linear Programs: although such methods require factorisation of some large matrices, the number of iterations needed to reach convergence is usually very small, achieving a good overall cost. Formulated in the standard node-arc way, a MMCF is a large LP problem with up to three nonzeroes for each column: not having any "dense" column, such a Linear Program seem to be a very good candidate for being solved with an IP method. Even if some specialized IP algorithms for block-structured LPs have been proposed [SM91] [KKR94], none of them have been made available to us, hence we tested two generic IP solvers: one is the Cplex-Barrier solver, and the other is the LOQO 2.21 code [Va92] [VC93] [Va94], one of the best implementations of the *primal-dual, path following, predictor-corrector* approach. As for simplex methods, some optimization parameters have to be carefully tuned for obtaining good performances: the most important is the choice between "myopic" (Minimum Local FillIn) and "global" (Priority Minimum Degree modified Cholesky and Multiple Minimum Degree respectively) heuristics for column ordering when computing the LLT factorisations, but there are other parameters, like the choice between leftward and rightward Cholesky factorisation in Cplex. Since the two codes use different (*normal equations* and *reduced-KKT* respectively) approaches to matrix factorisation, it is not strange that the best choices are different: while for LOQO the global heuristic is uniformly better, but does not give rise to dramatic improvements, for Cplex the myopic one turns out to be more efficient of up a factor of 2, as shown in Table 7 below

k	n	m	Size	Iter	MMD-L	MMD-R	MLF-L	MLF-R
10	50	625	6250	12	37.70	30.97	33.24	27.84
10	50	625	6250	12	37.35	30.98	33.13	27.60
100	30	518	51800	12	3709.12	3344.74	1789.23	1829.65
100	30	519	51900	13	4036.83	3382.39	1927.01	1935.82

Table 7: some examples from Cplex-Barrier optimization parameters tuning

The table shows some relevant details, like the fact that the total number of iterations is uniformly small, and that it is not always possible to find the "best" setting for a given parameter: in this case, while the rightward factorisation is consistently better on small problems, the leftward factorisation allows consistent savings (in absolute value, if not in percentage) on larger problems.
Even more important, however, is that the *preprocessing* done by the `Grafo` object is absolutely crucial in order to obtain better performances from the LOQO code. Once that a MMCF problem has been read, the Grafo object uses a rough preprocessing to identify and remove redundant mutual capacity constraints and fixed flow variables, and such informations can be used to diminish the dimension of the LP passed to LOQO: although the preprocessing is far from being smart, this can improve the performances of LOQO of well above a factor of 10 - compared to the few percent of improvement that are obtained by our own algorithm for the same reason, this result give an idea of the

practical significance of such details. In fact, reducing the number of mutual capacity constraints is far less important for Cplex, that embeds his own efficient preprocessing phase: the fact that some improvements can be obtained anyway shows that knowledge of the structure of the problem makes it easier to locate redundant constraints, and suggest that investing in a smarter preprocessing phase may be profitable (especially for the other approaches).

The last code is PPRN 1.0 [Ca94] [CN95], a recent implementation of a *Primal Partitioning* approach to MMCF, that is capable of dealing with (Generalised) Multicommodity problems with extra constraints an even with a Nonlinear objective function: when used on Linear Multicommodity problems, primal partitioning methods are specialized primal simplex algorithms, that extensively exploit the structure of the base matrices of the MMCF polyhedron to perform the normal operations of the simplex method using a working matrix of reduced dimension. In their computational experience [CN95], the authors show that PPRN generally outperform the other known primal partitioning implementation, the MCNF85 code [Ke79], hence we took PPRN as a good representative of this class of approaches. Like in the previous cases, the code was tested on some "benchmark" problem sets to tune its (several) optimization parameters, and as for the other codes a couple of them have shown to have a dramatic influence on the overall efficiency of the method: the main of them is the *Phase 0* strategy, telling whether just any feasible solution of the k MCFs is initially generated, or they are solved to optimality and their optimal solution used. Note that such a solution is used by Cplex for its advanced start options, and it is typically generated as the first iteration by any cost decomposition method: in fact, as it happened for Cplex, the running times obtained by solving the subproblems to optimality (Opt column) are up to a factor of 20 smaller than those obtained by just finding a feasible solution (Feas column), as shown in Table 8 below.

Size	*Opt*	*Feas*
7125	2.21	6.89
18250	7.87	29.62
28750	7.76	53.68
50783	9.43	43.24
63850	32.33	145.39
91250	11.24	271.73
207867	55.62	425.68
268200	29.28	626.73

Table 8: Phase 0 options in PPRN

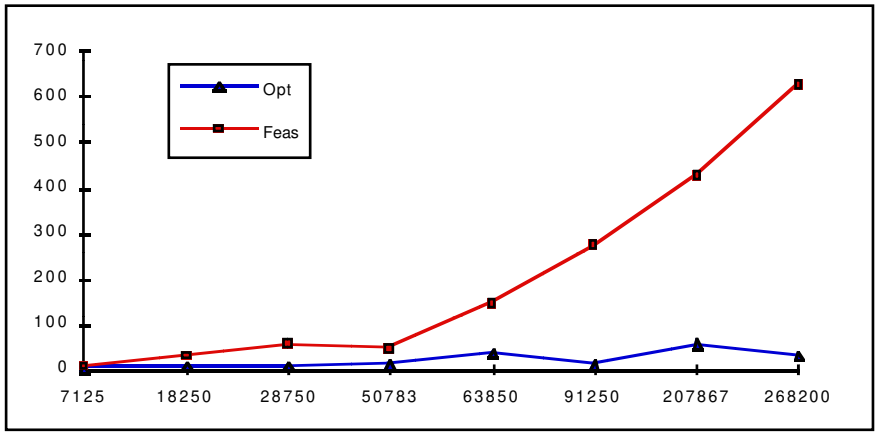

Chart 3: data in Table 8

Still, appropriate selection of the pricing strategy is necessary in order to obtain good results: as shown in Table 9, choosing two of the four pricing rules implemented in PPRN (whose description is out of the scope of this article) can result in performance differences of over a factor of 3 on some classes of instances.

Size	P1	P3
7125	21.99	64.02
18250	8.40	41.41
28750	30.13	32.84
50783	8.98	25.99
63850	4.67	5.88
91250	42.06	53.50
207867	31.85	36.33
268200	8.90	16.38

Table 9: pricing rules in PPRN

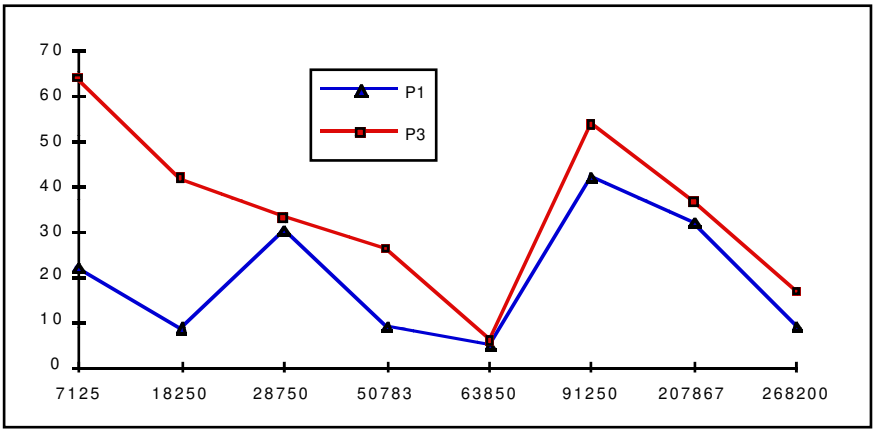

Chart 4: data in Table 9

When judging the overall performances of PPRN, some factors have to be taken into account: first, it is an experimental code at its first final version, while all the others (except our own) have a much longer history, and hence are possibly more sophisticated; secondly, it's mainly written in FORTRAN, while all the others have been coded in C, and FORTRAN compilers are believed to be somewhat less efficient of their C counterparts, at least on Unix workstations; finally, due to its input format it was impossible to make full use of the preprocessing informations generated by the `Grafo` object, and we have been told by one of the authors that no preprocessing phase has still been implemented inside the code.

This set of algorithms does not exhaust all the proposed computational approaches to MMCF: barring the fact that the IP algorithms used here are not specialized, we were not able to find a recent implementation of *resource decomposition* approaches [KS77] [AHK79] (that however are not believed to be competitive on very large scale problems), nor to properly state the value of *approximation algorithms* [LPS93], nor to satisfactorily compare our code with other recent cost decomposition approaches [JLF93] [Me94] [PZ94] [DP95]. However, it should at least give a better idea of the possibilities of cost decompositions codes in the current scenery of computational approaches to MMCF.

5. The test problems

The other fundamental ingredient for obtaining a fair computational comparison is obviously a adequate set of test problems: currently, several data sets and random generators of MMCF problems are freely available and have been used in the literature to test one or the other of the proposed approaches. However, many of the widely diffused test problems are small, and even the larger ones have a small number of commodities: this is mainly due to the fact the complexity of solving a MMCF (with most of the methods) grows very fast with the number of commodities, so that problems with hundreds of commodities have for long time been out of reach. For our experiments, we tried to collect the larger possible number of data sets: however, not all of them are described here, since tests on some of them are still in progress, therefore in the following we present the data sets on which a complete set of experiments have already been terminated.

The Canad problems

The first data set we used is a bunch of 90 randomly generated problems taken from [GC95], where they have been used to test bounding procedures for the Multicommodity Fixed Charge Network Design Problem, a difficult mixed integer problem that reduces to MMCF when all the (binary) design variables have been fixed: the problems have been generated with two custom generators, one for bipartite and the other for general nonbipartite graphs, and a parameter called *capacity ratio* was available as a "knob" to produce lightly or heavily capacitated problems. Since the generator had not been made available to us, we didn't had the choice of the generating parameters: however, the data set is interesting in that it is representative of some real applications where the number of commodities can be considerably larger that the number of nodes (up to 400 vs 30).

The Mnetgen generator

The Mnetgen generator [AK77] is a well-known and widely used Multicommodity problems random generator: the version we used for our tests had been slightly changed from the original one, while translating it to C++, by replacing the simple random number generator (that has shown not to be that portable) with the standard drand48() routine and enhancing the output file format to be more similar to the standard proposed in [JL92].
For our experiments, we developed a suite of 216 problems, generated with uniform rules in order to study the influence of several parameters on the efficiency of the algorithm: for each value of n in {64, 128, 256}, we generated 12 different problems for each value of k in {4, 8, ... , n} (Mnetgen cannot generate problems with $k > n$) for a total of 18 groups of problems characterised by a pair (n, k).
Within each group, we generated 6 "sparse" problems and 6 "dense" ones (with m/n respectively equal to about 3 and 8), and both these subgroups were subdivided into 3 "easy" and 3 "hard" problems: an easy problem have mutual capacity constraints on the 40% of the arcs and 10% of the arcs with an "high" cost, while in a hard problem the arcs with mutual capacity constraints and an "high" cost are the 80% and 30% respectively. Finally, the 3 subproblems characterised by the touple (n, k, s/d, e/h) differs in the number of arcs that have individual capacity constraints, fixed to 30%, 60% and 90% respectively. A criticism that is often done to randomly generated test suites is that real-world problems may be really different from randomly generated ones, and the latter may happen to be somehow easier to solve: however , generating problems with controlled characteristics may help in shedding some light onto the features that makes a given algorithm more or less efficient on a class of instances, and the "hard" MMCF problems that we generated (especially if dense, also) have shown a posteriori to be significantly harder to solve (by all the codes) then both the "easy" ones and other problems of equivalent dimensions.

The PDS problems

The PDS (Patient Distribution System) data set comes from a model for transportation of patients from a place of military conflict. The available data actually represents only one basic problem, but many MMCFs can be obtained from there since the model takes into account time: changing one parameter (the number of days covered by the model), larger and larger instances can be generated. In fact, the PDSi problems (i being the above *planning horizon*) have a time/space network, whose size grows about linearly with i: this characteristic is often encountered in MMCF models of practical

problems, such as the Freight Transportation Planning [FPL93], and therefore the PDS data set can be considered representative of a large set of applications. However, while in the typical models the number k of commodities usually grows with the planning horizon, in the PDS(i) it is uniformly fixed to 11: hence, the instances of this data set tend to have a very large m / k ratio, that, as we will see, makes them less actractive for cost decomposition methods. Moreover, although all the commodities always have a single source and a single destination, the MCF subproblems does not have a SSP structure because of the presence of single-commodity capacities: we suspect that it would be possible to reformulate the model to make it more easily solvable by cost decomposition, but this raises the entirely different question of the impact of the formulation over the relative efficiency of the solution algorithms, that would require separate investigations.

The JLF data set

A relatively large problems collection was presented in [JLF93] in the context of application of the DW algorithm to MMCF: it is formed from various groups of relatively small problems, often with a SPT structure, taken from different real-world applications. Although no problem of this set has more than 15 commodities, and the groups are all formed of few problems, this data set is interesting in that it comprises problems with different structures.

6. Computational results and conclusions

In the following, we illustrate the results that we obtained by applying the codes described in §3 and §4 to the solution of the test problems described in §5: we will initially present the data, and afterwards we will discuss some of the conclusions that can be drawn upon it.
For all the data sets, we consider as performance measure the running time in seconds taken by the solution of the problem, excluding loading and preprocessing time (see §5): unless explicitly stated, each entry in the tables will represent the average of a set of problems with similar *Size*, b ($< m$, if present) is the number of arcs that have a mutual capacity constraint and a "*" in some entry will mean that the algorithm was not able to solve (all) the problems of that group, usually due to memory problems. For all codes (but PPRN, that has not such a feature), we selected the required relative precision of the optimal solution to 10^{-6}.

The Canad problems

The results obtained for both kind of problems (on generic or bipartite graphs) are consistent: both the IP-based codes are over two orders of magnitude slower than the faster among the other approaches, and are not able to solve the largest problems due to memory scarcity, even with 128 Mb of (virtual) memory available. Among the remaining codes, MMCFB is always competitive with the others, and outperforms them on all but for the smallest problems (with smallest size and few commodities): being never twice slower than any of the other codes, it is a factor of 20 faster than Cplex on large problems, while PPRN is only among 1.5 to 2 times faster than the latter.

k	n	m	Size	MMCFB	Cplex	PPRN	LOQO	IPCplex
40	20	228	9110	0.29	0.98	0.48	198.24	135.21
40	20	288	11525	0.18	1.18	0.38	254.13	177.63
200	20	229	45750	1.09	7.24	5.02	663.09	1018.37
100	30	518	51775	0.70	6.89	3.97	*	1997.73
200	20	290	58050	0.59	7.47	4.68	989.60	1455.01
100	30	682	68211	0.61	7.48	2.56	*	3206.59
400	30	519	207600	2.35	40.01	35.91	*	*
400	30	684	273467	2.03	39.24	29.00	*	*

Table 10: Canad problems, generic graphs

k	n	m	Size	MMCFB	Cplex	PPRN	LOQO	IPCplex
10	50	400	4000	1.56	0.90	1.81	14.17	11.02
10	50	625	6250	2.47	1.81	2.06	48.53	33.89
10	100	1600	16000	7.65	6.01	8.39	105.60	*
10	100	2500	25000	6.49	5.47	6.96	394.96	*
100	50	400	40000	1.00	14.41	16.98	468.81	352.79
100	50	625	62500	2.15	101.23	41.02	*	*
100	100	1600	160000	1.47	36.76	30.08	*	*
100	100	2500	250000	1.66	33.05	27.94	*	*

Table 11: Canad problems, bipartite graphs

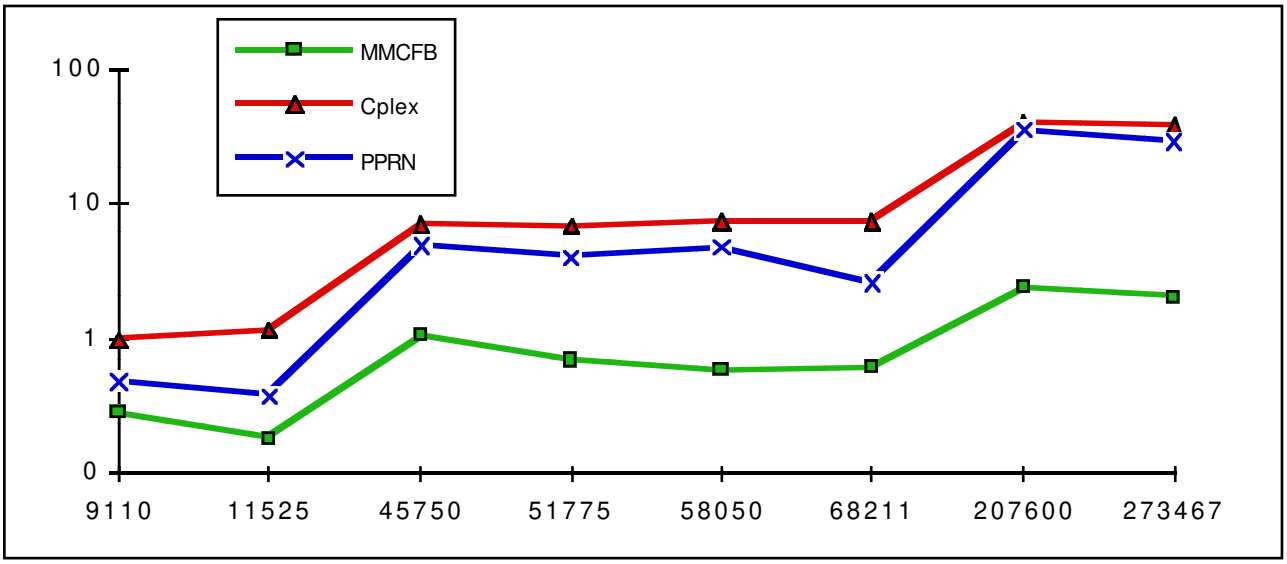

Chart 5: data in Table 10

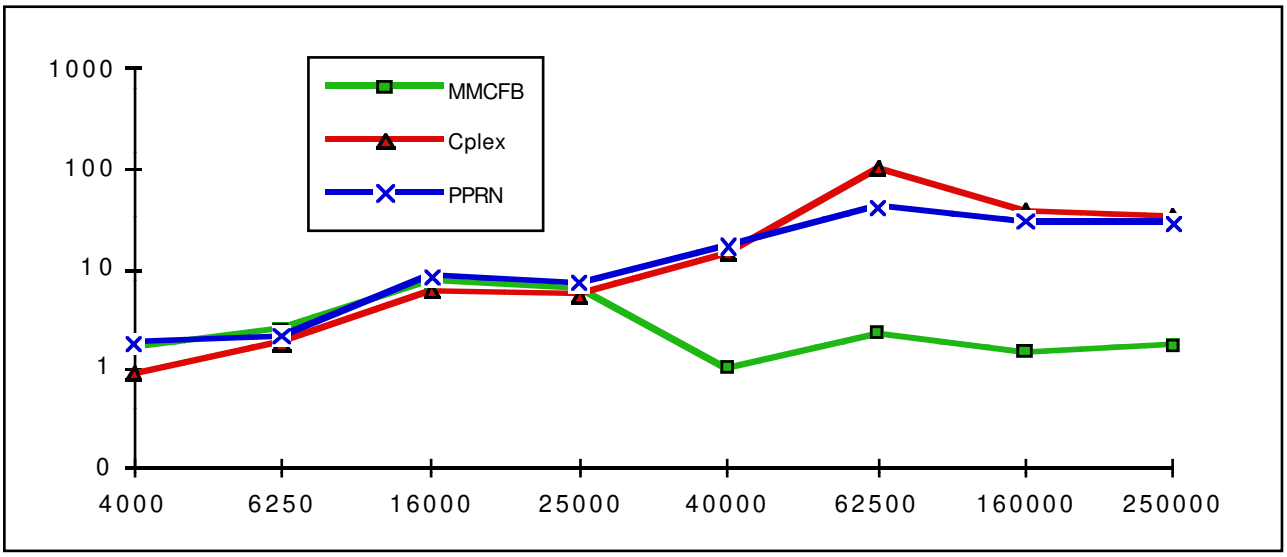

Chart 6: data in Table 11

Actually, there are some minor differences between the two problems groups, for instance MMCFB is always faster than the others on the non-bipartite graphs: however, these differences have probably to be imputed to the fact that the "small" instances in the latter group have fewer commodities rather than to some structural difference.

The Mnetgen problems

The set of problem generated with Mnetgen covers a wide range of different characteristics: there are small problems, large problems with few commodities and large problems (more than half a million variables) with large graphs and many commodities. The following table summarise the results we got, where all the 12 problems with the same (n, k) have been grouped together.

k	*n*	*m*	*b*	*Size*	*MMCFB*	*Cplex*	*PPRN*	*LOQO*	*IPCplex*
4	64	362	148	1449	0.08	0.21	0.16	1.73	1.45
8	64	371	183	2969	0.30	0.50	0.60	9.22	7.56
16	64	356	191	5695	1.18	2.08	3.45	58.96	40.14
32	64	362	208	11584	3.70	12.63	22.36	190.76	98.73
64	64	361	213	23131	9.76	117.42	147.40	244.66	216.44
4	128	694	293	2774	0.89	0.59	1.07	7.18	6.49
8	128	735	363	5883	3.49	1.94	5.47	66.65	50.56
16	128	766	424	12257	11.86	18.60	42.88	683.47	394.19
32	128	779	445	24923	27.75	223.97	507.40	*	*
64	128	784	469	50176	50.94	1206.80	2211.73	*	*
128	128	808	485	103467	60.82	5865.52	6535.88	*	*
4	256	1401	570	5602	19.95	3.20	15.38	50.70	40.37
8	256	1486	743	11884	51.70	22.77	112.13	568.02	377.87
16	256	1553	854	24852	82.21	217.69	972.79	*	*
32	256	1572	907	50301	144.83	1548.58	6628.00	*	*
64	256	1573	931	100677	176.11	7790.79	18435.99	*	*
128	256	1581	932	202315	272.21	35984.18	61587.64	*	*
256	256	1503	902	384725	441.33	110897(+)	187156(+)	*	*

Table 12: aggregated results on the Mnetgen problems

It can be immediately seen that, on the largest problems, our code outperforms the others of over two orders of magnitude: for instance, the hard and dense (256, 256) problems were solved in an average of about 1000 seconds by our code, while PPRN was stopped after over 400000 seconds and Cplex even earlier, since it required too much memory and our workstation went "trashing" i.e. spending only a few percent of the time in actual calculations, all the rest being taken by paging virtual memory faults. In a very mild estimate, that has been used for the Table 12 (the (+) entries), the expected running times of Cplex and PPRN on those instances would be (in absence of memory problems) of more than 300000 and 600000 seconds respectively: this actually gives us a maximum improvement of over three orders of magnitude, since the (256, 256, d, h, 90%) instance is solved by our code in less than 500 seconds. However, a closer examination of the data enlighten the potential weakness of our code on large instances with few commodities: this is better visualised on the following two tables and charts, where the ratio between the running times of Cplex (respectively PPRN) and MMCFB is reported for different values of (n, k) - the entries marked with a "*" are those based on estimates, and should actually be larger than that.

	4	*8*	*16*	*32*	*64*	*128*	*256*
64	2.54	1.67	1.77	3.41	12.03		
128	0.66	0.55	1.57	8.07	23.69	96.44	
256	0.16	0.44	2.65	10.69	44.24	132.20	251*

Table 13: ratio between Cplex and MMCFB times

	4	*8*	*16*	*32*	*64*	*128*	*256*
64	1.95	1.99	2.93	6.05	15.11		
128	1.21	1.57	3.61	18.28	43.42	107.46	
256	0.77	2.17	11.83	45.77	104.69	226.25	424*

Table 14 ratio between PPRN and MMCFB times

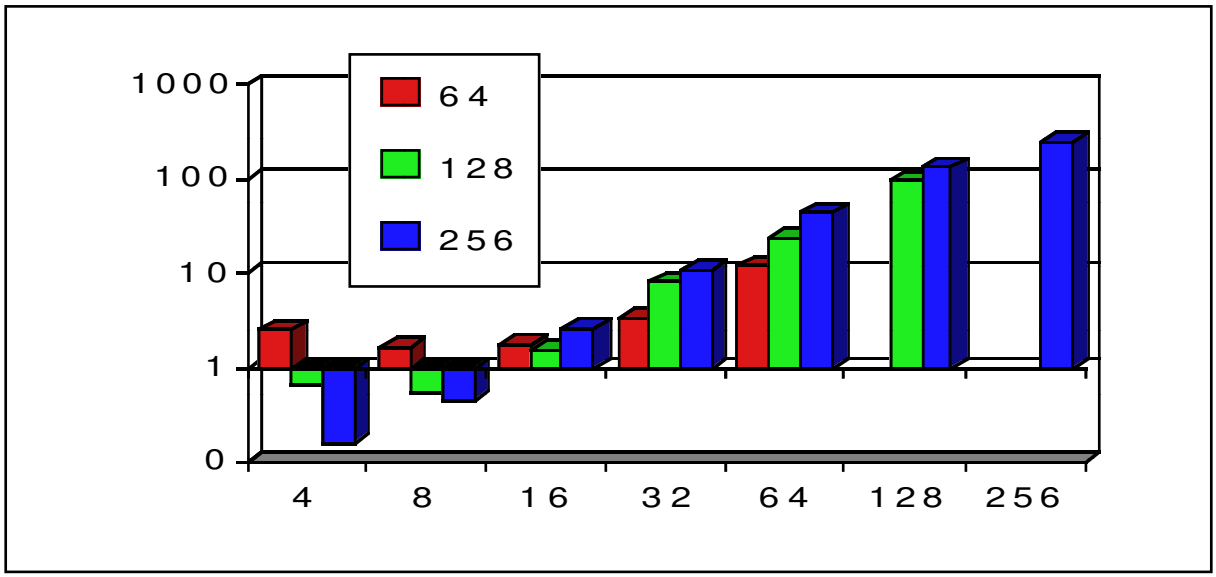

Chart 7: data of Table 13

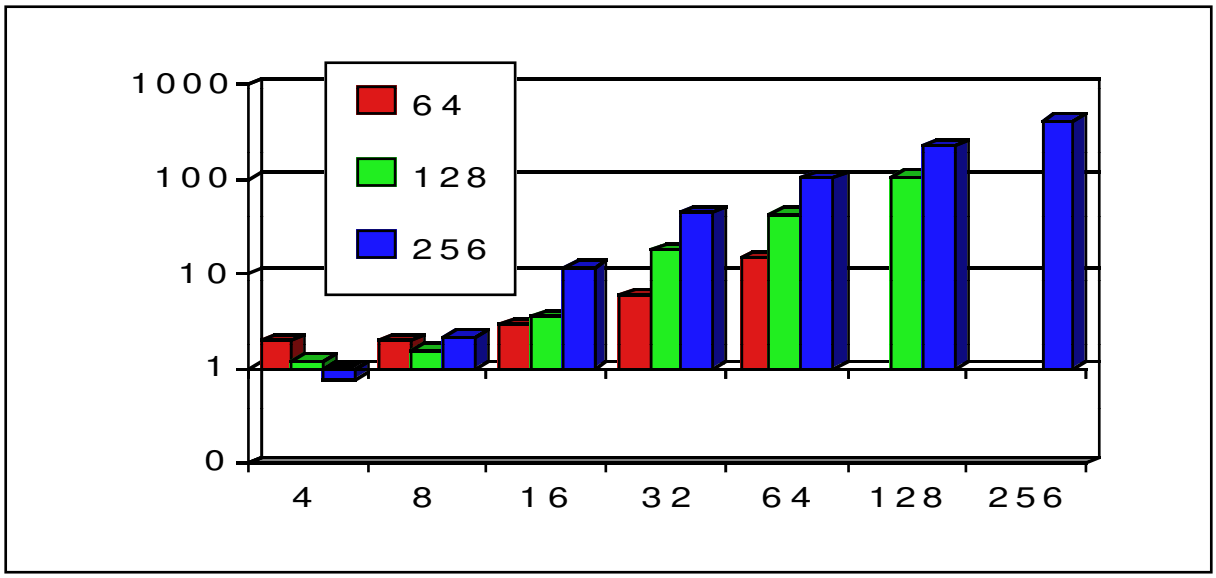

Chart 8: data of Table 14

As the table shows, our code is slower than Cplex (and sometimes than PPRN also) of up to a factor of 6 for problems that have a large n / k ratio, i.e. few commodities w.r.t. the size of the graph: to better understand this phenomenon, it may be worthwhile to report some more disaggregated data about some "critical" 256-nodes problems.

k	m	type	*MMCFB*	*Cplex*	*PPRN*	*LOQO*	*IPCplex*	***Cplex %***	***PPRN %***
4	779	s / e	1.25	0.70	1.86	8.24	6.49	0.56	1.49
4	781	s / h	9.25	1.86	7.13	10.76	8.95	0.20	0.77
4	2021	d / e	8.24	2.36	9.37	83.37	67.01	0.29	1.14
4	2021	d / h	61.06	7.87	43.18	100.43	79.01	0.13	0.71
8	795	s / e	2.33	1.95	6.39	83.86	53.78	0.84	2.75
8	798	s / h	16.71	5.07	16.63	119.28	73.19	0.30	1.00
8	2171	d / e	39.94	12.58	68.37	1021.59	676.98	0.31	1.71
8	2177	d / h	147.83	71.47	357.13	1047.36	707.51	0.48	2.42
16	799	s / e	5.15	6.30	29.11	373.20	157.16	1.22	5.65
16	831	s / h	25.31	26.58	80.40	1188.91	588.82	1.05	3.18
16	2280	d / e	71.45	87.69	338.57	9236.64	*	1.23	4.74
16	2303	d / h	226.92	750.20	3443.10	*	*	3.31	15.17
32	816	s / e	11.55	21.56	107.09	769.96	377.17	1.87	9.27
32	820	s / h	61.59	180.09	490.99	3416.71	1571.14	2.92	7.97
32	2331	d / e	112.93	686.41	1287.53	*	*	6.08	11.40
32	2321	d / h	393.23	5306.26	24626.37	*	*	13.49	62.63
64	825	s / e	21.47	120.25	515.45	1336.87	1022.85	5.60	24.00
64	801	s / h	46.89	438.75	1216.78	7322.86	3158.78	9.36	25.95
64	2337	d / e	218.87	6004.34	7159.48	*	*	27.43	32.71
64	2330	d / h	417.20	24599.80	64852.23	*	*	58.96	155.45

Table 15: disaggregated data about some 256-nodes problems

As a first observation, we remark the fact that "hard" problems are significantly harder than "easy" ones, for all the codes, up to over an order of magnitude for problems of the same size: moreover, they are also much harder than e.g. Canad problems of comparable size, since a 160.000 variables (100, 100, 1600) Canad problem can be solved in 1.5 seconds, while an hard 150.000 variables (64, 256,

2300) Mnetgen problem may take over 500 seconds to be solved. Moreover, as the table shows, MMCFB is 3 times slower than Cplex for easy problems, but over 6 times slower on hard ones as long as the number of commodities is small: conversely, when the number of commodities increase the *relative efficiency* (indicated with "Cplex %" and "PPRN %") of our code in neatly larger on hard problems than on easy ones, and it goes increasing with the graph size and the commodities number. This is probably due to the fact that the number of φ() evaluations required to solve a problem depends on the "hardness" of the problem, but much less on the number of commodities, while the number of simplex iterations needed probably tends to increase under the combined effect of *m* and *k*: there may be however other characteristics playing a role, such as the influence of the percentage of arcs having single-commodity upper bounds (*s*), that is illustrated in the following table.

k	*m*	*b*	*s*	*MMCFB*	*Cplex*	*PPRN*
4	774	610	30%	16.34	2.95	10.39
4	784	511	60%	8.79	1.73	7.83
4	785	185	90%	2.63	0.89	3.16
4	2027	1526	30%	110.19	12.58	60.91
4	2013	1212	60%	56.92	7.82	48.88
4	2023	537	90%	16.07	3.22	19.75
64	412	338	30%	23.48	71.89	208.79
64	406	328	60%	14.11	55.7	187.29
64	397	319	90%	6.22	70.51	192.1
64	1182	946	30%	219.16	3087.16	6488.83
64	1201	955	60%	114.11	4402.39	7453.66
64	1171	896	90%	125.01	4864.84	9535.96

Table 16: even more disaggregated data about some 256-nodes problems

In general, having more single-commodity upper bounds tends to make the problem easier, since it makes more probable that mutual upper bounds be redundant: in fact, although the number of capacity constraints that are generated by Mnetgen are fixed with a triplet (*n*, *k*, *m*), more and more such constraints are found useless (especially on sparse problems) by the preprocessing phase when *s* increases, so that the actual the number *b* of mutually capacitated arcs tends to decrease as *s* grows.
Having more capacitated arcs means having more Lagrangean multipliers, hence it is not strange that the running time of our algorithm is largely influenced from *b*, and therefore from *s*: for the simplex-based codes, however, the same effect is visible only until the number of commodities is small, but when *k* increases the running times go worse as *s* increases. This is presumably due to the fact that having more single-capacity constraints means having polytopes with a larger number of facets, where the number of simplex iterations tends to grow, cancelling the positive effects of having a simpler LP (with few rows). The fact that two different simplex-like method have a very similar behaviour with this respect seem to indicate that this may be a *structural* characteristics of this class of algorithms, i.e. that decomposition methods may *inherently* tend to have better performances on many-commodities problems with a large percentage of single-commodity capacitated arcs.

The PDS problems

We ran tests on the first 33 PDS problems (all the first fifteen, and then one every three), with sizes ranging to 4000 to 200000 variables. Starting with PDS15, the two IP-based methods began having

serious memory problems (trashing) that made it impossible to solve the large instances: however, even for the smaller ones, they had been always one order of magnitude slower than the simplex-based codes. Table 17 and Chart 9 show the the results obtained by all the five algorithms, averaged in groups of three the first fifteen:

	k	*n*	*m*	*b*	*Size*	*MMCFB*	*Cplex*	*PPRN*	*LOQO*	*IPCplex*
PDS1-3	11	256	779	190	8565	6.49	2.75	7.53	41.62	20.40
PDS4-6	11	687	2314	557	25454	102.38	24.79	68.73	744.74	234.76
PDS7-9	11	1109	3692	920	40608	317.58	107.95	221.79	2518.71	1318.96
PDS10-12	11	1544	5366	1298	59030	856.60	301.92	509.11	8347.95	5322.93
PDS13-15	11	1981	7159	1684	78753	1632.95	575.18	1233.85	17478 *	9279 *
PDS18	11	2558	9589	2184	105479	2237.47	2132.57	3093.51	*	*
PDS21	11	2996	11401	2553	125411	3541.34	5823.80	7475.30	*	*
PDS24	11	3419	13065	2893	143715	5796.18	11936.50	10817.80	*	*
PDS27	11	3823	14611	3201	160721	8761.89	28790.40	14670.20	*	*
PDS30	11	4223	16148	3491	177628	10343.20	51011.50	18995.00	*	*
PDS33	11	4643	17840	3829	196240	15459.30	54351.30	28869.60	*	*

Table 17: results for the PDS problems

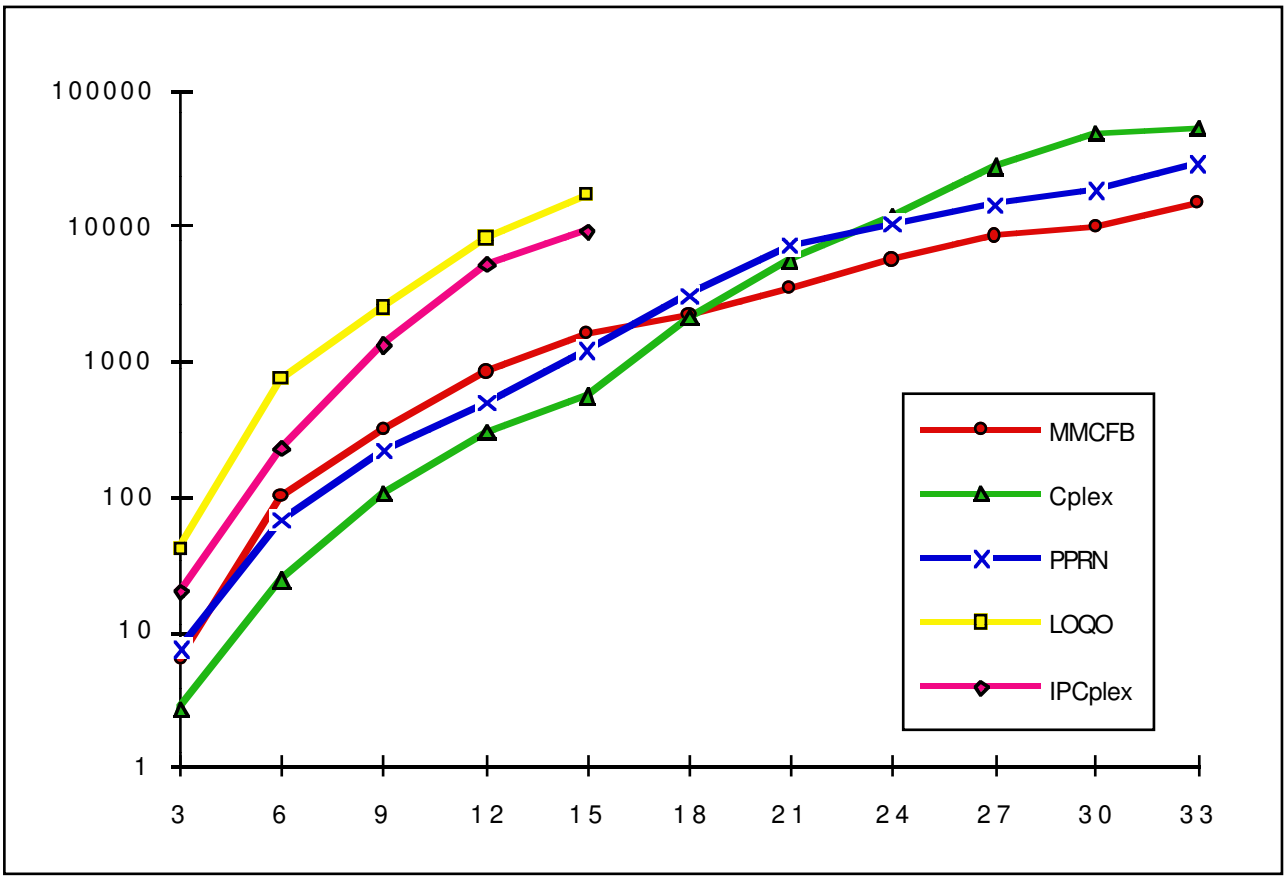

Chart 9: data of Table 17

On problems with less than 100000 variables, Cplex outperforms both PPRN and our code, the latter of even more than an order of magnitude: however, as the size of the problem increases the gap lessens, and for the largest instances MMCFB outperforms Cplex of a factor of four. It is also interesting to note that around the size of 150000 variables PPRN become competitive with Cplex, and

for larger instances the specialized simplex is better than the generic one of over a factor of two. For what has been shown in §3, our algorithm may become competitive earlier, and acheive larger speedups, on instances with analogous characteristics but SPT-structured subproblems: however, since we believe that the PPRN code can be quite easily enhanced to obtain significantly better running times, it is not clear whether or not our cost decomposition methods can obtain really large (say an order of magnitude) speedups w.r.t. simplex-based methods, as it happened for the Mnetgen problems.

It may be instructive to compare the behaviour of the algorithms on Mnetgen and PDS problems of the same size: in Table 18, results of some Mnetgen problems are compared with those of PDS instances with about the same number of LP variables. The Mnetgen instances are both of sparse (.s) and of dense (.d) type, and may be compared with the PDS*i* problems that also have a sparsity (m / n) of about 3, and slowly growing with i: for each class of Mnetgen problems, the aggregated results of both of hard (.h) and easy (.e) instances are reported. The table clearly shows that running times of the simplex-based methods on the PDS problems are always not larger, and often significantly smaller, than on the corresponding hard Mnetgen instances, but for the PDS33 where Cplex is signficantly slower: conversely, the MMCFB code can take almost two orders of magnitude more time to solve the PDS problems than the corresponding easy Mnetgen instances. Since the PDS problems are usually considered difficult MMCFs, we can assert that our code can is particularly effective on solving (hard) Multicommodity problems as long as the m / k ratio is "small enough", but it is still competitive on large problems, even when the number of commodities is small.

name	*k*	*n*	*Size*	*MMCFB*	*Cplex*	*PPRN*
16.256.s.e	16	256	12789	5.15	6.30	29.11
16.256.s.h	16	256	13301	25.31	26.58	80.40
PDS3	11	390	13398	13.10	5.32	15.88
32.256.s.e	32	256	26123	11.55	21.56	107.09
32.256.s.h	32	256	26229	61.59	180.09	490.99
PDS5	11	686	25575	127.03	19.54	60.16
64.256.s.e	64	256	52779	21.47	120.25	515.45
64.256.s.h	64	256	51264	46.89	438.75	1216.78
PDS10	11	1399	52712	928.14	207.64	467.49
64.256.d.e	64	256	149547	218.87	6004.34	7159.48
64.256.d.h	64	256	149120	417.20	24599.80	64852.23
PDS24	11	3419	143715	5796.18	11936.50	10817.80
256.256.s.e	256	256	210859	182.00	4457.03	16150.89
256.256.s.h	256	256	212480	329.48	16222.97	27625.40
PDS33	11	4643	196240	15459.30	54351.30	28869.60

Table 18: Mnetgen and PDS problems of the same size

The JLF problems

Table 19 shows the results obtained on some of the problems of the JLF data set, i.e. those with a SPT structure: it is interesting to note that, despite the small number of commodities, our algorithm is still competitive in most cases, especially (as expected) for easy problems, such as the "Assad" set.

	k	n	m	b	Size	MMCFB	Cplex	PPRN	LOQO	IPCplex
10term	10	190	510	146	5100	0.27	0.84	1.07	26.17	10.97
10term.0	10	190	507	143	5070	0.23	0.78	0.7	28.29	13.1
10term.50	10	190	498	134	4980	2	0.82	1.22	18.05	11.6
10term.100	10	190	491	127	4910	2.4	0.86	1.1	19.21	10.58
15term	15	285	796	253	11940	4.81	3.39	11.57	210.94	106.98
15term.0	15	285	745	202	11175	1.94	2.72	10.23	123.21	47.15
Chen0	4	26	117	43	468	0.3	0.25	0.32	0.44	0.38
Chen1	5	36	174	65	870	0.98	0.64	0.73	0.99	0.77
Chen2	7	41	358	155	2506	5.98	4.89	3.77	4.17	3.91
Chen3	15	31	149	56	2235	1.22	5.18	1.96	3.12	2.62
Chen4	15	55	420	176	6300	25.49	83.39	20.05	46.92	32.75
Chen5	10	65	569	242	5690	52.58	48.74	48.12	21.18	15.53
assad1.5k	3	47	98	98	294	0.03	0.11	0.09	0.39	0.22
assad1.6k	3	47	98	98	294	0.01	0.08	0.06	0.37	0.22
assad3.4k	6	85	204	95	1224	0.08	0.33	0.55	4.26	1.44
assad3.7k	6	85	204	95	1224	0.1	0.29	0.56	4.22	1.41

Table 19: results for some of the JLF problems

However, the small number and size of problems makes it difficult to draw conclusions: for instance, there are cases (the Chen5 problem) where IP-based methods, that are usually slow, largely outperform all the others, but the reasons are not clear. Actually, the only sensible observation that can be made is that, on these data sets, there does not seem to be any dominant code, so that other consideration (such as availability, ease-of-use, reoptimization etc) should be taken into account: in other words, since such small problems are well in the range of modern general-purpose LP solvers, it may make no sense, at least from a practical point of view, to use sophisticated ad-hoc technology where available standard techniques may do the same work almost as efficiently.

Conclusions

The above computational experience seem to show that our code (hence, in general, carefully designed cost-decomposition algorithms) can be a valuable mean of solving MMCF problems with many commodities: to underline a just simple figure, solving the hard, dense (256, 256) problems took on average less than 20 *minutes* of CPU with our code, but it would take more than 6 and 10 *days* of CPU with Cplex and PPRN respectively, even if the first would not have memory problems. This is interesting in practice, since many of the MMCF models appearing in the literature have this feature, and presumably the utilisation of such models have since now been prevented by the lack of reasonably efficient solvers. Conversely, our cost decomposition method may not be the best choice for problems with a small number of commodities, especially if not having a SPT structure, unless on very large instances: the very large PDS problems show that our method can still be competitive on problems with a m / k ratio of over 1000, but the speedup w.r.t. the simplex based codes is far from the ones that can be obtained on many-commodities problem.

Among the other codes we tested, IP-based generic LP solvers seem (to our surprise) to be uniformly slower than the other approaches, and to require much more memory so to make it impossible to solve even moderately-sized instances: however, we must stress the fact that none of the two codes is a specialized one, so that this does not tell much on the overall value of IP methods for the solution of

MMCF problems - actually, due to their excellent behaviour in terms of iteration counts, specialized method that reduce time/memory requirements for the matrix operations might become competitive in a near future. As for the simplex based methods, Cplex seems to be faster than PPRN, although not dramatically, on the difficult Mnetgen problems, while the converse is true for the easier Canad ones: however, the value of PPRN can be judged by its behaviour on the difficult PDS problems, where it neatly outperforms Cplex on the largest instances. Furthermore, since the latter is a sophisticated, commercial implementation while the former is an academic code, we believe that the performances of PPRN could be consistently improved, making it especially attractive for few-commodities problems, where cost decomposition methods seems to be less effective.

Despite the "folklore" about their most famous representative, the Dantzig-Wolfe algorithm, decomposition algorithms have then shown to be a promising approach for the continuous Multicommodity Min Cost Flow problem: by making use of state-of-the-art code for the subproblems to be solved at each step and a specialized Bundle algorithm as "main logic", we have obtained relevant savings w.r.t. some of the best generic LP codes and even w.r.t. alternative specialized approaches such as primal partitioning methods, efficiently solving problems with hundreds of commodities and hundreds of thousands of variables. A further advantage of such methods, exploited in our code, is that the development of a decomposition code is naturally spliced into independent modules: any advance in MCF or SPT technology, refined QP solvers or even alternative NDO algorithms can be very easily exploited to reduce the overall running time of the algorithm.
Future research in this field will comprise:

- extension of the approach to allow the introduction of extra constraints (often needed in real applications): this can be easily done since any additional constraint can be dualized together with the global capacity ones, but increasing the number of Lagrangean multipliers can have an adverse effect on the performances, that might have to be faced with ad-hoc strategies
- development of new subproblem solvers and refinement of the current ones, for instance by implementing reoptimization of the SPT algorithms rather than solving each subproblems from scratch at each iteration
- parallelization of the code on coarse-grained massively parallel architectures

Acknowledgements

The Authors wish to acknowledge all the people that, in various forms, have contributed to the research described in the present papers. Among them, we mention Jordi Castro Perez and Prof. Robert J. Vanderbei for giving us their codes and helping us in the setup, Prof. Irv Lustig for his precious advises and explanations about the JL input format, Prof. Stavros Zenios and Michel Gendreau for their kind interest in the matter. Moreover, our acknowledgement is also for Prof. Bernard Gendron, Joel Andrew Shapiro, R. Sarkissian and again Jordi Castro Perez for their help in obtaining the data sets.

References

[AHK79] A. ALI, R. HELGASON, J. KENNINGTON AND H. LALL "**Computational Comparison among Three Multicommodity Network Flow Algorithms**" *Oper. Research*, **28**(4), p 995-1000, 1979

[AK77] A.I. ALI , J. KENNINGTON "**MNETGEN program documentation**" *Technical Report IEOR* **77003**, Department of Industrial Engeneering and Operations Research, Southern Methodist University, 1977

[AKL93] A.I. ALI, J.L. KENNINGTON AND T.T. LIANG "**Assignmen with En-Route Training of Navy Personnel**" *Nav. Res. Log.* **40**, p. 581-592, 1993

[AKS88] A.I. ALI , J. KENNINGTON AND B. SHETTY "**The Equal Flow Problem**" *EJOR*, **36**, p. 107-115, 1988

[AMO93] R.K. AHUJA, T.L. MAGNANTI AND J.B. ORLIN "**Network Flows**" *Prentice Hall*, 1993

[As78] A.A. ASSAD "**Multicommodity Network Flows - A Survey**" *Networks*, **8**, p. 37-91, 1978

[Ba93] C. BARNHART "**Dual Ascent Methods for Large-Scale Multicommodity Flow Problems**" *Nav. Res. Log.*, **40**(3), p. 305-324, 1993

[Be91] D.P. BERTSEKAS "**Linear Network Optimization**" *The MIT Press*, 1991

[BHJ95] C. BARNHART, C.A. HANE, E.L. JHONSON AND G. SIGISMONDI "**A Column Generation and Partitioning Approach for Multicommodity Flow Problems**" *Telecommunication Systems*, **3**, p. 239-258, 1995

[BL84] S.N. BHATT, T. LEIGHTON "**A Framework for Solving VLSI Layout Problems**" *J. Comp. ans Syst. Sci.*, **28**(2), p. 300-343, 1984

[BS93] C. BARNHART, Y. SHEFFY "**A Network-Based Primal-Dual Heuristic for Multicommodity Network Flow Problems**" *Trans. Science*, **27**(2), p. 102-117, 1993

[Ca94] J. CASTRO "**PPRN 1.0 User's Guide**" *DR*, **94/06**, Statistics & Operations Research Dept., Univeristat Politècnica de Catalunya, 1994

[CD74] H. CHEN, C.G. DEWALD "**A Generalized Chain Labelling Algorithm for Solving Multicommodity Flow Problems**" *Comput. & Op. Res.*, **4**, p. 437-465, 1974

[CFN95] P. CARRARESI, A. FRANGIONI AND M. NONATO **"Applying Bundle Methods to Optimization of Polyhedral Functions: An Applications-Oriented Development"** *Technical Report* **TR 17/95**, Dipartimento di Informatica, Università di Pisa, 1995

[Cp94] CPLEX OPTIMIZATION "**Using the CPLEX® Callable Library**" *Version* **3.0**, CPLEX Optimization Inc., 1994

[CN95] J. CASTRO, N. NABONA "**An Implementation of Linear and Nonlinear Multicommodity Network Flows**", 1995 (to appear on *EJOR*)

[DP95] P. DHARMA, J.A. SHAPIRO "**Augmented Lagrangeans for Linearly Constrained Optimization Using AMPL**" *CIV* **518** Final Report, 1995

[FPL93] J.M. FARVOLDEN, W.B. POWELL AND I.J. LUSTIG "**A Primal Partitioning Solution for the Arc-Chain Formulation of a Multicommodity Network Flow Problem**" *Op. Res.*, **41**(4), p. 669-693, 1993

[Fr95] A. FRANGIONI "**Solving Semidefinite Quadratic Problems Within Nonsmooth Optimization Algorithms**" *Technical Report* **TR 10/95**, Dipartimento di Informatica, Università di Pisa, 1995 (to appear on *Computers & O.R.)*

[GC95] B. GENDRON, T.G. CRAINIC "**Bounding Procedures for Multicommodity Capacitated Network Design Problems**" *Publication* **CRT-965**, Centre de Recherche sur les Transports, Université de Montreal, 1995

[GG71] A.M. GEOFFRION, G.W. GRAVES "**Multicommodity Distribution System by Benders Decomposition**" *Mgmt. Sci.*, **20**(5), p. 822-844, 1971

[GVY93] N. GARG, V.V. VAZIRANI AND M. YANNAKAKIS "**Approximate Max-Flow Min-(Multi)Cut Theorems and their Applications**" *Proc. 25th STOC*, p. 698-707, 1993

[HL81] J.K. HO, E. LOUTE "**An advanced implementation of the Dantzig Wolfe decomposition algorithm for linear programming**" *Math. Prog.*, **20**(1), p. 303-326, 1981

[HL83] J.K. HO, E. LOUTE "**Computational Experience with Advanced Implementation of Decomposition Algorithms for Linear Programming**" *Math. Prog.*, **27**(3), p. 283-290, 1983

[HL93] J-B. HIRIART-URRUTY, C. LEMARÉCHAL "**Convex Analysis and Minimization Algorithms II**" *A Series of Comprehensive Studies in Mathematics*, vol. **306**, Springer-Verlag, 1993

[JL92] K.L. JONES , I.J. LUSTIG "**Input Format for Multicommodity Flow Problems**" Program in Statistics and Operations Research, Department of Civil Engeneering and Operations Research, Princeton University, 1992

[JLF93] K.L. JONES , I.J. LUSTIG, J.M. FARVOLDEN AND W.B. POWELL "**Multicommodity Network Flows: The Impact of Formulation on Decomposition**" *Math. Programming*, **62**, p. 95-117, 1993

[KAR90] P. KLEIN, A. AGRAWAL, R. RAVI AND S. RAO "**Approximation Through Multicommodity Flow**" *Proc. 31th FOCS*, p. 726-727, 1990

[Ke78] J.L. KENNINGTON "**A Survey of Linear Cost Multicommodity Network Flows**" *Op. Res.*, **26**(2), p. 209-236, 1978

[Ke79] J.L. KENNINGTON "**A Primal Partitioning Code for Solving Multicommodity Flow Problems**" *Technical Report* **79008**, Department of Industrial Engeneering and Operations Research, Southern Methodist University, 1978

[Ki89] K.C. KIWIEL "**A Survey of Bundle Methods for Nondifferentiable Optimization**" *in Mathematical Programming and its Applications* (M. Iri and K. Tanabe eds.), KTK Scientific Publishers, 1989

[KKR94] A.P. KAMATH, N.K. KARMARKAR AND K.G. RAMAKRISHNAN "**Computational and Complexity Results for an Interior Point Algorithm on Multicommodity Flow Problems**" *unpublished manuscript* , 1994

[KS77] J. KENNINGTON, M. SHALABY "**An Effective Subgradient Procedure for Minimal Cost Multicommodity Fow Problems**" *Mgmt. Sci.*, **23**, p. 994-1004, 1977

[Le89] C. LEMARÉCHAL "**Nondifferentiable Optimization**" *in Handbooks in Opns. Res. and Mgmt. Sci., vol. 1 : Optimization* (G.L. Nehmauser, A.H.G. Rinnoy Kan and M.J. Todd eds.), North-Holland, 1989

[LGM93] R. DE LEONE, M. GAUDIOSO AND M.F. MONACO "**Nonsmooth Optimization Methods for Parallel Decomposition of Multicommodity Flow Problems**" *Annals of O.R.*, **44**, p. 299-311, 1993

[LPS93] T. LEONG, P. SHOR AND C. STEIN "**Implementation of a Combinatorial Multicommodity Flow Algorithm**" *Proc. DIMACS Conference, New Brunswick*, 1993

[Me94] D. MEDHI "**Bundle-Based Decomposition for Large-Scale Convex Optimization: Error Estimate and Application to Block-Angular Linear Programs**" *Math. Prog.*, **66A**(1), p. 79-102, 1994

[Po92] G.G. POLAK "**On a Parametric Shortest Path Problem from Primal-Dual Multicommodity Network Optimization**" *Networks*, **22**, p. 283-295, 1992

[PS83] W.B. POWELL, Y. SHEFFI "**The Load Planning Problem of Motor Carriers: Problem Description and a Proposed Solution Approach**" *Trans. Res.*, **17A**, p. 471-480, 1983

[PT93] S.A. PLOTKIN, É. TARDOS "**Improved Bounds on the Max-Flow Min-Cut Ratio for Multicommodity Flows**" *Proc. 25th STOC*, p. 690-697, 1993

[PZ94] M.C. PINAR, S.A. ZENIOS "**On Smoothing Exact Penalty Functions for Convex Constrained Optimization**" *SIAM J. Optimization*, **4**(3), p. 486-511, 1994

[SM91] G.L. SHULTZ, R.R. MEYER "**An Interior Point Method for Block-Angular Optimization**" *SIAM J. on Optimization*, **1**(4), 1991

[SZ92] H. SCHRAMM, J. ZOWE "**A Version of the Bundle Idea for Minimizing a Nonsmooth Function: Conceptual Idea, Convergence Analysis, Numerical Results**" *SIAM J. OPtimization*, **2**(1), p. 121-152, 1992

[Ta94] É. TARDOS "**Approximate Min-Max Theorems and Fast Approximation Algoritms for Multicommodity Flow Problems**" *unpublished manuscript*, 1994

[Va92] R.J. VANDERBEI "**LOQO User's Manual**" *SOR* **92-05**, Princeton University, 1992

[Va94] R.J. VANDERBEI "**LOQO: An Interior Point Code for Quadratic Programming**" *SOR* **94-15**, Princeton University, 1994

[VC93] R.J. VANDERBEI, T.J. CARPENTER "**Symmetric Indefinite Systems for Interior Point Methods**" *Math. Prog.*, **58**, p. 1-32, 1993

[ZTT84] A. ZAHORIK, L.J. THOMAS AND W.W. TRIGEIRO "**Network Programming Models for Production Scheduling in Multi-Stage, Multi-Item Capacitated Systems**" *Mgmt. Sci*, **30**, p. 308-325, 1984