II.2 A Bundle Approach

In this Chapter, extensive computational experiences with a Bundle-type Cost Decomposition approach to (MMCF) are presented: to the best of our knowledge, only one attempt [Me90] [Me94] had already been done to use Bundle methods in this context, and just making use of pre-existent (and partly outdated) NDO software. A specialized Bundle code for (MMCF) has been developed, as a part of the larger research described in Part I, whose main characteristics are

- a simple "main" logic, based on a new heuristic for the t parameter handling;
- use of the efficient BTT (QP) solver, that allows a Lagrangean Variables Generation strategy;
- a "polymorphic" Lagrangean Relaxation solver, using specialized (and efficient) codes for the solution of the different types of single-commodity Min-Cost Flow subproblems.

The code was developed for being subsequently used - if found efficient - within approaches to more complex problems requiring the solution of (MMCF) as a subroutine: however, many interesting issues related to this usage, such as efficiency of the reoptimization in case of cost / capacity changes, have not yet been faced. The relevant details of the implementation are presented in §II.2.1, while §II.2.2 describes the other (specialized or not) codes against which it has been compared on the data sets described in §II.2.3: the computational results are presented and discussed in §II.2.4.

II.2.1 Implementation details

The MMCFB code follows the general scheme for Bundle algorithms presented in Part I, using a simple implementation that is described in Figure 9. At each iteration, the *predicted increase* $v = \varphi_\beta(\boldsymbol{d})$ is compared with the *obtained increase* $\Delta\varphi = \varphi(\boldsymbol{y}) - \varphi(\bar{\boldsymbol{y}})$, and the current point is moved to $\boldsymbol{y}$ only if a sufficient (relative) increase has been obtained. Otherwise, the "aggregated linearization error" σ is used to test the "reliability" of the newly obtained information: if α is not too large than σ, then the addition of $\boldsymbol{g}$ to β is expected to consistently improve the "accuracy" of $\varphi_\beta()$, otherwise t is decreased, forcing the next tentative point to be "closer" to $\bar{\boldsymbol{y}}$, where $\varphi_\beta()$ is (hopefully) accurate enough. The IncreaseT() and DecreaseT() functions are described in Figure 10, and are respectively based on the *(b)* and *(a)* formulae of §I.5.1: exploiting the (directional) derivative in $\boldsymbol{y}$ rather than in $\bar{\boldsymbol{y}}$ seems to be more reasonable when a step to $\boldsymbol{y}$ has already been decided, and the IncreaseT() heuristic guarantees that t will be increased (if and) only if $\boldsymbol{dg} > 0$, as in the ~SS(*ii*) condition of BT (Cf. §I.5.2). On the contrary, using "near" information might be better if $\varphi_\beta()$ is already known to be a "poor" approximation of φ() at $\boldsymbol{y}$: using the two different heuristics seems to slightly reduce, on the average, the iterations count.

MMCFB

$\langle$ choose $m_1 \in [0, 1]$, $0 < m_2$, $t_0 > 0$, $t^* > 0$ and $\varepsilon > 0$ $\rangle$;
$\bar{\boldsymbol{y}} = \boldsymbol{0}$; $t = t_0$; $\langle$ calculate $\varphi(\bar{\boldsymbol{y}})$ and $\boldsymbol{g}(\bar{\boldsymbol{y}})$ $\rangle$; $\beta = \{ (\boldsymbol{g}(\bar{\boldsymbol{y}}) , 0) \}$;
repeat
    $\langle$ β-strategy: eliminate outdated subgradients $\rangle$;
    $\langle$ solve $(\Delta'_{\beta t})$ and $(\Pi'_{\beta t})$ for θ and $\boldsymbol{d}$ $\rangle$; $\sigma = \alpha_\beta\theta$;
    if($t^*\| \boldsymbol{d} \|^2 + \sigma \leq \varepsilon\varphi(\bar{\boldsymbol{y}})$)
        then STOP;
        else $\boldsymbol{y} = \bar{\boldsymbol{y}} + \boldsymbol{d}$;
            $\langle$ calculate $\varphi(\boldsymbol{y})$, $\boldsymbol{g}(\boldsymbol{y})$ and the corresponding α $\rangle$;
            $\langle$ add ($\boldsymbol{g}$, α) to β $\rangle$;
            if($\Delta\varphi = \varphi(\boldsymbol{y}) - \varphi(\bar{\boldsymbol{y}}) \geq m_1\varphi_\beta(\boldsymbol{d})$)
                then $\bar{\boldsymbol{y}} = \boldsymbol{y}$; $\alpha = \alpha + \boldsymbol{d}\boldsymbol{G}$; t = IncreaseT();
                else **if**($\alpha > m_2\sigma$)
                        then t = DecreaseT();
until(STOP);

Figure 9: the "main" of the MMCFB code

$\langle$ let $M > 1$, $m < 1$ and $t_M \geq t_0 \geq t_m$ be fixed $\rangle$;

IncreaseT() = $\max\{ t , \min\{ t_M , Mt , 2tv(v - \Delta\varphi) \} \}$

DecreaseT() = $\min\{ t , \max\{ t_m , mt , (\alpha + \Delta\varphi) / 2\alpha \} \}$

Figure 10: the heuristic t-strategies

Since no "Long Term" t-strategy is implemented, MMCFB is prone to the risk of the "small t" phenomenon (Cf. §I.5.3): however, this is avoided in practice by setting $m_2 \approx 3$, a much larger value than the .9 suggested in [Ki89] and [SZ92]. As a result, the value of t never changes after the first few iterations: therefore, a "good" choice of t_0 is important, altough the heuristics usually suffice to remedy to blatantly wrong settings. In fact, a good guess for t_0 is usually just any value in the order of magnitude of the t's showing up in the first few iterations: this is also the way in which t^* can be chosen, say an order of magnitude larger than t_0. The choice of the other parameters is far less critical: m_1 seems to be properly fixed to 0.1, the absolute safeguards on t should never enter to play (this is easily obtained in the same way as for t_0 and t^*) and the increasing / decreasing factors seem to be properly fixed to 10 and .1 respectively.

An important characteristic of MMCFB is the *Lagrangean Variables Generation strategy*: only the Lagrangean variables that are really necessary, i.e. those that at some iteration have a strictly positive entry of the direction $\boldsymbol{d}$, are explicitly "created" in the (QP) solver. This approach can be seen from a primal viewpoint as a row generation strategy, where a constraint is added only when it is violated by the aggregate primal solution corresponding to $\boldsymbol{d}$: clearly, this is effective only thanks to the *two-level* *active set* strategy of BTT, that efficiently supports the on-line *creation and destruction of variables* of $(\Pi'_{\beta t})$. Especially for "easy" problems, few Lagrangean variables ever become nonzero, so that the size of $(\Pi'_{\beta t})$ can be kept much smaller than m: furthermore, the set of active Lagrangean variables is very stable after the first few iterations, so that "pricing" - determining whether new variables have to be generated - can be done with a low frequency.

Some other details of the Bundle-specific part of the implementation might be of interest:

- the β-strategy consists in eliminating the subgradients that have zero θ_i multiplier for more than a fixed number ($\approx$ 20) of consecutive iterations: no aggregation is ever performed, hence no limitations are imposed (in principle) on the Bundle size, but the β-strategy suffices to keep it small enough, and the computational experience showed that severe limitations on the size of β have a negative effect on the convergence speed;
- "pricing" in the LVG strategy is done for all the first p_1 ($\approx$ 30) iterations, and then once every p_2 ($\approx$ 10) iterations: it is not computationally convenient to eliminate an "active" variable y_i once it has been generated, so that the active set is monotonically nondecreasing;
- upon termination, MMCFB can report the "full" optimal solution

  $$[\,\bar{\boldsymbol{x}}^1 \,..\, \bar{\boldsymbol{x}}^k\,] = \Sigma_{i\,=\,1..l}\,[\,\boldsymbol{x}_i^1 \,..\, \boldsymbol{x}_i^k\,]\theta_i\,,$$

  where θ is the optimal solution of the last $(\Delta'_{\beta t})$, at the price of the extra memory required for storing all the flows $[\,\boldsymbol{x}_i^1 \,...\, \boldsymbol{x}_i^k\,]$ corresponding to subgradients that are still in β: however, performances in time are essentially not affected, and furthermore the *aggregate* optimal solution $\bar{\boldsymbol{x}}_A = \Sigma_{h\,\in\,\mathcal{K}}\bar{\boldsymbol{x}}^h$ can be recovered at essentially no cost as $\boldsymbol{u} - \boldsymbol{G}_\beta\theta$.

In most cases, and especially if k is large w.r.t. m, most of the time is spent in solving the Min-Cost Flow subproblems: hence, having a fast (MCF) solver capable to efficiently reoptimize after cost changes is obviously important. However, for some instances the subproblems are in fact Shortest Path (Tree) problems, that can be solved much faster than general (MCF)s: therefore, any general-purpose Cost Decomposition (MMCF) solver should have specialized solvers that are invoked according to the properties of each subproblem. In MMCFB, the encapsulation and polymorphism features of C++ are exploited for this purpose: the *abstract base class* `McfClass` defines a standard interface for (MCF) solvers, and *k objects* of proper *derived classes* are constructed within MMCFB according to the properties of each single subproblem, making it easy to deal even with instances with nonuniform subproblems. The *public methods* of `McfClass`, properly redefined in the derived classes, are used for changing the costs, solving the problems and collecting the results: hence, it is easy to extend MMCFB to special classes of instances by just developing the appropriate subproblem solver. Moreover, MMCFB is completely independent from upgrades, or even complete changes, in the (MCF) solver: up to now, two derived classes have been coded, the `RelaxIV` class implementing an efficient Relaxation-type (MCF) solver [Be91] (Cf. also §II.4.3), and the `SPTree` class implementing four classical (SPT) algorithms (queue, dequeue, Dykstra and c-heap [GP88]).

An analogous structure has been used for MMCFB itself: an *abstract base class* `MMCFClass` has been defined, and both MMCFB all the codes described in §II.2.2 has been inserted into *derived classes*, making it possible for an application to use any of them indifferently. Moreover, a service class `Graph` provides a unified mean to read (MMCF) instances (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 `Graph`, making it easy to use an existing (MMCF) solver by only writing small pieces of interface code.

Hence, MMCFB is composed of several independent modules, that are succintly described in Figure 12, where arrows represent functional interactions, thick lines inheritance relationships and dashed lines *virtual* methods redefined in a descendant class. This structure was really useful during the development process, since it made possible to improve the different parts independently and in different instants of time: for instance, the `RelaxIV` class was at a certain point almost entirely rewritten to follow the latest algorithmic developments, and the `SPTree` class was developed much later than `RelaxIV`, when the first instances with (SPT) subproblems had to be solved.

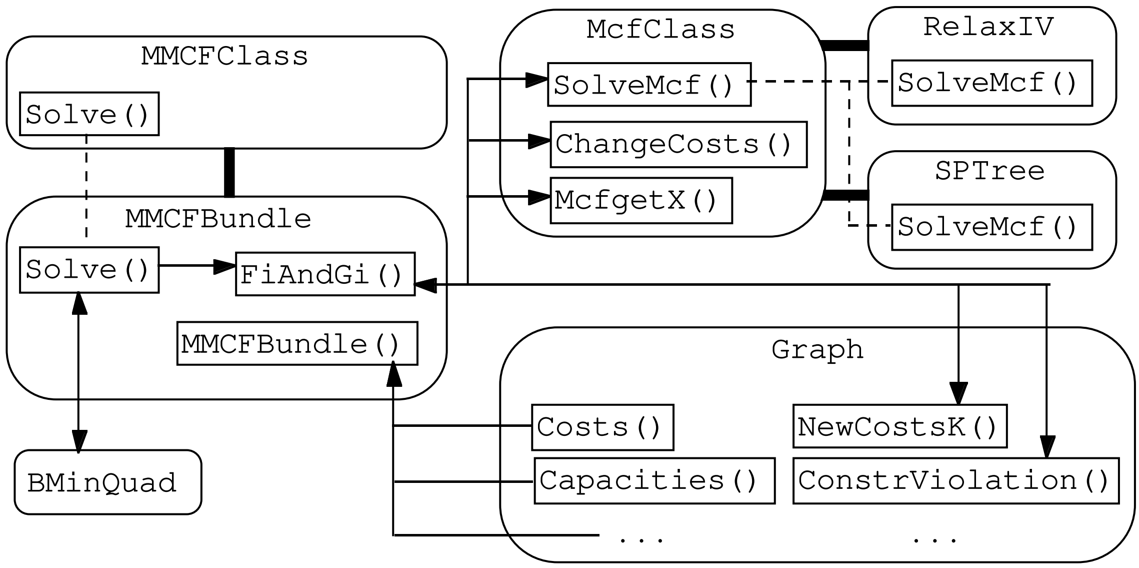

Figure 11: basic structure of the `MMCFClass`, `McfClass` and `Graph` classes

As in the `MinQuad` and `BMinQuad` classes discussed in §I.3.6, there were a number of different alternatives that was not practical to accommodate by means of the inheritance and polymorphism mechanisms, and that were made available in the "compile time switches" fashion: for instance, it is possible to choose among no Lagrangean Variables Generation at all, the standard "monotone" version where no attempt is ever made to delete an active variable and a "full" version where variables are dynamically deleted as well as created. Also, the "abstract" data type mechanism discussed in §I.3.6 was precious in this case: the two types `FNumber` and `CNumber`, designed to hold "flows" and "costs" respectively, can be uniformly changed throughout all the code by modifying a few lines in an appropriate include file, so that, knowing at compile time if flows and/or costs are integer, the (SPT) and (MCF) solver can perform some very basic operations more efficiently, such as all the "$x = 0$" tests that need not to be implemented as "$| x | \leq \varepsilon$".

Although apparently minor, many of these implementation details are critical for the overall efficiency of the code: for instance, the effects of the Lagrangean Variables Generation strategy are shown in Table 1, where the *Rel. Ratio* column shows the attained speedup in the $(\Delta_{\beta'_t})$ solution. That is, the LVG strategy decreases the *syncronization cost* by up to a factor of 5, significantly reducing the *total* execution time (the *Ratio* column): this is especially important for problems with relatively few commodities, where the ($\Delta'_{\beta t}$) cost is relatively high. On the contrary, for problems with many commodities the computation of $\varphi()$ is by far the bottleneck of the algorithm: however, Table 2 shows that, if the subproblems are (SPT)s, then relevant savings on the total execution time can be obtained by using the specialized (relatively naive) solver (*SPT* column) rather then the (sophisticated) generic (MCF) solver (*MCF* column), even on the problems with small graphs and few commodities.

k	*n*	*m*	Ratio	Rel. Ratio
10	50	400	1.47	4.88
10	50	625	1.51	2.72
10	100	1600	1.38	4.22
10	100	2500	1.68	5.18
100	50	535	1.52	2.67
100	30	518	1.72	3.08
100	30	682	1.68	3.02
100	100	1600	1.57	3.73
400	30	520	1.48	2.45
400	30	686	1.50	2.56

Table 1: 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 2: effect of using a (SPT) solver instead of a (MCF) one

However, the clearest example is the specialized BTT algorithm: this is shown in the second set of tables in §I.3.7, where it is compared with a non-specialized Interior-Point (QP) using an early version of MMCFB. The solution of ($\Delta'_{\beta t}$) requires up to 20% of the total running time, and the non-specialized (QP) solver is always at least 1.5 orders of magnitude slower than BTT, clearly making a Bundle code impractical for the excessive cost of syncronization phase: note that the other (QP) code tested in §I.3.7 (on ($\Delta_{\beta t}$) only) is exactly the one used within the previous Bundle approach to (MMCF) [Me94]. Furthermore, these results are prudential, since they do not take into account the extra advantages of the BTT algorithm, such as the ability to efficiently cope with variables creation and deletion that is necessary to implement the LVG strategy.

Other than providing good performances, the modular structure of the code made it easier to enhance and to extended it to different problems: in fact, both the parallel version of Chapter II.3 and the extended version for Fixed Charge problems of Chapter II.5 have been obtained by just specialising some of the classes (`MMCFBundle` and `Graph` respectively), while keeping all the rest unchanged.

II.2.2 The other codes

One of the objectives of this research was to understand where a modern Cost Decomposition code stands, in terms of efficiency, among the other solution methods described in §II.1.4: hence, an effort has been done to compare MMCFB with recent alternative (MMCF) solvers.

II.2.2.1 A general-purpose (LP) solver

In principle, any (LP) solver can be used to solve (MMCF) problems: actually, commercial general-purpose (LP) solvers can be the *best* choice for solving even quite large Multicommodity problems, since they are often the result of an impressive amount of work and experience obtained during many years of development, and exploit the state of the art in matrix factorisation algorithms, pricing rules, sophisticated data structures, preprocessing and software technology.

Cplex 3.0 [Cp94] is one of the best commercial (LP) solvers on the market: it offers Primal and Dual simplex algorithms, a Network simplex algorithm for exploiting the embedded network structure of the problems, and a Barrier (Interior Point) algorithm. As for the other codes, efforts have been made to use Cplex in its full power, by repeatedly solving subsets of the test problems with various combinations of the available optimization parameters in order to find a sensible setting: this process is very important in order to obtain a fair computational comparison, as the following data will show. Cplex can identify the network structure embedded in a (LP), and efficiently solve the restricted (LP) to obtain a good starting base for the "Pure" Primal and Dual (*PP* and *PD*) simplex algorithms: the performances of the resulting "Hybrid" Primal and Dual algorithms (*HP* and *HD*) are reported in Table 3, where *Size* = *mk* is the number of variables of the (LP) and figures are averaged within groups of several problems, and depicted in Chart 1.

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 3: different simplex options for Cplex

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 4: different pricing rules for Cplex

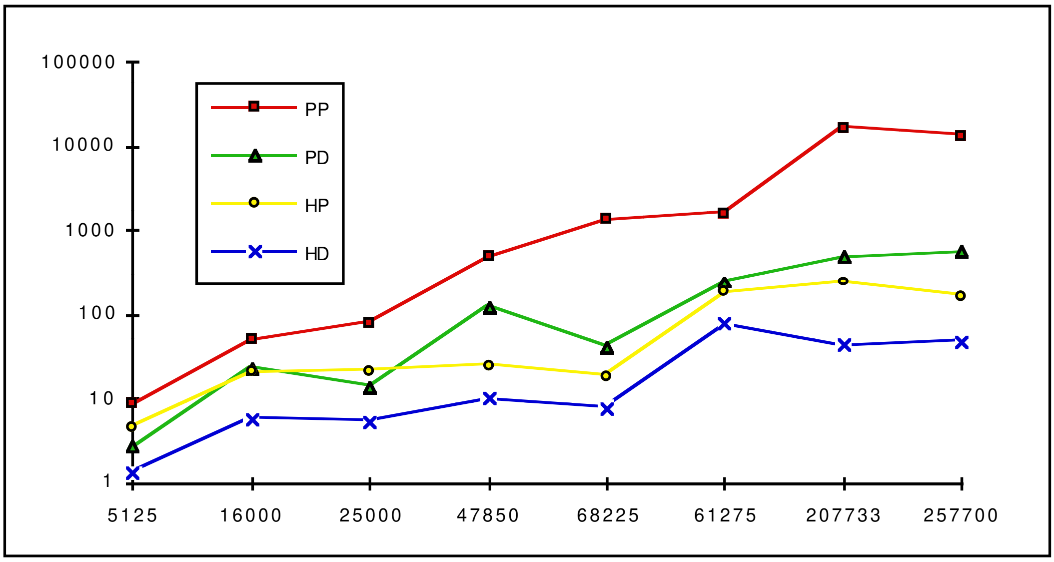

Chart 1: different simplex options for Cplex

For these instances, a Dual approach can be 30 times faster than a Primal one, and the "warm start" speeds up the the solution of up to a factor of 70, so that the Hybrid Dual version can be *400 times* faster than the Pure Primal one: hence, it is not unfair to regard Cplex as a *semi-specialized* algorithm for network-structured problems. Other details, such as the pricing strategy, can have a dramatic impact on the number of iterations and hence on the performances: Table 4 and Chart 2 report a comparison between the four available pricing rules for the Dual simplex, the standard dual pricing *HD1* and three variants of the steepest-edge rule *HD2-4*, showing that performance improvements of up to a factor of 15 (and more, since the instances marked with “*” had taken much longer than 1000 seconds to terminate) can be obtained by just selecting the appropriate rule.

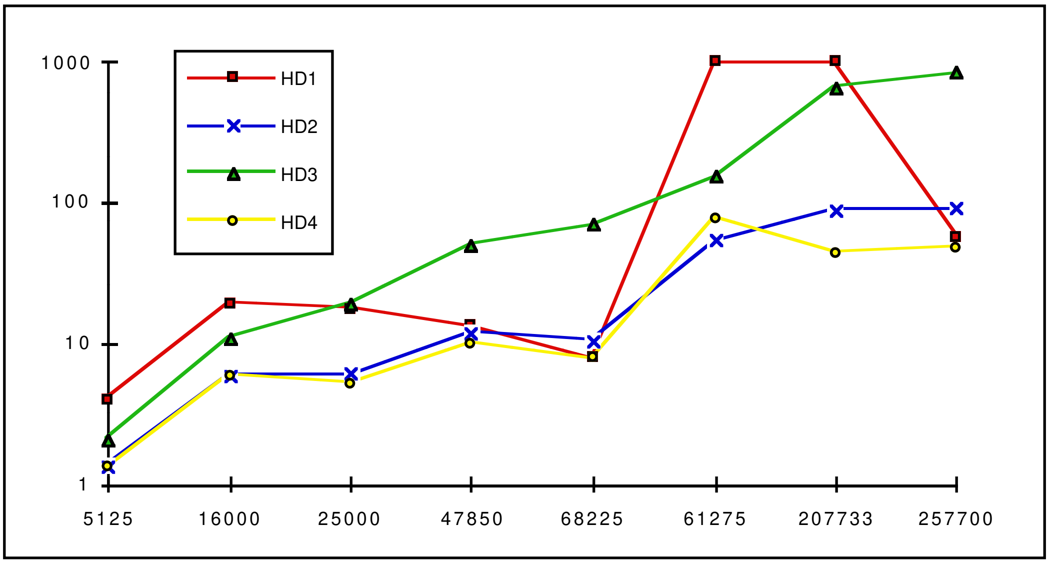

Chart 2: different pricing rules for Cplex

II.2.2.2 A Primal Partitioning code

The PPRN 1.0 [Ca94] code is an available¤ recent implementation of the Reduced Gradient algorithm for the Generalised Nonlinear (MMCF) discussed in Sec. II.1.4.1, i.e. a Primal Partitioning approach when applied to the linear case: in their computational experience [CN95], the authors show that it generally outperforms the known MCNF85 Primal Partitioning code [Ke79], so that it can be regarded as a good representative of this class of methods. As for the other codes, an appropriate tuning of its (several) optimization parameters can significantly enhance the performances: the most important choice concerns the *Phase 0* strategy, i.e. whether the starting base is obtained from any feasible solution of the *k* (MCF)s rather then from their optimal solutions, as in the “warm start” of Cplex. As shown in Table 5 and Chart 3, in this case also exploiting the optimal solutions decrease the running times of up to a factor of 20 (*Opt* vs *Feas* columns): moreover, appropriate selection of the pricing rule can result in a speedup of 3, as shown in Table 6 and Chart 4 (description of the pricing rules of PPRN is out of the scope of this work). Whem judging the performances of PPRN, it is necessary to take into account the fact that it is an academic code at its first final version, while all the others (except MMCFB) have a much longer history, and it has been written in FORTRAN while all the others have been coded in C (whose compilers may be more efficient): furthermore (from a personal communication by the authors) it does not have a preprocessing phase yet, and due to its input format it was impossible to exploit the preprocessing made by `Graph`.

¤ PPRN is available in a library (.a) format for some different architectures (mainly Unix workstations) at ftp://ftp-eio.upc.es/pub/onl/codes/pprn/libpprn: Jordi Castro can be contacted at jcastro@etse.urv.es

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 5: Phase 0 options for PPRN

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 6: pricing rules for PPRN

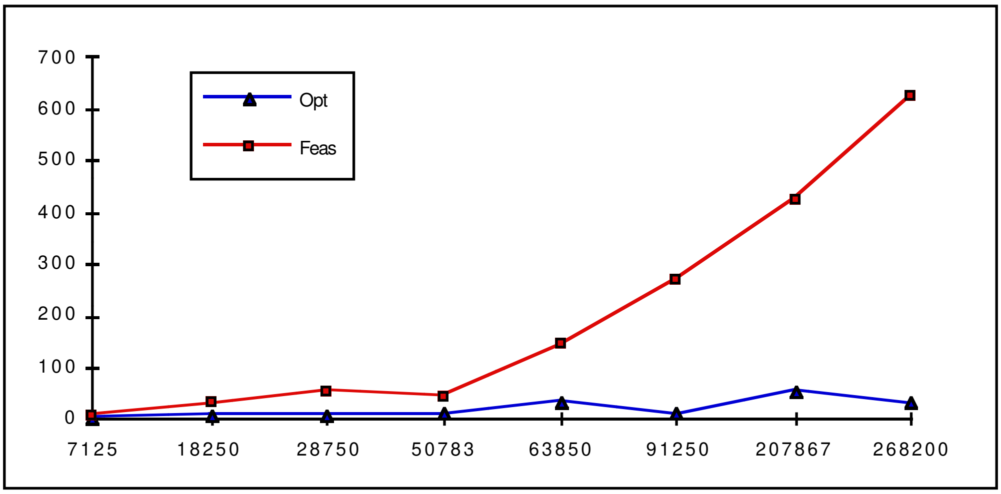

Chart 3: Phase 0 options for PPRN

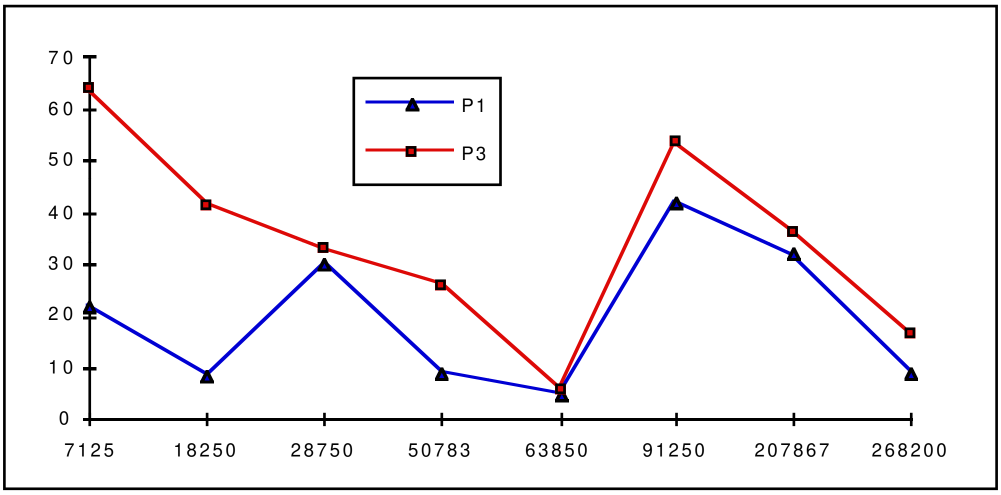

Chart 4: pricing rules for PPRN

II.2.2.3 Interior Point codes

Some preliminari versions of IPM, a specialized Interior Point code for (MMCF) [CN96] that is still evolving quite significantly, were kindly made available by the authors for comparison. The code does not currently handle zero single-commodity capacities, i.e. $u^h_{ij} = 0$, while all but one of the classes of instances have such “forbidden” arcs: the obvious workaround, i.e. setting u^h_{ij} to some very small number and/or c_{ij}^h to some very large number, might result in numerical problems, yet the code was able to solve (almost) all the instances; on the other hand, explicitly handling this case would require separate symbolic factorization of the k different blocks of the $\boldsymbol{A}\Theta^{-2}\boldsymbol{A}^T$ matrix (Cf. Sec. II.1.4.4), which is costly both in memory and time.

For the non-matrix-based IP method of [SM91], only (indirect) comparisons can be provided, and only for one class of problems: hence, two general-purpose (LP) solvers based on the IP technology were also tested, one being the Cplex 3.0 Barrier solver (that will be referred to as IPCplex) and the other the L0Q0 2.21 code [Va92] [VC93] [Va94], a recent implementation of the *primal-dual, path following, predictor-corrector* approach. Again, some optimization parameters were tuned for improving the 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 LL^T factorisation of the (full) $\boldsymbol{A}\Theta^{-2}\boldsymbol{A}^T$ matrix. For IPCplex, the myopic heuristic is more efficient, as shown in Table 7 where for each of *MMD* and *MLF* the two columns with L and R reports the time obtained with Leftward and Rightward Cholesky factorisation (another possible degree of freedom): the table also shows that the total number of iterations is very small, and that it is not always possible to find the "best" setting for a given parameter (MMD-L is better than MMD-D on small problems and worse on larger ones). For L0Q0, the global heuristic is uniformly better, but does not dramatically improve the performances: this difference is not surprising, since the two codes use different (*normal equations* and *reduced-KKT* respectively) approaches to matrix factorisation.

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

Interestingly enough, the *preprocessing* done by `Graph` improves the performances of L0Q0 of over a factor of 10 (vs only a few percent of MMCFB), since it is used to diminish the size of the actual (LP) passed to the solver: this far less important for Cplex, that embeds his own efficient preprocessing phase, but some improvements can still be obtained, showing that knowledge of the (LP) structure makes it easier to locate redundant constraints, and suggesting that a smarter preprocessing phase may be profitable.

II.2.2.4 A Column Generation code

A comparison was also possible with a modern Column Generation code, developed for (repeatedly) solving (MMCF) in a Branch & Bound approach to the Integer (MMCF) [BHV96], and using Cplex for solving the "Master" problem at each iteration. The BCG code was not directly available, anyway it only solves *undirected* (MMCF)s, so that it would not have been able to solve all the other classes of instances without (minor) modifications: however, the authors shared the test instances and the results, even some not reported in the article, allowing a quite satisfactory comparison. On the other hand, only (MMCFB) and the non-specialized codes were able to deal with undirected instances: that of being orthogonal to the type of subproblem is a seldom noticed but relevant advantage of decomposition approaches to (MMCF).

II.2.2.5 Other codes

The above codes do not exhaust all the proposed computational approaches to (MMCF): only indirect comparisons were possible with other IP methods [ZaG95] [SM91], different Cost Decomposition approaches [PZ94] [ZaA95] and ε-approximation algorithms [GK95a], while no comparison at all was possible with Resource Decomposition approaches [Ko94] or Primal-Dual methods [Ze91]. The indirect comparisons concern the PDS problems (Cf. Sec. II.2.3.3), that are difficult (MMCF) with large graphs but few commodities and (MCF) subproblems, and therefore are not representative of other classes of instances, such as those arising in telecommunications: furthermore, the codes were ran on a really etherogeneous set of computers, ranging from vector supercomputers to massively parallel supercomputers to (clusters of) workstations of different models, making it difficult to extract meaningful information from the time figures.

However, the available codes are at least sufficient to draw a picture of the current scenery of computational approaches to (MMCF), to see where Cost Decomposition methods stand: hopefully, other tests will come to further enhance our understanding.

II.2.3 The test problems

The other fundamental ingredient for obtaining a sensible computational comparison is an adequate set of test problems: several data sets and random generators of (MMCF)s are available, and have been used in the literature to test some of the proposed approaches. However, many of the “standard” test problems are small, and even the larger ones have a small number of commodities: this is probably due to the fact that, with most of the methods, the complexity of solving a (MMCF) grows rapidly with k, so that problems with hundreds of commodities have for long time been out of reach. For our experiments, some known data sets have been gathered that might be considered representative of classes of instances arising in practice: however, when no satisfactory test problems were available, different random generators have been used to produce data sets with controlled characteristics, in order to understand the impact of some of them, such as the number of commodities, on the relative performances of the various approaches.

II.2.3.1 The Canad problems

The first data set is a bunch of 96 problems, generated with two custom random generators (one for bipartite and the other for general graphs) developed in the context of heuristic approaches to the Fixed Charge (MMCF) problem [GC95]: the (MMCF) solved in this Chapter have been obtained by simply ignoring the “fixed charge costs”, but the same instances are used again in §II.5.2 in their original setting. A parameter called *capacity ratio* was available as a “knob” to produce lightly or heavily capacitated instances: as (MMCF)s, the problems have shown a posteriori to be quite “easy”, but the corresponding Fixed Charge problems are known to be “hard”, showing that efficient solution of “easy” (MMCF)s still can have an interest. Furthermore, this data set is the only one having instances where the number of commodities is considerably larger that the number of nodes (up to

400 vs 30), as in some real applications.
The instances are divided into the three groups:

- 32 bipartite problems (group A) with multiple sources and sinks for each commodity;
- 32 nonbipartite problems (group B) with single source/sink for each commodity;
- 32 nonbipartite problems (group C) with multiple sources and sinks for each commodity;

and the characteristics of the instances are shown in Table 8 of Sec. II.2.4.1: for each choice of (k, n, m), two "easy" and two "hard" instances have been generated, whose results will always be reported in an aggregate form.

II.2.3.2 The Mnetgen generator

Mnetgen [AK77] is a well-known random generator: an improved C++ version has been developed for these tests, where the internal random number generator (that has shown not to be portable) has been replaced with the `drand48()` routine in the standard C library, the output file format has been made more "standard" [JL92] and (what appeared to be) a minor bug has been fixed. 216 problems have been generated with uniform rules: the problems are divided in 18 groups of 12 problems each, each group being characterised by a pair (n, k) for n in $\{64, 128, 256\}$ and k in $\{4, 8, \ldots, n\}$ (as Mnetgen cannot generate problems with $k > n$). Within each group, 6 problems are "sparse" and 6 are "dense", with m / n respectively equal to about 3 and 8: in both these subgroups, 3 problems are "easy" and 3 are "hard", where an easy problem has mutual capacity constraints on the 40% of the arcs and 10% of the arcs with an "high" cost, while the figures are 80% and 30% respectively for a hard problem. Finally, the 3 subproblems characterised by the touple (n, k, S, D) $(S \in \{\text{sparse, dense}\}, D \in \{\text{easy, hard}\})$ have respectively 30%, 60% and 90% of arcs with individual capacity constraints. Randomly generated instances are believed to be generally "easier" than real-world problems: however, the "hard" Mnetgen instances, especially if "dense" also, have a posteriori shown to be significantly harder to solve (by all the codes) than both the "easy" ones and other instances of equivalent size.

II.2.3.3 The PDS problems

The PDS (Patient Distribution System) instances come from a probelm of transportation of patients from a place of military conflict: the available data represents only one basic problem, but the model is parametric in the *planning horizon* t (the number of days) in that the problems have a time-space network whose size grows about linearly with t. This characteristic is often encountered in (MMCF) models of transpostation and logistic problems, even though in typical cases k also grows with the planning horizon, while in the PDS problems it is uniformly fixed to 11: hence, the largest instances have a very large m / k ratio. Moreover, although all the commodities have a single source and destination, the subproblems are not Shortest Paths because of the individual arc capacities: it might be possible to reformulate the model to make it more easily solvable by Cost Decomposition methods, but this raises the question of the impact of the formulation over the relative efficiency of the different approaches, that requires separate investigations.

II.2.3.4 The JLF data set

A relatively large data set was presented in [JLF93] to test the DW approach to (MMCF): it is formed of various (small) groups of small real-world problems with different structures, often with Shortest Path subproblems and with never more than 15 commodities.

II.2.3.5 The dimacs2pprn "meta" generator

Several (MMCF)s, especially those arising in telecommunications, have large graphs, many commodities and Shortest Path subproblems: all these characteristics are separately present in some of the previous data sets, but never in the same instance. Hence, a generator was needed that allows a great degree of freedom in the choice of the problem structure: the *dimacs2pprn* "meta" generator originally proposed in [CN95] was suited to the task, and a slightly enhanced C++ version was developed for these tests. The generator inputs the description of a single-commodity (MCF) in DIMACS standard format*, i.e. a graph $\bar{G}$ with arc costs $\bar{\boldsymbol{c}}$, node deficits $\bar{\boldsymbol{b}}$ and arc capacities $\bar{\boldsymbol{u}}$, and three parameters k, r and f: it outputs an (MMCF) problem with k commodities on the same graph $\bar{G}$, where deficits and capacities of the i-th subproblem are a "scaling" of those of the original (MCF). More formally, k numbers $\{ r_1 .. r_k \}$ are uniformly drawn at random from $[1 .. r]$ and the deficits and individual capacities are chosen as $\boldsymbol{b}^h = r_h\bar{\boldsymbol{b}}$ and $\boldsymbol{u}^h = r_h\bar{\boldsymbol{u}}$; costs c^h_{ij} are independently and uniformly drawn at random from $[0 .. \bar{c}_{ij}]$, while mutual capacities $\boldsymbol{u}$ are initially fixed to $f\bar{\boldsymbol{u}}$ and eventually increased to accommodate the multiflow obtained by solving the k (MCF)s separately, with random costs uncorrelated with the c^h_{ij}: at the user's choice, the individual capacities can then be set to $+\infty$. This is a "meta" generator since (most of) the structure of the resulting (MMCF) problem is not "hard-wired" into the generator, like in Mnetgen, but depends on the initial (MCF): in turn, it can be obtained with a random generator, such as the well-known Rmfgen [GG88]. For two user-provided parameters a and b, Rmfgen produces a graph made of b squared grids of side a, with one source and one sink at the "far ends" of the graph: for all 6 combinations of a in $\{4, 8\}$ and b in $\{4, 8, 16\}$, an "easy" and a "hard" (MCF) were generated with maximum arc capacities respectively equal to 1/2 and 1/10 of the total flow to be shipped. Then, for each of these 12 (MCF)s 4 (MMCF)s were generated, with k in $\{4, 16, 64, 256\}$, $r = 16$ and $f = 2k$ (so that the mutual capacity constraints are four times "stricter" than the original capacities $\bar{\boldsymbol{u}}$): the individual capacities had been set to $+\infty$, yielding a total of 48 instances with up to one milion of variables, but with (SPT) subproblems.

II.2.3.6 The BHV problems

Two sets of randomly generated *undirected* (MMCF)s have been used to test the Column Generation approach of [BHV96]: in both sets, the commodities are intended to be origin / destination pairs, but the commodities with the same origin can be aggregated in a much smaller number of single origin - many destinations flows.

The 12 "q*x*" problems all have 50 nodes, 130 arcs (of which 96 capacitated) and 585 o/d pairs that can be aggregated into 48 commodities, yielding a total (LP) size of 6240: a posteriori, they have shown to be quite easy.

The 10 "r10-*x*-*y*" problems all have 301 nodes, 497 arcs (of which 297 capacitated) and 1320 o/d

---

* details on the format and the (MCF) generators can be found at ftp://dimacs.rutgers.edu/pub/netflow

pairs that can be aggregated into 270 commodities, yelding a total (LP) size of 134190 (note that the "disaggregated" (LP) sizes are respectively of $7.5 \cdot 10^4$ and $6.5 \cdot 10^5$ variables): the group is divided into 5 "r10-5-*y*" and 5 "r10-55-*y*" instances, that are similar but for the fact that the arc capacities in the first group are integral while those in the second one are highly fractional, and the two subgroups have shown to be of a surprisingly different "difficulty".

It would be nice to be able to tell (to an unknowledgeable end-user) which solution approach is more likely to be efficient by just looking at the instances (he/she has) to solve, but at the present day this is very far from being the case: all the above data sets will be made available to the researchers, in the hope to encourage more computational comparisons to be performed and hence to gather a deeper understanding of the matter.

II.2.4 Computational results

In this paragraph, MMCFB is compared to the other codes described in §II.2.2 for the solution of the test problems described in §II.2.3: since the approaches are very different, the performance measure will be the running time in seconds taken by the solution of the problem, excluding loading and preprocessing time. Unless explicitly stated, the times have been obtained on a HP9000/712-80 workstation: for the codes ran on different machines, the original figures will be reported and a qualitative estimate of the relative speed will be obtained - if possible - by comparing the SPECint92 and SPECfp92 performance figures (97.1 and 123.3 respectively for the reference machine¶). For all the codes, the required relative precision of the optimal solution was set to 10^{-6}, i.e. about six digits, exceptions being PPRN that do not allow such a choice, IPM where values smaller than 10^{-5} could cause numerical difficulties to the PCG and some of the runs reported in the literature, that will be explicitly discussed.

In all the following tables, b ($< m$) if present is the number of arcs that have a mutual capacity constraint, and for large data sets each entry is averaged from a set of instances with similar $size = mk$; a "*" means that the code was not able to solve (all) the instance(s of that group), usually due to memory problems.

II.2.4.1 The Canad problems

The results of the three groups of instances A, B and C are summarized in Table 8, where "*" means that the instance could not even be loaded due to memory problems, "**" that the code broke down with numerical difficulties (a quite unusual outcome for Cplex) and "***" that the computation had to be interrupted because the workstation was "trashing" (i.e. spending almost all the time in paging virtual memory faults): the results are also visualized in Chart 5, Chart 6 and Chart 7 for the three groups separatedly, where the running time of each algorithm is plotted as a function of the *size*.

If compared to the faster code, the general-purpose IP solvers are 2 to 3 orders of magnitude slower on groups B and C and 1 to 2 on group A; furthermore, they are not able to solve the largest instances

¶ most of the figures used here have been obtained from the Performance Database Server at the University of Tennessee http://performance.netlib.org/performance/html/spec.html due to memory or numerical problems.

	k	*n*	*m*	*size*	MMCFB	Cplex	PPRN	IPM	L0Q0	IPCplex
	10	50	400	4000	2.37	1.27	2.13	3.38	13.59	11.76
	10	50	625	6250	3.40	1.99	2.18	9.13	49.83	34.82
	10	100	1600	16000	3.01	3.66	4.66	17.87	100.09	* *
A	10	100	2500	25000	3.04	4.24	6.13	31.42	387.88	* *
	100	50	400	40000	0.76	10.31	10.23	18.53	472.95	358.47
	100	50	625	62500	2.43	89.96	29.79	42.49	2609.69	1729.01
	100	100	1600	160000	1.39	28.95	28.00	103.04	*	*
	100	100	2500	250000	1.46	36.51	27.03	174.03	*	*
	40	20	230	9200	0.06	0.90	0.26	6.11	193.53	100.64
	40	20	289	11560	0.06	1.14	0.28	3.71	154.13	176.97
	200	20	229	45800	0.24	5.56	2.02	26.61	645.80	988.65
B	100	30	517	51700	0.45	6.16	3.93	29.25	9895.41	1988.24
	200	20	287	57400	0.36	7.54	3.52	32.45	888.71	1343.36
	100	30	669	66900	0.25	7.07	1.64	32.11	14016.83	2915.02
	400	30	519	207600	1.03	30.18	24.96	145.15	*	* * *
	400	30	688	275200	1.19	38.32	23.32	170.14	*	* * *
	40	20	230	9200	0.23	1.02	0.37	5.64	223.79	108.88
	40	20	289	11560	0.15	1.22	0.31	5.79	182.43	189.19
	200	20	229	45800	0.90	7.86	6.23	22.92	727.55	1044.92
C	100	30	517	51700	1.10	8.05	6.51	25.99	12126.88	2156.04
	200	20	287	57400	0.51	6.93	4.28	25.85	947.17	1429.17
	100	30	669	66900	0.79	8.76	4.24	32.18	16093.50	3392.77
	400	30	519	207600	3.26	48.21	38.53	129.10	*	* * *
	400	30	688	275200	3.62	60.87	42.69	167.04	*	* * *

Table 8: results for groups A, B and C of the Canad problems

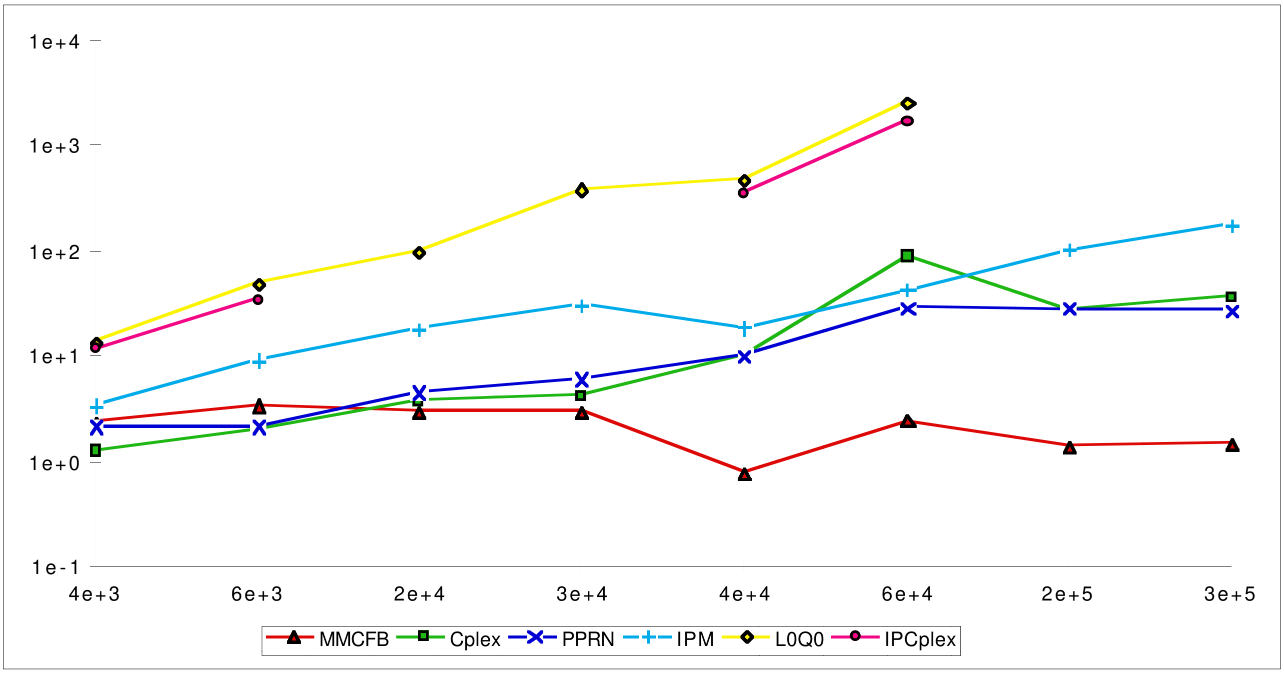

Chart 5: Canad problems, bipartite graphs (group A)

The specialized IPM is (approximately) 3 to 20 times faster on group A, 20 to 70 times faster on group B and 10 to 40 times faster on group C: however, it is almost never competitive with any of MMCFB, Cplex and PPRN.

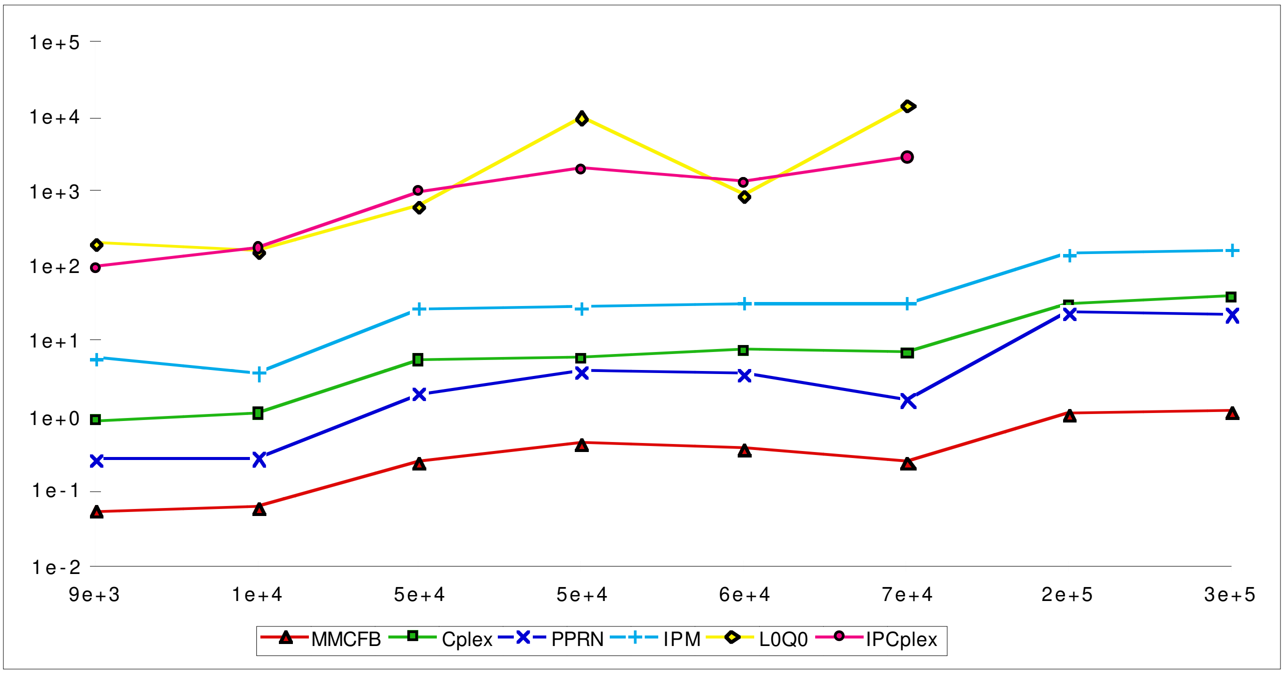

Chart 6: Canad problems, generic graphs (group B)

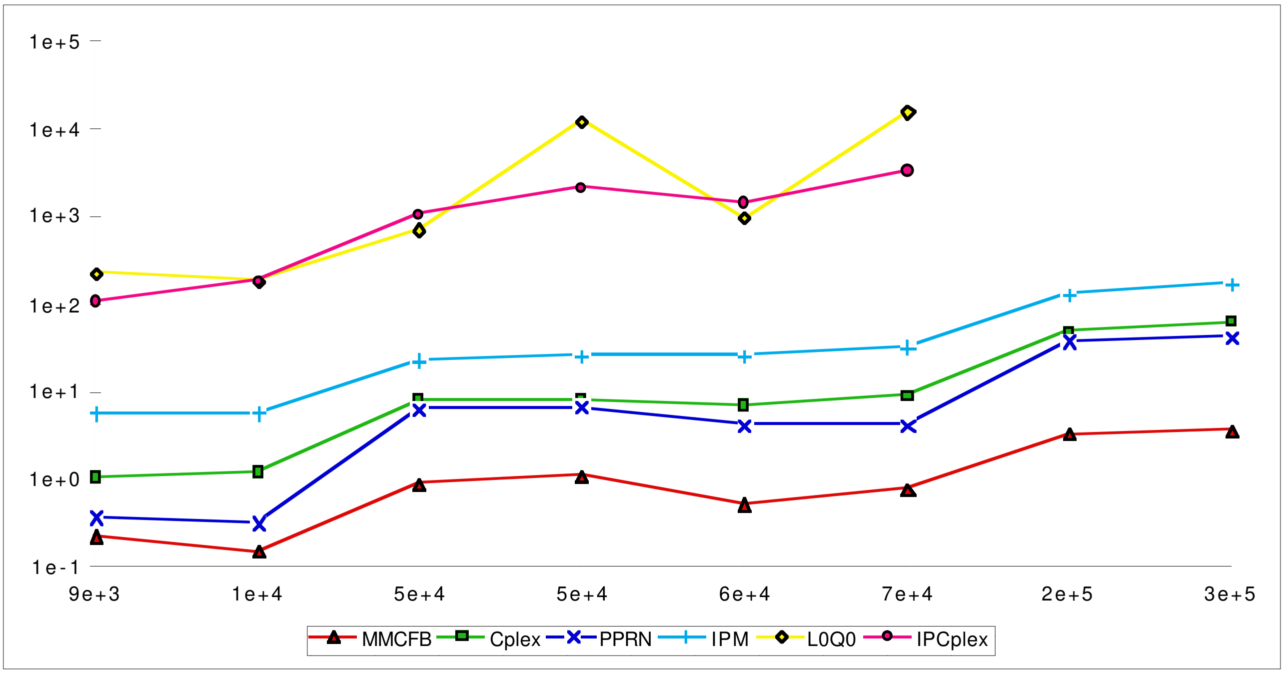

Chart 7: Canad problems, generic graphs (group C)

Among the latters, MMCFB is 10 to 40 times faster than Cplex and 5 to 20 times faster than PPRN on all instances but the first half of group A, where it is less than 2 times slower: these are "difficult" small instances with few commodities, where MMCFB requires considerably more iterations to converge than for all the other problem groups. This may lead to the conjecture that MMCFB is not competitive on "difficult" instances, but other experiences will show that this is true only if the number of commodities is also small: since there are not difficult Canad problems with many commodities, the relative efficiency of IPM might be underestimated by these results, because, differently from the other iterative methods, Interior Point methods always terminate in a small number of steps almost regardless of the "difficulty" of the instance.

II.2.4.2 The Mnetgen problems

The Mnetgen problems have a wide range of different characteristics, from tiny to very large sizes (more than half a million of variables), and from small to very large m / k ratios: Table 9 summarises the obtained results where all the 12 problems with the same (n, k) have been grouped together. Again, the general-purpose IP codes are never competitive, but this time the specialized IPM is better than both Cplex and PPRN on the instances with large k (greater than 32): the speedup also increases with both m and k, reaching over an order of magnitude in the (256, 256) instances. However, MMCFB is even faster, outperforming the simplex-based codes of three orders of magnitude on the largest instances; for example, the hard and dense (256, 256) problems were solved in about 10^3 seconds by MMCFB, while PPRN was not able to produce a result in $4 \cdot 10^6$ seconds and Cplex was trashing the workstation: the expected running times of Cplex and PPRN on those instances are, in the mild estimate used for Table 9 (the "+" entries), more than $3 \cdot 10^6$ and $6 \cdot 10^6$ seconds respectively.

k	*n*	*m*	*b*	*Size*	MMCFB	Cplex	PPRN	IPM	LOQO	IPCplex
4	64	362	148	1.4e+3	0.07	0.22	0.13	1.44	1.73	1.45
8	64	371	183	3.0e+3	0.26	0.50	0.52	4.26	9.22	7.56
16	64	356	191	5.7e+3	1.08	2.01	3.41	16.03	58.96	40.14
32	64	362	208	1.2e+4	3.42	12.99	22.04	43.27	190.76	98.73
64	64	361	213	2.3e+4	8.53	115.99	147.10	114.19	244.66	216.44
4	128	694	293	2.8e+3	0.58	0.54	0.85	6.45	7.18	6.49
8	128	735	363	5.9e+3	2.57	1.81	4.79	26.32	66.65	50.56
16	128	766	424	1.2e+4	11.30	17.31	40.57	116.26	683.47	394.19
32	128	779	445	2.5e+4	27.72	212.09	503.48	346.91	*	*
64	128	784	469	5.0e+4	44.04	1137.05	2215.48	719.69	*	*
128	128	808	485	1.0e+5	52.15	5816.54	6521.94	1546.91	*	*
4	256	1401	570	5.6e+3	7.54	2.38	9.88	51.00	50.70	40.37
8	256	1486	743	1.2e+4	25.09	15.48	105.89	208.10	568.02	377.87
16	256	1553	854	2.5e+4	60.85	180.06	955.20	844.09	*	*
32	256	1572	907	5.0e+4	107.54	1339.46	6605.45	1782.47	*	*
64	256	1573	931	1.0e+5	144.75	7463.14	18467.73	3441.62	*	*
128	256	1581	932	2.0e+5	223.13	35891.37	61522.94	9074.31	*	*
256	256	1503	902	3.8e+5	445.81	110897+	187156+	17279.00	*	*

Table 9: aggregated results on the Mnetgen problems

	4	8	16	32	64	128	256
64	2.93	1.90	1.87	3.80	13.60		
128	0.93	0.70	1.53	7.65	25.82	111.53	
256	0.32	0.62	2.96	12.46	51.56	160.85	251*

Table 10: ratio between Cplex and MMCFB times - problems in the same row have fixed n and varying k

However, a closer examination of the results enlightens the weakness of MMCFB on large instances with few commodities: this is better visualised in Table 10, Table 11 and Table 12, and the corresponding Chart 8, Chart 9 and Chart 10, where the ratio between the running times of Cplex (respectively PPRN and IPM) 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.

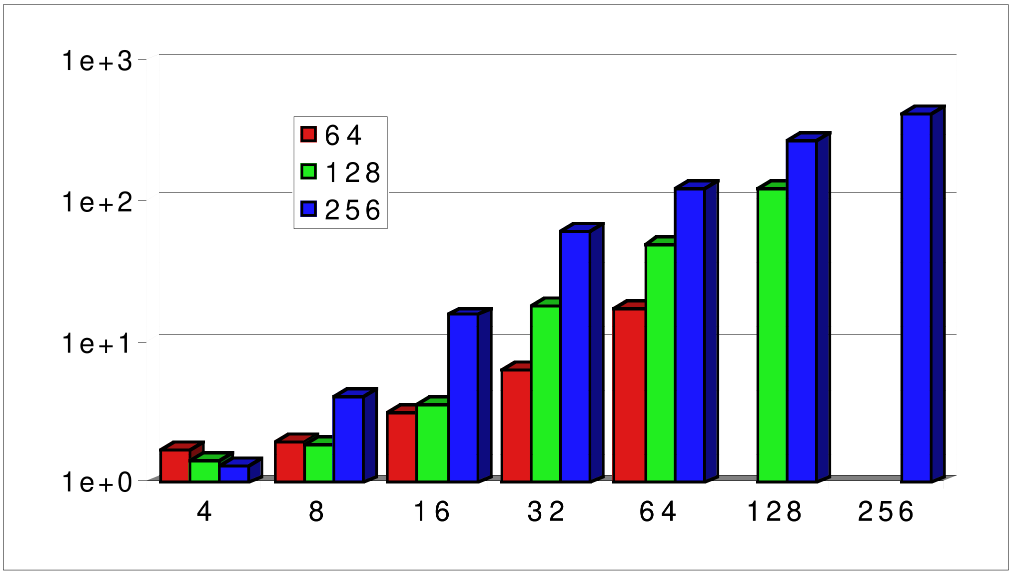

Chart 8: data of Table 10 - for each k, three columns are shown for the three values of n

	4	8	16	32	64	128	256
64	1.73	1.98	3.16	6.45	17.25		
128	1.47	1.86	3.59	18.17	50.30	125.06	
256	1.31	4.22	15.70	61.42	127.58	275.72	424*

Table 11 ratio between PPRN and MMCFB times - problems in the same row have fixed n and varying k

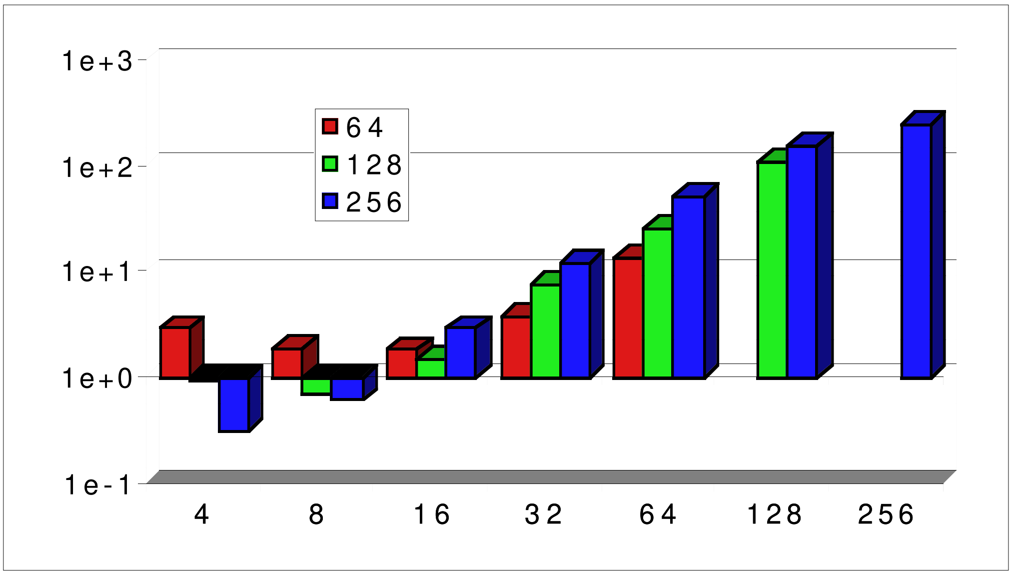

Chart 9: data of Table 11 - for each k, three columns are shown for the three values of n

As the table shows, MMCFB is slower than Cplex of up to a factor of 4 for instances with a large m / k ratio: more disaggregated data about some "critical" 256-nodes problems is reported in Table 13 for a better understanding of the phenomenon.

	4	8	16	32	64	128	256
64	19.69	16.24	14.85	12.66	13.39		
128	11.14	10.23	10.29	12.52	16.34	29.66	
256	6.77	8.29	13.87	16.57	23.78	40.67	38.76

Table 12 ratio between IPM and MMCFB times - problems in the same row have fixed *n* and varying *k*

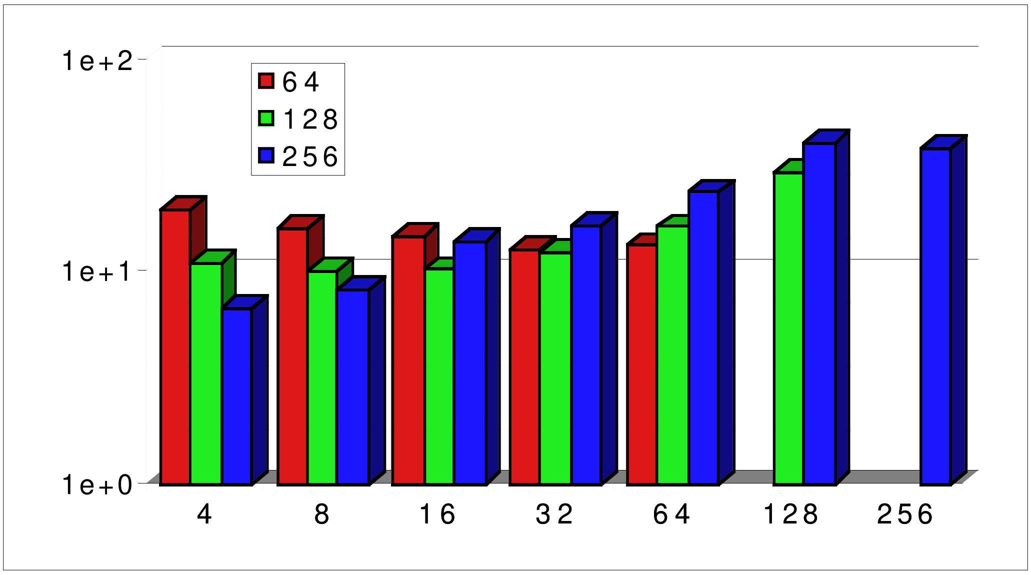

Chart 10: data of Table 12 - for each *k*, three columns are shown for the three values of *n*

k	*m*	type	MMCFB	Cplex	PPRN	IPM	Cplex%	PPRN%	IPM%
4	779	s / e	0.99	0.68	1.47	8.97	0.69	1.49	9.06
4	781	s / h	7.30	1.73	5.93	26.59	0.24	0.81	3.64
4	2021	d / e	3.21	1.76	5.88	42.06	0.55	1.83	13.09
4	2021	d / h	18.65	5.36	26.22	126.38	0.29	1.41	6.78
8	795	s / e	1.98	1.85	5.74	26.55	0.93	2.90	13.41
8	798	s / h	12.62	4.52	15.67	83.88	0.36	1.24	6.65
8	2171	d / e	21.32	10.09	55.33	181.09	0.47	2.60	8.49
8	2177	d / h	64.44	45.46	346.82	540.88	0.71	5.38	8.39
16	799	s / e	5.56	6.15	28.16	65.87	1.11	5.07	11.85
16	831	s / h	21.76	19.97	77.01	208.20	0.92	3.54	9.57
16	2280	d / e	45.74	54.53	272.50	735.33	1.19	5.96	16.08
16	2303	d / h	170.33	639.59	3443.10	2366.96	3.76	20.21	13.90
32	816	s / e	11.01	20.52	100.64	197.00	1.86	9.14	17.90
32	820	s / h	47.16	164.15	488.98	451.94	3.48	10.37	9.58
32	2331	d / e	79.03	464.85	1205.80	1391.59	5.88	15.26	17.61
32	2321	d / h	292.98	4708.31	24626.37	5089.33	16.07	84.06	17.37
64	825	s / e	20.20	116.21	522.45	437.24	5.75	25.86	21.64
64	801	s / h	54.73	362.61	1197.35	699.22	6.63	21.88	12.78
64	2337	d / e	146.11	5239.85	7298.90	3818.54	35.86	49.95	26.13
64	2330	d / h	357.97	24133.90	64852.23	8811.50	67.42	181.17	24.62

Table 13: disaggregated data about some 256-nodes problems

Noticeably, “hard” problems are really harder (for all the codes) to solve than “easy” ones of the same size: moreover, they are much harder than e.g. Canad problems of comparable size, since a $1.6{\cdot}10^5$

variables (100, 100, 1600) Canad instance can be solved in 1.5 seconds, while an hard $1.5 \cdot 10^5$ variables (64, 256, 2300) Mnetgen instance requires over 350 seconds. On easy instances with few commodities, MMCFB is about 2 times slower than Cplex, and about 4 times slower on hard ones: conversely, as the number of commodities increases the *relative efficiency* (the *Cplex%* and *PPRN%* columns) of MMCFB is larger on hard problems than on easy ones, and it increases with the problem size. This is probably due to the fact that the number of $\varphi()$ evaluations depends on the "hardness" of the instance, but much less on k, while the number of simplex iterations tends to increase under the combined effect of m and k: the fact that two different simplex-like method have a similar behaviour seem to indicate that this may be a *structural* characteristic of this class of algorithms, i.e. that decomposition methods may *inherently* have better performances on many-commodities problems. As we expected, IPM seem to be less affected by the "difficulty" of the instance than all the other codes: in fact, the relative efficiency of MMCFB (the *IPM%* column) is on the contrary lower on "hard" problems, confirming that the IP approach is especially promising on difficult problems. However, since MMCFB is more and more competitive with IPM as k increases, IPM is likely to be convenient only on "hard" problems with small k: this is in fact confirmed by the next data set, and by noting that, in Chart 10, the relative efficiency of MMCFB decreases as n increases for small k, but increases with n for larger values of k. It is important to note, however, that these results have to be taken with some care: in all the Mnetgen instances there are arcs with $u^k_{ij} = 0$, that have been given a tiny capacity and a very large cost, and this workaround may have an adverse effect on the performances of IPM that is currently impossible to estimate.

II.2.4.3 The PDS problems

As shown in Table 14, several PDSt instances were solved up to $t = 40$: the (LP) sizes range from $4.1 \cdot 10^3$ to about $2.4 \cdot 10^5$ variables, but k is always 11, so that the m / k ratio for PDS40 is about 2000. Starting with PDS15, the two general-purpose IP codes began trashing, but even for the smaller instances they are always one order of magnitude slower than the simplex-based codes, as depicted in Chart 11. For $t \leq 18$ Cplex outperforms both PPRN and MMCFB, but as the size increases the gap lessens, and for the largest instances MMCFB outperforms Cplex of a factor of four; also, for $t = 24$ PPRN is competitive with Cplex, and for larger instances the specialized simplex is better than the generic one of a factor of two: however, neither Cplex nor PPRN were able to solve PDS40 due to memory problems, while MMCFB was, although it required more than 6 hours. The only other code that was capable of solving all the instances is IPM, that for $t \geq 33$ have quite analogous (but slightly better) performances than MMCFB: recalling §II.2.3, the latter may become competitive earlier on instances with analogous characteristics but (SPT) subproblems, but it seems to be unlikely that a Bundle-based Cost Decomposition method can obtain consistent speedups on these classes of instances.

Since PDS problems have $m / n \approx 3$, it is possible (and instructive) to contrast the results with that on hard (.h) and easy (.e) "sparse" Mnetgen instances: Table 15 shows that MMCFB solves an hard Mnetgen instance much faster than the corresponding PDS one, while the running times of the other methods are far less different (however, the difference increases as k goes far from 11).

t	*n*	*m*	*b*	*Size*	MMCFB	Cplex	PPRN	IPM	LOQO	IPCplex
1	126	372	87	4.1e+3	0.70	0.81	1.12	3.80	7.38	3.76
2	252	746	181	8.2e+3	5.66	2.12	5.60	13.38	23.57	14.52
3	390	1218	303	1.3e+4	13.10	5.32	15.88	32.22	93.91	42.93
4	541	1790	421	2.0e+4	50.18	12.87	33.74	73.98	316.32	102.07
5	686	2325	553	2.6e+4	127.03	19.54	60.16	120.34	799.49	215.09
6	835	2827	696	3.1e+4	129.92	41.97	112.3	296.30	1118.41	387.11
7	971	3241	804	3.6e+4	241.13	79.31	171.64	536.86	1860.80	635.53
8	1104	3629	908	4.0e+4	287.03	96.61	221.46	429.26	2171.42	1039.43
9	1253	4205	1048	4.6e+4	424.59	147.93	272.28	667.90	3523.90	2281.92
10	1399	4792	1169	5.3e+4	928.14	207.64	467.49	823.81	5437.82	3485.19
11	1541	5342	1295	5.9e+4	813.08	324.78	499.58	1167.99	7820.92	5961.47
12	1692	5965	1430	6.6e+4	828.58	373.35	560.25	1625.69	11785.1	6522.1
13	1837	6571	1556	7.2e+4	1033.7	315.61	945.13	2060.11	15922.9	8281.6
14	1981	7151	1684	7.9e+4	2198.5	524.05	1325.3	2172.29	19033.1	10276.9
15	2125	7756	1812	8.5e+4	1666.6	885.88	1431.1	3054.57	*	*
18	2558	9589	2184	1.1e+5	2237.47	2132.57	3093.51	5182.26	*	*
20	2857	10858	2447	1.2e+5	3571.58	3766.72	5213.73	8910.87	*	*
21	2996	11401	2553	1.3e+5	3541.34	5823.80	7475.30	8237.08	*	*
24	3419	13065	2893	1.4e+5	5796.18	11936.5	10817.8	9151.14	*	*
27	3823	14611	3201	1.6e+5	8761.89	28790.4	14670.2	11687.2	*	*
30	4223	16148	3491	1.8e+5	10343.2	51011.5	18995.0	13935.1	*	*
33	4643	17840	3829	2.0e+5	15459.3	54351.3	28869.6	12537.9	*	*
36	5081	19673	4193	2.2e+5	17689.2	*	34453.8	17519.3	*	*
40	5652	22059	4672	2.4e+5	22888.5	*	*	20384.2	*	*

Table 14: results for the PDS problems

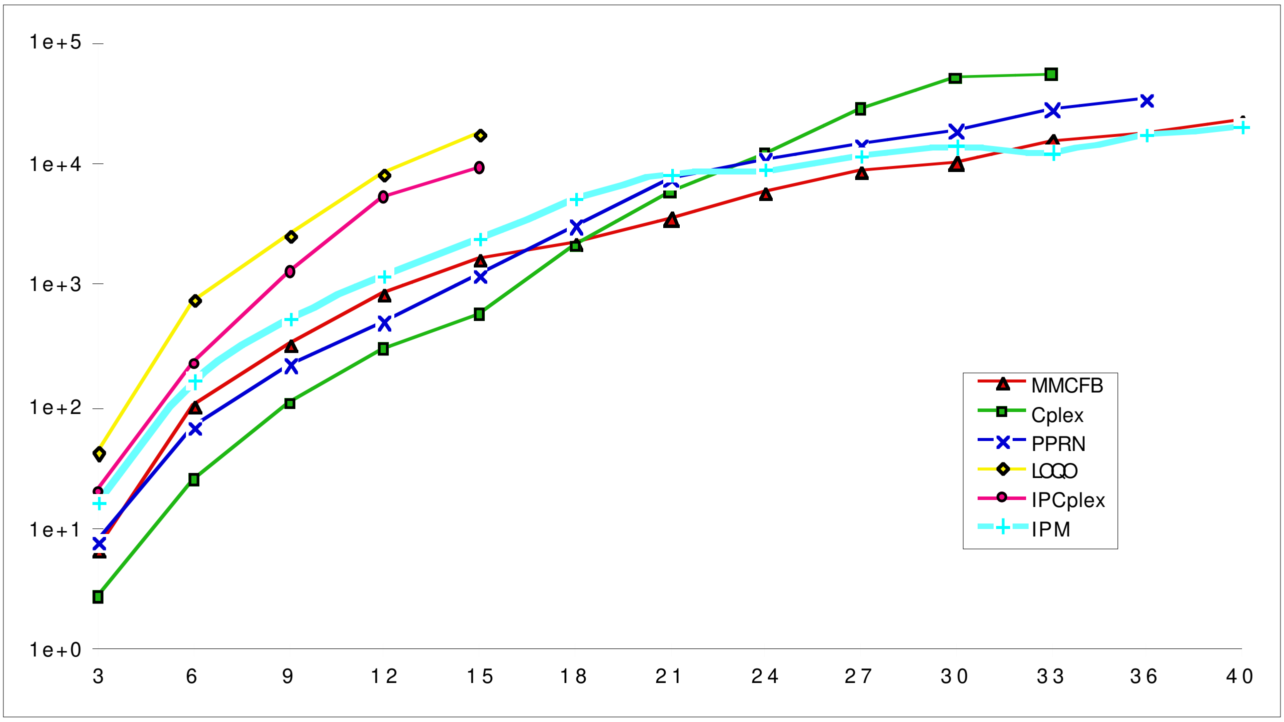

Chart 11: results for the PDS problems

The PDS problems are considered difficult, but the “measure of difficulty” has never been explicitly stated: by taking it as the solution time required by a simplex method, it would be possible to assert that the Mnetgen instances are about as hard, but they can be efficiently solved by MMCFB. Clearly, this statement can be criticised: however, it is reasonable that a Cost Decomposition method performs better on problems where the overall “difficulty” is given by a large *k* rather than by a large *m*.

	k	*n*	*Size*	MMCFB	Cplex	PPRN	IPM
16.256.e	16	256	12789	5.56	6.15	28.16	65.87
16.256.h	16	256	13301	21.76	19.97	77.01	208.20
PDS3	11	390	13398	13.10	5.32	15.88	32.22
32.256.e	32	256	26123	11.01	20.52	100.64	197.00
32.256.h	32	256	26229	47.16	164.15	488.98	451.94
PDS5	11	686	25575	127.03	19.54	60.16	120.34
64.256.e	64	256	52779	20.20	116.21	522.45	437.24
64.256.h	64	256	51264	54.73	362.61	1197.35	699.22
PDS10	11	1399	52712	928.14	207.64	467.49	823.81
256.256.e	256	256	210859	183.37	4173.05	16150.89	3346.51
256.256.h	256	256	212480	316.59	16222.97	27625.40	6735.61
PDS33	11	4643	196240	15459.30	54351.30	28869.60	12537.90

Table 15: Mnetgen and PDS problems of the same size

On the PDS problems, it is also possible to compare several of the approaches proposed in the literature: in Table 16, this is attempted for PDS1, 10, 20, 30 and 40.

	Cplex	MMCFB	ZaA	PZ	GK	SM	ZaG
1	1	1			7	840	
10	208	928	65	232	123	1920	45
20	3767	3572	497	1225	372	8220	253
30	51012	10343	557	5536	765	19380	654
40		22889	858		1448	38160	1434

Table 16: comparison with approaches from the literature for the PDS problems

Unfortunately, extracting significant information from the figures is very difficult: the other approaches are

- the parallel Nonlinear Jacobi Algorithm (ZaA) applied to the Augmented Lagrangean of [ZaA95], that was run on a Thinking Machine CM-5 with 64 processors, each one (apparently) capable of about 25 SPECfp92;
- the linear-quadratic exact penalty function method (PZ) of [PZ94], that was run on a Cray Y-MP vector supercomputer, whose performances w.r.t. scalar architectures are very hard to assess;
- the ε-approximation algorithm (GK) of [GK95a], that was run on a IBM RS6000-550 workstation (83.3 SPECfp92 and 40.7 SPECint92): however, the largest instances were only solved to the precision of 4-5 significant digits;
- the specialized IP method (SM) of [SM91], that was run on a DECstation 3100 (whose SPEC figures are unknown) and that obtained the solutions with the precision of 8 digits;
- a modification and parallelization of the above (ZaG) that was run on a cluster of SPARCstation 20 workstations [ZaG95]; unfortunately, the exact model is not specified, so that the (SPECfp92,

SPECint92) can be anything between (80.1, 76.9) and (208.2, 169.4).

For these instances, the ε-approximation algorithm provides reasonably good solutions in a very low time: however, the theoretical analysis of the method predicts O($k$log(m)) iterations, so that it is particularly well-suited for the PDS problems, but it may suffer large values of k. Among the other methods, the two parallel ones seem to provide good results, since MMCFB could not surely achieve a speedup larger than 11 (on the same machine): however, the comparison is very difficult, and it is perhaps better not to attempt any comment. An interesting remark can instead be done about the concept of "solution" that is underneath some of the figures in the table, since many of the codes use heuristic stopping criteria, essentially performing a fixed number of iterations before stopping: a posteriori, the solutions are found to be accurate, but the algorithm is not capable of "certifying"it by means of appropriate primal-dual estimates of the gap. To illustrate the issue, in Chart 12 and Chart 13 the value of $\varphi(\bar{y})$ and the φ^* estimate ($\varphi(\bar{y}) + t^*\| \boldsymbol{d} \|^2 + \sigma$) of MMCFB are plotted as a function of the time for PDS10 and PDS30 respectively, together with the "solution" of GK (top horizontal line) and the optimum value (middle line): the arrow indicates the (normalised) stop time of GK.

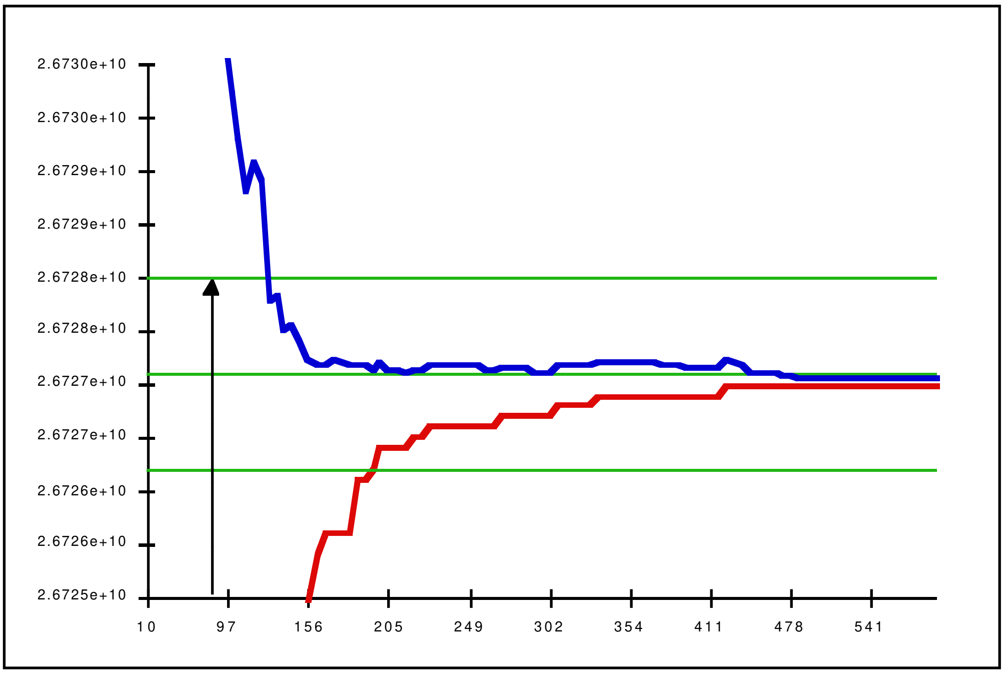

Chart 12: convergence of MMCFB on PDS10

Indeed, the GK algorithm rapidly obtains a good solution, but a comparable accuracy (bottom horizontal line) on the dual side is reached by MMCFB in a fraction of its "solution" time reported in the tables: surprisingly, the "primal" accuracy is even superior to the "dual" one, but MMCFB uses a dual method so that no primal solution corresponding to the estimate is actually available, and since the focus was on the "exact" solution no attempt has been done to obtain it; nonetheless, it may be possible to convert the aggregate unfeasible solution into a feasible (hopefully good) one, e.g. by means of "resource allocation" heuristics, turning MMCFB into an approximate solver.

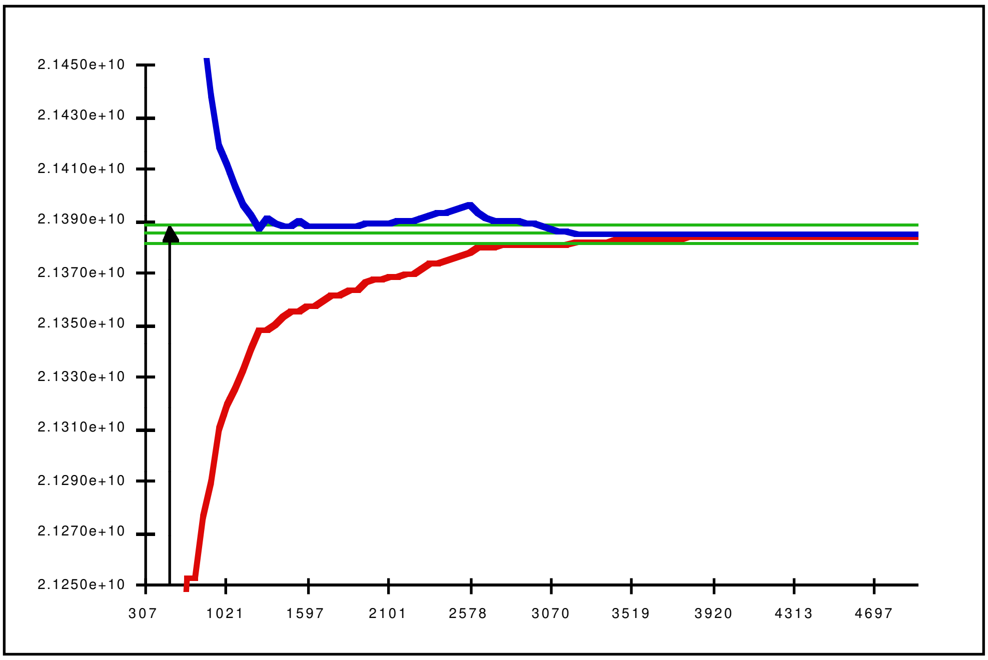

Chart 13: convergence of MMCFB on PDS30

II.2.4.4 The JLF problems

	k	*n*	*m*	*b*	*Size*	MMCFB	Cplex	PPRN	IPM	LOQO	IPCplex
10term	10	190	510	146	5100	0.27	0.84	1.07	9.16	26.17	10.97
10term.0	10	190	507	143	5070	0.23	0.78	0.70	7.04	28.29	13.10
10term.50	10	190	498	134	4980	2.00	0.82	1.22	7.97	18.05	11.60
10term.100	10	190	491	127	4910	2.40	0.86	1.10	8.69	19.21	10.58
15term	15	285	796	253	11940	4.81	3.39	11.57	41.35	210.94	106.98
15term.0	15	285	745	202	11175	1.94	2.72	10.23	30.35	123.21	47.15
Chen0	4	26	117	43	468	0.30	0.25	0.32	0.25	0.44	0.38
Chen1	5	36	174	65	870	0.98	0.64	0.73	0.41	0.99	0.77
Chen2	7	41	358	155	2506	5.98	4.89	3.77	1.59	4.17	3.91
Chen3	15	31	149	56	2235	1.22	5.18	1.96	1.10	3.12	2.62
Chen4	15	55	420	176	6300	25.49	83.39	20.05	5.92	46.92	32.75
Chen5	10	65	569	242	5690	52.58	48.74	48.12	5.24	21.18	15.53
assad1.5k	3	47	98	98	294	0.03	0.11	0.09	0.36	0.39	0.22
assad1.6k	3	47	98	98	294	0.01	0.08	0.06	0.39	0.37	0.22
assad3.4k	6	85	204	95	1224	0.08	0.33	0.55	1.41	4.26	1.44
assad3.7k	6	85	204	95	1224	0.10	0.29	0.56	1.57	4.22	1.41

Table 17: results for some of the JLF problems

Table 17 shows the results obtained on some JLF problems, those with a Shortest Path structure: despite the small number of commodities, MMCFB is still competitive on the "*x*term.*y*" and "Assad*x*.*y*" sets. The "Chen*x*" problems seem to be more "difficult" than the others, since they require an order of magnitude more time to be solved than "*x*term.*y*" instances of comparable size: for this data set, all the IP-based codes are at least competitive, and IPM largely outperform all the others, confirming the potential advantage of the IP technology on "hard" instances. However, such small (MMCF)s are well in the range of general-purpose (LP) solvers, so that it makes a little sense to use ad-hoc technology: in this case, other issues such as availability, ease-of-use and reoptimization should be taken into account instead.

II.2.4.5 The dimacs2pprn problems

Due to the previous results, the general-purpose IP codes were not run on this data set, that contains the largest instances of all: as described in Sec. II.2.3.5, for any given dimension a *hard* and an *easy* instance have been constructed, whose results are separatedly reported in Table 18 and Table 19 respectively (note the differences in the number of mutual capacity constraints *b*). As in the Mnetgen case, Cplex ran out of memory on the largest instances (the "*" entries), and PPRN was sometimes stopped after having run for a very long time without producing a result (the "+" entries). Noticeably, a $2.5 \cdot 10^5$ variables (MMCF) requires about 11, 20 and 129 megabytes of memory to be solved by MMCFB, PPRN and Cplex respectively, so that MMCFB requires an order of magnitude less memory than Cplex, making it possible to solve large (MMCF)s even on low-end computers such as modern PCs; moreover, due to the "abstract" data types described in §II.2.2 memory consumption could also be further reduced. In this case, also IPM experienced memory problems on the largest instances with 1.2 millions of variables: hence, the specialized IP code is more demanding in memory than the specialized simplex code, although being remarkably faster as *k* increases.

k	*n*	*m*	*size*	*b*	MMCFB	Cplex	PPRN	IPM
4	64	240	9.6e+2	240	0.46	0.32	0.37	1.23
4	128	496	2.0e+3	112	1.30	1.05	2.36	2.65
4	256	1008	4.0e+3	1007	21.50	8.85	14.82	23.13
4	256	1088	4.4e+3	192	0.36	1.38	2.22	8.86
4	512	2240	9.0e+3	447	5.97	14.39	32.26	63.44
4	1024	4544	1.8e+4	960	39.76	70.41	178.06	266.79
16	64	240	3.8e+3	240	0.44	1.23	1.68	7.58
16	128	496	7.9e+3	112	4.31	19.72	29.75	24.74
16	256	1008	1.6e+4	240	64.79	230.82	550.11	437.41
16	256	1088	1.7e+4	192	1.08	5.38	22.14	39.70
16	512	2240	3.6e+4	448	14.15	218.55	479.96	540.90
16	1024	4544	7.3e+4	960	69.59	1079.63	2796.85	2363.02
64	64	240	1.5e+4	240	2.89	50.70	92.77	51.55
64	128	496	3.2e+4	112	9.23	222.65	562.71	410.42
64	256	1008	6.5e+4	1008	94.07	3256.52	10011.30	4556.57
64	256	1088	7.0e+4	192	4.37	93.14	535.47	283.19
64	512	2240	1.4e+5	448	37.69	2362.37	7146.33	2703.51
64	1024	4544	2.9e+5	960	215.70	18763.9	52787.6	22899.4
256	64	240	6.1e+4	240	17.52	668.27	1415.55	258.87
256	128	496	1.3e+5	112	48.72	3525.27	11443.50	501.80
256	256	1008	2.6e+5	1008	458.36	56756.9	200000+	6069.46
256	256	1088	2.8e+5	192	15.30	821.86	7832.38	1475.91
256	512	2240	5.7e+5	448	218.84	*	138225.0	22266.8
256	1024	4544	1.2e+6	960	898.51	*	400000+	*

Table 18: results of the "hard" dimacs2pprn instances

The results obtained for these instances further confirm those of the previous data sets: MMCFB is faster than the two simplex-based codes on all but the smallest instances, the specialized IP code is competitive with both Cplex and PPRN on the instances with large *k*, but MMCFB is even more efficient - for the 256-commodities problems it is always over an order of magnitude faster than the other codes, approaching the three orders of magnitude on the largest instances. Yet, especially on large problems, MMCFB spends most of the time in the (SPT) solver, that could probably be significantly enhanced, for instance by using more sophisticated algorithms or exploiting the solution of the previous iteration.

k	*n*	*m*	*size*	*b*	MMCFB	Cplex	PPRN	IPM
4	64	240	9.6e+2	48	0.07	0.17	0.16	0.54
4	128	496	2.0e+3	112	0.18	0.24	0.26	1.61
4	256	1008	4.0e+3	239	1.12	1.81	3.40	5.37
4	256	1088	4.4e+3	191	0.08	0.54	1.80	4.17
4	512	2240	9.0e+3	447	0.75	6.41	12.08	35.58
4	1024	4544	1.8e+4	960	1.84	14.02	34.84	101.49
16	64	240	3.8e+3	48	0.14	0.88	1.51	3.19
16	128	496	7.9e+3	112	0.66	3.50	7.67	10.88
16	256	1008	1.6e+4	240	3.53	20.27	65.31	63.66
16	256	1088	1.7e+4	192	0.37	3.42	8.05	31.60
16	512	2240	3.6e+4	448	2.24	43.30	275.89	166.14
16	1024	4544	7.3e+4	960	6.64	74.82	1286.26	514.74
64	64	240	1.5e+4	48	1.37	11.03	34.00	19.72
64	128	496	3.2e+4	112	2.12	25.80	109.97	51.61
64	256	1008	6.5e+4	240	15.75	445.28	1395.74	333.13
64	256	1088	7.0e+4	192	2.75	47.05	339.72	195.77
64	512	2240	1.4e+5	448	8.83	261.62	4492.95	754.25
64	1024	4544	2.9e+5	960	32.18	978.31	22511.3	3411.4
256	64	240	6.1e+4	48	9.21	208.90	694.94	94.94
256	128	496	1.3e+5	112	11.36	401.02	2662.57	228.16
256	256	1008	2.6e+5	240	96.34	7324.71	28435.0	1340.8
256	256	1088	2.8e+5	192	13.33	474.34	5270.80	1178.85
256	512	2240	5.7e+5	448	85.17	*	81419.1	4236.8
256	1024	4544	1.2e+6	960	159.33	*	400000+	*

Table 19: results of the “easy” dimacs2pprn instances

The results are also depicted in Chart 14, Chart 15, Chart 16, Chart 17, Chart 18 and Chart 19, where the ratio of the running times of Cplex, PPRN and IPM vs MMCFB is reported for “hard” and “easy” instances separatedly: it is interesting to note that the relative efficiency of MMCFB w.r.t. the simplex-based codes for $k = 4$ is lower on easy problems than on hard ones, while the converse is true for the relative efficiency of w.r.t. IPM.

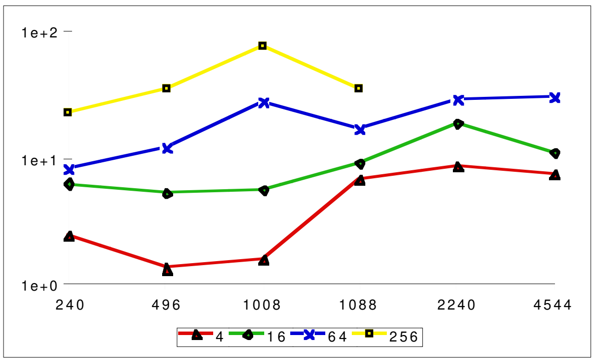

Chart 14: Cplex / MMCFB ratio for easy problems

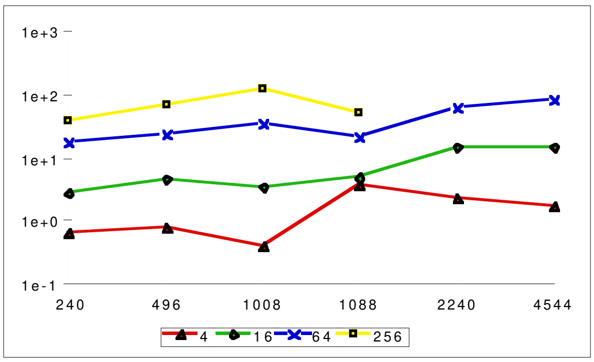

Chart 15: Cplex / MMCFB ratio for hard problems

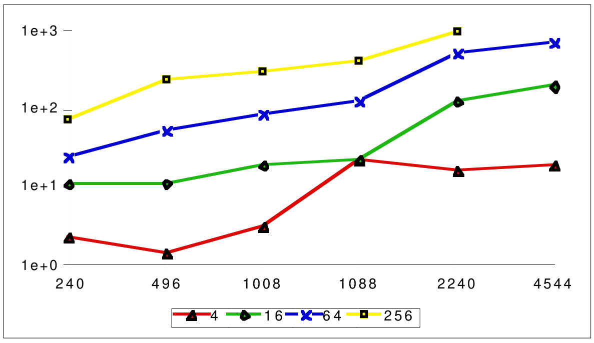

Chart 16: PPRN / MMCFB ratio for easy problems

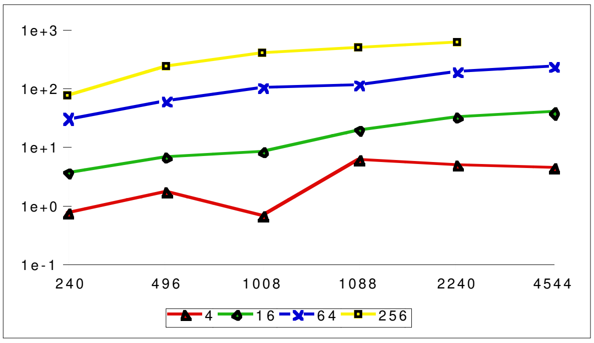

Chart 17: PPRN / MMCFB ratio for hard problems

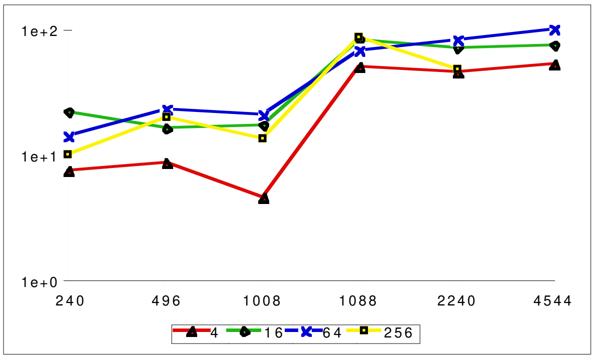

Chart 18: IPM / MMCFB ratio for easy problems

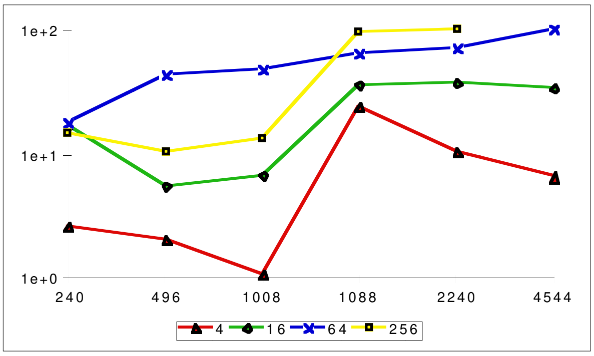

Chart 19: IPM / MMCFB ratio for hard problems

II.2.4.6 The BHV problems

	MMCFB		Cplex	BCG	
	Iter	*Time*	*Time*	*Iter*	*Time*
q1	42	0.63	7.93	4	5.06
q2	34	0.51	9.03	5	4.15
q3	42	0.63	12.45	5	5.15
q4	55	0.83	16.08	6	6.78
q5	85	1.33	20.3	6	6.78
q6	100	1.55	26.75	7	7.78
q7	100	1.58	31.07	7	8.69
q8	156	2.66	37.41	8	10.66
q9	221	3.76	44.96	8	10.60
q10	172	2.92	51.95	9	12.88
q11	196	3.37	61.58	8	12.55
q12	270	4.75	64.96	9	15.95

Table 20: results for the “px” problems

	MMCFB		Cplex	BCG	
	Iter	*Time*	*Time*	*Iter*	*Time*
r10-5-1	25	8.73	6570.19	4391	662
r10-5-2	26	9.01	9249.35	4208	607
r10-5-3	38	13.02	6598.73	3237	398
r10-5-4	20	7.14	8481.97	4149	501
r10-5-5	23	8.16	9262.55	3633	420
r10-55-1	877	311.01	10843.10	2455	162
r10-55-2	934	330.21	12978.80	2690	199
r10-55-3	1243	445.27	13849.00	2694	224
r10-55-4	999	349.20	11581.60	2511	218
r10-55-5	877	318.00	*	2706	234

Table 21: results for the “r10-*x*-*y*” problems

Among the available solvers, only MMCFB was “naturally” capable of solving undirected instances, but the three general-purpose (LP) solvers could be adapted for the task; due to the previous results, only Cplex was actually run on these data sets. The times for the Column Generation algorithm (*BCG*) were obtained on a IBM RS6000-590 workstation, credited for 121.6 SPECint92 and 259.7 SPECfp92, i.e. essentially twice as fast as the reference machine: Table 20 shows that on the easy “px” problems MMCFB is about a factor of 10 faster than the other approach. The results on the other data set, shown in Table 21, are quite surprising: the two methods are essentially equivalent on the (highly fractional) "r10-55-*y*" instances, but MMCFB is able to solve the similar (integral) "r10-5-*y*" instances 30 times faster, while BCG needed about twice the time. A rationale for the behaviour of MMCFB exists, since a smaller number of trees is likely to be necessary to form the optimal flow as a convex combination in the integral data case, but it is really unclear why the same is not true for the CG approach: should it be proved that this is due to the Bundle-type approach, the result could well be defined encouraging.

II.2.4.7 Conclusions

The above computational experience seems to show that MMCFB can be a valuable mean of solving (MMCF) with many commodities: to stress one figure, MMCFB solved the hard, dense (256, 256) Mnetgen instances in less than 20 *minutes*, but more than 6 and 10 *days* were needed by Cplex and PPRN respectively, while IPM required a little less than 5 *hours*. As the experience with the dimacs2pprn data set shows, (MMCF)s with over one million of variables can be solved in less than 15 minutes of CPU by a middle-end workstation, provided that the subproblems have a (SPT) structure. Conversely, MMCFB does not appear to be a good choice for problems with a small number of commodities, unless on very large instances: the PDS problems show that MMCFB can be competitive even on problems with an m / k ratio larger than 1000, but the speedup w.r.t. simplex-based codes is far smaller than on many-commodities problem.

The general-purpose IP solvers are not competitive with the other approaches, and they are not able to solve even moderately-sized instances due to excessive memory requirements: however, the specialized one dramatically improves on them and it is competitive on difficult problems with few commodities, being also competitive with the simplex-based methods (although not with MMCFB) for larger values of k. Among the latters, Cplex is (sometimes consistently) faster than PPRN on the Mnetgen and dimacs2pprn problems, while the converse is true for the Canad ones: however, on the largest (difficult) PDS problems PPRN neatly outperforms Cplex.

The Column Generation code was found (on a limited set of problems) to be less effective than MMCFB, even though in some cases not dramatically: for the subset of the problems where the speedup is really impressive, it is still not clear which features of the codes determine such difference, and further investigations are needed. Finally, the indirect comparisons with methods from the literature seem to indicate that alternative approaches can be at least comparably efficient, or even much more efficient for rapidly finding approximate solutions, at least on some classes of problems: even more clearly, they show the need for further computational experiences that help in assessing the effectiveness of the various approaches on the very different class of problems that might need to be solved.

Acknowledgements

Many people have contributed to the work described in this Chapter, mainly (Cf. Acknowledgements) by providing me the codes and/or the test problems for the computational comparisons. Here, I wish to acknowledge Claudio Gentile for his excellent work in coding the largest part of the `RelaxIV` class, and for contributing to the definition of the `McfClass` class: that state-of-the-art implementation of an efficient Min-Cost Flow algorithm was clearly crucial for obtaining an overall effective Cost Decomposition code.