Solving a Network Design Problem

Alain Chabrier (achabrier@ilog.fr)
ILOG Spain
Gobelas 21
28023 Madrid
Spain

Emilie Danna (edanna@ilog.fr)
ILOG SA
9 rue de Verdun
F-94253 Gentilly Cedex
France

Laboratoire d'Informatique d'Avignon.
CRNS - FRE 2487
339, chemin des Meinajariès
Agroparc, BP 1228
F-84911 Avignon Cedex 9
France

Claude Le Pape (clepape@ilog.fr) and Laurent Perron (lperron@ilog.fr)
ILOG SA
9 rue de Verdun
F-94253 Gentilly Cedex
France

Abstract. Industrial optimization applications must be "robust," *i.e.*, they must provide good solutions to problem instances of different size and numerical characteristics, and continue to work well when side constraints are added. This paper presents a case study that addresses this requirement and its consequences on the applicability of different optimization techniques. An extensive benchmark suite, built on real network design data, is used to test multiple algorithms for robustness against variations in problem size, numerical characteristics, and side constraints. The experimental results illustrate the performance discrepancies that have occurred and how some have been corrected. In the end, the results suggest that we shall remain very humble when assessing the adequacy of a given algorithm for a given problem, and that a new generation of public optimization benchmark suites is needed for the academic community to attack the issue of algorithm robustness as it is encountered in industrial settings.

Keywords: Network Design, Constraint Programming, Mixed Integer Programming, Branch and Price, Industrial Benchmark

Keywords: Network Design, Constraint Programming, Mixed Integer Programming, Branch and Price, Industrial Benchmark

Introduction

In the design and development of industrial optimization applications, one major concern is that the optimization algorithm must be robust. By "robust," we mean not only that the algorithm must provide "good" solutions to problem instances of different size and numerical characteristics, but also that the algorithm must continue to work well when constraints are added or removed. This expectation is heightened in constraint programming as its inherent flexibility is often put forward as its main advantage over other optimization techniques. Yet this requirement for robustness is rarely recognized as the top priority when the application is designed. Similarly, the benchmark problem suites that are used by the academic community generally do not reflect this requirement. In practice, it has important effects on the reinforcement of problem formulation, search management, the advantages of parallel search, the applicability of different optimization techniques including hybrid combinations, *etc.* This paper presents a specific case study in which such questions are addressed.

An extensive benchmark suite, presented in Section 1, has been built on the basis of real network design data provided by France Telecom R&D (Bernhard et al., 2002). The suite includes three series of problem instances corresponding to different characteristics of the numerical data. In each series, seven instances of different size are provided. In addition, six potential side constraints are defined, leading to 64 versions of each instance. The goal is to design an algorithm which provides the best results on average when launched on each of the $3 * 7 * 64 = 1344$ instances with a CPU time limit of 10 minutes. Indeed, the network designer wants to get a good cost estimate in a few minutes. In some cases, this estimate will be refined through an overnight run. The differences between the 1344 instances make it hard to design an algorithm that performs well on all instances. Note that in the context of the current application, both the introduction of new technologies and the evolution of network usage can have an impact on problem size, numerical characteristics, and side constraints. It is believed that an optimization technique which applies well to all of the 1344 problem instances is more likely to remain applicable in

the future than an optimization technique which performs particularly well on some instances, but fails to provide reasonable solutions on some others. Note also that the 10-minute time limit corresponds to a practical usage purpose. Running an algorithm for 10 minutes on a multiprocessor machine might be perfectly acceptable, provided the cost of the machine is not prohibitive. Most experimental results provided in this paper will correspond to the use of 10 minutes on a 700 MHz PC with 256 MB of RAM or to the use of 10 minutes on a 4*700 MHz PC with 2GB of RAM.

Several rounds of design, implementation, and experimentation have been performed in the context of the ROCOCO project, supported by the French MENRT, while the benchmark was in construction. The aim of the first round (Section 2) was to select a few basic optimization techniques for the problem. This round was focused on the easiest versions of rather small instances. This enabled a detailed examination of the behavior of several algorithms and led to a better understanding of the complex nature of the base problem. The second round (Section 3) extended the study to middle-sized instances with different numerical characteristics and side constraints. We selected, improved, and compared three basic algorithms, based on mixed integer programming, column generation, and constraint programming. At the end of this round, the constraint programming algorithm appeared as the most robust (which does not mean that the other algorithms could no longer be improved).

The practical interest and the challenging nature of the benchmark led us to use it well after the end of the ROCOCO project, as a test case for new problem-solving methods. This includes new heuristics developed in a generic mixed integer programming context, the use of branch-and-price in place of simple column generation, and sequential and parallel implementations of constraint-based large neighborhood search. Sections 4, 5, and 6 present these recent evolutions of the three basic algorithms. Section 7 summarizes the results and concludes. In the end, both Constraint Programming with Large Neighborhood Search and Branch and Price and Cut appear as good algorithms for solving the problem.

1. The Network Design Benchmark

The benchmark problem consists in dimensioning the arcs of a telecommunications network, so that a number of commodities can be simultaneously routed over the network without exceeding the chosen arc capacities. The capacity to be installed on an arc must be chosen in a

discrete set and the cost of each arc depends on the chosen capacity. The objective is to minimize the total cost of the network.

In practice, two main variants of the problem can be considered. In the "mono routing" variant, each commodity must be routed along a unique path; in the "multi flow" variant, each commodity can be split and routed along several paths. We have focused on the mono routing variant which is more difficult and has been less studied in the literature. Rothlauf, Goldberg, and Heinzl (Rothlauf et al., 2002) have worked on a similar problem (data from Deutsche Telekom) but require the resulting network to be a tree, which makes mono routing equivalent to multi flow. Gabrel, Knippel, and Minoux (Gabrel et al., 1999) and Bienstock and Günlük (Bienstock and Günlük, 1996) have developed an exact method for network design problems with a discrete set of possible arc capacities and multi flow routing. To our knowledge, these studies are the closest to the one we report in this paper.

Given are a set of n nodes and a set of m arcs (i, j) between these nodes. A set of d demands (commodities) is also defined. Each demand associates to a pair of nodes (p, q) an integer quantity Dem_{pq} of flow to be routed along a unique path from p to q. In principle, there could be several demands for the same pair (p, q), in which case each demand can be routed along a different path. Yet, to condense notation and keep the problem description easy to read, we will use a triple (p, q, Dem_{pq}) to represent such a demand.

For each arc (i, j), K_{ij} possible capacities $Capa^k_{ij}$, $1 \leq k \leq K_{ij}$, are given, to which we add the null capacity $Capa^0_{ij} = 0$. One and only one of these $K_{ij} + 1$ capacities must be chosen. However, it is permitted to multiply this capacity by an integer between a given minimal value $Wmin^k_{ij}$ and a given maximal value $Wmax^k_{ij}$. Hence, the problem consists in selecting for each arc (i, j) a capacity $Capa^k_{ij}$ and an integer coefficient w^k_{ij} in $[Wmin^k_{ij}, Wmax^k_{ij}]$. The choices made for the arcs (i, j) and (j, i) are linked. If capacity $Capa^k_{ij}$ is retained for arc (i, j) with a non-null coefficient w^k_{ij}, then capacity $Capa^k_{ji}$ must be retained for arc (j, i) with the same coefficient $w^k_{ji} = w^k_{ij}$, and the overall cost for both (i, j) and (j, i) is $w^k_{ij} * Cost^k_{ij}$.

Six classes of side constraints are defined. Each of them is optional, leading to 64 variants of each problem instance, identified by a six-bit vector. For example, "011000" indicates that only the second constraint *nomult* and the third constraint *symdem*, as defined below, are active.

– The security (*sec*) constraint states that some demands must be secured. For each node i, an indicator $Risk_i$ states whether the node is considered "risky" or "secured." Similarly, for each arc

(i, j) and each k, $1 \leq k \leq K_{ij}$, an indicator $Risk_{ij}^k$ states whether the arc (i, j) in configuration k is considered risky or secured. When a demand must be secured, it is forbidden to route this demand through a node or an arc that is not secured.

- The no capacity multiplier (*nomult*) constraint forbids the use of capacity multipliers. For each arc (i, j), two cases must be considered: if there is a k with $Wmin_{ij}^k \geq 1$, the choice of $Capa_{ij}^k$ with multiplier $w_{ij}^k = Wmin_{ij}^k$ is imposed; otherwise, the choice of $Capa_{ij}^k$ is free, but $w_{ij}^k \leq 1$ is imposed.

- The symmetric routing of symmetric demands (*symdem*) constraint states that for each demand from p to q, if there exists a demand from q to p, then the paths used to route these demands must be symmetric. Similarly, if there are several demands between the same nodes p and q, these demands must be routed on the same path.

- The maximal number of bounds (*bmax*) constraint associates to each demand (p, q, Dem_{pq}) a limit $Bmax_{pq}$ on the number of bounds (also called "hops") used to route the demand, *i.e.*, on the number of arcs in the path followed by the demand. In particular, if $Bmax_{pq} = 1$, the demand must be routed directly on the arc (p, q).

- The maximal number of ports (*pmax*) constraint associates to each node i a maximal number of incoming ports Pin_i and a maximal number of outgoing ports $Pout_i$. For each node i, it imposes $\sum_j \sum_{k:Capa_{ij}^k \neq 0} w_{ij}^k \leq Pout_i$ and $\sum_j \sum_{k:Capa_{ji}^k \neq 0} w_{ji}^k \leq Pin_i$.

- The maximal traffic (*tmax*) constraint associates to each node i a limit $Tmax_i$ on the total traffic managed by i. This includes the traffic that starts from i ($\sum_{q \neq i} Dem_{iq}$), the traffic that ends at i ($\sum_{p \neq i} Dem_{pi}$), and the traffic that goes through i (the sum of the demands $Dem_{pq,p \neq i,q \neq i}$, for which the chosen path goes through i). Note that it is possible to transform this constraint into a limit on the traffic that enters i (which must be smaller than or equal to $Tmax_i - \sum_{q \neq i} Dem_{iq}$) or, equivalently, into a limit on the traffic that leaves from i (which must be smaller than or equal to $Tmax_i - \sum_{p \neq i} Dem_{pi}$).

Twenty-one data files, organized in three series, are available. Each data file is identified by its series (A, B, or C) and an integer that indicates the number of nodes of the considered network. Series A includes the smallest instances, from 4 to 10 nodes. Series B and C include larger instances with 10, 11, 12, 15, 16, 20, and 25 nodes.

The optimal solutions to the 64 variants of A04, A05, A06, and A07, are known. Proven optimal solutions are currently available for only 44 variants of A08, one variant of A09, one variant of A10, and 12 variants of C10. This means 1030 instances are still open.

The instances of series B have more choices of capacities than the instances of series A, which have more choices of capacities than the instances of series C. So, in practice, instances of series B tend to be harder because the search space is larger, whereas instances of series C tend to be harder because each mistake has a higher relative cost. Some instances of series C also exhibit special characteristics. For example, in C10, constraints on the maximal number of ports per node impose that the overall network is either a chain or a single loop; C16 consists of extending such a loop; C11 includes cases in which $Capa_{ij}^k = 0$ and $Capa_{ji}^k > 0$, which leads to an asymetric situation in terms of number of ports; and C20 includes two types of traffic, with different security and number of bounds constraints, between the same nodes.

2. Solving the Basic Problem

In the first round, five algorithms were developed and tested on the simplest instances of the problem: series A with only the *nomult* and *symdem* constraints active. Focusing on simple instances enabled us to compute the optimal solutions of these instances and trace the behavior of algorithms on the basic problem, without being confused by side constraints. The drawback is that focusing on simple instances does not allow the anticipation of the effect of side constraints. The same remark holds for problem size: it is easier to understand what algorithmically happens on small problems, but some algorithmic behaviors show up on large problems that are not observable on smaller problems. Five algorithms were tested:

- MIP: the CPLEX(Cplex, 2001) mixed integer programming algorithm, with the emphasis on finding feasible solutions, applied to a minor variant of the MIP formulation given in (Bernhard et al., 2002).

- CG: a column generation algorithm, which consists of progressively generating possible paths for each demand and possible capacities for each arc. At each iteration, a linear programming solver is used to select paths and capacities and guide the generation of new paths and new capacities. In addition, a mixed integer version of the linear program is regularly used to generate deasible solutions.

– CP: a simple constraint programming algorithm, based on ILOG Solver(Solver, 2002) integer and set variables, developed by France Telecom R&D.

– CP-PATH: a more complex algorithm which combines classical constraint programming with a shortest path algorithm.

– GA: an ad-hoc genetic algorithm.

Optimal solutions were found using the CPLEX algorithm, version 7.5, with no CPU time limit. For the A10 instance, however, the CPLEX team at ILOG suggested a different set of parameters for the CPLEX MIP (emphasis on optimality, strong branching) which resulted in fewer nodes being explored. With this parameterization, the first integer solution was found in more than three hours. The optimal solution was found in more than six days. Further work, with intermediate (beta) versions of CPLEX, showed that this could be reduced to a few hours. Yet at this point we do not believe the problem can be exactly solved in 10 minutes or less.

Table I provides, for each instance, the optimal solution and the value of the best solution found by each algorithm within the CPU time limit. The last column provides the mean relative error (MRE) of each algorithm: for each algorithm, we compute for each instance the relative distance $(c - o)/o$ between the cost c of the proposed solution and the optimal cost (or best known solution if the optimal solution is not known) o, and report the average value of $(c - o)/o$ over the 7 instances.

Table I. Initial results on series A, parameter 011000

	A04	A05	A06	A07	A08	A09	A10	MRE
Optimum	22267	30744	37716	47728	56576	70885	82306	
MIP	22267	30744	37716	47728	56576	73180	99438	3.4%
CG	22267	30744	37716	47728	57185	72133	87148	1.2%
CP	22267	30744	37716	49812	74127	97386	104316	14.2%
CP-PATH	22267	30744	37716	47728	56576	70885	83446	0.2%
GA	22267	30744	37716	48716	60631	75527	88650	3.4%

3. Extensions and Tests with Side Constraints

In the second round, the study was extended to the middle-sized instances (10 to 12 nodes) and, most importantly, to the six side constraints. We decided to focus mostly on three algorithms, MIP, CG, and CP-PATH. Indeed, given the previous results, CG and CP-PATH appeared as the most promising. The MIP algorithm was *a priori* less promising, but different ideas for improving it had emerged during the first round, and it had also provided us with optimal solutions, although with much longer CPU time. This section describes the main difficulties we encountered in extending these algorithms to the six side constraints of the benchmark.

3.1. Mixed Integer Programming

Various difficulties emerged with the first tests of the MIP algorithm. First, no solution was found in 10 minutes on A10 with parameters "010111" and "110111," *i.e.*, when *nomult*, *bmax*, *pmax*, and *tmax* are active, and *symdem* (which roughly divides the problem size by 2) is not. On the A series, the results also showed a degradation of performance when *bmax* and *tmax* were active.

Numerous attemps were made to improve the situation. First, we tried to add "cuts," *i.e.*, redundant constraints that might help the MIP algorithm:

- For each demand and each node, at most one arc entering (or leaving) the node can be used.
- For each node, the sum of the capacities of the arcs entering (or leaving) the node must be greater than or equal to the sum of the demands arriving at (or starting from) the node plus the sum of the demands traversing the node.
- For each demand and each arc, the routing of the demand through the arc excludes, for this arc, the capacity levels strictly inferior to the demand.

In general, these cuts resulted in an improvement of the lower bounds, but did not allow the generation of better solutions within the time limit of 10 minutes. We eventually removed them.

A cumulative MIP formulation (CMIP) of arc capacity levels was also tested. Rather than using a 0-1 variable y^k for each level k, this formulation uses a 0-1 variable δ^k to represent the decision to go from one capacity level to the next, *i.e.*, $\delta^k = y^{k+1} - y^k$. As for the cuts, the main effect of this change was an improvement of lower bounds.

We also tried to program a search strategy inspired by the one used in the CP-PATH algorithm. This allowed the program to generate solutions more often in less than 10 minutes, but the solutions were of poor quality.

Hence, the results are globally not satisfactory. However, the MIP algorithm sometimes finds better solutions than the CP-PATH algorithm. For example, on C11, the MIP algorithm (with the cumulative formulation) generates solutions within 10 minutes for only 31 variants out of 64. But amount these 31 variants, the MIP solution is better that the solution obtained by the CP-PATH in 18 cases. In such configuration, it might be worthwhile applying both algorithms and keeping the best overall solution.

3.2. Column Generation

The six side constraints are integrated in very different ways within the column generation algorithm:

- The *symdem* constraint halves the number of routes that need to be built. The presence of this constraint simplifies the problem.
- The *bmax* constraint is directly integrated in the column generation subproblem. For each demand Dem_{pq}, only paths with at most $Bmax_{pq}$ arcs must be considered.
- Similarly, the *nomult* constraint is used to limit the number of capacity levels to consider for each arc.
- The *pmax* and *tmax* constraints are directly integrated in the master linear program. They cause no particular difficulty for the column generation method *per se*, but make it harder to generate integer solutions.
- The *sec* constraint is the hardest to integrate. Constraints linking the choice of a path for a given demand and the choice of a capacity level for a given arc can be added to the master linear program when the relevant columns are added. But, before that, the impact of these constraints on the economic value of a path cannot be evaluated, which means that many paths which are not really interesting can be generated. This slows down the overall column generation process. Also, just as for *pmax* and *tmax*, the addition of the *sec* constraint makes integer solutions harder to generate.

The first results were very bad. In most cases, no solution was obtained within the 10 minutes. This was improved by calling the mixed

integer version of the master linear program at each iteration, each time with a CPU time limit evolving quadratically with the number of performed iterations. This enabled the generation of more solutions, but sometimes resulted in a degradation of the quality of the generated solutions (MRE of 2.1% in place of 1.2% for the seven instances used in the first round). Also, the algorithm remained unable to find a solution to A10 with parameter "100011" in less than 10 minutes. This is the parameter for which the *sec*, *pmax*, and *tmax* constraints are active, while the *nomult*, *symdem*, and *bmax* constraints, which tend to help column generation, are not active. Over B10, B11, B12, C10, C11 and C12, 128 such failures occur. The *pmax* constraint is active in 126 of these cases. In the others, both *sec* and *tmax* are active.

3.3. Constraint Programming with Shortest Paths

3.3.1. *A Graph Extension to Constraint Programming*

To simplify the implementation, we basically introduced a new type of variable representing a path from a given node p to a given node q of a graph. More precisely, a path is represented by two set variables, representing the set of nodes and the set of arcs of the path, and constraints between these two variables.

- If an arc belongs to the path, its two extremities belong to the path.
- One and only one arc leaving p must belong to the path.
- One and only one arc entering q must belong to the path.
- If a node i, $i \neq p, i \neq q$, belongs to the path, then one and only one arc entering i and one and only one arc leaving i must belong to the path.

Several global constraints have been implemented on such path variables to determine nodes and arcs that must belong to a given path (*i.e.*, for connexity reasons), to eliminate nodes and arcs that cannot belong to a given path, and to relate the path variables to other variables of the problem, representing the capacities and security levels of each arc.

3.3.2. *Solving the Network Design Problem with the Graph Library*

At each step of the CP-PATH algorithm, we choose an uninstantiated path for which the demand Dem_{pq} was greatest. We then determine the path with the smallest marginal cost to route this demand (to this end, we solve a constrained shortest path problem). A choice point is then created. In the left branch, we constrain the demand to go through the

last uninstantiated arc of this shortest path. In case of backtracking, we disallow this same arc for this demand. Once a demand is completely instantiated, we switch to the next one. A new solution is obtained when all demands are routed. The optimization process then continues in Discrepancy-Bounded Depth-First Search (DBDFS(Beck and Perron, 2000)) with a new upper bound on the objective.

First experiments exhibited the following difficulties: (1) Performances deteriorated when the $tmax$ constraint was active; (2) For 3 sets of parameters on the A10 instance, the algorithm was unable to find a feasible solution in less than 10 minutes. In fact, it turned out that the combination of the maximal number of ports constraint ($pmax$) and the maximal traffic constraint ($tmax$) made the problem quite difficult. (3) Bad results on B10 stemmed from an asymmetric level of traffic. For example, between the first two nodes of the B10 instance, the traffic was equal to 186 in one direction and 14 in the other. (4) Performance was unsatisfactory in the presence of the maximal number of bounds constraint ($bmax$).

Several modifications of the program thus became necessary. First, a "scalar product"-like constraint was implemented. This constraint directly links the traffic at each node with the paths used for the routing. This constraint propagates directly from the variable representing the traffic at each node to the variables representing the demands, and vice versa, without the intermediate use of the traffic on each arc. This allowed more constraint propagation to take place and solved difficulties (1) and (2), even though the combination of the $pmax$ and $tmax$ constraints remains "difficult."

The third difficulty (3) was partly resolved by modifying the order in which the various demands are routed. In the initial algorithm, the biggest demand was routed first. Given a network with 6 nodes and the demands $Dem_{01} = 1800$, $Dem_{10} = 950$, $Dem_{23} = 1000$, $Dem_{32} = 1000$, $Dem_{45} = 1900$ and $Dem_{54} = 50$, the previous heuristic behaved as follows:

- In the case of symmetrical routing, ($symdem = true$), the demands are routed in the following order: Dem_{01} and Dem_{10}, then Dem_{23} and Dem_{32}, then Dem_{45} and Dem_{54}.

- In the case of nonsymmetrical routing, ($symdem = false$), the order is Dem_{45}, Dem_{01}, Dem_{23}, Dem_{32}, Dem_{10}, Dem_{54}.

In the case of symmetrical routing, it is a pity to wait so long before routing Dem_{45}, as a large capacity will be needed to route this demand.

Likewise, in the case of nonsymmetrical routing, it could be worthwhile to route Dem_{10} before Dem_{23} and Dem_{32}, given that the routing

of Dem_{01} has created a path which is probably more advantageous to use for Dem_{10} than for Dem_{23} and Dem_{32}.

The heuristic was therefore modified:

- In the case of symmetrical routing, each demand and its reverse demand are grouped and routed together. The weight of these demands is then the sum of the smallest demand plus twice the biggest demand. The demands are then ordered by decreasing weight. This results in the following order: Dem_{01} and Dem_{10}, then Dem_{45} and Dem_{54}, and finally Dem_{23} and Dem_{32}.

- In the case of nonsymmetrical routing, the weight of each demand is the sum of twice the considered demand plus the reverse demand. This results in the following order: Dem_{01}, Dem_{45}, Dem_{10}, Dem_{23}, Dem_{32}, Dem_{54}.

The average gain on the B10 instance is 3%. On the C instances, however, this change deteriorated performances by roughly 1%. The new heuristic was therefore kept, although it was not a complete answer to the previous problem.

The last difficulty was solved by strengthening constraint propagation on the length of each path. The following algorithm was used in order to identify the nodes and the arcs through which a demand from p to q needs to be routed:

- We use the Ford algorithm (as described in (Gondran and Minoux, 1995)) to identify the shortest admissible path between p and each node of the graph, and between each node of the graph and q.

- We use the path lengths computed in this way to (i) eliminate nodes through which no path of length less than $Bmax_{pq}$ arcs can pass and (ii) mark nodes such that the demand can be routed around the node by a path of length less than $Bmax_{pq}$ arcs.

- We use the Ford algorithm again on each unmarked node to determine if there exists a path from p to q with less than $Bmax_{pq}$ arcs not going through the node.

Using this algorithm was finally worthwhile, although its worst case complexity is $O(nmBmax_{pq})$, *i.e.* $O(n^4)$. The best improvement was of 1.73% on the 64 variations of the B12 problem, meaning an improvement of 3.46% on the 32 variations where $bmax$ is active. On average, this modification also improved the results on the C series. Nevertheless, on the C12 problem, the results were worse by a factor of 0.4%.

3.4. Experimental Results

Tables II and III summarize the results on series A and on the instances with 10, 11, and 12 nodes of series B and C. There are four lines per algorithm. The "Proofs" line indicates the number of parameter values for which the algorithm found the optimal solution and made the proof of optimality. The "Best" line indicates the number of parameter values for which the algorithm found the best solution known today. The "Sum" line provides the sum of the costs of the solutions found for the 64 values of the parameter. A "Fail" in this line signifies that for f values of the parameter, the algorithm was not able to generate any solution within the 10 minutes. The number of failures f is denoted within parentheses. Finally, the "MRE" line provides the mean relative error between the solutions found by the algorithm and the best solutions known today. Notice that the MRE is given relative to the best solutions known today, found either by one of the four algorithms in the table or by other algorithms, in some cases with more CPU time. These reference solutions may not be optimal, so all the four algorithms might in fact be farther from the optimal solutions. (Actually, the numbers that appear in Tables II and III are, for this reason, greater than those that appeared in previous papers.) Note also that each algorithm is the result of a few modifications of the algorithm initially applied to the instances of series A with parameter "011000." Similar efforts have been made for each of them. Yet it is obvious that further work on each of them might lead to further improvements.

The differences with the results of Section 2 are worth noting: a large degradation of performance with the introduction of side constraints and the increase in problem size; and important variations with the numerical characteristics of the problem as shown by the differences between A10, B10, and C10.

4. Improving the MIP approach with new heuristics

It appears that MIP is not the algorithm of choice for solving this benchmark of problems:

- the continuous relaxation is of poor quality. This is mainly a consequence of (a) the problem is a mono routing problem while the relaxation is in fact a simpler flow problem and (b) the cost function is relaxed to a linear one which may be quite far from the reality.

Table II. Solutions found in 10 minutes, series A, for 64 parameter values

		A04	A05	A06	A07	A08	A09	A10	Total
MIP	Proofs	64	64	64	27	1	0	0	220
	Best	64	64	64	27	13	2	0	234
	Sum	1782558	2351778	2708264	3318572	4219647	5640421	Fail (2)	Fail (2)
	MRE	0.00%	0.00%	0.00%	0.88%	4.21%	13.94%	33.84%	7.43%
CMIP	Proofs	64	64	64	7	0	0	0	199
	Best	64	64	64	31	3	0	0	226
	Sum	1782558	2351778	2708264	3337284	4417611	5934187	Fail (3)	Fail (3)
	MRE	0.00%	0.00%	0.00%	1.42%	9.07%	19.77%	29.57%	8.41%
CG	Proofs	64	64	36	20	0	0	0	184
	Best	64	64	64	45	12	1	1	251
	Sum	1782558	2351778	2708264	3310007	4263830	5621264	Fail (1)	Fail (1)
	MRE	0.00%	0.00%	0.00%	0.60%	5.13%	13.17%	22.76%	5.91%
CP-P	Proofs	64	64	64	33	7	0	0	232
	Best	64	64	64	62	43	18	17	332
	Sum	1782558	2351778	2708264	3290940	4076785	5027246	5934297	25171868
	MRE	0.00%	0.00%	0.00%	0.01%	0.70%	1.52%	2.20%	0.63%

Table III. Solutions found in 10 minutes, series B and C, for 64 parameter values

		B10	B11	B12	C10	C11	C12
MIP	Proofs	0	0	0	0	0	0
	Best	0	0	0	4	0	0
	Sum	Fail (16)	Fail (20)	Fail (39)	Fail (10)	Fail (24)	Fail (63)
	MRE	30.56%	33.76%	27.27%	13.32%	57.97%	23.50%
CMIP	Proofs	0	0	0	0	0	0
	Best	0	0	0	0	0	0
	Sum	Fail (14)	Fail (26)	Fail (29)	Fail (15)	Fail (33)	Fail (42)
	MRE	20.46%	24.87%	30.19%	11.67%	16.57%	34.70%
CG	Proofs	0	0	0	0	0	0
	Best	0	0	0	0	0	0
	Sum	Fail (26)	Fail (24)	Fail (19)	Fail (32)	Fail (10)	Fail (17)
	MRE	34.03%	49.16%	61.15%	30.15%	88.96%	36.28%
CP-P	Proofs	0	0	0	10	0	0
	Best	0	0	0	20	0	0
	Sum	1626006	3080608	2571936	1110966	2008833	2825499
	MRE	14.64%	22.54%	18.66%	6.34%	17.47%	23.16%

– when the size of the network increases, solving the continuous relaxation itself is difficult: beyond 15 nodes, it takes at least a third of the allowed time.

– only a few integer solutions (if any) are found in 10 minutes and they are far above the best known solutions in most cases.

There was a lot of room for improvement in this last item so we experimented three new strategies described in detail in (Danna et al., 2003)

that proved to be useful for improving the quality of integer solutions in the case of very difficult MIPs:

- *local branching* (Fischetti and Lodi, 2002) is a local improvement heuristic that solves the original MIP with an added cut stating that, in subsequent solutions, less than k variables can take a value that differs from their value in the current integer solution. This temporarily reduces the search space to a sub-space where good solutions are likely to be found.

- *Relaxation Induced Neighborhood Search* (RINS)(Danna et al., 2003) is a heuristic that fixes the variables that have common values in the current continuous relaxation and in the current integer solution and solves a sub-MIP on the rest of the variables. Local branching can only be called each time the MIP finds a new integer solution. In contrast, RINS can be called at each node of the original branch-and-bound tree, which is of high importance in this benchmark where CPLEX rarely finds integer solutions.

- *guided dives*(Danna et al., 2003) is a tree traversal strategy based on depth first search that always chooses the branch in which the decisions in the current integer solution are valid. Specifically, given a choice between two child nodes, one where the branching variable is fixed to 0 and the other where it is fixed to 1, this strategy always chooses the child node where the value of the branching variable is equal to the value of that variable in the current integer solution. Thus, this strategy is constantly steering the search towards the neighborhood of the current integer solution.

It appears that RINS outperformed the two other strategies two main reasons. First, it is much stronger at improving bad solutions and on this benchmark, CPLEX first solutions can be of extremely poor quality. Second, it is strong at producing quickly good solutions and it can thus take advantage of the short time limit.

However, RINS still needs a first integer solution to be called, but CPLEX often fails to find any integer solution within the time limit. Therefore, we implemented an additional heuristic to find a first integer solution: fix all variables with integer values in the current relaxation, solve a sub-MIP on the rest of variables. If this fixing heuristic does not find an integer solution when called at the root node, it can be called at subsequent nodes of the general branch and bound tree.

On some models, the fixing heuristic finds integer solutions whereas CPLEX does not. Since this heuristic is expensive, it happens in turn,

although more rarely, that CPLEX finds an integer solution with its cheaper heuristics or because it has more time to branch whereas the fixing heuristic fails to build a first integer solution. Therefore, the MRE is compared only on the instances for which each method yields at least one integer solution.

Table IV. Solutions found in 10 minutes, series A, for 64 parameter values

Algorithm	A04	A05	A06	A07	A08	A09	A10
Fails CPLEX 8.0	0	0	0	0	0	3	14
Fails CPLEX 8.0 + RINS + fix	0	0	0	0	0	0	5
MRE Improvement	0.00%	0.00%	-0.03%	1.51%	8.04%	8.13%	13.69%

Table V. Solutions found in 10 minutes, series B and C, for 64 parameter values

Algorithm	B10	B11	B12	C10	C11	C12
Fails CPLEX 8.0	45	51	38	27	51	36
Fails CPLEX 8.0 + RINS + fix	35	38	20	28	31	22
MRE Improvement	15.88%	31.95%	32.99%	38.16%	15.19%	29.40%

Tables IV and V show the results obtained with a time limit of 5 minutes on a 1.5 GHz Pentium IV. They show that the fixing heuristic greatly reduces the number of instances where the MIP does not find any solution (*Fails* line) and that RINS significantly improves the quality of the integer solution found. Notice that these results are not compatible with those of Section 3, because a more recent version of CPLEX (8.0) was used, with different emphasis settings.

The final results are still far from those obtained with constraint programming. Indeed, MIP performances were so bad to start with that the new heuristics only improve on the overall best known solutions in 3 instances. The average MRE obtained on series A is 5.33%. On some instances of series B and C, CPLEX spends a long time at the root node and does not allow for RINS to generate good solutions within the time limit. Different settings of CPLEX parameters could be worth investigating.

Other possible improvements include a hybridization with constraint programming: run constraint programming in the first 5 minutes for example, and use its best solution instead of the poor CPLEX first solution as the starting point for RINS. It remains to be seen whether RINS outperforms other algorithms in improving this intermediate solution.

There seems to be little hope of solving big problems with a MIP approach because the continuous relaxation itself is too long to solve. *A priori* decomposition approaches might be successful: partition the network into several sub-networks with a criterion yet to be found, solve the corresponding sub-MIPs and combine the sub-solutions into a global solution.

5. Improving Column Generation with Branch and Price and Cut

As our first column generation model was not satisfactory when some of the side constraints were activated, we embedded the generation of valid paths and capacities into a branch-and-bound search. This kind of procedure where column and cuts can be generated at any node is referred to as *Branch-and-Price-and-Cut* (BPC). The same decomposed model is used but all the *capacity* columns are added at the beginning and dynamic generation of columns is only applied to *path* columns. The algorithm has been developed in three phases. First, an initial Branch-and-Price algorithm is implemented using a pair of branching rules. Then, several cutting planes are introduced that improve the lower bounds available at each node. Finally, several heuristic modifications are introduced so that good solutions can be robustly found in a limited amount of time.

5.1. Branch-and-Price

Two different branching rules have been used. The first one applies to capacity columns and branches on the capacity level than can be used for some given arc. The other applies to path columns and is similar to the typical rules used in path-based column generation models. Our strategy has been to first look for violated capacity rules and then for violated path rules.

We used a more complex pricing scheme than in our first column generation algorithm. A pool of promising columns is created and filled during the initialization with all paths smaller than some fixed length. When a new column is needed, the pool is first explored, and then several heuristically limited generators are invoked in increasing order of complexity. As soon as new favorable columns are found, they are added to the current node and to the pool, so that they can be quickly reused at some other node. This pool allowed an efficient sharing of good columns between nodes as complex generators are less invoked. Finally, if all heuristic generators fail, a complete enumerative generator is used to ensure the completeness of the pricing procedure.

In the Branch-and-Price context, only path rules result in a modification of the pricing algorithms which consists in removing some arcs of the underlying graph.

5.2. Branch-and-Price-and-Cut

We used four different classes of cutting planes. The three first classes only apply to capacity columns and hence do not require the modification of the pricing algorithms.

- *Set* cuts use a lower bound on total capacity of links opened that cross the frontier of some given set of nodes;
- *BigDemand* cuts use a lower bound on the number of links opened with a certain minimal capacity that cross the frontier of some given set of nodes;
- *Connected* cuts use a lower bound on the number of links opened that have at least one extremity in a given set of nodes which respect some connectivity property on the associated demands;
- *Link* cuts constrain the link value for some arc to be greater than the path value for each path going through that arc.

Each cutting plane class is described in more details in (Chabrier, 2003).

The separation algorithm used is very simple. Violation is tested for every possible cutting planes on all sets of size lower than some predefined value. This part of our algorithm could be greatly improved.

The pricing algorithms only need to be modified when the last cut class is used. The dual value of the cutting plane has to be subtracted from the cost of the corresponding arc when generating paths for the corresponding demand.

Table VI gives an idea of the gap reduction obtained using cutting planes at the root node for some A instances with the 011000 configuration. LB_C correspond to the lower bound using only the three first kinds of cuts, and LB_{CL} to the lower bound with all kinds of cuts. However, good feasible solutions were not always found in the given amount of time. We hence applied heuristic reduction to the algorithm.

5.3. Heuristic Branch-and-Price-and-Cut

Even with the good lower bounds resulting in an important pruning of the search tree, the solutions found within the time limit were not

Table VI. Gaps Improvements on some A instances using cuts.

Instance	LB	LB_C	LB_C gap closed	LB_{CL}	LB_{CL} gap closed
A04	11622.49	21033.00	88.41%	21033.00	88.41%
A07	24021.86	36420.20	52.30%	42042.01	76.01%
A10	42094.65	62202.40	50.01%	71813.23	73.91%

good enough. Several heuristic modifications have hence been applied to the algorithm.

We first used a MIP solver in a similar manner than with our first column generation. This operation being costly, we used the MIP solver only at some selected nodes where the percentage of fractional variables is lower than some fixed parameter. Only the columns from the invoking node are used. Most of our final solutions are obtained using this method at a node with a high depth. We think that the modifications implied by the branching help good columns to be priced.

Even with good lower bounds pruning big parts of the search tree, *Depth First Search* can spend much time into a branch corresponding to the same bad N first decisions. A *Limited Discrepancy Search* strategy has been used that allows more different parts of the tree to be explored.

Finally, a pruning criterion is used to discard all nodes where the gap is smaller that the number of fractional variables multiplied by a fixed parameter.

6. Improving the CP approach

A lot of effort was spent on the search part of the CP approach. It consisted in using parallelism to improve the amount of computation done, local search to post-optimize the first solution and Large Neighborhood Search as the final optimization process.

6.1. Exploiting Parallel Computing

ILOG Parallel Solver is a parallel extension of ILOG Solver (Solver, 2002). It was first described in (Perron, 1999). It implements *or*-parallelism on shared memory multiprocessor computers. ILOG Parallel Solver provides services to share a single search tree among workers, ensuring that no worker starves when there are still parts of the search tree to explore, and that each worker is synchronized at the end of the search.

First experiments with ILOG Parallel Solver were actually performed during the first and second rounds. These experiments are descibed in

(Perron, 2002) and (Bernhard et al., 2002). Switching from the sequential version to the parallel version required a minimal code change of a few lines, and so we were immediately able to experiment with parallel methods. The parallel version was run on a four processor Pentium III 700MHz computer, thus exactly 4 times the computer used in our sequential runs.

6.2. Adding Local Search

An analysis of the first solutions found demonstrated that the algorithm had a tendency to build networks having a large number of low-capacity arcs. This turned out to be quite unfortunate as better solutions could be constructed quite easily from them using a smaller number of arcs, but with greater capacities. In the case of bigger instances with homogeneous demands, such mistakes were common and took quite some time to be corrected as the absence of big demands does not help the propagation of the constraints involved in this benchmark. This suggested a postoptimization phase implemented using local search. This was the most natural way to correct these mistakes as it was lightweight both in terms of code and performances. Any other tentative correction of these mistakes through the modification of the heuristic resulted in deteriorated overall quality as specializing the heuristic for one particular instance of the problem had the tendency to make it less robust on the average.

The addition of local search is implemented using a three-part search. The first part consists of a search for a feasible solution where the search would be penalized if it took the decision to open a new arc (by doubling the cost of the arc).

The second phase consists of a postoptimization of this first solution based on local search. This local search phase is implemented on top of the ILOG Solver Local Search framework (Shaw et al., 2000; Shaw et al., 2002). In our case we created a neighborhood which had as neighbors the removal of each arc from the graph. Such a destructive move requires some rerouting to maintain feasibility of the solution. As local and tree-based search mechanisms can be combined in Solver, at each such move we used traditional tree-based search to reroute paths in order to attempt to maintain feasibility. The neighborhood and tree searches are naturally combined in the same search goal. The local search process we employed was entirely greedy. At each stage, we removed the arc from the graph which decreased the cost by the greatest amount (after rerouting), stopping when there was no arc we could remove without being able to legally reroute the traffic. This whole mechanism was coded in less than fifty lines of code.

The last phase is the original optimization tree-based search, but with an improved upper bound.

6.3. Breaking Barriers with Randomization and Large Neighborhood Search

6.3.1. *Using Randomization and Large Neighborhood Search*

The previous three phase schema evolved dramatically as we replaced the third phase (complete tree search) by a large neighborhood search (LNS) schema (Shaw, 1998). Starting from an instantiated solution stating routes for each demand, we chose to freeze a large portion of this solution and to re-optimize the unfrozen part.

The unfrozen part was first choosen in a random way: Given a number n of demands and $size$, the total number of demands, we compute the ratio $\rho = n/size$. Then we iterate on all demands and freeze them with a probability $(1 - \rho)$. In the following, we use $n = 30$ as it gives good results. Then we loop over the optimization part, each time with a new neighborhood.

This technique was subsumed by a structured LNS implementation. This means that the fragments to reoptimize are not chosen randomly but according to a given structure. We adopted and implemented the following schema. We randomly pick two used arcs of the current solution and set the subproblem to re-optimize to be all demands that use any of these two arcs. This simple method is very effective as it re-optimizes each arc and thus concentrates directly on individual components of the total cost.

As described in Section 3.3, the instantiation order between demands is fixed using heuristic weights associated with each demand. We tried to give a random order over demands, hoping that this would correct mistakes in the fixed instantiation order. We believe that the fixed instantiation order, while quite efficient, makes some big mistakes in first routing big unconstrained demands. This means that small critical demands are routed afterwards on a network already full of traffic.

Following the guidelines in (Gomes and Selman, 1997), we experimented with various fast restart policies. This work was described in (Perron, 2003). The resulting fast restart policy limits the search in each iteration loop to be a DBDFS(Beck and Perron, 2000) search truncated both by a maximum number of discrepancies set to one and by a fail limit of 20.

6.3.2. *Using a Portfolio of Algorithms*

While working on the routing of each demand, it appeared that each modification we made that dramatically improved the solution of one

or two problem instances would deteriorate in a significant way the solution of one or two other instances. This was the perfect case for applying portfolios of algorithms (Gomes and Selman, 1997).

Our first implementation of the portfolio of algorithms technique used a round-robin schema. Each large neighborhood search loop uses one algorithm, that is selected in a round-robin way. To implement different algorithms, we examined the routing of each demand. As described in Section 3.3, we compute for each demand a shortest path from the source to the sink of the unrouted part of the demand. The last arc of this shortest path is then chosen and a choice point is created. The left branch of the choice point states that the route must use this arc and the right branch states that this arc is forbidden for this route. This selection is applied until the demand is completely routed. We create a portfolio of algorithms by computing different kinds of penalties on the cost of each arc and by choosing different combinations of standard cost and penalties.

While attractive, this implementation of a portfolio of algorithms tends to waste a lot of resources. Imagine that we have n algorithms and that only one algorithm can improve our routing problem, then we spend $(n-1)/n$ of our time in a speculative and unproductive way. We then decided to implement a specialization mechanism. Given n algorithms A_i, we use an array of integer weights w_i. Initially, each weight is set to 3. We then choose one algorithm against its weight probability $(w_i)/(\sum_j w_j)$. In the event of the success of a LNS loop, the weight of the successful algorithm is increased by 1 (with an upper bound arbitrarily set to 12). In case of repeated failure of an algorithm (in our implementation, 20 consecutive failures), the weight w_i is decreased by 1 (with another arbitrarily chosen lower bound of 2). The result is a specializing schema which concentrates on the successful algorithms for a given problem.

6.3.3. *Parallelizing the LNS phase*
Given the success of the previous sequential methods, we decided to test them on our quad processor 700MHz Pentium III.

The first parallelization of our LNS + random search was very simple. Simply Using the standard API of ILOG Parallel Solver and with minor changes to the sequential code (less than five lines of code), we were able to implement parallel LNS + random + portfolio of algorithms. Unfortunately, this gave poor results. Deeper investigation showed that parallelization was inefficient as, on average, only 2 out of 4 processors were used at a time. This is a consequence of the degenerated nature of the search tree explored while re-optimizing a fragment.

We then decided to investigate another kind of parallelization. We implemented the original portfolio of algorithms (with no specialization schema) where each method was run in parallel. Furthermore, in order to hide latency and idle workers, we decided to use more algorithms than processors (6 algorithms on a 4-processor box). The results were improved a little with this approach. Examination of the computer workload revealed that approximately 60% of the total computing power is used at any given time. This is a little better than the previous naive implementation, but this implementation has a serious flaw: it is not scalable. In fact, it will not be efficient if, for example, there are more processors than different algorithms in use.

Therefore, we decided to implement another schema, which combined ideas from the two previous implementations. We implemented multipoint LNS with specialization. This means that at each LNS loop, the algorithm is chosen using the specialization schema. The instantiation order of the demands is chosen randomly, but in the same way for each worker. However, each worker works on a different fragment of the whole problem to re-optimize, using different randomly chosen parts. Furthermore, in order to hide latency, we use a few more workers than processors (7 workers and 4 processors). This last implementation gave the best results and was kept for the experiments. It combines excellent results, scalability, and robustness. The load was constant between 90% and 100% during the whole search process.

7. Conclusion

Tables VII, VIII, and IX summarize the overall results for different algorithms (CPLS = CP-PATH + LS, CPRLNS = CP-PATH + Random LNS, CPSLNS = CP-PATH + Structured LNS, CPSLNS4 = CP-PATH + Structured LNS + parallel with 4 processors, BPC = Branch and Price and Cut). The current implementation of the BPC algorithm assumes that there is at most one demand between two given nodes, so BPC results on C20 are not available. BPC appears as the best algorithm on the least-constrained series B, while CP-PATH with structured LNS appears as the best algorithm on series A and C. Notice that many of the best-known solutions are not found directly by any algorithm, but inferred from solutions found by these algorithms on more constrained versions of the same problem. These results do not suggest that we have found the ultimate algorithm for this benchmark. On the contrary, we believe that all the algorithms we tried can still be improved, and that there are many other algorithms to design and test on this benchmark.

The following results are run on a 700MHz pentium III in ten minutes for the sequential CP variation, on a quad processor 700MHz in ten minutes for the parallel version and on a 1.13GHz pentium III in five minutes for the BPC version. Note that for this last version, doubling the time to ten minutes and still using a 1.13GHz pentium III leads to a 2% improvement on the obtained results.

Table VII. Solutions found in 10 minutes, series A, for 64 parameter values

Algorithm		A04	A05	A06	A07	A08	A09	A10	Total
CPLS	Best	64	64	64	62	43	18	15	330
	Sum	1782558	2351778	2708264	3290940	4076785	5027246	5934297	25171868
	MRE	0.00%	0.00%	0.00%	0.01%	0.70%	1.53%	2.24%	0.64%
CPRLNS	Best	64	64	64	43	21	0	0	256
	Sum	1782558	2351778	2708264	3305817	4095341	5173110	6150161	25567029
	MRE	0.00%	0.00%	0.00%	0.48%	1.17%	4.45%	5.97%	1.72%
CPSLNS	Best	64	59	60	61	36	23	9	312
	Sum	1782558	2357988	2711536	3292312	4072458	5000657	5923959	25141468
	MRE	0.00%	0.23%	0.12%	0.05%	0.59%	0.97%	2.05%	0.57%
CPSLNS4	Best	64	60	61	64	51	35	18	353
	Sum	1782558	2354226	2709350	3290590	4063772	4978969	5853995	25033460
	MRE	0.00%	0.10%	0.03%	0.00%	0.37%	0.52%	0.84%	0.27%

Table VIII. Solutions found in 10 minutes, series B, for 64 parameter values

Algorithm		B10	B11	B12	B15	B16	B20	B25	Total
CPLS	Best	0	0	0	0	0	0	0	0
	Sum	1610770	3023086	2555469	2616749	2521154	4809675	6878867	24015770
	MRE	13.54%	20.09%	17.71%	22.96%	17.27%	27.42%	21.42%	20.06%
CPRLNS	Best	0	0	0	1	0	0	0	1
	Sum	1559876	2867269	2495065	2545590	2482189	4633576	6792567	23376132
	MRE	10.00%	13.88%	14.87%	19.32%	15.33%	22.63%	19.84%	16.56%
CPSLNS	Best	9	2	0	2	0	0	2	15
	Sum	1462431	2672609	2287097	2359371	2299906	4439418	6760833	22281665
	MRE	3.15%	6.11%	5.52%	10.47%	6.71%	17.61%	19.28%	9.83%
CPSLNS4	Best	19	5	8	3	10	1	5	51
	Sum	1443323	2617710	2254242	2313863	2269249	4361197	6573861	21833445
	MRE	1.83%	4.04%	3.96%	8.45%	5.24%	15.55%	15.99%	7.86%
BPC	Best	0	1	2	14	4	15	6	42
	Sum	1488661	2685764	2307613	2241966	2309586	4038697	6386837	21459124
	MRE	5.11%	6.74%	6.28%	5.61%	7.22%	5.77%	12.26%	7.00%

One of our aims in this paper was to show the type of performance discrepancies that can occur when industrial optimization applications are developed and to describe some types of corrections that can be applied: (1) put more or less emphasis on the generation of admissible solutions; (2) strengthen problem formulation; (3) strengthen constraint

Table IX. Solutions found in 10 minutes, series C, for 64 parameter values

Algorithm		C10	C11	C12	C15	C16	C20	C25	Total
CPLS	Best	20	0	0	0	0	0	0	20
	Sum	1110966	2003101	2801849	4207669	2013729	7218196	7444034	26799544
	MRE	6.34%	17.14%	22.12%	24.15%	16.86%	31.67%	21.20%	19.93%
CPRLNS	Best	18	1	0	0	0	0	0	19
	Sum	1074942	1856626	2603355	3860496	1858298	6990726	7340627	25585070
	MRE	2.85%	8.65%	13.44%	14.84%	7.84%	27.66%	19.55%	13.55%
CPSLNS	Best	31	6	0	9	4	4	2	56
	Sum	1059816	1793732	2406664	3539685	1758174	5846432	6933942	23338445
	MRE	1.35%	5.19%	4.89%	5.04%	2.02%	6.77%	12.10%	5.34%
CPSLNS4	Best	37	5	4	14	8	11	10	89
	Sum	1054491	1775844	2369699	3473590	1755041	5754809	6711360	22894834
	MRE	0.84%	4.19%	3.28%	3.19%	1.85%	4.99%	8.78%	3.87%
BPC	Best	1	1	1	0	5	NA	12	NA
	Sum	1095281	1922720	2443932	3635519	1769432	NA	7002042	NA
	MRE	4.88%	12.36%	6.51%	8.43%	2.70%	NA	14.69%	NA

propagation; (4) adapt variable selection heuristics to symmetries or asymmetries in the problem; (5) use *or*-parallelism; (6) adapt the tree search traversal strategy to the characteristics of the problem; (7) use local search to improve the first solution(s) found by a tree search algorithm. The ability to implement and test such corrections with minimal development effort is crucial.

To design robust algorithms, one needs a wide benchmark suite similar to the one we used during this project. For example, the constraint programming method quickly appeared as globally more efficient than the other approaches, but its counterperformance on larger instances or on series B compared to series A triggered work on both standard local search and large neighborhood search, which eventually became important components of the overall algorithm. It was (and still is) also interesting to compare the results of different algorithms in the presence or in the absence of the various optional constraints. For example, Table X provides for each value of the *sec*, *nomult*, *symdem*, *bmax*, *pmax*, and *tmax* parameters, the mean relative error obtained by "CPSLNS" and "BPC" when the corresponding optional constraint is on or off (mean over 7*32 instances on series B, and 6*32 instances on series C, C20 being omitted). When looking at this table, we shall recall that the relative errors are computed against the best-known solutions, so a small figure indicates that the algorithm under consideration performs well **in comparison** to the other algorithms that have been tested. Several things immediately appear. First, the constraint programming algorithm deals well with the constraints of symmetry, maximal number

of bounds, and maximal number of ports. In the particular case of the maximal number of bounds, branch-and-price even appears not to perform that well when the constraint is active, which means in fact that branch-and-price benefits less than constraint programming from the presence of the constraint. On the other hand, the constraint programming algorithm comparatively does not perform well with the security and maximal node traffic constraints, which suggests that it could be worth improving the propagation of these constraints.

Table X. Mean relative errors when optional constraints are or are not active

Series	B	B	C	C
Algorithm	CPSLNS	BPC	CPSLNS	BPC
sec = 0	8.98%	7.68%	3.50%	9.08%
sec = 1	10.39%	6.85%	6.36%	7.77%
nomult = 0	9.38%	8.89%	4.71%	9.81%
nomult = 1	9.99%	5.63%	5.15%	7.04%
symdem = 0	10.00%	8.48%	4.95%	8.91%
symdem = 1	9.37%	6.04%	4.90%	7.95%
bmax = 0	11.98%	5.99%	5.89%	6.54%
bmax = 1	7.39%	8.54%	3.97%	10.31%
pmax = 0	10.09%	7.68%	5.59%	8.91%
pmax = 1	9.28%	6.84%	4.27%	7.95%
tmax = 0	9.23%	7.73%	4.47%	8.40%
tmax = 1	10.13%	6.80%	5.39%	8.45%

The benchmark suite we used is public. Instances are available at `http://www.prism.uvsq.fr/Rococo`. We believe other benchmark suites of a similar kind are needed for the academic community to attack the issue of algorithm robustness as it is encountered in industrial settings, where data are neither random nor uniform and where the presence of side constraints can require significant adaptations of the basic models and problem-solving techniques found in the literature.

Acknowledgements

This work has been partially financed by the French MENRT, as part of RNRT project ROCOCO. We wish to thank our partners in this project, particularly Jacques Chambon and Raphaël Bernhard from France Telecom R&D, Dominique Barth, Bertrand Le Cun and Thierry Mautor from the PRiSM laboratory, and Claude Lemaréchal from INRIA Rhône-Alpes. The very first CP program was developed by Olivier

Schmeltzer, the initial CP-PATH program by Jean-Charles Régin, and the very first MIP program by Philippe Refalo. Ed Rothberg and Paul Shaw also contributed to the design of the MIP heuristics and of the various local search methods. We thank Jean-Charles, Paul, Philippe, Ed, and the CPLEX team for many enlightening discussions over the course of the ROCOCO project.

List of Abbreviations

The following abbreviations are used in this article:

BPC Branch and Price and Cut

CG Column Generation

CMIP Cumulative MIP formulation of the problem

CP Constraint Programming

CPLS Constraint Programming + local search post optimization of the first solution

CPRLNS CPLS + Large Neighborhood Search with pure random neighborhoods

CPSLNS CPLS + Large Neighborhood Search with structured neighborhoods

CP-PATH A Dedicated graph library built on top of ILOG Solver

DBDFS Discrepancy Bounded Depth First Search

GA Genetic Algorithm

LNS Large Neighborhood Search

LS Local Search

MIP Mixed Integer Programming

RINS Relaxation Induced Neighborhood Search

References

Beck, J. C. and L. Perron: 2000, 'Discrepancy-Bounded Depth First Search'. In: *Proceedings of CP-AI-OR 00.*

Bernhard, R., J. Chambon, C. Lepape, L. Perron, and J. C. Régin: 2002, 'Résolution d'un problème de conception de réseau avec Parallel Solver'. In: *Proceeding of JFPLC*. (In French).

Bienstock, D. and O. Günlük: 1996, 'Capacitated Network Design: Polyhedral Structure and Computation'. *ORSA Journal of Computing* **1996**, 243–260.

Chabrier, A.: 2003, 'Heuristic Branch-and-Price-and-Cut to solve a Network Design Problem'. In: *CPAIOR'03*.

Cplex: 2001, 'ILOG CPLEX 7.5 User's Manual and Reference Manual'. ILOG, S.A.

Danna, E., E. Rothberg, and C. Le Pape: 2003, 'Exploring Relaxation Induced Neighborhoods to Improve MIP Solution'. Technical report, ILOG.

Fischetti, M. and A. Lodi: 2002, 'Local Branching'. In: *Proceedings of the Integer Programming Conference in honor of Egon Balas*.

Gabrel, V., A. Knippel, and M. Minoux: 1999, 'Exact Solution of Multicommodity Network Optimization Problems with General Step Cost Functions'. *Operations Research Letters* **25**, 15–23.

Gomes, C. P. and B. Selman: 1997, 'Algorithm Portfolio Design: Theory vs. Practice'. In: *Proceedings of the Thirteenth Conference On Uncertainty in Artificial Intelligence (UAI-97)*. New Providence, Morgan Kaufmann.

Gondran, M. and M. Minoux: 1995, *Graphes et algorithmes*. Eyrolles.

Perron, L.: 1999, 'Search procedures and parallelism in constraint programming'. In: J. Jaffar (ed.): *Proceedings of CP'99*. pp. 346–360, Springer-Verlag.

Perron, L.: 2002, 'Practical Parallelism in Constraint Programming'. In: *Proceedings of CP-AI-OR 2002*. pp. 261–276.

Perron, L.: 2003, 'Fast Restart Policies and Large Neighborhood Search'. In: *Proceedings of CPAIOR 2003*.

Rothlauf, F., D. E. Goldberg, and A. Heinzl: 2002, 'Network Random Keys: A Tree Representation Scheme for Genetic and Evolutionary Algorithms'. *Evolutionary Computation* **10**(1), 75–97.

Shaw, P.: 1998, 'Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems'. In: M. Maher and J.-F. Puget (eds.): *Proceeding of CP'98*. pp. 417–431, Springer-Verlag.

Shaw, P., V. Furnon, and B. de Backer: 2000, 'A Lightweight Addition to CP Frameworks for Improved Local Search'. In: U. Junker, S. E. Karisch, and T. Fahle (eds.): *Proceedings of CP-AI-OR 2000*.

Shaw, P., V. Furnon, and B. De Backer: 2002, 'A Constraint Programming Toolkit for Local Search'. In: S. Voss and D. L. Woodruff (eds.): *Optimization Software Class Libraries*. Kluwer Academic Publishers, pp. 219–262.

Solver: 2002, 'ILOG Solver 5.3 User's Manual and Reference Manual'. ILOG, S.A.