Robust and Parallel Solving of a Network Design Problem

Claude Le Pape, Laurent Perron, Jean-Charles Régin, and Paul Shaw

ILOG SA
9 rue de Verdun
F-94253 Gentilly Cedex
{clepape,lperron,jcregin,pshaw}@ilog.fr

Abstract. Industrial optimization applications must be "robust," *i.e.*, 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 in which this requirement and its consequences on the applicability of different optimization techniques have been addressed. An extensive benchmark suite, built on real network design data provided by France Telecom R&D, has been 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.

1 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 the inherent flexibility of constraint programming 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 have been addressed.

An extensive benchmark suite, presented in Section 2, has been built on the basis of real network design data provided by France Telecom R&D [?]. 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. In practice, the differences between the 1344 instances make it hard to design an algorithm that performs well on all instances. Notice 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.

Three rounds of design, implementation, and experimentation have been performed while the benchmark was in construction. The aim of the first round (Section 3) 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 4) extended the study to middle-size instances with different numeric characteristics and side constraints. We selected, improved, and compared three basic algorithms, based on constraint programming, mixed integer programming, and column generation. At the end of this round, the constraint programming algorithm appeared as the most robust (which does not mean that the other algorithms can no longer be improved). Finally, the third round (Section 5) aimed at improving this algorithm. The algorithm which currently performs best on the overall benchmark combines constraint programming and local search.

2 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, while in the "multi-flow" variant, each commodity can be split and routed along many paths. We have focused on the mono-routing variant which has been less studied in the literature. Rothlauf, Goldberg, and Heinzl [?] 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 [?] 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-bits 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 $Risk^k_{ij}$ 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 which 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 $Wmin^k_{ij} \geq 1$, the choice of $Capa^k_{ij}$ with multiplier $w^k_{ij} = Wmin^k_{ij}$ is imposed; otherwise, the choice of $Capa^k_{ij}$ is free, but $w^k_{ij} \leq 1$ 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.) In practice, this constraint reduces (in most cases divides by 2) the number of routes to be constructed for the given demands.

- 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, the constraint imposes $\sum_{j,k} w_{ij}^k \leq Pout_i$ and $\sum_{j,k} w_{ji}^k \leq Pin_i$.
- 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). Notice 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 which indicates the number of nodes of the considered network. Series A includes the smallest instances, from 4 to 10 nodes. The optimal solutions to the 64 variants of A04, A05, A06, and A07, are known. At this point, proved optimal solutions are available for only 44 variants of A08, one variant of A09, and one variant of A10. Series B and C include larger instances with 10, 11, 12, 15, 16, 20, and 25 nodes. Proved optimal solutions are available only for 12 variants of C10. 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, while instances of series C tend to be harder because each mistake has a higher relative cost.

3 Application of Multiple Optimization Techniques to the Base 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 base problem, without noise due to side constraints. The drawback is that focusing on simple instances does not allow for 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 which are not observable on smaller problems. Five algorithms were tested:

- CP: a "pure" constraint programming algorithm developed by France Télécom R&D.
- CP-PATH: an hybrid algorithm using ILOG Solver[?] which combines classical constraint programming with a shortest path algorithm.
- MIP: the CPLEX[?] mixed integer programming algorithm, with the emphasis on finding feasible solutions, applied to a natural MIP formulation of the problem.
- CG: a column generation algorithm, which consists in 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 legal solutions.
- GA: an ad-hoc genetic algorithm.

Attempts to use local search to improve the solutions found by the CP-PATH algorithm were also made, with two distinct neighborhoods: (1) reroute one demand, (2) decrease the capacity of an arc, allow the capacity of another arc to increase, reroute all demands with CP-PATH. In practice, these combinations of CP-PATH and local search did not provide better solutions than CP-PATH alone.

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 parameterization of the CPLEX MIP (emphasis on optimality, strong branching) which resulted in many less nodes being explored. With this parameterization, the first integer solution was found in more than three hours, far above the time limit of 10 minutes. 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 1 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 o, and report the average value of $(c-o)/o$ over the 7 instances.

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

Table 1. Initial results on series A, parameter 011000

4 Extensions and Tests with Side Constraints

In the second round, the study was extended to the mid-size instances (10 to 12 nodes) and, most importantly, to the six side constraints. We decided to focus mostly on three algorithms, CP-PATH, CG, and MIP. Indeed, given the previous results, CP-PATH and CG 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.

4.1 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. Therefore, 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 significance 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 current version of the algorithm is still unable to find a solution to A10 with parameter "100011" in less than 10 minutes. This is precisely 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.

4.2 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 divides the problem size by 2) is not. On the A series, the results also show a degradation of performance when $bmax$ and $tmax$ are 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 formulation 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 a 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 a 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, there are only 31 variants out of 64 on which the MIP algorithm (with the cumulative formulation) generates solutions in 10 minutes, but out of these 31 variants, there are 18 for which the solution is better than the solution obtained by CP-PATH. It might be worthwhile applying both algorithms and keeping the best overall solution.

4.3 Constraint Programming with Shortest Paths

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.

Solving the Network Design Problem with Graph Library At each step of the CP-PATH algorithm, we chose an uninstantiated path for which the demand Dem_{pq} was greatest. We then determined the shortest path to route this demand (to this end, we solved a constrained shortest path problem). A choice point was then created. In the left branch, we constrained the demand to go through the last uninstantiated arc of this shortest path. In case of backtrack, we disallowed this same arc for this demand. Once a demand was completely instantiated, we switched to the next one. A new solution was obtained when all demands were routed. The optimization process then continued in Discrepancy-Bounded Depth-First Search (DBDFS[?]) 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 a quite asymmetric traffic. For instance, 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 were thus made 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}, since a large capacity will be needed to route this demand.

Likewise, in the case of nonsymmetrical routing, we can wonder if it would be worthwhile to route Dem_{10} before Dem_{23} and Dem_{32}, given that their routing has surely created a path which is probably more advantageous to use (in the other direction for Dem_{10} than for Dem_{23} and Dem_{32}).

The heuristic was therefore modified:

- In the case of symmetrical routing, the weight of each demand is the sum of twice the biggest demand plus the smallest 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 of the various sets of parameters 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 [?]) 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 this information to eliminate nodes through which no demand can pass and that have a path of length less than $Bmax_{pq}$ arcs.
- We use the path lengths computed in this way to mark nodes such that a demand can be routed around 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 $O(nmBmax_{pq})$, $O(n^4)$. The most spectacular 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%.

4.4 Experimental Results

Tables 2 and 3 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 to date. 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 to date. Notice that the MRE is given relative to the best solutions known to date, 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. 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 3 are worth noticing: a large degradation of performance with the introduction of side constraints and with an increase in problem size; and important variations with the numerical characteristics of the problem as shown by the differences between A10, B10, and C10.

Algorithm		A04	A05	A06	A07	A08	A09	A10	Total
CP-PATHS	Proofs	64	64	64	33	7	0	0	232
	Best	64	64	64	62	43	23	25	345
	Sum	1782558	2351778	2708264	3290940	4076785	5027246	5934297	25171868
	MRE	0.00%	0.00%	0.00%	0.01%	0.69%	1.25%	1.57%	0.50%
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%	4.20%	13.62%	33.11%	7.29%
CUMULATIVE MIP	Proofs	64	64	64	7	0	0	0	199
	Best	64	64	64	31	4	0	0	227
	Sum	1782558	2351778	2708264	3337284	4417611	5934187	Fail (3)	Fail (3)
	MRE	0.00%	0.00%	0.00%	1.42%	9.06%	19.44%	29.02%	8.28%
CG	Proofs	64	64	36	20	0	0	0	184
	Best	64	64	64	45	12	2	1	252
	Sum	1782558	2351778	2708264	3310007	4263830	5621264	Fail (1)	Fail (1)
	MRE	0.00%	0.00%	0.00%	0.60%	5.11%	12.85%	22.09%	5.77%

Table 2. Solutions found in 10 minutes, series A, for 64 parameter values

Algorithm		B10	B11	B12	C10	C11	C12
CP-PATH	Proofs	0	0	0	10	0	0
	Best	4	1	2	20	0	0
	Sum	1626006	3080608	2571936	1110966	2008833	2825499
	MRE	7.96%	10.67%	8.71%	5.77%	11.60%	14.95%
MIP	Proofs	0	0	0	0	0	0
	Best	3	6	1	6	0	0
	Sum	Fail (16)	Fail (20)	Fail (39)	Fail (10)	Fail (24)	Fail (63)
	MRE	23.68%	22.28%	19.24%	12.82%	51.20%	17.42%
CUMULATIVE MIP	Proofs	0	0	0	0	0	0
	Best	3	1	0	1	1	0
	Sum	Fail (14)	Fail (26)	Fail (29)	Fail (15)	Fail (33)	Fail (42)
	MRE	14.16%	14.05%	20.27%	10.92%	12.83%	26.61%
CG	Proofs	0	0	0	0	0	0
	Best	0	0	0	1	0	1
	Sum	Fail (26)	Fail (24)	Fail (19)	Fail (32)	Fail (10)	Fail (17)
	MRE	27.27%	35.49%	47.67%	28.75%	79.42%	27.49%

Table 3. Solutions found in 10 minutes, series B and C, for 64 parameter values

5 Scaling

The results of the second stage have shown that the constraint programming algorithm which calculates the shortest paths is the strongest. As already mentioned, it is obvious that all algorithms can be improved. Yet we decided to focus mostly on improving the CP-PATH algorithm.

5.1 Analysis of Previous Results

Two important elements were acknowledged. First, an analysis of the explored search trees showed that during the search almost all the improving solutions (especially in the B series) questioned one of the first routing decisions that had been taken in order to build the previous solution. This suggested, on the one hand, the construction of a parallel search on a multiprocessor, and, on the other hand, the questioning of the decisions made in the upper part of the tree first. Besides, an analysis of the first found solution 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 of correcting these mistakes as it was lightweight both in term of code and performances. Any other tentative correction of these mistakes through the modification of the heuristics 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.

5.2 Extending and Refining the Previous Framework

Exploiting Parallel Computing ILOG Parallel Solver is a parallel extension of ILOG Solver [?]. It was first described in [?]. It implements *or*-parallelism

on shared memory multi-processor 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 round. These experiments are described in [?] and [?]. 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.

It should also be noted that the parallel version uses four times more processing power than the sequential on the machine we used.

Changing the Search Tree Traversal CP-PATH uses the DBDFS[?] search procedure to explore the search tree. As described in [?], open nodes of the search tree are evaluated and stored in a priority queue according to their evaluation. The DBDFS strategy evaluates nodes by counting the number of right moves from the root of the search tree to the current position in the search tree. Two variations have been implemented to change this evaluation and to add weights to discrepancies (*cf.* [?]). Both variations rely on the depth of the search tree. The first one tries a fixed weight schema, while the second adapts the weight mechanism to focus even more on the top of the search tree in case of a deep search tree.

Adding Local Search The analysis of the second round suggested that the search procedure should be split in three. 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[?,?]. This framework is built upon the essential principles of local search: those of a current solution, a neighborhood structure, ways of exploring this neighborhood structure, move acceptance criteria, and metaheuristics. Each of these concepts translates into one or more Solver objects which can be naturally instantiated for the problem at hand. Like ILOG Solver's other fundamental objects such as constraints and goals, new local search objects, such as neighborhoods, can be defined or refined by users, resulting in a close match between the solving methods and the problem structure. Importantly, local moves are made in ILOG Solver by the application of search goals, in the same manner as for complete search. This facilitates combinations of local and tree-based search, which is in fact what we used here.

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 would then be the original optimization tree-based search, but with an improved upper bound.

5.3 Results

Table 4 gives the results of the four new algorithms, compared to CP-PATH, on the instances of size 10, 11, and 12 of the B and C series. It also provides the results of CP-PATH + LS executed in parallel with four processors. Globally, the best improvement comes from parallelism, closely followed by local search. The two search procedures (variations 1 and 2) which were promising on eight variants of the problems as described in [?] turned out to be not robust enough. They improved the results on the B series but worsened them on the C series.

Algorithm		B10	B11	B12	C10	C11	C12
CP-PATH	Best	4	1	2	20	0	0
	Sum	1626006	3080608	2571936	1110966	2008833	2825499
	MRE	7.96%	10.67%	8.71%	5.77%	11.60%	14.95%
CP-PATH Parallel	Best	6	1	1	36	0	0
	Sum	1597793	3009386	2548122	1084577	2002557	2794864
	MRE	5.98%	8.00%	7.73%	3.28%	11.13%	13.65%
CP-PATH + Variation 1	Best	1	0	4	10	0	0
	Sum	1608752	3015921	2563522	1133098	2024920	2862054
	MRE	6.80%	8.28%	8.32%	7.80%	12.43%	16.42%
CP-PATH + Variation 2	Best	2	2	3	10	0	0
	Sum	1601555	3018046	2547425	1123779	2034917	2838541
	MRE	6.27%	8.32%	7.66%	6.87%	12.99%	15.44%
CP-PATH + LS	Best	4	3	2	20	0	0
	Sum	1610770	3023086	2555469	1110966	2003101	2801849
	MRE	6.91%	8.39%	7.81%	5.77%	11.30%	13.97%
CP-PATH + LS Parallel	Best	4	4	0	34	0	0
	Sum	1592778	2967717	2535516	1085266	2005714	2777129
	MRE	5.55%	6.53%	7.12%	3.36%	11.29%	12.91%

Table 4. Solutions found in 10 minutes, series B and C, for 64 parameters values

We applied the CP-PATH + LS algorithm to the A series and the results were exactly identical to those of CP-PATH, *i.e.*, local search brought no improvement on this series. We also applied CP-PATH and CP-PATH + LS on the larger instances with 15 to 25 nodes. The MRE ranges from 6.24% (on C16) to 38.03% (on B25) for CP-PATH and from 2.71% (on B15) to 7.63% (on C25) for CP-PATH + LS. We believe that these figures underestimate the deviation from the optimal solutions as fewer algorithms provided reasonably good solutions on the larger instances. It is interesting to notice that local search had a significant impact mostly on series B, when the number of possible capacity levels for each arc is the highest.

	B10	B11	B12	C10	C11	C12
sec = 0	5.52%	10.50%	9.43%	2.04%	10.39%	14.94%
sec = 1	5.58%	2.57%	4.81%	4.67%	12.20%	10.88%
nomult = 0	6.75%	8.91%	8.13%	3.26%	11.97%	13.54%
nomult = 1	4.35%	4.16%	6.12%	3.45%	10.61%	12.29%
symdem = 0	7.72%	7.91%	9.58%	4.74%	12.23%	12.50%
symdem = 1	3.38%	5.16%	4.67%	1.97%	10.35%	13.32%
bmax = 0	5.38%	6.23%	6.38%	3.87%	10.71%	15.76%
bmax = 1	5.72%	6.84%	7.86%	2.84%	11.87%	10.06%
pmax = 0	5.88%	7.48%	6.65%	6.69%	12.79%	12.85%
pmax = 1	5.21%	5.59%	7.60%	0.03%	9.79%	12.97%
tmax = 0	4.14%	6.87%	7.48%	2.32%	7.22%	13.03%
tmax = 1	6.95%	6.20%	6.76%	4.40%	15.36%	12.80%

Table 5. Effect of each constraint on the MRE

Table 5 shows the effect of the presence of each constraint on the results. For each optional constraint, it provides the MRE obtained with CP-PATH + LS in Parallel when the constraint is inactive (parameter set to 0) and when the parameter is active (parameter set to 1). Hence, each percentage in the table is the average of 32 numbers. Once again, these figures should be taken with care as the MRE is computed with respect to best known solutions. However, when the MRE is significantly greater when a parameter is set to 1 than when it is set to 0, it indicates that the performance of the algorithm is affected by the presence of the constraint. This occurs with $tmax$ on B10, C10 and C11, and to a lesser extent with *sec*. On the other hand, *nomult* and *symdem* tend to make the problem easier to solve.

6 Conclusion

In this paper, we have presented a case study based on a benchmark aimed at evaluating and improving the robustness of algorithms. The 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.

Our aim in this paper was to show the type of performance discrepancies that can occur when industrial optimization applications are developed and 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 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.

One of the most important aspects of this study has been the ability to implement and test such corrections with minimal development effort.

The results, and our everyday industrial practice, compel us to be modest when stating that an algorithm is appropriate for a given problem. The differences between our initial results on A10 and the results obtained even on instances of the same size like B10 and C10 show that we ought to be cautious.

As mentioned, the benchmark suite we used is public. 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 necessitate significant adaptations of the basic models and problem-solving techniques found in the literature.

7 Acknowledgments

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 Télécom R&D, Dominique Barth from the PRiSM laboratory, and Claude Lemaréchal from INRIA Rhône-Alpes. The very first CP program was developed by Olivier Schmeltzer, the very first CG program by Alain Chabrier, and the very first MIP program by Philippe Réfalo. We thank Alain and Philippe and the CPLEX team for many enlightening discussions over the course of the ROCOCO project.