Parallel Large Neighborhood Search

Laurent Perron and Paul Shaw
ILOG SA
{lperron,pshaw}@ilog.fr

September 2003

Abstract

Industrial optimization applications must be robust: they must provide good solutions to problem instances of different size and numerical characteristics, and must continue to work well when side constraints are added. A good testbed for constructing such a robust solver is a set of network design problem instances recently made public by France Telecom. Together, these instances comprise the desired diversity in the aforementioned qualities of scale, numerical attributes, and additional constraints.

We apply Constraint Programming to the above problems. In order to provide a robust solver, however, we forgo traditional depth first search in favor of another search method typically providing more predictable performance. Large Neighorhood Search (LNS) is a method using the full power of constraint programming, while maintaining the robustness benefits of local search. To further enhance robustness we parallelize our search in a variety of ways. Most interesting is a method based on a portfolio of algorithms which is shown to outperform all previously known CP-based methods for this problem set.

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.

An extensive benchmark suite, presented in two joint papers [5, 11], 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 sizes are provided. In addition, six potential side constraints are defined, leading to 64 versions of each instance, and 1344 in total.

We believe 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.

From our extensive study of the problems in question from a solution construction prespective, *robust* solving specifically requires dealing with three difficulties:

Size: The size of the problem instances varies substantially. This is illustrated by the depth of the first solution found by the CP program. In easy cases, the depth is around 30; in hard cases, it is between 2,000 and 3,000. While exploring the complete search tree is feasible in the first case because propagation helps to cut branches of the tree, it is completely unthinkable in the second case. Moreover, even making a few mistakes early in the search tree is fatal as no search procedure can attempt to correct them in a reasonable time period.

Selection: Deciding in which order demands are to be routed is error prone because of the cumulative nature of the cost. This decision also has major consequences on the minimization process afterward.
Topology: The side constraints and the numerical data make each problem instance very different. This makes it difficult to design an algorithm that is efficient on each aspect of the problem. In practice, we have found that efforts to improve the algorithm on one aspect of the problem typically reduce its robustness as results are deteriorated on instances that do not contain this aspect.

Due to the above difficulties experienced by pure constructive (tree-search) approaches to these problems, we examine a local search method. Although such methods cannot prove optimality, they have been shown to be robust over a large set of problems, for example, see [1]. Here, however, we take a less traditional line, using constraint programming to explore the neighborhood in a technique known as Large Neighborhood Search.

Large Neighborhood Search (LNS) [18] is a local search paradigm based on two main ideas to define and search large neighborhoods. The first key is to define neighborhoods by fixing or *freezing* a part of an existing solution. The elements of the solution that are fixed are usually explicit or implicit variables of the model. For example, in a scheduling model, one may choose to fix the values of the start times of each activity (explicit variables) or one may add additional constraints that force one activity to be scheduled before another (precedence constraints). The rest of the variables, often referred to as a fragment, are *released*: they are free to change their values. The resulting neighborhood is the set of solutions to the sub-problem created. The neighborhood created is usually much larger than typical local search neighborhoods, and can be exponential in the number of released variables.

The size of the so-defined neighborhood requires a powerful algorithm to search it; one cannot reasonably rely on simple enumeration or heuristics to quickly find neighboring solutions. The second key idea of LNS is to use some form of tree search, constraint programming (CP) or mixed integer programming (MIP) to search the neighborhood. However, even with such algorithms, in practice, the tree search is most often truncated with a possibly adaptive time, node, or discrepancy limit. Disrepancy-based searches are often used to try to examine the most promising parts of the search tree early on in search.

LNS methods have been proved to be very successful on a variety of hard combinatorial problems, *e.g.*, vehicle routing [4, 7, 18], scheduling (job-shop: shuffle [2], shifting bottleneck [10], forget-and-extend [6]; RCPSP: block neighborhood [14]), network design [8, 11], frequency allocation [13].

This article discusses the benchmark instances examined, the basic LNS solving approach taken, subsequent efforts to parallelize the basic algorithm, and the problems encountered. We then settle on a parallization based on a portfilio of algorithms and examine its effectiveness on the benchmark suite.

2 A network design problem

The benchmark problem consists of 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.

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 with a pair of nodes (p, q) an integer quantity Dem_{pq} of flow to be routed along a unique path from p to q.

For each arc (i, j), K_{ij} possible capacities $Capa_{ij}^k$, $1 \leq k \leq K_{ij}$, are given, to which we add the zero capacity $Capa_{ij}^0 = 0$. Exactly 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 $Wmin_{ij}^k$ and a given maximal value $Wmax_{ij}^k$. Hence, the problem consists in selecting for each arc (i, j) a capacity $Capa_{ij}^k$ and an integer coefficient w_{ij}^k in $[Wmin_{ij}^k, Wmax_{ij}^k]$. The choices made for the

arcs (i, j) and (j, i) are linked. If capacity $Capa_{ij}^k$ is retained for arc (i, j) with a non-zero coefficient w_{ij}^k, then capacity $Capa_{ji}^k$ must be retained for arc (j, i) with the same coefficient $w_{ji}^k = w_{ij}^k$, and the overall cost for both (i, j) and (j, i) is $w_{ij}^k \times Cost_{ij}^k$.

Six classes of side constraints are defined. Each of them is optional, leading to 64 variants of each problem instance, which we denote 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. Secured demands must be routed through secured arcs.
- The no capacity multiplier (*nomult*) constraint forbids the use of capacity multipliers.
- 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.
- The maximal number of bounds (*bmax*) constraint associates with each demand 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.
- The maximal number of ports (*pmax*) constraint associates with each node i a maximal number of incoming ports Pin_i and a maximal number of outgoing ports $Pout_i$.
- The maximal traffic (*tmax*) constraint associates with each node i a limit $Tmax_i$ on the total traffic managed by i.

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 in 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 series B instances have more choices of capacity those of series A, which in turn have more capacity choices that those of series C. In practice, instances of series B tend to be hard because the search space is larger, while instances of series C tend to be hard because each branching mistake has a higher relative cost.

As described in [11], we have attacked this problem using three approaches: Constraint Programming using ILOG Solver, MIP using ILOG CPLEX and Colum Generation using both. In this article, we focus on the CP approach.

3 Using a subset of the full benchmark

The benchmark problems are specified such that 10 minutes of CPU time are available for each instance; the goal being to produce the best solution possible in those 10 minutes. As the whole suite takes 1344×10 minutes (more than nine days) to run, we first concentrated our efforts on a small subset, which served to calibrate and tune our algorithms. We chose to focus on five particular problem instances:

B10 000000 An average problem with a best CP-found solution of 20510 and the best known solution of 19395.[1]

B10 100111 A problem where a first solution may become very difficult to find if the heuristic is not robust enough. The best CP-found solution is 28083; the best known solution is 25534, found by CPLEX on a MIP formulation.

[1] This second solution is not actually found, but inferred from the solution of a more constrained variation of the problem.

C10 100011 A problem where the first solution may become very difficult to solve if the heuristic is not robust enough. The best CP-found solution is 18925, the same as the best known solution.

C12 000000 A problem where column generation gives better solutions. CP finds 40099 and column generation finds 37385.

C25 100000 A large problem where CP is currently the only technique finding feasible solutions.[2] The best CP solution is 149500 and the best known solution is 109293, inferred from a solution found by CP on a more constrainted version of the example.

We believe this selection, while somewhat arbitrary, will allow us to conduct fruitful experiments. While not all options are represented, these five examples will guide us in our experiments. In the end, we will validate our results by using the best of the breed method on the complete benchmark to evaluate how it performs.

We will display each set of results[3] in this order in an table. For instance, below are the best results from a pure CP approach:

20510	28083	18925	40099	145900

In order to take into account of any randomness involved in the results (due to randomization of the solving algorithms), we will display the results of one standalone run for each experiment that we will choose as representative of the multiple runs. In practice, we choose the median run.

And as a standard, we will always compare to the best known results noted below:

19395	25534	18925	37385	109223

4 Large neighborhood search and randomization

Our first addition to the standard CP framework was a large neighborhood search method [18]. 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 fragment.

The unfrozen fragment was chosen using the following algorithm: 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 $(1 - \rho)$. Throughout the rest of the paper, the $n = 30$.

We continually freeze the randomly chosen part, re-optimizing the remainder, searching for a solution which is better than the best solution previously encountered. To ensure the last condition holds, a constraint on the objective variable is added before each iteration. This allows the CP propagation engine to make additional deductions, accelerating the search.

The above technique was later subsumed by a more structured choice method, of which more details are given in section 7. However, all experiments outside of section 7 are performed using the random choice procedure just described.

4.1 A fast restart strategy

As mentioned in the introduction, it is often practical to limit the CP search for an improving neighbor, the premise being that is can be more beneficial to perform more subproblem selections and partial explorations than to spend a long time on a particular sub-problem. This mechanism is often called a fast restart strategy [9].

A promising technique that can be used is to cut branches from the CP search tree which we heuristically decide as less promising. We examined two approaches:

[2] We expect this to change when our column generation approach is improved.

[3] Each sequential run was conducted on a Pentium III 1.13 GHz running Windows XP, using the Microsoft Visual Studio.NET C++ Compiler.

- The first approach relies on the Discrepancy-Bounded Depth First Search (DBDFS) procedure [3] with a maximum discrepancy usually set to 1.
- The second approach, known as amortized search, uses the search techniques described in [12]. The basic idea is to cut the right branch at each binary choice point with a fixed probability (set to 0.93 throughout this paper).

All experiments on our five target problems are based on these two fast restart methods. [4] The first method (DBDFS) gave the results:

20760	27414	18925	39738	140738

while the second method (amortized search) gave the results:

20188	27592	18925	40099	138929

As we can see, no method strictly dominates the other and they both improve previous results by a small margin.

4.2 Randomizing the instantiation order

As described in [11], 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, can make serious mistakes in the initial routing of large unconstrained demands. This means that small critical demands will be routed afterward on a network already full of traffic.

The DBDFS method was judged best under instantiation order randomization and gave the results:

20778	27417	18925	36603	140922

As we can see, this method gives a significant improvement in the fourth test (C12 000000). There is a deterioration in results on the fifth test. The amortized search process does not greatly benefit from this change.

5 A portfolio of algorithms

While designing the CP search for 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, therefore, seemed the perfect case to which to apply portfolios of algorithms [9].

5.1 Simple implementation of portfolio of algorithms

Our first implementation of the portfolio of algorithms technique was made using a round-robin method. Each large neighborhood search loop would be made using one algorithm, with each algorithm chosen in a round-robin fashion. To implement different algorithms, we examined the routing of each demand. As described in [11], for each demand, we compute 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

[4] Subsequent work [17] has shown that these two methods do not perform better than a more traditional depth first search with a fast restart based on a fail limit. However, fast restarting of DBDFS using a fail limit was shown to outperform both depth first search with a fail limit and the two methods used here.

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.

This method was altered by computing different kinds of penalties on the cost of each arc and by choosing different combinations of standard cost and penalties. This allowed us to create different *algorithms* by applying different coefficients to each component of the cost of one arc.

The standard "cost" is equal to the increase in capacity needed when instantiating a route for a demand. This cost is not very informative because it depends on the instantiation order of demands and thus does not reflect the cumulative nature of the cost function. To remedy this somewhat, we implemented a differerent way of computing a "cost" associated with the decision to route a demand through one arc. For instance, one of these ways computed the relative cost of a route as the ratio (cost of the current arc $\times$ size of the demand / maximum capacity for this arc and this current cost). In total, five different ways of computing marginal cost were implemented. Finally, each of the five + one (standard cost) were associated into seven different cost functions by using seven linear combinations of the atomic costs. All the coefficients involved in these linear combinations were chosen after long investigations on benchmarks suite.

One again, the DBDFS method performed better and gave the results:

20524	27498	18925	38805	131323

This method is most useful for the fifth test, where only the additional algorithms (which are not particulaly usefule on the other problems) were able to go below the 135000 barrier.

5.2 Specialization schema

While attractive, this implementation of a portfolio of algorithms tends to waste a lot of resources. Let us 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 decided to implement a specialization mechanism. Given n algorithms, we use an array of integer weights w. Initially, each weight is set to 3. We then choose each algorithm A_i with probability $(w_i)/(\sum_j w_j)$. In the event of the success of an LNS loop, the weight of the successful algorithm is increased by 1 (with an upper bound 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 a lower bound of 2).

The first method (DBDFS) gave the results:

19627	27249	18925	37495	125327

The second method (amortized) gave the results:

19605	28219	18925	36429	125327

As we can see, this specialization schema is very effective. No test is deteriorated by this approach. Also, there is no clear winner between the DBDFS approach and the amortized search one.

6 Parallelizing the portfolio of algorithms

Given the success of the previous sequential methods, we decided to parallelize our efforts using a four processor 700MHz Pentium III.

6.1 Parallel solver

ILOG Parallel Solver is a parallel extension of ILOG Solver [19] which was first described in [15]. 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 are described in [16] and [5]. 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.

6.2 Simple parallelism

The first parallelization of our LNS approach was very simple. We used the usual API of ILOG Parallel Solver and with minor changes to the sequential code (around five lines of code), we were able to implement parallel LNS + portfolio of algorithms. This approach parallelizes at the level of the resolution of the neighborhood exploration sub-problem, and so is rather fine-grained.

The DBDFS method gave the results:

19960	27357	18598	36963	126530

while the amortized search method gave the results:

20057	27627	18598	37583	125327

These results are rather disappointing. If we do simple math, 4×700 MHz $= 2.8$GHz (compared to 1.13 GHz for our sequential machine). We expected better results. In fact, we have a degradation in the first and second test results, a breakthrough in the third where we improve the best known solution, a small improvement in the fourth test, and a small degradation in the fifth one.

Deeper investigation shows that parallelization is inefficient (only around 50% of the time is spent solving). This is due to the nature of the search trees generated, and on the operation of ILOG Parallel Solver. The trees generated by DBDFS (with a discrepancy limit of 1 as used here) and amortized search are very "left heavy". ILOG Parallel Solver implements work stealing, where a single thread continues the left dive, and the right branches then define sub-trees which are given out to other threads. However, in the two methods used, the right branches do not occur frequently and when they do, they typically define very small sub-trees. These problems result in an increase in swithing overhead and idling. For more traditional 'bushy' search trees, such as those created in standard depth-first search, ILOG Parallel Solver works at much higher efficiency.

6.3 Concurrent execution of the portfolio of algorithms

As shown in the previous section, parallelizing the LNS neighborhood search is inefficient when using DBDFS or amortized search. We therefore decided to investigate another kind of parallelization. We parallize at the portfolio level and run each *algorithm* in the portfolio in parallel with the others. Furthermore, in order to reduce the effects of latency and idle workers, we decided to use more algorithms than processors (6 algorithms on a 4-processor machine). The general mechansim is as follows: First, we choose the LNS subproblem by fixing a some of the problem variables. Then, on the released fragment, we run all algorithms in parallel, looking for an improvment in the global cost from any of the algorithms. When all algorithms have completed, the process is repeated with a new choice of subproblem.

The first method (DBDFS) gave the results:

20326	27573	18598	36507	126530

The second method (amortized search) gave the results:

20173	24940	18925	36582	126530

The results, compared to the previous test, are equivalent in the DBDFS approach and improved in the amortized approach. So, again, the results are disappointing.

Although the algorithms are now run in parallel, the important algorithms, which were previously identified by the specialization schema, are afforded no more time than the others. This is an important cause of the disappointing results.

Regarding parallelization efficiency, study of the computer workload revealed that approximately 60% of the total computing power is used at any given time. This is due to the fact that on the average, some algorithms fail rapidly and do not improve the solution, and typically 40% of the workers are waiting for the others to finish exploring their search tree. This is little better than the previous naive implementation. However, it has another serious flaw—it is not scalable. We cannot simply add more processors to delivery speed ups; we are obliged to concoct new algorithms to occupy them.

6.4 Multi-fragment large neighborhood search

Given the above results, we decided to examined another schema, which combines ideas from the two previous ones; use of parallelism and the specialization schema, while maintaining scalability. The important point to note here is that although diverse algorithms are not particularly simple to generate, the LNS subproblems are, and so this can be used as a way of delivering different work to all processors. Moreover, as there is no obligation here to run different *algorithms* in parallel, the specialization schema can still be used.

In each LNS loop, an algorithm is chosen using the specialization schema. Then, a different LNS fragment is chosen by each processor, and all processors go to work using the chosen algorithm. In addition, as new tasks are so easy to generate, if some workers finish before the slowest, they can restart using a new sub-problem, and continue to do this until the slowest completes, at which point all workers are stopped.

In our experiments, in order to reduce the effects of latency, we use a few more workers than processors (7 workers and 4 processors).

The first method (DBDFS) gave the results:

20021	25382	18448	36191	123628

The second method (amortized search) gave the results:

19855	25412	18592	35931	122190

This last implementation is the best so far. It combines excellent results, scalability, and robustness. The load was constant between 90% and 100% during the whole search process.

7 Results on the complete benchmark

Here we compare full results of the algorithm described in section 6.4 (and an improved variant) against results described other work [5, 11]. We look at four different CP-based methods:

CPLS A constraint programming based method which also includes a neighborhood search which removes arcs from the network and reroutes the demands which passed through them.

CPRLNS A sequential version of the algorithm presented in section 6.4.

CPSLNS Equivalent to CPRLNS, except in the choice of the released fragment, which is more structured than the random choice of CPRLNS. We randomly choose two arcs of the current solution, and then release all routes which use any of the arcs in question. This has the benefit of releasing routes which have something in common, resulting in more potential for

optimization. Importantly, by rerouting all traffic off an arc, one may close arcs. In the standard random route selection scheme in CPRLNS, the chances that one would release all routes passing though an arc are low.

CPSLNS4 Equivalent to CPSLNS, but using four processors.

Tables 1 and 2 summarize the overall results for the above algorithms, noting the number of times an algorithm produced or bettered the best known result, the total cost sum, and the mean percentage above the best known solutions. The results were obtained on a 700MHz Pentium III in ten minutes for the sequential CP variations, and on a four processor 700MHz Pentium III in ten minutes for the parallel version.

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%

Table 1: Mean solutions found in 10 minutes, series B, for all 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%

Table 2: Mean solutions found in 10 minutes, series C, for 64 parameter values

The figures demonstrate a significant improvement in terms of robustness from the combination of parallelism, LNS, randomness, and portfolios of algorithms.

8 The effect of the number of workers

In this section, we vary the number of workers used on six benchmark instances (B10, B15, B20, C12, C16, C25). For each instance, all 64 versions involving all combinations of side constraints were run. We examine the behavior of DBDFS with a fast restart based on a fail limit and the structured neighborhood implementation (CPSLNS4). We list results using from one to eight

	Time (s)										
Threads	30	60	90	120	150	180	210	240	270	300	600
1	28.63%	23.75%	20.52%	18.75%	17.16%	16.02%	12.95%	11.54%	10.53%	9.75%	7.92%
2	28.07%	23.08%	20.02%	17.96%	16.61%	15.67%	12.82%	11.33%	10.43%	9.81%	7.93%
3	25.60%	21.90%	18.66%	16.71%	15.59%	14.63%	11.73%	10.33%	9.37%	8.62%	6.73%
4	24.98%	21.74%	18.76%	16.55%	15.33%	14.38%	11.75%	10.20%	9.26%	8.59%	6.64%
5	27.01%	22.45%	18.60%	16.76%	15.21%	14.49%	11.20%	9.78%	8.83%	8.15%	6.04%
6	38.28%	21.17%	19.17%	17.39%	16.12%	14.95%	11.81%	10.41%	9.26%	8.45%	6.27%
7	23.64%	21.33%	20.57%	17.85%	16.78%	15.54%	11.83%	10.21%	9.15%	8.35%	6.18%
8	24.42%	22.43%	20.02%	18.09%	16.73%	15.90%	11.73%	10.12%	9.24%	8.43%	5.87%

Table 3: Results for 1 to 8 threads on all six problems in ten minutes

workers on a 700MHz four processor pentium III machine. The runs are ten minutes long and we report time in seconds.

The results show, as expected, that solution quality is improved as we move from one to four workers, the latter providing one worker per processor. Quality is further improved, however, from the use of five workers, as the effects of latency are reduced. The use of six or seven workers appears to slightly degrade results, which could be due to additional switching overhead. However, the use of eight workers produces the best results of all. Further examination of this effect is necessary, but an initial conjecture is that the multi-threading system is more efficient with an integer ratio of threads to processors.

9 Conclusion

The contributions of this article can be summarized as follows. First, the use randomness and algorithm portfolios to increase robustness. The use of randomness eliminates any pathological behavior of heuristics, while the portfolios increase the probability that any particular problem will be addressed by an algorithm having the means to solve it well. The use of these two techniques led to significant improvements.

Second, the use of a the specialization mechanism in order to 'match up' the correct algorithms from the portfolio with the problem at hand. This reinforcement learning mechanism was shown to be indispensable when a large number of algorithms were being used.

Third, as our final results show, adding more structure to the initial random approach is important as it improves quality of the neighborhoods generated.

Looking to the future, we would like to extend our multi-fragment LNS paradigm into a full multi-point search, using a population of solutions. This could have various advantages. First, in the generation of initial solutions, parallelizing with Parallel Solver is not efficient as such an initial solution is roughly a dive down the left branch of the search tree with little scope for parallelization. However, in a multi-processor machine, we could use the additional computational resources to generate several solutions simultaneously, with practically no loss of efficiency. During the improvment phase, the presence of a population of solutions would allow different types of decision to be made on where to place resources. One could, for example, dedicate different levels of resource to different solutions as well as to different algorithms. Finally, populations allow the creation of new solutions from existing ones via techniques such as genetic crossover or path relinking. We look forward to exploring such directions in the course of our future work.

References

[1] E. H. L. Aarts and J. K. Lenstra, editors. *Local Search in Combinatorial Optimization*. Wiley, Chichester, 1997.

[2] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3(2):149–156, 1991.

[3] J. Christopher Beck and Laurent Perron. Discrepancy-Bounded Depth First Search. In *Proceedings of CP-AI-OR 00*, March 2000.

[4] R. Bent and P. Van Hentenryck. A two-stage hybrid local search for the vehicle routing problem with time windows. Technical Report CS-01-06, Brown University, september 2001.

[5] Raphaël Bernhard, Jacques Chambon, Claude Le Pape, Laurent Perron, and Jean-Charles Régin. Résolution d'un problème de conception de réseau avec parallel solver. In *Proceeding of JFPLC*, 2002. (In French).

[6] Yves Caseau and François Laburthe. Effective forget-and-extend heuristics for scheduling problems. In *Proceedings of the First International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'99)*, 1999.

[7] Alain Chabrier, Emilie Danna, and Claude Le Pape. Coopération entre génération de colonnes avec tournées sans cycle et recherche locale appliquée au routage de véhicules (in french). In *Huitièmes Journées Nationales sur la résolution de Problèmes NP-Complets (JNPC'2002)*, 2002.

[8] Alain Chabrier, Emilie Danna, Claude Le Pape, and Laurent Perron. Solving a network design problem. *To appear in Annals of Operations Research, Special Issue following CP-AI-OR'2002*, 2003.

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

[10] E. Balas J. Adams and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34(3):391–401, 1988.

[11] Claude Le Pape, Laurent Perron, Jean-Charles Régin, and Paul Shaw. Robust and parallel solving of a network design problem. In Pascal Van Hentenryck, editor, *Proceedings of CP 2002*, pages 633–648, Ithaca, NY, USA, September 2002.

[12] Olivier Lhomme. Amortized random backtracking. In *Proceedings of CP-AI-OR 2002*, pages 21–32, Le Croisic, France, March 2002.

[13] Mireille Palpant, Christian Artigues, and Philippe Michelon. A heuristic for solving the frequency assignment problem. In *XI Latin-Iberian American Congress of Operations Research (CLAIO)*, 2002.

[14] Mireille Palpant, Christian Artigues, and Philippe Michelon. Solving the resource-constrained project scheduling problem by integrating exact resolution and local search. In *8th International Workshop on Project Management and Scheduling PMS 2002*, pages 289–292, 2002.

[15] Laurent Perron. Search procedures and parallelism in constraint programming. In Joxan Jaffar, editor, *Proceedings of CP '99*, pages 346–360. Springer-Verlag, 1999.

[16] Laurent Perron. Practical parallelism in constraint programming. In *Proceedings of CP-AI-OR 2002*, pages 261–276, March 2002.

[17] Laurent Perron. Fast restart policies and large neighborhood search. In *Proceedings of CPAIOR 2003*, 2003.

[18] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In M. Maher and J.-F. Puget, editors, *Proceeding of CP '98*, pages 417–431. Springer-Verlag, 1998.

[19] Solver. *ILOG Solver 5.2 User's Manual and Reference Manual*. ILOG, S.A., 2001.