The Knowledge Engineering Review, Vol. 16:1, 2001, 41–68
Printed in the United Kingdom

Combining local and global search in a constraint programming environment

YVES CASEAU[1], FRANÇOIS LABURTHE[1], CLAUDE LE PAPE[2] and BENOÎT ROTTEMBOURG[1]

[1] *Bouygues SA, DTN, 1, avenue Eugène Freyssinet, F-78061 Saint Quentin-en-Yvelines, France*
E-mail: {ycs,flaburth,brottemb}@challenger.bouygues.fr
[2] *Bouygues Telecom, R&D, 51, avenue de l'Europe, F-78944 Vélizy, France. E-mail: clepapeg@bouyguestelecom.fr*

Abstract

This paper presents several case studies which illustrate how constraint programming can benefit from the combination of global and local search techniques, offering a flexible and efficient platform for the design of combinatorial optimisation applications. For job-shop scheduling, we relate experiments with local search procedures that use global search to intensively explore a given neighbourhood, in the spirit of "shuffle" methods. For preemptive job-shop scheduling, two basic search strategies, Depth-First Search and Limited Discrepancy Search, are compared. For Vehicle Routing we report an Incremental Local Optimisation heuristic, combined with Limited Discrepancy Search. Finally, we show how ad hoc algebras can considerably enhance the design of heuristics based on local and global search within a constraint-programming environment. Experiments on vehicle routing will enlighten how such a language for "search and insert" control can enable automated tuning and discovery of new strategies adapted to the instances typology of the problem at stake.

1 Introduction

Broadly speaking, constraint programming can be defined as a programming method based on three main principles:

1. The problem to be solved is explicitly represented in terms of variables and constraints on these variables. In a constraint-based program, this explicit problem definition is clearly separated from the algorithm used to solve the problem. This separation guarantees that the problem to be solved is precisely defined. In many cases, it also simplifies the revision or extension of a constraint programming application when the corresponding problem changes. For example, the replacement of old machines by new machines in a manufacturing shop can lead to the introduction of new constraints and the removal of old constraints; yet, in some cases, the same problem-solving algorithms will continue to apply, with a different problem definition as input.
2. Given a constraint-based definition of the problem to be solved and a set of decisions, themselves translated into constraints, a purely deductive process referred to as *constraint propagation* is used to propagate the consequences of the constraints. This process is applied each time a new decision is made and is clearly separated from the decision-making algorithm. This allows the software developer to implement the constraint propagation code and the decision-making code independently of one another. The same constraint propagation code can then be used to propagate decisions made by different decision-making algorithms, as well as decisions made by a human user. In an optimisation context, this may lead to the development of several decision-making algorithms dedicated, for example, to distinct combinations of optimisation criteria.

3. The overall constraint propagation process results from the combination of several local and incremental processes, each of which is associated with a particular constraint or a particular constraint class. This *locality* principle (Steele, 1980) is fundamental, since it enables the efficient combination of multiple constraint propagation techniques associated with different classes of constraints. This allows the developer of a constraint programming application to reuse constraint propagation techniques developed for other applications. It also allows multiple developers to "share" libraries of constraints and augment such libraries with whatever new specific constraints are required for a given application.

One could imagine from this rough definition that Constraint Programming (CP) is an alternative to the usual techniques of Operations Research (OR), i.e. specialised algorithms for specific classes of problems, linear programming, local search, etc. However, it appears that (at least) two conditions must be met for a CP algorithm to efficiently solve a given problem:

1. Constraint propagation must allow the elimination of most of the potential elementary decisions that could be made in the course of solving the problem, without requiring too much computational time. Specialised algorithms inspired by previous work in the OR field often provide the best basis for efficient and effective constraint propagation.
2. The search space must be explored in an efficient manner, i.e., promising regions of the search space must be explored early – before less promising regions. This concern is also a major one in traditional OR algorithms. Even though CP brings about specific issues about the exploration of the search space, many ideas can be applied to both CP and non-CP search algorithms, e.g., extensive use of dominance properties and lower bound calculations, "shaving" (Carlier & Pinson, 1994; Martin and Shmoys, 1996), and forms of local search.

This paper presents several case studies that illustrate these points. In particular, we explore the distinction between global search and local search. Global search consists of a systematic exploration of a structure that represents the set of solutions to the problem under consideration, until an optimal solution is found and proven to be optimal. On the contrary, local search consists of an unsystematic exploration of the solution space based on replacing a solution (or a set of solutions) with one of its neighbours. Constraint programming is generally considered as a global search technique. As a matter of fact, the easiest way of exploiting the benefits of constraint propagation is to move in a search tree, making and retracting decisions one by one. Each time a new decision is made, it is converted into a constraint (or a set of constraints) and propagated. Each time a decision is undone, in a chronological backtracking fashion, the state before the making of the decision is restored, and the search can restart from the restored state. This search strategy is referred to as Depth-First Search (DFS).

Yet constraint programming can be used to implement variants of local search. The idea is to explore, at each step, a search tree that represents some neighbourhood of a given solution. For this technique to be efficient, one must, nevertheless, design the neighbourhood in such a way that the constraints that delimit the neighbourhood propagate well.

Two examples of techniques that have given good results in practice are *shuffle* (Applegate & Cook, 1991) (also called *large neighbourhood search* (Shaw, 1998)) and *limited discrepancy search* (Harvey & Ginsberg, 1995).

- The basic idea behind *shuffle* is straightforward. Given a solution, a neighbourhood is constructed by relaxing some, but not all, of the decisions in the solution. For example, in a scheduling problem, one can keep some of the sequencing decisions that correspond to the current solution and relax the others. A new solution is then searched for, in a global manner, in this neighbourhood. The decisions to be relaxed at each step can be chosen randomly or according to some given patterns. In the case of a scheduling problem, one can, for example, randomly select a set of resources and relax all the sequencing decisions between the activities to be executed on these resources.
- Limited Discrepancy Search (LDS) is an alternative to DFS, which relies on the intuition that heuristics make few mistakes through the search tree. Thus, considering the path from the root node of the tree to the first solution found by a DFS algorithm, there should be few "wrong turns" (i.e.

few nodes which were not immediately selected by the heuristic). The basic idea is to restrict the search to paths that do not diverge more than w times from the choices recommended by the heuristic. When $w = 0$, only the leftmost branch of the search tree is explored. When $w = 1$, the number of paths explored is linear in the depth of the search tree, since only one alternative turn is allowed for each path.[1] Each time this limited search fails, w is incremented and the process is iterated, until either a solution is found or it is proven that there is no solution. It is easy to prove that when w gets large enough, LDS is complete. At each iteration, the branches where the discrepancies occur close to the root of the tree are explored first (which makes sense when the heuristics are more likely to make mistakes early in the search; see Harvey and Ginsberg (1995) for details). Walsh (1997) proposes a variant called "Depth-bounded Discrepancy Search" (DDS) in which any number of discrepancies is allowed, provided that all the discrepancies occur up to a given depth. This variant is recommended when the heuristic is very unlikely to make mistakes in the middle and at the bottom of the search tree.

The remainder of the paper is organised as follows. Section 2 will be devoted to the Job-Shop Scheduling Problem, where local search using intensive exploration of neighbourhoods will be compared to standard local search techniques; section 3 will deal with the Preemptive Job-Shop Scheduling Problem for which basic strategies will be combined with different heuristics for task selection, and section 4 will report on strategies for the Vehicle Routing Problem (VRP) based on incremental local optimisation procedures. Finally, section 5 will consist of a first step towards the automated discovery of meta-heuristics based upon a specific algebra and applied to the Vehicle Routing Problem.

2 Job-shop scheduling

This section relates experiments on the Job-Shop Scheduling Problem (JSSP), with combinations of local and global search procedures. The master algorithm is an iterative improvement loop. Global search is used to find the best solution within a neighbourhood. Such a combination has two advantages: on the one hand, it supports the exploration of large neighbourhoods; on the other hand, the use of implicit enumeration techniques (constraint propagation, branch and bound, incomplete search paradigms) limits the exploration time.

We review several such procedures, in the spirit of *shuffle* (Applegate & Cook, 1991) and compare them with standard local search approaches, based on smaller neighbourhoods.

2.1 Problem definition

The JSSP can be defined as follows. A set of activities and a set of resources are given as data, and one has to plan a processing date for all activities. This planning must respect precedence constraints and resource constraints.

Resources may perform only one activity at a time (unary resources), and tasks activities have no flexibility. The precedence graph has a specific structure: tasks activities are organised into jobs, where each job is a sequence of tasks activities (exactly one per resource) fully ordered by precedence constraints. The objective is to minimise the global completion time of the schedule (the makespan) defined as the latest end time over all activities. Instances are usually described by the number of resources m and the number of jobs n (for an overall figure of $n \times m$ activities).

The JSSP has been widely studied by the Operations Research community. It is known to be NP-Complete [GJ79], and a wealth of methods and techniques have been proposed to solve it. State-of-the-art Branch-and-Bound methods now solve to optimality within minutes 10×10 instances;

[1] Note that when the search tree is not binary, it can be considered that the ith best choice according to the heuristics corresponds either to 1 or to $(i - 1)$ discrepancies.

however, for hard instances with a few hundred activities (15×10 to 20×20), exact methods are too slow, and there is a definite need for approximate methods producing good solutions.

Similar scheduling problems with unary resources also have a wide range of practical applications in production planning or process organisation. In such areas, the optimisation problem often arises in a reactive setting: some event occurs while the schedule is being processed, which changes the problem and calls for online rescheduling decisions. Such reactive optimisation applications advocate the development of local search methods.

Finally, it should be kept in mind that practical applications all have their specific constraints and seldom fall into the pure JSSP model. Thus, it is worthwhile to develop methods that are generic enough to solve a somewhat broader class of scheduling problems and which do not rely exclusively on the particular job precedence structure.

2.2 Global search

Branch-and-bound procedures are the most widely used exact methods for solving the JSSP. In addition to lower bound cuts, most procedures perform some constraint propagation at each node of the search tree. For instance, Carlier and Pinson (1989) gives an algorithm for updating the heads and tails of activities, which amounts to their earliest start time and latest end time. The propagation of resource constraints tightens the time windows of activities and infers the relative ordering for some pairs of activities sharing a common resource. Such "edge-finding" propagation algorithms have been proposed (Carlier & Pinson, 1989; Nuijten, 1994; Caseau & Laburthe, 1994; Baptiste & Le Pape, 1996).

Several branching schemes may be used for exploring the solution space. In order for such decisions to have significant impact through constraint propagation, algorithms often branch on ordering decisions. Simple binary alternatives select a pair of tasks activities on the same resource and consider both possible relative orders. Another possibility consists of selecting a set of activities on the same resource and deciding which one is scheduled first (resp. last). Finally, a third possibility consists of ranking activities on the same resource.

Throughout the experiments presented in this section, we use the branching scheme presented in Caseau and Laburthe (1994). The procedure selects the tightest set of activities among all resources, considers a pair of activities that could be scheduled first or last among this set and branches on both possible sequencing orders for such a pair.

Last, a time-based approach can also be used for breadth-first search. This is the *shaving* mechanism proposed by Carlier and Pinson (1994) and Martin and Shmoya (1996). The idea is to detect those dates that are infeasible (through propagation) and remove them from the activities' time windows. This technique has a high complexity, but it has proven useful for some hard instances.

2.3 Local search

A solution to a scheduling problem with unary resources can be fully characterised by the ordered sequence of activities on each resource: once the relative order of activities sharing a common resource is settled, finding the makespan as well as feasible processing dates amounts to solving a longest path problem. This can be done in $O(mn)$ (Adams *et al.*, 1988). A longest path from the start of the schedule to the end is called a *critical path*. All activities on such paths are critical in the sense that if the processing time of any of these activities was increased by any amount $\Delta > 0$, then the makespan of this solution would also increase by Δ.

Most neighbourhoods rely on this description of a solution as a set of ranked sequences and use the notion of critical tasks activities. Several neighbourhoods have been proposed:

- Van Laarhoven, Aarts and Lenstra (1992) consider a simple neighbourhood (N1) based on the interchange of two critical activities that are sequenced one after the other on the same resource;

- Dell'Amico and Trubian (1993) consider a neighbourhood (N2) based on the permutation of three consecutive tasks activities on the same resource such that at least two of them are critical;
- Dell'Amico and Trubian (1993) also consider a neighbourhood (N3) which flips an activity A_i after an activity A_k or before an activity A_l, where $A_l, A_2, \ldots, A_k$ is a "block" of consecutive critical activities processed on the same resource;
- Nowicki and Smutnicki (1996) consider a subset of the neighbourhood (N1) where neither of the two activities to be interchanged may be internal to a block of consecutive critical activities.

Such moves may easily get trapped in local optima that may be far away from the optimum. Indeed, several such moves may be required in order to see an improvement in the solution. Thus, authors have often settled for control schemes more sophisticated than hill-climbing which accept moves that degrade the objective functions, such as simulated annealing (Van Laarhoven *et al*., 1992) or tabu search (Nowicki & Smutnicki, 1996). Another approach, proposed in Balas and Vazacopoulos (1998), foresees sequences of such interchange moves. Thus, the blind evaluation of a move (its direct impact on the objective function) is replaced by an analysis that looks forward several moves ahead. Therefore, the process of move selection is performed in a more informed manner, which suggests that the optimisation loop is less likely to end up trapped in a local optimum. Such neighbourhoods can be related to travel optimisation problems for which simple edge exchange procedures are best used in a tabu search framework (Rochat & Taillard, 1995), but can be further improved by considering sequences of edge-exchanges. This is the case for the heuristic from Lin and Kernighan (1973) for the Traveling Salesman Problem (TSP), which searches for a move exchanging an arbitrary number of edges. The "relocation tree" heuristic for the VRP (Caseau & Laburthe, 1999), also searches for an alternating sequence of node removals and node insertions.

2.4 Combining local and global search

Another possibility for improving the efficiency of local search methods is to consider larger neighbourhoods. Instead of fully specifying the repair paths from a solution to its neighbours, we may only specify the "least common factor" between all solutions in the neighbourhood and use a global search algorithm to complete this common fragment into neighbour solutions.

Several such neighbourhoods, based on partial schedules, have been proposed for the JSSP:

- The shifting bottleneck procedure from Adams *et al*. (1988) considers only a subset S_k of the m machines, keeps the processing orders on all machines from S_k, and, one by one, reschedules the remaining resources by solving each time to optimality a one machine sequencing problem.
- The shuffling procedure from Applegate and Cook (1991) also keeps the processing order on a small subset S_k of the machines and solves to optimality the scheduling problem on the remainder of the machines. Different subsets S_k are tried until a makespan-improving solution is found.
- The procedure from Baptiste *et al*. (1995) keeps a random subset of the ordering decisions in the schedule (the relative order of two activities on the same resource) and completes the solution by a Branch-and-Bound algorithm.
- The *Forget-and-Extend* procedure, designed in Caseau and Labourthe (1995), considers several types of resources fragments, and constrains the search for neighbours to find improving solutions.

Note that there is an important difference between small neighbourhoods presented in the previous section and such large neighbourhoods: small neighbourhoods can easily be explored exhaustively; for large neighbourhoods, searching for the best neighbour may become a difficult optimisation problem. Neighbours of a solution S with makespan M are searched as follows: a fragment F of S is recorded and global search is then used to find a solution extending F. This global search algorithm can reduce the search space of the neighbourhood with optimisation cuts, i.e., with an improvement constraint stating that the makespan must be better than M. Thus, the method uses global search as a way to explore a neighbourhood. Such hybrid procedures combining local and global search have also been

investigated on a few other combinatorial problems (see Pesant and Gendreau (1996) for timetabling, Mautor and Michelon (1997) for quadratic assignment, and Shaw (1998) and Rousseau *et al*. (1999) for vehicle routing).

We now describe the Forget-and-Extend procedure in more details. The fragment F is a set of f pairs of activities (A, A') requiring the same machine. For each such pair, we enforce that A and A' should have the same relative order in all solutions of the neighbourhood as in S. Such an order constraint can be easily enforced since the branching scheme is based on such binary ordering decisions. Therefore, the size of the neighbourhood is about $1/(2^f)$ of the size of the full solution space. The size of the fragment f must be chosen according to antagonistic criteria: large values of f yield small neighbourhoods, which seldom contain improving solutions and, therefore, the optimisation loop may converge towards local optima far away from the global optimum; small values of f yield large neighbourhoods which are time consuming to explore. Thus a trade-off has to be found. We settled for values of f around 10 because we would rather perform a few iterations to explore search spaces about 1000 times smaller $1/(2^{10})$ than the original solution space, and, therefore, have a faster procedure for finding one solution.

In practice, the search algorithm takes much more time to prove the optimality of a problem than to find solutions. Since the improvement constraint (optimisation cut) may lead to empty neighbourhoods, the search space is not explored exhaustively: we stop at the first solution found or when no solutions have been found after a small number nb of backtracks. On one hand, in those situations when there are improving solutions in the neighbourhood, one can hope that they will be found quickly by the search process. On the other hand, if we have already backtracked some times without finding any solutions, this may indicate that the neighbourhood could be empty and that we would be better off keeping another fragment from the previous solution.

As for all local search methods that explore large neighbourhoods, the key factor to their success is the ability to find a good neighbour quickly. The efficiency of global search methods enables us to keep only small solution fragments, which leads to a larger neighbourhood leaving more room for improvement. Our method has three original features: the fragments that are kept are small, different types of fragments are considered, and the search of the neighbourhood is incomplete.

Four kinds of fragments are considered:

1. The first kind of fragment is taken from the shuffle procedure of Applegate and Cook (1991): the processing order is kept for all activities processed on k (among the m) machines. Small values are taken for $k : k = 1$ for 10×10 instances to $k = 2$ for 20×20 instances.
2. The second kind of fragment is adapted from the first one: the processing order is kept for all activities that are not critical and that are processed on a subset of k machines. Such a neighbourhood allows the system to rearrange all activities on the critical paths. Small values are taken for k ($k \leq 3$).
3. The third kind of fragment consists of a "temporal slice" of the schedule. Two dates a and b are selected between 0 and M. The processing order is kept for all pairs of activities (A, A') which are processed on the same machine and scheduled to start after a and to end before b. The width of the time window b-a is selected in such a way that the fragment F contains the target number of pairs (f).
4. The last kind of fragment also amounts to a temporal slice of the schedule, but it is based on the ranked model and is somewhat specific to the job-shop structure. On all machines, the number of processed activities is the same (n). For all i in $(1 \ldots n - 1)$, we consider a fragment made of the ordering decision between the ith and $i + 1$th activities on all machines.

Once the fragment of the solution has been selected, an upper bound $M - \delta$ is set on the makespan. This optimisation constraint removes all the solutions from the neighbourhood that do not yield an improvement of at least δ units over the makespan M of the current solution. The search starts with high values of the optimisation grain δ (about 1% of M) and is progressively decreased to 1 time unit.

Before starting the initial propagation, the upper bound constraint and the ordering constraints from the fragment are propagated. After this initial propagation, the scheduler is left with a partial schedule and searches for a valid complete schedule. The bound on the size of the search tree starts with very small values ($nb = 10$ to 30) and progressively increases them (up to $nb = 100$ to 1000). The parameters nb and δ are initialised with optimistic values (small nb, large δ); they are updated (nb is increased, δ decreased) after an iteration where no move can be found using any of the neighbourhoods.

2.5 Experimental results

This optimisation algorithm has been tested on job-shop benchmarks from the literature. We report two sets of experiments with our hybrid method combining local and global search, one for comparing it to pure global search and the other for comparing it to other local search techniques.

The first experiment is made on the ten 10×10 instances selected by Applegate and Cook (1991) for their computational study. For each problem, we compare the Forget-and-Extend procedure to a complete global search procedure. This global search approach is used for completing fragments into solutions in the local search approach (same branching scheme). Instead of performing one single branch and bound exploration, a new search tree is started each time a solution is found. The optimisation process is performed in a dichotomic manner: each time a solution with makespan M is found, we search for a solution with makespan less than $(LB + M)/2$, where LB is the current lower bound.

Moreover, to make a fair comparison between both methods, we give the global search algorithm the optimal value as lower bound LB. Therefore, the global search algorithm, like the hybrid one, never solves infeasible problems (such as proving the optimality of a solution), which are known to take more computational effort than feasible ones. Thus, the comparison between both approaches should be fair and underline the very use of fragments as guidelines for improving a solution. For both methods, we report in Table 1 the value of the best solution that was found, the number of iterations, the total number of backtracks in the search trees, and the running time (on a Pentium II 200 MHz).

Several comments can be made on these experiments. First, the Forget-and-Extend method has a good performance level, since it is able to find the optimal solution in most cases and, otherwise, comes close to it. The comparison shows that, for easy instances such as ABZ6 or LA19, which can be solved easily by global search (with few backtracks), no gain should be expected since the search trees are already very small. Moreover, it is interesting to notice that there is not much loss: going from solution to solution, through common fragments did not prevent finding the optimum. This suggests that the landscape induced by the Forget-and-Extend neighbourhoods is relatively smooth (not too many local optima far away from the global optimum). The situation is different for harder problems,

Table 1 Comparing the Forget-and-Extend procedure with global search (number of iterations, backtracks, and running time on a Pentium II 200 MHz)

n	*m*	Instance	Opt.	UB	Dichotomic global search	Forget-and-extend hybrid search
10	10	MT10	930	1013	930: 5 itns, 41349b, 709 s.	930: 11 itns, 402b, 18,8 s.
10	10	ABZ5	1234	1330	1234: 5 itns, 2445b, 43,7 s.	1236: 14 itns, 20153b, 159 s.
10	10	ABZ6	943	1052	943: 5 itns, 90b, 3,8 s.	943: 13 itns, 4b, 12,7 s.
10	10	LA19	842	966	842: 4 itns, 6b, 2,1 s.	842: 13 itns, 115b, 12,8 s.
10	10	LA20	902	970	902: 4 itns, 3583b, 51,8 s.	902: 8 itns, 7b, 16,8 s.
10	10	ORB1	1059	1168	1059: 5 itns, 15943b, 278 s.	1059: 23 itns, 683b, 35,8 s.
10	10	ORB2	888	952	888: 6 itns, 757b, 15,5 s.	889: 9 itns, 952b, 51,7 s.
10	10	ORB3	1005	1201	1005: 6 itns, 499572b, 2411 s.	1005: 19 itns, 1071b, 34,8 s.
10	10	ORB4	1005	1158	1005: 5 itns, 2219b, 37 s.	1013: 23 itns, 5008b, 85 s.
10	10	ORB5	887	957	887: 5 itns, 1308b, 25,7 s.	887: 12 itns, 533b, 71 s.

Table 2 The Forget-and-Extend heuristic on 13 hard job-shop instances. For each instance, we report the last solution returned by the local optimisation procedure with various CPU time limits (Pentium II 200 MHz)

n	*m*	Instance	opt.	UB	1 minute	5 minute	10 minutes	1 hour	Nowicki and Smutnicki (1996)
10	10	MT10	930	1013	930	930	930	930	930
10	5	LA02	655	699	655	655	655	655	655
10	10	LA19	842	966	842	842	842	842	842
15	10	LA21	1046	1211	1070	1053	1046	1046	1047
15	10	LA24	935	1017	952	938	938	938	939
20	10	LA25	977	1168	988	979	979	979	977
20	10	LA27	1235	1466	1381	1286	1266	1237	1236
20	10	LA29	1152	1428	1363	1230	1182	1163	1160
15	15	LA36	1268	1443	1341	1274	1268	1268	1268
15	15	LA37	1397	1614	1502	1405	1397	1397	1407
15	15	LA38	1196	1380	1258	1206	1196	1196	1196
15	15	LA39	1233	1394	1316	1235	1233	1233	1233
15	15	LA40	1222	1502	1373	1230	1226	1222	1229
All 13 instances				opt + 15%	opt + 5.6%	opt + 1.1%	opt + 0.45%	opt + 0.126%	opt + 0.199%

such as MT10, ORB1, or ORB3. Here, the Forget-and-Extend procedure yields a definite improvement: trying to keep a fragment of the current solution as a guide for finding another solution with a better makespan is a good idea; the fragment acts as a shortcut in the solution space, and the procedure is faster than global search.

A second experiment was performed on the 13 hard instances that were selected by Vaessens *et al*. (1996) as a test-bed for comparing local search methods on the Job-Shop Problem. Among all techniques that were compared by Vaessens *et al*. (1996), the Tabu Search algorithm of Nowicki and Smutnicki (1996) obtained the best results with an average distance to the optimum[2] of 0.199%; shuffle-related procedures inspired by Applegate and Cook (1991) achieved distances between 1.8% and 5%, the simulated annealing approach of Vaessens *et al*. (1992) came in at 1.9%. Table 2 reports the results of the Forget-and-Extend procedure, with various running time limits from a minute to an hour (still on a Pentium II 200 MHz). For the sake of comparison, we include the results from the reference procedure of Nowicki and Smutnicki (1996).

The Forget-and-Extend algorithm significantly improves the efficiency of the initial shuffling method of Applegate and Cook (1991) and improves on the Tabu Search algorithm (with a mean relative error of 0.126%) when given a CPU time limit of one hour. This indicates that such combinations of global and local search can obtain state-of-the-art results, significantly improving the results from the early shuffle procedure. The key factors to this improved performance are the use of variable neighbourhoods (considering a variety of fragments) and the use of limited search to explore the neighbourhoods.

2.6 Summary

This section has presented the Forget-and-Extend heuristic, a method combining local and global search on the JSSP. The procedure has shown to be competitive compared to both the pure global search procedure and simple local search methods based on simpler neighbourhoods.

[2] Mean Relative Error (MRE) computed as follows: for each instance, the relative error is computed as the difference between the obtained makespan and the optimal value, divided by the optimal value. The MRE is the average relative error over the thirteen instances.

From a local search point of view, one can draw an analogy with several features common in Tabu Search heuristics:

1. **Intensification**: the search for the completion of a fragment amounts to a systematic exploration of a neighbourhood;
2. **Diversification**: many fragments are considered, which amounts to considering many neighbourhoods, or a variable neighbourhood;
3. **Aspiration criterion**: we only look for solutions which strictly improve on the current objective value.

Among these features, the diversification item is crucial to the efficiency of the method: the more fragments that may be considered, the better the results of the algorithm.

From a global search point of view, the procedure could probably be improved by taking advantage of some recent advances in global search for JSSP. In our point of view, two most promising directions consist of:

1. Using smarter limited search schemes in the exploration of the neighbourhood, instead of a simple backtrack-bounded depth-first search. LDS would probably be better suited for exploring such neighbourhoods (as is the case with large neighbourhood search in vehicle routing (Shaw, 1998)).
2. Using enhanced propagation mechanisms, such as shaving, for strengthening the solution fragment before trying to complete it into a solution. Preliminary experiments suggest that shaving can indeed improve the efficiency of Forget-and-Extend procedures on hard problems.

3 Preemptive job-shop scheduling

This section presents a combination of global and local search for the Preemptive Job-Shop Scheduling Problem (PJSSP). More precisely, the components which we use include:

1. Various variants of LDS.
2. The exploitation of the best found solution as a guideline for the global search.
3. The application of local optimisation operators interleaved between global searches.

3.1 Problem definition

PJSSP is a variant of the JSSP in which activities can be interrupted. It can be defined as follows. Given are a set of jobs and a set of machines. Each job consists of a set of activities to be processed in a given order. Each activity is given an integer processing time and a machine on which it has to be processed. A machine can process at most one activity at a time. Activities may be interrupted at any time, an unlimited number of times. The problem is to find a schedule, i.e. a set of integer execution times for each activity, that minimises the makespan, i.e. the time at which all activities are finished. (We note that, since we require integer durations and integer execution times, the total number of interruptions of a given activity is bounded by its duration minus 1.)

In more formal constraint programming terms, a start time variable $start(A)$, an end time variable $end(A)$, an integer duration $duration(A)$, and a set variable $set(A)$ are associated with each activity A. The set variable $set(A)$ represents the set of times at which A executes; $start(A)$ represents the time at which A starts, $end(A)$, the time at which A end; and $duration(A)$ the is number of time units required for A. An additional variable, *makespan*, represents the makespan of the schedule. The following constraints apply:

- For every activity A, $duration(A) = |set(A)|$.
- For every activity A, $start(A) = \min_{t \in set(A)}(t)$ and $end(A) = \max_{t \in set(A)}(t+1)$. This implies $end(A) \geq start(A) + duration(A)$. We note that, in a given solution, we have $end(A) = start(A) + duration(A)$ if and only if A is not interrupted.
- For every job $J = (A_1, A_2, \ldots, A_m)$ and every i in $\{1, 2, \ldots, m-1\}$, $end(A_i) \leq start(A_{i+1})$.

- For every machine M, if $acts(M)$ denotes the set of activities to be processed on M, then for every pair of activities (A, B) in $acts(M)$, $set(A)$ and $set(B)$ are disjoint.
- For every activity A, $0 \leq start(A)$.
- For every activity A, $end(A) \leq makespan$.

The goal is to minimise the value of the *makespan* variable. In the following, this minimisation objective is achieved by solving the decision variant of the problem (Garey & Johnson, 1979) with different bounds imposed on the makespan. At each iteration, an additional constraint $makespan \leq v$ (where v is a given integer) is imposed, and the problem consists of determining a value for each variable such that all the constraints, including $makespan \leq v$, are satisfied. If such a solution is found, its makespan can be used as a new upper bound for the optimal makespan. On the contrary, if it is proven that no such solution exists, $v + 1$ can be used as a new lower bound. The "decision variant" of the PJSSP (the problem of determining whether there exists a solution with $makespan \leq v$), is NP-Complete in the strong sense (Garey & Johnson, 1979). Contrarily to the non-preemptive JSSP, the PJSSP has not drawn much attention from the Operations Research community.

3.2 Basic tree search algorithm

The search space for the PJSSP is larger than the search space of the non-preemptive JSSP. Indeed, each $set(A)$ variable *a priori* accepts up to $(v*(v-1)*\ldots*(v-duration(A)+1))/(1*2*\ldots*duration(A))$ values. However, the dominance criterion introduced below allows the design of branching schemes which, in a sense, order the activities that require the same machine and, thus, explore a reduced search space. The basic idea is that it does not make sense to let an activity A interrupt an activity B by which it was previously interrupted. In addition, A shall not interrupt B if the successor of A (in its job) starts after the successor of B. The following definitions and theorem, proven in Le Pape and Baptiste (1999), provide a formal characterisation of the dominance property.

Definition 1 *For any schedule S and any activity A, we define the "due date of A in S" $d_S(A)$ as*:

- *the makespan of S if A is the last activity of its job*;
- *the start time of the successor of A (in its job) otherwise*.

Definition 2 *For any schedule S, an activity A_k has priority over an activity A_l in S ($A_k <_S A_l$) if and only if either $d_S(A_k) < d_S(A_l)$ or $d_S(A_k) = d_S(A_l)$ and $k \leq l$. Note that $<_S$ is a total order.*

Theorem 1 *For any schedule S, there exists a schedule $J(S)$ such that*:

1. ***J(S) meets the due dates***: $\forall A$, *the end time of A in $J(S)$ is at most $d_S(A)$.*
2. ***J(S) is "active"***: $\forall M, \forall t$, *if some activity $A \in acts(M)$ is available at time t, M is not idle at time t (where "available" means that the predecessor of A is finished and A is not finished).*
3. ***J(S) follows the $<_S$ priority order***: $\forall M, \forall t, \forall A_k \in acts(M), \forall A_l \in acts(M), A_l \neq A_k$, *if A_k executes at time t, either A_l is not available at time t or $A_k <_S A_l$.*

We call $J(S)$ the "Jackson derivation" of S. Since the makespan of $J(S)$ does not exceed the makespan of S, at least one optimal schedule is the Jackson derivation of another schedule. Thus, in the search for an optimal schedule, we can impose the characteristics of a Jackson derivation onto the schedule under construction. This led us to the following branching scheme (which heavily exploits the dominance criterion):

1. Let t be the earliest date such that there is an activity A available (and not scheduled yet!) at t.
2. Compute K, the set of activities available at t on the same machine than A.
3. Compute NDK, the set of activities which are not "dominated" in K.
4. Select an activity A_k in NDK. Schedule A_k to execute at t. Propagate the decision and its consequences. Keep the other activities of NDK as alternatives to be tried upon backtracking.
5. Iterate until all the activities are scheduled or until all alternatives have been tried.

Needless to say, the power of this branching scheme highly depends upon the rules that are used to eliminate "dominated" activities in step 3 and propagate "consequences" of the choice of A_k in step 4. Full details on our search strategy are available in Le Pape and Baptiste (1999).

3.3 Constraint propagation

We distinguish three categories of constraints in the PJSSP. The first category includes all the constraints that relate the variables of a given activity A: $duration(A) = |set(A)|$, $start(A) = \min_{t \in set(A)}(t)$, $end(A) = \max_{t \in set(A)}(t + 1)$. These constraints are easily propagated by maintaining a lower bound and an upper bound for the set variable $set(A)$. The second category is composed of temporal constraints, $var_i + d_{ij} \leq var_j$, where var_i and var_j are either start and end times of activities, the *makespan* variable, or even the constant 0, and d_{ij} is an integer (the minimal delay between the two time points var_i and var_j). These constraints, to which we add the redundant inequalities $start(A) + duration(A) \leq end(A)$, are easily propagated through an incremental version of Ford's algorithm (Gondron & Minoux, 1984; Le Pape, 1988; Cesta & Oddi, 1996). This guarantees that the earliest and latest start and end times of activities are always consistent with respect to the temporal constraints. In other words, if no contradiction is detected, then the earliest start and end times of activities satisfy all the temporal constraints, and the latest start and end times of activities satisfy all the temporal constraints.

The third category of constraints includes the resource constraints, stating that each machine M can execute at most one activity at a time. These constraints are the most complex to propagate. Two constraint propagation techniques, inspired by previous work on non-preemptive scheduling, have been developed.

The first technique relies on resource timetables (Le Pape, 1994) and applies to both unary resources, which can execute only one activity at a time, and "cumulative" resources of capacity $c > 1$. A timetable is maintained for each resource, in order to keep track of the required and available capacity at any time t. Propagation occurs both from activities to resource timetables, and from resource timetables to activities.

- *From activities to resources*: when an activity is known to execute at time t, this activity requires its resources at time t. The timetable of each required resource can, consequently, be updated.
- *From resources to activities*: the resource timetable is used to incrementally update time-bounds of activities. The earliest end time of each activity is updated to ensure that between its earliest start time and its earliest end time, there are enough "free" time intervals on the resource to let the activity execute. Needless to say, if the resource timetable proves that the activity cannot start before time t, its earliest start time is updated, and the timetable mechanism is iterated. A similar technique is used to update the latest start time and the latest end time of the activity.

The second resource constraint propagation technique relies on edge-finding. It consists of determining whether an activity A can, cannot, or must, start or end before (or after) a set of activities Ω. A mixed edge-finding algorithm, allowing both interruptible and non-interruptible activities, is proposed in Le Pape and Baptiste (1996). Given a machine M and the current time-bounds of activities in $acts(M)$, it computes, for each activity A in $acts(M)$:

- when A is not interruptible: the earliest time at which A could start and the latest time at which A could end, if all the other activities in $acts(M)$ were interruptible;
- when A is interruptible: the earliest time at which A could end and the latest time at which A could start, if all the other activities in $acts(M)$ were interruptible.

In practice, the edge-finding algorithm enables the deduction of better bounds, but it is expensive in CPU time and, in particular, much less incremental than the timetable-based algorithm. Indeed, for a given machine M, each run of the edge-finding algorithm involves all the activities in $acts(M)$. Table 3 provides the average time per node we have observed for a few PJSSP instances of different sizes,

Table 3 CPU time per node with timetables vs. edge-finding

n	m	Instances	$c(tt)$	$c(tt)/(n*m)$	$c(ef)$	$c(ef)/(n*m)$
10	5	LA02, LA03	0.62	0.0124	1.76	0.0352
15	5	LA07, LA08	0.99	0.0132	2.27	0.0303
20	5	LA12, LA15	1.49	0.0149	3.13	0.0313
10	10	FT10, ABZ5, ORB3	0.81	0.0081	3.36	0.0336
15	10	LA24, LA25	0.94	0.0063	4.71	0.0314
20	10	LA27, LA29	1.40	0.0070	6.12	0.0306
15	15	LA36, LA38, LA40	1.25	0.0056	6.85	0.0304
30	10	LA31, LA35	2.16	0.0072	8.70	0.0290

using the ten problem-solving strategies described in the next sections. Only the hardest instances have been considered because, on easier instances, most of the time is spent proving that the optimal solution has been found. In this table, n denotes the number of jobs, m the number of machines, $c(tt)$ the average cost (CPU time in milliseconds on a PC Dell at 200 MHz) per node when the timetable mechanism is used, and $c(ef)$ the average cost per node when the edge-finding algorithm is used.

Several important things appear. First, the table confirms that the edge-finding algorithm is costly: on the largest instances, the time spent per node is multiplied by a factor between 4 and 5.5 when the edge-finding algorithm is used. Second, and more surprisingly, $c(ef)$ increases linearly (and even a little less than linearly) with the size of the problem. The $c(ef)/(n*m)$ ratio varies between 0.035, for the smallest problems, and 0.029. The $c(tt)/(n*m)$ ratio seems to depend upon the number of machines: it varies between 0.012 and 0.015 for the instances with five machines, and between 0.0056 and 0.0081 for the larger instances.

Needless to say, many factors impact the measured time per node. Firstly, the edge-finding algorithm tends to find dead ends sooner than the timetable propagation algorithm and, thus, is more often in a situation where many activities are unscheduled. Secondly, the jobs tend to compete less for the machines when the number of machines is high (because the propagation of the temporal constraints results in spreading the competing activities in time). This impacts both the time needed for constraint propagation to reach quiescence (at a given node) and the shape of the search tree. When constraint propagation is restricted to the use of timetables, the making of one decision tends to trigger resource constraint propagation on a limited number of machines, which may explain the behaviour of the $c(tt)/(n*m)$ ratio. A more systematic computational study would be necessary to further explain this behaviour.

3.4 Search

Two basic search strategies, DFS and LDS, have been considered. We also considered two different heuristics for selecting the activity A_k to be scheduled first in *NDK*.

- The EDD (Earliest Due Date rule) heuristics selects the activity with the smallest latest end time. Using this heuristic seems reasonable since it corresponds to the rule which optimally solves the preemptive one-machine problem (see Carlier and Pinson (1990)).
- The CBS (Current Best Schedule) heuristics relies on the current best schedule S computed so far. It selects the activity A_k with minimal $d_S(A_k)$. This should help to find a better schedule when there exists one that is close to the previous one.

Each time a new improving solution S is found, the Jackson derivation operator J and its symmetric counterpart K are used to improve the complete schedule S, prior to restarting the search with a new upper bound on the makespan (equal to the best makespan found so far minus 1). Several strategies

can be considered, apply only *J*, apply only *K*, and apply a sequence of *J*s and *K*s. The following experimental results are based on using the following scheme:

1. Compute $J(S)$ and $K(S)$;
2. Replace S with the best schedule among $J(S)$ and $K(S)$, if this schedule is strictly better than S (in our implementation, $J(S)$ is chosen if $J(S)$ and $K(S)$ have the same makespan);
3. If S has been replaced by either $J(S)$ or $K(S)$, iterate.

3.5 Experimental results

Table 4 provides the results obtained on the preemptive variant of the ten 10×10 instances used in Applegate and Cook (1991). Each line of the table corresponds to a given variant of the search algorithm and provides the mean relative error (MRE, in percentage) obtained after 1, 2, 3, 4, 5 and 10 minutes of CPU time. The optimal values have been obtained by running an exact algorithm, described in Le Pape and Baptiste (1998), with an average CPU time of 3.4 hours and a maximum of 27 hours (for the ORB3 instance) on a PC Dell at 200 MHz.

Table 5 provides the results obtained on the thirteen instances used in Vaessens *et al*. (1996). As these instances differ in size, we allocated to each instance an amount of time proportional to the square of the number of activities in the instance. We used the square of the number of activities

Table 4 Results on the ten instances used in Applegate and Cook (1991)

Propagation algorithm	Search strategy	1	2	3	4	5	10
Timetable	DFS-EDD	8.95	8.95	8.95	8.95	8.95	8.33
	DFS-CBS	8.32	8.16	7.74	7.73	7.73	7.34
	LDS-EDD	7.68	6.75	6.75	6.75	6.75	5.98
	LDS-CBS	6.14	5.67	5.59	5.14	5.07	4.20
Edge-finding	DFS-EDD	4.29	3.67	3.17	2.55	2.42	1.62
	DFS-CBS	1.69	1.32	0.86	0.80	0.79	0.65
	LDS-EDD	2.17	2.03	1.86	1.71	1.38	1.24
	LDS-CBS	0.64	0.64	0.55	0.36	0.32	0.23

Table 5 Results on the thirteen instances used in Vaessens *et al*. (1996)

Propagation algorithm	Search strategy	1	2	3	4	5	10
Timetable	DFS-EDD	9.22	9.22	9.22	9.22	9.22	9.04
	DFS-CBS	8.82	8.70	8.60	8.59	8.59	7.84
	LDS-EDD	7.98	7.97	7.95	7.89	7.87	7.40
	LDS-CBS	5.93	5.82	5.78	5.66	5.66	4.60
Edge-finding	DFS-EDD	4.02	3.74	3.32	3.26	3.22	3.03
	DFS-CBS	2.26	2.12	1.94	1.77	1.77	1.72
	LDS-EDD	2.43	2.20	2.03	1.82	1.73	1.61
	LDS-CBS	1.75	1.28	1.03	0.92	0.92	0.79

because the time spent per node is approximately linear in the number of activities, and the number of decisions necessary to construct a schedule (i.e. the depth of the search tree) is also proportional to the number of activities (hence the time necessary to reach the first solution tends to increase as the square of the number of activities). To our knowledge, five of the thirteen instances are, in the preemptive case, open and we have been unable to solve them with our exact algorithm. For these instances, we applied a variant of edge-finding and "shaving" (Carlier & Pinson, 1994; Martin & Schmoys, 1996; Péridy, 1996) to obtain a lower bound and, thus, to estimate the relative error.

Tables 6 and 7 provide results obtained with variants of LDS (based on the CBS heuristic and the edge-finding propagation algorithm). LDS^N and DDS^N refer to the application of the LDS and DDS principles on the N-ary tree, where each son of a node corresponds to scheduling one candidate activity on the corresponding machine. LDS and DDS (without the N) correspond to the application of LDS and DDS to the binary tree in which each decision consists of either scheduling or postponing the best candidate. Figure 1 illustrates the difference between the five algorithms, DFS, LDS, LDS^N, DDS and DDS^N on a ternary tree of depth three. ITE shows at which iteration each leaf of the tree is attained and ORD shows in which order the leaves are visited. It shall be noted that, to explore a complete tree, LDS and DDS have a high overhead over DFS. Hence, LDS and DDS must quickly repair the bad decisions to beat DFS. LDS^N and DDS^N have a much smaller overhead on a complete tree. Hence, their performance is less dependent on the quick repair of the worst decisions. However, in LDS^N and DDS^N, the 2nd, 3rd, . . . and *N*th sons of a node are considered equal, which can lead to the early exploration of unpromising branches.

Globally, it appears that the edge-finding technique is the most crucial of the techniques we used to provide good solutions to the PJSSP. When edge-finding is not used, LDS appears as the second most important technique. LDS on the binary tree appears to be the best overall alternative. LDS^N also performs very well, in particular on the thirteen instances of Vaessens *et al*. (1996). By contrast, DDS and DDS^N do not perform that well. In particular, DDS and DDS^N do not perform better than DFS in Table 6 and for the first values of the time factor in Table 7. When edge-finding is used, the contribution of LDS is less important than the reliance on the Current Best Schedule (CBS). This confirms the observation in Beck *et al*. (1997) that LDS appears to help the weaker algorithms to a greater extent than the stronger algorithms.

Table 6 LDS variants on the ten instances used in Applegate and Cook (1991)

Search strategy	1	2	3	4	5	10
DFS-CBS	1.69	1.32	0.86	0.80	0.79	0.65
LDS-CBS	0.64	0.64	0.55	0.36	0.32	0.23
LDS^N-CBS	1.10	0.91	0.70	0.64	0.53	0.47
DDS-CBS	2.01	1.68	1.29	1.11	0.89	0.72
DDS^N-CBS	1.98	1.63	1.34	1.21	1.03	0.81

Table 7 LDS variants on the thirteen instances used in Vaessens *et al*. (1996)

Search strategy	1	2	3	4	5	10
DFS-CBS	2.26	2.12	1.94	1.77	1.77	1.72
LDS-CBS	1.75	1.28	1.03	0.92	0.92	0.79
LDS^N-CBS	1.50	0.99	0.87	0.86	0.86	0.78
DDS-CBS	2.68	2.47	2.20	1.56	1.51	1.26
DDS^N-CBS	2.68	2.40	1.87	1.59	1.56	1.32

ITE																									
DFS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LDS	0	1	2	1	2	3	2	3	4	1	2	3	2	3	4	3	4	5	2	3	4	3	4	5	4	5	6
LDS^N	0	1	1	1	2	2	1	2	2	1	2	2	2	3	3	2	3	3	1	2	2	2	3	3	2	3	3
DDS	0	3	4	2	4	5	3	4	5	1	4	5	3	5	6	4	5	6	2	4	5	3	5	6	4	5	6
DDS^N	0	3	3	2	3	3	2	3	3	1	3	3	2	3	3	2	3	3	1	3	3	2	3	3	2	3	3
ORD																									
DFS	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	&
LDS	A	D	J	C	I	Q	H	P	W	B	G	O	F	N	V	M	U	Z	E	L	T	K	S	Y	R	X	&
LDS^N	A	F	G	D	P	Q	E	R	S	B	J	K	H	T	U	I	V	W	C	N	O	L	X	Y	M	Z	&
DDS	A	E	I	C	J	P	F	K	Q	B	L	R	G	S	X	M	T	Y	D	N	U	H	V	Z	O	W	&
DDS^N	A	J	K	D	L	M	E	N	O	B	P	Q	F	R	S	G	T	U	C	V	W	H	X	Y	I	Z	&

Figure 1 The behavior of the five algorithms on a balanced ternary tree

4 Vehicle routing

Vehicle Routing Problems (VRP) are optimisation problems from the transportation industry. A VRP is defined by a set of customers (or nodes) i and a distance matrix ($d[i, j]$). Additionally, each customer may be given a duration, in which case the matrix d denotes traveling times, and a load which represents some weight to be picked up in a capacitated VRP. The goal is to find a set of routes that start from a given node (called the depot) and return to it, so that each customer is visited only once. The routes usually hold capacity constraints limiting the length or the load (the sum of the weights of the visited customers) of the routes. A VRP is an optimisation problem, where the objective function is the sum of the lengths of the routes, either for a bounded number of routes or for the minimal number of routes. When the number of routes is bounded by 1, the VRP is referred to as a Traveling Salesman Problem (TSP). The problem can be enriched with time windows specifying when each customer can be visited. In this case the problem is called VRP with time windows: VRPTW (resp. TSPTW). We refer the reader to Laporte (1992) or Gendreau *et al.* (1997) for a survey on the subject.

This section presents combinations of local and global search techniques for VRPTW on mid-size instances with 100 customers (see Solomon, 1987):

1. ILO: we show how to enrich a greedy construction algorithm with local improvements.
2. LDS: in VRP the heuristics are such that the confidence on the recommended choices can significantly vary throughout the nodes of the search tree. Thus, the idea is to modify LDS so that only one branch is explored when the heuristic strongly supports this branch, compared to the others.
3. Exact TSPTW optimisation: the sequence of visits on a route is completely optimised with an exact algorithm. Such local improvements are performed systematically throughout the greedy construction phase.

4.1 Basic components of vehicle routing

Real-world routing problems include a lot of side constraints (not every truck can go everywhere at anytime, drivers have breaks and meals, some tasks have higher priorities, etc.). Because of this additional complexity, the most commonly used algorithm is an insertion algorithm, one of the simplest algorithms for solving a VRP. The tasks to be visited are placed in a stack that may be sorted statically (once) or dynamically, and a set of empty routes is created. For each task, a set of candidate routes is selected, and the feasibility of the insertion is evaluated. The task is inserted into the best route found during this evaluation. This loop is run until all tasks have been inserted. Notice that an

important parameter is the valuation of the feasible route. A common and effective strategy is to pick the route for which the increase in length, due to the insertion, is minimal.

The key component is the node insertion procedure, since it must check all the side constraints. The use of constraint programming techniques is quite natural and has become popular in the last few years. CP techniques can be used either through the full resolution of the one-vehicle problem, which is the resolution of a small TSP with side-constraints, or it can be used to supplement a simple insertion heuristic by doing all the side-constraint checking. The solution that is most frequently used for industrial application is to combine an insertion heuristic with a post-optimisation phase where local moves are applied. However, the insertion heuristic is known to produce low-quality solutions, and the post-optimisation cannot correct much if the problem is large.

Rather than applying the local moves once the first solution is built, the principle of incremental local optimisation (ILO) is to apply local moves after each insertion and only for those moves that involve the new node that got inserted. The idea of ILO within an insertion algorithm had already produced good results in Gendreau *et al*. (1994), Russell (1995) and Kontoravdis and Bard (1995), and we proposed a first hybridisation in Caseau and Laburthe (1999).

Many local moves have been proposed in the literature (Gendreau *et al*., 1997). Most of the neighbourhoods are based on *k*-exchanges (or *k*-opt): *k* edges are removed from a solution and disconnected fractions of route are joined with *k* new edges. Those edges can be selected within one route (TSP neighbourhood) or across different routes. Specialised versions of such neighbourhoods transfer a node or a chain of nodes from one route to another.

The ILO procedure is used within a parameterised greedy insertion algorithm: $\text{INSERT}^{(i)}$. $\text{INSERT}^{(0)}$ greedily inserts the current node into the route, yielding the least increase in cost. $\text{INSERT}^{(1)}$ performs some 2-opt moves before selecting the best route. $\text{INSERT}^{(2)}$ and $\text{INSERT}^{(3)}$ add some 3-opt moves, and $\text{INSERT}^{(4)}$ adds a recovery strategy. When the insertion of the node seems infeasible, we try to reconstruct the route by inserting the new node first.

4.2 Exact optimisation of individual TSPTW

After each insertion, the incremental local optimisation tries to take advantage of any easy improvement that could be made on the set of (partial) tours. One can, however, be more systematic about the individual tour in which a new customer has just been inserted. Since, in most practical examples, the individual tours contain fewer than 15 nodes, finding the optimal tour is a small optimisation problem, which can be solved to optimality.

The ILO method incorporates such a component for the exact optimisation of individual tours which are Traveling Salesman Problems with Time Windows (TSPTW). We used an exact constraint-based branch-and-bound algorithm for the resolution of these small TSPTWs. This algorithm is presented in detail in Caseau and Laburthe (1997) (a similar approach is also presented in Pesant *et al*. (1998)) and relies on the formulation of a Hamiltonian cycle as a perfect matching in a modified bipartite graph. The minimal weight perfect matching constraint is propagated together with a sub-cycle elimination constraint, with a lower bound based on a look-ahead evaluation of regrets. The branch-and-bound algorithm explores a search tree based on decisions considering whether a node is followed in the tour by its closest reachable neighbour or not. This CP algorithm has many advantages over dynamic programming. Compared to a simple dynamic programming approach, it is faster and scales to somewhat larger problems (supporting the resolution of up to 30 nodes problems); compared to more complex dynamic programming algorithms such as that of Dumas *et al*. (1995), it does not rely on a strong discretisation of the data, which may lead to infeasible solutions. In any case, the CP algorithm is able to take into account many other additional constraints from a real dispatch problem, and it is fast. In particular, it was able to solve within 50 ms. all 15 node problems that we encountered.

This constraint algorithm was used for tours up to 30 nodes with a limit of 500 on the number of backtracks up to 15 nodes, 200 up to 20 nodes, and 25 from 20 to 30 nodes. Such a truncated search no longer guarantees the optimality of the solution. This exact algorithm is used in several places:

1. Within the incremental local optimisation process, each time a *k*-exchange modifies a route *r*, *r* is re-optimised with the exact TSPTW algorithm if *r* contains less than 30 nodes, and with 3-opt otherwise.
2. For each insertion, when the best route *r* in which to insert the current node has been found and after ILO has taken place, one tries to further optimise *r* with the exact TSPTW algorithm.

4.3 Producing better quality results with limited global search

The ILO method is well-suited for large problems because it is quite fast. For medium-sized problems, it can be used within a more complex strategy to obtain even better results. For instance, a Tabu Search can be applied using a different set of moves (such as those of Thompson and Psaraftis or Gendreau *et al*. (1994)) or three edge-exchanges that would not be considered by the ILO. Another approach is to extend the ILO with such a new set of moves that can also be applied incrementally between two insertions. Although we have made some successful experiments with these approaches, we want to propose in this paper a different approach based on extending the insertion algorithm into a search algorithm. It is quite natural to transform a greedy insertion algorithm into a branching one. For each node, we can branch on which route to insert the node into. However, this is not realistic since the size of the search space is too large, and there is no good bounding function available for pruning the tree. If we apply the previous branching scheme and let the algorithm run for a limited but long period of time (a few minutes), the improvement over the value of the heuristic is dismal.

A better idea is to apply a variant of LDS. The idea is to select only a few nodes for which the heuristic gave a weaker indication and, for these nodes, to branch on one alternative route. In Figure 2 we have marked the nodes for which the heuristic is not confident enough. The figure shows which leaves are visited depending on the maximal number of discrepancies.

If we limit the number of such discrepancies, we obtain an algorithm that is still polynomial in the size of the problem. The total number of failures during the execution of LDS(k) is at most 2^k. To decide whether the heuristic is confident enough or not, we consider the difference for the insertion cost between the best route that was selected and the next best route. If that difference is small enough (i.e. smaller than a parameter), we create a choice point and branch on the two best routes. If the parameter is too large, the few allowed choice points will be created early on in the search tree and no branching will be available deeper in the search. If it is too small, we may never find enough opportunities to branch.

A very interesting feature of LDS is that it can be used to limit the sensitivity to the external parameters of the greedy heuristics when we find that these parameters are hard to tune precisely. We can decide to branch on those insertions where a slightly different setting of the parameters would give a different decision. We can now give a range of values for each of those parameters and use the following discrepancy scheme. Each time the different values of the parameters would yield to a different route into which to insert the current node, we create a choice point and branch on all these possible routes. For instance, we can consider the bonus that we give to empty routes as the guiding parameter. We will create a choice point each time an empty route is selected that would not have been

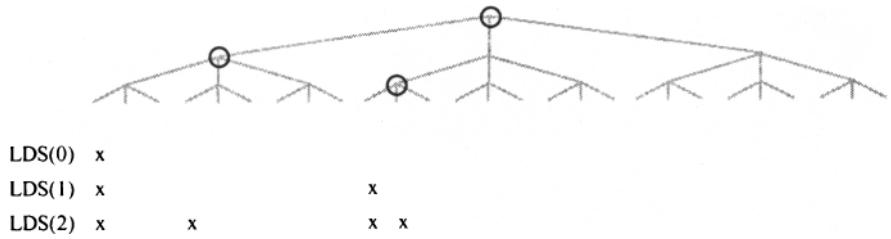

Figure 2 Nodes visited for three discrepancy levels of the modified LDS

selected if we had decided to use a smaller value for the bonus. We will branch on two routes: the empty one that got selected and the non-empty one that would have been selected otherwise (when it exists).

The LDS algorithm that we have implemented uses four forms of alternate branching:

1. When we evaluate each possible route, we keep both the best and the second best route. If the cost difference is less than a given parameter p_1, and if the allowed number of choice points is strictly positive, we branch using the previously defined method. This is only used if the second route is not empty.
2. When the best insertion of node i corresponds to a route creation, we consider all possibilities of removing a node j from some route r in order to make the insertion of i in r feasible. We then recursively search for feasible insertions for j. When the difference in cost between both possibilities (route creation versus best unfeasible insertion followed by a sequence of vertex relocations) is less than p_1, we consider both branches.
3. To avoid considering all the 2-opt moves across two routes (a large neighbourhood), after each tentative insertion, we only search for improving 2-opt moves close to the insertion point (a geographical filter is applied). After all tentative insertions, if the best route is an empty route, we further look for improving 2-opt moves: we select the best candidate among the attempted insertions and check that its insertion cost is less than p_2% of the insertion cost into the empty route. If such a 2-exchange is feasible we create a branch.
4. If the best route is an empty route, and if there was a better candidate that is full, we try to create a new route by splitting this full route. The principle is that it is sometimes better to split a route than to start a new one, if the new node to be inserted was close enough from an existing route.

For the Solomon benchmarks, we called the algorithm with *LDS*(4), $p_1 = 50$ and $p_2 = 60\%$. Hence, the total number of nodes in the search tree is bound by 2^4.

4.4 Results on Solomon's benchmarks

We now compare the heuristic H1 that was developed for large routing problems using ILO (INSERT$^{(4)}$) and a TSP constraint solver for the node insertion, with the heuristic H2 which is similar but also uses LDS with a limited number of branching points (the number of discrepancies is limited to 4).

Table 8 reports experiments on the famous set of 100 customers benchmarks proposed by Solomon (1987). For each set of instances, we report the average route length followed by the average number of routes per instance (indicated between parentheses). The first heuristic H1 is extremely fast (a few seconds) and gives results using, on average, 0.20 more routes per problem and 3.12% more distance than the solutions obtained by Rochat and Taillard (1995). If we add limited discrepancy search H2, we can further tighten the results by using only 0.1 more routes per problem and 1.5% more distance, while keeping running times around five minutes (PC Pentium II 300 MHz). If we restrict the distance comparison to the set of problems where we obtain the same number of routes as the heuristics, we are 6.5% away from Rochat and Taillard (1995). Since the tabu approach in Rochat and Taillard (1995)

Table 8 Comparing heuristics on the VRPTW benchmarks from Solomon

	H1: Full ILO	H2: LDS	Reference [RT 95]
R1 (12 instances)	1243.53 (12.67)	1233.34 (12.41)	1208.5 (12.16)
C1 (9 instances)	829.29 (10)	828.38 (10)	828.38 (10)
RC1 (8 instances)	1438.17 (12.12)	1403.74 (12)	1377.39 (11.87)
R2 (11 instances)	983.45 (3.18)	990.99 (3.09)	961.71 (2.91)
C2 (8 instances)	599.81 (3)	596.63 (3)	589.85 (3)
RC2 (8 instances)	1295.58 (3.37)	1220.99 (3.37)	1119.59 (3.37)

was allowed up to 1000s. per problem (on a 100 Mhz Silicon Graphics station), these heuristics are good challengers: the heuristic H1 is much faster and yields good results, while H2 is faster and yields excellent results.

We shall see in the next section that these results can be improved by also using shuffling (called Large Neighbourhood Search (LNS) in the context of vehicle routing). However, the improvements brought by LDS in this approach are significant and worth noting.

5 Meta-programming for meta-heuristics

Recent years have seen the rise of hybrid algorithms in most fields as well as the apparition of generic techniques that seem to prove useful in many different domains. In the following, we will use the term *meta-heuristics* to refer to all such techniques, including LDS, LNS and shuffling. Yet hybrid algorithms are not a panacea. They imply a very large amount of tuning and are often not robust enough: the combination that works well for a given data set does poorly on another one. Hybrid algorithms also cause software engineering problems because of their very nature, and the real world application of an algorithm that works well on academic benchmarks is often a challenging task.

The field of Vehicle Routing is an interesting example since real-world applications mostly rely on insertion algorithms, which are known to be poor heuristics but have two major advantages: they are incremental by nature, and they support easily the addition of domain-dependent side constraints. The use of constraint programming makes these heuristics even more attractive since the constraint solver can handle the side constraints and produces a high-quality insertion (Caseau & Laburthe, 1999). Real problems have lots of side-constraints, and the software engineering issues become rapidly overwhelming. This is why the use of local optimisation and meta-heuristics is still limited in real-world applications.

The problem that we want to address is twofold:

1. How can we build a library of meta-heuristics that is totally problem-independent, so that any insertion algorithm based on a constraint solver can be plugged?
2. How can we achieve the necessary tuning to produce, at a low cost, a robust solution for each different configuration?

The first question is motivated by the fact that the meta-heuristic aspect (especially with local optimisation) is the part of the software that is the most difficult to maintain when new constraints are added. There is a tremendous value in confining the domain-dependent part to where constraint-programming techniques can be used. The second question is drawn from our practical experience: the speed of the hardware, the runtime constraints, and the objective functions (total travel, number of routes, priority respect, etc.) all have a strong influence when designing a hybrid algorithm. For instance, it is actually a difficult problem to re-tune a hybrid algorithm when faster hardware is installed.

This section presents a preliminary contribution to these two questions, which already yields impressive and encouraging results. We built on the ideas expressed in the SALSA language (Caseau & Laburthe, 1998) and propose an algebra of hybrid algorithms for VRPTW. This algebra is generated by a set of operators that correspond to the various meta-heuristics that have been proposed in the literature. Each term in this algebra represents a combination of these meta-heuristics. The base heuristic to which the meta-components apply is simply an insertion heuristic where each new task is inserted into the current solution using a constraint solver (the one-route sub-problem is a TSP). Thus, the link with the routing domain is ensured only at the heuristic level, and the same algebra can be used for crew scheduling, where the TSP solver is replaced by a task scheduler.

The algebra can be used to test various combinations, and we identify some interesting hybrid approaches. However, the real value comes from applying learning techniques to either automatically tune an existing combination or discover a brand new one, following an approach similar to MULTI-TAC (Minton, 1997). Although we have only implemented a very preliminary learning algorithm, we report positive results that demonstrate the value of this approach. The consequence is that, instead of

simply providing yet another hybrid algorithm for the Solomon vehicle routing benchmarks, we have a framework that can be used for a large range of problems and configurations.

5.1 Meta-heuristics for insertion algorithms

We now present a set of well-known meta-heuristics that are all based on insertions and removals of nodes from routes. Thus, if we can use a node insertion procedure that checks all domain-dependent side-constraints, we can apply these meta-heuristics freely. It might be argued that the actual implementation or the parameter adjustment of these heuristics could be improved by taking the domain-specific aspects into account. However, the gain is rather small for a big loss since we can no longer reuse the library of meta-heuristics for different domains.

It is important to note that not all meta-heuristics share this property (splitting and recombining routes does not), but the ones that do are already a large subset and will be shown, together with ILO, to yield powerful hybrid combinations.

5.1.1 Limited discrepancy search

In this section, we use the term LDS loosely to describe a strategy simpler than the one presented in section 4 but very similar. Here the choice heuristic is to pick the feasible route for which the increase in travel is minimal. Applying the idea of LDS, we branch when two routes have very similar "insertion costs" and pick the obvious choice when one route clearly dominates the others. There are two parameters in our LDS scheme: the maximum number of branching points along a path in the search tree and the threshold for branching. These two parameters control the shape of the search tree and have, needless to say, a definite impact on the quality of the solutions.

5.1.2 Ejection chains

The search for ejection chains is a technique that was proposed a few years ago for tabu search approaches (Rego & Roucairol, 1996). We define the ejection digraph as follows: the vertices are the customers and an arc between two nodes a and b represents the fact that a can be inserted in the route that contains b once b is removed. An ejection chain is a chain of ejection arcs where the last node is free, which means that it can be inserted freely in a route that does not intersect the ejection chain, without removing any other node. Each time an ejection chain is found, we can compute its cost, which is the difference in total length once all the substitutions have been performed. The search for ejection chains can be used in two cases:

1. To insert a node that cannot be inserted otherwise. In this case, any chain is acceptable, but we prefer to use the chain with minimal cost.
2. As an optimisation technique, we remove one node n and try to find the ejection chain with root n with minimal cost.

The optimisation performed by the search and evaluation of ejection chains can be obtained by a classical push/pull local optimisation scheme (with a gradient control strategy), but the search for the lowest cost chain is much more efficient.

When we use ejection chains as an optimisation strategy, we need to pick a heuristic to choose the order in which the nodes are removed and then re-inserted through a chain. We used a simple approach where we try to first remove the nodes with the higher insertion cost, which is the difference between the lengths of the route with and without the node.

5.1.3 Ejection trees

We propose an extension of ejection chains. In an ejection tree we allow multiple arcs from one node a to $b_1, \ldots, b_n$ to represent the fact that the forced insertion of a into a route r is made possible by the ejection of $b_1, \ldots, b_n$. For one node a and a route r there are usually multiple subsets of such $\{b_1, \ldots, b_n\}$, so we use a heuristic to find a set as small as possible. An ejection tree is then a tree of root a such that all leaves are free nodes which can be inserted into different routes which all have an

empty intersection with the tree. There are many more ejection trees than there are chains, and the search for ejection trees with lowest possible cost must be controlled with topological parameters (maximum depth, maximum width, etc.). The use of ejection trees is very similar to the use of ejection chains. We can use it to insert a node that has no feasible route, or as a meta-heuristic by removing and re-inserting nodes.

To make this algorithm work, it is necessary to put an upper bound on the depth of the tree and on the branching factor. We only select no more than *k* routes for the forced insertion, by filtering the *k* best routes once all possible routes have been tried. Furthermore, we use a LDS scheme: each time we use a route that was not the best route found (the valuation is simply the weight of the ejected set), we count one discrepancy. The LDS approach simply means to cut all trees that would require more than *D* (a fixed parameter) discrepancies.

5.1.4 Large neighbourhood search

Large Neighbourhood Search (LNS) is the name given by Shaw (1998) to the application of shuffling (cf. section 2) to routing. The principle is to forget (remove) a fragment of the current solution and to rebuild it using Limited Discrepancy Search algorithm. For Job-Shop Scheduling, we have developed a large variety of heuristics to determine the fragment that is forgotten and used these heuristics in rotation until a fix-point was reached. Shaw introduced a heuristic randomised criterion for computing the forgotten set and proposed to use LDS to re-build the solution. We have implemented the same heuristic to select the set of *n* (an integer parameter) nodes that are removed from the current solution. We have extended this heuristic so that the nodes that are already without a route are picked first (when they exist).

The implementation of LNS is then straightforward: select a set of *k* nodes using Shaw's procedure and, then, remove them from the current solution. These nodes are then re-inserted using a LDS insertion algorithm. There are, therefore, four parameters needed to describe this algorithm: two for LDS (number of discrepancies and threshold), a randomness parameter (used for varying the heuristic from deterministic to totally random) and the number of nodes to be reinserted. As we shall see later, the procedure for re-constructing the solution could be anything, which opens many possible combinations.

5.2 An algebra for hybrid algorithms

5.2.1 Building terms

To experiment, automatically tune and create new hybrid algorithms, we need an abstract and concise representation. We could have used the SALSA language, but we found that the expressive power of SALSA was more than we needed for the coarse grain description of algorithmic components that we were considering. We decided instead to "project" SALSA onto our own domain; that is, to create an algebra whose terms represent a hybrid algorithm for routing using the previously defined techniques.

The algebra is defined in Figure 3. There are two kinds of terms: **<Build>** terms represent an algorithm that creates a solution; **<Optimise>** terms represent an algorithm that improves the (current) solution.

The definition of the elementary operators is straightforward.

INSERT$^{(i)}$ builds a solution by applying a greedy insertion approach and a varying level of ILO.

LDS(i,n,l) builds a solution by applying a limited discrepancy search on top of the INSERT$^{(i)}$ greedy heuristic. The parameter *n* represents the maximum number of discrepancies (number of branching points for one solution) and *l* represents the threshold. A LDS term can also be used as a generator of different solutions when it is used in a FORALL. A slightly enhanced version, *LDS+(i,n,l,T)*, applies the optimising term *T* at each node of the search tree generated by *LDS(i,n,l)*.

FORALL(T1,T2) produces all the solutions that can be built with *T1* which is necessarily a LDS (as opposed to only the best), and applies the post-optimisation step *T2* to each of them. The result is the best solution that was found.

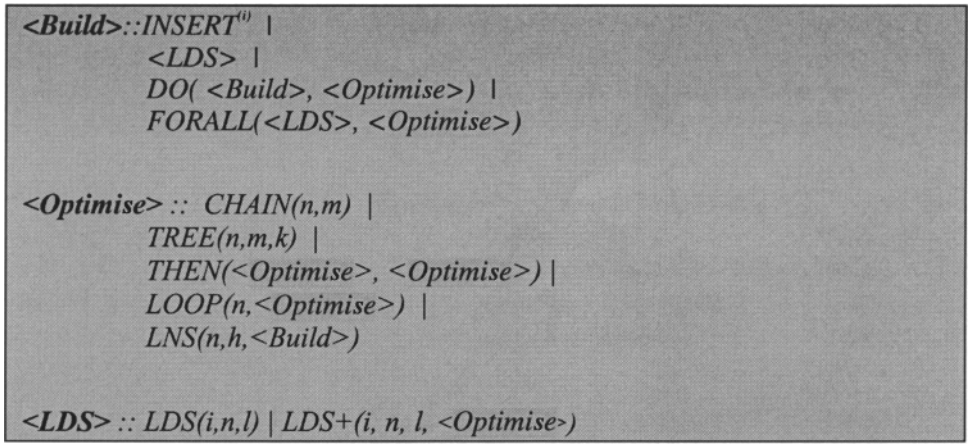

Figure 3 *Algebra for hybrid algorithms.*

CHAIN(n,m) is a post-optimisation step that selects *n* nodes using the heuristic represented by *m*, successively removes them (one at a time), and tries to re-insert them using an ejection chain.

TREE(n,m,k) is similar but uses an ejection tree strategy for the post-optimisation. The extra-parameter represents the number of discrepancies for the LDS search (of the ejection tree).

LNS(*n*,*h*,*T*) applies Large Neighbourhood Search as a post-optimisation step. We select *n* nodes using Shaws's heuristics with the *h* randomness parameter, and we rebuild the solution using the algorithm represented by *T*, which must be a **<Build>**.

DO(T1,T2) simply applies *T1* to build a solution and *T2* to post-optimise it.

THEN(T1,T2) is the composition of two optimisation algorithms *T1* and *T2*.

LOOP(n,T) repeats *n* times the optimisation algorithm *T*.

Here are some examples of algebraic terms.

*LDS(*3,3,100*)* represents a LDS search using the third level of ILO (every move is tried) but with the regular insertion procedure, trying 2^3 solutions (three choice points when the difference between the two best routes is less than 100) and returning the best.

DO(INSERT$^{(2)}$*,CHAIN(80,2))* is an algorithm obtained by combining a regular greedy heuristic with the second level of ILO with a post-optimisation phase of 80 removal/re-insertion through an ejection chain.

FORALL(LDS(0,4,100),LOOP(3,TREE(5,2))) is an algorithm that performs a LDS search with no ILO and 2^4 branching points and, then, applies three times an ejection tree post-optimisation step for each intermediate solution.

5.2.2 Implementation results

The grammar is embedded into a CLAIRE class hierarchy: each operator is represented by a class and each term A(. . .) is an instance of the class A. The use of constructors and pretty-printing makes the algebra an extension of the programming language, making it very easy to manipulate terms (they can be interpreted at the top-level, written in a file, cut-and-pasted, etc.).

To evaluate the algorithms represented by the terms, we have defined a small interpreter.[3] The method *run* is defined on each operator class and applies the algorithm represented by the operator (a **<Build>** or an **<Optimise>**) to the current problem (and the current solution for an **<Optimise>**).

The metric for complexity that we use is the number of calls to the insertion procedure. The advantage of this approach is that it is machine independent and is easier to predict based on the

[3] Only the control represented by the algebraic term is interpreted; everything else is compiled through the generation of C++ code (Caseau and Laburthe, 1996).

structure of the term. The CPU time is roughly linear in the number of insertions, and it is convenient to use a metric that is independent from the CP insertion procedure, which we would like to see as an external functional parameter to the whole system.

To evaluate the value of a term, we have defined a *test* method which takes a set of test files, runs the algorithm and produces average results. Since we use a randomised criteria, we need to make multiple runs. However, the standard deviation is smaller than with other approaches (Rochat & Taillard, 1995; Taillard *et al*., 1995).

In the experiments, all figures report the number of insertions and the average value of the objective function, which is defined as the sum of the total lengths plus a penalty for the excess in the number of routes over a pre-defined objective.

A few hand-crafted examples

We have used the algebra to evaluate the relative strength of the various optimisation techniques. Since we are using a Solomon benchmark as the test set (the R1* set), we have mostly investigated terms that are using a few hundred of thousands insertions, corresponding to run times of a few seconds. Table 9 shows a set of terms that we have produced manually to best represent each meta-heuristic. The complexities are adjusted so that they all have the same order of magnitude.

In Tables 9 through 11, we use an objective function combining the total distance with the number of trucks. The coefficients of the combination can be tuned either for minimising the total travel or for minimising the number of trucks and, then, the total travel. For each algorithm, we report the average objective value over all R1 instances in the Solomon suite and the number of insertions (as an indicator of time performance).

LNS and complex strategies

We have also used the algebra to compare more complex strategies that produce better quality solutions with run times that are comparable to what is found in the literature for the Solomon benchmarks. The runtime per experiment varies from 5 to 30 minutes of CPU time on a Pentium II 366 MHz laptop. We have compared our average results to those ones provided by Taillard *et al*. (1995) and Shaw (1998), since they provide average results for various running-times. This makes the comparison difficult because of the different hardware, but is still meaningful. It is important to note that the standard deviation of our results is relatively small, especially when the complexity is large.

These preliminary results are impressive, especially with a limited amount of CPU time. If we limit ourselves to 5 minutes of CPU time on our machine (10 minutes for Shaw (1998) or 30 minutes for

Table 9 Combining ejected chains and trees

Terms	Objective (#trucks, travel)	Performance	Objective (travel)	Performance
LDS(3,8,50)	53190	161Ki	22314	182Ki
DO(LDS(4,3,100),CHAIN(80,2))	59113	203Ki	22041	187Ki
DO(LDS(3,3,100), LOOP(30,LNS(10,4,LDS(4,4,1000)))	43813	155Ki	21981	300Ki
FORALL(LDS(3,2,100),CHAIN(20,2))	59732	190Ki	22179	142Ki
DO(INSERT$^{(3)}$,CHAIN(90,2))	70857	169Ki	22165	154Ki
DO(LDS(3,2,100),LOOP(2,TREE(40,2)))	47038	278Ki	22383	120Ki
DO(LDS(3,2,100),LOOP(6,CHAIN(25,2)))	61003	271Ki	22127	235Ki
DO(LDS(3,0,100),THEN (LOOP(30,LNS(4,4,LDS(3,3,1000)))) THEN(LOOP(25,LNS(6,4,LDS(3,3,1000)))) THEN(LOOP(20,LNS(8,4,LDS(3,3,1000)))) THEN(LOOP(10,LNS(10,4,LDS(3,3,1000)))) THEN(LOOP(8,LNS(12,4,LDS(3,3,1000))))))	38095	271Ki	21970	307Ki

Taillard *et al*. (1995)), we get an average number of routes of 12.23 on R1 and 12.0 on RC1, whereas Taillard *et al*. (1995) and Shaw (1998), respectively, get 12.64 and 12.08, and 12.45 and 12.05. If we raise the available CPU time by a factor of 3, we get the same results for RC1 (12.0) and still a little better for R1 (12.19 vs. 12.39 and 12.33). The increase in CPU time has a smaller effect and we cannot obtain results as good as those of the best hybrid combination of Tabu and Genetic algorithms if a longer run-time is acceptable. More generally, it can be said that the approach presented here is a significant contribution to solving the Solomon benchmarks with a limited amount of time.

This demonstrates that LNS + ILO is a powerful combination for these types of problems. It is more powerful than the combination of ILO and LDS that we presented in the previous section. Moreover, the code for this new combination is much shorter and much simpler than our previous hybrid algorithm.

5.3 Learning new terms

The efficiency of the previous combinations (I + LO, LDS + ILO, LNS + ILO) suggests that we take advantage of the algebra for expressing algorithms and experiment with automatic parameter tuning and automatic discovery of better combinations of methods.

5.3.1 Invention and mutations

The first step to discovering new terms is to create them randomly. We have implemented an invention method that is defined by structural induction from the grammar definition. The choice among the different subclasses (e.g., what to pick when we need an **< Optimise >**?) is done by using a random number generator and a pre-determined distribution. The result is that we can create terms with an arbitrary complexity (there are no boundaries on the level of recursion, but the invention algorithm terminates with probability 1). We worked in the admissible following directions.

- A key concept to guide the invention is the expected complexity for a term which can roughly be predicted by structural induction. There is no point in generating terms that are obviously too complex, so we guide the invention by giving a target complexity.
- The second step is to be able to mutate terms, i.e., to modify them partially. Mutation is also defined by structural induction according to two parameters. A first parameter tells if we want a shallow modification, an average modification, or a deep modification. In the first case, the structure of the term does not change and the mutation algorithm only changes the leaf constants that are involved in the term (integers). Moreover, only small changes for these parameters are supported. In the second case, the type (class) does not change, but large changes in the parameters are allowed and, with a given probability, some sub-terms can be replaced by terms of other classes. In the last case, a complete substitution with a different term is allowed (with a given probability). The second parameter gives the actual performance of the term, as measured in the experiment. The mutation algorithm tries to adjust the term in such a way that the (real) complexity of the new term is as close as possible to the global objective. This compensates the biases of the complexity oracle quite effectively.

5.3.2 Learning new terms

We have implemented a learning algorithm as a proof-of-concept for our approach. It turns out that, even with this crude approach, we get interesting results, but the development of a true learning system is beyond the scope of this paper and constitutes the topic of the remainder of our research project.

The principle of our learning algorithm is explained as a combination of hill climbing and Monte-Carlo optimisation, and incorporates some mechanisms to support a genetic algorithm in the future. We use a pool of current terms (we currently use 10 terms), and run the following step for a given number of iterations (20 in our examples). We run the tests using the sample data set and store the results (objective function and performance). We select the three best candidates and mutate them at three different levels (level 1, 2, and 3 as explained earlier). We keep the current best, add eight out of the nine terms produced and add a brand new invented term.

The next step in developing a better learning algorithm will include raising the size of the pool and the number of iterations, as well as introducing some crossing between terms. In addition, we will explore the use of abstract patterns such as the relational cliché approach described in Siverstein and Pazzani (1991) to expand the search space around the most promising terms and to abstract commonalties from terms that have been successful in the past. Although our initial experiments have shown that changing even the performance objective for a problem can lead to the discovery of different terms, it makes good intuitive sense that the ways in which the meta-heuristics themselves can be effectively combined are more structured. The abstract patterns uncovered by a 'cliché learner' could very well capture the implicit knowledge of how to combine the strategies employed by these meta-heuristics. The keys, here, are to form the appropriate generalisation language to describe the pattern abstractions and to learn patterns that are restrictive enough to effectively guide the search, yet, general enough to uncover any natural patterns that may exist.

5.3.3 Tailoring algorithms for travel optimisation

The first experiment applies our learning algorithm to the discovery of an algorithm for minimising the total travel time within a relatively strict computation time (the complexity goal is set at 50Ki). We first report in Table 10 the scores of different terms: the first four were built manually to solve this problem, the last term is the result of the learning algorithm.

This is quite an astonishing result, since the term found by the algorithm is significantly better than those that we had thought about initially, which undoubtedly offer new ideas for hand-picked terms.

5.3.4 Tailoring algorithms for route optimisation

The second experiment is similar, but we changed the objective function since our goal now is to minimise the number of trucks. We have included in the list the term invented for the previous experiment. The result, shown on Table 11, is really interesting since the learning algorithm found a

Table 10 Learning travel optimisation

Term	Objective	Run-time (i)
LDS(3,5,0)	22385	27Ki
DO(LDS(3,3,100), LOOP(8,LNS(10,4,LDS(4,4,1000))))	22147	99Ki
DO(LDS(3,2,100),TREE(20,2))	22439	23Ki
FORALL(LDS(3,2,100),CHAIN(8,2))	22310	59Ki
FORALL(LDS + (0,2,2936,CHAIN(1,1)), LOOP(48,LOOP(4, LNS(3,16,LDS(3,1,287)))))	**21946** ***(invented !)***	**57ki**

Table 11 Learning truck optimisation

Term	Objective	Run-time (i)
LDS(3,5,0)	53471	24Ki
DO(LDS(3,3,100), LOOP(8,LNS(10,4,LDS(4,4,1000))))	45478	49Ki
DO(LDS(3,2,100),TREE(20,2))	45463	52Ki
FORALL(LDS(3,2,100),CHAIN(8,2))	54837	84Ki
FORALL(LDS + (0,2,2936,CHAIN(1,1)), LOOP(48,LOOP(4, LNS(3,16,LDS(3,1,287)))))	39730 (old invented !) 57ki	
DO(LDS(2,5,145), THEN(CHAIN(5,2),TREE(9,2,2)))	**36531** ***(new invented!)***	**56Ki**

combination of techniques that is different from the one used in the previous experiment and which works significantly better (this is an average of 12 problems).

5.3.5 Tailoring algorithms for robust results

Our meta-heuristic factory appeared to be robust even for different complexity goals. We have tried to simulate what happens when a new piece of hardware is introduced. In most applications, the hybrid algorithm needs to be re-tuned, and it is not always easy to find out how the parameters need to be adjusted to best take advantage of the increased computing power (when the same limit in computational time is required). We applied the learning algorithm with a complexity goal three times larger (in number of insertions) than the results reported in sections 5.3.3 and 5.3.4. The automatic learner produced an algorithm yielding results that were 3% better than the result of hand-tuning the previous (old invented) best solution and 9% better than our hand-picked one.

Our last experiment illustrates our initial remark that hybrid algorithms require non-trivial tuning as part of their maintenance. We have considered that we had a new fleet of trucks with a load capacity reduced by half. It is well known that the modification in truck capacity causes an important change in the performances of the various algorithms. The new term that was invented is not only slightly better in quality (about 0.1%), it is also much closer to the original complexity goal (47Ki when the old invented term necessitated 74Ki).

6 Conclusion

This paper has presented combinations of global and local search on three combinatorial optimisation problems. For job-shop scheduling, we have shown that global search could be used as a way to explore neighbourhoods defined by fragments of solutions. For preemptive job-shop scheduling, we have shown that very good results are obtained by LDS combined with extensive constraint propagation. For vehicle routing, we have shown that insertion heuristics, incremental local optimisation, LDS, and LNS could be combined in many interesting ways. For all three problems, these new combinations have been shown to produce significant improvements on other methods based on one single optimisation technique.

Starting from the experiments for vehicle routing, we have proposed an algebra for representing hybrid algorithms that use push/pull meta-heuristics. Following the principles that were established with SALSA, which offers a common formalism for both CP branching schemes and classical OR branch-and-bounds algorithms, it is possible to represent a large family of meta-heuristics with a small set of parameterised operators. This abstraction has many advantages.

Genericity: the library can be reused for other task assignment problems.
Ease of maintenance: the abstraction provides a better capture of the design, as was noticed for SALSA.
Ease of tuning: the parameters are more visible and the handling of new algorithms is made easier.
Opportunity of automatic discovery.

The most novel aspect of this work is probably the automatic discovery of meta-heuristics. Obtaining yet another great set of solutions on a few academic benchmarks may be of little significance if the algorithm is not robust or is difficult to scale. The fact that real-world problems are often both bigger and more complex than academic benchmarks advocates automatic tuning. Our hope is that this approach will make hybrid algorithms more practical.

We consider this project to be a first step towards a new paradigm for industrial applications where we decompose the application into two separate pieces that can be combined independently. The first piece uses constraint programming to model the domain-dependent part of the problem, because CP is the best technology for representing side constraints. The second piece is a meta-heuristic package that is totally generic. There is a small price to be paid for genericity (no smart trick to guide the meta-heuristics), but the result is much more maintainable and can be tested on a variety of

problem-dependent insertion modules. The flexibility of this meta-heuristic package is achieved by the algebraic model and the automatic tuning algorithm.

References

Adams, J, Balas, E and Zawack, D, 1988. "The shifting bottleneck procedure for job shop scheduling" *Management Science* **34** 391–401.

Applegate, D and Cook, B, 1991. "A computational study of the job shop scheduling problem" *ORSA Journal of Computing* **3**(2).

Balas, E and Vazacopoulos, A, 1998. "Guided local search with shifting bottleneck for job shop scheduling" *Management Science* **44**(2).

Baptiste, P and Le Pape, C, 1996. "Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling" *Proceedings of the 15th Workshop of the U.K. Planning Special Interest Group* Liverpool, UK.

Baptiste, P, Le Pape, C and Nuijten, W, 1995. "Constraint-based optimization and approximation for job-shop scheduling" *Proceedings of the AAAI-SIGMAN Workshop on Intelligent Manufacturing Systems, IJCAI* Montréal, Québec.

Beck, JC, Davenport, AJ, Sitarski, EM and Fox, MS, 1997. "Texture-based heuristics for scheduling revisited" *Proceedings of the 14th National Conference on Artificial Intelligence* MIT Press.

Carlier, J and Pinson, E, 1989. "An algorithm for solving the job shop problem" *Management Science* **35**(2).

Carlier, J and Pinson, E, 1990. "A practical use of Jackson's preemptive schedule for solving the job-shop problem" *Annals of Operations Research* **26** 269–287.

Carlier, J and Pinson, E, 1994. "Adjustments of heads and tails for the job-shop problem" *Euro Journal of Operations Research* **78** 146–161.

Caseau, Y and Laburthe, F, 1994. "Improved CLP scheduling with task intervals" *Proceedings of the 11th International Conference on Logic Programming* MIT Press, 369–383.

Caseau, Y and Laburthe, F, 1995. "Disjunctive scheduling with task interval" *LIENS Technical Report 95-25* École Normale Supérieure, 45 rue d'Ulm, 75005 Paris, France.

Caseau, Y and Laburthe, F, 1996. "Introduction to the Claire programming language" *LIENS Report 96-15*, École Normale Supérieure.

Caseau, Y and Laburthe, F, 1997. "Solving small TSPs with constraints" *Proceedings of the 14th International Conference on Logic Programming* MIT Press.

Caseau, Y and Laburthe, F, 1998. "SALSA: A language for search algorithms" *Proceedings of the 4th Conference on Principles and Practice of Constraint Programming* Springer-Verlag, LNCS 1520, 310–324.

Caseau, Y and Laburthe, F, 1999. "Heuristics for large constrained vehicle routing problems" *Journal of Heuristics* **5**(3) 281–303.

Cesta, A and Oddi, A, 1996. "Gaining efficiency and flexibility in the simple temporal problem" *Proceedings of the 3rd International Workshop on Temporal Representation and Reasoning* 45–50.

Dell'Amico, M and Trubian, M, 1993. "Applying tabu-search to the job-shop scheduling problem" *Annals of Operations Research* **41** 231–252.

Dumas *et al*., 1995. "An optimal algorithm for the traveling salesman problem with time windows" *Operations Research* **43** 367–371.

Garey, MR and Johnson, DS, 1979. *Computers and Intractability. A Guide to the Theory of NP-Completeness* WH Freeman.

Gendreau, M, Hertz, A and Laporte, G, 1994. "A tabu search heuristic for the vehicle routing problem" *Management Science* **40** 1276–1290.

Gendreau, M, Laporte, G and Potvin, JY, 1997. "Vehicle routing: modern heuristics" in *Local Search in Combinatorial Optimization* E Aarts and JK Lenstra (eds.), Wiley.

Gondran, M and Minoux, M, 1984. *Graphs and Algorithms* Wiley, New York.

Harvey, W and Ginsberg, M, 1995. "Limited discrepancy search" *Proceedings of the 14th International Joint Conference on Artificial Intelligence* Morgan Kaufmann, 607–615.

Kontoravdis, G and Bard, J, 1995. "A GRASP for the vehicle routing problem with time windows" *ORSA Journal of Computing* **7**(1).

Laporte, G, 1992. "The vehicle routing problem: an overview of exact and approximate algorithms" *Euro. Journal of Operations Research* **59** 345–358.

Le Pape, C, 1988. "Des systèmes d'ordonnancement flexibles et opportunistes" *PhD Thesis*, University Paris XI, Orsay, France (in French).

Le Pape, C, 1994. "Implementation of Resource Constraints in ILOG SCHEDULE: A Library for the Development of Constraint-Based Scheduling Systems" *Intelligent Systems Engineering* **3** 55–66.

Le Pape, C and Baptiste, P, 1996. "Constraint propagation techniques for disjunctive scheduling: The preemptive case" *Proceedings of the 12th European Conference on Artificial Intelligence* Wiley, 619–623.

Le Pape, C and Baptiste, P, 1998. "Resource constraints for preemptive job-shop scheduling" *Constraints* **3** 263–287.

Le Pape, C and Baptiste, P, 1999. "Heuristic control of a constraint-based algorithm for the preemptive job-shop scheduling problem" *Journal of Heuristics* **5**(3) 305–325.

Lin, S and Kernighan, BW, 1973. "An effective heuristic for the traveling salesman problem" *Operations Research* **21** 498–516.

Mautor, T and Michelon, P, 1997. "MIMAUSA: A new hybrid method combining exact solution and local search" *Proceedings of the 2nd International Conference on Meta-Heuristics*.

Martin, P and Shmoys, D, 1996. "A time-based approach to the Jobshop problem" *Proceedings of IPCO'5* Vancouver. M. Queyranne (ed.), Springer-Verlag, LNCS 1084, 389–403.

Minton, S, 1997. "Configurable solvers: Tailoring general methods to specific applications" *Proceedings of the 3rd Conference on Principles and Practice of Constraint Programming* G Smolka (ed.), Springer-Verlag, LNCS 1330, 372–374.

Nowicki, E and Smutnicki, C, 1996. "A fast tabu search algorithm for the job-shop problem" *Management Science* **42**(6) 797–813.

Nuijten, W, 1994. "Time and Resource Constrainted Scheduling, a Constraint Satisfaction Approach" *PhD Dissertation*, University of Eindhoven.

Péridy, L, 1996. "Le problème de job-shop: arbitrages et ajustements" *PhD Thesis*, Université de Technologie de Compiègne, Compiègne, France (in French).

Pesant, G, Gendreau, M, Potvin, J-Y and Rousseau, J-M, 1998. "An exact constraint logic programming algorithm for the travelling salesman problem with time windows" *Transportation Science* **32**(1) 12–29.

Pesant, G and Gendreau, M, 1996. "A view of local search in constraint programming" *Proceedings of the 2nd Conference on Principles and Practice of Constraint Programming* Springer-Verlag, LNCS, 353–366.

Rego, C and Roucairol, C, 1996. "A parallel tabu search algorithm using ejection chains for the vehicle routing problem" in *Meta-Heuristics: Theory and Applications* Kluwer.

Rochat, E and Taillard, E, 1995. "Probabilistic diversification and intensification in local search for vehicle routing" *Journal of Heuristics* **1** 147–167.

Rousseau, LM, Gendreau, M and Pesant, G, 1999. "Using constraint-based operators with variable neighborhood search to solve the vehicle routing problem with time windows" *CP-AI-OR'99 Workshop*, Ferrara.

Russell, R, 1995. "Hybrid heuristics for the vehicle routing problem with time windows" *Transportation Science* **29**(2).

Shaw, P, 1998. "Using constraint programming and local search methods to solve vehicle routing problems" *Proceedings of the 4th Conference on Principles and Practice of Constraint Programming*, M Maher and JF Puget (eds.), Springer-Verlag, LNCS 1520.

Silverstein, G and Pazzani, M, 1991. "Relational clichés: constraining constructive induction during relational learning" *Machine Learning, Proceedings of the Eighth International Workshop (ML91)* Morgan Kaufmann, 203–207.

Solomon, M, 1987. "Algorithms for the vehicle routing and scheduling problems with time window constraints" *Operations Research* **35**(2).

Steele, GL, Jr., 1980. "The Definition and Implementation of a Computer Programming Language Based on Constraints" *PhD Thesis*, Massachusetts Institute of Technology.

Taillard, E, Badeau, P, Gendreau, M, Guertain, F and Rousseau, JY, 1995. "A New Neighborhood Structure for the Vehicle Routing Problem with Time Windows" *Technical Report CRT-95-66*, Université de Montréal.

Thompson, P and Psaraftis, H, 1993. "Cyclic transfer algorithms for multi-vehicle routing and scheduling problems" *Operations Research* **41**(5).

Vaessens, RJM, Aarts, EHL and Lenstra, JK, 1996. "Job shop scheduling by local search" *Informs Journal of Computing* **8** 302–317.

Van Laarhoven, P, Aarts, E and Lenstra, JK, 1992. "Job shop by simulated annealing" *Operations Research* **40**(1).

Walsh, T, 1997. "Depth-bounded discrepancy search" *Proceedings of the 15th International Joint Conference on Artificial Intelligence* Morgan Kaufmann.