Combining Forces to Solve the Car Sequencing Problem

Laurent Perron and Paul Shaw

ILOG SA
9 rue de Verdun, 94253 Gentilly cedex, France
{lperron,pshaw}@ilog.fr

Abstract. Car sequencing is a well-known difficult problem. It has resisted and still resists the best techniques launched against it. Instead of creating a sophisticated search technique specifically designed and tuned for this problem, we will combine different simple local search-like methods using a portfolio of algorithms framework. In practice, we will base our solver on a powerful LNS algorithm and we will use the other local search-like algorithms as a diversification schema for it. The result is an algorithm is competitive with the best known approaches.

1 Introduction

Car sequencing [1, 2, 3, 4, 5, 6] is a standard feasibility problem in the Constraint Programming community. It is known for its difficulty and there exists no definitive method to solve it. Some instances are part of the CSP Lib repository.

As with any difficult problem, we can find two approaches to solve it. The first one is based on complete search and has the ability to prove the existence or the non-existence of a solution. This approach uses the maximum amount of propagation [5]. The second approach is based on local search methods and derivatives. These methods are, by nature, built to find feasible solutions and are not able to prove the non-existence of feasible solutions. Recent efforts have shown important improvements in this area [7].

Structured Large Neighborhood Search methods have been proved to be successful on a variety of hard combinatorial problems, *e.g.*, vehicle routing [8, 9], scheduling (job-shop: shuffle [10], shifting bottleneck [11], forget-and-extend [12]; RCPSP: block neighborhood [13]), network design [14], frequency allocation [15]. It was natural to try structured LNS on the car sequencing problem.

The first phase of our work was rewarded by mixed results. Even with the implementation of special improvements (or tricks), the Large Neighborhood Search Solver was not giving satisfactory results as the search was stuck in local minima. At this point, we saw two possible routes. The first one involved creating dedicated and complex neighborhoods. The second one relied on the portfolio of algorithms [16] and uses specific search algorithms as diversification routines on top of the LNS solver. We chose the second route as it required much less effort and, to this effect, we designed two simple and specific local search improvement routines.

J.-C. Régin and M. Rueher (Eds.): CPAIOR 2004, LNCS 3011, pp. 225–239, 2004.

In the end, we ended up with a effective solver for the car sequencing problem, built with pieces of rather simple search algorithms.

The rest of the article is divided as follows: section 2 describes the car sequencing problem in more detail, section 3 describes the LNS solver and all the improvements we tried to add to it, the successful ones as well as the unsuccessful ones. We then present the block swapping method in section 4 and the sequence shifting method in section 5. These three methods are combined using an algorithm portfolio. We give experimental results in section 6.

2 The Car Sequencing Problem

The car sequencing problem is concerned with ordering the set of cars to be fed along a production line so that various options on the cars can be installed without over-burdening the production line. Options on the cars are things like air conditioning, sunroof, DVD player, and so on. Each option is installed by a different working bay. Each of these bays has a different capacity which is specified as the proportion of cars on which the option can be installed.

2.1 The Original Problem

The original car sequencing problem is stated as follows: We are given a set of options O and a set of configurations $K = \{k | k \subseteq O\}$. For each configuration $k \in K$, we associate a demand d_k which is the number of cars to be built with that configuration. We denote by n the total number of cars to be built: $n = \sum_{k \in K} d_k$. For each option $o \in O$ we define a *sequence length* l_o and a *capacity* c_o which state that no more than c_o cars in any sequence of l_o cars can have option o installed. The throughput of an option o is given as a fraction c_o/l_o. Given a sequence of configurations on the production line $s =< s_1, \ldots, s_n >$, we can state that:

$$\forall o \in O, \forall x \in \{1 \ldots n - l_o + 1\} \sum_{i=x}^{x+l_o-1} U_o(s_i) \leq c_o$$

where $U_o(k) = 1$ if $o \in k$ and 0 otherwise.

This statement of the problem seeks to produce a sequence s of cars which violates none of the capacity restrictions of the installation bays; it is a decision problem.

2.2 From a Decision Problem to an Optimization Problem

In this paper, we wish to apply local search techniques to the car sequencing problem. As such, we need available a notion of quality of a sequence even if it does not satisfy all the capacity constraints. Approaches in the past have softened the capacity constraints and added a cost when the capacity of any installation bay is exceeded; for example, see [7, 17]. Such an approach then seeks a solution of violation zero by a process of improvement (or reduction) of this violation.

However, when Constraint Programming is used, this violation-based representation can be undesirable as it results in little propagation until the violation of the partial sequence closely approaches or meets its upper bound. (This upper bound could be imposed by a local search, indicating that a solution better than a certain quality should be found.) Only at this point does any propagation into the sequence variables begin to occur.

We instead investigate an alternative model which keeps the capacity constraints as hard constraints but relaxes the problem by adding some additional cars of a single new configuration. This additional configuration is an 'empty' configuration: it requires no options and can be thought of as a stall in the production line. Such configurations can be inserted into a sequence when no 'real' configurations are possible there. This allows capacity restrictions to be respected by lengthening the sequence of real configurations. The idea, then, is to process all the original cars in the least possible number of slots. If all empty configurations come at the end of the sequence, then a solution to the original decision problem has been found. We introduce a cost variable c which is defined to be the number of slots required to install all cars minus n. So, when $c = 0$, no empty configurations are interspersed with real ones, and a solution to the original decision problem has been found.

3 Large Neighborhood Search

Large Neighborhood Search (LNS) is a technique which is based upon a combination of local and tree-based search methods. As such, it is an ideal candidate to be used when one wishes to perform local search in a Constraint Programming environment. The basic idea is that one iteratively relaxes a part of the problem, and then re-optimizes that part, hoping to find better solutions at each iteration. Constraint programming can be used to add bounds on the search variable to ensure that the new solution found is not worse than the current one.

Large Neighborhood Search has its roots in the shuffling procedures of [10], but has become more popular of late thanks to its successes [8, 18, 19].

The main challenge in Large Neighborhood Search is knowing which part of the problem should be relaxed and re-optimized. A random choice rarely does as well as a more reasoned one, as was demonstrated in [8], for example. As such, in this paper, much of the investigation will be based on exactly how LNS can be used successfully on this car sequencing model, and what choices of relaxation need to be made to assure this.

3.1 Basic Model

A LNS Solver is based on two components: the search part and the neighborhood builder. Here, the search part is a depth-first search assignment procedure that instantiates cars in one direction from the beginning to the end of the sequence. The values are chosen randomly with a probability of choosing a value v proportional to the number of unassigned cars for the same configuration v.

We have implemented two types of LNS neighborhoods. The solver chooses one of them randomly at each iteration of the improvement loop with the same probability. They both rely on a size parameter s. The first one is a purely random one that freezes all cars except for the $4s$ randomly chosen ones. The second one freezes all cars except cars in an interval of length s randomly placed on the sequence, and except cars in the tail. Cars in the tail are defined as all cars beyond the initial number of cars. (These cars are in fact pushed there by stalls.) We have observed that removing one neighborhood degrades performances.

The solver is then organized in the following way. Given an intermediate solution, we try to improve it with LNS. Thus, we choose one neighborhood randomly and freeze all variables that do not appear in the neighborhood. Then we choose a small search limit (here a fail limit), and we start a search on the unfrozen variables with the goal described above. At each iteration, we select the first acceptable solution (strictly better or equal in case of walking).

Finally, the goal is written in such a way that it will never insert a stall in the sequence, thus stalls only appears through propagation and not through goal programming.

3.2 Improvements and Non-improvements

In the first phase of our work, we tried to improve the performance of our solver by adding special techniques to the base LNS solver. We briefly describe them and indicate how successful these modifications were.

Walking. This improvement is important as we can see at the end of the experimental results section. Walking implies accepting equivalent intermediate solutions in a search iteration instead of requiring a strictly better one. Intuitively, it allows walking over plateaus.

Tail Optimization. We tried to periodically fully re-optimize the tail of the sequence. As this tail is generally small, this is not computationally expensive. Unfortunately, no improvements came from this method.

Swap-based Local Search. We also tried to apply a round of moves by swapping cars. Unfortunately, the space of feasible solutions is sparse and to find possible moves with this method, we had to relax the objective constraint. This allowed the local search to diverge towards high cost solutions and it never came back. We did not keep this method.

Circle Optimization. When there is only one stall remaining in the sequence, we can try to re-optimize a section around this stall. Thus, given a distance ρ, we can freeze all cars whose distance to the stall is greater than ρ. Then we can try to re-optimize the unfrozen part. Unfortunately, this never worked. Even with ρ around 15, meaning 31 cars to re-optimize, this method just wasted time and never found one feasible solution to the problem.

Growing the Neighborhood. We also tried to grow the size of the neighborhood slowly after repeated failures. Unfortunately, this did not allow the solver to escape the local minimum and just slowed down each iteration. Thus the system was slower and was not finding new solutions. We did not keep this change.

4 Block Swapping Methods

Following the long sequence of unsuccessful attempts at improving the LNS solver, we thought of inserting between each LNS iteration another improvement technique that would have the same characteristics in terms of running time, and that would perform a search that would be orthogonal. Thus we would not spend too much time on this method while it could act as a diversification schema for the LNS solver.

The first diversification schema we implemented was a method that cuts the sequence into blocks and tries to rearrange them.

The current solution is cut into blocks of random length (the minimum and the maximum of this length are parameters of this method). We also make sure there are no 'empty' configurations in the block and we remove the portions before the first and after the last empty configuration. Then we try to rearrange these blocks without changing the sequence inside a block.

This method is simple but describes moves that are not achievable with our LNS neighborhood and with the LNS approach in general, as it may changes many (and potentially all) variables if we insert a block near the beginning of the sequence.

There are many improvements that can be added to this simple block swapping schema.

4.1 Walking

Walking is always possible as a meta-heuristic on top of the block swapping routine. We will show the effect of block swapping on the performance of the complete solver in the experimental section.

4.2 Changing the Order of Instantiations of Blocks

As we have a limited number of fails, only a small number of swaps will be tried in a call to the block swapping improvement method. We have tried different block instantiation heuristics. The first one is simply to assign them from the beginning to the end of the sequence and to do a simple depth first search. This heuristic can be improved by using SBS [20] instead of depth-first search.

We can also change the order of blocks when they are assigned, trying to prioritize the small blocks or to give a random order. These two orders did not improve the performance of the solver.

However, we kept the SBS modification as it did improve the results of our solver a little. This is compatible with the results we obtained in [19].

4.3 Changing the Size of Blocks

We also tried to change the size of blocks dynamically during the search in case of repeated failure to improve the current solution. We found no interesting results

when doing so and thus we did not keep any block size modification schema in our solver.

Yet, we did experiment with different maximum block sizes. The results are reported in the experimental section. The minimum block size is set to three.

5 Sequence Shifting Methods

The block swapping method did improve the performance of the solver a lot as can be seen in the experimental section phase. Unfortunately, there were still soluble problems that were not solved with the combination of LNS and block swapping. We then decided to implement another improvement method that would also add something to the block swapping and the LNS methods.

We decided to implement a basic improvement schema that tries to shift the sequence of cars towards the beginning, leaving the tail to be re-optimized.

Here is another search method that could be seen as a specialized kind of LNS. We shift the cars towards the start of the sequence by a given amount and re-optimize the unassigned tail using the default search goal.

As with the other two methods, we tried different modifications to this method.

5.1 Walking

At this point in our work, walking was a natural idea to try. Strangely, it did not improve our solver and even degraded its performance. Thus it was not kept.

5.2 Growing

The conclusion here is the same as with the block swapping phase. During all our experiments, we did not find any simple length changing schema that would improve the results of the solver. Thus, we did not keep any in our solver.

Yet, we did experiment with different fixed maximum lengths and these results are visible in the experimental results section. In those experiments, the minimum shift length is set to one.

6 Experimental Results

We will give two kinds of results in this section. The first part deals with comparative results between our solver and a re-implementation of the Comet article [7]. The second part describes the effect of parameter tuning on the performance of our solver.

6.1 Experimental Context

In order to compare ourselves with the competition, we used what we thought was the best competitor, namely the Comet code from [7]. Unfortunately, at the time of writing the article, we did not have access to the comet solver, thus we re-implemented the algorithm described in Pascal and Laurent's article from scratch (hereafter, we refer to this algorithm as RComet, for "re-implemented Comet"). While this was helpful, it is not completely satisfactory and we plan to do a comparison with the real thing (the real Comet code) as soon as possible.

Note that the Comet optimizer code uses a different optimization criterion from us: namely violations of the capacity constraints. To perform a more reasonable comparison, we decoded the order obtained by RComet to produce a number of empty configurations. This comparison is more meaningful than comparing violations to empty configurations, but it is far from perfect. The main problem is that RComet code has no incentive to minimize stalls, and it is possible to have two solutions a and b where a has lower violations but b has lower stalls. Thus, although we give figures in terms of stalls for the RComet code, comparisons only have real merit between run times for which both solvers found a solution of cost 0.

All tests were run on a Pentium-M 1.4 GHz with 1 GB of RAM. The code was compiled with Microsoft Visual Studio .NET 2003. Unless otherwise specified, the LNS approach uses block swapping with minimum size 3 and maximum size 10 and sequence shifts with minimum shift 1 and maximum shift 30. Furthermore, as we have tried avoid the addition of more control parameters, all three basic methods use the same fail limit, and are applied in sequence with the same number of calls.

All our experiments rely on randomness. To deal with unstable results, we propose to give results of typical runs. By typical, we mean that, given a seed s for the random generator, we run all instances with the same seed and then we do multiple full runs with different seeds. Then we select the results for the seed that produced the median of the results. We applied this methodology to both the RComet runs and runs from our solver.

All tests use a time limit of ten minutes. The tables report the time to get to the best solution in the median run.

6.2 Results on the CSP Lib Data Set

Table 1 presents the result we obtained on all the randomly generated instances of the CSP library. Unfortunately, except for one notable exception (90-05), all are easy and most can be solved in a few seconds. Thus, except for the 90-05 instance, we will not use these problems in the rest of the article.

Table 2 presents our results on the hard problems from the CSP Lib.

This result demonstrates the competitiveness of the LNS approach. In particular, our implementation find solutions where RComet does and, on average, finds solutions for problem 16-81, when RComet does not. On the other hand, when both solvers find a solution, the RComet is significantly faster than our solver.

Table 1. Results on random CSP Lib instances

Problem	Solving Time (s)
60-**	0-9
65-**	1-10
70-**	1-9
75-**	2-11
80-**	2-19
85-**	2-27
90-**	1-92

Table 2. Results on hard CSP Lib instances

Problem	our stalls	our time (s)	RComet stalls	RComet time (s)
10-93	3	228	6	50
16-81	0	296	1	250
19-71	2	51	2	55
21-90	2	38	2	15
26-82	0	93	0	29
36-92	2	241	2	65
4-72	0	412	0	95
41.66	0	23	0	12
6-76	6	12	6	9

6.3 Results on Randomly Generated Hard Instances

As we can see from the previous section, nearly all instances of the CSP Lib are easy and all except one are solved in less than one minute of CPU time. Thus we decided to generate hard instances. The random generator is described in the next section. The following section describes the results we obtained on these instances.

The Random Generator. The random generator we created takes three parameters: the number of cars n, the number of options o, and the number of configurations c. We first generate the o options, without replacement on the set of option throughputs: $\{p/q \mid p, q \in Z, 1 \leq p \leq 3, p < q \leq p + 2\}$.

We then generate the c configurations. The first o configurations are structured, the ith configuration involving only one option: option i. We generate the remaining $c - o$ configurations independently of each other, except for the stipulation that no duplicate nor empty configurations are created. Each of these configurations is generated by including each option independently with probability 1/3. Finally, the set of cars to construct is generated by choosing a configuration randomly and uniformly for each car.

Problems are then rejected on two simple criteria: too easy, or trivially infeasible. A problem is considered too easy if more than half of its options are used at less than 91%, and trivially infeasible if at least one of its options is

Table 3. Results on hard random instances

Problem	our stalls	our time (s)	RComet stalls	RComet time (s)
00	0	24	0	11
01	8	41	10	60
02	5	43	5	180
03	9	25	10	30
04	0	85	1	90
05	2	421	7	50
06	0	19	0	13
07	3	119	4	20
08	0	232	0	90
09	1	152	1	580
10	5	80	7	100
11	0	17	1	130
12	14	14	14	20
13	0	34	0	13
14	3	397	5	190
15	4	132	6	340
16	6	50	7	140
17	7	47	8	412
18	0	47	0	21
19	0	160	1	520

used at over 100%. The use of an option o is defined as: $\mu_o(\sum_{k \in K | U_o(k)=1} d_k)/n$. where $\mu_o(t)$ is the minimum length sequence needed to accommodate option o occurring t times: $\mu_o(t) = l_o((t-1)$ **div** $c_o) + ((t-1)$ **mod** $c_o) + 1$.

This approach can take a long time to generate challenging instances due to the large number of instances that are rejected. However, it does have the advantage that there is little bias in the generation, especially when it comes to the distribution of configurations. The random generator and the generated instances are freely available from the authors. For this paper, we generated problems with $n = 100, o = 8, c = 20$.

The Results. Table 3 give results on the 20 instances we have created with the previous problem generator with $n = 100, o = 8, c = 20$.

On this problem suite, again our solver finds solutions to all five problems solved by RComet, plus three more. On the problems where both methods find solutions, RComet is again faster, but to a much lesser extent than on the CSP Lib problems.

The conclusion from our experiments is that our solver is more robust and provides solutions to harder problems when RComet cannot. However, for the easier of the problems, RComet's local search approach is more efficient.

6.4 Analysis of the Portfolio of Algorithms

We measure the effect of each method on the overall results. To achieve this, we test all combinations of algorithms on a selected subset of problems.

Focus on Hard Feasible Instances. In this section, we focus on 12 hard feasible instances. We will try to evaluate the effect of different parameters on our search performance. We will consider instances 16-81, 26-92, 4-72, 41-22, 90-05 of the CSP Lib and instances 00, 04, 06, 08, 11, 18, 19 of our generated problems. For these problems and each parameter set, we will give the number of feasible solutions found with a cost of 0 (feasible for the original problem) and the running time, setting the time of a run to 600 seconds if the optimal solution is not found.

Experimental Results. Table 4 shows the results of all combinations of the three methods (LNS, Block Swapping and Shift) on the twelve hard feasible instances. The following table shows, for a given time, the number of problems solved in a time below the given time.

The results are clear. None of the extra methods (Block Swapping and Sequence Shifting) are effective solving methods as they are not able to find one feasible solution on the twelve problems.

Furthermore, block swapping is much better at improving our LNS implementation than sequence shifting. Furthermore, only the combination of the three methods is able to find results for all twelve problems.

Finally, we can see that LNS + shift is slower than LNS alone. Thus sequence shifting can only be seen as an extra diversification routine and not as a stand-alone improvement routine at all.

6.5 Tuning the Search Parameters

We keep the same twelve hard feasible instances to investigate the effect of parameter tuning on the result of the search. In particular, we will check the effect of changing the maximum size of blocks used in block swapping, of changing the

Table 4. Removing methods from the portfolio of algorithms

Methods	50s	100s	150s	200s	250s	300s	350s	400s	450s	500s	550s	600s
Full	3	6	7	9	10	11	11	11	12	12	12	12
Block + Shift	0	0	0	0	0	0	0	0	0	0	0	0
LNS + Shift	0	0	0	1	1	2	4	4	4	4	4	4
LNS + Block	3	6	7	7	7	8	8	8	8	8	8	8
LNS	0	1	2	2	3	3	4	4	4	4	4	4
Block	0	0	0	0	0	0	0	0	0	0	0	0
Shift	0	0	0	0	0	0	0	0	0	0	0	0

Table 5. Modifying the maximum block size (MBS)

MBS	50s	100s	150s	200s	250s	300s	350s	400s	450s	500s	550s	600s
5	4	6	7	7	7	7	7	8	8	8	8	9
8	4	7	7	8	8	9	9	10	11	11	11	11
10	3	6	7	9	10	11	11	11	12	12	12	12
12	3	6	6	8	9	9	9	9	9	9	10	10
15	3	5	6	6	8	8	8	8	9	9	10	10
18	2	6	7	8	9	9	9	9	9	9	9	9
20	2	5	5	6	7	8	9	10	10	10	10	10
22	2	5	6	6	7	7	8	9	9	9	10	10

maximum length of a shift, and the effect of LNS walking, block walking or shift walking.

In order to conduct our experiments, we fix all parameters except one and see the effect of changing one parameter only.

Changing Block Swapping Maximum Size. By changing the maximum size of the block built in the block swapping phase, we can influence the complexity of the underlying search and we can influence the total number of blocks. The bigger the blocks, the fewer they are and the smaller the search will be.

With small blocks, we can better explore the effect of rearranging the end of the sequence using block swapping. Thus we find easy solutions more quickly. However, due to the depth-first search method employed, the search is concentrated at the end of the sequence. Thus if a non-optimal group of cars is located early in the sequence, it is likely it will not be touched by the block swapping routine as the maximum block size is small. Thus we will find easy solutions faster and may not find the hard solutions at all.

On the other hand, if the maximum block size is big, then we add diversification at the expense of intensification; more tries will be unsuccessful, thus slowing down the complete solver. For example, the number of solutions found around 200s goes down as the maximum size of the blocks increases. Furthermore, the number of solutions found after fifty seconds slowly decreases from four to two as the maximum block size grows from four to twenty-two.

Changing the Fail Limit. By changing the fail limit used in the small searches of each loop, we explore the trade off between searching more and searching more often. Our experience tell us that above a given threshold which gives poor results, the performance of the LNS system degrades rapidly as we allow more breadth in the inner search loops.

We see a peak at around eighty fails per inner search. There is also a definite trend indicating that having too small a search limit is better than having a too large one, which is consistent with our expectations. Also interesting is the fact that a larger fail limit results in more solutions earlier, but is overtaken by

Table 6. Modifying the fail limit (FL)

FL	50s	100s	150s	200s	250s	300s	350s	400s	450s	500s	550s	600s
20	3	4	5	7	7	8	9	10	10	10	10	10
30	1	5	6	7	9	9	9	9	9	9	9	9
40	2	5	8	8	9	9	9	9	10	10	10	10
50	4	5	6	7	7	8	9	9	9	9	9	9
60	4	6	7	8	8	9	9	9	9	9	9	9
70	3	6	7	7	8	8	8	8	8	9	9	10
80	3	6	7	9	10	11	11	11	12	12	12	12
90	3	4	5	6	7	7	8	8	9	10	10	10
100	3	6	7	7	9	9	9	9	9	9	9	9
110	1	6	7	7	8	8	8	8	8	8	8	8
120	3	5	6	7	7	8	8	9	9	9	9	9
130	3	6	8	9	9	9	9	9	9	9	9	9

smaller limit after around three minutes of search. Thus, it would appear that a larger fail limit can be efficient on easier problems, by tends to be a burden on more difficult ones.

Changing the Sequence Shifting. By modifying the maximum length of a shift, we can quite radically change the structure of the problem solved. Indeed, the space filled by the shift is solved rather naively by the default search goal. Thus a longer shift allows a fast recycling of a problematic zone by shifting it out of the sequence, at the expense of creating long badly optimized tails. A shorter shift sequence implies less freedom in escaping a local minima while keeping quite optimized tails.

Changing the maximum length of a shift is hard to interpret. There is a specific improvement around thirty where we can find solutions for all twelve problems. But we lack statistical results to offer a definitive conclusion.

Table 7. Modifying the shift length (MSL)

MSL	50s	100s	150s	200s	250s	300s	350s	400s	450s	500s	550s	600s
05	3	6	8	9	9	9	10	10	10	10	10	10
10	3	5	8	9	9	9	9	9	10	10	10	10
15	3	6	7	7	7	7	8	8	8	8	8	9
20	3	5	6	7	7	7	7	8	10	10	10	11
25	4	6	6	6	7	7	9	9	9	9	9	9
30	3	6	7	9	10	11	11	11	12	12	12	12
35	3	7	8	8	9	9	9	10	10	10	10	10
40	4	7	7	8	8	9	9	9	9	9	9	10
45	3	7	8	8	9	9	9	9	9	9	9	9
50	4	5	6	8	8	9	9	9	9	10	10	10

Table 8. Modifying the LNS walking parameter (LNSW)

LNSW	50s	100s	150s	200s	250s	300s	350s	400s	450s	500s	550s	600s
Allowed	3	6	7	9	10	11	11	11	12	12	12	12
Forbidden	0	0	0	0	0	0	0	0	0	0	0	0

Table 9. Modifying the block walking parameter (BW)

BW	50s	100s	150s	200s	250s	300s	350s	400s	450s	500s	550s	600s
Allowed	3	6	7	9	10	11	11	11	12	12	12	12
Forbidden	3	6	7	8	8	8	10	10	10	10	10	10

Table 10. Modifying the shift walking parameter (SW)

SW	50s	100s	150s	200s	250s	300s	350s	400s	450s	500s	550s	600s
Allowed	3	6	6	7	8	8	8	8	8	8	8	9
Forbidden	3	6	7	9	10	11	11	11	12	12	12	12

Forbidding LNS Walking. We can make the same tests with LNS and walking meta-heuristics. These results are shown in table 8.

LNS walking is a mandatory improvement to the LNS solver. Without it, our solver is not able to find any solution to the twelve problems. This is quite different from our experience with other problems as this parameter was not so effective. This is probably due to structure of the cost function: in car sequencing problems, there are large numbers of neighboring solutions with equal cost.

Forbidding Block Walking. As we have seen in the basic Large Neighborhood Search approach, walking in the LNS part is a key feature with regard to the robustness of the search. We can wonder if this is the case with block swapping, because if walking allows the solver to escape plateau areas, it also consumes time and walking too much may in the end consume too much time and degrade search performance. These results are shown in table 9.

Walking is a plus for the block swapping routine as it allows the solver to find more solutions more quickly. In particular, we are able to find all twelve solutions with it while we find only ten of them without it.

Allowing Shift Walking. We can make the same tests with shift and walking meta-heuristics. These results are shown in table 9.

Strangely enough, allowing shift walking does not improve the results of our solver and even degrades its performance. We do not have any clear explanation at this time. We just report the results.

7 Conclusion and Future Work

The main contribution of this article is the different usage of the portfolio of algorithms framework. Instead of combining multiple competing search techniques to achieve robustness, we combine a master search technique (LNS) with other algorithms whose purpose is not to solve the problem, but to diversify and create opportunities for the master search. This approach means that several simple searches can be used instead of one complicated search, leading to a more modular and easily understood system. We also investigated many possible improvements and modifications to the base LNS architecture and report on them, even if most of them were not successful.

Our method works well on the car sequencing problem, especially harder instances. We have achieved good results by being able to solve all twelve hard instances. In the future, we hope to compare against the Comet solver itself when it becomes available.

examples in order to have real statistics on the performance of our solver.

In the future, we would like to further study our model based on insertion of stalls vs. the more standard violation-based model. Good algorithms for one model may not carry over into the other, and the full consequences of the modeling differences warrant further analysis and understanding.

Finally, we would like to examine the effect of the introduction of problem-specific heuristics into the LNS search tree, such as those explored in [17]. These heuristics are reported to work well, even using a randomized backtracking search, and such a comparison merits attention.

We would like to thank our referees for taking the the time to provide excellent feedback and critical comment.

References

[1] Parrello, B., Kabat, W., Wos, L.: Job-shop scheduling using automated reasoning: a case study of the car-sequencing problem. Journal of Automated Reasoning **2** (1986) 1–42 225

[2] Dincbas, M., Simonis, H., Hentenryck, P. V.: Solving the car-sequencing problem in constraint logic programming. In Kodratoff, Y., ed.: Proceedings ECAI-88. (1988) 290–295 225

[3] Hentenryck, P. V., Simonis, H., Dincbas, M.: Constraint satisfaction using constraint logic programming. Artificial Intelligence **58** (1992) 113–159 225

[4] Smith, B.: Succeed-first or fail-first: A case study in variable and value ordering heuristics. In: Proceedings of PACT'97. (1997) 321–330 (Presented at the ILOG Solver and ILOG Scheduler 2nd International Users' Conference, Paris, July 1996) 225

[5] Regin, J. C., Puget, J. F.: A filtering algorithm for global sequencing constraints. In Smolka, G., ed.: Principles and Practice of Constraint Programming - CP97, Springer-Verlag (1997) 32–46 LNCS 1330 225

[6] Gent, I.: Two results on car sequencing problems. Technical Report APES-02-1998, University of St. Andrews (1998) 225

[7] Michel, L., Hentenryck, P. V.: A constraint-based architecture for local search. In: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM Press (2002) 83–100 225, 226, 230, 231
[8] Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In Maher, M., Puget, J. F., eds.: Proceeding of CP '98, Springer-Verlag (1998) 417–431 225, 227
[9] Bent, R., Hentenryck, P. V.: A two-stage hybrid local search for the vehicle routing problem with time windows. Technical Report CS-01-06, Brown University (2001) 225
[10] Applegate, D., Cook, W.: A computational study of the job-shop scheduling problem. ORSA Journal on Computing **3** (1991) 149–156 225, 227
[11] J. Adams, E. B., Zawack, D.: The shifting bottleneck procedure for job shop scheduling. Management Science **34** (1988) 391–401 225
[12] Caseau, Y., Laburthe, F.: Effective forget-and-extend heuristics for scheduling problems. In: Proceedings of the First International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'99). (1999) 225
[13] Palpant, M., Artigues, C., Michelon, P.: Solving the resource-constrained project scheduling problem by integrating exact resolution and local search. In: 8th International Workshop on Project Management and Scheduling PMS 2002. (2002) 289–292 225
[14] Le Pape, C., Perron, L., Régin, J. C., Shaw, P.: Robust and parallel solving of a network design problem. In Hentenryck, P. V., ed.: Proceedings of CP 2002, Ithaca, NY, USA (2002) 633–648 225
[15] Palpant, M., Artigues, C., Michelon, P.: A heuristic for solving the frequency assignment problem. In: XI Latin-Iberian American Congress of Operations Research (CLAIO). (2002) 225
[16] Gomes, C. P., Selman, B.: Algorithm Portfolio Design: Theory vs. Practice. In: Proceedings of the Thirteenth Conference On Uncertainty in Artificial Intelligence (UAI-97), New Providence, Morgan Kaufmann (1997) 225
[17] Gottlieb, J., Puchta, M., Solnon, C.: A study of greedy, local search and ant colony optimization approaches for car sequencing problems. In: Applications of evolutionary computing (EvoCOP 2003), Springer Verlag (2003) 246–257 LNCS 2611 226, 238
[18] Chabrier, A., Danna, E., Le Pape, C., Perron, L.: Solving a network design problem. To appear in Annals of Operations Research, Special Issue following CP-AI-OR'2002 (2003) 227
[19] Perron, L.: Fast restart policies and large neighborhood search. In: Proceedings of CPAIOR 2003. (2003) 227, 229
[20] Beck, J. C., Perron, L.: Discrepancy-Bounded Depth First Search. In: Proceedings of CP-AI-OR 00. (2000) 229