Principles for the Design of Large Neighborhood Search [*]

Tom Carchrae (t.carchrae@4c.ucc.ie)
Cork Constraint Computation Centre, University College Cork, Ireland

J. Christopher Beck (jcb@mie.utoronto.ca)
Department of Mechanical & Industrial Engineering, University of Toronto

Abstract. Constraint programming (CP) has proved to be a very effective tool for solving complex optimization problems. However, the practice of applying CP to real world problems remains an art which requires a great deal of expertise. In this paper, we show how to reduce the required expertise using adaptive techniques to create algorithms that adjust their behavior to suit the problem instance being solved. To achieve this, we use the framework of Large Neighborhood Search, where we present three design principles: cost based neighborhood heuristics, growing neighborhood sizes, and the application of learning algorithms to combine portfolios of neighborhood heuristics. Our results show that the application of these principles gives strong performance on a challenging set of job shop scheduling problems. More importantly, we are able to achieve robust solving performance across problem sets and time limits.

Keywords: constraint programming, large neighborhood search, optimization, machine learning, scheduling

1. Introduction

Constraint programming (CP) has proved to be a very effective tool for solving complex optimization problems. However, the practice of applying CP to real world problems remains an art which requires a great deal of expertise (Le Pape et al., 2002). In this paper, we show how to reduce the required expertise using adaptive techniques to create algorithms that adjust their behavior to suit the problem instance being solved.

A great deal of research over the last decade has produced CP algorithms which are able to compete well on many small to medium sized problems. The core strengths of CP are strong inference methods, which are often able to reduce the search space and quickly detect infeasibility, and good heuristics that guide search toward areas of the search space that are likely to contain solutions. CP typically addresses optimization using branch and bound, where the cost is represented by an objective

[*] This material is based upon works supported by the Science Foundation Ireland under Grant No. 00/PI.1/C075, the Embark Initiative of the Irish Research Council of Science Engineering and Technology under Grant PD2002/21, and ILOG, S.A.

variable. By placing a constraint on the objective variable it is required that new solutions have a lower cost than the best found so far.

These methods alone, however, are often ineffective at optimizing larger problem instances without significant tuning and effort on the part of an expert in constraint programming. The sheer complexity of many problems renders even the most advanced CP methods useless unless a careful problem decomposition strategy is applied.

In this paper, we address the issue of automatically decomposing the problem into smaller sub-problems. We achieve this through a combination of large neighborhood search (Shaw, 1998) and algorithm control methods (Carchrae and Beck, 2005) which adapt runtime performance to the best performing methods. We present three general design principles for large neighborhood search and then show how these can be applied using a challenging set of scheduling problems. Our results show strong performance, however, more impressive is the ability of the control methods to adapt to suit the problem being solved. Robust solving performance is achieved across two problem sets and all time limits indicating that it is possible to reduce the human expertise required in applying these methods.

1.1. Large Neighborhood Search

Large neighborhood search (LNS) (Shaw, 1998) is a framework which combines the search power of CP with the scaling performance of local search. As in local search, we modify an existing solution to the problem. However, instead of making small changes to a solution, as is typical with local search move operators, we select a subset of variables from the problem. Once variables are selected, we unassign them, lock the remaining variables to take the values assigned in the current solution, and then search for a better solution by changing only the unassigned variables.

For example, a typical local search move for a scheduling problem would be to swap the order of two operations, eg $A_i < A_j \rightarrow A_j < A_i$. In the LNS framework, we remove the assignment to a subset of variables and then search for an improved assignment. Since CP can efficiently deal with more than two variables, many variables are typically selected when searching for improvements. In local search, a neighborhood is explicitly defined by a set of possible moves. In LNS, the neighborhood is implicitly defined by the possible reassignments to the selected variables. Methods which select sets of variables to search are called neighborhood heuristics.

The central idea of LNS is very simple and presented in Algorithm 1. Starting with a solution to the entire problem, we select a subset of

Algorithm 1 Large Neighborhood Search

```
1: N := variables in problem
2: $Solution$ := Create initial solution
3: while Not Optimal and Time Left do
4:    Using neighborhood heuristic choose $S \subseteq N$
5:    Unassign $S$ in $Solution$
6:    Lock the assignment in $Solution$ for all $r \in N$ where $r \notin S$
7:    if Search finds improvement then
8:       $Solution$ := Update solution
9:    else
10:      $Solution$ := Restore solution
11:   end if
12: end while
```

variables (the neighborhood) and perform a search which only changes the values of the selected variables; all other variables keep their assignments. By focusing efforts on improving only a part of the solution, we restrict the size of the search space and intensify search to improve the current solution. The key benefit from a CP perspective is that we can exploit the strong CP propagation techniques while avoiding the weaknesses inherent in exploring a single search tree in a large problem space (Gomes et al., 1998).

The choice of variables to search, the neighborhood, is crucial to the performance of LNS. We wish to select variables which are likely to reduce the cost of the current solution but we are also concerned with the number of variables in each search: fewer variables results in a quicker search, but also reduces the likelihood than an improved solution exists within the search sub-space that is explored.

1.2. Design Principles for Large Neighborhood Search

In this paper, we investigate three domain independent principles to create large neighborhood search implementations.

Cost Based Neighborhood We wish to focus search effort on the part of the problem which is contributing the most to the cost. Thus, we suggest that the design of a neighborhood heuristic should select the variables to search which are most directly affecting the cost of the current solution. The intuition is that changing these assignments provides the most promising opportunity to find a better solution.

Growing Neighborhood Previous work (Perron et al., 2004; Godard et al., 2005) has attempted to determine the best number of variables to select in a neighborhood. Instead, we propose that neighborhood heuristics should be designed so that they slowly grow. As we continue to search without finding any improvement, the neighborhood heuristic increases the number of variables to search until eventually, we consider the entire problem. If no search limits are used, this makes LNS a complete method.

Combinations and Learning It is unlikely that a single neighborhood heuristic will work well in all cases. Even on the same problem instance, as search progresses, the performance of each neighborhood may change. Thus, we suggest the principle of combinations of neighborhoods. Based on our previous work (Carchrae and Beck, 2005) in applying learning to select the best search algorithm, we are interested in investigating the use of learning algorithms to determine the most promising neighborhood heuristics based on experience.

1.3. Paper Outline

We present several contributions in this paper. In Section 2 we review previous work on large neighborhood search. In Section 3 we formally describe several neighborhood heuristics for scheduling including a new neighborhood heuristic based on resource load. We present the principle of a cost based neighborhood in Section 4. In Section 5 we introduce the idea of growing neighborhoods as an important design principle for neighborhood heuristics. We then present two learning methods to combine neighborhood heuristics in Section 6. In Section 7 we evaluate these concepts on two challenging sizes of job shop scheduling problems and discuss the results in Section 8. We then discuss promising future work in Section 9 and conclude with Section 10.

2. Previous Work

The initial work in the constraint programming community using the term LNS (Shaw, 1998) applied the idea to the problem of vehicle routing with time-windows. However, other work has since been considered as an instance of LNS. In (Nuijten and Le Pape, 1998) they randomly select neighborhoods and are able to boost performance in the domain of job shop scheduling. Since then, several papers have appeared, where different neighborhood heuristics have been developed for applications such as network design (Chabrier et al., 2004), earliness/tardiness scheduling (Danna and Perron, 2003) and most recently car assembly sequencing (Perron and Shaw, 2004). The work of Propagation Guided-LNS (Perron et al., 2004), and RINS (Danna and Perron, 2003) is particularly interesting as they develop methods which do not require domain knowledge to build neighborhoods.

In (Perron and Shaw, 2004) they apply a very similar learning strategy to our previous algorithm control work (Carchrae and Beck, 2005). However, this learning strategy is only applied to the search algorithm used to explore the neighborhood. As far as we are aware, no research has been carried out on applying learning to the choice of neighborhood heuristic.

It is interesting to note that all prior work on large neighborhood search has made a extensive use of randomization. It seems that randomly choosing neighborhoods remains a very useful approach to selecting a neighborhood, certainly it seems to be a useful method when interleaving neighborhoods. We do not make extensive use of randomization in this paper however. The only randomized method we explore is the pure random neighborhood heuristic.

3. LNS Applied to Scheduling

In this paper, we use the problem of makespan minimization in job shop scheduling to investigate our design principles. The job shop scheduling (JSP) problem involves scheduling n jobs across m machines. Each job j consists of m ordered operations, such that each operation is scheduled one after another. For each job, every operation requires a unique machine for a specified duration, and the sequence of these requirements differs among jobs. There are two types of constraints we must satisfy: precedence constraints between operations in a job, and resource constraints which specify that two operations may not overlap if they share the same resource. We look at the objective of minimizing makespan: the total time required from the start of the earliest scheduled operation to the finish of the latest scheduled operation. Minimizing makespan in the JSP is NP-hard (Garey and Johnson, 1979).

A solution to a JSP can be represented by a total ordering of operations on each resource. A convenient data structure to represent this type of solution is a linked list. We let $o_i = head(r_j)$ be the first operation o_i scheduled on resource r_j, and $o_j = next(o_i)$ be the operation that immediately follows o_i (on the same resource) in the current solution. This information is sufficient to compute the start time range of each operation as well as the overall makespan of the schedule (Dechter et al., 1991). For convenience, we define $est(o_i)$ and $lst(o_i)$

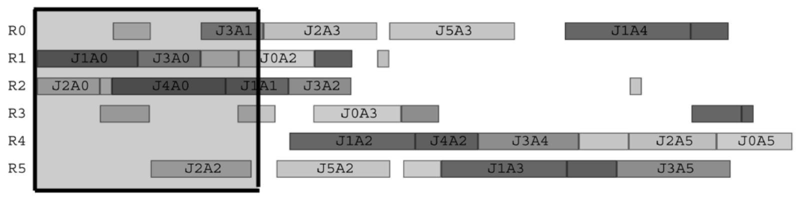

Figure 1. The time window neighborhood heuristic selects all operations whose earliest start time is in the time window interval.

which represent the earliest and latest start times for an operation o_i that are allowed by a solution.

This solution representation enables a straightforward implementation of LNS. To remove the assignment of a particular operation o_i, we simply remove any information that refers to o_i. If there is an operation scheduled before o_i on the resource, eg, $o_i = next(o_j)$ then we remove the relation $o_i = next(o_j)$ from the solution. Likewise, if there is a following operation o_k on the resource, eg, $o_k = next(o_i)$ then we also remove this relation. If an operation has no preceding operation, we also remove $o_i = head(r_j)$ from the solution. Once an operation has been removed in this manner, we can search to see if there exists a better place for o_i in the current solution. Let O be the set of all operations in the schedule. We define a function $unschedule(O_{remove}, Sol)$ which performs this process for each operation $o_i \in O_{remove}$ on the solution Sol. The task then for the neighborhood heuristic is to determine an effective choice for $O_{remove} \subseteq O$ given a solution Sol.

3.1. Time Window Neighborhood

The time window neighborhood heuristic selects all the operations which are scheduled in a particular time interval. We refer to the interval as a time window. Each time the neighborhood heuristic is called, the time window slides forward across the schedule to select some new operations. We overlap time windows, so that each selection includes some of the previously selected operations, but also contains new operations. In this manner, we locally optimize all the operations within each time window. An example the time window neighborhood heuristic is shown in Figure 1.

More formally, let the earliest time in the schedule be $T_{min} = min(est(o_i)|o_i \in O)$ and the latest time be $T_{max} = max(est(o_i)|o_i \in O)$. This gives us the total time range of the current solution, $Range =$

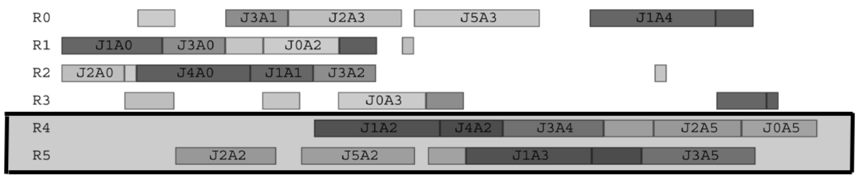

Figure 2. The resource load neighborhood heuristic selects all operations on resources in the resource window.

$T_{max} - T_{min}$. We divide *Range* into W_t windows, thus each window has the interval length $I = \lceil Range/W_t \rceil$. Each time window is defined by t_{min} and t_{max} such that $I = t_{max} - t_{min}$. The windows are considered by increasing time intervals, starting with $t_{min} = T_{min}$. We select all operations, $o_i \in O$ whose minimum start time $est(o_i)$ is in the interval t_{min} to t_{max}. For the next interval, we shift the window by a fraction S_t of the window interval length I, which gives us $shift_t = S_t \times I$. Thus, after searching the first interval, we add $shift_t$ to both t_{min} and t_{max}. We stop when $t_{max} >= T_{max}$.

3.2. Resource Load Neighborhood

In this neighborhood heuristic, we choose a subset of resources and select all the operations which are scheduled on those resources. Resources are sorted by resource load, the intuition being that the most loaded resources will have a greater chance of conflicts when being scheduled. In a manner similar to the time window neighborhood, we select a 'window' of resources from the ordered list of resources, starting with the most loaded resource and working towards the least loaded resource. We show an example of this in Figure 2. We note that the idea of a neighborhood heuristic which uses a window of selected resources, based on load, is a novel idea and to our knowledge has not been presented before in the context of LNS.

Let R be a sorted list of N resources, where R_1 is the most loaded resource, and R_N is the least loaded resource. Given a ratio S_R of resources to select, we determine the number $r = S_R \times N$ of resources to select for each window. Let r_{min} be the starting resource index of the window, and r_{max} be the ending resource index of the window. Hence in a neighborhood, we select all operations on resources $R_{r_{min}}$ to $R_{r_{max}}$ from the sorted list of resources R. The first neighborhood sets $r_{min} = 1$ and $r_{max} = r$. For successive neighborhoods we increase r_{min} and r_{max} by $shift_r$. We stop when $r_{max} = N$.

3.3. Random Neighborhood

The random neighborhood is perhaps the simplest to describe. We randomly choose a subset of operations and remove them. Formally, given a ratio S_R of operations to select from the set of all operations O, we compute $S = S_R \times |O|$. This neighborhood never halts, as we can continue to draw S operations randomly for as long as we wish.

4. Cost Based Neighborhood

We now introduce the idea of a cost-based neighborhood. The central idea is that we rank variables based on their impact on the objective in the current solution. The first neighborhood only consists of the most highly ranked variables. Successive neighborhoods include variables of a lower rank at each step, but keep the previously selected variables as well. Hence, a cost based neighborhood starts with the variables with the highest cost impact and successively adds less important variables at each step. The challenge then, is to determine an effective ranking strategy which focuses search on the variables that are contributing the most to the objective.

$$CostBasedNeighborhood_i = \{o_j | o_j \in O, Rank_{o_j} \leq i\} \tag{1}$$

We note that when CP uses branch and bound to optimize, the objective variable and associated constraints are treated in the same manner as any other constraint. That is, search is only concerned with satisfying these constraints. This means that the search heuristics are guided towards feasible solutions rather than the best solutions. In this context a feasible solution will indeed be an improved solution, however we would prefer that search is directed towards the best possible solution rather than just an improved one. The cost based neighborhood heuristic is a method for directing search on variables which have the highest impact to the objective variable in the current solution. This idea is similar to that of sensitivity analysis in linear programming.

4.1. Cost Based Neighborhood using Operation Criticality

For the task of minimizing makespan in the job shop scheduling problem it is well known that to improve a solution it is necessary to reorder some operations along a critical path. A critical path is defined as a sequence of critical operations that are connected by precedence constraints, either induced by a sequence of operations on a resource in the current solution, or by the problem definition. A critical operation o_i has a slack value of 0, where slack is computed by $slack(o_i) = lst(o_i) - est(o_i)$. In other words, slack represents how much an operation may move in the schedule, with 'critical' denoting an operation which cannot move at all.

We choose our neighborhood to exploit the intuition that re-optimizing critical operations is likely to improve solution quality. The ranking then for the objective of minimizing makespan is based on the slack value of each operation. Operations with less slack are ranked before operations with more slack. By ranking neighborhoods based on the slack we hope that we are choosing parts of the problem in increasing order of importance to the objective function.

The idea of using the critical path to perturb the schedule is standard in many local search approaches as well as in other techniques such as iFlat (Cesta et al., 2000) and iFlatRelax (Michel and Van Hentenryck, 2004). As far as we know it has not been explicitly proposed within an LNS framework.

5. Growing Neighborhoods

The cost based neighborhood has an interesting property: it starts off small, and grows slowly so that it eventually includes all of the variables. This property allows us to avoid fixing a neighborhood size parameter and it also allows LNS to be complete if the entire set of variables are in the neighborhood. We now discuss how the other neighborhood heuristics described previously can be modified so that they also have this property.

For the time window neighborhood, we grow the size of neighborhood by decreasing the number of time windows, W_t. We do this after one full sweep across the scheduling horizon. This has the effect of increasing the number of operations selected until we reach $W_t = 1$ where we select all operations. As the number of windows decreases, the number of operations selected in each neighborhood increases.

The resource load neighborhood is modified in a similar fashion. We grow the neighborhood size by increasing r, the number of selected resources, by 1. This has the effect of increasing the neighborhood until we consider all resources, at which point we select all operations.

In our experiments we did not explore growing when applied to random neighborhoods. However, a policy suggested in (Nuijten and

Le Pape, 1998) is to slowly increase R, the number of randomly chosen operations, if no solutions are found after k iterations.

We note that even though we are growing neighborhoods which allows more flexibility, for performance reasons we still need to determine an initial parameter for each neighborhood.

6. Combining and Learning

In our previous work (Carchrae and Beck, 2005), we had success in applying reinforcement learning to allocate computational resources to scheduling algorithms. It seems reasonable that with several neighborhoods to choose from, we can boost performance by applying these ideas to select a neighborhood heuristic. Indeed, even simply alternating between different neighborhood heuristics appears to be a promising approach (Danna and Perron, 2003; Perron et al., 2004).

We explore two learning methodologies here. The first method, *adaptive runtime*, allocates running time based on the performance, over time, of each neighborhood heuristic. We investigate the use of allocating both a fixed amount of time as well as increasing the amount of time allocated after each iteration. The second method, *adaptive probability*, adjusts the probability of choosing a neighborhood based on past performance. Hence, a neighborhood that has made a big improvement recently will have a higher probability of being selected.

6.1. Adaptive Runtime

The *adaptive runtime* method allocates runtime based on past performance. At each step i in the algorithm, we have t_i seconds to allocate and we run each neighborhood heuristic exactly once. The running time is determined by a weight w_j which represents the proportion of time to allocate to the neighborhood heuristic j. Hence $t_i \times w_j$ gives the allocated run time for neighborhood heuristic j at step i. We note that the sum of weights is always 1. All weights are updated after one step using the formula $w_j := w_j * (1 - \alpha) + p_j * \alpha$, where α is the learning rate and p_j is the normalized performance over time for neighborhood heuristic j in the last iteration. The performance values p_j for all neighborhood heuristics are normalized so that they sum to 1. The learning rate can be a value from 0..1; we use a learning rate of $\alpha = 0.5$.

We explore two variants of this approach. The first *AdaptR-double* doubles the amount of time t_i available at each step, starting from an initial value of 1 second. The second approach *AdaptR-static* has a constant time of $t_i = 10$ seconds. All weights are initialized to equal values. This is exactly the same approach as used in (Carchrae and Beck, 2005).

6.2. Adaptive Probability

The second method, which is presented for the first time in this paper, is *adaptive probability*. Instead of allocating more time to the neighborhood heuristics which perform better, we increase the likelihood that we will select the neighborhood heuristics which have performed better. In this context, the normalized weight w_j represents the probability of selecting neighborhood heuristic j.

We update the weights after each neighborhood heuristic has been run, rather than updating all weights after running all of the neighborhood heuristics. Since the weights are updated independently, we compute probabilities in a different way. For each neighborhood, we record the history of improvements $imp_{i,j}$ made at each step i by neighborhood heuristic j. We compute the weight based on a discounted average of the improvements giving a bias to recent improvements over older improvements. That is, for each improvement $imp_{i,j}$ we apply a discount of $d_{i,j} = 1/(steps_j - (i-1))$ where $steps_j$ is the number of times neighborhood heuristic j has been applied so far. This gives us the discounted improvement $dimp_{i,j} = imp_{i,j} \times d_{i_j}$. This means that the most recent improvement is not discounted at all, but older improvements are increasingly discounted. The weight for a neighborhood then is the normalized mean discounted improvement. Let $davg_j = average(dimp_{i,j} | i \in (1..steps_j))$ and then the weight w_j is $davg_j$ normalized so that all weights sum to 1.

In this paper we refer to this method as *AdaptP*. We also include a method, *RandP*, which operates in the same manner but does not alter the weights, hence it randomly chooses a neighborhood heuristic.

7. Experiments

We now look at how the ideas presented in the previous sections perform in practice. In this section, we describe the problem instances and CP search algorithms that are used, how we choose parameters for the neighborhoods and how we compare performance among different neighborhoods.

7.1. Problem Instances

We evaluate our ideas on two sets of scheduling problems. The first set, 20x20, consists of medium sized scheduling problems with 20 jobs and 20 machines, each instance containing 400 operations. The second set, 40x40, contains larger problem instances with 40 jobs and 40 machines, each instance containing 1600 operations. The operation durations are drawn randomly with uniform probability from the interval [1, 99]. The routing of each job is randomly assigned so that each job has exactly one operation on each machine, but each job requires machines in a random order. Each set consists of 60 problem instances.

7.2. Constraint Programming Algorithms

In these experiments, we search using a constructive search technique employing texture-based heuristics (Beck and Fox, 2000), strong constraint propagation (Nuijten, 1994; Laborie, 2003) and bounded chronological backtracking. The texture-based heuristic identifies a resource and time point with maximum competition among the activities and chooses a pair of unordered activities, branching on the two possible orders. The heuristic is randomized by specifying that the resource and time point is chosen with uniform probability from the top 10% most critical resources and time points. The bound on backtracks follows the pattern of $BT = \{1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 4, 8, 1, 1, 2, 4, 8, 16, \ldots\}$. Formally, $BT_i = \lceil 2^{k-2} \rceil$ where $k = n - (m(m+1)/2) + 1$, $m = \lfloor (\sqrt{8n+1} - 1)/2 \rfloor$ and $n = i + 2$. This was inspired by the optimal, zero-knowledge pattern of (Luby et al., 1993) but is more aggressive in how it increases the bound (Wu and van Beek, 2003).

7.3. Time Slices and Fail Limits

One challenge when comparing different neighborhoods is that they may have very different performance behavior. Some neighborhood heuristics may select small neighborhoods which are very fast to search, but the chance of improvement may be low. On the other hand, a neighborhood heuristic may try to tackle a large neighborhood, but may have a much higher chance of finding an improvement. Therefore the exploration of one neighborhood may expend considerably more computational resources than another. To account for this, we evaluate the neighborhood heuristics by letting them run for an equal amount of time. During this time, they may search many neighborhoods, or perhaps not even be able finish searching even a single neighborhood. Regardless, this time slicing gives us a fair way of comparing neighborhood heuristics, based on an equal amount of processing time.

During the search of a single neighborhood, we limit the total number of backtracks, or fails, which can occur. While the scheduling heuristics in constraint programming are very powerful and manage to avoid many infeasible solutions, they can still get stuck in proving infeasibility in some areas of the search space. It follows then, that if a solution is to be found quickly, it happens with a very small number of fails. In other cases, such as there being no improved solution in a neighborhood, it can typically require a significantly higher number of fails to prove this. This has lead to the practical technique of using a failure limit to avoid expensive searches through infeasible areas. See the work of (Gomes et al., 1998) on randomized restarts for backtracking algorithms for a further explanation of this phenomenon. Since we are interested in finding improved solutions quickly, we place such a limit on every search that is performed.

It is important to be clear about the distinction between time slices, which limit the total amount of time spent applying a particular neighborhood heuristic, and failure limits, which limit the amount of effort spent searching a single neighborhood that a neighborhood heuristic has selected. Time slices allow us to compare different neighborhood heuristics by giving them equal amounts of time. Fail limits allow us to avoid searching unpromising neighborhoods, hoping that we find improved solutions quickly or not all. However, it may be the case that both the time slice and the failure limit are too small to find any improvements in a neighborhood. For this reason, we double both limits if no improved solution is found after ten time slices. Each neighborhood heuristic is then reset so that it restarts searching neighborhoods. We restart since we need to revisit previously explored neighborhoods with these new search limits. The initial time slice is one second and the initial failure limit is 100 fails.

7.4. Parameter Tuning

Recall that several of the neighborhoods require parameters. We must choose a window size for the time window and resource load neighborhood heuristics and also decide how much to overlap these windows. For the random neighborhood, we need to determine an appropriate ratio of operations to select given the problem size. The task of choosing the correct parameters is something we have not addressed completely, however it is an important aspect which cannot be ignored. We discuss more thorough approaches in Section 9.

We ran a selection of 20 different configurations on each problem set for a limited amount of time. For the time window neighborhood, we tried time window W_t values of 2, 4, 6, 8, 10, 12, 14, 16, 18 and

Table I. Performance results for 20x20 problems

NH	Parameter Setting	Probability of Improvement	Average Improvement	Utility (Prob x Avg)
Time Window	2	0.24	10.70	2.57
Random	45%	0.21	8.25	1.73
Resource Load	65%	0.21	8.16	1.71
Cost Based	-	0.15	11.30	1.70
Resource Load	45%	0.15	9.09	1.36
Random	25%	0.16	6.34	1.01
Resource Load	85%	0.10	9.03	0.90
Time Window	4	0.12	7.23	0.87
Resource Load	25%	0.08	6.02	0.48
Random	85%	0.06	7.71	0.46
Time Window	6	0.09	4.25	0.38
Random	65%	0.08	3.79	0.30
Time Window	8	0.08	3.21	0.26
Time Window	10	0.07	3.07	0.21
Time Window	12	0.07	2.90	0.20
Time Window	14	0.06	2.67	0.16
Time Window	16	0.05	2.57	0.13
Time Window	18	0.04	2.27	0.09
Random	5%	0.03	2.21	0.07
Time Window	20	0.03	1.93	0.06
Resource Load	5%	0.02	1.71	0.03

20 and use a window overlap ratio of $S_t = 50\%$. The resource load heuristic requires a parameter S_R to specify the ratio of resources to select in a subproblem. We tried the following values for S_R : 5%, 25%, 45%, 65% and 85%. We use an overlap value of $shift_r = 1$, indicating that 1 resource will be added and 1 resource will be removed as we visit subsequent neighborhoods. Finally, the random neighborhood heuristic also requires a parameter S_R specifying how many operations to select. We tried the following values: 5%, 25%, 45%, 65% and 85%. We also include the cost based neighborhood in these experiments which takes no parameters.

The following procedure was used to evaluate the configurations. Given the current best solution Sol, we give each neighborhood heuristic one second of processing time to improve Sol. Once we have tried all of the neighborhood heuristics, we update Sol with the best solution found. In the case that more than one neighborhood finds an equally

Table II. Performance results for 40x40 problems

NH	Parameter Setting	Probability of Improvement	Average Improvement	Utility (Prob x Avg)
Cost Based	-	0.45	6.46	2.91
Time Window	8	0.13	3.05	0.40
Time Window	10	0.12	3.05	0.37
Time Window	12	0.11	3.17	0.35
Resource Load	25%	0.11	2.90	0.32
Time Window	6	0.11	2.66	0.29
Time Window	14	0.09	3.02	0.27
Time Window	4	0.08	2.29	0.18
Time Window	16	0.07	2.38	0.17
Random	25%	0.07	2.32	0.16
Time Window	18	0.06	2.35	0.14
Resource Load	45%	0.07	1.71	0.12
Time Window	20	0.05	2.18	0.11
Resource Load	5%	0.05	1.94	0.10
Time Window	2	0.04	1.63	0.07
Resource Load	65%	0.03	1.36	0.04
Random	5%	0.02	1.43	0.03
Random	65%	0.02	1.00	0.02

good improved solution, then we choose the best one randomly. We then run each neighborhood heuristic again on the new solution and repeat this process.

This method was chosen since it avoids the case where a single neighborhood may get stuck, unable to improve *Sol*. We do not claim this method to be the best possible way of choosing parameters, however it helped to significantly improve the performance of neighborhoods requiring parameters. For a more rigorous approach to selecting parameters, we refer the reader to (Birrattari et al., 2002).

In Table I and II we show the results of running the procedure described above on 60 problem instances for 30 minutes on each instance. The average improvement of makespan (in time units) is shown along with the probability of each neighborhood making an improvement. The probability is computed simply by the number of times an improvement was found divided by the number of times that neighborhood heuristic was tried. We multiply the probability of improvement by the average improvement value to compute the utility of each neighborhood. We then select the parameter with the highest utility for each neighborhood.

As we can see, the results change depending on the problem set. For the medium problem set, the winners are time windows with 2 windows, resource load with 65% resources and random with 45% of the operations, and then finally, the cost based neighborhood heuristic. We note that even though the cost based neighborhood heuristic has the highest average improvement, it is less likely to make improvements and hence has a lower utility. For the larger problems we see a different picture. Now the cost based neighborhood heuristic is the best performer, both in terms of average improvement and probability of improvement. Next is time window with 8, 10, and 12 windows. We choose only the best parameter setting, 8. The next best type of neighborhood is the resource load neighborhood with a setting of 25% of resources selected. And finally we have random, also with a setting of 25%.

7.5. Mean Relative Error

Our primary evaluation criteria is mean relative error (MRE), a measure of the mean extent to which an algorithm finds solutions worse than the best solutions found during our experiments. MRE is defined as follows:

$$MRE(a,K) = \frac{1}{|K|} \sum_{k \in K} \frac{c(a,k) - c^*(k)}{c^*(k)} \tag{2}$$

where K is a set of problem instances, $c(a,k)$ is the lowest cost solution found by algorithm a on problem instance k, and $c^*(k)$ is the lowest cost solution found during our experiments for problem instance k.

7.6. Pure Neighborhood Experiments

Having determined good parameter settings for each neighborhood heuristic on each problem set, we ran each of the neighborhood heuristics independently. For the medium sized 20x20 problems, we used a time limit of 10 minutes. For the large 40x40 problems, we used a time limit of 20 minutes. We show the results in Figure 3 and Figure 4. To highlight the improvement of using LNS, we also include a standard CP search algorithm, *texture with bounded BTs* which performs far worse than any LNS method. On the medium problem set, we see that the cost based neighborhood heuristics does not perform very well at all. It is quickly outpaced by all of the other neighborhood heuristics,

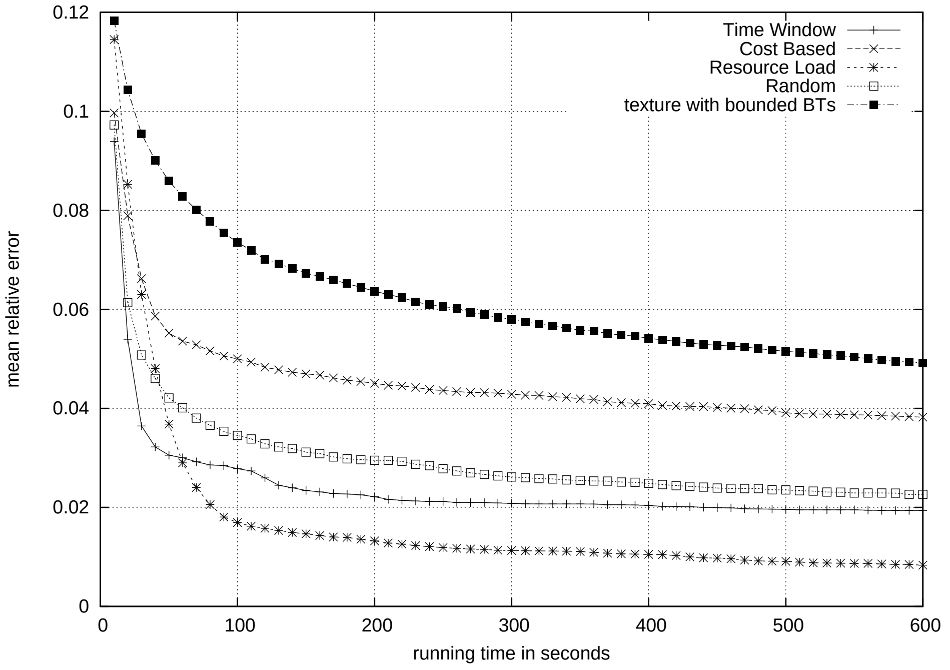

Figure 3. 20x20 neighborhoods run independently

the best one being the resource load neighborhood. However, on the large problems the situation is different. The cost based neighborhood heuristics has a strong lead at the beginning of search, but it plateaus and it overtaken after 400 seconds. In the end, the 'slow and steady' performance of the time window neighborhood heuristics dominates the larger problems, although the random neighborhood heuristic also has strong performance.

7.7. Combined Neighborhood Experiments

It would seem, given the variance in performance, that combining neighborhoods should increase performance. We show the results of applying the combination and learning methods described in Section 6 in Figures 5 and 6. On the medium sized problems, we see that the performance of static adaptive runtime (*AdaptR-static*), adaptive probability(*AdaptP*), and randomly alternating (*RandP*) is competitive with the best neighborhood. Adaptive runtime with a doubling time limit (*AdaptR-double*) does not perform as well, but the performance is not far behind the other methods. While the learning methods have not increased the performance on this problem set, they were able to perform as well as the best technique. On the larger problems, we see

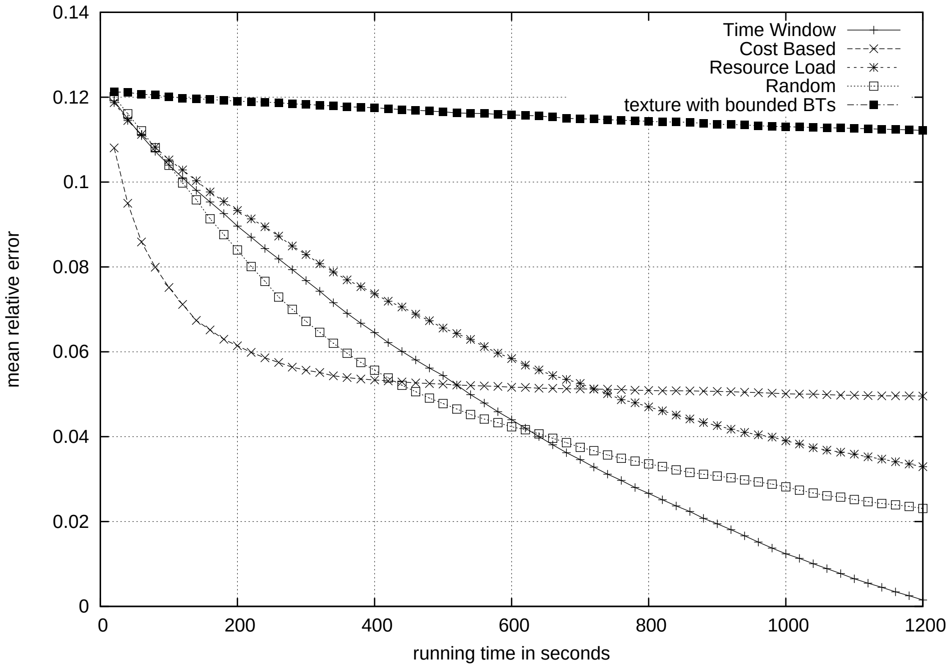

Figure 4. 40x40 neighborhoods run independently

that the best learning method, *AdaptP*, has good performance. In the beginning, it is slightly worse than the cost based neighborhood heuristic, however, it does not suffer the same plateau effect. Later in search, this method is only slightly worse than the time window neighborhood heuristic. The other learning methods do not perform much worse than *AdaptP* with the exception of *AdaptR-double*.

8. Discussion

8.1. Pure Neighborhoods

The most striking thing we notice about the results in Figure 3 and 4 is that all of the neighborhood heuristics outperform a state-of-the art CP algorithm, *texture with bounded backtracking*. The performance improvement on the 40x40 problems is even more dramatic than the 20x20 problems. The CP algorithm is barely able to improve the solution at all on the larger problem instances. This type of result has lead people to believe that CP methods do not scale, and indeed, it seems that in the case of optimization, a single tree search is not an effective method on large problems. However, when we apply large neighborhood search, even randomly choosing neighborhoods appears to be quite effective for optimization. This indicates that CP techniques may well be useful on large problems, however, the approach of a single tree search does not appear promising.

When we compare the neighborhood heuristics, the best performing neighborhood changes depending on the problem set, and indeed, even depending on the amount of time allowed for processing. On the 20x20 problems, the resource load neighborhood heuristic outperforms all of the other neighborhood heuristics, except at the start when the time window neighborhood heuristic is stronger. Although it is not shown on the 20x20 graph, the cost based neighborhood is actually the strongest performer at the very beginning, however it is quickly overtaken. This behavior is repeated on the 40x40 problems, although in that case the cost based neighborhood outperforms the other neighborhood heuristics for a longer time.

In our initial experiment in Section 7.4, we showed that the cost based neighborhood consistently produced the best average improvement of all neighborhoods. However, it was not always able to make an improvement, as the performance on the 20x20 sized problems shows. The reason why lies in the fact that, unlike other neighborhood heuristics, the cost based neighborhood persistently tries to improve the same operations. If it is unable to make an improvement, it adds more operations to the neighborhood. If it is the case that the assignment of operations with a high slack, and thus a low ranking, must be changed to improve the solution, then the cost based neighborhood will only attempt to change these when the neighborhood has grown to a large size. If this happens, then the search space complexity of the neighborhood may prohibit finding an improved solution within the search limits. This type of effect is likely for any neighborhood heuristic which focuses solely on the cost.

We can say that the cost based neighborhood heuristic is intensifying, as it focuses search effort on high impact variables. It is interesting to compare against the other neighborhood heuristics, which are inherently diversifying in their nature and do not suffer from this malady. However, they are also slower to make improvements on the 40x40 problem set.

Regardless, these results are very promising. We have shown strong performance when applying the neighborhood heuristics individually. The difference in performance depending on the problem set appears to suggest that a hybrid approach may be able to outperform a single neighborhood. We discuss this in more detail in the next section.

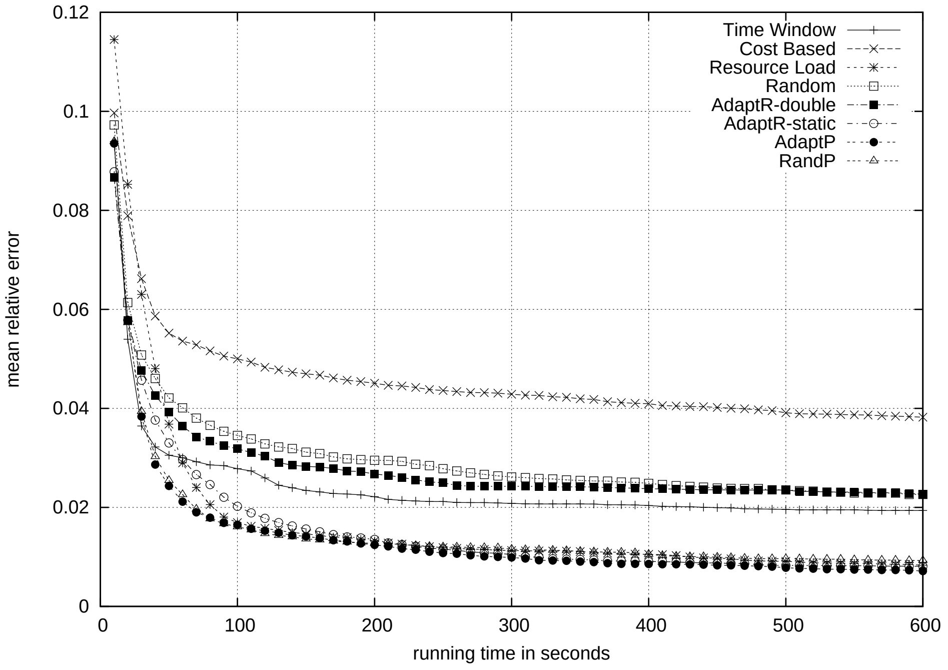

Figure 5. 20x20 combined neighborhoods

Table III. 20x20 percent of best solutions

Neighborhood	Time (seconds)					
Heuristic	100	200	300	400	500	600
Time Window	6.7 %	1.7 %	3.3 %	3.3 %	1.7 %	1.7 %
Cost Based	1.7 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %
Resource Load	25.0 %	26.7 %	20.0 %	20.0 %	23.3 %	25.0 %
Random	1.7 %	1.7 %	1.7 %	3.3 %	1.7 %	1.7 %
AdaptR-double	1.7 %	1.7 %	0.0 %	0.0 %	0.0 %	0.0 %
AdaptR-static	8.3 %	13.3 %	20.0 %	18.3 %	18.3 %	18.3 %
AdaptP	30.0 %	35.0 %	40.0 %	35.0 %	38.3 %	40.0 %
RandP	31.7 %	25.0 %	20.0 %	21.7 %	20.0 %	15.0 %

8.2. Combination and Learning

In combining different neighborhood heuristics we hope to achieve two things: robustness and synergy. Robustness refers to the ability to achieve strong performance, no matter what problem set or processing time is available. Robust solving is a shortcoming of many optimization

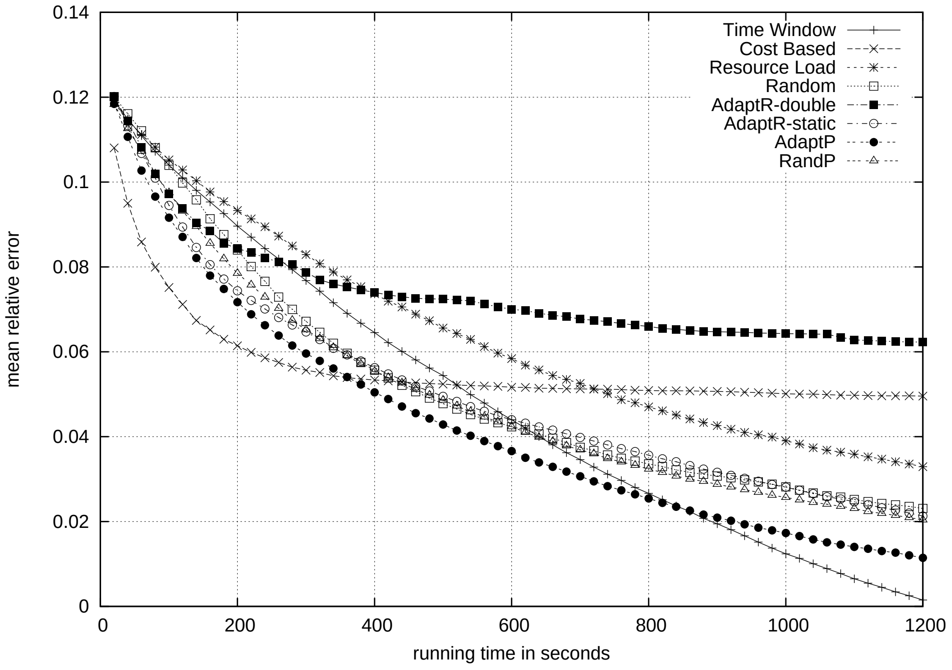

Figure 6. 40x40 combined neighborhoods

Table IV. 40x40 Percentage of Best Solutions Found at Different Time Periods

Neighborhood	Time (seconds)					
Heuristic	200	400	600	800	1000	1200
Time Window	0.0 %	5.0 %	18.3 %	38.3 %	68.3 %	80.0 %
Cost Based	71.7 %	26.7 %	8.3 %	0.0 %	0.0 %	0.0 %
Resource Load	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %
Random	0.0 %	13.3 %	13.3 %	5.0 %	1.7 %	0.0 %
AdaptR-double	3.3 %	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %
AdaptR-static	6.7 %	13.3 %	5.0 %	3.3 %	0.0 %	0.0 %
AdaptP	18.3 %	35.0 %	45.0 %	45.0 %	28.3 %	20.0 %
RandP	1.7 %	8.3 %	11.7 %	11.7 %	3.3 %	3.3 %

techniques, and to achieve robust solving implies that we can reduce the requirement for human effort and expertise in applying these methods. For instance, with a number of pure neighborhood heuristics to choose from, each of which performs differently depending on the time available and the problem set, we would need human expertise and experimentation to find the best one. On the other hand, a robust technique is able to perform close to best across all problem sets and time limits. Synergy refers to a combination of neighborhood heuristics outperforming a single neighborhood heuristic. In our previous work on algorithm control, we experienced a synergistic effect when combining different search algorithms.

On the 20x20 problem set, we clearly achieve robustness, as three of the combined methods are comparable in MRE to the best performing neighborhood. We also see some synergy in this problem set, between 50 to 80 seconds, when the combined methods are able to outperform the pure neighborhood heuristics. When we look at Table III we see a different perspective on the results. For all time limits shown, the three combined methods are finding the best solution for over 70% of the problem instances.

On the 40x40 problem set shown in Figure 4, the best combined method, adaptive probability (*AdaptP*), leads all other neighborhood heuristics (except the cost based heuristic in the beginning) for processing times up to 800 seconds. However, in the long run, the time window neighborhood eventually outperforms adaptive probability. We believe the reason for this is that the combined methods have to pay a computational price when learning: they must also explore the less promising neighborhoods in order to determine which methods perform the best. It may be possible to 'tune' adaptive probability to outperform the best method for all time limits, however, it is likely that doing so may reduce robustness when applying these methods to different types of problems.

When we look at Table IV we see a similar picture. The time window neighborhood eventually finds the best solutions in 80% of the cases. Yet it is interesting to note that even at 1200 seconds, adaptive probability is still finding better solutions in 20% of the problem instances.

It is evident on both problem sets that we have achieved robustness, and in doing so, have reduced the need for human expertise. The only method which does not perform well is adaptive runtime with growing time allocation (*AdaptR-double*). It appears important not to commit to a single neighborhood heuristic for a long period of time. The other three neighborhood heuristics which interleave neighborhoods are able to remain competitive while adaptive runtime with growing time allocation performs poorly. This is an interesting result, as we experienced the opposite result in our previous work on algorithm control.

When we compare the successful combination approaches, the winner on the large problem set is clearly adaptive probability (*AdaptP*). However, both randomly picking a neighborhood heuristic (*RandP*) and adapting runtime with a constant time (*AdaptR-static*) also perform quite well. We believe that as the number of available neighborhoods increases, we would see the gap between the learning and non-learning methods grow. Since we only have 4 neighborhood heuristics to choose from and they are all fairly strong contenders, good performance is achieved simply by alternating.

9. Future Work

We are interested to see if we can boost the performance of the learning algorithms. At the moment, all of the algorithms perform on-line learning, which means they start with no prior knowledge. However, it seems to be the case that learning could be performed across problem instances. Indeed, our initial experiment which identified promising parameter settings could be generalized so that information gathered can be used to guide future searches. For instance, we can try and learn more specific probabilities for improvement, based on how the search has progressed.

It would also be interesting to look at further developments using the cost based concept. We presented a heuristic based only on critical operations. It seems that this idea works well to focus on the part of the problem causing the most obvious contribution to the objective. However, it is inherent that this approach does not diversify search. Instead it has the effect of intensifying search, which can lead to getting stuck. It would be interesting to combine the cost based approach with the time window and resource neighborhood heuristics. For example, instead of sweeping across these neighborhoods in a predefined order, as we have in this paper, it would be interesting to visit neighborhoods based on their cost impact. So, for a resource load neighborhood heuristic, we would first select the resources which have the largest number of critical activities on it, and so on.

The cost based neighborhood principle is general and we have only applied it to scheduling with makespan minimization. It would be interesting to see how easy it is to develop cost based neighborhoods for other problem domains. Our intuition is that a cost based neighborhood is a natural concept and will be relatively straightforward to implement given some domain knowledge. Thus, perhaps it is possible to generalize the work on cost based neighborhoods so that a non-expert user is allowed to decide which elements of the solution have the highest impact.

Finally, we are interested to see what happens when we combine the cost based neighborhood approach with the idea of propagation guided LNS presented in (Perron et al., 2004). Propagation Guided LNS is a very effective way to create neighborhoods containing related variables. We expect the combination of cost based guidance with propagation guided LNS will perform very well indeed.

10. Conclusion

We have presented design principles for large neighborhood search and evaluated them on some challenging job shop scheduling problems. We also introduced two new neighborhood heuristics, cost based and resource load, which have strong performance but, like all neighborhood heuristics, this performance varies across different problem sets and time limits. By combining neighborhoods with the use of a learning algorithm we were able to achieve robust solving performance, so that combined methods always performed as well as the best neighborhood heuristic.

This result is important as it reduces the need for human expertise in applying optimization algorithms. We believe this work is a significant step towards the goal to deploy a CP toolkit which is able to automatically solve problems without the need for extensive tuning.

References

Beck, J. C. and M. S. Fox: 2000, 'Dynamic Problem Structure Analysis as a Basis for Constraint-Directed Scheduling Heuristics'. *Artificial Intelligence* **117**(1), 31–81.

Birrattari, M., T. Sttzle, L. Paquete, and K. Varrentrapp: 2002, 'A Racing Algorithm For Configuring Metaheuristics'. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. pp. 11–18.

Carchrae, T. and J. Beck: 2005, 'Applying Machine Learning to Low-Knowledge Control of Optimization Algorithms'. *Computational Intelligence* **21**(4), 372–387.

Cesta, A., A. Oddi, and S. Smith: 2000, 'Iterative Flattening: A Scalable Method for Solving Multi-Capacity Scheduling Problems'. In: *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*. pp. 742–747, AAAI Press / The MIT Press.

Chabrier, A., E. Danna, C. Le Pape, and L. Perron: 2004, 'Solving a Network Design Problem'. *Annals of Operations Research* **130**, 217–239.

Danna, E. and L. Perron: 2003, 'Structured vs. Unstructured Large Neighborhood Search: A Case Study on Job-Shop Scheduling Problems with Earliness and Tardiness Costs.'. In: *Ninth International Conference on Principles and Practice of Constraint Programming*. pp. 817–821.

Dechter, R., I. Meiri, and J. Pearl: 1991, 'Temporal constraint networks'. *Artificial Intelligence* **49**, 61–95.

Garey, M. R. and D. S. Johnson: 1979, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York.

Godard, D., P. Laborie, and W. Nuijten: 2005, ‘Randomized Large Neighborhood Search for Cumulative Scheduling’. In: *Proceedings of Fifteenth International Conference on Automated Planning and Scheduling*. pp. 81–89.

Gomes, C. P., B. Selman, and H. Kautz: 1998, ‘Boosting combinatorial search through randomization’. In: *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*. pp. 431–437.

Laborie, P.: 2003, ‘Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results’. *Artificial Intelligence* **143**, 151–188.

Le Pape, C., L. Perron, J. Régin, and P. Shaw: 2002, ‘Robust and parallel solving of a network design problem’. In: *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP02)*. pp. 633–648.

Luby, M., A. Sinclair, and D. Zuckerman: 1993, ‘Optimal speedup of Las Vegas algorithms’. *Information Processing Letters* **47**, 173–180.

Michel, L. and P. Van Hentenryck: 2004, ‘Iterative Relaxations for Iterative Flattening in Cumulative Scheduling’. In: *Proceedings of Fourteenth International Conference on Automated Planning and Scheduling*. pp. 200–208.

Nuijten, W. and C. Le Pape: 1998, ‘Constraint-based job shop scheduling with ILOG scheduler’. *Journal of Heuristics* **3**, 271286.

Nuijten, W. P. M.: 1994, ‘Time and resource constrained scheduling: a constraint satisfaction approach’. Ph.D. thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology.

Perron, L. and P. Shaw: 2004, ‘Combining Forces to Solve the Car Sequencing Problem.’. In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. pp. 225–239.

Perron, L., P. Shaw, and V. Furnon: 2004, ‘Propagation Guided Large Neighborhood Search’. In: *Proceedings of Tenth International Conference on Principles and Practice of Constraint Programming*. pp. 468–481.

Shaw, P.: 1998, ‘Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems’. *Lecture Notes in Computer Science* **1520**, 417.

Wu, H. and P. van Beek: 2003, ‘Restart Strategies: Analysis and Simulation’. In: *Ninth International Conference on Principles and Practice of Constraint Programming*. p. 1001.