Large Neighborhood Search for Robust Solutions for Constraint Satisfaction Problems with Ordered Domains

Jheisson López[1]
University College Cork, School of Computer Science, Ireland
SFI Centre for Research Training in Artificial Intelligence, Cork, Ireland

Alejandro Arbelaez
Department of Computer Engineering, Autonomous University of Madrid, Spain

Laura Climent
Department of Computer Engineering, Autonomous University of Madrid, Spain

Abstract

Often, real-world Constraint Satisfaction Problems (CSPs) are subject to uncertainty/dynamism not known in advance. Some techniques in the literature offer robust solutions for CSPs. Here, we analyze a previous exact/complete approach from the state-of-the-art that focuses on CSPs with ordered domains and dynamic bounds. However, this approach has low performance in large-scale CSPs. For this reason, in this paper, we present an inexact/incomplete approach that is faster at finding robust solutions for large-scale CSPs. It is useful when the computation time available for finding a solution is limited and/or in situations where a new one must be re-computed online because the dynamism invalidated the original one. Specifically, we present a Large Neighbourhood Search (LNS) algorithm combined with Constraint Programming (CP) and Branch-and-bound (B&B) that searches for robust solutions. We also present a robust-value selection heuristic that guides the search toward more promising branches. We evaluate our approach with large-scale CSPs instances, including the case study of scheduling problems. The evaluation shows a considerable improvement in the robustness of the solutions achieved by our algorithm for large-scale CSPs.

2012 ACM Subject Classification Computing methodologies → Artificial intelligence

Keywords and phrases Constraint Programming, Large Neighbourhood Search, Robust Solutions

Digital Object Identifier 10.4230/LIPIcs.CP.2022.33

Funding This publication has emanated from research conducted with the financial support of Science Foundation Ireland under Grant number 18/CRT/6223.

Acknowledgements We thank Christophe Lecoutre for his assistance with the ACE solver.

1 Introduction

Most real Combinatorial Optimization Problems (COP) have unknown dynamism associated. Therefore the techniques used for solving COPs should deal with the uncertainty. The uncertainties can be associated with environmental factors; i.e., unexpected events in the context of the problem, or system factors; e.g., implementation errors or operational failures. Eventually, the uncertainties could affect the feasibility and the cost of a solution [21]. Even if the magnitude of the perturbations in the problem treated is small, they can cause huge deviations in the solutions of the models and in the model itself [3]. Hence, non-robust solutions for real applications under uncertainty can lead to serious economic losses.

[1] Contact Author

There are two ways to cope with uncertainty: reactive and proactive approaches. Reactive approaches re-solve the problem by using the experience gained to solve the previous states of the problem. Then, they provide a new solution. The disadvantages of these techniques are the extra computation time required in the new solving and the loss of the original solution. In many real applications, such as online planning and scheduling, the time required to compute a new solution may be too long for actions to be taken on time. Instead, proactive approaches use prior information about the uncertainty for finding robust solutions, which have a high probability of remaining valid despite future possible changes [7]. The disadvantage of proactive approaches is that they require a certain degree of prior knowledge about uncertainty. However, in real applications, this information is typically limited.

While many researchers have addressed the topic of robust and stable solutions in CP for problem dependent, much less effort has been made for problem independent, i.e. using CP as a black-box search. For this reason, we propose a generic proactive approach that addresses these scenarios without the need of prior detailed information about the uncertainty of the problem. In the same vein as [7] we assume that future events can restrict the search space (i.e. constraints can become more restrictive) in Constraint Satisfaction Problems (CSPs) with ordered domains. We also consider as robustness objective function the feasibility checking of neighbour values of the solution. Note that the CSPs must have several feasible neighbour solutions so that the robustness measurement is applicable. The authors of [7], propose a Branch and Bound (B&B) algorithm that uses CP to obtain robust solutions for CSPs, but their complete/exact technique presents a low performance in large-scale CSPs even using a large amount of computational time.

The above mentioned disadvantage has motivated the work presented in this paper. We present an incomplete/inexact approach that, unlike complete/exact algorithms, do not guarantee to find a globally optimal solution. The advantage of incomplete/inexact algorithms is that they can provide near-optimal solutions when the computation time is limited. Our approach aims to obtain robust solutions (not necessarily the most robust solution) for CSPs that typically do not scale well with complete techniques (large-scale and NP-complete CSPs). It is especially useful when the computation time available for finding a solution is limited and/or a new solution must be re-computed on-line (because the original one is lost due to the dynamism). Specifically, we present the following contributions:

- A Large Neighbourhood Search (LNS) algorithm with CP and B&B that considers the same assumptions as [7]. And therefore, it computes robust solutions for CSPs that have a high number of feasible neighbour values (Algorithms 1 and 2, Section 4).
- A robust-value selection heuristic (Algorithm 4, Section 4) that guides the search towards more promising branches (more likely to contain more feasible neighbour values solutions).

LNS is a technique that takes an initial solution and gradually improves it by alternately destroying and repairing the solution. LNS combined with CP adds the constraint propagation process into the LNS iterations to perform partial assignments that are feasible.

Our approach works for many domains that can be modeled as CSPs with ordered domains (where the bounds can undergo changes). In this paper, we evaluate general CSPs randomly generated, which show the generality of the applicability of our approach; and the well-known scheduling problems [11, 2]. The experiments suggest that our approach is effective on large-scale CSPs and that it outperforms the current approach from the state-of-the-art.

The rest of the paper is organized as follows: Section 2 is about the literature review. In Section 3 the technical background is described. Section 4 describes our algorithm. Section 5 describes the experiments and results; finally, in Section 6 we explain the conclusions and future work.

2 Literature Review

Since our approach is proactive and it aims to find robust solutions with the use of LNS with CP, in this section, we present a literature review of these two topics.

2.1 Proactive Approaches

The purpose of finding robust solutions is present in both Mathematical Programming (MP) and Constraint Programming (CP). Toklu [20] presents a revision of the approaches to deal with the uncertainty in Mathematical Programming. Furthermore, [22] provides a detailed review of approaches to model uncertainty with CP. The authors state that in CP the efforts had been directed to find solutions that can easily be adapted to obtain a new solution after a change in the conditions of the problem occurs (flexible solutions) and solutions that resist the possible changes in the input data (robust solutions).

Below, we present two subsections about proactive approaches. We classify them based on the amount of information available about the uncertainty that the approaches require.

2.1.1 With detailed uncertainty information

One of the most popular proactive approaches is *Robust Optimization* (RO). RO expresses the uncertainty of the variables as sets of possible values (uncertainty sets). The optimal solution is the one that remains feasible for the constraints, whatever the realization of the data within the uncertainty sets [3]. Another popular proactive approach is *Stochastic Optimization* (SO). In SO some variables are considered random with a probability function associated that expresses its likelihood to take any particular value in its uncertain domain [22]. One technique for solving this type of problem is by using *chance constraints* [1]. These constraints contain at least one random variable and they fix a minimum threshold of probability of being satisfied. A possible objective could be to find a solution that maximises its probability of satisfaction. Finally, some proactive approaches recur to the *Fuzzy Set* theory. In that approach, the uncertainty parameters of the model can be expressed as fuzzy sets and subjective membership functions defined using expert opinions [8].

In the search for robust solutions in CSPs, the *Mixed CSP* [10] approach introduces the concept of uncontrollable variables and the idea is to find assignments to the decision variables that satisfy all the possible values of the uncontrollable variables (a kind of Robust Optimization approach). Another outstanding proposal is the *Stochastic CSP* which assumes a probability distribution associated with the uncertain domain of each uncontrollable variable and tries to find a solution with the maximum probability of feasibility [23].

2.1.2 Without detailed uncertainty information

Even if the proactive approaches mentioned in the previous subsection, work very well when all the needed information about the uncertainty is available, unfortunately, they can not be used if such information is missing. This is the case of many real applications for which it is not possible to obtain a probability distributions associated with the uncertain variables [6].

A prominent work aiming to find flexible solutions in presented in [13]. In this work the author proposed the concept of *(a, b)-super-solution*, a set of solutions for which the loss of values of at most a variables can be repaired by assigning other values to those variables and changing the values of at most b other variables. Flexibility is a desired property of a CSP solution but our main goal is to avoid the necessity of modifying the solution after identifying inconsistencies in the solution due to certain uncertain events.

Climent, et al. [7] state that the solution can be lost when the constraints become more restrictive (and the solution space becomes narrow). Therefore *"the most robust solution of a CSP with ordered domains without detailed dynamism data is the solution that maximizes the distance from all the dynamic bounds of the solution space"*. In this paper, we take the same assumptions about the uncertainty and the same robustness definition.

2.2 Large Neighbourhood Search

Large Neighborhood Search was proposed by P. Shaw [18] to solve Vehicle Routing Problems using CP techniques. Since then, LNS and CP has been applied to tackle multiple problems ranging from the protein structure prediction problem [9] to balance bike-sharing systems [12].

Carchrae and Beck [5] propose three general principles for the design of LNS algorithms: i) define neighbourhoods unassigning the variables that most affect the cost of the current solution and ii) gradually increment the neighbourhood size and iii) apply learning algorithms to determine the most promising neighborhood heuristics. Some works fix a constant size for the neighborhood and other works use a variable neighbourhood size. For example, in [17], Perron, et. al fix the size of the neighbourhood based on the sum of the domain size logarithms of unassigned variables (until reaching an upper bound). For scheduling, for example, Carchrae and Beck [5] fix the size of the neighbourhood based on different time windows (all the variables in such time windows are unassigned) or based on resources (the variables associated with a subset of resources are unassigned). Pacino and Van Hentenryck [16] increment the size of the neighborhoods after five iterations without improvements in the solutions found. The increments are no longer effective when a better solution is found.

However, in this paper, we focus our attention on value selection heuristics as it significantly impacts the robustness of the solutions. For this reason, we have developed Algorithm 4 (see Section 4), which guides the search towards more robust promising branches.

3 Background

In this section, we explain the background associated with CSPs and their robust solutions.

3.1 Constraint Satisfaction Problems

CSPs are characterized by a set of variables with their corresponding domains and a set of associated constraints. A formal definition of a CSPs is presented below.

▶ **Definition 1** (Constraint Satisfaction Problems). *A Constraint Satisfaction Problem (CSP) is represented as a triple $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where $\mathcal{X} = \{x_1, ..., x_n\}$ is a finite set of variables, $\mathcal{D} = \{\mathcal{D}(x_1), ..., \mathcal{D}(x_n)\}$ is the set of domains of the variables in $\mathcal{X}$, and $\mathcal{C} = \{C_1, C_2, ...C_n\}$ is the set of constraints which restrict the values that the variables can simultaneously take.*

Solving a CSP involves finding a solution, i.e., an assignment of values to variables such that all the constraints are satisfied. If a solution exists, the problem is stated as satisfiable, and unsatisfiable otherwise. CSPs can be tackled using depth-first search backtracking algorithms in which a value is assigned to some variable at each step to compute partial assignments. Below, we present the formal definitions of these concepts.

▶ **Definition 2** (Partial Assignment). *A partial/complete assignment s is an assignment of values to a subset of variables $\mathcal{X}_s \subseteq \mathcal{X}$ in their specific domain $\mathcal{D}_s$.*

▶ **Definition 3** (Solution). *A solution s is a complete assignment to the whole set of variables $\mathcal{X}$ that satisfies all the constraints in $\mathcal{C}$.*

3.2 Robust Solutions by Feasible Neighbour Values

As mentioned in Section 2.1, we take the same limited assumptions about robustness as in [7]. In the absence of detailed information about the uncertainty, the most robust solution is the one with the maximal distance from all the dynamic bounds of the solution space. In Constraint Programming the solution spaces can be non-convex. Then, the distance to the bounds can not be computed as linear equations. Therefore, for CSPs with ordered domains, the authors stated: *"we can only ensure that a solution s is located at least at a distance d from a bound in a certain direction of the n-dimensional space if all the tuples (possible solutions) at distances lower or equal to d from s in this direction are feasible"*.

The authors of [7] define the feasible neighbour values set $\mathcal{N}_k(x, v, s, \oplus)$ by introducing a parameter that fixes the maximum distance k of the feasibility checking for a variable x and its assigned value v, with respect a partial/complete assignment s. $\mathcal{D}_s(x) \subseteq \mathcal{D}(x)$ represents the subset of domains values of the variable x that are consistent with the feasible partial assignment s. $\oplus$ is a set of operators pairs which indicate the directions (order) of the neighbour values to check. The list of operators used is $\oplus = \{\{>, +\}, \{<, -\}\}$. Each operator pair is denoted as $\oplus_i$. The set $\{>, +\}$ ($\oplus_1$) refers to values greater than v (increasing direction) and the set $\{<, -\}$ ($\oplus_2$) refers to values lower than v (decreasing direction). For each operator pair, the operator in the position j is referenced as $\oplus_{ij}$ (for instance, $\oplus_{12}$ references the operator $+$). Below, we describe the equation of the feasible neighbours values presented by the authors.

$$\begin{aligned}\mathcal{N}_k(x, v, s, \oplus) = \{&w \in \mathcal{D}_s(x) : \exists \oplus_i, w \oplus_{i1} v \wedge |v - w| \leq k \\ &\wedge \forall \oplus_z \, \forall j \in [1 \; (|v - w| - 1)], (v \oplus_{z2} j) \in \mathcal{D}_s(x)\}\end{aligned} \tag{1}$$

The first condition (first line) of Equation 1 checks that there are values w, in the domain of x ($\mathcal{D}_s(x)$), which are greater or lower than v according to the corresponding operator ($>$ or $<$) and that the distance between v and w is lower or equal to k. The second condition (second line) ensures that all values that are closer to v than w (values selected by applying the operator $+$ and/or $-$ to v with the iterator j) are also feasible values for s. If at least one of them is not feasible, the value w cannot belong to $\mathcal{N}_k(x, v, s, \oplus)$ because it must be a set of contiguous feasible neighbour values. Note that the concept of neighbour values (associated with the values whose feasibility has to be checked) differs from the neighbourhood term used in LNS (associated with the unassigned variables to optimize in each iteration).

3.3 Objective Function

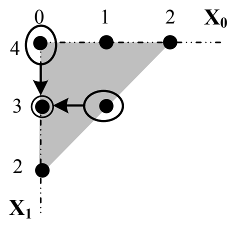
The objective function (o.f.) described in Equation 2 is the sum of the feasible neighbour values sets of all the variables of the solution s. Note that the k parameter is an input used to indicate the grade of robustness checking during the search process. Then, the k parameter determines the upper bound of the neighborhood size $\mathcal{N}_k$ (see Equation 1).

$$f(s, k, \oplus) = \sum_{x \in \mathcal{X}_s} |\mathcal{N}_k(x, s(x), s, \oplus)| \tag{2}$$

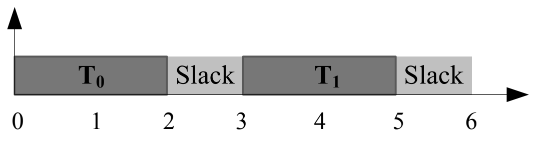
For example, in Figure 1a the grey area represents the solution space (for which the bounds can become more restrictive). The most robust solution for $k = 1$, considering only the increasing direction $\oplus_1 = \{>, +\}$, is highlighted ($x_0 = 0, x_1 = 3$). The greater feasible neighbour values (1 for x_0 and 4 for x_1) are also highlighted in the figure. Note that the variable x_0 can also be equal to 1 and the solution would remain valid (in case that 0 is not feasible anymore due to a restrictive change of the upper bound of the solution space).

Realize also of the alternative, which is that the variable x_1 can also be equal to 4 (in case that 3 is not feasible anymore for the same reason mentioned before). Therefore, the o.f. value of this solution is two (because it is the sum of its feasible neighbour values).

3.4 A case study: robust Scheduling



(a) Robust solution.



(b) Associated robust schedule.

Figure 1 Example of a robust solution for scheduling.

Without loss of generality, we selected a scheduling problem as a case study. We would like to recall that our approach works for many applications suitable for being modelled as CSPs with ordered domains in which the bounds can undergo restrictive changes. Therefore, it is not a scheduling specific approach. But considering scheduling problems as a case study is especially interesting because unexpected delays in tasks are common and such uncertainty must be considered in the solving process. Below, we present some robust-schedules concepts.

Scheduling problems consist in assigning a set of n tasks, of varying processing times, to m machines with a limited capacity. In the case study presented in Section 5, we focus on the Job Shop Scheduling Problem (JSP). This type of scheduling problem is characterized by having tasks with a specific precedence order within a job. Typically, the CSPs models of scheduling problems consider as variables the start times of the tasks and the objective is to minimize the makespan (the finish time of the last task). However, as stated in Section 1, we aim to search for robust solutions using the o.f. in Equation 2. For this reason, we consider scheduling instances that have a fixed makespan (then, their models are CSPs and not CSOPs). For example, in Figure 1b, the makespan is fixed to six.

When possible future changes over the constraints are equally probable and independent of each other, the more slacks a schedule has, the more robust it is (ideally, they are uniformly distributed and of similar size). The slack/buffer (see Figure 1b) is a time slot that can absorb a delay of the task placed before the slack without affecting the schedule. Then, we fix the set of operands in Equation 2 to $\{>, +\}$ to check the greater feasible neighbour values (equivalent to the slack/buffer between tasks in the schedules). The selection of these operands is because tasks that last longer than expected can invalidate the solution, while shorter tasks can not. Figure 1b shows the associated robust schedule with the highlighted solution of Figure 1a. Note that each task has a buffer of one unit, which corresponds to each feasible neighbour value associated with each variable. Equation 3 shows a standard measure of the slack of a schedule [19]. R^s_{Slack} is the average size of the slacks minus the standard deviation of their sizes. The β parameter regulates the importance of slack size uniformity. The authors suggest $\beta = 0.25$.

$$R^s_{Slack} = avg(slack) - \beta * std(slack). \tag{3}$$

4 LNS for Robust Solutions (LNSR)

In this section, we explain the LNS algorithm with CP for finding robust solutions for CSPs (LNSR). Figure 2 describes our LNSR algorithm (Algorithm 1). It is an iterative algorithm that takes the most robust solution found (incumbent) and creates a sub-problem by defining a neighbourhood of variables to unassign (Algorithm 2). Subsequently, it optimizes the sub-problem by propagating the constraints (Algorithm 3). For such purpose, we design a robust-value selection heuristic (Algorithm 4) that selects the most robust value based on the count of the feasible neighbour values of the already assigned variables. If a more robust solution is found, then the incumbent solution is updated. The iterative process continues by computing a new sub-problem according to a neighbourhood heuristic. In the following subsections, we present all the above-mentioned algorithms.

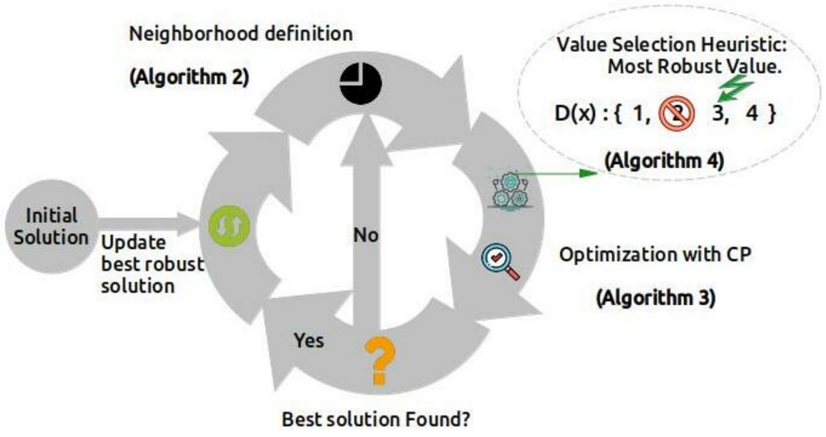


Figure 2 LNS for Robust Solutions (LNSR) (**Algorithm 1**).

4.1 The Main Iterative Process

Algorithm 1 describes the main iterative process of our LNSR. In line 1, the best solution bS and the current solution s are initialized by using a CSP solver that finds a simple solution (non-robust). In line 2, Equation 2 is applied to calculate the number of feasible neighbours values of the initial solution ($b\mathcal{N}$). In line 3, the counter of the number of LNS neighbourhoods explored ($neighC$, a.k.a. no. of restarts) is zero-initialized. Lines 4-10 contain the main LNS loop, which runs during $timeLim$. Every iteration corresponds to a neighbourhood exploration. In line 6, Algorithm 2 (*neighbourhood*) defines the new neighbourhood to explore (a new sub-problem) and returns $\mathcal{X}_s$ (the set of variables that will remain assigned) and the corresponding partial solution s. In line 7, Algorithm 3 (*opt*) explores the neighbourhood until reaching the failures limit ($fLim$). In addition, fC (zero-initialized in line 5) keeps the count of fails committed during the constraints propagation in every neighbourhood exploration. Then, *opt* returns the best solution found so far (the most robust solution found) and its corresponding number of feasible neighbour values. Note that if this algorithm does not find a better solution in the current neighbourhood explored, bS and $b\mathcal{N}$ remain the same. At the end of every iteration, the $fLim$ parameter is updated using a geometrical strategy [2] [24] (line 9).

[2] $fLim = base * (1.1^{neighC})$

Algorithm 1 *LNSR*: LNS for Robust Solutions.

```
   Input: $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, $k$, $\oplus$, $timeLim$, $fLim$
   Output: $bS, b\mathcal{N}$
1  $bS \leftarrow s \leftarrow solve(\mathcal{P})$ // Most robust solution found
2  $b\mathcal{N} \leftarrow f(s, k, \oplus)$ // Best no. of feasible neighbour values found
3  $neighC \leftarrow 0$ // Number of LNS neighbourhoods explored
4  repeat
5      $fC \leftarrow 0$ // Number of failures count
6      $(\mathcal{X}_s, s) \leftarrow neighbourhood(\mathcal{X}, k, bS, s)$
7      $(bS, b\mathcal{N}) \leftarrow opt(\mathcal{P}, k, \mathcal{X}_s, \oplus, s, bS, b\mathcal{N}, fLim, fC)$
8      $neighC{+}{+}$
9      $fLim \leftarrow geometricUpdateFails(neighC)$
10 until $timeLim$
```

4.2 Neighborhood Selection Heuristic

Algorithm 2 describes the neighbourhood heuristic which determines the new neighbourhood to explore (a new sub-problem) for the next iteration of the Algorithm 1. First, the size of the neighbourhood ($|\mathcal{X}| - n$) is fixed to the 20% of the number of variables (line 1). We use two variable selection heuristics: random or the well-known *Wdeg/domSize* heuristic from the literature [4], which is already implemented in the ACE solver.

In LNSR, if in the previous iteration of the main algorithm a better solution has not been found (in this case the current solution s and the best solution bS are different, line 2), then, the algorithm selects the variables to unassign randomly (line 3). Selecting a random neighbourhood is a typical technique in the literature for achieving getting out from a local optimum (avoiding then the repeatedly exploration of the same neighborhood).

However, when a better solution is found (line 4), the variables with the lowest *Wdeg/-domSize* are selected to remain assigned (line 5). The *Wdeg* is the number of unsatisfied constraints (during the propagation) associated with the variable. Variables with small domains tend to fail more during propagation. Therefore, the greatest *Wdeg/domSize* heuristic selects the variables that have the greatest tendency to fail; in such a way difficult sub-problems are explored (fail-first principle [4]). Note that line 5 fixes the set of variables

Algorithm 2 *neighbourhood*: NBHD heuristic.

```
   Input: $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, $bS$, $s$
   Output: $\mathcal{X}_s$, $s$
1  $n \leftarrow round(80\% \ |\mathcal{X}|)$
2  if $s \neq bS$ then // No better solution found
3      $\mathcal{X}_s \leftarrow selRandomVars(\mathcal{X}, n)$
4  else // New best solution found
5      $\mathcal{X}_s \leftarrow selVarsLowestWdegOnDom(\mathcal{X}, n)$
6  $s \leftarrow \emptyset$
7  foreach $x$ in $\mathcal{X}_s$ do
8      $s \leftarrow s \cup \{x = bS(x)\}$
9  $propagate(\mathcal{P}, s)$
```

that will remain assigned ($\mathcal{X}_s$) to the variables with the lowest *Wdeg/domSize*, consequently, the unassigned variables are the ones with the greatest *Wdeg/domSize*. In lines 6-8, the partial assignment s is built by selecting from bS the values corresponding to $\mathcal{X}_s$.

We would like to remind that because of the inference CP process, all the constraints associated with the variables that will not belong to the new neighbourhood to explore (i.e. assigned variables) have to be propagated (line 9), and consequently, the domains of such variables will be pruned (if there are unfeasible values).

4.3 Optimization Algorithm

The *opt* algorithm (Algorithm 3) explores the new neighbourhood (previously selected by the Algorithm 2) to find a better solution than the best one found so far (bS). This optimization algorithm is similar to the one presented in [7], but we have adapted it to the LNS algorithm behaviour (only a sub-problem is optimized each time). We also include a more informed way to compute the o.f. bound and a robust-value selection heuristic (Section 4.4). The *opt* algorithm is recursive and tries to assign a variable of the sub-problem in each call.

In line 1, the algorithm selects the unassigned variable x with the greatest *Wdeg/domSize* value (explained in Algorithm 2) [4]. Then, it updates $\mathcal{X}_s$ by adding x to this set (line 2). In line 3, it saves all the domains of the variables and the $\mathcal{N}_k$ sets of the already assigned ones (since they must be restored when backtracking occurs). A loop (lines 4-21) iterates over the values in $\mathcal{D}(x)$. In line 5, the values are selected based on the robust value selection heuristic (*selMostRobVal*, Algorithm 4) or using the typical first value selection, a.k.a. lexicographical order, (*selFirstVal*). We tested several combinations of both heuristics in the evaluation section (Section 5). As is typical in CSP solvers, the search process restarts after several value assignment failures are reached ($fLim$) (second condition of line 4). In this case, LNSR would restart in a new iteration of Algorithm 1 with a new neighbourhood.

In line 6, the *propagate* procedure prunes the domains of the unassigned variables and the $\mathcal{N}_k$ sets of the assigned ones according to the constraints propagation when $x = val$. Typically, CSP solvers propagate the constraints only over the domains of the unassigned variables. However, we also need to keep updated the $\mathcal{N}_k$ sets. For this reason, we extend the *propagate* procedure so that it also prunes the $\mathcal{N}_k$ sets (Equation 1) of the previously assigned variables and creates the $\mathcal{N}_k$ set for the recently assigned one. This procedure returns *true* if the assignation is feasible, otherwise *false*. In line 7, the algorithm updates the current solution s ($x = val$), so that it can be used in the o.f. calculation.

In line 8, the algorithm calculates an upper *bound* of the o.f. (Equation 2) as the sum of the o.f. of the assigned variables (i.e. the partial assignment s) and the max. possible feasible neighbour values of the unassigned variable. The latest is calculated as the lowest value between $|\oplus| * k$ and its domain size. Thus, the maximum size of the set of neighbour values for each variable is $2k$ if $\oplus$ is composed of two operator pairs (such as in random CSPs) or k if $\oplus$ is composed of only one operator pair (such as in scheduling CSPs). Note that, as mentioned above, the $\mathcal{N}_k$ sets of the already assigned variables have to be updated after every variable assignment so that the o.f. value can be properly computed.

Then, in line 9, if the *bound* of robustness is greater than the best one found so far ($b\mathcal{N}$), it means that it is possible to find a better solution by exploring such branch (i.e. partial assignment s). In this case, the search continues by calling recursively to the *opt* algorithm (line 14). When a complete solution is found (line 10) the best solution and its associated robustness are updated, which are denoted as bS and $b\mathcal{N}$ correspondingly (lines 11-12).

However, if the *bound* of robustness is lower (line 15) or s is not feasible (line 17), the count of failures fC is incremented by one (lines 16 and 18). In both cases, this branch of the search tree is discarded since it cannot produce a better solution than the best one so

Algorithm 3 *opt*: optimization algorithm.

```
Input: $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle, k, \mathcal{X}_s, \oplus, s, bS, b\mathcal{N}, fLim, fC$
Output: $bS$, $b\mathcal{N}$
1  $x \leftarrow selVarGreatestWdegOnDom(\mathcal{X} \setminus \mathcal{X}_s)$
2  $\mathcal{X}_s \leftarrow \mathcal{X}_s \cup x$
3  $save(\mathcal{D}, \mathcal{N}_k)$
4  while $\mathcal{D}(x) \neq \emptyset \wedge (fC < fLim)$ do
5      $val \leftarrow selMostRobVal(\mathcal{P}, \mathcal{X}_s, s, x, \oplus)$ or $selFirstVal(\mathcal{P}, x)$
6      if $propagate(\mathcal{P}, s, x = val)$ then
7          $s \leftarrow s \cup \{x = val\}$
8          $bound \leftarrow f(s, k, \oplus) + \sum_{y \in \mathcal{X} \setminus \mathcal{X}_s} \min((|\oplus| * k), |\mathcal{D}(y)|)$
9          if $bound > b\mathcal{N}$ then
10             if $\mathcal{X}_s = \mathcal{X}$ then
11                 $b\mathcal{N} \leftarrow bound$
12                 $bS \leftarrow s$
13             else
14                 $opt(\mathcal{P}, k, \mathcal{X}_s, \oplus, s, bS, b\mathcal{N}, fLim, fC)$
15         else
16             $fC{+}{+}$ // Number of failures count
17     else
18         $fC{+}{+}$ // Number of failures count
19     $restore(\mathcal{D} \setminus \mathcal{D}(x), \mathcal{N}_k))$
20     $\mathcal{D}(x) \leftarrow \mathcal{D}(x) \setminus val$
21     $s \leftarrow s \setminus \{x = val\}$
22 $restore(\mathcal{D}(x))$
23 $\mathcal{X}_s \leftarrow \mathcal{X}_s \setminus x$
```

far (Branch and Bound) [25]. Since a different branch has to be explored, it is necessary to restore all the domains, except the one of the analyzed variable ($\mathcal{D}(x)$), and the $\mathcal{N}_k$ sets of the already assigned ones (line 19), so that the propagation effect of the assignation $x = val$ is re-established. In addition, the value val is erased from $\mathcal{D}(x)$ (line 20) and s (line 21). The search will continue with the next most robust value in $\mathcal{D}(x)$ (line 5). When a wipeout occurs ($\mathcal{D}(x) = \emptyset$, in line 4), it is necessary to restore the domain of the variable x (line 22), exclude it from $\mathcal{X}_s$ (line 23) and make a backtrack to the previous variable.

4.4 Robust Value Selection Heuristic

This section describes the robust-value selection heuristic, which is novel and represents a contribution of this paper. The main idea of this heuristic is to assign a value to the current variable to assign that has the lowest negative impact over the robustness of the partial assignment s (composed by the previous variables assigned). This means that the value assigned will be the one that less reduces the total number of feasible neighbours of s.

In line 1 of Algorithm 4 , the Q queue is initialized with the assigned variables that share a constraint with the variable x (the variable to assign). We use $var(c)$ to denote the scope of the constraint c (i.e. the variables involved in such constraint). Line 2 initializes the set

Algorithm 4 *selMostRobVal*: value sel. heuristic.

```
  Input: $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ , $\mathcal{X}_s$, $s$, $x$, $\oplus$
  Output: $bestValue$
1 $Q \leftarrow \{y, \forall y \in var(c), y \in \mathcal{X}_s \wedge x \in var(c)$ $, \forall c \in \mathcal{C}\}$
2 $sum\mathcal{N}_x \leftarrow \emptyset$ // var(c) is the scope of c
3 foreach $val \in \mathcal{D}(x)$ do
4     $s \leftarrow s \cup \{x = val\}$
5     $sum\mathcal{N}_x(val) = \sum_{y \in Q} |\mathcal{N}_k(y, s(y), s, \oplus)|$ // Eq.2
6     $s \leftarrow s \setminus \{x = val\}$
7 $bestValue \leftarrow val$ , $\max(sum\mathcal{N}_x(val))$ $\forall val \in \mathcal{D}(x))$
```

$sum\mathcal{N}_x$, which will be used to determine the estimation of robustness of the values of the domain of the variable x. Subsequently, a loop through all possible values val in the $\mathcal{D}(x)$ begins (line 3). In line 4, the value val is added to the partial solution to observe its effect over the neighbour feasible values of the assigned variables. This calculation is similar to the objective function (Equation 1) but considering only the variables in Q instead of all the variables (line 5). In line 6, the previously added value is erased from the partial solution s to continue with the next iteration (the next value of $\mathcal{D}(x)$). Finally, the value with the greatest number of feasible neighbour values in $sum\mathcal{N}_x$ is selected as $bestValue$ (line 7). The lexicographical order resolve ties.

5 Evaluation

For the evaluation, we implemented LNSR and the complete algorithm from [7] (we denote it as B&B) and we embedded them into the ACE CSP solver[3] (previously ABSCON) [15]. We used ACE for the constraint propagation, the restarts, the weighted degree (*Wdeg*) computation and the B&B. The experiments were run on a 2xIntel(R) Xeon(R) CPU (E5620 @ 2.40GHz) and 32 GB of RAM memory running Ubuntu Server 18.04. We run three times each LNSR configuration for each instance (because LNSR sometimes selects random variables, see Algorithm 2) and we present the average results.

5.1 Experimental Settings

We performed experiments for $k = 1, k = 3$ and $k = 5$. For the LNSR algorithm, we empirically tested several constant sizes of the neighbourhood, 20% was the most adequate. We used the same configurations for the common parameters of B&B and LNSR. The function in line 8 of Algorithm 3 was used as the upper bound of robustness. For the geometric restart strategy, we use the formula $fLim = base * (1.1^{neighC})$ with $base$ initially equal to 100 and double it every 50 restarts. We evaluated the following value selection heuristics:

- *First* (LNSR-1 and B&B-1): first value selection (lexicographical order).
- *First-Rob* (B&B-2): first value selection combined with the robust value selection heuristic: *selMostRobVal* (Algorithm 4). In the beginning, the first value selection heuristic is used in every restart only until reaching the first solution, then *selMostRobVal* is used for the rest of the search (including the backtracking).

[3] `https://github.com/xcsp3team/ace`

- *First-Rob* variant (LNSR-2 and B&B-2'): this is a *first-rob* variant. The first solution is computed with the first value selection heuristic. Then, for LNS $selMostRobVal$ (Algorithm 4) is applied to the variables in the LNS neighbourhood. This is equivalent to applying $selMostRobVal$ to the last n variables to assign in B&B, where n is the size of the LNS neighbourhood. Note that B&B-2' has the same value selection heuristic as LNS-2, while B&B-2 does not.

We also evaluated *Rob-Rob* value selection, where the first solution is also computed selecting the robust value ($selMostRobVal$). However, its performance was extremely poor, especially in scheduling instances. The problem is that selecting the most robust values for all the variables of the scheduling leads to dead-ends because such assignments exceed the maximum makespan allowed (especially in the critical path). Therefore, this algorithm configuration could not find even the first solution in the allocated time.

5.2 Evaluation with General CPSs

In this section, we present the results of our algorithm using general CSP instances generated with the uniform random generator URBCSP[4]. The random instances used have 110 variables, domain sizes of 120 and 1119 constraints (corresponding to the 20% of all the possible binary constraints). We use three different tightness values to generate each one of the instances: 1.3, 1.6 and 1.9. We fixed a time limit of 10 min.

The results are presented in Table 1. We show the number of solutions found ($\#S$), the number of visited nodes ($\#Nodes$), the number of failed assignments ($\#FAssigns$), the number of variables with at least one neighbour value ($\#varR$), the number of neighbour values of the solution ($\#N$) and a measure of the distribution of the feasible neighbour values across the solution ($Ndist$). The $Ndist$ measurement is computed with Equation 3, but considering the number of feasible neighbour values instead of the slack.

Table 1 Comparison of value selection heuristics and algorithms in random CSPs.

k	*Heuristic*	*Algorithm*	*#S*	*#Nodes*	*#FAssigns*	*#varR*	*#N*	*Ndist*
1	First	B&B-1	7.7	5291006.0	1928269.7	39.0	39.0	0.236
		LNSR-1	16.0	4076714.4	1619716.0	47.3	47.7	0.309
	First-Rob	B&B-2	7.3	5373612.3	1978766.7	39.7	39.7	0.241
		B&B-2'	9.0	4406818.3	1620720.0	44.0	46.3	0.290
		LNSR-2	**17.0**	3163868.8	1285432.4	**62.7**	**74.3**	**0.520**
3	First	B&B-1	8.7	4837787.3	2022192.0	36.7	43.7	0.245
		LNSR-1	15.3	3912019.8	1674863.6	43.3	51.3	0.308
	First-Rob	B&B-2	10.0	4838004.7	2045275.3	40.7	46.0	0.270
		B&B-2'	6.7	4010733.7	1699203.3	41.7	48.0	0.287
		LNSR-2	**18.0**	2920458.8	1245622.8	**59.0**	**88.7**	**0.581**
5	First	B&B-1	7.3	4803303.3	2067019.0	38.0	42.3	0.244
		LNSR-1	18.0	3845131.6	1650786.6	44.3	54.3	0.327
	First-Rob	B&B-2	9.3	4683463.0	1965024.7	38.0	43.7	0.249
		B&B-2'	6.7	4085214.7	1746265.3	41.7	49.0	0.291
		LNSR-2	**25.0**	2946410.4	1264325.4	**60.0**	**90.3**	**0.590**

Table 1 clearly shows that LNSR-2 has the best results for the three robustness measurements for all the k values. LNSR-2 finds solutions with the greatest number of robust variables ($\#varR$), which are the variables that have at least one feasible neighbour value.

[4] `http://www.lirmm.fr/~bessiere/generator.html`

This robustness measure shows how many variables of the solution would resist a perturbation of magnitude one in the bounds of the solution space (delimited by the constraints and the domains). LNSR-2 also obtains the greatest sum of the feasible neighbour values across all the variables (a.k.a. no. of neighbours, $\#N$). In addition, the solutions found by LNSR-2 have the best distribution of the feasible neighbour values through all the variables ($Ndist$). These results confirm our hypothesis that our LNSR algorithm performs better than B&B.

Note that the LNSR algorithm finds a higher number of solutions ($\#S$) than the B&B algorithm (especially, LNSR-2). Accordingly, the number of failed assignments ($\#FAssigns$) is lower. We believe that this is due to the diversification in the search space exploration that LNSR offers by exploring neighbourhoods around the current most robust solution. On the contrary, B&B gets stuck exploring very specific branches of the search tree.

Another important conclusion from the evaluation is that our heuristic for selecting the most robust values (Algorithm 4) has a very good performance, especially when used in LNSR. For the B&B algorithm, there is an improvement of B&B-2' over B&B-2 and from both over B&B-1. We believe that the robust value selection heuristic did not have a big impact on B&B because the instances are large-scale and therefore, as mentioned above, B&B get stuck exploring very specific branches of the search tree. The improvement is even more pronounced in the case of the LNSR algorithm. In almost all the cases, LNSR-2 obtains a robustness increment of about 70% of the values obtained by LNSR-1 (for the three robustness measures). This fact shows the usefulness of our robust value selection heuristic to guide the search toward more robust solutions.

The number of visited nodes ($\#Nodes$) is lower for B&B-2' and LNSR-2 than the other configurations. It is due to the most robust value selection heuristic is more costly than the first value selection heuristic. Even if our heuristic is costly, it has shown to be very effective, especially combined with our LNSR (LNSR-2).

5.3 Evaluation with Scheduling Instances

We evaluated our LNRS algorithm using six Taillard Job Shop scheduling instances with 300 tasks each[5]. Recall that these instances are modeled as CSPs (rather than CSOPs) because the makespan is fixed. The variables of the CSPs models are the starting and ending times of the tasks and the domains are limited by their possible maximum ending times. Note that the variables associated with the ending times are excluded from the robustness search since they are just equal to the start time variables plus the duration of the tasks. Equality constraints are used to ensure that the end time of a job is equal to the end time of its last operation. Precedence constraints are used to indicate the order of the operations in every job and non-overlapping constraints are used to indicate the correspondence between operations and the single capacity resources. Finally, the last constraints are used to indicate the minimum possible duration of every job.

In these executions, we fixed a time limit of 20 min. The average results of the evaluation are presented in Table 2. We show the same measurements as in the previous section, except for $\#B$, which is the number of buffers and R^{s}_{Slack}, which measures the buffers' distribution (see Equation 3). Note that $\#B$ is equivalent to $\#varR$ and R^{s}_{Slack} is equivalent to $Ndist$.

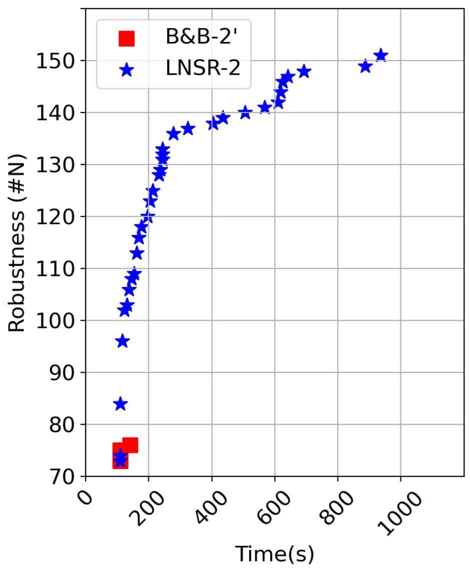
The results obtained in the scheduling instances are similar, in terms of the approaches ranking, to the results obtained in general CSPs instances. LNSR-2 outperforms the other approaches in the robustness measurements. Although in the scheduling instances, the differences between the three variants of B&B are lower than in the random CSPs (Table 1).

[5] Instances from `http://www.xcsp.org/instances/`: Taillard-js-020-15-{0,2,3,6,8,9}. Instances {1,4,5,7} could not be solved in the given cut-off time.

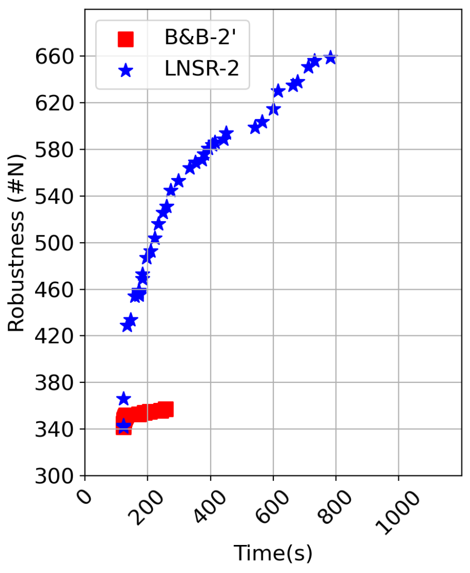
Table 2 Comparison of value selection heuristics and algorithms in Taillard-js-020-15 instance.

k	*Heuristic*	*Algorithm*	$\#S$	$\#Nodes$	$\#FAssigns$	$\#B$	$\#N$	R^s_{Slack}
1	First	B&B-1	2.5	5752220.8	2389769.0	75.7	75.7	0.144
		LNSR-1	14.2	3861320.2	501682.4	87.7	87.7	0.179
	First-Rob	B&B-2	2.5	5760396.8	2390411.2	75.7	75.7	0.144
		B&B-2'	2.5	5780979.8	2400829.3	75.7	75.7	0.144
		LNSR-2	**30.0**	3432879.6	508581.8	**157.2**	**157.2**	**0.400**
3	First	B&B-1	5.3	5156986.5	2125819.3	75.7	219.3	0.412
		LNSR-1	**40.0**	3153637.7	491531.3	88.8	257.5	0.524
	First-Rob	B&B-2	4.0	5146224.7	2118215.8	75.7	219.5	0.413
		B&B-2'	5.3	5167244.8	2132102.5	75.7	219.3	0.412
		LNSR-2	30.8	2739318.9	488374.4	**143.5**	**417.7**	**1.024**
5	First	B&B-1	8.0	4788771.7	1965952.7	75.7	352.5	0.656
		LNSR-1	**54.8**	2734267.1	484041.2	86.7	405.8	0.809
	First-Rob	B&B-2	5.5	4634154.7	1891387.0	75.7	352.7	0.656
		B&B-2'	8.0	4717497.0	1934862.7	75.7	352.5	0.656
		LNSR-2	33.9	2352468.3	478499.9	**139.6**	**661.7**	**1.599**

Table 2 shows that the robustness of the solutions obtained by LNSR-2 is about two thirds more than the other approaches (for $\#B$ and $\#N$). Regarding the R^s_{Slack} measurement, LNSR-2 achieves results close to double the other approaches. This result indicate that LNSR-2 offers a better distribution of robustness (number of feasible neighbour values) across all the variables.



(a) k=1



(b) k=5

Figure 3 Solutions found over the time(s) and their robustness (Taillard-js-020-15-9 instance).

The results of the scheduling evaluation show a very high difference in the total number of feasible neighbours achieved ($\#N$) when the k is increased (for LNSR-2, it is more than 200 neighbours for each k increment of 2 units, i.e. $k = 3, k = 5$). Such increment is much lower in the random CSPs (see Table 1). We believe that the reason could be associated with the tightness of the instances. The higher it is, the lower the likelihood of finding a great number of neighbours.

We also present Figures 3a and 3b which show the solutions found over time for B&B-2' and LNSR-2 for $k = 1$ and $k = 5$ (for a particular instance and execution). For other instances, executions and k's, the graphs are similar to the presented here. Note that LNSR-2 finds a much higher number of solutions than B&B-2'. The robustness of the solutions found by LNSR-2 increments quickly over time, specially at the beginning of the execution. However, B&B-2' finds solutions that have similar robustness between them. In addition, B&B-2' is unnable to improve the robustness after a short time (less than 200 sec. for k=1 and less than 300 sec. for k=5). We believe that this is because the B&B approaches spend a lot of time searching in the same branches of the search tree while LNSR diversifies more the search by exploring a large number of neighbours of the current most robust solution.

6 Conclusions and Future Work

In this paper, we presented an LNS algorithm with CP and B&B for searching for robust solutions for CSPs (LNSR). LNSR considers the robustness concepts and assumptions as in [7]. And therefore, it computes robust solutions for large-scale CSPs that have a high number of feasible neighbour values (Algorithms 1 and 2, from Section 4). In addition, we presented a robust-value selection heuristic (Algorithm 4, from Section 4) for effectively guiding the search towards more promising branches of the search space (that are more likely to contain more feasible neighbour values solutions). Specifically, the focus is on problems that do not have associated detailed information about the uncertainty and with ordered domains.

In this paper, we used general CSPs instances and scheduling problems as a case study (due to their eligibility for such criteria). The experimental evaluation shows that our LNSR algorithm, combined with our robust value selection heuristic, outperforms the previous approach from the state-of-the-art. This good performance is especially usefull in large-scale problems in which the computation time available for finding a solution is limited and/or when several solutions must be re-computed over time (due to the dynamism associated with the real-world application problems).

As a future work, among others, we would like to explore other domains, such as data centres, sports, etc. In addition, we plan to apply adaptive strategies to tune the neighbourhood size, the variables selection and the limit of failures allowed during the neighbourhood exploration. We believe that it could be useful to propose a variable selection heuristic that considers the robustness of the variables (for example, combining a robustness measurement with *Wdeg*). We will explore ideas such as Cost-Impact based variable selection, We will also consider the combination of several heuristics into a portfolio, in the same vein as previous "adaptive" LNS approaches from the literature.

Furthermore, we would like to explore different ways of applying our LNSR to CSOPs by using multi-objective strategies and computing the Pareto frontier. In addition, we believe that it is also interesting to evaluate the performance of accepting less robust solutions in some iterations of the LNSR algorithm (as proposed in [14, Chapter 4]).

References

1 A. Charnes and W. W. Cooper. Chance-Constrained Programming. *Management Science*, 6(1):73–79, 1959.

2 J. Christopher Beck, T. K. Feng, and Jean Paul Watson. Combining constraint programming and local search for job-shop scheduling. *INFORMS Journal on Computing*, 23(1):1–14, 2011.

3 Aharon Ben-Tal, Laurent El Ghaoui, and Arkadi Nemirovski. *Robust optimization*. Princeton University Press, 2009.

4 Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. *Frontiers in Artificial Intelligence and Applications*, 110:146–150, 2004.

5 Tom Carchrae and J. Christopher Beck. Principles for the design of large neighborhood search. *Journal of Mathematical Modelling and Algorithms*, 8(3):245–270, 2009.

6 Laura Climent, Richard J. Wallace, Barry O'Sullivan, and Eugene C. Freuder. Extrapolating from limited uncertain information in large-scale combinatorial optimization problems to obtain robust solutions. *International Journal on Artificial Intelligence Tools*, 25:1–21, 2016. In this paper a way to use extrapolation to enable Stochastic approach with limited knowledge is treated. Important: Ordered Domains. doi:10.1142/S0218213016600058.

7 Laura Climent, Richard J. Wallace, Miguel A. Salido, and Federico Barber. Robustness and stability in constraint programming under dynamism and uncertainty. *Journal of Artificial Intelligence Research*, 49:49–78, 2014.

8 Thierry Denœux. Belief functions induced by random fuzzy sets: A general framework for representing uncertain and fuzzy evidence. *Fuzzy Sets and Systems*, 1:1–29, 2020.

9 Ivan Dotu, Manuel Cebrián, Pascal Van Hentenryck, and Peter Clote. Protein Structure Prediction with Large Neighborhood Constraint Programming Search. In Peter J. Stuckey, editor, *Principles and Practice of Constraint Programming*, pages 82–96. Springer, 2008.

10 Helene Fargier, Jerome Lang, and Thomas Schiex. Mixed constraint satisfaction: a framework for decision problems under incomplete knowledge. *Proceedings of the National Conference on Artificial Intelligence*, 1(January 1996):175–180, 1996.

11 Markus P J Fromherz. Constraint-based scheduling. *Proceedings of the American Control Conference*, pages 3231–3244, 2001.

12 Luca Di Gaspero, Andrea Rendl, and Tommaso Urli. Balancing bike sharing systems with constraint programming. *Constraints*, 21(2):318–348, 2016.

13 Emmanuel Hebrard. *Robust Solutions for Constraint Satisfaction and Optimisation under Uncertainty*. PhD thesis, University of New South Wales, 2007.

14 Michellgendreauu · Jean-Yvesspotvin. *Handbook of Metaheuristics*. International Series in Operations Research & Management Science, third edit edition, 2010.

15 Christophe Lecoutre and Sebastien Tabary. Abscon 112 : towards more robustness. In *3rd International Constraint Solver Competition*, pages 41–48, 2013.

16 Dario Pacino and Pascal Van Hentenryck. Large neighborhood search and adaptive randomized decompositions for flexible jobshop scheduling. *IJCAI International Joint Conference on Artificial Intelligence*, pages 1997–2002, 2011.

17 Laurent Perron, Paul Shaw, and Vincent Furnon. Propagation guided Large Neighbourhood Search. In Mark Wallace, editor, *Principles and Practice of Constraint Programming*, volume 3258, pages 469–481, Toronto, 2004. Springer.

18 Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1520:417–431, 1998.

19 Michele Surico, Uzay Kaymak, David Naso, and Rommert Dekker. Hybrid Meta-Heuristics for Robust Scheduling. *ERIM Report Research in Management*, 2006.

20 Nihat Engin Toklu. *Matheuristics for Robust Optimization – Application to Real-World Problems*. PhD thesis, Università della Svizzera Italiana, 2014.

21 Behnam Vahdani, Reza Tavakkoli-Moghaddam, Fariborz Jolai, and Arman Baboli. Reliable design of a closed loop supply chain network under uncertainty: An interval fuzzy possibilistic chance-constrained model. *Engineering Optimization*, 45(6):745–765, 2013.

22 Gérard Verfaillie and Narendra Jussien. Constraint solving in uncertain and dynamic environments: A survey. *Constraints*, 10(3):253–281, 2005.

23 Toby Walsh. Stochastic constraint programming. *Proceedings of the 15th Eureopean Conference on Artificial Intelligence*, pages 111–115, 2002.

24 Huayue Wu. *Randomization and Restart Strategies*. PhD thesis, University of Waterloo, 2006.

25 Weixiong Zhang. Branch-and-Bound Search Algorithms and Their Computational Complexity. Technical report, Information Sciences Institute, Marina del Rey, California, 1996.