Journal of Artificial Intelligence Research 49 (2014) 49-78 Submitted 07/13; published 01/14

Robustness and Stability in Constraint Programming under Dynamism and Uncertainty

Laura Climent LCLIMENT@DSIC.UPV.ES
Instituto de Automática e Informática Industrial
Universidad Politécnica de Valencia, Spain.

Richard J. Wallace R.WALLACE@4C.UCC.IE
INSIGHT Center for Data Analytics
Department of Computer Science. University College Cork, Ireland.

Miguel A. Salido MSALIDO@DSIC.UPV.ES
Instituto de Automática e Informática Industrial
Universidad Politécnica de Valencia, Spain.

Federico Barber FBARBER@DSIC.UPV.ES
Instituto de Automática e Informática Industrial
Universidad Politécnica de Valencia, Spain.

Abstract

Many real life problems that can be solved by constraint programming, come from uncertain and dynamic environments. Because of the dynamism, the original problem may change over time, and thus the solution found for the original problem may become invalid. For this reason, dealing with such problems has become an important issue in the fields of constraint programming. In some cases, there is extant knowledge about the uncertain and dynamic environment. In other cases, this information is fragmentary or unknown. In this paper, we extend the concept of *robustness* and *stability* for Constraint Satisfaction Problems (CSPs) with ordered domains, where only limited assumptions need to be made as to possible changes. We present a search algorithm that searches for both robust and stable solutions for CSPs of this nature. It is well-known that meeting both criteria simultaneously is a desirable objective for constraint solving in uncertain and dynamic environments. We also present compelling evidence that our search algorithm outperforms other general-purpose algorithms for dynamic CSPs using random instances and benchmarks derived from real life problems.

1. Introduction

Constraint programming is a powerful tool for solving many artificial intelligence problems that can be modeled as CSPs. Much effort has been spent on increasing the efficiency of algorithms for solving CSPs, as reflected in the literature. However, most of these techniques assume that the set of variables, domains and constraints involved in the CSP are known and fixed when the problem is modeled. This is a strong limitation when we deal with real life situations because these problems may come from uncertain and dynamic environments. Due to the dynamism in the environment, both the original problem and its corresponding modeled CSP may evolve. In addition, since the real world is uncertain in its nature, information about the dynamism of the environment may be incomplete, erroneous or even may not exist. In such situations, a solution that holds for the original model can become invalid after changes in the original problem.

The approaches that deal with this situation can be classified as: (i) *reactive approaches*, whose main objective is to obtain a new solution as similar as possible to the previous solution (the solution found before the changes occurred) in a efficient way, and (ii) *proactive approaches*, which use knowledge about possible future changes in order to avoid or minimize their effects (for a survey see Verfaillie & Jussien, 2005). Thus, proactive approaches are applied before the changes occur, while reactive approaches are only applied when the changes invalidate the original solution.

Reactive approaches re-solve the CSP after each solution loss, which consumes computational time. That is a clear inconvenience, especially when we deal with short-term changes, where solution loss is very frequent. In addition, in many applications, such as online planning and scheduling, the time required to calculate a new solution may be too long for actions to be taken to redress the situation. In addition, the loss of a solution can have several negative effects in the modeled situation. For example, in a task assignment of a production system with several machines, it could cause the shutdown of the production system, the breakage of machines, the loss of the material/object in production, etc. In a transport timetabling problem, a solution loss, due to some disruption at a point, may produce a delay that propagates through the entire schedule. All these negative effects will probably entail an economic loss as well.

Proactive approaches try to avoid the drawbacks just stated and, therefore, they are highly valued for dealing with problems in uncertain and dynamic environments. Given the advantages that proactive approaches potentially offer, in this paper we restrict ourselves to this approach. Heretofore two main types of proactive approaches have been considered, which can be distinguished on the basis of the characteristics of the solutions that they obtain, which are called *robust* and *flexible* (see Section 2). In an important survey on constraint solving in uncertain and dynamic environments (Verfaillie & Jussien, 2005), the authors mention the possibility of developing proactive strategies that combine the solution features of robustness and flexibility. They state: "The production of solutions that are at the same time robust and flexible, that have every chance to resist changes and can be easily adapted when they did not resist, is obviously a desirable objective." In this paper, we present an algorithm that meets the objective of combining solution robustness and *stability*. The solution feature of stability is a special case of flexibility.

Many proactive approaches proposed in the literature assume the existence of knowledge about the uncertain and dynamic environment (see Section 3). In these cases it is difficult to characterize the robustness of the solutions when detailed information about possible future changes is not available. We consider situations where there is an added difficulty stemming from the fact that the only limited assumptions about changes can be made. Our discussion focuses on CSPs with ordered and discrete domains that model problems for which the order over the elements of the domain is significant. In these cases, a common type of change that problems may undergo is restrictive modifications over the bounds of the solution space. These assumptions and their motivations were introduced by Climent et al. (2013). Moreover, examples of real life problems that exhibit this type of dynamism were described, specifically, temporal reasoning-based problems, spatial and geometric reasoning problems, and design problems. In temporal problems, delays are an inherent feature, which implies restrictive modifications of the bounds involved with such disruptions. For instance, Fu, Lau, Varakantham, and Xiao (2012) stated that unexpected external events such as manpower availability, weather changes, etc. lead to delays or advances in completion of activities in scheduling problems. In spatial and geometric reasoning problems, the constraints can be readjusted due to measurement errors. The latter can also occur in design problems, in which the data is not completely certain.

In this paper, we present an algorithm that searches for solutions to CSPs with ordered domains, which are robust and are also stable because they can often be repaired using a value of similar magnitude if they undergo a value loss. The paper is organized as follows. The next section recalls some general definitions. Section 3 gives a brief account of earlier proactive procedures. Section 4 presents a new conception of robustness and stability when there exists an order over the elements of the domain. Sections 5 and 6 describe the main objective for finding solutions that meet the stability and robustness criteria simultaneously. Then, in Section 7 the search algorithm that meets these objectives is explained. Section 8 presents a case study of scheduling problems. Section 9 describes experiments with various types of CSPs, showing the effectiveness of the present approach for finding solutions that are both stable and robust. Section 10 gives conclusions.

2. Technical Background

In this section we give some basic definitions that are used in the rest of the paper, following standard notations and definitions in the literature.

Definition 2.1 *A Constraint Satisfaction Problem (CSP) is represented as a triple $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where $\mathcal{X}$ is a finite set of variables $\mathcal{X} = \{x_1, x_2, ..., x_n\}$, $\mathcal{D}$ is a set of domains $\mathcal{D} = \{D_1, D_2, ..., D_n\}$ such that for each variable $x_i \in \mathcal{X}$ there is a set of values that the variable can take, and $\mathcal{C}$ is a finite set of constraints $\mathcal{C} = \{C_1, C_2, ..., C_m\}$ which restrict the values that the variables can simultaneously take. We denote by $\mathcal{DC}$ the set of unary constraints associated with $\mathcal{D}$.*

Definition 2.2 *A tuple t is an assignment of values to a subset of variables $\mathcal{X}_t \subseteq \mathcal{X}$.*

If a tuple *t* is feasible we call it s. This means that s is an assignment of the domain values to some variables that does not violate any constraint. If s is a complete assignment (it involves all the variables of the CSP), then it is a solution of the CSP. $\mathcal{X}_s$ is the subset of variables that are involved in s. Then $\mathcal{X} \backslash \mathcal{X}_s$ is the set of unassigned variables in s. The value assigned to a variable x in s is denoted as $s(x)$. In addition, we denote $D_s(x) \subseteq D(x)$ to the subset of domain values of the variable x that are consistent with s.

The number of possible tuples of a constraint $C_i \in \mathcal{C}$ is composed of the elements of the Cartesian product of the domains of $var(C_i)$: $\prod_{x_j \in var(C_i)} D_j$, where $var(C_i) \subseteq \mathcal{X}$ is the set of variables involved in C_i (scope of C_i).

Definition 2.3 *The tightness of a constraint is the ratio of the number of forbidden tuples to the number of possible tuples. Tightness is defined within the interval [0,1].*

Inferential processes for CSPs narrow the search space of possible partial solutions. In this work we use one of the most known and used consistency procedure: *arc-consistency*.

Definition 2.4 *A CSP is arc-consistent (Mackworth, 1977a) iff for any pair of constrained variables x_i and x_j, for each value a in D_i there exists at least one value b in D_j such that the partial assignment $(x_i = a, x_j = b)$ satisfies all the constraints related to both x_i and x_j. Any value in* *the domain of a variable which is not arc-consistent can be eliminated as they can not be part of any solution. The domain of a variable is arc-consistent iff all values are arc-consistent. Thus, a problem is arc-consistent iff all its arcs are arc-consistent:*

$$\forall\, C_{ij} \in \mathcal{C}, \forall a \in D(x_i), \exists b \in D(x_j)\text{: } a \text{ and } b \text{ satisfy } C_{ij}.$$

In the following, several properties associated with the solutions of problems that come from dynamic environments are defined.

Definition 2.5 *The most robust solution of a CSP within a set of solutions is the one with the highest likelihood of remaining a solution after a given set of changes in the CSP.*

Definition 2.6 *A flexible solution is anything (a partial solution, complete solution, conditional solution, set of solutions, etc.) that, in case of change, can be easily modified to produce a solution to the new problem (Verfaillie & Jussien, 2005).*

A more specific concept of flexibility is the concept of stability.

Definition 2.7 *A solution $s1$ is more stable than another solution $s2$ if and only if, in the event of a change that invalidates them, a closer alternative to $s1$ than to $s2$ exists (modified from the work presented by Hebrard, 2006).*

The main difference between Definition 2.7 and Definition 2.6 is that the former introduces the concept of 'closer' solution. The measurement of this closeness is made by calculating distances between solutions. More concrete information about the distance equations is explained in following sections. We would like to remark that Definition 2.5 does not consider the alterations in the original solution but only its resistance to changes in the problem. On the other hand, Definition 2.6 and Definition 2.7 do consider changes to the original solution when a new solution is produced after a change in the problem.

3. Related Work: Proactive Approaches

Several approaches have been proposed in the past for handling this type of problem, which can be classified based on the kind of solutions they obtain. Thus, there are techniques that search for robust solutions and others that search for flexible solutions (for a survey see Verfaillie & Jussien, 2005). In this section we describe some techniques that search for robust solutions and their limitations. Then we discuss a technique that searches a certain type of stable solutions: *super-solutions*.

3.1 Searching for Robust Solutions

Many earlier approaches that search for robust solutions use additional information about the uncertain and dynamic environment in which the problem occurs, and most often this involves probabilistic representations. In one example of this type, information is gathered in the form of penalties when values have been invalidated after changes in the problem (Wallace & Freuder, 1998). Nevertheless, in the Probabilistic CSP model (PCSP) (Fargier & Lang, 1993), there exists information associated with each constraint, expressing its probability of existence. Other techniques focus on the dynamism of the variables of the CSP. For instance, the Mixed CSP model (MCSP) (Fargier, Lang, & Schiex, 1996) and the subsequent Uncertain CSP model (UCSP) (Yorke-Smith & Gervet, 2009) consider the dynamism of certain *uncontrollable* variables that can take on different values of their uncertain domains. A related model, uses Simple Temporal Networks, but adds data about the time uncertainties and preferences, which represent the starting or ending times of events (STP-PUs) (Rossi, Venable, & Yorke-Smith, 2006). The Stochastic CSP model (SCSP) (Walsh, 2002) also considers probability distributions associated with the uncontrollable variables. The Branching CSP model (BCSP) considers the possible addition of variables (with a certain associated gain) to the current problem (Fowler & Brown, 2000).

In most of these models, the form of the algorithm is dependent on detailed knowledge about the dynamic environment. For this purpose, a list of possible changes is required or there is an explicit representation of uncertainty, often in the form of an associated probability distribution. As a result, these approaches cannot be used if the necessary information is unknown. In many real problems, however, knowledge about possible further changes is either limited or non-existent. Hence, there is an important need for techniques that find robust solutions in this kind of environment.

For instance, Climent et al. (2013) cope with CSPs that model problems for which the order over the domain elements is significant. Specifically, these CSPs are modeled as Weighted Constraint Satisfaction Problems (WCSPs) (Larrosa & Schiex, 2004) by penalizing valid tuples based on their *coverings*. Instead of requiring extra detailed dynamism information, the authors only make limited assumptions concerning the changes that might occur, which is related to the nature of CSPs with ordered domains. Specifically, dynamism is assumed to take the form of restrictions on the bounds of the solution space. In this paper, we make the same assumptions about the dynamism. The previous WCSP modeling approach computes robustness based on feasible neighbours that compose a covering which surrounds the analyzed value with respect to each constraint boundary. Thus, in cases in which a neighbour is feasible with respect to one bound but is not for another bound, this neighbour is not feasible in the solution space. For this reason, this approach obtains robustness approximations in problems in which there is a high relation between constraints. On the other hand, the algorithm described in this paper computes feasible assignments with respect to the entire solution space, which avoids the weakness of the WCSP modeling approach explained above. A comparison between these approaches is found in Section 9.

3.2 Searching for Super-Solutions

Techniques that search for stable solutions of a certain type, which are denoted as super-solutions, were presented by Hebrard (2006). The goal is to be able to repair an invalid solution after changes occur, with minimal changes that can be specified in advance. Since this is another approach that does not require detailed additional information about changes in a problem, it is of interest to compare it to the search algorithm introduced in this paper.

Definition 3.1 *A solution is an (a, b)-super-solution if the loss of values of at most a variables can be repaired by assigning other values to these variables and changing the values of at most b other variables (Hebrard, 2006).*

For CSPs, a major focus has been on finding (1, 0)-super-solutions. This is because of the high computational cost of computing $b > 0$ or $a > 1$. This is one of the reasons why we analyze this particular super-solution case in this paper. The other reason is given by Verfaillie and Jussien (2005), where the authors state that a desirable objective is to “limit as much as possible changes in the produced solution", which motives the search of (a, 0)-super-solutions. In general, it is unusual to find (1, 0)-super-solutions where all variables can be repaired. For this reason, Hebrard (2006) also developed a *branch and bound*-based algorithm for finding solutions that are close to (1, 0)-super-solutions, i.e., where the number of repairable variables is maximized (also called maximizing the (1, 0)-repairability).

4. Extending Robustness and Stability to CSPs with Ordered Domains

In this section we extend the original definition of solution robustness (Definition 2.5) and solution stability (Definition 2.7) to consider CSPs with ordered domains, where only limited assumptions are made about changes in the problem that are derived from their inherent structure. Given this framework and therefore the existence of a significant order over the values in the domains, it is reasonable to assume that the original bounds of the solution space can only be restricted or relaxed, even if this does not cover all possible changes. The bounds of the solution space are delimited by the domains and constraints of the CSP. Note that the possibility of solution loss only exists when changes over the original *bounds* of the solution space are restrictive. For this reason, a solution that is located farther away from the bounds is more likely to remain a solution. Given these assumptions, we specialize Definition 2.5 for this framework as follows.

Definition 4.1 *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.*

Furthermore, the definition of stable solutions for CSPs with ordered domains can be made more precise because it is possible to define a more specific notion of closeness between two solutions due to the existent order over the domain values. Hebrard (2006) measures the level of dissimilarity of two solutions by counting the number of variables that take different values in both solutions, i.e., the Hamming distance ($\sum_{i=1}^{n}(s1_i \neq s2_i)$). Later, Hebrard, O'Sullivan, and Walsh (2007) consider another similarity measure: the Manhattan distance. This measure uses the sum of the absolute difference of values (of each variable) for both solutions ($\sum_{i=1}^{n}|s1_i - s2_i|$). Note that unlike Hamming distance, Manhattan distance requires an order over the elements in order to calculate the absolute difference of the values. In the following definition, we apply the Manhattan distance to the notion of stable solutions for CSPs with ordered domains.

Definition 4.2 *Given an order relationship over the values of a set of solutions, a solution $s1$ is more stable than another solution $s2$ iff, in the event of a change that invalidates them, there exists an alternative solution to $s1$ with lower Manhattan distance than the Manhattan distance of any alternative solution to $s2$.*

Furthermore, we present an extension of Definition 3.1 for CSPs with ordered domains by fixing a maximum Manhattan distance between the original solution and the repaired solution, which is called c.

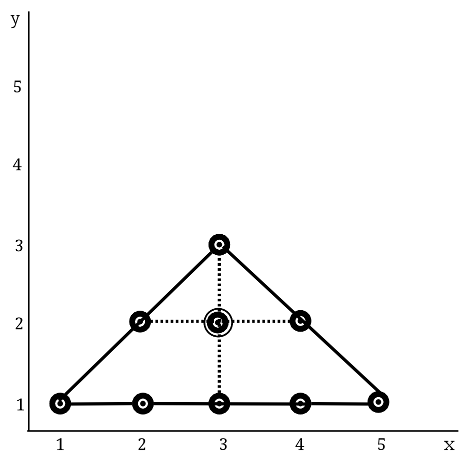
Definition 4.3 *A solution is an (a, b, c)-super-solution if the loss of values of a variables at most, can be repaired by assigning other values whose Manhattan distance with respect the original values is lower or equal to c, and this involves changing the values of b variables at most.*

The above definition also holds for the (1, 0, c)-super-solutions and the (1, 0, c)-repairability, which are the main focus of the stability analysis in this paper.

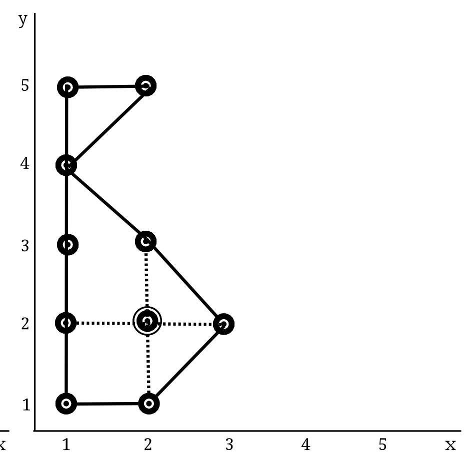
5. Searching for Robust and Stable Solutions: General Main Objective

In order to find robust and stable solutions for CSPs with ordered domains under our assumptions, we combine the robustness and stability criteria presented in Section 4. As mentioned, calculating distances is required for the search of robust solutions in this framework. However, a measure of the distance from the dynamic bounds of the solution space is not always obvious or easy to derive, since the bounds are delimited by the domains and the constraints of the CSP, and the latter may be extensionally expressed. However, some deductions about minimum distances to the bounds can be made based on the feasibility of the neighbours of a solution. This idea is first motivated with a very simple example and then it is formalized.

Example 5.1 *Figure 1 shows two solution spaces (one convex and the other non-convex) whose dynamic bounds are marked by contiguous lines. The most robust solutions according to Definition 4.1 are highlighted. Note that there are two contiguous feasible neighbours on both sides of each assignment (discontinuous lines).*



(a) Convex Solution Space



(b) Non-convex Solution Space

Figure 1: Most Robust solutions for different solution spaces.

From this example, we can conclude that *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 at distances lower or equal to d from s in this direction are feasible*. Therefore, the number of feasible contiguous surrounding neighbours of the solution is a measure of the robustness of the solution in the face of restrictive changes that affect the original bounds of the solution space (see Definition 4.1). In addition to fulfilling the main objective of finding solutions whose values have a high number of feasible neighbours close to each assignment, this criterion can be used to obtain solutions with high stability. This is because if the value assigned to a variable has at least one of these feasible neighbour values, then this variable is repairable. That is, if its assigned value is lost, it can easily be repaired by assigning the neighbour value (since this value is consistent with the rest of the values of the assignment). Regarding the stability notion of Definition 4.2, note that the difference between the lost value and the repairable value is very low, since they are immediate neighbours. In fact, their value difference is one, which is the minimum possible.

The set of feasible contiguous neighbour values of the value v that have differences not greater than k with respect to v in increasing, or decreasing, or both directions with respect to the order relationship is denoted as $\mathcal{N}_k(x, v, s, \oplus)$. Value v is a feasible value for variable x in the feasible partial/complete assignment s. Here, when we say that other values are feasible, we mean that they are also feasible with respect to s. (Recall that we use $D_s(x) \subseteq D(x)$ the subset of domain values that are consistent with the feasible partial assignment s.) The list of operators $\oplus$ is composed of a set of paired elements, or operator pairs. Each operator pair is denoted as $\oplus_i \in \{\{>, +\}, \{<, -\}\}$. The operator pairs fix the order directions to analyze. Thus, the set $\{>, +\}$ refers to values greater than v (increasing direction) and the set $\{<, -\}$ refers to values lower than v (decreasing direction). For each operator pair, the operator in position j is referenced as $\oplus_{ij}$. For instance, if the list of operators is $\oplus = \{\{>, +\}, \{<, -\}\}$, the operator pair $\oplus_1$ references $\{>, +\}$ and the operator $\oplus_{12}$ references the operator $+$. Given this notation, we define $\mathcal{N}_k(x, v, s, \oplus)$ as:

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

The first condition of Equation 1 ensures that the value w is greater or lower than v according to the operator $\oplus_{i1} \in \{>, <\}$ and that the distance between these values is less or equal to k. The second condition ensures that all values that are closer to v than w are also feasible values for s. If at least one of them is not, the value w cannot belong to $\mathcal{N}_k(x, v, s, \oplus)$. As mentioned previously, the set of feasible neighbours of a value has to be contiguous. Otherwise, there is an infeasible space between this value and another feasible value. For instance, in Figure 1(b) the value 5 does not belong to $\mathcal{N}_k(y, 2, \{x = 2\}, \oplus)$ for any $\oplus$ or k because the value 4 is not a feasible value and therefore it is outside the bounds of the solution space.

For the general case of CSPs with ordered domains in which we assume that all the bounds are dynamic, the desirable objective is to find contiguous surrounding feasible neighbours on both sides. For this reason $\oplus = \{\{>, +\}, \{<, -\}\}$. For this list of operator pairs, the last condition of Equation 1 checks that all the values in both directions that are closer to v than w, are also feasible values for s. For instance, in Figure 1(b), $\mathcal{N}_k(y, 2, \{x = 2\}, \{\{>, +\}, \{<, -\}\}) = \{1, 3\}$ for any k value. Note that these neighbours are on both sides the value 2 with respect to the y axis. In Section 8, we will show a specific case for which it is desirable to apply only one operator pair due to the nature of the problem.

To apply Equation 1 to domains that are not ordered in $\mathbb{Z}$, a monotonic and order-preserving function has to be applied. For instance, if we consider $\mathcal{D} = \{freezing, cold, mild, warm, hot, boiling\}$, a monotonic function that assigns greater values to values with higher temperatures could be defined. For example, $f(freezing) = 1, f(cold) = 2, f(mild) = 3, f(warm) = 4, f(hot) = 5$ and $f(boiling) = 6$.

6. Objective Function

In Section 5 we stated that the main desirable objective for a selected value is to have as many contiguous feasible neighbours in a certain direction, because they determine the minimum distance of this value from the bound in this direction. For approximating the distance of several values assigned (partial or complete assignment), we compute the number of neighbours of each value. Therefore, we define as an objective function of our search algorithm the sum of the size of $\mathcal{N}_k(x, v, s, \oplus)$ (denoted $|\mathcal{N}_k(x, v, s, \oplus)|$) for each variable $x \in \mathcal{X}$. If s is an incomplete assignment, we calculate the maximum $|\mathcal{N}_k(x, v, s, \oplus)|$ for each $v \in D_s(x)$ of each unassigned variable $x \in \mathcal{X} \backslash \mathcal{X}_s$ (upper bound). Note that the maximum size of the set of neighbour values for each variable is $| \oplus | * k$, where $| \oplus |$ is the number of pair operators. Thus, the maximum size of the set of neighbour values is $2k$ if $\oplus$ is composed of two operator pairs or k if $\oplus$ is composed of only one operator pair. Note also that it is not necessary to check all the values of $D_s(x)$ if, for at least one of them, the size of the set is the maximum possible. In the following equation, we formalize the objective function that is used by our search algorithm.

$$f(s, k, \oplus) = \{ \sum_{x \in \mathcal{X} \backslash \mathcal{X}_s} max\{|\mathcal{N}_k(x, v, s, \oplus)|, \forall v \in D_s(x)\} + \sum_{y \in \mathcal{X}_s} |\mathcal{N}_k(y, s(y), s, \oplus)|\} \tag{2}$$

For Example 5.1, for the most robust solutions of both Figures 1(a) and 1(b) (highlighted solutions) $f(s, k, \{\{>, +\}, \{<, -\}\}) = 4$, for $k \geq 1$, since every value assigned to each solution has two contiguous neighbours on both sides.

Next, we give a formal rationale for using the total number of neighbours of the solution (sum of feasible surrounding neighbours of each value of the solution) as a measure of robustness.

For $k = 1$, in a convex solution space, each value has either zero, one or two feasible neighbours. Here we can discount the case of zero neighbours because if an assignment has zero feasible neighbours, then it must be part of a singleton domain, and it will be part of all solutions. So we need only consider values with one or two feasible neighbours.

In this case, a solution with a greater sum is one whose assignments have more feasible neighbour pairs. This can be easily seen if we consider the difference between a solution all of whose values have only one feasible neighbour and any other solution; this difference will be equal to the number of feasible neighbour pairs associated with the latter's assignments.

Proposition 1. *If we assume that having two feasible neighbours confers greater robustness than having one and that the probabilities of single changes are independent, then a solution with a greater feasible neighbour-sum than another will also be more robust, and vice versa.*

In the non-convex case, it is unfortunately possible for one assignment to have zero feasible neighbours, while other assignments to the same variable have one or two. In this case, we cannot assume Proposition 1. However, as the number of variables in the problem increases, it becomes increasingly unlikely that a variable with an assignment having zero feasible neighbours will be associated with the largest neighbour-sum for the remaining variables.

Regarding the measure of stability, a solution that maximizes the (1, 0, k)-repairability (see Definition 4.3) also maximizes the number of variables that can be repaired by a neighbour value at a distance less or equal to k (without modifying any other variable). However, to obtain robust solutions we maximize the sum of neighbour values of each value of the solution. Note that even if both maximization criteria are not identical, as mentioned, when the number of variables in the problem increases, it becomes increasingly unlikely that a non-repairable variable will be associated with the largest neighbour-sum for the remaining variables. So in this work we will use the same technique for finding robust and stable solutions for CSPs with ordered domains. Nevertheless, the basic units of measure for both criteria are different.

7. Search Algorithm

In this section we present an algorithm for finding robust and stable solutions according to the main objective described in Section 6. For this purpose, we have incorporated this optimization criterion into a Branch & Bound algorithm (Algorithm 1) that maximizes the objective function $f(s, k, \oplus)$ (see Equation 2). As mentioned, this function sums $|\mathcal{N}_k|$ of each assigned variable and the maximum possible $|\mathcal{N}_k|$ of each unassigned variable. Note that this computation is an upper bound of the final total number of feasible contiguous neighbours of the solution.

Algorithm 1 (B&B-$\mathcal{N}_k$) is an 'anytime' algorithm that uses an inference process and prunes the branches whose objective function value is lower or equal to the current maximum function value obtained, referred to as lb (lower bound). The process stops when all the branches have been explored or pruned, providing the solution s with the maximum $f(s, k, \oplus)$. On the other hand, we can limit the search time and therefore the quality of the best solution found by fixing a time cutoff. Of course, the more time Algorithm 1 spends searching, the more robust and stable the solution provided can be. In addition, we compute the maximum possible objective function value, which is the maximum number of neighbours for each variable multiplied by the number of variables of the CSP, denoted as ub (upper bound). Thus, if the objective function value of a new solution found is equal to ub, the algorithm stops, since this solution is optimal.

Algorithm 1: B&B-$\mathcal{N}_k$: Branch & Bound anytime algorithm

Data: $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle, \oplus, k, scale, m,$ *time cutoff* (optional)
Result: $s, \mathcal{N}_k, lb$
$s \leftarrow \emptyset$; // Partial assignment
$\mathcal{X}_s \leftarrow \emptyset$; // Set of variables assigned
$\mathcal{N}_k \leftarrow \emptyset$; // Set of contiguous surrounding neighbours
$lb \leftarrow -1$; // Maximum $f(s, k, \oplus)$ for the solutions
$ub \leftarrow |\oplus| * k * |\mathcal{X}|$;
$i \leftarrow 1$;
GAC3-$\mathcal{N}_k(P, s, \mathcal{X}_s, \mathcal{N}_k, \oplus, k, lb)$;
repeat
  if *restarting-scratch* $\wedge$ *new solution found* **then**
    $i \leftarrow 1$;
  $C \leftarrow scale * m^i$; //number of fails cutoff
  $i \leftarrow i + 1$;
until *time cutoff* $\vee$ ***not*** *MGAC3-*$\mathcal{N}_k(P, s, \mathcal{X}_s, \mathcal{N}_k, \oplus, k, lb, 0, C, ub)$;

We have implemented the Branch & Bound algorithm using a Geometric restart strategy (Walsh, 1999) in order to reduce the repetition of fails in the search due to very early wrong assignments (*thrashing*). Thus, each time the number of failures (referenced as nbF) reaches the number-of-fails cutoff value condition (C) that is checked in Algorithm 3, the algorithm restarts the search from scratch, except for the constraint weights stored by the $dom/wdeg$ heuristic variable selection (Boussemart et al., 2004). The value of the number of fails cutoff is increased geometrically in Algorithm 1 according to the scale factor (referred to as $scale$) and the multiplicative factor (referred to as m). We have implemented two different options to carry out after a solution is found. In the first, called *restarting-completion*, when the first solution is found, the algorithm continues to search until completion (this is done by assigning a huge number representing ∞ to the number of fails cutoff). In the second option, called *restarting-scratch*, after each solution found, the algorithm restarts the search from scratch and also restarts the number of fails cutoff computation (the constraint weighs remain the same). For instances with very large domain sizes, this restarting option can be effective because it avoids spending a large amount of time in a specific branch. The latter happens when Algorithm 1 checks many domain values of variables located at low levels of the search tree, because the objective function of the partial assignment is better than the current maximum (lb). In this case, if there exists a time cutoff, Algorithm 1 could not analyze other branches of the tree that may contain solutions of better quality.

The inference process is carried out by Algorithm 2 (GAC3-$\mathcal{N}_k$), which is an extension of the well-known AC3 (Mackworth, 1977b) that performs Generalized Arc Consistency (GAC) (Mohr & Henderson, 1986; Bessiere, 2006). Some specific notation has been included, as $var(c)$, which is the scope of $c \in \mathcal{C}$. The original `seekSupport` function of GAC3 searches for a support for each domain value. We have modified this function slightly by providing the set of values to be analysed as a parameter of the function. Thus, if any of these values is deleted because there does not exist any consistent support with respect the partial assignment, `seekSupport` returns `false`. This function is first called with the values of the domain of the variables (for checking if the partial assignment s is GAC3) and later with $\mathcal{N}_k$ just for assigned variables (for checking if each $\mathcal{N}_k(x, s(x), s, \oplus)$ is GAC3 with respect s). In order to ensure the contiguity of the values in $\mathcal{N}_k$, Algorithm 2 checks the consistency of subsets of $\mathcal{N}_i \subseteq \mathcal{N}_k$, where i is equal to one initially, and it is increased by one unit until at least one of the values of $\mathcal{N}_i$ is inconsistent or i reaches the value of k. The complexity of updating $\mathcal{N}_i$ can be reduced to $| \oplus | * i$ if the domains are ordered. Note that in the case where both greater and lower values are candidates to be in the set, the updating cost is $2 * i$. After composing the set of contiguous neighbour values that are GAC3 with respect s, Algorithm 2 analyzes if the objective function $f(s, k, \oplus)$ is greater than lb. If it is not, or s is not GAC3, it returns `false`.

Algorithm 3 (MGAC3-$\mathcal{N}_k$) performs a Maintaining GAC3 procedure by assigning to each variable $x \in \mathcal{X}$ a new value $v \in D(x)$, until the value selected is GAC3-$\mathcal{N}_k$ with respect s. We have implemented two value selection heuristics: lexicographical order and selection of the value that maximizes $|\mathcal{N}_k(x, v, s, \oplus)|$, starting from intermediate values. There are some real life problems for which the lexicographical selection order is effective in finding feasible solutions quickly. An example is scheduling problems, whose domain values represent time units; hence the importance of selecting low values in order not to exceed the maximum fixed makespan. However, if it is not important to select low values, the heuristic that starts with intermediate values may offer better results because it is selecting values that maximize the objective function at the current node of the search tree. Furthermore, since search starts with intermediate values, the likelihood of selecting values located far from the domain bounds is higher.

Algorithm 3 is also responsible for updating the set of assigned variables $\mathcal{X}_s$, the partial assignment s and the maximum objective function value lb (for each solution found). Furthermore, it stores the domains and the set of neighbours of all the variables before making an assignment. Note that after a variable x is assigned, $D(x)$ contains a single value that is the value assigned to x. If Algorithm 2 (GAC3-$\mathcal{N}_k$) returns `false`, then Algorithm 3 (MGAC3-$\mathcal{N}_k$) carries out the backtracking process and also restores the domains and set of neighbours of all the variables.

To reduce computational time when we deal with CSPs with convex domains, we have implemented Bounds Arc Consistency for discrete CSPs (Lhomme, 1993). The main feature of this

Algorithm 2: GAC3-$\mathcal{N}_k$: Global Arc Consistency algorithm

Data: $P, s, \mathcal{X}_s, \mathcal{N}_k, \oplus, k, lb, nbF$
Result: $\mathcal{D}, \mathcal{N}_k, nbF$
$Q \leftarrow \{(x, c), \forall c \in \mathcal{C}, \forall x \in var(c)\}$ // $var(c)$ is the scope of c
while $Q \neq \emptyset$ **do**
  $(x, c) \leftarrow takeElement(Q)$;
  $seek\mathcal{D} \leftarrow$ `seekSupport`$(x, D(x), c)$; // Found support for all $D(x)$ for c?
  if $D(x) = \emptyset$ **then**
    $nbF \leftarrow nbF + 1$; // number of failures
    return `false`
  if ***not*** $seek\mathcal{D}$ **then**
    $Q \leftarrow Q \cup \{(y, c'), \forall c' \in \mathcal{C} \wedge c' \neq c \wedge \forall x, y \in var(c') \wedge x \neq y\}$
  if $x \in \mathcal{X}_s$ **then**
    $i \leftarrow 1$;
    repeat
      update $\mathcal{N}_i(x, s(x), s, \oplus)$ applying Equation 1;
      $seek\mathcal{N} \leftarrow$ `seekSupport`$(x, \mathcal{N}_i(x, s(x), s, \oplus), c)$;
      $i \leftarrow i + 1$;
    until $seek\mathcal{N} =$ `false` $\vee\, i > k$;
    $\mathcal{N}_k(x, s(x), s, \oplus) \leftarrow \mathcal{N}_i(x, s(x), s, \oplus)$
return $f(s, k, \oplus) > lb$ // See Equation 2

consistency technique is that the arc consistency is restricted with respect to the bounds of each convex domain. Thus, including it in the search algorithm only affects to the `seekSupport` function, which instead of seeking for a support for all the set of values, just checks the minimum and maximum bounds. Note that this implementation is not necessary for the search of robust and stable solutions; however it allows a significant reduction of the search time. We only apply bounds consistency to the tentative values of the assignment but not to their set of neighbours, since they require a complete consistency check. Otherwise there could exist infeasible gaps, which would break the contiguity requirement that ensures minimum distances to the bounds.

8. Case Study: Searching for Robust and Stable Schedules

There are some types of real life problems whose structure can provide us with specific information about their dynamism. In this section we analyze a well known type of problem from the literature: scheduling problems. These problems can be converted into satisfiability problems by fixing a maximum makespan, and they can then be modeled as CSPs. The CSP modeling usually consists of associating the start or end time of each task with a particular variable (in this paper we use the start time). The domain associated with a variable represents the possible time units, and by means of them it is possible to fix a maximum desired makespan. Finally, the duration of the tasks and their order (if it exists) can be fixed by means of CSP constraints.

In this section, we will first explain some robustness scheduling measurement units, and then we describe the objective function for CSPs that model scheduling problems and give an example of its application.

Algorithm 3: MGAC3-$\mathcal{N}_k$: Maintaining Global Arc Consistency

```
Data: P, s, X_s, N_k, ⊕, k, lb, nbF, C, ub
Result: s, N_k, lb
select x ∈ X\X_s; // dom/wdeg heuristic
X_s ← X_s ∪ x;
save D and N_k;
while D(x) ≠ ∅ ∧ nbF < C do
    select min(v) ∈ D(x); // Heuristic 1: lexicographical value order
    select v ∈ D(x), max{|N_k(x, v, s, ⊕)|} starting by intermediate values; // Heuristic 2
    s ← s ∪ {x = v}
    D(x) ← v;
    if GAC3-N_k(P, s, X_s, N_k, k, lb, nbF) then
        if X_s = X then
            // New solution found
            lb ← f(s, k, ⊕);
            if lb = ub then
                return true // Best possible sum achieved
            C ← ∞; // restarting-completion
            return false // restarting-scratch
        if MGAC3-N_k(P, s, X_s, N_k, k, lb, nbF, C, uB) then
            return true
    restore D\D(x) and N_k;
    s ← s\{x = v};
X_s ← X_s\x;
return false
```

8.1 Robustness Measurement in Scheduling

In this section, we introduce several criteria for measuring the scheduling robustness. For such purpose, we use the terms buffers and slack to refer to the spare time between related tasks. There are two main factors that enhance the capability of a schedule to absorb unexpected delays in its activities: the number of buffers and their duration. Ideally, according to the robustness criterion, a buffer time should be as long as possible because the longer it is, the longer are the delays that is able to absorb. For this reason another straight-forward robustness measurement was proposed by Leon et al. (1994) as the slack average in the schedule. The combination of the duration of the buffers and their distribution across the schedule provides a more accurate robustness measure denoted as R^s_{slack}. It is a slight variant of a measure introduced by Surico et al. (2008) that consists in maximizing the slack average (shorted as avg) and minimizing their standard deviation (shorted as std) for a schedule s. For regulating the importance of the standard deviation term, the authors use a parameter called α, which can take any value in the interval [0.2,0.25], according to the authors considerations.

$$R^s_{slack} = avg(slack) - \alpha\ std(slack) \tag{3}$$

Another means of measuring the robustness of a system, defined by Kitano (2007) is related to its resistance to perturbations having a certain probability of occurrence. This approach was extended by Escamilla et al. (2012) to scheduling problems in which the probabilities of task delays are unknown. The robustness measure is denoted as $R^s_{F,Z}$, where Z is the discrete set of unexpected delays in the duration of tasks, F measures whether the schedule s is still feasible after the disruption ($F(z) = 1$ when is satisfiable, otherwise $F(z) = 0$) and $p(z) = \frac{1}{|z|}, \forall z \in Z$ is the probability for an instance $z \in Z$ (i.e., all delays are considered to have the same probability of occurrence).

$$R^s_{F,Z} = \sum_{z \in Z} p(z) * F(z) \tag{4}$$

8.2 Objective Function for Scheduling

In CSP models of scheduling problems, the fact that domain values represent time units has implications with respect to measures of robustness and stability. For these problems, when a value of the solution is lost, lower values cannot be used for replacing this unfeasible value because they represent time units that have already taken place. Thus, if there is an incident, and the time point t is not available, neither are the values lower than t. Therefore, having lower feasible neighbours does not improve the robustness nor the stability of a solution of a CSP that models a scheduling problem (since they cannot absorb delays nor be used as repairable values). Given these characteristics, the main desirable objective is to search for neighbours greater than the value assigned. To do this, we fix the set of operators to $\oplus = \{\{>, +\}\}$ for scheduling problems. This is illustrated below.

Example 8.1 *We consider a toy scheduling problem with two tasks: T_0 and T_1. Both have a duration of two time units and they must be executed in the order listed. The maximum makespan allowed is six time units. In Figure 2 we can see the associated CSP model and its solution space. The variables X_0 and X_1 represent the start times of tasks T_0 and T_1, respectively. The domain of both variables (represented by discontinuous lines) is $[0 \dots 4]$, which preserves the maximum makespan of six time units (the maximum start time of a task is the maximum makespan minus the duration of the aforesaid task). There is one constraint controlling the execution order of the tasks (T_0 must start before T_1), which is $C_0 : X_1 \geq X_0 + 2$. The solution space is represented by a dark gray area, where there are six solutions (black dots).*

If no specific information is given about the dynamic environment, which schedule is the most robust? As stated in Section 8.1, the greater the number of time buffers and the greater their duration, the more robust the schedule is. But how can we determine which solution of the modeled CSP meets these requirements? The answer is obtained by determining the feasible contiguous neighbours with greater values, located at distances less or equal to k from the solution. However, depending on the value of k, we will either prioritize the selection of schedules with a large number of short time buffers or we will prioritize the selection of schedules with lower number of long time buffers. The number of greater feasible neighbours associated with a value of a variable corresponds to the total amount of slack that is located after the task represented by this variable. Thus, the slack is able to absorb a delay in the previous task as long as itself, without modifying the other tasks of the present schedule (robustness feature). Furthermore, if the slack following a task is not sufficient to absorb a delay, the start of the following task can be delayed (after repairing the broken assigned value) if there is a long enough buffer associated with this later task (stability feature).

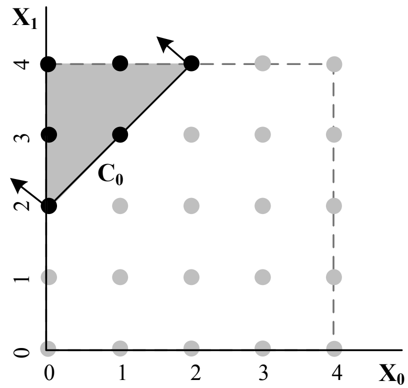


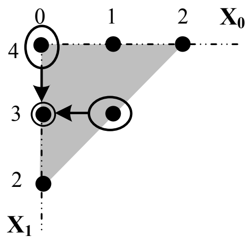
Figure 2: CSP model associated with Example 8.1 and its solution space.

For the above example, which is a two-dimensional CSP representing a scheduling problem with two tasks, there are three schedules that are most robust according to the criteria stated above. If we maximize the sum of distances for greater values located at a distance one ($k = 1$) from each value of the assignment, we obtain the solution shown in Figure 3(a), whose sum is $f(s_0, k = 1, \{>, +\}) = 1 + 1$. The first number is $\mathcal{N}_k(x_0, v_0, s, \{>, +\})$ and second is $\mathcal{N}_k(x_1, v_1, s, \{>, +\})$, where v_0 and v_1 are the values assigned to the variables x_0 and x_1 respectively. Note that the sums for the neighbours greater than the solution values and located at a distance one from them are $f(s_1, k = 1, \{>, +\}) = 1 + 0$ and $f(s_2, k = 1, \{>, +\}) = 0 + 1$, respectively. In the following (a) figures, the greater neighbours are indicated by an elipse, with an arrow pointing to the solution (the circled dot). In the associated (b) figures, the schedules equivalent to the solutions marked in (a) are shown. Note that the greater neighbours indicated in the (a) figures correspond to the slack in the (b) figures. For instance, in Figure 3(b) each task has an associated slack of duration one, which corresponds to the existence of one greater neighbour for each value assignment in Figure 3(a).

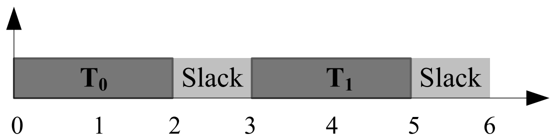
On the other hand, if we maximize the sum of greater neighbors values for $k > 1$, the three solutions represented in Figures 3(a), 4(a) and 5(a) are all classified as best solutions according to our objective function. The computation of the sum of neighbours located at a distance lower or equal to k, for $k > 1$ is: $f(s_0, k > 1, \{>, +\}) = 1 + 1$ (Figure 3(a)), $f(s_1, k > 1, \{>, +\}) = 2 + 0$ (Figure 4(a)) and $f(s_2, k > 1, \{>, +\}) = 0 + 2$ (Figure 5(a)). Note that the schedules in Figures 4(b) and 5(b) each have only one time buffer, but its duration is two time units, unlike the schedule represented in Figure 3(b) that has two time buffers of one unit each. Thus, by fixing $k = 1$ we prioritize the seek of a high number of time buffers. However, for greater k values, we prioritize their duration, even if in this case their distribution may not be optimal.

If we consider the stability of the solutions, small modifications in the solutions are always preferred. In this case, if it is not possible that a task starts at the scheduled time, by reassigning its start time to a closer greater neighbour, we are composing another schedule that is very similar to the original one. Therefore, the search of the feasible greater neighbours (which introduces buffers into the tasks of the schedule) are improving both, the robustness and stability of the obtained schedules.

The search of schedules with buffers that are up to k time units can also be achieved with model reformulation techniques. This is achieved by adding two variables to each original variable (the variables that represent the start time of the tasks). One variable represents the slack that is following the task and the other variable represents the sum of this slack and the original starting time. For

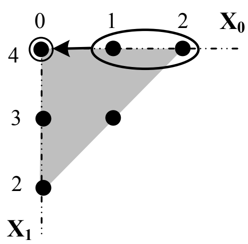


(a) Solution space.

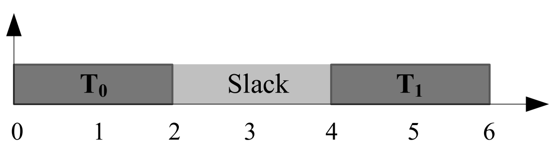


(b) Schedule of the marked solution.

Figure 3: Robust schedule $s_0 = (x_0 = 0, x_1 = 3)$ for Example 8.1 and its greater neighbours for $k \geq 1$.

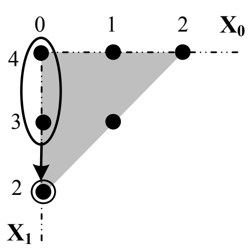


(a) Solution space.

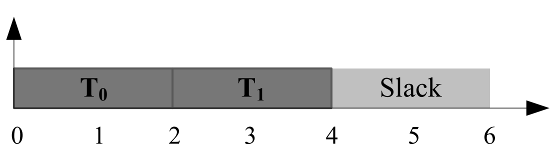


(b) Schedule of the marked solution.

Figure 4: Robust schedule $s_1 = (x_0 = 0, x_1 = 4)$ for Example 8.1 and its greater neighbours for $k > 1$.



(a) Solution space.



(b) Schedule of the marked solution.

Figure 5: Robust schedule $s_2 = (x_0 = 0, x_1 = 2)$ for Example 8.1 and its greater neighbours for $k > 1$.

instance, let p_i be the starting time of the task x_i. Thus, we would add the constraint $p_i' = p_i + s_i$, where s_i represents the slack associated to task x_i. In addition, depending on the maximum desired duration of the buffers, another constraint may be added, such as $s_i \leq k$. In this case, the delay is up to k time units. In addition, an objective function that express the goal of maximizing the total slack ($max \sum_{i=1}^{n} s_i$) must be defined.

Furthermore, other proactive specific approaches for scheduling problems that do not involve CSP can be found in the Herroelen and Leus (2005) survey. The main advantage of the approach presented in this paper over these proactive alternatives for scheduling problems is that our approach can be applied when all slack-values require a consistency check. This requirement is necessary in scheduling problems where intermediate non-valid slack values are possible. Examples of this type of problem are scheduling problems with limited machine availability (see, for instance Schmidt, 2000). In these cases, some machines are unavailable in certain time intervals; for this reason, tasks that require these resources cannot be executed in such time units. The same happens with some scheduling with operators, where the workers have some breaks during the day. Moreover, there also exist reactive approaches, which re-schedule the activities when a disruption invalidates the original schedule found. For example, solving dynamic Resource-Constrained Project Scheduling Problems (RCPSP) (Elkhyari, Guéret, & Jussien, 2004).

9. Experimental Results

In this section, we present results from experiments designed to evaluate the performance of Algorithm 1. Solutions obtained by the *restarting-completion* procedure are referred to as "neighbour solutions" in the graphs and tables throughout this section. Solutions obtained by *restarting-scratch* are referred to as "neighbour solutions(R)". Experiments were done with random problems and benchmarks presented in the literature. The random instances generator (RBGenerator 2.0), the benchmarks and the parser for the XCSP instances can be found on Christophe Lecoutre's web page [1].

In addition to assessing search Algorithm 1, we also evaluated two other proactive methods that do not require specific additional information about the dynamism. One of them is the WCSP modeling technique (Climent et al., 2013), which is based on the same dynamism assumptions as in the present work. The solutions obtained by this technique are referred to as "WCSP-mod solutions". We have not evaluated this approach with scheduling problems because it does not consider the adaptation for scheduling problems that we have presented in Section 8.2 (neighbouring values that are lower in magnitude are not considered). The other proactive approach maximizes the (1, 0)-repairability (see Section 3.2). To implement this technique, we have modified Algorithm 1 (B&B-$\mathcal{N}_k$) by exchanging MGAC3-$\mathcal{N}_k$ and GAC3-$\mathcal{N}_k$ algorithms for MAC+ and GAC+ (Hebrard, 2006), respectively. The solutions obtained by this technique are referred to as "(1, 0)-super-solutions". In addition, solutions of an ordinary CSP solver have been analyzed (referred to as "simple solutions"), in order to detect whether there are cases in which all solutions have similar robustness and/or stability.

In addition, we added the geometric restart (restarting-completion) and bounds consistency techniques explained in Section 7 to the ordinary CSP solver and the super-solutions solver in order to provide them with these computational advantages. For the approach that models CSPs as WCSP,

1. *http://www.cril.univ-artois.fr/ lecoutre/index.html*

we have used the same solver as the one used by Climent et al. (2013): ToulBar2[2]. It was necessary to use a different solver for the evaluation of this technique because this approach requires a WCSP solver. For these other approaches evaluated, values were selected in lexicographical order and a time cutoff was fixed to 100 seconds. Experiments were run on an Intel Core i5-650 Processor (3.20 Ghz). In addition, for the geometric restart, the scale factor was fixed to 10 and the multiplicative factor to 1.5.

The evaluation is based on the two main features of solutions obtained by proactive approaches: stability and robustness. In all the tables of this section, the best robustness/stability results obtained are marked in bold. In accordance with the assumptions laid out in the previous sections, we use the robustness and stability measures described in Section 4. Here, we note that the ordinary CSP solver and the super-solutions solver do not consider the same dynamism assumptions as the WCSP modeling technique and the approach presented in this paper. That is to say, they do not consider possible future restrictive modifications over the bounds of the solution space of CSPs with ordered domains. Regarding stability, only the technique that maximizes the (1, 0)-repairability searches for stable solutions according to Definition 3.1. However, as mentioned above, in this paper we analyze a more precise concept of stability for CSPs with ordered domains, the (1, 0, c)-repairability (see Definition 4.3).

9.1 Robustness Analysis with General CSPs

In this section we analyze the robustness and stability of solutions obtained over a wide range of tightness values. For this purpose, random CSPs were generated by the RBGenerator 2.0, which have non-convex constraints represented extensionally. Because of the non-convexity of the domains, the bounds consistency technique cannot be used. The CSPs generated have 25 variables with domain size 30 and 200 binary constraints. Domain values are integer values in the interval $[0, 29]$. The tightness values analyzed are 0.1, 0.2, and 0.3. (Note: 0.34 is the critical value of the tightness of this CSP typology.) For each tightness we generated 10 random instances that were solved by Algorithm 1 for $k = 1$. Because in this analysis we deal with the general case of CSPs with ordered domains (see Section 5.1) we have fixed the set of operators for our search algorithm to $\oplus = \{\{>, +\}, \{<, -\}\}$ and the value selection is heuristic 2 (see Section 7), which maximizes $|\mathcal{N}_k(x, v, s, \oplus)|$ starting from intermediate values.

As mentioned previously, it is not usually feasible to compute the complete set of solutions of a CSP. For this reason, in order to measure the robustness of the solutions obtained with the four approaches, we have sampled the closest surrounding neighbours ($k = 1$). Thus, if a closest surrounding neighbour is not a solution of the CSP, it means that the analyzed solution could become infeasible after a change of magnitude one or greater to the original bound/s that invalidate such neighbour. On the other hand, if the neighbour is a solution of the CSP, this means that this restrictive modification would not invalidate the analyzed solution. Therefore, satisfiability checking of a random sample of the neighbours of the solutions provides an estimation of the likelihood that the solutions will remain valid, that is to say, an estimation of their robustness.

For sampling the feasibility of the neighbourhood of the solutions, we made a certain number of random modifications of magnitude k over the values assigned to the variables of the solutions. The number of values assigned to the variables of the solutions that are modified, is denoted as $nbVarMod \in [1 \dots 10]$. For each value of $nbVarMod$, we sampled 500 neighbours over the

2. *http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/ToolBarIntro*

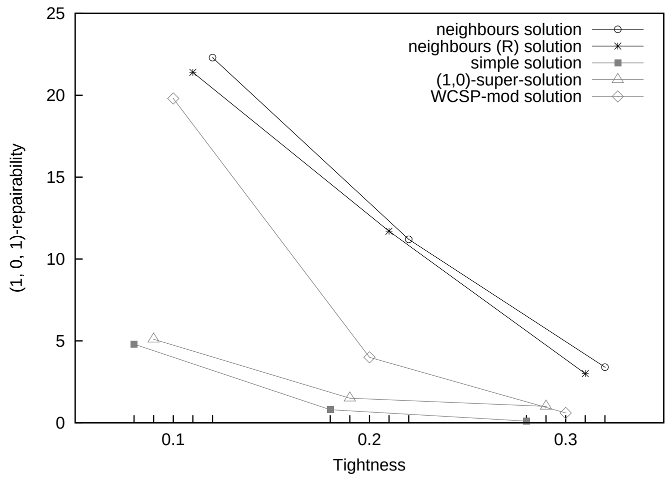
solution analyzed and checked their feasibility. The average number of feasible neighbours for each type of solution are shown in Table 9.1. It can be observed that Algorithm 1 with either restarting option dramatically outperformed the ordinary CSP solver and the technique that maximizes the (1, 0)-repairability. It also outperformed the WCSP modeling approach for tightness 0.2 and 0.3. The weakness of the latter approach is that it obtains robustness approximations in problems in which there is a high relation between constraints, because it computes feasible neighbours for each constraint boundary. Thus, the higher the tightness, the higher the likelihood of the existence of neighbour tuples that are feasible for one constraint/domain but not for another one. These conflicting situations are less frequent in very unconstrained instances. For this reason, for tightness 0.1 the performance of the modeling approach is better. However, it only obtains better robustness results than Algorithm 1 for highly unrestricted instances for high $nbVarMod$ values. In regard to our Algorithm 1, the restarting-completion option provides better results than restarting-scratch (differentiated with "R") for very unconstrained instances, while they preform similarly for higher tightness values. In Figure 6(b) we selected the $nbVarMod = 2$ to emphasize trends in robustness and stability as a function of varying tightness.

tightness	0.1					0.2					0.3		
$nbVarMod$	2	4	6	8	10	2	4	6	8	10	2	4	6
Approach	Average Number of feasible neighbours in the sample												
simple	7.2	0.6	0	0	0	0.2	0	0	0	0	0	0	0
super	8.8	0.6	0	0	0	1	0	0	0	0	0.4	0	0
WCSP-m	152.4	60.2	24.5	**12.8**	**5.8**	5.7	0.1	0	0	0	0.4	**0.1**	0
neigh	**206.8**	**75**	**27.4**	10.8	3.8	**36.2**	**2.5**	0	0	0	**2.8**	**0.1**	0
neigh(R)	191.6	74.4	24.4	7.9	2.9	33.9	2	**0.5**	**0.1**	0	2.5	0	0

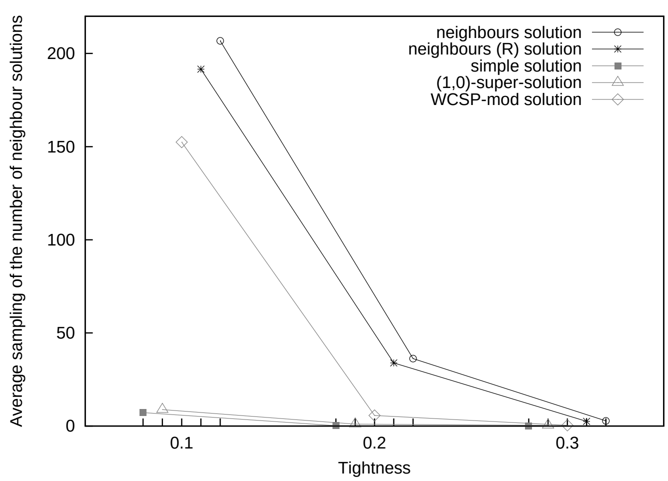
Table 1: Robustness Analysis Based on the tightness $(< 2, 25, 30, 200, \textit{tightness} >)$.

For the stability measurement, (1, 0, 1)-repairability is used (see Definition 4.3), which measures the number of variables that can be replaced by a value located at a distance of one from the value assigned without modifying the rest of values in the solution. Stability results are shown in Figure 6(a). As mentioned, if a solution value is lost, the objective is to find the closest repairable values. For this reason, our algorithm does not consider feasible values that are k units greater or smaller than the value assigned, since this could result in future solutions where the Manhattan distance between the new solution and the original one would be exaggeratedly great (see Section 4). On the other hand, the technique that maximizes the (1, 0)-repairability considers any value as a repairable value. This fact represents a disadvantage when searching for repairable values in ordered domains. This can be observed in Figure 6(a), where we can see the poor performance of super-solutions for the (1, 0, 1)-repairability.

We would like to note that for CSPs that are very highly restricted, the stability and robustness of the solutions obtained by all the evaluated methods are very similar. This is due to the fact that in these cases the CSPs have very few solutions and consequently the distances of *all* solutions from the bounds is very low. For most of these instances, the number of solutions is so low that the solutions are scattered within the tuple-space, so the likelihood of a solution being located on the bounds of the solution space is very high. For the same reason, the likelihood that a variable has a



(a) Stability analysis



(b) Robustness analysis for $nbVarMod = 2$

Figure 6: Combined robustness-stability based on the tightness ($< 2, 25, 30, 200, tightness >$). The curves are shifted for improving the clarity of the graph.

feasible repairable value that is near-by is very low. It can even be the case that none of the solutions has an assignment with feasible neighbours located at a distance k. In this case, all the solutions are equally robust and stable for this k value.

9.2 Scheduling Benchmarks Evaluation

In this section, we evaluate the approaches with scheduling benchmarks from the literature in order to determine the robustness of schedules obtained over a wide range of k values. We analyzed five sets of 10 job-shop CSP instances, studied by Sadeh and Fox (1996). Each instance is composed of 10 jobs of five tasks each and there are five resources. Each job has a random linear sequence of resources to visit, with the exception of the *bottleneck* resources, which are visited after a fixed number of operations (in order to further increase resource contention).

Because this analysis deals with scheduling problems (see Section 8) we have fixed the set of operators for our search Algorithm 1 to $\oplus = \{\{>, +\}\}$ and the value selection is done with heuristic 1, in which values are selected in lexicographical order. Regarding the other proactive technique evaluated, the author of the (1, 0)-repairability approach (Hebrard, 2006) made an extension to the concept of *breakage* (the loss of an assigned value) for scheduling problems. A breakage for this kind of problem was considered a delay of duration d in a task. Therefore, only values that are greater than the value assigned in d time units are considered repairable values. For this reason, for the evaluation of scheduling problems, we have incorporated this condition to the (1, 0)-repairability approach. For a proper comparison of this approach with our approaches, we used the same values for k and d parameters. In the following, in order to avoid term repetition, we assume that $d = k$. Note that an ordinary CSP solver does not use this parameter, so it obtains the same schedule for any value of k.

For measuring the robustness of the schedules obtained, we used the robustness measures introduced in Section 8.1. A first robustness assessment is made by measuring the total slack whose duration does not exceed k, which is denoted as $tS(k)$. In addition, a more accurate measure is also used, $R^s_{slack}(k)$ (see Equation 3), which measures the average total slack, minus the standard deviation multiplied by the α parameter. The α parameter was fixed to 0.25, which is inside the interval that the authors consider appropriate for this parameter. Another robustness measure used is based on the resistance of a schedule faced with perturbations, and is denoted as $R^s_{F,Z}$ (see Equation 4), where Z is the set of incidents that consist in delays of durations up to max_d over the tasks. We have used 2 different values for max_d: 1 and k. In each case, we independently simulated 500 delays up to max_d units with equal probability over the entire schedule and checked if the schedule remained valid. For the stability measurement, again, (1, 0, 1)-repairability is used (see Definition 4.3), which is equivalent to the measurement of the number of buffers of the schedule, denoted as nbB. Note that the desired objective is that in cases where repairs are necessary, the start time of a task is delayed for as short a time as possible.

The following figures and tables show the evaluation for two of the Sadeh problem sets. For the other problem sets we obtained similar results. We show results for the *e0ddr1* and *e0ddr2* benchmarks in order to compare robustness and stability of schedules obtained with different numbers of bottlenecks in the problem (other parameters are fixed). Sadeh stated that the *e0ddr1* benchmark contained just one bottleneck and on the other hand, *e0ddr2* benchmark contained two bottlenecks. Tables 9.2 and 9.2 show the means for the robustness and stability measures for scheduling problems. In addition, other measurements are shown, including the number of schedules obtained nbS, total number of restarts done by the search algorithm nbR, the total number of nodes explored nbN and the total number of failures nbF. Figure 7 shows the stability and robustness measurements (vertical axis): the mean number of buffers and mean $R^s_{slack}(k)$ for the *e0ddr1*. The horizontal axis of the figures represents the value of the ratio of parameters k.

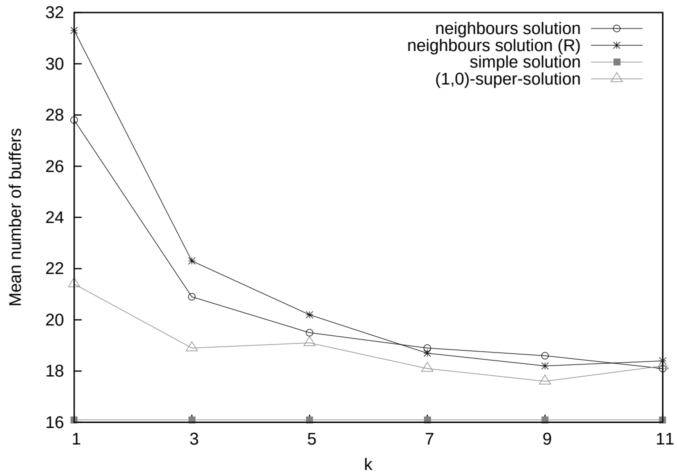
k	Approach	nbS	nbR	nbN	nbF	nbB	$tS(k)$	$R^s_{slack}(k)$	$R^s_{F,Z}(1)$	$R^s_{F,Z}(k)$
1	simple	1	3.1	208	85	16.1	16.1	0.208	0.328	0.338
	super	9.1	3.1	7770.4	3043.7	21.4	21.4	0.308	0.434	0.43
	neigh	12.7	3.1	10171.1	2465.9	27.8	27.8	0.436	0.562	0.555
	neigh(R)	15.5	28.8	2820.4	628.1	**31.3**	**31.3**	**0.509**	**0.628**	**0.618**
3	simple	1	3.1	208	85	16.1	44.3	0.555	0.328	0.311
	super	7.3	3.1	8138.2	2619.2	18.9	52.4	0.702	0.384	0.36
	neigh	15.9	3.1	5880.5	2485.1	20.9	59.4	0.832	0.424	0.409
	neigh(R)	15.5	27.7	2406.5	670.5	**22.3**	**62.1**	**0.886**	**0.448**	**0.413**
5	simple	1	3.1	208	85	16.1	67.8	0.832	0.328	0.288
	super	7.1	3.1	8373.2	2654.2	19.1	82.9	1.101	0.388	0.343
	neigh	19	3.1	3947.7	1674.7	19.5	86.3	1.159	0.396	**0.364**
	neigh(R)	12.9	23.8	2082.3	517.8	**20.2**	**87.3**	**1.182**	**0.406**	0.35
7	simple	1	3.1	208	85	16.1	88.1	1.057	0.328	0.271
	super	6.5	3.1	8319.1	3219.5	18.1	101.8	1.298	0.368	0.303
	neigh	19.9	3.1	3205.9	1032.6	**18.9**	107.8	1.4	**0.384**	**0.331**
	neigh(R)	11.5	21.2	1871.8	489.5	18.7	**108.6**	**1.413**	0.376	0.314
9	simple	1	3.1	208	85	16.1	105.7	1.242	0.328	0.257
	super	5.7	3.1	8715.7	2620.7	17.6	117.6	1.452	0.358	0.277
	neigh	20.8	3.1	2793.6	974.2	**18.6**	**126.5**	**1.602**	**0.378**	**0.303**
	neigh(R)	11.4	19.7	1711.4	462.8	18.2	126	1.588	0.368	0.293
11	simple	1	3.1	208	85	16.1	120.5	1.389	0.328	0.244
	super	6	3.1	8019.9	1775.8	18.2	133.6	1.629	0.37	0.256
	neigh	19	3.1	2518.5	844.4	18.1	**140.2**	**1.72**	0.368	**0.28**
	neigh(R)	7.9	16.9	1593.9	435.5	**18.4**	138.8	1.693	**0.374**	0.277

Table 2: Evaluation of 'e0ddr1' benchmark.

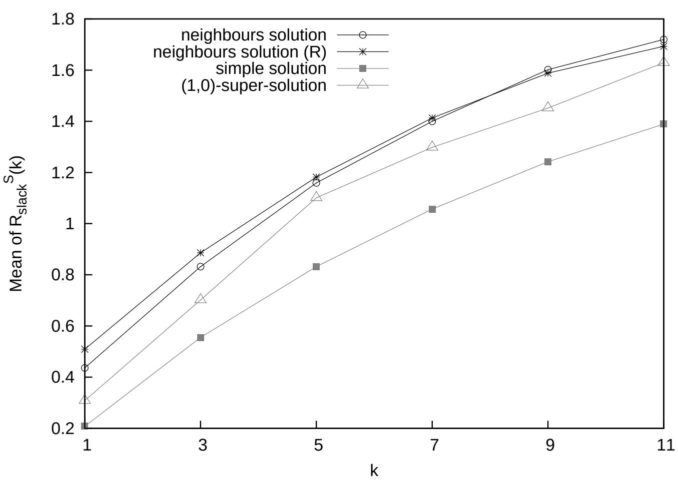
As expected, schedules obtained by all of the approaches for the e0ddr1 benchmark are more robust and stable than those for the e0ddr2 benchmark (see Tables 9.2 and 9.2) From the robustness analysis, we see that our algorithm for $k = 11$ (for both restarting options) increased the robustness measure $R^s_{F,Z}(k)$ by more than 0.5 units for problems with only one bottleneck. Therefore, as expected, the fewer bottlenecks a scheduling problem has, the more robust the schedule obtained by our algorithm. Detailed results for all robustness measures are found under columns $tS(k)$, $R^s_{slack}(k)$, $R^s_{F,Z}(1)$ and $R^s_{F,Z}(k)$ in the tables. For instance, for the largest k value analyzed ($k = 11$), the total sum of all the buffer times of duration up to k of the schedule obtained by Algorithm 1 for restarting-completion is 140.2 time units for the *e0ddr1* benchmark and 109.67 time units for the *e0ddr2* benchmark (more than 30 time units difference). Regarding the stability analysis, our algorithm for $k = 1$ restarting-scratch (differentiated with "R") found schedules with four mean number of buffers (nbB) more for the problems with one bottleneck than for the problems with two

k	Approach	nbS	nbR	nbN	nbF	nbB	$tS(k)$	$R^s_{slack}(k)$	$R^s_{F,Z}(1)$	$R^s_{F,Z}(k)$
1	simple	0.9	4	227	98.89	14.11	14.11	0.17	0.28	0.28
	super	6.5	3.9	8024.33	1975.67	19.89	19.89	0.28	0.4	0.41
	neigh	10.1	4	11264.22	1865.22	24.33	24.33	0.37	0.49	0.49
	neigh(R)	12.4	25.4	2555.67	709.33	**27.44**	**27.44**	**0.43**	**0.55**	**0.55**
3	simple	0.9	4	227	98.89	14.11	37.56	0.44	0.28	0.25
	super	4.9	3.9	8602.67	1198.67	17.33	47.22	0.61	0.35	0.32
	neigh	16.4	3.9	7141.78	1711.33	**20.22**	**56**	**0.77**	**0.4**	**0.37**
	neigh(R)	11.9	22.2	2282.67	503.89	20.11	55.22	0.76	**0.4**	**0.37**
5	simple	0.9	4	227	98.89	14.11	55.11	0.63	0.28	0.22
	super	4.4	3.9	9102.78	743.67	17.11	70.89	0.89	0.34	0.29
	neigh	17.3	3.9	5755.22	1657.33	**18.11**	**76.67**	**0.99**	**0.36**	**0.31**
	neigh(R)	8.6	18.9	2036.11	452.78	17.89	73.89	0.95	**0.36**	0.3
7	simple	0.9	4	227	98.89	14.11	68.22	0.75	0.28	0.2
	super	3.8	3.9	9721.67	914.22	15.78	82.22	0.97	0.32	0.25
	neigh	15.2	3.9	4903	1272.56	16.89	**88.78**	**1.09**	0.34	**0.26**
	neigh(R)	7.7	17.5	1827.44	428.78	**17.22**	88.67	**1.09**	**0.35**	**0.26**
9	simple	0.9	4	227	98.89	14.11	78.67	0.84	0.28	0.18
	super	3.1	3.9	9971	959.56	15.56	92.89	1.06	0.31	0.21
	neigh	15.7	3.9	4344.44	1161.78	**16.78**	**101.11**	**1.2**	**0.34**	**0.24**
	neigh(R)	6.4	16.2	1657.56	449.22	**16.78**	100.44	1.19	**0.34**	0.23
11	simple	0.9	4	227	98.89	14.11	87.44	0.91	0.28	0.16
	super	2.3	3.9	10698.22	1090.44	15.22	98.89	1.08	0.3	0.19
	neigh	14.7	3.9	4251.78	1223.56	**16.22**	**109.67**	**1.25**	**0.32**	**0.21**
	neigh(R)	5.6	14.4	1588.89	390	16.11	107.67	1.22	**0.32**	0.2

Table 3: Evaluation of ‘e0ddr2’ benchmark.



(a) Stability analysis



(b) Robustness analysis

Figure 7: Combined robustness-stability for k parameter: mean measures for the e0ddr1 benchmark.

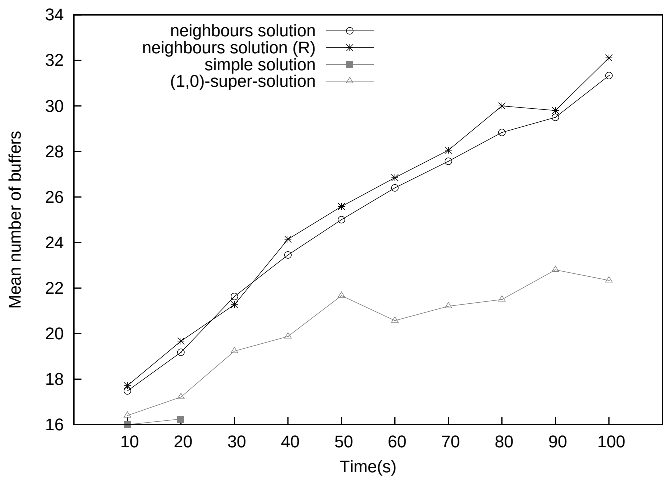
bottlenecks for the best case. Therefore, as expected, the fewer bottlenecks a scheduling problem has, the more stable the schedule obtained by our algorithm.

In both tables and the figure, we can see that Algorithm 1 with either restarting option outperformed both the ordinary CSP solver and the technique that maximizes the (1, 0)-repairability. Furthermore, the analysis of the k parameter shows that when these parameters have the lowest values, the number of buffers of the schedules found by our algorithm are markedly greater than these two techniques (see Figure 7(a)). In contrast, the improvement in robustness for our algorithm with respect to the ordinary solver is a little more marked for greater k values. The comparison with the (1, 0)-repairability technique shows the same tendency for the *e0ddr2* benchmark (see nbB in Table 9.2).

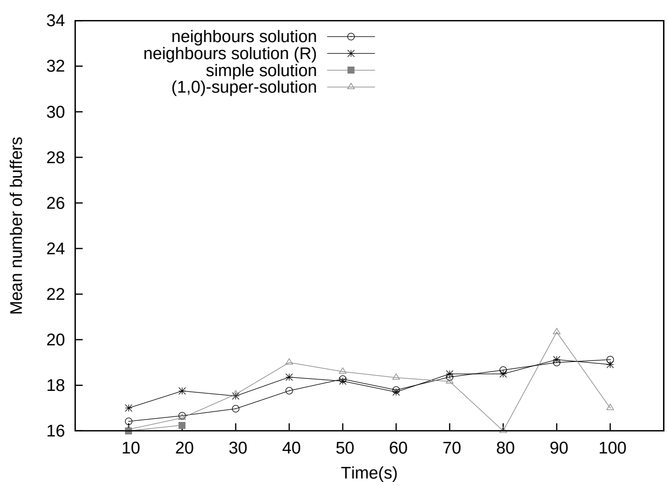
Regarding the other robustness measures that are not plotted in the figure but are shown in Tables 9.2 and 9.2, we see that there is a correlation between the $R^s_{F,Z}(1)$ measure and the number of buffers. This relation is expected, since the random incidents generated for measuring $R^s_{F,Z}(1)$ were delays of one unit time. Therefore, the more buffers there are (whatever is their duration) the greater the likelihood that a schedule can absorb delays of one time unit. In addition, the $tS(k)$, $R^s_{slack}(k)$ and $R^s_{F,Z}(k)$ measures are correlated. Recall that $tS(k)$ is the total slack whose duration does not exceed k and $R^s_{slack}(k)$ is its average minus the standard deviation multiplied by an α parameter. Therefore, unless the distribution of the slack is very poor, the two values must be proportional. Note that the lower the α parameter for $R^s_{slack}(k)$, the greater the proportionality with respect the other two robustness measures. The $R^s_{F,Z}(k)$ measure is calculated by generating random delays up to duration k over the schedule. For this reason, this robustness measure is strongly related with the two aforementioned. A example of the relation of all the aforementioned measurement units can be observed in Table 9.2 for $k = 11$, where the schedules obtained with restarting-scratch option (differentiated with "R") have greater numbers of buffers and $R^s_{F,Z}(1)$ values, and schedules obtained with restarting-completion option have greater $tS(k)$, $R^s_{slack}(k)$ and $R^s_{F,Z}(k)$ values. This means that the latter has a greater total slack whose duration does not exceed k, but its distribution is more limited.

In Tables 9.2 and 9.2 we also observe measurements that are not correlated with robustness or stability, but important information can still be extracted from them. For $k > 1$, the restarting-completion for our algorithm finds the greater mean number of solutions (nbS). Only for $k = 1$ does the restarting-scratch (differentiated with "R") find more solutions. The greater k is, the easier it is to find new solutions whose objective function is better than the maximum one (if the instance is not highly restricted). For this reason, the mean number of solutions found is greater for high k values. For both restarting options, the mean number of solutions is considerably higher than for the approach that maximizes the (1, 0)-repairability. This effect is stronger for greater values of k because the condition of a repairable value for the latter technique becomes more restrictive. Moreover, this technique considers all feasible values in the domains as repairable values; as a result, feasibility checking is slower than for techniques that assume only k neighbours (as our technique does). As expected, the mean number of restarts (nbR) is much greater for the restarting-scratch option because the other techniques only restart until finding the first solution. As a consequence, their mean number of nodes explored (nbN) and mean number of failures (nbF) is lower.

The schedules obtained by Algorithm 1 for the lowest k value had the highest number of buffers. On the other hand, the robustness measures are greater for the greater k values. Depending on the dynamic nature of the problem, it would be desirable to prioritize between a higher number of buffers of short duration and a lower number of buffers of long duration (if the two features cannot both be maximized). Thus, if it is known that the possible future delays will have a duration of at least d time units, it does not make sense to compute k values lower than d because the obtained time buffers could not absorb the delay. On the other hand, if it is known that possible future delays cannot have a duration greater than d, then it does not make sense to compute k values greater than d time units because this may decrease the number of buffers. Hence, the more information about possible future changes we have, the better the robustness results we can obtain. However, even if this information is unknown, we can obtain a schedule with certain level of both robustness and stability by setting k to an intermediate value in Algorithm 1.



(a) $k = 1$



(b) $k = 7$

Figure 8: Mean number of buffers over the time intervals for the e0ddr1 benchmark.

The above evaluation consists of analyzing the best results obtained for each technique for the fixed cutoff time. However, we also wanted to analyze the change in the degree of robustness and stability of the schedules found over the time. For this evaluation, we used the e0ddr1 benchmark and determined the mean for 50 instances for each interval of time with a discretization of 10 seconds. Figures 8(a) and 8(b) show the mean number of buffers found by each approach for $k = 1$ and $k = 7$. Other measures are not shown since similar trends were found in these cases. We would like to highlight that after 20 seconds the simple solution technique does not find better schedules because it only searches for one schedule for each instance (which is done in less or equal to 20 seconds). The most remarkable aspect is that for $k = 1$ Algorithm 1 for both restarting options obtains a greater number of buffer times than the approach that maximizes the (1, 0)-repairability, for $k = 1$ for all time intervals (see Figure 8(a)).

On the other hand, Figure 8(b), which represents $k = 7$, shows more unstable results. Since it is difficult to find buffers with up to seven time units, it may happen than our algorithm sacrifices some shorter buffers in order to find one buffer of seven time units. Thus, even if the overall tendency is for the measure to increase over the time, it is not entirely uniform. On the other hand, the upward shape of the trend for the approach that searches for super-solutions is due to the fact that it considers values as repairable if there is any possible alternative for the start time of a task that follows a task sharing the same resource, which is not equivalent to have a slack associated to this task in the schedule. For this reason, schedules that are better for this technique may contain lower number of buffers. This feature is more marked for greater values of k, since the repairable values have to be at least k unit times greater than the assigned values, and therefore it is more unlikely to find repairable values that are close to the assigned ones.

It can be concluded that in general the approach that maximizes the (1, 0)-repairability finds solutions with lower robustness and stability (considering the closest repairable values) than our approach for the aforementioned reason. Another disadvantage is that it only assumes that delays are of duration d. Thus, only values greater than this value are considered as repairable values. However, we consider up to k neighbours and therefore, slacks of duration lower than k are also valued by our objective function in contrast to the (1, 0)-repairability objective function.

On the basis of this evaluation, we can conclude that the difference in performance between the two restarting options (restarting-completion and restarting-scratch) is not very significant. Sometimes, the time needed to restart from scratch after each solution makes this option less effective than restarting-completion. In other cases, the restarting-completion option loses time in branches in which there are no better solutions, while restarting-scratch explores other branches. For instance, for the random experiments, restarting-completion provided slightly better results generally (see Table 9.1 and Figure 6(b)), while for the scheduling problems, restarting-scratch obtained schedules that were a bit more robust and stable for lower k values (see $k \in [1, 5]$ in Figure 7). For greater k values, both restarting options gave similar results.

10. Conclusions

In this paper we extend the concept of robustness and stability for CSPs with discrete and ordered domains where only limited assumptions can be made about changes in these problems. In particular, there are no uncertainty statistics nor probabilities about the incidences that can occur in the original problem. In this context, it is reasonable to assume that the original bounds of the solution space may undergo restrictive modifications, such as introduced by Climent et al. (2013). Therefore, the main objective in searching for robust solutions is to find solutions that maximize all the Euclidean distances from the dynamic bounds of the solution space. On the other hand, the main objective in searching for stable solutions in terms of repairable variables is to find solutions whose repairable values are as close as possible to the broken assignments.

In this paper, we present a new search algorithm that combines criteria for both robustness and stability in this framework. The algorithm developed in this paper searches for a solution that maximizes the sum of contiguous feasible surrounding neighbours at distances of k or less from the values of the solution. The obtained solutions have a high probability of remaining valid after possible future restrictive changes over the constraints and domains of the original problem (robustness criterion), and they also have a high number of variables that can be easily repaired with a value at a distance lower or equal to k if they undergo a value loss (stability criterion).

We have evaluated the new algorithm in experiments on well-known scheduling benchmarks as well as random CSPs. We have shown that both versions of the new algorithm outperform three other approaches evaluated: the ordinary CSP solvers, the technique that maximizes the $(1, 0)$-repairability, and the approach that models CSPs as WCSPs under many conditions where there are real differences in the robustness of solutions that might be obtained. The latter occurs when the problem is not so constrained that there are only a few valid solutions. With respect to the two restarting options developed for our algorithm, we found that their performance is not significantly different, although in certain situations there is an advantage of one over the other.

For slightly constrained CSPs, our algorithm obtains solutions with the greatest number of closer neighbour solutions, the greatest (1, 0, 1)-repairability and highest values for specific measures of scheduling robustness. Furthermore, we have shown that by increasing k for large problems, we can also increase the robustness, although it may happen that (1, 0, 1)-repairability decreases. For instance, with scheduling problems the schedules obtained with lower k values tend to maximize the number of buffers even if their size is small. However, the computation of higher k values tends to give priority to the duration of the buffers and as consequence, the number of buffers obtained can be lower. Therefore, depending on the dynamic nature of the problem, it would be desirable to prioritize between a higher number of short buffers or a lower number of long buffers (if it is not possible to maximize both features).

The extension of the robustness and stability definition for CSPs with discrete and ordered domains and the development of a search algorithm for finding robust and stable solutions in this context, are useful and practical in many real life situations where problems can undergo restrictive changes and there is the added difficulty that information about the possible future changes is limited or non-existent. Even under these difficult conditions, our search algorithm is able to provide stable and robust solutions. Finding solutions located far away from the dynamic bounds is important when we face restrictive modifications over the bounds of the solution space. Moreover, in cases where a value is lost, it is important to replace it by a nearby value in order to have a solution as similar as possible to the original one. This closeness feature is handled by our algorithm but is not by the approach that searches for super-solutions.

Acknowledgments

This work has been partially supported by the research project TIN2010-20976-C02-01 and FPU program fellowship (Min. de Ciencia e Innovación, Spain). We wish to thank Dr. Christophe Lecoutre and Dr. Diarmuid Grimes for their assistance.

References

Bessiere, C. (2006). Constraint propagation. *Foundations of Artificial Intelligence*, *2*, 29–83.

Boussemart, F., Hemery, F., Lecoutre, C., & Sais, L. (2004). Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-04)*, Vol. 16, p. 146.

Climent, L., Wallace, R. J., Salido, M. A., & Barber, F. (2013). Modeling robustness in CSPs as weighted CSPs. In *Proceedings of the 10th International Conference on Integration of Artificial Intelligence and Operations Research techniques in Constraint Programming (CPAIOR-13)*, pp. 44–60.

Elkhyari, A., Guéret, C., & Jussien, N. (2004). Constraint programming for dynamic scheduling problems. *Hiroshi Kise, editor, ISS'04 International Scheduling Symposium*, pp. 84–89.

Escamilla, J., Rodriguez-Molins, M., Salido, M., Sierra, M., Mencía, C., & Barber, F. (2012). Robust solutions to job-shop scheduling problems with operators. In *24th IEEE International Conference on Tools with Artificial Intelligence (ICTAI-12)*, pp. 209–306.

Fargier, H., & Lang, J. (1993). Uncertainty in Constraint Satisfaction Problems: a probabilistic approach. In *Proceedings of the Symbolic and Quantitative Approaches to Reasoning and Uncertainty (EC-SQARU-93)*, pp. 97–104.

Fargier, H., Lang, J., & Schiex, T. (1996). Mixed Constraint Satisfaction: A framework for decision problems under incomplete knowledge. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pp. 175–180.

Fowler, D., & Brown, K. (2000). Branching Constraint Satisfaction Problems for solutions robust under likely changes. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP-2000)*, pp. 500–504.

Fu, N., Lau, H., Varakantham, P., & Xiao, F. (2012). Robust local search for solving RCPSP/max with durational uncertainty. *Journal of Artificial Intelligence Research*, *43*, 43–86.

Hebrard, E. (2006). *Robust Solutions for Constraint Satisfaction and Optimisation under Uncertainty*. Ph.D. thesis, University of New South Wales.

Hebrard, E., O'Sullivan, B., & Walsh, T. (2007). Distance constraints in Constraint Satisfaction. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, pp. 106–111.

Herroelen, W., & Leus, R. (2005). Project scheduling under uncertainty: Survey and research potentials. *European Journal of Operational Research*, *165*(2), 289–306.

Kitano, H. (2007). Towards a theory of biological robustness. *Molecular Systems Biology*, *3*(1).

Larrosa, J., & Schiex, T. (2004). Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence*, *159*, 1–26.

Leon, V., Wu, S., & Robert, H. (1994). Robustness measures and robust scheduling for job shops. *IIE transactions*, *26*(5), 32–43.

Lhomme, O. (1993). Consistency techniques for numeric CSPs. In *Proceedings of 13th the International Joint Conference on Artificial Intelligence (IJCAI-93)*, Vol. 13, pp. 232–232.

Mackworth, A. (1977a). Consistency in network of relations. *Artificial Intelligence*, *8*, 99–118.

Mackworth, A. (1977b). On reading sketch maps. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI-77)*, pp. 598–606.

Mohr, R., & Henderson, T. C. (1986). Arc and path consistency revisited. *Artificial intelligence*, *28*(2), 225–233.

Rossi, F., Venable, K., & Yorke-Smith, N. (2006). Uncertainty in soft temporal constraint problems: a general framework and controllability algorithms for the fuzzy case. *Journal of Artificial Intelligence Research*, *27*(1), 617–674.

Sadeh, N., & Fox, M. (1996). Variable and value ordering heuristics for the job shop scheduling Constraint Satisfaction Problem. *Artificial Intelligence*, *86*(1), 1–41.

Schmidt, G. (2000). Scheduling with limited machine availability. *European Journal of Operational Research*, *121*(1), 1–15.

Surico, M., Kaymak, U., Naso, D., & Dekker, R. (2008). Hybrid meta-heuristics for robust scheduling. *ERIM Report Series Reference No. ERS-2006-018-LIS*, *Available at SSRN: http://ssrn.com/abstract=902747*.

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

Wallace, R., & Freuder, E. (1998). Stable solutions for Dynamic Constraint Satisfaction Problems. In *Proceedings 4th International Conference on Principles and Practice of Constraint Programming (CP-98)*, pp. 447–461.

Walsh, T. (1999). Search in a small world. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Vol. 16, pp. 1172–1177.

Walsh, T. (2002). Stochastic Constraint Programming. In *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-02)*, pp. 111–115.

Yorke-Smith, N., & Gervet, C. (2009). Certainty closure: Reliable constraint reasoning with incomplete or erroneous data. *Journal of ACM Transactions on Computational Logic (TOCL)*, *10*(1), 3.