Modeling Robustness in CSPs as Weighted CSPs

Laura Climent[1], Richard J. Wallace[2], Miguel A. Salido[1] and Federico Barber[1]

[1] Instituto de Automática e Informática Industrial. Universitat Politècnica de València, Spain.
{lcliment, msalido, fbarber}@dsic.upv.es
[2] Cork Constraint Computation Centre. University College Cork, Ireland.
r.wallace@4c.ucc.ie

Abstract. Many real life problems come from uncertain and dynamic environments, where the initial constraints and/or domains may undergo changes. Thus, a solution found for the problem may become invalid later. Hence, searching for *robust* solutions for Constraint Satisfaction Problems (CSPs) becomes an important goal. In some cases, no knowledge about the uncertain and dynamic environment exits or it is hard to obtain it. In this paper, we consider CSPs with discrete and ordered domains where only limited assumptions are made commensurate with the structure of these problems. In this context, we model a CSP as a weighted CSP (WCSP) by assigning weights to each valid constraint tuple based on its distance from the edge of the space of valid tuples. This distance is estimated by a new concept introduced in this paper: *coverings*. Thus, the best solution for the modeled WCSP can be considered as a robust solution for the original CSP according to our assumptions.

Keywords: Robustness, Uncertainty, Dynamic CSPs.

1 Introduction

In many real-life situations both the original problem and the corresponding CSP model may evolve because of the environment, or the user or other agents. In such situations, a solution that holds for the original model can become invalid after changes in the original problem. This solution loss can produce negative effects in the situation modeled, which may entail serious economic loss. For example, it could cause the shutdown of the production system, the breakage of machines, delays in transports, or the loss of the material/object in production.

In order to deal with these situations, a number of *proactive approaches* that seek to obtain robust solutions for a given problem, have been proposed. These methods rely on knowledge about possible future changes in order to find solutions with a high probability of remaining valid when faced with these changes (see [13] for a survey). Most of these approaches assume the existence of additional knowledge about the uncertain and dynamic environment (see Section 3), which is often scarce or nonexistent in practice. In this paper, we propose a proactive technique that searches for robust solutions in situations in which only limited (and intuitively reasonable) assumptions are made about possible changes that can occur in CSPs with ordered and discrete domains: namely that changes always take the form of restrictions at the borders of a domain or constraint.

In the simplest case, that of a domain with a single range of values, $[a \dots b]$, this means increasing the lower bound (a) and/or decreasing the upper bound (b).

Example 1. Figure 1 shows the solution space (continuous lines) of a CSP called *P*, which is composed of two variables x_0 and x_1. It can be observed that P has 29 solutions (black points).

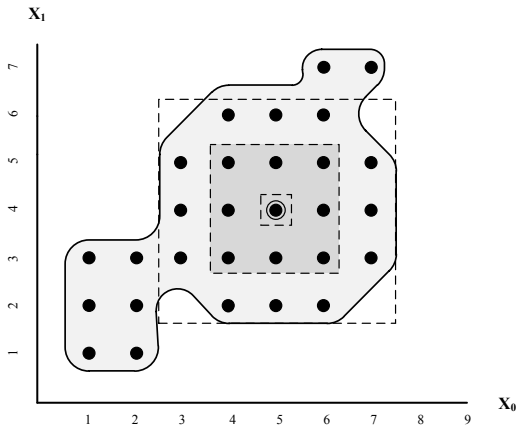

Fig. 1. Solution space of *P*.

If no information is known about unexpected changes in the CSP, it is reasonable to assume that the original constraints and/or domains undergo restrictive or relaxing modifications in the form of range restrictions or expansions, even if this does not cover all possible changes. For instance, in Figure 1, the darkest area represents restrictive modifications. Examples of real life problems that motivate this assumption include scheduling problems. For example, a task may undergo a delay, and therefore, the domains of the subsequent tasks must be reduced.

In this situation, since larger restrictions always include (some) smaller ones, we will assume that values affected by larger restrictions are, in general, less likely to be removed. This is shown in Figure 1, where solutions in the lighter area have a higher probability of becoming invalid. Given these assumptions, *the most robust solution of a CSP with discrete and ordered domains is the solution that is located as far as possible from the bounds of the solution space* and our basic strategy is to search for such solutions. In Example 1, the most robust solution is $(x_0 = 5, x_1 = 4)$.

In this paper we present a way of estimating this distance, which is based on the concept of *coverings*. Informally, a covering represents a set of partial or complete solutions that surround another solution and therefore it confers certain level of robustness. These calculations can be incorporated into a WCSP model by assigning weights to the valid tuples based on their coverings. Finally, the modeled WCSP is solved by a general WCSP solver; the best solution is considered to be one of the most robust solutions for the original CSP.

Previously, some work on weighted reformulation have been reported in [2] for Linear CSPs, and [1] for Boolean satisfiability (SAT). These approaches and the approach presented in this paper consider different robustness criterion and/or robustness computation.

Following some technical background (Section 2) and literature review (Section 3), the coverings framework is presented in Section 4. An algorithm for calculating coverings is described in Section 5. In Section 6 our enumeration-based technique is described; an example is given in Section 7. The experimental results obtained with our technique are presented in Section 8. Section 9 gives conclusions.

2 Technical Background

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

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

For a subset B of $\mathcal{X}_t$, the projection of t over B is denoted as $t \downarrow_B$. For a variable $x_i \in \mathcal{X}_t$, the projection of t over x_i is denoted as t_i. The possible tuples of $C_i \in \mathcal{C}$ are $\prod_{x_j \in var(C_i)} D_j$, where $var(C_i) \subseteq \mathcal{X}$. We denote the set of valid tuples of a constraint $C_i \in (\mathcal{C} \cup \mathcal{DC})$ as $T(C_i)$.

Definition 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].*

Definition 4. *A Dynamic Constraint Satisfaction Problem (DynCSP) [3] is a sequence of static CSPs* $\langle CSP_{(0)}, CSP_{(1)}..., CSP_{(l)} \rangle$*, each* $CSP_{(i)}$ *resulting from a change in the previous one (*$CSP_{(i-1)}$*) and representing new facts about the dynamic environment being modeled. As a result of such incremental change, the set of solutions of each* $CSP_{(i)}$ *can potentially decrease (in which case it is considered a restriction) or increase (in which case it is considered a relaxation).*

We only analyze DynCSPs in which the solution space of each $CSP_{(i)}$ decreases over $CSP_{(i-1)}$ (restriction) because relaxations cannot invalidate a solution found previously. As stated, our technique is applied before changes occur. Thus, it is applied to the original CSP ($CSP_{(0)}$) of the DynCSP.

Definition 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 changes in the CSP.*

Definition 6. *A Weighted Constraint Satisfaction Problem (WCSP) is a specific subclass of Valued CSP (see [12]). Here, we consider a variant of WCSP, formalized in [10]. This variant of WCSP is defined as* $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C}, S(W) \rangle$*, where:*

- $\mathcal{X}$ *and* $\mathcal{D}$ *are the set of variables and domains, respectively, as in standard CSPs.*
- $S(W) = \langle \{0, 1, ..., W\}, \oplus, > \rangle$ *is the valuation structure, where* $\{0, 1, ..., W\}$ *is the set of costs bounded by the maximum cost* $W \in \mathbb{N}^+$*,* $\oplus$ *is the sum of costs* $(\forall a, b \in \{0, 1, ..., W\}, a \oplus b = min\{W, a + b\})$ *and* $>$ *is the standard order among natural numbers.*
- $\mathcal{C}$ *is the set of constraints as cost functions* $(C_i : \prod_{x_j \in var(C_i)} D_j \rightarrow \{0, 1, ..., W\})$*.*

If a tuple *t* has the maximum cost W for C_i, it means that *t* is an invalid tuple. The global cost of a tuple *t*, denoted $\mathcal{V}(t)$, is the sum of all the applicable costs:

$$\mathcal{V}(t) = \bigoplus_{C_i \in \mathcal{C}, var(C_i) \subseteq X_t} C_i(t \downarrow_{var(C_i)}) \tag{1}$$

The tuple t is $consistent$ if $\mathcal{V}(t) < W$. The main objective of a WCSP is to find a complete assignment with the minimum cost.

3 Limitations of Earlier Proactive Techniques

The majority of earlier proactive approaches use additional information about the uncertain and dynamic environment and usually involve probabilistic methodologies.

In one example of this type, information is gathered in the form of penalties, in which values that are no longer valid after changes in the problem are penalized [14]. On the other hand, in the Probabilistic CSP model (PCSP) [4], 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) [5], considers the dynamism of certain *uncontrollable* variables that can take on different values of their uncertain domains. The Uncertain CSP model (UCSP) is an extension of MCSP whose main innovation is that it considers continuous domains [18]. The Stochastic CSP model (SCSP) [15] assumes a probability distribution associated with the uncertain domain of each uncontrollable variable. The Branching CSP model (BCSP) considers the possible addition of variables to the current problem [6]. For each variable, there is a gain associated with an assignment.

In most of these models, a list of the possible changes or the representation of uncertainty is required, often in the form of an associated probability distribution. As a result, these approaches cannot be used if the required information is not known. In many real problems, however, knowledge about possible further changes is either limited or nonexistent.

However, there is one proactive technique that does not consider detailed additional information. In this approach, one searches for *super-solutions* ([8]), which are solutions that can be repaired after changes occur, with minimal changes that can be specified in advance. For CSPs, the focus has been on finding (1,0)-super-solutions.

Definition 7. *A solution is a (1, 0)-super-solution if the loss of the value of one variable at most can be repaired by assigning another value to this variable without changing the value of any other variable ([8]).*

In general, it is unusual to find (1,0)-super-solutions where all variables can be repaired. For this reason, in [8] the author 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 Finding Coverings

In order to locate robust solutions, we have developed a technique for calculating *coverings* for valid tuples. The covering of a tuple measures the protection of the tuple against perturbations. It is based on an 'onion topology'. A valid tuple with more layers (its distance to the bounds is higher) is presumed to have a higher probability of remaining valid than a tuple with fewer layers. Here, a layer of a tuple is a convex hull of valid tuples.

To the best of our knowledge, the application of the 'onion structure' to CSPs is a novel idea, although it has been used in robustness for dynamic networks with targeted attacks and random failures. In [9] the authors state: "Our results show that robust networks have a novel "onion-like" topology consisting of a core of highly connected nodes surrounded by rings of nodes with decreasing degree".

We first introduce the concept of *topology* as it provides formulations about proximity relations between the elements that compose it. Subsequently, our own definitions and the associated techniques will be explained.

4.1 Topology of the CSPs

We consider the search space of a CSP with discrete and ordered domains as a set of n-dimensional hyperpolyhedra, where the set of valid tuples of the hyperpolyhedra is denoted by T. There are several distance functions $d(x, y) : T \times T \rightarrow \mathbb{R}^+$ that can be defined over each pair of tuples x and y. We will use the Chebyshev distance, also called L_∞ metric, which measures the maximum absolute differences along any coordinate dimension of two vectors.

$$d(x, y)_{Chebyshev} = \max_i (|x_i - y_i|) \tag{2}$$

The main reason for selecting this distance metric is that it distinguishes between hypercubes in n-dimensional spaces, which are analogous to squares for $n = 2$ (see Figure 1) and cubes for $n = 3$. In particular, the corners of a cube are at the same distance from the central point as the edges, a feature not obtained with Euclidean distance metric. By checking areas of satisfiability inside these hypercubes, we can ensure minimum distances to the bounds, which is used for the robustness computation (see Section 4.2).

For CSPs with symbolic domains the above definition cannot be applied unless there is an ordering relationship between their values. In this case, a monotonic function has

to be applied in order to map the elements by preserving their order. To this end, a monotonic mapping function f is defined over the elements of the CSP domain: $f(x) : \mathcal{D} \rightarrow \mathbb{Z}$.

Example 2. We consider a CSP with a symbolic and ordered domain $\mathcal{D}$ which represents clothing sizes: {extra small, small, medium, large, extra large}. In this case, a monotonic function that assigns greater values to the bigger clothing sizes can be defined. For example, $f(extrasmall) = 1$, $f(small) = 2$, $f(medium) = 3$, $f(large) = 4$, etc.

As shown above, a CSP with discrete and ordered domains (both numeric and symbolic) can have a metric function defined over their set of valid tuples T. Therefore, T is a topological space, and the following topological definition presented in [16] can be applied to CSPs:

Definition 8. *A closed ball of a valid tuple $t \in T$ at distance ϵ, is the neighborhood N ($N \subseteq T$) of t composed of $\{y \in T : d(t, y) \leq \epsilon\}$.*

4.2 The Concept of Coverings

As stated earlier, the core of an 'onion structure' is the most robust part of the structure, since the surrounding layers protect it against perturbations. The same is true for the solutions of a CSP: the further away a solution is from the bounds of the solution space, the more robust the solution is. In order to determine how far a valid tuple is from the bounds, we analyze its coverings ('onion layers').

Definition 9. *We define the k-covering(t) of a valid tuple $t \in T$ as its neighborhood $\{y \in T : y \neq t \wedge d(t, y)_{Chebyshev} \leq k\}$, where $k \in \mathbb{N}$.*

From Definition 9 the following property can be deduced: k-covering(t)$\supseteq$(k-1)-covering(t).

Definition 10. *|k-covering(t)| denotes the number of valid tuples inside k-covering(t), without including t.*

Definition 11. *$maxTup(k, |t|)$ denotes the maximum number of tuples that can make up a k-covering(t) (without including t) in a CSP with discrete and ordered domains, where k is the k-covering and $|t|$ represents the arity of t.*

$$maxTup(k, |t|) = (2k + 1)^{|t|} - 1 \tag{3}$$

Definition 12. *A k-covering(t) is complete if |k-covering(t)| = maxTup$(k, |t|)$.*

If k-covering(t) is complete, it means that t is located at a distance of at least k from the bounds, because inside the k-covering(t) all the tuples are valid. On the other hand, if at least one invalid tuple is inside of the k-covering(t), the unsatisfiability space is not completely outside of k-covering(t) and the minimum distance of t from the bounds of the solution space is the distance to the closest invalid tuple. Note that if at least one

of the closest neighbours of t is invalid, it means that t is located on a bound of the solution space.

If there are several tuples with the same number of complete coverings, are they equally robust? The answer is obtained by calculating the number of valid tuples in the minimum incomplete covering (the minimal covering beyond the maximum complete covering). Considering the 'onion topology', if there are holes in an 'onion layer', it is preferable that they be as small as possible.

In Figure 1 (see Example 1) the 1-covering(t) (darkest area) and 2-covering(t) (biggest square) for $t = (x_0 = 5, x_1 = 4)$ are represented with dashed lines. We can see that the 1-covering(t) is complete because |1-covering(t)| =maxTup(1,|t|)= 8. However, the 2-covering(t) is not complete because $maxTup(2, |t|) = 24$ and |2-covering(t)| $= 20$. Thus, we can only ensure that $(x_0 = 5, x_1 = 4)$ is located at a distance of at least 1 from the bounds (it has only one completed layer). Note that some bounds of the solution space are located inside the 2-covering(t).

5 Algorithm for Calculating Coverings

To search for a solution located as far as possible from the bounds of the solution space (core of the 'onion'), we must implicitly or explicitly examine the entire solution space of the CSP, which is an NP-complete problem. Our method of search is based on the coverings for each valid tuple of each constraint and domain of the CSP. In this section we present an algorithm for calculating this set of coverings.

5.1 Algorithm Description

Algorithm 1 calculates the coverings of the valid tuples, after first carrying out a global arc-consistency (GAC) process (line 1). In this preliminary step, it searches for a support of each domain value in order to detect tuples that are not globally consistent. For this purpose, we have implemented the well known GAC3 ([11]), but other consistency techniques could be applied. For calculating coverings, Algorithm 1 begins with $k = 1$ (1-covering(t)), increasing this value by one unit in each iteration until the maximum covering of a CSP is reached (or, optionally, a lower bound U). In each iteration, $\forall t \in T(C_i), \forall C_i \in (\mathcal{C} \cup \mathcal{DC})$, k-covering($t$) is computed, iff $k = 1$ or (k-1)-covering(t) is complete (see Definition 12).

Definition 13. *The maximum covering of a CSP is denoted as max-covering(CSP) and it is reached when there is no tuple whose k-covering is complete for some* $C_i \in (\mathcal{C} \cup \mathcal{DC})$.

Furthermore, if the user desires to obtain a lower k-covering(t), she/he can optionally fix a lower bound U. In this case, the algorithm stops after calculating $min(U,$ max-covering$(CSP))$.

Definition 14. *We define last-covering(t) to be the last k-covering(*t*) computed by the algorithm for the tuple* t*. The value of 'last' in last-covering(t) term is equal to* $min(U,$ *max-covering*$(CSP), (k + 1))$*, if k-covering(t) is the greatest completed covering of* t*.*

Algorithm 1 returns the size of last-covering(t) computed for each valid tuple t of each constraint and domain of the CSP. Thus, it gives a measure of robustness for each constraint. In addition, the value of max-covering(CSP) is returned (unless U is provided). The algorithm stores the number of valid tuples whose k-covering(t) is complete for each C_i in $kC[i]$, which is checked in line 11. Thus, it can check whether the current analyzed covering is max-covering(CSP).

Since k-covering(t) $\supseteq$ (k-1)-covering(t), the algorithm only analyzes the new possible neighbors of t, which are the neighbors that belong to it but do not belong to (k-1)-covering(t) (the neighbors that are in the k 'onion layer'). This is due to the fact that the neighbors of the lower coverings have already been calculated in previous iterations and stored in last-covering(t).

Algorithm 1: calculateCoverings (CSP P)

Data: A CSP $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ and U (optional)
Result: |last-covering(t)| $\forall t \in T(C_i) \forall C_i \in (\mathcal{C} \cup \mathcal{DC})$ and max-covering(P)

```
1  GAC3(P);
2  foreach C_i ∈ (C ∪ DC) do
       k ← 1;
3      max-covering(P) ← ∞;
       T(C_i) ← Ordered list of valid tuples of C_i;
4      foreach t ∈ T(C_i) do
           |last-covering(t)| ← 0;
5  repeat
6      foreach C_i ∈ (C ∪ DC) do
           kC[i] ← 0;
7          foreach t ∈ T(C_i) do
8              foreach {y ∈ T(C_i): y < t ∧ d(t_1, y_1)_Chebyshev ≤ k } do
9                  if isNewNeighbor(k,t,y) then
                       addNeighbor(k,|last-covering(t)|,t);
                       addNeighbor(k,|last-covering(y)|,y);
10                     if isComplete(k,|last-covering(y)|,|y|) then
                           kC[i] = kC[i] + 1;
11         if kC[i] = 0 then
               max-covering(P)= k;
       k ← k + 1;
   until k >max-covering(P) ;
   return |last-coverings|, max-covering(P) (unless U is provided)
```

Firstly, Algorithm 1 initializes some necessary structures (see loop beginning on line 2). Then the sets of valid tuples $T(C_i)$ of each constraint $C_i \in (\mathcal{C} \cup \mathcal{DC})$ are ordered by the value of the first variable of the valid tuples. In this way, a tuple a can only be located in a lower position than a tuple b if $a_1 \leq b_1$ (considering that the subindex 1 indicates the first variable that makes up the tuple). Thus, the tuples whose first variable has the minimum possible value will be placed in the lowest positions. For expressing

Procedure `isNewNeighbor` (k,t,y) : Boolean

```
   equalDist ← 0;
1  for i ← 1 to |t| do
2      if d(t_i, y_i)_Chebyshev > k then
           return False
3      if d(t_i, y_i)_Chebyshev = k then
           equalDist = equalDist + 1;
4  if equalDist > 0 then
       return True
5  else
       return False
```

Procedure `isComplete` (k,|last-covering(t)|,$|t|$) : Boolean

```
   maxTup(k, |t|) := (2k + 1)^|t|-1;
1  if |last-covering(t)| ≥ maxTup(k, |t|) then
       return True
2  else
       return False
```

the order of the tuples, we use the notation $a < b$, which means that the tuple a is located in a lower position than b in the list of ordered tuples.

The implementation of an algorithm for calculating coverings does not strictly require an ordered list of $T(C_i)$, but this ordering allows for a reduction in computation time, because it avoids checking a pair of tuples twice. This temporal benefit is achieved by selecting a reduced subset of possible neighbors of each valid tuple $t \in T(C_i)$ which are located in a lower position of t. If we do not select this reduced set, the algorithm must check all the valid tuples for each valid tuple. The reduced set is a subset composed of the valid tuples that are ordered in a lower position than t in the ordered list of $T(C_i)$ and whose difference between the value of their first variable with respect to t is lower or equal to k. Thus, the reduced set of t is composed of $y \in T(C_i) : y < t \wedge d(t_1, y_1)_{Chebyshev} \leq k$ (line 8).

The procedure `isNewNeighbor` checks if a valid tuple y is a new neighbor in k-covering(t). This condition is determined by checking that at least one of the variables of y has a value difference of k with respect to t (line 3) and the rest of the variables have a value difference lower or equal to k (line 2). The procedure `isComplete` determines if a k-covering(t) is complete. Firstly, it calculates the maximum number of tuples of k-covering(t): maxTup($k, |t|$) (see Equation 3). Subsequently, it checks if |last-covering(t)| is equal to maxTup($k, |t|$).

The procedure `addNeighbor` adds 1 to |last-covering(t)| iff $k = 1$ or (k-1)-covering(t) is complete. Algorithm 1 calls the procedure `addNeighbor` twice with two different tuples: t and y (y is a valid tuple of the reduced set of t) iff y is a new neighbor of k-covering(t) (line 10). With this process a new neighbor can be added to the last-coverings of two different tuples by doing only one check, which allows further reduction in computation time. Later, the algorithm checks if k-covering(y) is complete

Procedure addNeighbor(k,|last-covering(t)|,t)

1 **if** $k = 1$ **or** isComplete($k-1$,|last-covering(t)|,$|t|$) **then**
  |last-covering(t)| $\leftarrow$ |last-covering(t)| + 1;

(line 11). Checking the completeness of k-covering(t) is not necessary yet since there are possible new neighbors of k-covering(t) that are ordered in higher positions than k and therefore have not been analyzed yet. They will be analyzed when the tuple t will be an element of their reduced set.

As mentioned previously, Algorithm 1 does not compute k-covering(t) if (k-1)-covering(t) is not complete. However, this restriction could be deleted by skipping the completeness check of the coverings (developed by the isComplete procedure). In this case, we cannot use the principle of minimum distance from the constraint (see Section 4.2). In this paper, this principle is necessary because the objective is to find solutions located far away from the constraint boundaries. Nevertheless, the coverings computation does not require the completeness property. Thus, both the covering concepts for CSPs and Algorithm 1 (with the modification discussed above) can be also applied to other areas that do not require this property.

6 Modeling Robustness in a CSP as a WCSP

In this section we introduce an enumeration-based technique for modeling a CSP as a WCSP. The intention is to obtain a CSP model based on the |last-coverings| of the solutions of all the constraints. As argued earlier, |last-covering($s \downarrow_{var(C_i)}$)| for a solution s is a reasonable measure of its robustness for C_i, and it is moreover reasonable to use the sum of |last-covering($s \downarrow_{var(C_i)}$)| for each $C_i \in (\mathcal{C} \cup \mathcal{DC})$ as an approximation of the robustness of s for the CSP. The WCSP model is based on the assumption that the sum of the costs assigned to the tuples of each constraint determines how good a solution is for the WCSP. Thus, we model a CSP as a WCSP after obtaining its |last-coverings| with Algorithm 1.

Although there are other valued CSPs that could conceivably be used to model robustness, these models either involve operations that are questionable (e.g. probabilistic CSP, where valuations based on |last-coverings| would be multiplied) or are insufficiently discriminating (e.g. fuzzy CSP). In addition, the WCSP model adequately incorporates the enumeration aspect of coverings, unlike the other valued CSP models.

The modeling process begins by assigning a cost to each valid tuple t involved in each constraint, which represents its penalty as a function of its |last-covering(t)|. Tuples with the highest last-covering for C_i will have the lowest associated cost, because this value indicates the minimum distance of t from the bounds of C_i. As already noted, the latter can be taken as a measure of its robustness, because it is resistant to more possible changes over the constraint.

Definition 15. *We define* max-|last-covering(C_i)| = *max* $\{|$*last-covering*$(t)|, \forall t \in T(C_i)\}$.

The penalty of a valid tuple t for a constraint C_i without considering the rest of the constraints of the CSP is denoted as $p_i(t)$ (see Equation 4).

$$p_i(t) = \text{max-|last-covering}(C_i)| - |\text{last-covering}(t)| \tag{4}$$

However, this is not the final cost assigned to t. We must normalize the penalties of each C_i with respect to the other constraints, because the maximum possible size of the coverings depends on the arity of the tuples (see Equation 3). Otherwise, constraints with higher arities would have higher cost ranges and therefore higher penalties. In this case, we would be assuming that these constraints have a higher likelihood of undergoing restrictive modifications, which is not necessarily true according to the limited assumptions we are making for CSPs with discrete and ordered domains. By using a normalization process, we can achieve the same cost range for all the constraints. To obtain normalized scores, we use the maximum penalty assigned to the tuples of each constraint and the maximum penalty assigned to the tuples across all constraints (e is the number of constraints of the CSP according to Definition 1). In this way, the cost function for the tuples of C_i assigns a normalized cost to every tuple of C_i, which is denoted as $C_i(t \downarrow_{var(C_i)})$ (see Equation 5).

$$C_i(t \downarrow_{var(C_i)}) = \begin{cases} 0 & \text{if } t \in T(C_i) \text{ and } p_i(t) = 0 \\ \left\lfloor \frac{p_i(t) * max\{p_j(x), \forall j \in [1 \ldots e] \forall x \in T(C_j)\}}{max\{p_i(y), \forall y \in T(C_i)\}} \right\rfloor & \text{if } t \in T(C_i) \text{ and } p_i(t) \neq 0 \\ W, (W \approx \infty) & \text{if } t \notin T(C_i) \end{cases} \tag{5}$$

In line with the version of WCSP that we are using, $C_i(t \downarrow_{var(C_i)})$ assigns a cost of W to each tuple *t* that does not satisfy the constraint C_i because it is not a partial solution. Note that for valid tuples $C_i(t \downarrow_{var(C_i)}) \in [0, max\{p_j(x), \forall j \in [1 \ldots e] \forall x \in T(C_j)\}]$. A valid tuple t whose |last-covering(t)| = max-|last-covering(C_i)| ($p_i(t) = 0$) has an associated cost of 0. This tuple is not penalized because it has the highest likelihood of remaining valid when faced with future changes in C_i. On the other hand, if |last-covering(t)| $= 0$ for C_i, which means that t does not have any neighbour in its 1-covering(t) (t is completely non-robust for C_i), t has the maximum possible cost associated: $max\{p_j(x), \forall j \in [1 \ldots e] \forall x \in T(C_j)\}$.

Once the costs have been assigned, the WCSP is generated and solved using a WCSP solver. The solutions obtained for the modeled WCSP are also solutions of the original CSP. In addition, the best solution *s* with the minimum $\mathcal{V}(s)$ (see Equation 1) is taken to be one of the most robust solutions for the original CSP.

7 Example

In order to clarify the covering concepts, we present an example with three variables (*3-dimensional* hyperpolyhedron CSP). We have modeled this problem as well as the problems presented in Section 8 as WCSPs by following the WCSP file format[3]. In addition, ToulBar2[4] has been used for solving the resultant WCSPs.

[3] *http://graphmod.ics.uci.edu/group/WCSP_file_format*

[4] *http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/ToolBarIntro*

Example 3. Figure 2 shows a CSP *R*, which is composed of variables x_0, x_1 and x_2 with domain $D : \{0, 1, 2\}$. There are three extensional constraints: C_0 (3-ary), C_1 and C_2 (binary constraints) (Figures 2(a), 2(b) and 2(c), respectively). The valid tuples of the constraints and domains are represented with points; the invalid tuples are represented with crosses. Figure 2(d) shows the solutions of *R* ordered by their relative robustness, as assessed by our technique for calculating coverings.

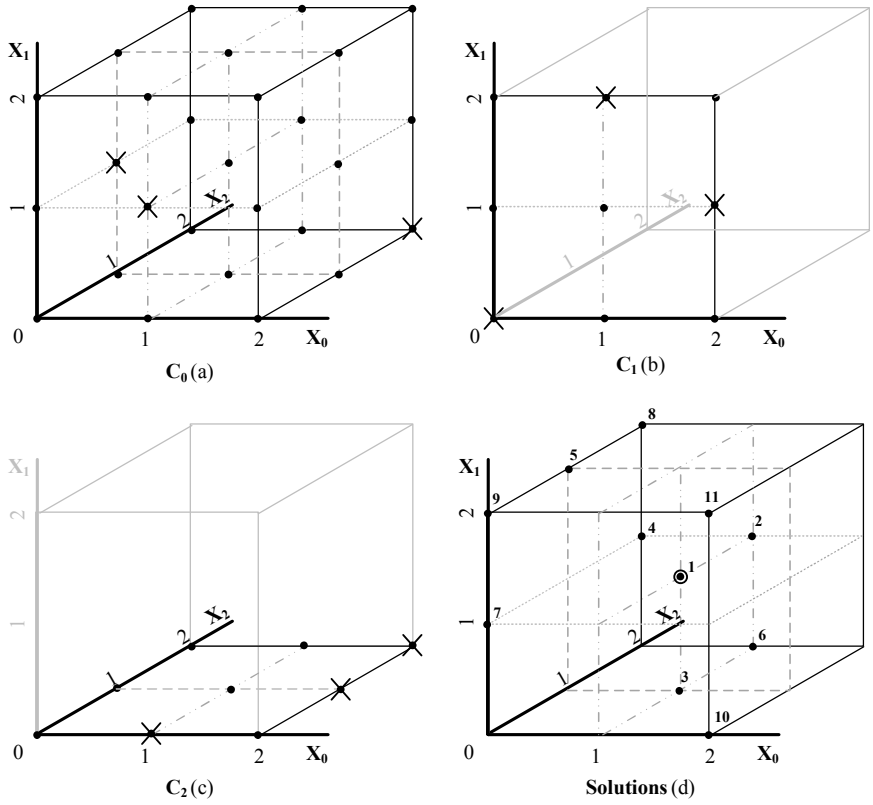

Fig. 2. Example of the 3-dimensional CSP *R*.

Table 1 shows the solutions of the CSP *R* in decreasing order of robustness (s_i). This order is inversely proportional to the $\mathcal{V}(s)$ obtained after solving the modeled WCSP. In addition, we present |last-covering(s)| for the solution space of *R*. Note that its calculation is viable because we are analyzing a toy problem. As expected, the robustness results obtained match with the |last-coverings| order in the solution space: the solutions with higher |last-coverings| in the solution space are identified as more robust by our technique (see the highlighted columns in Table 1).

Table 1 also shows the costs assigned by the constraints (C_i) and domains (X_i), whose sum is $\mathcal{V}(s)$ (see Equation 1). To clarify the process of cost assignment, we

explain one case, $C_0(s_2)$, in detail. Taking into account that *max-covering(R)*$= 1$, the penalty for $s_2 = (1, 1, 2)$ (before the normalization process) is $p_0(s_2) = 23 - 15 = 8$ (see Equation 4), since max-|last-covering$(C_0)| = 23$ (|last-covering$((1, 1, 1))| = 23$ for C_0, which is the maximum for C_0) and |last-covering$(s_2)| = 15$ for C_0. The associated cost (considering the normalization) is: $C_0(s_2) = \lfloor \frac{8*18}{18} \rfloor = 8$ (see Equation 5). Because $max\{p_0(y), \forall y \in T(C_0)\} = 18$ ($p_0((0, 2, 0)) = 23 - 5 = 18$) and this cost is also the maximum penalization for *R*, since $max\{p_j(x), \forall j \in [1 \dots e] \forall x \in T(C_j)\} = 18$.

Table 1. Solutions ordered by their robustness.

s_i	$s = (x_0, x_1, x_2)$	*|last-covering(s)|*	$\mathcal{V}(s)$	$X_0(s)$	$X_1(s)$	$X_2(s)$	$C_0(s)$	$C_1(s)$	$C_2(s)$
1	(1,1,1)	10	0	0	0	0	0	0	0
2	(1,1,2)	6	35	0	0	18	8	0	9
3	(1,0,1)	6	36	0	18	0	9	9	0
4	(0,1,2)	6	67	18	0	18	13	9	0
5	(0,2,1)	6	67	18	18	0	14	13	4
6	(1,0,2)	4	68	0	18	18	14	9	9
7	(0,1,0)	4	72	18	0	18	14	9	13
8	(0,2,2)	4	93	18	18	18	17	13	9
9	(0,2,0)	3	98	18	18	18	18	13	13
10	(2,0,0)	2	102	18	18	18	17	13	18
11	(2,2,0)	1	107	18	18	18	17	18	18

The best solution found is the solution $s1 = (x_0 = 1, x_1 = 1, x_2 = 1)$ and its |*last-covering*$(s1)| = 10$ in the solution space, which is the highest for R (see Figure 2(d)). As previously mentioned, the solution s with the highest |last-covering(s)| in the solution space, has the highest likelihood of remaining valid faced with future possible restrictive modifications over the bounds of the solution space. Therefore, it is considered the most robust solution for the original CSP. In contrast, the solution $s_{11} = (x_0 = 2, x_1 = 2, x_2 = 0)$ is classified by our technique as the least robust solution. Note that s_{11} only has one neighbor in the solution space (see Figure 2(d)).

8 Experimental Results

In this section, we present results from experiments designed to evaluate the behaviour of our technique for calculating coverings. To the best of our knowledge, there are no benchmarks of DynCSPs (see Definition 4) in the literature, so we generated 500 DynCSPs composed of $l+1$ static CSPs: $\langle CSP_{(0)}, CSP_{(1)}, ..., CSP_{(l)} \rangle$, where $CSP_{(0)}$ is generated randomly by RBGenerator 2.0 [5] and each $CSP_{(i)}$ is derived from $CSP_{(i-1)}$ by making a restrictive modification in some bound (constraint or domain). Thus, an invalid tuple located next to any bound is selected randomly and all the tuples surrounding it to a distance of 1 are invalidated. Thus, only valid tuples located on a bound became

[5] *http://www.cril.univ-artois.fr/ lecoutre*

invalid. Thus, l indicates the number of restrictive changes that the solution was able to resist, and can be considered a measure of the robustness of the solutions. In order to select equally constraints and domains, we have selected them based on their relative frequency. All tests were executed on a Intel Core i5-650 Processor (3.20 Ghz).

In addition to assessing our technique for max-covering(CSP), we analyzed the robustness of solutions obtained with the proactive technique for maximizing the number of repairable variables for (1,0)-super-solutions by using Branch and Bound and MAC+ algorithms ([8]) and fixing the cutoff to 200 seconds. The main reason for choosing this technique for comparison is that, like our technique, it does not require specific additional information about the future possible changes. In addition, a solution calculated by an ordinary CSP solver has been computed to determine if there are cases in which all solutions have similar robustness. For both techniques, values were selected in lexicographical order. Note that neither of these techniques searches for robust solutions in the same way that our technique does, nor are they based on the same assumptions regarding changes that are inherent in the structure of CSPs with discrete and ordered domains.

Figure 3 shows a robustness analysis when tightness of the constraints is varied. The measure on the left vertical axis is the number of supported changes (both means (continuous lines) and standard deviations are shown in the figure), while the right vertical axis shows the modeling time required by our technique (dashed line). We can observe that with the exception of very highly restricted intances, the mean number of changes before solution breakage for solutions obtained by the k-covering technique is significantly higher than the means obtained with the other two solutions. For instances that are very highly restricted, the number of changes allowed by solutions obtained by the three methods is similar. This is because for this type of 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 instance, for the maximum tightness evaluated (0.9), there are only 5 solutions. Even in this case the k-covering algorithm was able to find solutions that remained valid for a few more changes than the solutions found by the other algorithms.

All of the experimental results were evaluated statistically, first with a two-factor Analysis of Variance (ANOVA), followed by the Tukey HSD test for differences between pairs of individual means ([7, 17]). The ANOVA gave F values that were highly significant statistically. Given the small values for HSD (about 0.1 and 0.08), nearly all differences between individual means were statistically significant for $p = 0.01$. Further tests have been done with the other CSP parameters (number of variables, domain size, constraint graph density and constraint arity), and we have obtained the same general results: solutions found with our technique remained valid for a greater number of restrictive changes than the other techniques, and the difference is marked when the problems are only slightly constrained CSPs.

The modeling time required by our technique is higher for slight tightness values. This is is related to the number of valid tuples of the constraints, which it has a strong impact in the time spent by Algorithm 1 for finding the coverings.

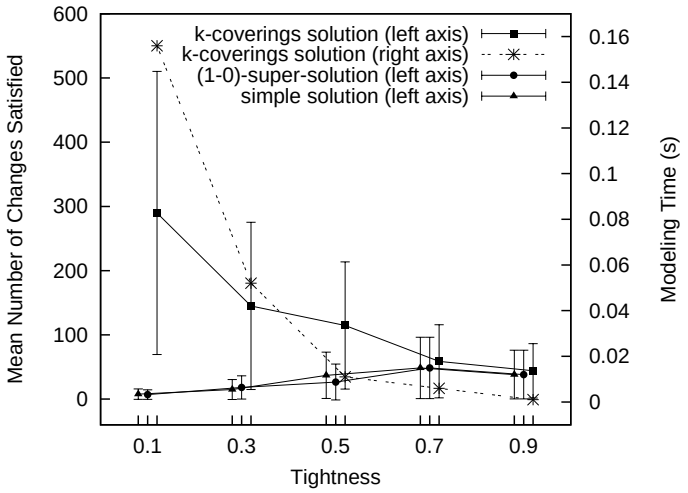

Fig. 3. Robustness analysis ($< arity = 2, |\mathcal{X}| = 40, |D| = 25, |\mathcal{C}| = 120 >$).

9 Conclusions

In this paper we have extended the concept of robustness to CSPs with discrete and ordered domains where only limited assumptions are made about the kinds of changes, commensurate with the structure of these problems. In this context, it is reasonable to assume that the original bounds of the solution space may undergo restrictive modifications. Therefore, the main objective in searching for robust solutions is to find solutions located as far away as possible from these bounds.

In addition, we presented an enumeration-based technique for modeling these CSPs as WCSPs by assigning a cost to each valid tuple of every constraint. The cost of each valid tuple t is obtained by calculating |last-covering(t)| for the corresponding constraint. The obtained solution for the WCSP is a solution for the original CSP that has a higher probability of remaining valid after changes in the original problem.

Another contribution of this paper is the framework introduced in Section 4, which involves the novel concept of an 'onion topology' as applied to CSPs, as well as concepts and definitions built around the idea of coverings. To the best of our knowledge, Algorithm 1 is the first practical method for calculating coverings for CSPs.

In experimental tests we have shown that these techniques can dramatically outperform both ordinary CSP algorithms and algorithms that find (1,0)-super-solutions (or maximize the number of repairable variables in case that there does not exist a (1,0)-super-solution) under a variety of conditions where there are real differences in the robustness of solutions that might be obtained (when the problem is not so tight that there are only a few valid solutions).

Acknowledgements. This work has been partially supported by the research project TIN2010-20976-C02-01 and FPU program fellowship (Min. de Ciencia e Innovacin, Spain).

Bibliography

[1] M. Bofill, D. Busquets, and M. Villaret. A declarative approach to robust weighted Max-SAT. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP-10)*, pages 67–76, 2010.

[2] L. Climent, M. Salido, and F. Barber. Reformulating dynamic linear constraint satisfaction problems as weighted CSPs for searching robust solutions. In *Proceedings of the 9th Symposium of Abstraction, Reformulation, and Approximation (SARA-11)*, pages 34–41, 2011.

[3] R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI-88)*, pages 37–42, 1988.

[4] H. Fargier and J. Lang. Uncertainty in constraint satisfaction problems: a probabilistic approach. In *Proceedings of the Symbolic and Quantitative Approaches to Reasoning and Uncertainty (EC-SQARU-93)*, pages 97–104, 1993.

[5] H. Fargier, J. Lang, and T. Schiex. Mixed constraint satisfaction: A framework for decision problems under incomplete knowledge. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 175–180, 1996.

[6] D. Fowler and K. Brown. 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)*, pages 500–504, 2000.

[7] W. Hays. *Statistics for the social sciences (2nd. edit.)*, volume 410. Holt, Rinehart and Winston New York, 1973.

[8] E. Hebrard. *Robust Solutions for Constraint Satisfaction and Optimisation under Uncertainty*. PhD thesis, University of New South Wales, 2006.

[9] H. Herrmann, C. Schneider, A. Moreira, J. Andrade Jr, and S. Havlin. Onion-like network topology enhances robustness against malicious attacks. *Journal of Statistical Mechanics: Theory and Experiment*, 2011(1):P01027, 2011.

[10] J. Larrosa and T. Schiex. Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence*, 159:1–26, 2004.

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

[12] T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: hard and easy problems. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 631–637, 1995.

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

[14] R. Wallace and E. Freuder. Stable solutions for dynamic constraint satisfaction problems. In *Proceedings 4th International Conference on Principles and Practice of Constraint Programming (CP-98)*, pages 447–461, 1998.

[15] T. Walsh. Stochastic constraint programming. In *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-02)*, pages 111–115, 2002.

[16] F. William. *Topology and its applications*. John Wiley & Sons, 2006.

[17] B. Winer. *Statistical principles in experimental design (2nd. edit.)*. McGraw-Hill Book Company, 1971.

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