Enhancing Neighbourhood Substitutability Thanks to Singleton Arc Consistency

Dominique D'Almeida Lakhdar Saïs
CRIL-CNRS, Université d'Artois
Rue Jean Souvraz SP18
F-62307 Lens Cedex France
{dalmeida,sais}@cril.fr

Abstract

In this paper, a semantic generalization of neighbourhood substitutability is presented. Instead of the syntactical concept of supports, our generalization originally exploits a new semantic measure based on Single Arc-Consistency (SAC) checking. This generalization is then exploited in two different ways. Firstly, a new pretreatment of constraints networks is proposed. Secondly, as SAC is a basic operation in backtrack search-based algorithms like MAC*, we show that our generalization can be easily applied dynamically. The experimental results of our approach show interesting improvements on some classes of* CSP *instances and demonstrate the feasibility of the dynamic integration of our generalized neighbourhood substitutability.*

1. Introduction

The Constraint Satisfaction Problem (CSP) is a general modelisation framework to solve combinatorial problems appearing in a variety of application domains. They involve finding solution in a *constraints network* i.e. finding *values* for network *variables* subject to *constraints* on which combinations are acceptable. This decision problem is NP-Complete, and it conducts a lot of researchers to design new algorithmic approaches to solve as efficiently as possible the general case.

The usual technique to solve CSPs is systematic backtracking. It repeatedly chooses a variable, attempts to assign it one of its values, and then goes to the next variable, or backtracks in case of failure. This technique is at the basis of almost all the CSP solving engines. But if we want to tackle highly combinatorial problems, we need to enhance this basic search procedure with clever improvements. They concern several important components of the constraint solving engines. Some of them usually qualified as look-ahead based improvements aim to choose the best decision or to deduce further information in order to forecast the future of the search. This mainly concerns pretreatment, constraint propagation and variable and value ordering heuristics. The second kind of improvements usually known as look-back based enhancements try to avoid thrashing by analysing the past of the search or conflicts. Conflict analysis might help to achieve non chronological backtracking and to deduce nogoods (set of inconsistent value assignments) that can be exploited to avoid the exploration of the same portions of the search space.

However, managing such a set of nogoods might be time consuming and hard to integrate in CSP search algorithms. In look-ahead based strategies, it is possible to consider constraints propagation techniques, which allow, for example, to eliminate local inconsistent values. Also, there exists some efficient treatments combining both look-back and look-ahead strategies. For example, the variable ordering heuristic dom/wdeg which computes the next variable to assign according to the weight of constraints i.e. the number of times it appears violated during search.

Other kinds of treatments also recognized as important for tackling real world CSP instances, are those based on the detection and exploitation of different forms of problem structures. Among them, we can cite symmetries [2, 9], or weaker forms of symmetry such as neighbourhood interchangeability or substitutability of values [1, 4, 6]. These last kind of symmetries are easier to detect (checking constraint supports) and to exploit (eliminating values from the domain of the variables), whereas the problem of detecting general symmetries is equivalent to graph automorphism, for which no polynomial time algorithm is known. Usualy, these different kinds of symmetries, in their syntactical forms, are detected during the preprocessing phase and exploited either in a static or a dynamic way [5, 7].

In our framework, the proposed approach is based on a semantic generalization of the neighbourhood substitutability (NS) and, in the same way, of the neighbourhood interchangeability (NI) of values. These generalizations are obtained by substituting the syntactical concept of supports by a semantic measure of the constraints propagation obtained

by a classical use of singleton arc-consistency. Our proposed generalization provides two important benefits. First, it extends the power of SAC [3], one of the well-known and used strong local consistency. The second benefit lies in the simplicity of its dynamic integration to CSP solvers. Indeed, since singleton arc-consistency is a basic step of the MAC-like algorithms, it is possible to exploit our generalization in a dynamic context after an assignment of a value to a variable. More precisely, each time a variable is assigned a value and arc consistency is maintained, some information on the current state of the constraints network are recorded. When another value is assigned to the same variable, our approach is able to avoid the exploration of the same state.

The rest of the paper is organized as follows. After some technical background and preliminary definitions, our generalization of the neighbourhood substitutability is then described in section 3.1. Sections 3.2 and 3.3 present the static and dynamic integration of our generalized neighbourhood substitutability and interchangeability and their theoretical complexity. Experimental results are provided (Section 4) and discussed before concluding.

2. Preliminary definitions and technical background

A Constraints Network (CN) $\mathcal{P}$ is a pair $(\mathcal{X}, \mathcal{C})$ with $\mathcal{X} = \{x_1, ..., x_n\}$ a set of *variables* and $\mathcal{C} = \{c_1, ..., c_e\}$ a set of *contraints* between variables in $\mathcal{X}$. To each variable x_i of $\mathcal{X}$ is associated a finite definition set $dom(x_i) = \{v_1, ..., v_d\}$, called x_i *domain*. An *assignment* (resp. *refutation*) is a pair $(x, v) \in (\mathcal{X}, dom(x))$, denoted $x = v$ (resp. $x \neq v$) i.e. $dom(x) = \{v\}$ (resp. v is removed from $dom(x)$).

Each constraint c_i is defined by a pair (s_i, r_i) such that : $s_i \subseteq \mathcal{X}$ is called *scope* of c_i and refers to the variables involved in c_i. We denote by $scp(c_i)$ the scope s_i and $card(s_i) = a_i$ the *arity* of the constraint c_i. $r_i \subseteq \prod_{x \in s_i} dom(x)$ is called a *relation* and stands for a set of *tuples* satisfying the constraint c_i.

Using the scope of the constraints, two distinct variables x and y are *neighbour* iff there exists a constraint $c_i \in \mathcal{C}$, called *neighbour constraint* of x and y, such that $\{x, y\} \subseteq scp(c_i)$. A set of variables $X \subseteq \mathcal{X}$ (resp. constraints $C \subseteq \mathcal{C}$) is in the *neighbourhood* of a variable x ($x \notin X$) if and only if for all $x_i \in X$ (resp. $c_i \in C$), x is a neighbour of x_i (resp. c_i).

Considering the relation of the constraints, we denote by $rel(c_i) = r_i$ the definition domain of c_i. Each tuple $t \in rel(c_i)$ is defined as an $a_i - tuple$ $(v_1, \ldots, v_{a_i})$. For a given variable $x_j \in scp(c_i)$, $t[x_j]$ (resp. $t[\hat{x_j}]$) denotes the value taken by x_j in t (resp. the tuple t without the value associated to the variable x_j). Let $t \in rel(c_i)$ and $x \in scp(c_i)$ such that $t[x] = v$, $t[\hat{x}]$ is called a support of $x = v$. The set of supports for $x = v$ in c_i is denoted $supports(c_i|_{x=v})$.

We now define the notion of *instantiation* for a CN $\mathcal{P} = (\mathcal{X}, \mathcal{C})$. The instantiation $\mathcal{I}_X$ of the variables in $X \subseteq \mathcal{X}$ is defined as a set of assignment $\{x = v | x \in X, v \in dom(x)\}$. $\mathcal{I}_X$ is called *full instantiation* if $X = \mathcal{X}$ and *partial instantiation* otherwise. Let $X \subseteq Y$, $Y \subseteq \mathcal{X}$ and $\mathcal{I}_Y$ be an instantiation, we define $\mathcal{I}_Y[X]$ (resp. $\mathcal{I}_Y[\hat{X}]$) the tuple $\prod_{v \in dom(x)}$ such that $x \in X$ (resp. $x \in Y$ and $x \notin X$). A solution for a CN $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ is an instantiation $\mathcal{I}_\mathcal{X}$ satisfying all of the constraints i.e. $\forall c_i \in \mathcal{C}, \mathcal{I}_\mathcal{X}[scp(c_i)] \in rel(c_i)$. A CN $\mathcal{P}$ is said *consistent* (or *satisfiable*) if it admits at least one solution, and *inconsistent* (or *unsatisfiable*) otherwise. The Constraints Satisfaction Problem CSP is the problem (*NP-Complete*) of deciding if a given CN $\mathcal{P}$ admits or not a solution. We talk about the resolution of a CSP instance, defined by a CN $\mathcal{P}$, as the search of any solution or the proof of its unsatisfiability.

In the sequel, to solve a CN, we consider the well-known depth-first tree-based search algorithms that maintains arc-consistency at each node of the search tree (MAC). This decision/propagation based algorithm is complete that is to say it necessarily provides an answer for the satisfiability problem. In a CN, a value $v \in dom(x)$ is *arc-consistent* in a constraint c_i ($x \in scp(c_i)$) iff there exists a support for $x = v$ in c_i. A variable is arc-consistent in a constraint iff all its values are arc-consistent. A variable is arc-consistent iff it is arc-consistent for its entire neighbour set of constraints. A CN is arc-consistent iff all its variables are arc-consistent. Without exception due to the arity of the constraints, Arc-Consistency is called *Generalized Arc-Consistency* (GAC).

We define $\phi(\mathcal{P})$ the CN obtained after applying the constraints propagation algorithm ϕ on the CN $\mathcal{P}$. For instance, $\phi =$ GAC means that all the values non arc-consistent in $\mathcal{P}$ are removed. Afterwards, we will denote the CN $\mathcal{P}$ obtained after enforcing arc-consistency by AC($\mathcal{P}$). If there exists a variable with an empty domain in $\phi(\mathcal{P})$, we note such inconsistency of $\mathcal{P}$ by $\phi : \phi(\mathcal{P}) = \bot$. The subnetwork obtained by assigning $x = v$ is denoted by $\mathcal{P}|_{x=v}$. The *Singleton Arc-Consistency* filtering (SAC) is a strong local consistency which guarantees $\phi(\mathcal{P}|_{x=v}) \neq \bot$, for each pair variable-value (x, v). We additionally denote $\mathcal{P}^*|_{x=v}$ as the network $\mathcal{P}|_{x=v}$ with the variable x and its neighbour constraints disconnected. Also, for a CN $\mathcal{P}$, we define a subnetwork $\mathcal{P}'$ of $\mathcal{P}$ as a CN with the same set of variables such that for each variable x of $\mathcal{P}'$, its domain is included in the domain of the variable x in $\mathcal{P}$. A partial instantiation $\mathcal{I}_X$ is locally consistent iff $\phi(\mathcal{P}|_{\mathcal{I}_X}) \neq \bot$ and is globally consistent if it can be extended to a solution of $\mathcal{P}$. A full and consistent instantiation is a *solution*.

When MAC-like solving algorithm is used, one needs to specify the kind of branching used to build the search tree. Indeed, it is possible to consider different approaches like binary branching (*dual-branching* or *dual*) or non binary branching (*dway-branching* or *dway*). For dual-branching,

at each node in the search tree, a pair variable-value (x, v) is selected and two cases are then considered : the assignment $x = v$ and, if it did not lead to a solution, the refutation $x \neq v$ followed by a new choice for the pair variable-value. For dway-branching, at each step, a variable x is selected and assigned to a value v in $dom(x)$. If it is inconsistent, v is eliminated from the domain of x and another value in $dom(x)$ is chosen and then assigned. This process occurs until a refutation of all the value in the domain is proved. Each of those approaches has its assets, however the dual is shown to be theoretically stronger than dway [8].

3. SAC-based generalization of neighbourhood substitutability

3.1. A formal description

The substitutability and, the related notion of interchangeability are the main keywords of our contribution. These weak forms of symmetries have been subjects of many research works. In [6], Freuder introduced the two well-known forms of symmetry: *full* and *local* substitutability and interchangeability.

For a CN $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ and two distinct pairs $(x, v), (x, v') \in (\mathcal{X}, dom(x))$, v is *fully substitutable* by v' iff for each solution $\mathcal{I}_{\mathcal{X}}$ such that $\mathcal{I}_{\mathcal{X}}[x] = (v)$, there exists a solution $\mathcal{I}'_{\mathcal{X}}$ such that $\mathcal{I}'_{\mathcal{X}}[x] = (v')$ and $\mathcal{I}'_{\mathcal{X}}[\hat{x}] = \mathcal{I}_{\mathcal{X}}[\hat{x}]$. It is easy to see that the full interchangeability and the full substitutability are related. Indeed, two values v and v' are fully interchangeable iff v is fully substitutable by v' and v' is fully substitutable by v. So, two fully interchangeable values are fully substitutable, the converse is not true. As we need to enumerate all the solutions, checking for full interchangeability or substitutability of two values is clearly intractable. However, a weakening of those forms gives rise to a local and tractable forms of substitutability and interchangeability of two values : *neighbourhood* substitutability and interchangeability.

These local forms are described by restricting both substitutability and interchangeability between two values of a variable x on its neighbourhood variables. In this paper, we focus on the generalization of neighbourhood substitutability. Neighbourhood interchangeability can be defined in similar way. In our framework, exploiting our generalization of neighbourhood substitutability is sufficient. Let us reformulate and generalize the definition of neighbourhood substitutability given in [6] for the binary constraints networks.

Definition 1 *Let $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ be a* CN *and $(x, v), (x, v')$ two pairs in $(\mathcal{X}, dom(x))$. v is substitutable by v' in the neighbourhood of x if and only if, for each constraint c_i neighbour of x, $supports(c_i|_{x=v}) \subseteq supports(c_i|_{x=v'})$.*

For our purposes, let us derive a more convenient definition of neighbourhood substitutability from full substitutability. This new definition is obtained by considering partial and consistent instantiation instead of solution.

Definition 2 *Let $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ be a* CN*, $(x, v), (x, v')$ be two different pairs in $(\mathcal{X}, dom(x))$ and X the set of variables including x and its neighbours. v is substitutable by v' in the neighbourhood of x if and only if, for each partial and consistent instantiation $\mathcal{I}_X$ such that $\mathcal{I}_X[x] = (v)$, it exists a partial and consistent instantiation $\mathcal{I}'_X$ such that $\mathcal{I}'_X[x] = (v')$ and $\mathcal{I}'_{\mathcal{X}}[\hat{x}] = \mathcal{I}_{\mathcal{X}}[\hat{x}]$.*

The definition 2 is stronger than definition 1. Indeed, if for each partial and consistent instantiation $\mathcal{I}_X$ such that $\mathcal{I}_X[x] = (v)$, it exists a partial and consistent instantiation $\mathcal{I}'_X$ such that $\mathcal{I}'_X[x] = (v')$ and $\mathcal{I}'_{\mathcal{X}}[\hat{x}] = \mathcal{I}_{\mathcal{X}}[\hat{x}]$, then we can deduce inclusion in term of supports. Obviously, the converse is not true. Intuitively, the notion of supports may be associated to the notion of constraints propagation to introduce the following propositions :

Proposition 1 *Let $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ be a* CN*, $(x, v), (x, v')$ be two different pairs in $(\mathcal{X}, dom(x))$, and ϕ be a constraints propagation operator. v is substitutable by v' in the neighbourdhood of x if and only if the* CN *$\phi(\mathcal{P}^*|_{x=v})$ is a subnetwork of $\phi(\mathcal{P}^*|_{x=v'})$.*

Proposition 2 *Let $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ be a* CN*, $(x, v), (x, v')$ be two pairs in $(\mathcal{X}, dom(x))$ such that v is substitutable by v' in the neighbourhood of x, and ϕ be a constraints propagation operator. If $\phi(\mathcal{P}|_{x=v'}) = \perp$ then $\phi(\mathcal{P}|_{x=v}) = \perp$.*

The two propositions above are central in our contribution. The proof follows from the definition 2 of NS. For proposition 1, since v is substitutable by v' in the neighbourhood of x, we have, as mentioned above by linking the definitions 2 with 1, that $supports(c_i|_{x=v}) \subseteq supports(c_i|_{x=v'})$. By iterating the constraints propagation until reaching a fix point, the supports $supports(c_i|_{x=v})$ of $\phi(\mathcal{P}|_{x=v})$ are included in the supports $supports(c_i|_{x=v'})$ for all constraints c_i. Hence the CN $\phi(\mathcal{P}^*|_{x=v})$ is a subnetwork of $\phi(\mathcal{P}^*|_{x=v'})$. On the other hand, we have $\phi(\mathcal{P}^*|_{x=v})$ a subnetwork of $\phi(\mathcal{P}^*|_{x=v'})$, and ϕ an operator able to check the local consistency of the network. Then, for each partial and locally consistent instantiation $\mathcal{I}_X$ of $\phi(\mathcal{P}|_{x=v})$ (X the set of variables including x and its neighbours), there exists a partial and locally consistent instantiation $\mathcal{I}'_X$ of $\phi(\mathcal{P}|_{x=v'})$ such that $\mathcal{I}'_X[\hat{x}] = \mathcal{I}_X[\hat{x}]$. Including the decision $x = v$ and $x = v'$ in each instantiation, we have $\mathcal{I}_X[x] = (v)$ and $\mathcal{I}'_X[x] = (v')$ respectively. Therefore, v is substitutable by v' in the neighbourhood of x using the definition 2.

Besides, to prove the proposition 2, we can use proposition 1. Each solution for the CN $\phi(\mathcal{P}^*|_{x=v})$ is also a solution for $\phi(\mathcal{P}^*|_{x=v'})$. Therefore, if the CN $\phi(\mathcal{P}^*|_{x=v'})$ is

inconsistent, then $\phi(\mathcal{P}^*|_{x=v})$ is too. As the inconsistency of $\phi(\mathcal{P}^*|_{x=v})$ induces the inconsistency of $\phi(\mathcal{P}|_{x=v})$, this completes the proof.

Our goal is to efficiently exploit the previous properties in a pretreatment phase using SAC or dynamically in MAC based algorithms.

Of course, before exploiting the substitutable values, we have to satisfy the substitutability condition. To do that, we just have to use the definition 1 and to check the inclusion between supports of the constraints after achieving a local consistency ϕ. Observe that the use of supports after maintaining a local consistency ϕ to compute inclusion between states is equivalent to checking inclusion between the subnetworks $\phi(\mathcal{P}^*|_{x=v})$ and $\phi(\mathcal{P}^*|_{x=v'})$. Also, since different local consistency operators may give different results on the same network, the efficiency, in term of the number of detected substitutable values, depends on the used operator ϕ.

In the sequel, we will use Arc-Consistency as a local consistency operator, usually exploited in SAC and MAC algorithms.

3.2. Preprocessing with substitutability

As mentioned above, since arc consistency is used, our proposed generalization is clearly adapted to SAC based filtering technique [3]. For a CN $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ and a variable $x \in \mathcal{X}$, a value $v \in dom(x)$ is *singleton arc-consistent* (SAC) iff $AC(\mathcal{P}|_{x=v}) \neq \bot$. A variable x is SAC iff, for all $v \in dom(x)$, v is SAC. To establish SAC each pair (x, v) is verified, then it is clearly convenient for the integration of our proposed generalization of neighbourhood substitutability (see proposition 1).

The pretreatment resulting from the integration of our approach, called SNS for "*SAC and Neighbouhood Substitutability*", is defined by two steps.

The first one, allows to find substitutable values. To this end, the supports of the neighbour constraints after performing $\mathsf{AC}(\mathcal{P}^*|_{x=v})$, are stored in a structure (*state*) corresponding to the tuples of the constraints in neighbourhood of x (denoted $\Delta_{x=v}$). Inclusion between such states can be achieved by checking inclusion between supports. Formally, $\Delta_{x=v} \subseteq \Delta_{x=v'}$ iff, $\forall c$ in neighbourhood of x, $\forall t \in \Delta_{x=v}(c)$, there exists a tuple $t' \in \Delta_{x=v'}(c)$ such that $t[\hat{x}] = t'[\hat{x}]$.

The second step exploits proposition 1. In the case of SNS, if v is substitutable by v' in the neighbourhood of x, a solution including the assignment $x = v$ allows to deduce a solution for $x = v'$. Consequently the value v can be removed from x.

Interestingly, associated with SAC, our approach can eliminate both singleton arc inconsistent values and generalized neighbourhood substitutable values. This clearly shows that our preprocessing is stronger than SAC. In the algorithm 1, only the lines from 7 to 10 are dedicated for checking substitutable values, the other lines describe the classical SAC algorithm. This really shows the simplicity of such integration. Therefore, the time and space complexity for such supports inclusion computation directly depend on the constraints arity. If a constraint c involves a variables with maximal domain size d, the constraint may have, in the worst case, d^a supports.

Algorithm 1: SNS

```
   Input: P = (X, C) : a CN
   Output: SNS(P)
1  begin
2      repeat
3          foreach pair (x, v) ∈ (X, dom(x)) do
4              if AC(P|x=v) = ⊥ then
5                  Remove (x, v)
6              else
7                  Store Δx=v
8                  foreach (x, v') proved SAC do
9                      Δx=v ⊆ Δx=v' ⇒ Remove (x, v); Break
10                     Δx=v' ⊂ Δx=v ⇒ Remove (x, v')
11     until no pair (variable, value) is removable
12 end
```

Before analyzing the worst-case algorithmic complexity of SNS, let us remind that we consider CN with n variables and domain size at most d, e constraints of maximal arity r.

For the space complexity, let us first evaluates the space needed by a state defined by a pair (variable,value) (x, v). Since the arity of constraints is r and each variable has $r-1$ neighbour variables. Then, there is d^{r-1} supports by constraint and thus $(r-1)d^{r-1}$ values for each constraint. In the worst case, since each variable is involved in the e constraints, we deduce that there is $e(r-1)d^{r-1}$ values by state in the worst-case. To compare the states associated to each value of a given variable, SNS needs to store d states. Consequently, the worst-case space complexity is in $O(erd^r)$ for storing all the states during SNS pretreatment.

To compute the worst-case time complexity of SNS, we have to keep in mind that this algorithm is embedded in a SAC algorithm. First, we can see that a loop (line 3) is executed for each variable-value pair, i.e nd times. The iteration includes the arc-consistency filtering (1), the storage of the states associated to each pair (2) and the comparison with the existing states for the same variable (3). The worst-case time complexity of GAC is in $O(erd^r)$ (1). To store a state, we need to check its $e(r-1)d^{r-1}$ values (2). Each of these states must be compared with the previous stored states. The inclusion is checked in both directions, which needs $d-1$ inclusion checks to each pair variable-value. Inclusion of two states implies to see the whole states, that is to say erd^r values for inclusion checks (3). To sum up, we have a worst-case time complexity for the iteration of $nd \times (erd^r + e(r-1)d^{r-1} + erd^r)$, thus $O(ernd^{r+1})$. However, since in a classical SAC algorithm, a value removal

modifies the CN, the process has to be repeated. As the network has nd values, the iteration have, in the worst-case, to be executed nd times. The worst-case time complexity of our SNS preprocessing is in $O(ern^2d^{r+2})$.

In spite of high time and space complexity due to the use of an exponential number of supports, the efficiency of this approach in a practical case depends, obviously, on the implementation and used data-structures. Also, since the complexity depends on the arity of the constraints, it is interesting to investigates the particular case of binary constraints. In this last case, the worst-case space (respectively time) complexity is $O(nd^2)$ (respectively $O(n^3d^4)$). It is important to note that such complexity is comparable to those of the classical SAC algorithm.

3.3. Dynamic search for substitutability

The implementation of the SNS preprocessing above illustrates all the treatments that can be performed dynamically to detect NS.

First of all, the dynamic side of our integration can be related to the repeated modification of the CN and then of the states associated to the neighbourhood of the variables. In the MAC-based algorithm, maintaining arc-consistency in the network is performed for all decisions. Using the assignment and constraint propagation process, the detection of the neighbourhood substitutable values for a variable can easily be embedded, and computed at each node of the search tree.

Like in SNS approach, the supports in the neighbourhood of a variable are stored after each assignment $x = v$ which is followed by constraints propagation. The states associated to a given variable correspond to the states describing the last assignment of a variable to a value (the current state) and its previously refuted values (past states). Using proposition 2, if the current state $\Delta_{x=v}$ is included in a past state $\Delta_{x=v'}$, then the value v is substitutable by v' in the neighbourhood of x. However, as $AC(\mathcal{P}|_{x=v'}) = \bot$ then $AC(\mathcal{P}|_{x=v}) = \bot$. So, v has to be removed from the domain of x, avoiding to explore redundant parts of the search space.

The lines from 4 to 6 of the MAC_{NS} algorithm 2 show each step defined above, integrated to the classical dual-branching MAC-based algorithm.

Obviously, this approach can be built in dway-branching scheme since the difference between dual and dway are independent from our implementation. However, in spite of the easy integration, the condition in line 12 of the MAC_{NS} algorithm underline an important fact in our dynamic approach. Indeed, the comparison between states have to be done under some conditions, related to the state of the network before the last assignment. Without no focus on particular case, a necessary condition to the comparison between

Algorithm 2: MAC_{NS}

```
   Input: P = (X, C) : a CN
   Output: ⊤ if P is consistent, ⊥ otherwise
1  begin
2      while there exists an unassigned variable do
3          Choose (x, v) ∈ (X, dom(x)) such that x is not assigned
4          Do GAC(P|x=v) and store the state Δx=v
5          foreach stored state Δx=v' do
6              Δx=v ⊆ Δx=v' ⇒ P ← ⊥, break
7          while P = ⊥ and ∃ an assigned variable do
8              Backtrack to the previous decision x' = v'
9              Remove v' from dom(x')
10             Apply arc-consistency on P
11             if P = ⊥ then
12                 Delete states which are stored after the current level
13         if P = ⊥ and there is no assigned variable then
14             return ⊥
15     return ⊤
16 end
```

states follows from instantiations which involve them. For instance, can a state stored at the root of the search tree be compared to a state produced at any node of the search tree? The answer is given in the following proposition.

Proposition 3 *Let $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ be a* CN*, $(x, v), (x, v')$ two pairs in $(\mathcal{X}, dom(x))$, and ϕ a constraint propagation operator. Let $\mathcal{I}_X$ be a consistent instantiation such that $X \subseteq \mathcal{X} \setminus \{x\}$. v is substitutable by v' in the neighbourhood of x if and only if the* CN *$\phi((\mathcal{P}|_{\mathcal{I}_X})^*|_{x=v})$ is a subnetwork of $\phi(\mathcal{P}^*|_{x=v'})$.*

Using proposition 1, the proof follows. If v is substitutable to v' in the neighbourhood of x, then $\phi(\mathcal{P}^*|_{x=v})$ is a subnetwork of $\phi(\mathcal{P}^*|_{x=v'})$. For any instantiation $\mathcal{I}_X$ such that $X \subseteq \mathcal{X} \setminus \{x\}$, $\phi(\mathcal{P}|_{\mathcal{I}_X, x=v})$ is a subnetwork of $\phi(\mathcal{P}|_{x=v})$. Hence, $\phi((\mathcal{P}|_{\mathcal{I}_X})^*|_{x=v})$ is a subnetwork of $\phi(\mathcal{P}^*|_{x=v})$. On the other hand, if $\phi((\mathcal{P}|_{\mathcal{I}_X})^*|_{x=v})$ is a subnetwork of $\phi(\mathcal{P}^*|_{x=v'})$ then supports of $\phi((\mathcal{P}|_{\mathcal{I}_X})^*|_{x=v})$ are included in supports of $\phi(\mathcal{P}^*|_{x=v'})$, in particular in the neighbourhood of x. Consequently, v is substitutable to v' in the neighbourdhood of x iff $\phi((\mathcal{P}|_{\mathcal{I}_X})^*|_{x=v})$ is a subnetwork of $\phi(\mathcal{P}^*|_{x=v'})$.

However, this result only holds if the inclusion test between the state obtained after instantiation and the state obtained at the root of the search tree holds. Indeed, the removal of v from the domain of x at the root of the search tree is due to the global inconsistency of $\mathcal{P}|_{x=v}$. So, the $neighbourhood$ substitutable values of (x, v) are locally inconsistent, with respect to a given partial instantiation. On the other hand, the inconsistency resulting from a partial instantiation does not allow to infer, in the general case, the global inconsistency. Consequently, this does not allow to deduce, again in the general case, the values removal at the root of the search tree.

In addition, this result can also be applied if we consider networks $\phi(\mathcal{P}|_{\mathcal{I}_X, x=v})$ and $\phi(\mathcal{P}|_{\mathcal{I}_Y, x=v'})$ produced

by two instantiations $\mathcal{I}_X$ and $\mathcal{I}_Y$ with $Y \subseteq X$. Indeed, if we set $\mathcal{P}' = \phi(\mathcal{P}|_{\mathcal{I}_Y})$, then proposition 3 for the networks $\phi(\mathcal{P}'|_{\mathcal{I}_{X\setminus Y}, x=v})$ and $\phi(\mathcal{P}'|_{x=v'})$ holds. Similarly, this assertion is true only if $Y \subseteq X$. The comparison between two states for the same variable is thereby reduced, in our framework, to the comparison between the current state obtained from the current partial instantiation and the past states.

This comparative criterion has an important purpose in the dual-branching case, since it allows to find substitutable values, even if the assignments are made at different levels in the search tree. In the dway-branching case, the detected substitutable values of a given variable are obtained from the constraints subnetwork associated to the same partial instantiation.

In our framework, the worst-case complexity of algorithms $\mathsf{MAC_{NS}}$ in dual and dway-branching schemes are equivalent. First of all, to remove d values from the domain of a given variable, we need to store the states corresponding to each of its values, i.e d states. Since each state has $e(r-1)d^{r-1}$ values, the worst-case space complexity is $O(ernd^r)$, to compare each state during the whole search process.

To study the worst-case time complexity, we have to define what the "worst-case" is, because of the exponential size of the search tree. Our choice is to compute the worst-case time complexity for substitutable values detection before a variable refutation in the search tree. This case occurs when d values have been removed and each i^{th} assigned value implies to check the inclusion in the $i-1$ past states. So, there is $\binom{n}{2}$ comparisons between states in the worst-case. Since each comparison needs $e(r-1)d^{r-1}$ values comparison, the worst-case time complexity is $O(erd^{r+1})$. In case of binary constraints, the worst-case space (respectively time) complexity is in $O(end^2)$ (respectively $O(ed^3)$) to find all substitutable values of a variable before its refutation.

4. Experiments

Before presenting the evaluation of our proposed static and dynamic exploitation of the generalized neighbourhood substitutability, let us describe the used experimental protocol. All the tests were made on a Xeon 3.2GHz (2 GB RAM) cluster.

The first experiment presents a comparison of the static implementation (preprocessing) of SNS (respectively SAC), using a MAC algorithm with the classical dual-branching scheme, called $\mathsf{SNS_{MAC}}$ (respectively $\mathsf{SAC_{MAC}}$). The second experiment gives the evaluation of our dynamic detection and exploitation of the neighbourhood substitutability. This evaluation compares the MAC algorithm using dual-branching scheme with and without dynamic use of SNS, called MAC and $\mathsf{MAC_{NS}}$ respectively.

As the proposed approaches present high worst-case complexity for n-ary constraints, we only present the experimentation on binary CSPs. The n-ary case is currently under investigation. The CSP instances are taken from the 3^{rd} international CSP competition (*http://cpai.ucc.ie/08/*). The time limit is set to 1200 seconds whereas the memory limit is set to 900Mb.

4.1. Static approach

Satisfiable instances - Constraints in extension

Solver	#Solved	Time (s)	#Filtered	#Eliminated values
$\mathsf{SAC_{MAC}}$	296	20624	57	4373
$\mathsf{SNS_{MAC}}$	296	21019	72	41909

Unsatisfiable instances - Constraints in extension

Solver	#Solved	Time (s)	#Filtered	#Eliminated values
$\mathsf{SAC_{MAC}}$	195	13824	68	6869
$\mathsf{SNS_{MAC}}$	195	13807	78	8372

Satisfiable instances - Constraints in intension

Solver	#Solved	Time (s)	#Filtered	#Eliminated values
$\mathsf{SAC_{MAC}}$	253	14867	57	16404
$\mathsf{SNS_{MAC}}$	254	18054	110	82432
$-\mathsf{SNS_{MAC}}$	253	17855	109	79618

Unsatisfiable instances - Constraints in intension

Solver	#Solved	Time (s)	#Filtered	#Eliminated values
$\mathsf{SAC_{MAC}}$	202	8307	47	24382
$\mathsf{SNS_{MAC}}$	205	10857	51	41546
$-\mathsf{SNS_{MAC}}$	202	8017	49	37690

Table 1. Global results : $\mathsf{SAC_{MAC}}$ vs $\mathsf{SNS_{MAC}}$

First of all, Table 1 presents a global view on the results obtained by our SNS preprocessing on all the binary CSP instances. Each comparative table gives the results obtained on each category of instances (Satisfiable/Unsatisfiable, Constraints in extension/intension), in terms of the number of solved instances (#Solved), of the total solving time in seconds (Time), of the number of instances on which at least one value has been removed (#Filtered) and of the number of eliminated values (#Eliminated values).

Also, to get a finer comparison, we also give the results of each solver on the same set of solved instances. Such results are illustrated using the name of the solver preceded by the minus symbol. In table 1, we only show $-\mathsf{SNS_{MAC}}$ because the set of common solved instances corresponds to those solved by $\mathsf{SAC_{MAC}}$.

We can first observe that $\mathsf{SNS_{MAC}}$ solves globally 4 more instances than $\mathsf{SAC_{MAC}}$. It can be noted that in terms of CPU times, $\mathsf{SNS_{MAC}}$ is better (respectively worse) than $\mathsf{SAC_{MAC}}$ on the two categories of unsatisfiable (respectively satisfiable) CSP instances. The results also show that $\mathsf{SNS_{MAC}}$ is stronger than $\mathsf{SAC_{MAC}}$ in term of removed values. Indeed, SAC (respectively SNS) remove 59771 (respectively 182041) values in the preprocessing phase.

The scatter plot given in figure 1 illustrates the comparative results of $\mathsf{SNS_{MAC}}$ and $\mathsf{SAC_{MAC}}$ on all the binary CSPs

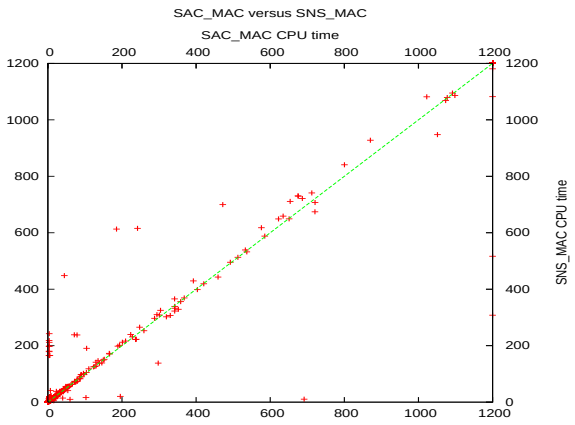

Figure 1. $\mathsf{SNS}_{\mathsf{MAC}}$ **and** $\mathsf{SAC}_{\mathsf{MAC}}$**: a comparison in term of CPU times**

instances. The x-axis (resp. y-axis) corresponds to $\mathsf{SAC}_{\mathsf{MAC}}$ (resp. $\mathsf{SNS}_{\mathsf{MAC}}$). In this plot, the CPU times tx and ty obtained by the solvers on a particular instance represent a (tx, ty) dot. The 4 instances solved only by $\mathsf{SNS}_{\mathsf{MAC}}$ correspond to the points at $tx = 1200\ seconds$. On some classes of instances our approach is clearly better. In figure 1, we can distinguish the bad behaviour of our approach on instances represented in the figure by the points around $(0, 200)$, and a good behaviour shown by the points below $ty = tx$. A more detailed analysis shows that $\mathsf{SNS}_{\mathsf{MAC}}$ is very useful on two classes of instances : jobShop and Taillard OpenShop problems.

Results on jobShop instances

Solver	#Solved	#Trivial	#Filt.	#Nodes	Ptime (s)	Time (s)
$\mathsf{SAC}_{\mathsf{MAC}}$	45	29	0	20789	93	175
$\mathsf{SNS}_{\mathsf{MAC}}$	45	35	19293	8801	2208	2276

Results for Taillard-OS problems

Solver	#Solved	#Trivial	#Filt.	#Nodes	Ptime (s)	Time (s)
$\mathsf{SAC}_{\mathsf{MAC}}$	25	3	10850	870351	590	1161
$\mathsf{SNS}_{\mathsf{MAC}}$	25	4	67105	151663	1775	1950

Table 2. Results for jobShop and Taillard-OS problems

Table 2 presents a detailed comparison between $\mathsf{SAC}_{\mathsf{MAC}}$ and $\mathsf{SNS}_{\mathsf{MAC}}$ on jobShop and Taillard-OS CSP instances. In addition to the measures given in table 1, we also give the number of satisfiable instances proven without backtracking or unsatisfiable instances without search process (#Trivial), the number of filtered values (#Filt.), the number of nodes in the search tree (#Nodes), the preprocessing and the entire solving time in seconds, called Ptime and Time respectively. Clearly our approach detects more neighbourhood substitutable values than SAC.

First of all, the jobShop problems are satisfiable, and contain 50 variables with domain size ranging from 100 to 250, and 265 binary constraints. The major part of such problems are easily solved by the $\mathsf{SAC}_{\mathsf{MAC}}$ approach without eliminating values at the preprocessing phase. For 46 instances solved by the two solvers, 29 are trivially satisfied, i.e backtrack free. Some problems are easily satisfied and the number of nodes is strongly reduced. However, the preprocessing time takes an important part of the solving time. Note that the number of instances shown in the table is 45 whereas 46 were mentioned previously. Indeed, on this particular instance $\mathsf{SNS}_{\mathsf{MAC}}$ eliminates 517 values and solves it in 192 nodes and $10s$ ($8s$ for Ptime), whereas $\mathsf{SAC}_{\mathsf{MAC}}$ do not eliminate any value, and answers its satisfiability in 4 millions of nodes in $691s$ ($2s$ for Ptime).

To complete our analysis for the static framework, the second class of problems for which $\mathsf{SNS}_{\mathsf{MAC}}$ is efficient is the Taillard OpenShop instances (in particular *os-taillard-n*). These instances have n^2 variables, with maximal domain 300 in average, and $4n^2$ constraints for $n = 4, 5, 7, 10$ in our case. 43 instances are solved by the two solvers. On 18 instances the two solvers do not achieve any domain filtering. On these last instances the preprocessing time is about $305s$ and $303s$, respectively for $\mathsf{SNS}_{\mathsf{MAC}}$ and $\mathsf{SAC}_{\mathsf{MAC}}$.

Our preprocessing is able to eliminates both singleton arc inconsistent values and generalized neighbourhood substitutable values. In spite of the fact that our preprocessing is less efficient when the number of values filtered is high, it clearly simplify the instance and reduce the time of the solving phase, which allows $\mathsf{SNS}_{\mathsf{MAC}}$ to solve more instances than $\mathsf{SAC}_{\mathsf{MAC}}$.

4.2. Dynamic approach

Satisfiable instances - Constraints in extension

Solver	# Solved	Time	#Filt.	#Eliminated values
MAC	296	20845	-	-
$\mathsf{MAC}_{\mathsf{NS}}$	294	17972	28	129
MAC	294	18600	-	-

Unsatisfiable instances - Constraints in extension

Solver	#Solved	Time	#Filtered	#Eliminated values
MAC	196	13771	-	-
$\mathsf{MAC}_{\mathsf{NS}}$	196	13716	36	66302

Satisfiable instances - Constraints in intension

Solver	#Solved	Time	# Filtered	#Eliminated values
MAC	253	9391	-	-
$\mathsf{MAC}_{\mathsf{NS}}$	254	9465	58	22439
$-$MAC	251	9466	-	-
$-\mathsf{MAC}_{\mathsf{NS}}$	251	8706	55	18441

Unsatisfiable instances - Constraints in intension

Solver	#Solved	Time	#Filtered	#Eliminated values
MAC	202	8268	-	-
$\mathsf{MAC}_{\mathsf{NS}}$	203	8435	29	152014
$-\mathsf{MAC}_{\mathsf{NS}}$	202	7789	28	104917

Table 3. Global results : MAC **vs** $\mathsf{MAC}_{\mathsf{NS}}$

Table 3 shows that MAC and $\mathsf{MAC}_{\mathsf{NS}}$ solve the same number of instances. The scatter plot (figure 2) clearly

shows that our MAC_{NS} dynamic approach is always faster i.e. the majority of points are below the diagonal.

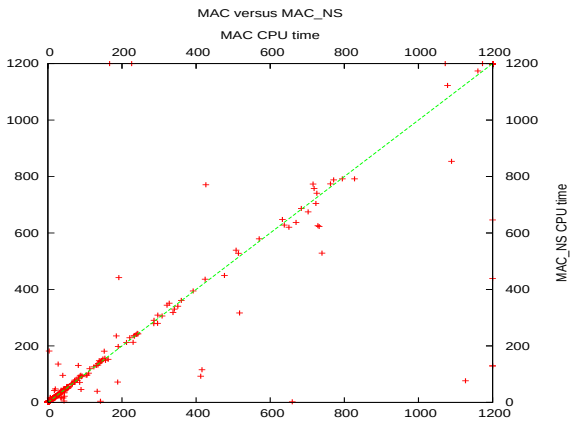

Figure 2. Comparison of the CPU times for MAC and MAC_{NS}

The performance of our dynamic approach on jobshops instances presents some similarity with our previous analysis. With 46 instances solved by the two solvers, 29 are trivially satisfiable. On the 17 other instances, only 10 involve the NS detection, and 9 of them are easy to solve, with time of $15s$ and $17s$ respectively for MAC_{NS} and MAC. Like in static analysis, the hardest instance for which MAC solver finds a solution in $660s$ while MAC_{NS} finds one in 1 second. This clearly shows that on hard instances, we also obtain important gain in term of CPU time. Also, for Taillard-OS instances, 2 instances in "intension/satisfiable" category are only solved by MAC_{NS} solver. On 45 common solved instances, 25 are filtered by the substitutability detection and 23 are solved by the two solvers, MAC in $1724s$ and MAC_{NS} in $1443s$, which eliminates 2611, reducing the number of nodes in the search tree (1046687 for MAC and 460024 for MAC_{NS}). Note that the other 2 instances are only solved by MAC_{NS} finding 397 substitutable values and solved the two instances in $556s$. Again, the neighbourhood dynamic substitutability detection for Taillard instances is better than the classical approach. As a summary, the dynamic approach MAC_{NS} shows better results in CPU time than the basic MAC solver.

5. Conclusion et future works

In this paper, a semantic generalization of the neighbourhood substitutability, a weaker form of symmetry defined by Freuder [6], is presented. This original generalization is obtained by a substitution of the syntactical concept of supports with a semantic measure of constraints propagation, in particular using the singleton arc-consistency filtering. A preprocessing embeded into the SAC algorithm was defined as a the static approach. Interestingly, since assignments of variables to values and maintaining arc-consistency are the basics steps for MAC-based search algorithms, a dynamic implementation was also proposed. Experimental results on binary CSPs show the usefulness of our static and dynamic approaches.

Even if the static approach does not show an overall efficiency w.r.t. time performance, however on hard instances, our approach achieves better results and solves more instances. Also, our approaches outperform the classical ones with respect to the number of filtered values and to the number of nodes in the search tree. The dynamic approach shows interesting results in time, even if the number of problems on which substitutable values have been found is low. Finally, on some classes of problems, such as jobShop and Taillard OpenShop the NS value detection is clearly relevant. On these classes, our approaches allow to solve instances out of reach of the classical one.

There are many future directions for this work. Beyond the study of the general case implementation (n-ary CSPs) and improvements of the implemented approaches, we plan to investigate how other constraints propagation algorithms can be exploited to deal with these forms of symmetry.

References

[1] Amit Bellicha, Christian Capelle, Michel Habib, Tibor Kökény, and Marie catherine Vilarem. Csp techniques using partial orders on domain values. In *Proceedings of ECAI'94 Wks. on CSP Issues Raised by Practical Applications*, 1994.

[2] Belaïd Benhamou and Lakhdar Saïs. Tractability through symmetries in propositional calculus. *Journal Automated Reasoning*, 12(1):89–102, 1994.

[3] Christian Bessière and Romuald Debruyne. Theoretical analysis of singleton arc consistency and its extensions. *Artificial Intelligence*, 172(1):29–41, 2008.

[4] Assef Chmeiss and Lakhdar Saïs. About neighborhood substitutability in csps. In *Proceedings of SymCon'2003*, pages 41–45, 2003.

[5] James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Proceedings of KR'96*, pages 148–159, 1996.

[6] Eugene C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of AAAI'91*, pages 227–233, 1991.

[7] Ian P. Gent and Barbara Smith. Symmetry breaking during search in constraint programming. In *Proceedings of ECAI'2000*, pages 599–603, 2000.

[8] Joey Hwang and David G. Mitchell. 2-way vs d-way branching for csp. In *In Proceedings of CP'05*, pages 343–357, 2005.

[9] Jean-François Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *Proceedings of ISMIS'93*, pages 350–361, 1993.