Overlapping Alldifferent Constraints and the Sudoku puzzle

F. Lardeux[3], E. Monfroy[1,2], F. Saubion[3], B. Crawford[1,4], and C. Castro[1]

[1] Universidad Técnica Federico Santa María, Valparaíso, Chile
[2] LINA, Université de Nantes, France
[3] LERIA, Université d'Angers, France
[4] Pontificia Universidad Católica de Valparaíso, PUCV, Chile

Abstract. Combinatorial problems can be modeled as constraint satisfaction problems (CSP). Modeling and resolution of CSP is often strengthened by *global constraints* (e.g., Alldiff constraint). We propose a formal system and some propagation rules for reducing domains of variables appearing in several Alldiff global constraints. We illustrate our approach on the well-known Sudoku puzzle which presents 27 overlapping Alldiff constraints in its 9×9 standard size. We also present some preliminary results we obtained in CHR and GeCode.

1 Introduction

Constraint satisfaction problems (CSP) cover a wide range of practical applications (planning, timetabling, scheduling, ...) in various domains (transportation, telecommunication, manufacturing, ...). Problems are classically expressed by a set of decision variables whose values belong to finite integer domains. The constraints model the relationships between these variables. Many resolution methods have been developed to produce sophisticated and efficient constraint solving systems. To improve the resolution efficiency and the expressive power provided by the constraint languages, some studies characterized specific constraint relations. These so-called global constraints ([10]) are frequently used in different problems and specific resolution techniques could be developed.
The first global constraint was certainly the alldifferent constraint (Alldiff in short) [6] which expresses that a set of variables have all different values. This constraint is very useful since it naturally appears in many classical combinatorial problems. However, it is well known that usual resolution techniques based on constraint propagation and local consistencies (such as arc-consistency [4]) are inefficient for this kind of constraint, i.e., reduction is rather limited when decomposing the Alldiff into $n*(n-1)/2$ disequalities. Therefore, specific algorithms have been proposed to handle this particular feature in CSP [12].

Alldifferent constraint can often be interleaved, e.g., 2 alldifferent constraints on two sets of variables $\mathcal{V}$ and $\mathcal{W}$ such that $\mathcal{V} \cap \mathcal{W} \neq \emptyset$. In this paper, we propose to handle the peculiarities of overlapping Alldiff constraints. Our goal is not to design a new algorithm (such as in [5]), nor to impel the user to

XXXIV Conferencia Latinoamericana de Informática (CLEI 2008), pp. 280-289, 2008.

change his model using new constraints (such as in [7]) but to strengthen the constraint propagation of a standard solver with some new reduction rules. To this end, we present a uniform propagation framework to handle constraints and more especially overlapping Alldiff constraints. From an operational point of view, we define new propagation rules to improve reduction capacities by taking into account specific properties induced by these overlapping Alldiff. We illustrate our approach with the Sudoku puzzle which presents 27 overlapping Alldiff constraints for its 9×9 standard size ($3.m$ for a m^2 grid). We present some preliminary results using CHR [2] and adding new propagators in the GeCode constraint programming library [9].

2 Reduction rules for solving CSP

We refer the reader to [1, 2] for an introduction to CSP solving.
A CSP is a tuple (X, D, C) where $X = \{x_1, \cdots, x_n\}$ is a set of variables taking their values in their respective domains $D = \{D_1, \cdots, D_n\}$. A constraint $c \in C$ is a relation $c \subseteq D_1 \times \cdots \times D_n$. In order to simplify notations, D will also denote the Cartesian product of D_i and C the union of its constraints. A tuple $d \in D$ is a solution of a CSP (X, D, C) if and only if $\forall c \in C, d \in c$.

Usual resolution interleaves reduction and search/enumeration: such solvers have been integrated in constraint programming languages (GeCode, Eclipse, ...). A search strategy consists in enumerating the possible values of a given variable in order to progressively built an assignment and reach a solution. To reduce the search tree, reduction techniques are added at each node. Local consistency mechanisms such as GAC [4] allow the algorithms to prune the search space by deleting inconsistent values from variables domains.

Definition 1 (Generalized Arc Consistency (GAC)). *A constraint c over variables $(x_1, \cdots, x_n)$ is generalized arc-consistent (GAC in short) iff $\forall i \in 1..n$, $\forall d \in D_i$, $\exists (d_1, \cdots, d_{i-1}, d_{i+1}, \cdots d_n) \in D_1 \times \cdots \times D_{i-1}, D_{i+1} \times \cdots \times D_n$, s.t. $(d_1, \cdots, d_n) \in c$. A CSP is GAC if all its constraints are GAC.*

The reduction/enumeration approach requires an important computational effort and thus encounters difficulties with large scale problems. Performances can be significantly improved by adding specific techniques such as efficient propagation algorithms, global constraints, ... Here, we are concerned with global constraints, and more specifically, with reduction rules for overlapping Alldiff.

2.1 Modeling the problems: Sudoku as a CSP

The Sudoku (a more constrained Latin Square) is a puzzle (e.g., [8]) which can be easily encoded as a CSP: it is played on a 9×9 partially filled grid which must be completed using numbers from 1 to 9 such that the numbers in each row, column, and major 3×3 blocks are different. A $n \times n$ Sudoku puzzle (with $n = m^2$) can be modeled by $3.n$ Alldiff over n^2 variables with domain $[1..n]$:

- A set of n^2 variables $\mathcal{X} = \{x_{i,j} | i \in [1..n], j \in [1..n]\}$
- A domain function D such that $\forall x \in \mathcal{X}, D(x) = [1..n]$
- A set of $3.n$ variables subsets $\mathcal{B} = \{C_1...C_n, L_1...L_n, B_{1,1}, B_{1,2}, \ldots, B_{m,m}\}$ defined by $\forall x_{ij} \in \mathcal{X}, \quad x_{ij} \in C_j, \quad x_{ij} \in L_i, \quad x_{ij} \in B_{((i-1)\div m)+1,((j-1)\div m)+1}$
- A set of $3.n$ alldifferent constraints: $\forall i \in [1..n]$ $Alldiff(C_i)$ and $\forall j \in [1..n]$ $Alldiff(L_j)$ and $\forall k, k' \in [1..m]$ $Alldiff(B_{kk'})$

2.2 The framework: rule based resolution

We propose a formal system (inspired by [1]) to define rules to reduce domains w.r.t. constraints. Reductions are abstracted as transitions over states. Given a CSP (X, D, C), a resolution state is a couple $< C, D >$ where C is a set of constraints, and D is a set of domains[5]. We define a domain reduction rule as:

$$\frac{< C, D > | \Sigma}{< C', D' > | \Sigma'}$$

where $D' \subseteq D$ and Σ and Σ' are first order formulas such that $\Sigma \wedge \Sigma'$ is consistent. Given a state $< C^k, D^k >$, a transition can be performed to a state $< C^{k+1}, D^{k+1} >$ if there is an instance of a rule

$$\frac{< C^k, D^k > | \Sigma^k}{< C^{k+1}, D^{k+1} > | \Sigma^{k+1}}$$

such that $D^k \models \bigwedge_{x \in X} x \in D^k_x \wedge \Sigma^k$, and D^{k+1} is the greatest subset of D^k such that $D^{k+1} \models \bigwedge_{x \in X} x \in D^{k+1}_x \wedge \Sigma^{k+1}$.
Here, we will only use variables, sets, domains, and values, and the operators $\subseteq$ and $=$ in conditions of rules (Σ and Σ'). Given a set of variables V we denote D_V the union of the domains of the variables in V: $D_V = \bigcup_{x \in V} D_x$. Given 2 sets V_i and V_j; V_{ij} (respectively $V_{i,j}$) denotes $V_i \cup V_j$ (respectively $V_i \cap V_j$).
Operationally, $d \notin D_x$ in a conclusion Σ' means that the value d can be removed from the domain of the variable x without loss of solution. $d \notin D_V$ means that d can be removed from each of the domain of the variables of the set V. $d_1, d_2 \notin D_x$ (resp. D_V) is a shortcut for $d_1 \notin D_x \ \wedge \ d_2 \notin D_x$ (resp. $d_1 \notin D_V \ \wedge \ d_2 \notin D_V$).
$< C, D > \rightarrow_R < C, D' >$ denotes the transition relation using the rule R, and $\rightarrow_R *$ is the reflexive transitive closure of $\rightarrow_R$. $\rightarrow_R *$ terminates due to the decreasing criterion on domains in the definition of the rules ([1]). This transition is extended to sets of rules $\mathcal{R}$. Note also that the result of $\rightarrow_{\mathcal{R}} *$ is independent from the order of application of the rules [1]: this is very important in practice for efficiency reasons (first scheduling "simple" rules before "complex" ones).

2.3 Two first rules: enforcing GAC of an Alldiff constraint

In the following, we classically note $Alldiff(V)$ the Alldiff constraint over a set of variables V. This semantically corresponds to the conjunction of $n*(n-1)/2$ pairwise disequality constraints $\bigwedge_{x_i, x_j \in V, i \neq j} x_i \neq x_j$.

[5] We canonically generalize $\subseteq$ to sets of domains as $D' \subseteq D$ iff $\forall x \in X \ D'_x \subseteq D_x$.

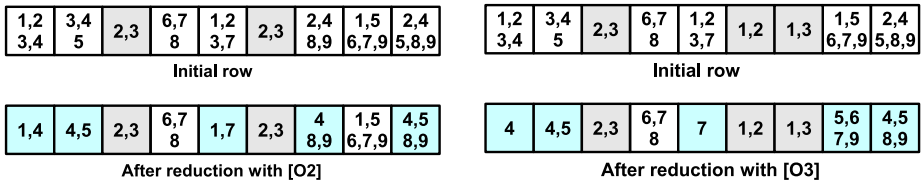

Fig. 1. Application of $[O2]$ (left) and $[O3]$ (right) on a Sudoku or Latin Square row

We first reformulate a well known consistency property [6, 10]. Consider a CSP $(X, C \wedge Alldiff(V), D)$. $[O1]$ says: if a variable x has been assigned a value d, then d must be discarded from the domains of the other variables of the Alldiff:

$$[O1] \quad \frac{< C \wedge Alldiff(V), D > \ \mid \ x \in V \ \wedge \ D_x = \{d\}}{< C \wedge Alldiff(V), D' > \ \mid \ d \notin D'_{V \setminus \{x\}}}$$

Property 1. Given $< Alldiff(V), D > \rightarrow^*_{[O1]} < Alldiff(V), D' >$, then the constraint $\bigwedge_{x_i, x_j \in V} x_i \neq x_j$ (i.e., the transformation of the Alldiff constraint into $n * (n-1)/2$ pairwise disequalities) is GAC w.r.t. D' . Note that enforcing arc consistency of the transformed constraint reduces less the domains than enforcing arc consistency of the initial Alldiff constraint.

Consider m such that $1 \leq m \leq (\#V - 1)$. Then, $[O1]$ can be generalized when considering a subset V' of m variables with m possible values.

$$[Om] \quad \frac{< C \wedge Alldiff(V), D > \ \mid \ V' \subset V \ \wedge \ D_{V'} = \{d_1, \ldots, d_m\}}{< C \wedge Alldiff(V), D' > \ \mid \ d_1, \ldots, d_m \notin D'_{V \setminus V'}}$$

Consider $m = 2$, and that two variables of an Alldiff only have the same two possible values. Then it is trivial to see that these two values cannot belong to the domains of the other variables (see Figure 1).

Property 2. Given $< Alldiff(V), D > \rightarrow^*_{[Om]_{1 \leq m \leq (\#V-1)}} < Alldiff(V), D' >$, then $< Alldiff(V), D' >$ has the GAC property.

The proof can be obtained from [6].
Rows, columns, and blocks are patterns for applying $[O2]$ and $[Om]$ on a Sudoku puzzle. Figure 1 is an illustration of $[O2]$ on a row: 2 variables contain only 2 values (2 and 3); $[O2]$ removes 2 and 3 from the over variables. Figure 1 illustrates of $[O3]$: this time, not all 3 variables contain the 3 values. For Sudoku, these rules are referred as naked pair ($[O2]$), naked triple ($[O3]$), ...

3 Rules for multiple Alldiff Constraints

For several overlapping Alldiff constraints, specific local consistency properties can be enforced according to the number of common variables, their possible values, and the number of overlaps. To simplify, we consider Alldiff constraints $Alldiff(V)$ such that $\#V = \#D_V$, i.e., as many values in the union of domains D_V as variables in the set V of variables. This restriction could be weaken but it is generally needed in classical problems (e.g., Sudoku or Latin squares). We now study typical patterns of Alldiff constraints.

3.1 Overlapping Alldiff constraints with one intersection

We start with rules for several Alldiff constraints joined by one intersection: conditions to apply the rules are thus on the values of this intersection.
Consider 2 Alldiff such that a value d appears in some variable(s) of the intersection of the two Alldiff, and that it does not appear in the rest of one of the Alldiff. Then d can be safely removed from the variables (except the ones of the intersection) domains of the second Alldiff (see Figure 2 for an illustration).

$$[OI2] \quad \frac{< C \wedge Alldiff(V_1) \ \wedge \ Alldiff(V_2), D > \ \mid \ d \in D_{V_1 \cap V_2} \ \wedge \ d \notin D_{V_2 \setminus V_1}}{< C \wedge Alldiff(V_1) \ \wedge \ Alldiff(V_2), D' > \ \mid \ d \notin D'_{V_1 \setminus V_2}}$$

In terms of Sudoku, $[OI2]$ is called the technique of locked candidates. Obviously, $[OI2]$ can be generalized to handle m ($m \geq 2$) $Alldiff$ constraints connected by one intersection. Let V be the set of variables of the intersection: $V = \bigcap_{i=1}^{m} V_i$:

$$[OIm] \quad \frac{< C \bigwedge_{i=1}^{m} Alldiff(V_i), D > \ \mid \ d \in D_V \ \wedge \ d \notin D_{V_1 \setminus V}}{< C \bigwedge_{i=1}^{m} Alldiff(V_i), D' > \ \mid \ d \notin \bigcup_{i=1}^{m} D'_{V_i \setminus V}}$$

$[OIm]$ means that if a value d appears in the intersection of m Alldiff and not in the rest of one of the Alldiff, then d can be safely removed from the variables (except the ones of the intersection) of all the other Alldiff. This rule can be implicitly applied to the different symmetrical possible orderings of the m Alldiff. Although one could argue that $[OIm]$ is useless (Prop. 3) for reduction, in practice $[OIm]$ is interesting in terms of the number of rules to be applied. Moreover, $[OIm]$ can be scheduled before $[OI2]$ to reduce the CSP at low cost.

Property 3. Consider $m > 2$ Alldiff with a non empty intersection. Given $< C, D > \rightarrow^*_{[OIm]} < C, D' >$ and $< C, D > \rightarrow^*_{[OI2]} < C, D'' >$, then $D'' \subseteq D'$.

The proof is straightforward. We illustrate it on Figure 2.

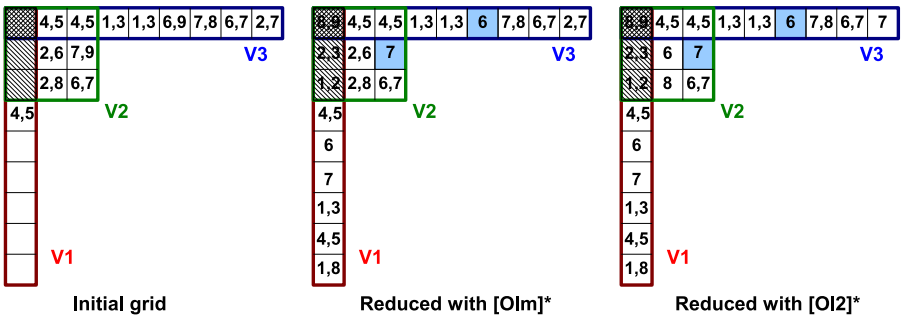

Fig. 2. $[OI2]^*$ reduces more than $[OIm]$

3.2 Overlapping Alldiff constraints with several intersections

We now consider Alldiff constraints with several (non-empty) intersections.

We first consider 4 $Alldiff$ having four **non-empty** intersections two by two. V now denotes the union of the four intersections: $V = V_{1,2} \cup V_{2,3} \cup V_{3,4} \cup V_{1,4}$.

$$[SI4.4] \quad \frac{< C \bigwedge_{i=1}^{4} Alldiff(V_i), D > | \; V_{1,2} \neq \emptyset \wedge V_{2,3} \neq \emptyset \wedge V_{3,4} \neq \emptyset \wedge V_{1,4} \neq \emptyset \;\wedge\; d \in D_V \;\wedge\; d \notin D_{V_{13} \setminus V_{24}}}{< C \bigwedge_{i=1}^{4} Alldiff(V_i), D' > | d \notin D'_{V_{24} \setminus V_{13}}}$$

Note that d must at least be an element of 2 opposite intersections (at least $d \in V_{1,2} \cap V_{3,4}$ or $d \in V_{2,3} \cap V_{1,4}$); otherwise, the problem has no solution. However, our rule is still valid in this case, and its reduction will help showing that there is no solution.
In terms of Sudoku, this rule can be applied on of Alldiff constraints that are either: 2 rows and 2 columns, 2 rows and 2 blocks, or 2 columns and to blocks. Figure 3 is an example of 2 rows and 2 columns.
$[SI4.4]$ can be extended to a **ring** of $2.m$ Alldiff with $2.m$ **non-empty** intersections. Let V be the union of the variables of the $2.m$ intersections: $V = \bigcup_{i=1}^{2.m} (V_i \cap V_{(i \bmod 2.m)+1})$. V_{odd} (respectively V_{even}) represents the union of the V_k such that k is odd (resp. even): $\bigcup_{i=0}^{m-1} V_{2.i+1}$ (resp. $\bigcup_{i=1}^{m} V_{2.i}$).

$$[SI2m.2m] \quad \frac{< C \bigwedge_{i=1}^{2.m} Alldiff(V_i), D > \;\; | \; \bigwedge_{i=1}^{2.m} (V_i \cap V_{(i \bmod 2.m)+1} \neq \emptyset) \;\wedge\; d \in D_V \;\wedge\; d \notin D_{V_{odd} \setminus V}}{< C \bigwedge_{i=1}^{2.m} Alldiff(V_i), D' > \;\; | \; d \notin D'_{V_{even} \setminus V}}$$

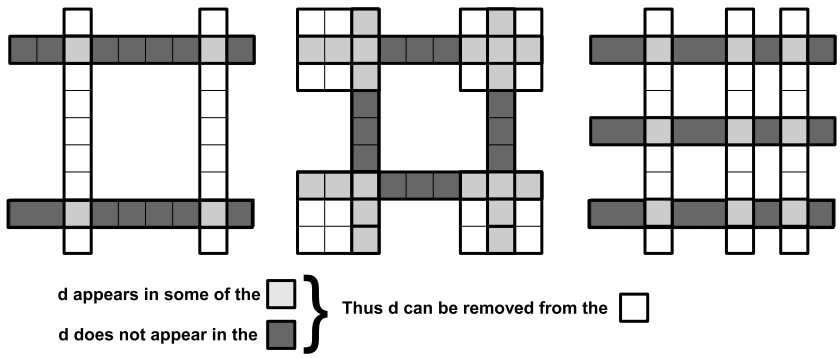

Fig. 3. Patterns for $[SI4.4]$ (left), $[SI2m.2m]$ with $m = 4$ (middle), and $[SI69]$ (right)

This rule can be applied for Sudoku puzzle: the pattern is made of 4 blocks, 2 rows and 2 columns, as shown in Figure 3.

$[SI2m.2m]$ extends $[SI4.4]$ in terms of ring, and now, $[SI2m.m^2]$ extends the matrix notion of $[SI4.4]$: a set of m Alldiff (e.g., rows) that cross another set of m Alldiff (e.g., columns), forming m^2 intersections. Consider

- V, the union of the variables of the m^2 intersections: $V = \bigcup_{i=1..m,\ j=m+1..2.m} V_{i,j}$

- V_r, the union of the first m Alldiff (e.g., rows) deprived of the intersections with the m other Alldiff (e.g., columns): $V_r = \bigcup_{i=1}^{m}[V_i \setminus (\cup_{j=m+1}^{2.m} V_{i,j})]$
- V_c is the union of the second set of m Alldiff (e.g., columns) deprived of the intersections with the m first Alldiff (e.g., rows): $V_c = \bigcup_{j=m+1}^{2.m}[V_j \setminus (\cup_{i=1}^{m} V_{j,i})]$

$$\text{Then:} \quad [SI2m.m^2] \quad \frac{< C \bigwedge_{i=1}^{2.m} Alldiff(V_i), D > \ \mid \ d \in D_V \ \wedge \ d \notin D_{V_r}}{< C \bigwedge_{i=1}^{2.m} Alldiff(V_i), D' > \ \mid \ d \notin D'_{V_c}}$$

Sketch of the proof. Consider m Alldiff (rows) and m other Alldiff (columns), and d occurres in the intersections rows/columns but not in rows. To cover the rows, one d from one intersection for each row will be used. Moreover, these d will not appear in the same column. Since we have m rows and m columns, the d of the intersection will be sufficient, and thus, d can be removed from the columns, except at the intersections with the rows.

Note that for $m = 3$, $[SI2m.m^2]$ represents the Swordfish technique for 9×9 Sudoku grids (Figure 3). Similarly to $[SI4.4]$ we can see some conditions that make that the problem has or not solution: e.g., in the case of $[SI6.9]$, d must be found in at least 2 intersections of each row and at least 2 intersections of each column; otherwise, if $d \notin D_{V_r}$, the problem has no solution.

3.3 Rules for application specific overlapping Alldiff constraints

Although they are not only problem specific, the following rules are well suited for the Sudoku puzzle. The aim of this section is to show how easy our rule based reduction framework can be used to model new rules.

Sudoku technique: XY-Wing This technique consider 4 Alldiff constraints (2 rows and 2 columns, 2 blocks and 2 rows, or 2 blocks and 2 columns) that form a "ring", i.e., the four 2 by 2 intersections are non empty. The XY-Wing technique (see Figure 4 for an example) can be formalized by the following rule:

$$[R.XY - W] \quad \frac{\begin{array}{c} < C \bigwedge_{i=1}^{4} Alldiff(V_i), D > | V_{1,2} \neq \emptyset \ \wedge \ V_{2,3} \neq \emptyset \ \wedge \ V_{3,4} \neq \emptyset \\ \wedge \ V_{4,1} \neq \emptyset \ \wedge \ v_0 \in V_{1,2} \ \wedge \ v_1 \in V_1 \ \wedge \ v_2 \in V_2 \ \wedge \\ \{d_1, d_2\} = D_{v_0} \ \wedge \ \{d_1, d_3\} = D_{v_1} \ \wedge \ \{d_2, d_3\} = D_{v_2} \end{array}}{< C \bigwedge_{i=1}^{4} Alldiff(V_i), D' > \ \mid \ d_3 \notin D'_{V_{1,2} \cup V_{3,4}}}$$

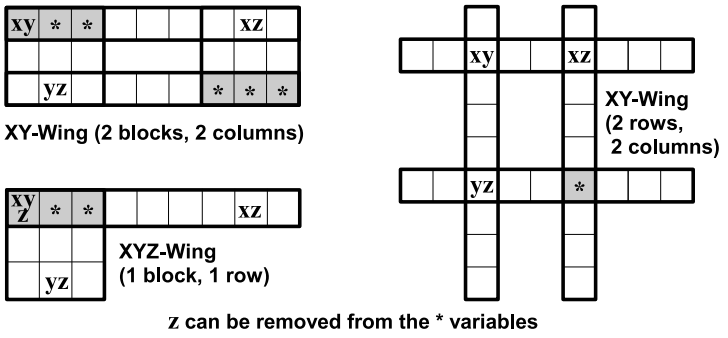

Fig. 4. $[R.XY - W]$ (XY-Wing technique) and $[R.XYZ - W]$ (XYZ-Wing technique)

For 2 blocks and 2 rows (or columns), in the best case this rule removes 5 values for a 9×9 Sudoku. For 2 columns and 2 blocks, this rule removes at most one value for a $n \times n$ Sudoku, or a $n \times n$ Latin Square.

Sudoku technique: XYZ-Wing this is a variant of the XY-Wing, based on the intersection of a block and a column/row: if 3 values x, y, z appear in a variable of the intersection, and that the column/row contains a variable with the 2 values x, z, and that the block contains a variable with the 2 values y, z, then, z can be removed from the other variables of the intersection (Figure 4).

$$[R.XYZ{-}W]\ \frac{< C \wedge Alldiff(V_1) \wedge Alldiff(V_2), D > \mid v_0 \in V_{1,2} \ \wedge\ v_1 \in V_1 \ \wedge\ v_2 \in V_2 \ \wedge\ \{d_1,d_2,d_3\} = D_{v_0} \ \wedge\ \{d_1,d_3\} = D_{v_1} \ \wedge\ \{d_2,d_3\} = D_{v_2}}{< C \wedge Alldiff(V_1) \wedge Alldiff(V_2), D' > \mid d_3 \notin D'_{V_{1,2}\setminus\{v_0\}}}$$

3.4 Reduction strength of the rules

The reduction we obtain by applying rules for a single Alldiff and rules for several Alldiff is stronger than enforcing GAC:

Property 4. Given a conjunction of constraints $C = \bigwedge_{i=1}^{k} Alldiff(V_i)$ and a set of domains D. Given 2 sets of rules $R' = \bigcup_{l=1}^{max_i\{\#V_i\}}\{[Ol]\}$ and $R \subseteq \{[OI2], \ldots, [OIm], [SI4.4], \ldots, [SI2m.2m], [SI3.9], \ldots, [SI2m.m^2], [R.XY{-}W], [R.XYZ{-}W], [SI2.1]\}$. Consider $< C, D > \rightarrow^*_{R'} < C, D' >$ and $< C, D > \rightarrow^*_{R' \cup R} < C, D'' >$ then $< C, D' >$ and $< C, D'' >$ are GAC and moreover $D'' \subseteq D'$.

The proof is based on the fact that $\bigcup_{l=1}^{max_i\{\#V_i\}}\{[Ol]\}$ already enforces GAC (see [6]) and that the rules of R preserves GAC (they do not loose any solution).

4 Experimental results

We discuss here some implementation possibilities with respect to 2 different systems, not some quantified results (tables).

Implementation in CHR [2] We have few rules to manage, but the combinatoric is pushed in rule applications, i.e., in the matching of the head and test of the guards. A generic (re-usable rules for any problems) and simple implementation consists in a direct translation of our rules into CHR propagation rules. Although straightforward and very easy, this implementation is rather inefficient, and more especially for complex rules: the matching is very weak and a huge number of conditions are pushed in the guard. Thus, the interpreter has to try the guards for all matching patterns. We thus specialized the CHR rules for arrays of variables, which is well suited for problems such as Latin Squares and Sudoku. The rules are also scheduled in order to apply first the rules that are faster to apply (strong matching condition and few conditions in the guard), and which have more chance to effectively reduce the CSP.

The results correspond to our expectations: for easy Sudoku, simple rules (i.e., $[O2]$, $[O3]$, $[O4]$) are sufficient to solve the puzzle without backtracking. More

difficult Sudoku requires more and more complex rules ([Om], and rules managing several Alldiff). In general, applying complex rules is longer than performing enumeration. However, we are still working to improve the matching: e.g., by particularizing [O7], we obtained a speed up of up to 1000 for some Sudokus. The main advantage of this implementation is to have an encoding which is as closed as possible from the formal rules. It is thus very easy to try a new rule, test its efficiency (in terms of the number of application of the rule, not in time), and to study whether the pattern of the rule appears often or not in problems.

New propagators in GeCode GeCode is a library for constraint programming that allows one to easily add new types of constraints, new search techniques, or new propagation/reduction algorithms called propagators. We implemented some rules as new propagators in GeCode [9]. We report here on $[R.XYZ - W]$ (XYZ-Wing of Sudoku). We made some tests using the standard Alldiff of GeCode and this new propagator. On average, we obtained 15% more propagations (i.e., application of propagators that effectively reduce the CSP) and 25% less failure cases (enumerations that lead to no solution). For easy Sudoku, the new propagator was not useful (no application effectively lead to a reduction), but the overhead was negligible. For more complex instances, the rule significantly reduced the number of backtracks (i.e., more reductions are done, and less failed nodes are explored due to enumeration). In terms of time, the results are heterogeneous: applying $[R.XYZ - W]$ significantly speeds up some problems, while it also significantly slows down others; up to now, we could not characterize for which Sudoku grids the propagators behave better or worse. The average time, however seems to be better on the collection of Sudoku we tried. We are thus confident in the use of such rules. Indeed, the propagator for $[R.XYZ - W]$ is considered at the same level as the others. We could thus improve reduction time by better scheduling propagators, such as applying $[R.XYZ - W]$ when all standard propagators (cheaper to apply) have reached a fixed point. The preliminary results are promising: we plan to improve propagation time with a better scheduling, and to implement more propagators.

As usual in CSP solving, appears the tradeoff between propagation and enumeration: the best pruning will not always lead to the best results in time. In our case, the most expensive and inefficient rules could be skipped at the cost of some more enumerations. For example, considering a 9x9 Sudoku in CHR, the matching for O1 and O2 (and also for O8 and O7) is much cheaper than for O3, O4, O5, and O6. Thus, O3...O6 could be skipped. However, this selection of rules is problem- and solver-dependent, and thus out of the scope of this paper.

5 Related Work and Conclusion

Global constraints in CSP Recent works deal with the combination of several global constraints (e.g., [11] for the sequence constraint) [5] describes the cardinality matrix constraint which imposes that the same value appears p times in the variables of each row (of size n), and q times in the variables of each column (of size m). Some Alldiff constraints on the rows and columns of a matrix is a

special case of this constraint with $p = q = 1$. However, this constraint forces each Alldiff to be of size n or m while with our rules, they can be of different sizes. Nevertheless, theses approaches require some specialized and complex algorithms for reducing the domains, while our approach allows us to simplify and unify the presentation of the propagation rules and attempts at addressing a wider range of possible combinations of Alldiff.
From the modeling point of view, [7] evaluates the difficulty of the Sudoku problem: various models using different types of constraints are proposed (e.g., the Row/Column interaction is described by the cardinality matrix global constraint; together with the row/block interaction this should compared to the application of our rule [OI2] on all intersections of a column and a row, and block and row (or column)). [3] also evaluates Sudoku grid difficulty by proposing several models together with the associated reduction rules. In our approach, we only use the classical model and do not change it: we only add more propagation rules. Moreover, our rules can be used with other problems.

Conclusion We have defined a framework and some reduction rules for overlapping Alldiff that can be easily implemented in usual constraint solvers. We have illustrated the use of these rules on the Sudoku puzzle. Preliminaries results show that such rules may improve domain reduction efficiency and could be even more efficient in Gecode by scheduling more finely the application of propagators. In the future, we plan to apply our technique to other overlapping global constraints, such as the atmost or cardinality global constraints.

References

1. K. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.
2. T. Fruewirth and S. Abdennadher. *Essentials of Constraint Programming.* 2003.
3. F. Laburthe, G. Rochart, and N. Jussien. Évaluer la difficulté d'une grille de sudoku à l'aide d'un modèle contraintes. In *JFPC'06*, pages 239–248, 2006.
4. A. Mackworth. *Encyclopedia on Artificial Intelligence*, chapter Constraint Satisfaction. John Wiley, 1987.
5. J.-C. Régin and C. Gomes. The cardinality matrix constraint. In *Proc. of CP 2004*, pages 572–587, 2004.
6. J.C. Régin. A filtering algorithm for constraint of difference in csps. In *National Conference of Artificial Intelligence*, pages 362–367, 1994.
7. H. Simonis. Sudoku as a constraint problem. In *Proc. of the 4th CP Int. Work. on Modelling and Reformulating Constraint Satisfaction Problems*, pages 17–27, 2005.
8. Sudopedia. http://www.sudopedia.org/wiki/Main_Page.
9. Gecode: generic constraint development environment. http://www.gecode.org/.
10. W.-J. van Hoeve and I. Katriel. *Handbook of Constraint Programming*, chapter Global Constraints. Elsevier, 2006.
11. W.-J. van Hoeve, G. Pesant, L.-M. Rousseau, and A. Sabharwal. New filtering algorithms for combinations of among constraints, 2008. Under review.
12. W.J. van Hoeve. The alldifferent constraint: A survey. In *Proc. of ERCIM Workshop*, 2001. http://arxiv.org/abs/cs.PL/0105015.