Preserving Partial Solutions while Relaxing Constraint Networks*

Éric Grégoire^1 and **Jean-Marie Lagniez**^{2} and **Bertrand Mazure**^{1}
^{1}CRIL, Université d'Artois & CNRS, Lens, France
^{2}FMV, Johannes Kepler University, Linz, Austria
{gregoire,mazure}@cril.fr Jean-Marie.Lagniez@jku.at

Abstract

This paper is about transforming constraint networks to accommodate additional constraints in specific ways. The focus is on two intertwined issues. First, we investigate how partial solutions to an initial network can be preserved from the potential impact of additional constraints. Second, we study how more permissive constraints, which are intended to enlarge the set of solutions, can be accommodated in a constraint network. These two problems are studied in the general case and the light is shed on their relationship. A case study is then investigated where a more permissive additional constraint is taken into account through a form of network relaxation, while some previous partial solutions are preserved at the same time.

1 Introduction

The study of CSPs (Constraint Satisfaction Problems) has long been a fertile research domain in Artificial Intelligence (see for example [Milano, 2012; Beldiceanu *et al.*, 2012; Pesant, 2012; Rossi *et al.*, 2006] as major current conferences, journals and handbooks). In this paper, we are concerned with CSPs about constraint networks that model problems through a set of constraints linking various variables where each variable is provided with a finite instantiation domain. A solution to a constraint network $\mathcal{P}$ is an assignment of values to all variables occurring in $\mathcal{P}$ such that all constraints of $\mathcal{P}$ are satisfied. A CSP is an NP-hard problem that consists in checking whether $\mathcal{P}$ exhibits solutions or not, and in computing one solution in the positive case.

In this paper, we are *not* directly concerned with the problems of finding out a solution to a constraint network or showing that no solution exists. Instead, we are concerned with how a network must be *transformed* in order to accommodate an additional constraint in such a way that some *specific objectives* can be met.

The usual, simplest and most direct way to accommodate an additional constraint c consists in inserting c inside the network, which is enlarged to include possible additional variables occurring in c, and then in checking previous solutions or re-starting the search for solution(s) while reusing previous computations as much as possible. Although such a direct insertion and treatment of c often matches the user's intended goal, there exist however circumstances where this approach is not the right one. In this paper, several such situations are investigated.

Preserving some partial solutions.
Assume that a manager needs to take decisions based on *all* solutions provided by a network $\mathcal{P}$ that models constraints guiding the allocation of resources among several tasks. She has to investigate how an additional task of less importance could also be performed under the constraint that this would not restrict the range of current solutions for the most important tasks. At the extreme, she might want to consider the case where all current solutions about all variables are to be preserved when the possible additional task is taken into account.

A solution to $\mathcal{P}$ being an assignment of values to all the variables occurring in $\mathcal{P}$ and satisfying all constraints of $\mathcal{P}$, she thus wants to preserve some partial solutions, namely the tuples of values assigned by global solutions to some given variables of $\mathcal{P}$. These variables express the allocation of resources for the most important tasks. Assume that c represents the constraints about the additional task. Clearly, inserting c inside $\mathcal{P}$ would not always provide the intended result as some previous solutions to $\mathcal{P}$ and partial solutions about the most important tasks might be lost in this way. Indeed, inserting an additional constraint c can decrease the set of solutions. Note that rank ordering constraints and assigning a lower priority to c would not solve the issue in the general case: previous solutions to $\mathcal{P}$ that conflict with c would still be dismissed. Introducing one or several additional constraints to represent and handle the partial solutions to be preserved must make sure that each one (vs. at least one) of these solutions which are mutually incompatible takes part in at least one solution of the resulting network that will include c. In decision-support systems, the structure and form of the CSP must often be modified as less as possible when an additional constraint is handled: for example, transforming the CSP into a set of tuples that solve the new problem would be a conceptually simple

*This work has been partially supported by FWF, NFN Grant S11408-N23 (RiSE), an EC FEDER grant and the *Conseil Régional du Nord/Pas-de-Calais.*

-but inappropriate- approach. Indeed, the specifications of the CSP might further evolve and the constraints must be available for possible additional subsequent evolutions of the system, which must be well-understood and explainable in terms of the constraints, which can be written in intentional form.

Accommodating more permissive constraints.
We also consider circumstances where an additional constraint is intended to be *more permissive* and allow for more solutions. Obviously, inserting an additional constraint inside a network $\mathcal{P}$ to form a new network $\mathcal{P}'$ cannot enlarge the set of possible solutions from $\mathcal{P}$ that will take part in the solutions to $\mathcal{P}'$. In the above resource allocation problem, this coincided with relaxing some resource availability. When it is easy to locate the constraints or domain variables that forbid the additional solutions, the problem is straightforward. However, it happens that this relaxation of the problem specifications is to be performed among dispersed constraints and variable domains in interaction that are not easy to pinpoint. Let us illustrate the issue through a simpler example from another CSP application domain. Consider a complex constraint network $\mathcal{P}$ in the legal domain that models, among other things, under which circumstances some specific parental benefits can be claimed. For this issue, it delivers as solutions all situations where the applicant is the parent of at least 3 children. This constraint is not necessarily explicit in $\mathcal{P}$ but can result from the interaction of several other ones. Now assume that the legislation changes and that a more permissive rule is adopted allowing these benefits to be claimed by parents of at least two children. If this new constraint is merely inserted as an additional one within $\mathcal{P}$ then it will be preempted by the constraints requiring at least three children. Being the parent of two children would still not allow the aforementioned benefits. Indeed, since solving $\mathcal{P}'$ will try to satisfy all constraints together, the additional solution translated by the new more permissive rule will be subsumed in the process by the more restrictive ones: having at least two children is satisfied whenever having at least three children is satisfied. More generally, whenever a new constraint c is intended to enlarge a set of solutions of a given constraint network, it cannot be merely inserted within the network. A form of *network relaxation* should take place instead. Locating constraints and domain variables to be relaxed is not always a straightforward issue: specific methods and tools are required to that end. Investigating this second issue is another goal of the paper.

Combining both issues.
In the resource allocation problem, we can also be faced with situations were both problems are intertwined and in mutual influence. Especially, we can be asked to enforce all additional solutions encoded by a new constraint that are more permissive than the current network. Locating *where* this relaxation can be done in the network is not always straightforward. At the same time, the process can be required to have a limited impact on the current model of the problem. Especially, this can occur when the model is a large-scale incomplete one and when we do not want to lose any of all its current possible solutions. More precisely, we might not want the new constraint to impact solutions outside some variables (in particular, variables not mentioned in the additional constraint). A similar issue can be raised in the legal domain example. As the CSP can only be a rough and incomplete approximation of the existing huge set of fiscal rules, the relaxation process (e.g., about the parental benefits) can then required to be local in the sense that it has to be circumscribed to a given set variables and have no impact on solution values for other variables about other fiscal issues implemented in the system.

In the paper, we investigate the three issues successively and propose several methods and algorithms. Experimental results for the last two issues are also provided. As CSP emerge as a technology in the real-word for large-scale applications, we believe that these problems that concern the transformation of constraint networks in order to take evolving specifications into account can only become of increasing interest.

2 Constraint Networks and MUCs

Definition 1 (Constraint network) *A constraint network $\mathcal{P}$ is a pair $\langle \mathcal{X}, \mathcal{C} \rangle$ where*
(1) $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a finite set of variables s.t. any variable x_i is given a finite domain written $dom(x_i)$
(2) $\mathcal{C} = \{c_1, \ldots, c_m\}$ is a finite set of constraints s.t. every $c_i \in \mathcal{C}$ is defined by a pair $(var(c_i), rel(c_i))$ where

- *$var(c_i)$ is the set of variables $\{x_{i_1}, \ldots, x_{i_{n_i}}\} \subseteq \mathcal{X}$ occurring in c_i,*
- *$rel(c_i)$ is a subset of the cartesian product $\prod_{j=1}^{n_i} dom(x_{i_j})$ of the domains of the variables of $var(c_i)$ representing the tuples of values allowed for c_i (i.e., satisfying c_i).*

Without loss of generality, we will often interpret sets of variables or constraints as tuples of variables or constraints. Note that $rel(c_i)$ is sometimes given in intentional form. Also, $for(c_i)$ represents the set of forbidden tuples by c_i, which is defined by $rel(c_i) \cup for(c_i) = \prod_{j=1}^{n_i} dom(x_{i_j})$.

Definition 2 (Instantiation of a constraint network) *Let $\mathcal{P} = \langle \mathcal{X}, \mathcal{C} \rangle$ be a constraint network. An instantiation $\mathcal{A}$ of $\mathcal{P}$ assigns to each variable x_i of $\mathcal{X}$ a value from $dom(x_i)$. A partial instantiation $\mathcal{A}$ of $\mathcal{P}$ w.r.t. a set $\mathcal{Y}$ of variables assigns to each variable y_i of $\mathcal{Y}$ a value from $dom(y_i)$.*

$\mathcal{Y} \subseteq \mathcal{X}$ is not required: we allow a partial instantiation to instantiate variables not occurring in the network. The tuple of values assigned by $\mathcal{A}$ to the variables of $\mathcal{Y}$ (resp. to the variables occurring in c) is noted $\mathcal{A}(\mathcal{Y})$ (resp. $\mathcal{A}(c)$).

Definition 3 (Solution to a constraint network) *A solution $\mathcal{A}$ to a constraint network $\mathcal{P} = \langle \mathcal{X}, \mathcal{C} \rangle$ is an instantiation $\mathcal{A}$ of $\mathcal{P}$ such that for all c in $\mathcal{C}$ we have $\mathcal{A}(c)$ belongs to $rel(c)$. We say that $\mathcal{A}$ satisfies $\mathcal{P}$ and that $\mathcal{A}$ is a model of $\mathcal{P}$. When $\mathcal{P}$ possesses at least one model, $\mathcal{P}$ is said satisfiable. Otherwise $\mathcal{P}$ is said unsatisfiable.*

Unless explicitly indicated, the constraint network $\mathcal{P}$ in which a new constraint c need be inserted is assumed satisfiable. However, so-called MUCs and MSSes concepts that are related to unsatisfiable networks are of importance here. Any unsatisfiable constraint network $\mathcal{P} = \langle \mathcal{X}, \mathcal{C} \rangle$ involves at least one MUC (*Minimal Unsatisfiable Core*), in short *core*. A MUC is a subset of constraints of $\mathcal{C}$ that, at the same time, is unsatisfiable and that is such that every one of its strict subsets is satisfiable. In a dual way, a *Maximal Satisfiable Sub-network* (MSS) of $\mathcal{P} = \langle \mathcal{X}, \mathcal{C} \rangle$ is defined as any satisfiable $\mathcal{P}' = (\mathcal{X}', \mathcal{C}')$ where $\mathcal{X}'$ (resp. $\mathcal{C}'$) is a subset of $\mathcal{X}$ (resp. $\mathcal{C}$) and such that for all c in $\mathcal{C} \setminus \mathcal{C}'$, $\langle \mathcal{X}' \cup var(c), \mathcal{C}' \cup \{c\} \rangle$ is unsatisfiable. Checking whether a constraint belongs to a MUC or not belongs to Σ_2^P [Eiter and Gottlob, 1992]. In the worst case, the number of MUCs can be exponential in the number m of constraints (it is in $\mathcal{O}(C_m^{m/2})$). Despite bad worst-case complexity results, in many real-life situations, the number of MUCs can remain of a manageable size and both MUCs and MSSes can be computed in realistic time (see for example results in [Grégoire *et al.*, 2007; 2008]). One specific well-studied problem about MSS is called WCSP (as Weighted CSP) which consists in delivering one MSS of a constraint network obeying a priority scale between constraints. Each constraint is given a numerical value and constraints with the higher values are preferred candidates for belonging to the resulting MSS. More on MUC and MSS can be found in e.g., [Bakker *et al.*, 1993; Han and Lee, 1999; Jussien and Barichard, 2000; Junker, 2001; Petit *et al.*, 2003; Junker, 2004; Hémery *et al.*, 2006; Grégoire *et al.*, 2008; Marques-Silva and Lynce, 2011; Belov and Marques-Silva, 2011; Janota and Marques-Silva, 2011; Belov and Marques-Silva, 2012].

3 Preserving Partial Solutions

Let us go back to the first issue raised in the introduction, where some given partial solutions to a network $\mathcal{P}$ are to be preserved while a new constraint c is to be added. First, let us define a concept of partial solution to a constraint network.

Definition 4 (Partial solution to a constraint network) *A partial solution $\mathcal{A}$ to a constraint network $\mathcal{P} = \langle \mathcal{X}, \mathcal{C} \rangle$ w.r.t. a set of variables $\mathcal{X}'$ is a partial instantiation of $\mathcal{P}$ w.r.t. $\mathcal{X}'$ such that $\mathcal{A}$ can be extended into a model of $\mathcal{P}$.*

Let $\mathcal{X}$ be a set of variables. Consider a set $\mathcal{S}_{(\mathcal{P},\mathcal{X})}$ of partial solutions $\mathcal{A}$ to $\mathcal{P}$ with respect to $\mathcal{X}$ and assume that these partial solutions must be preserved. In the example, the values assigned to variables of $\mathcal{X}$ express some possible allocations of resources to the most important tasks so that a global solution involving all tasks could be reached. $\mathcal{P}$ should be transformed into a network $\mathcal{P}'$ such that (1) any solution to $\mathcal{P}'$ satisfies c, (2) any solution to $\mathcal{P}'$ extends at least one solution to $\mathcal{P}$, and (3) any element of $\mathcal{S}_{(\mathcal{P},\mathcal{X})}$ can be extended into a solution to $\mathcal{P}'$.

Obviously, when c is incompatible with $\mathcal{P} = \langle \mathcal{X}, \mathcal{C} \rangle$ in the sense that $\langle \mathcal{X} \cup var(c), \mathcal{C} \cup \{c\} \rangle$ possesses no solution, the above transformation problem exhibits no solution: in the example, the new task cannot be accommodated together with the other ones. Note that even when $\langle \mathcal{X} \cup var(c), \mathcal{C} \cup \{c\} \rangle$ possesses solutions, it might not be possible to handle c in the above way: in the example, this coincides with situations where the new task can only be accommodated in a way that narrows the space of solutions for the most important tasks.

Algorithm 1: Insert-and-Preserve

```
input: $\mathcal{P} = \langle \mathcal{X}, \mathcal{C} \rangle$: a constraint network ;
       $c$: the additional constraint ;
       $\mathcal{S}_{(\mathcal{P},\mathcal{X})}$: a set of partial solutions of $\mathcal{P}$
       to preserve ;
output: $\langle \mathcal{X}', \mathcal{C}' \rangle$: a constraint network where $\mathcal{S}_{(\mathcal{P},\mathcal{X})}$ is
        preserved and $c$ inserted ;
1 $\mathcal{C}' \leftarrow \mathcal{C} \cup \{c\}$ ; $\mathcal{X}' \leftarrow \mathcal{X} \cup var(c)$ ;
2 foreach $\mathcal{A} \in \mathcal{S}_{(\mathcal{P},\mathcal{X})}$ do
3     Let $c_{\mathcal{A}}$ be the constraint s.t. $var(c_{\mathcal{A}}) = var(\mathcal{A})$ and
      $rel(c_{\mathcal{A}}) = \mathcal{A}$ ;
4     while $(\mathcal{X}', \mathcal{C}' \cup \{c_{\mathcal{A}}\})$ is unsat do
5         $\langle \mathcal{X}', \mathcal{C}'' \rangle \leftarrow$ MUC $(\mathcal{X}', \mathcal{C}' \cup \{c_{\mathcal{A}}\})$ ;
6         if $\exists c'' \in \mathcal{C}''$ s.t. $c'' \neq c_{\mathcal{A}}$ and $c'' \neq c$ then
7             $\mathcal{C}' \leftarrow \mathcal{C}' \setminus \{c''\}$ ;
8         else return $\langle \emptyset, \emptyset \rangle$ ;
9 return $\langle \mathcal{X}', \mathcal{C}' \rangle$;
```

To some extent, partial solutions appear as a form of integrity constraints that must be taken into account when a constraint network is enriched by some additional constraints. They express sets of values for some given variables leading to solutions that must be kept when the network evolves. In the general case, there is no direct way to implement them as a new constraint in the network so that the structure of the network is preserved as much as possible. For example, a constraint made of the disjunction of the partial solutions to preserve would not ensure that any of its disjuncts remains a partial solution.

Insert-and-Preserve is a direct algorithm to preserve a set $\mathcal{S}_{(\mathcal{P},\mathcal{X})}$ of partial solutions $\mathcal{A}$ while inserting c within $\mathcal{P}$. First, c is inserted within $\mathcal{P}$ to yield $\mathcal{P}' = \langle \mathcal{X}' = \mathcal{X} \cup var(c), \mathcal{C}' = \mathcal{C} \cup \{c\} \rangle$. If $\mathcal{P}'$ is unsatisfiable then no solution exists. Otherwise, each partial solution $\mathcal{A}$ from $\mathcal{S}_{(\mathcal{P},\mathcal{X})}$ is considered successively and rewritten as a constraint $c_{\mathcal{A}}$. While $\langle \mathcal{X}', C' \cup \{c_{\mathcal{A}}\} \rangle$ is unsatisfiable, one MUC is extracted and some of its constraints (different from both c and $c_{\mathcal{A}}$) is expelled from $\mathcal{P}'$. When a MUC just made of both c and $\mathcal{A}$ is found then no solution exists. In parallel to the possible high cost of computing enough MUCs to restore satisfiability, one major factor influencing the time-efficiency of the algorithm clearly depends on the number of partial solutions to be preserved. Note that a dual algorithm based on MSSes can be proposed, too.

4 Handling more Permissive Constraints

A formal definition of *more permissive* constraints is:

Definition 5 (More permissive constraint) *Let $\mathcal{P}$ be a constraint network. A constraint c is more permissive than $\mathcal{P}$ iff (1) there exists a partial instantiation of $\mathcal{P}$ that satisfies c and cannot be extended into a model of $\mathcal{P}$, and (2) every model of $\mathcal{P}$ satisfies c.*

In the parental benefits issue, a more permissive constraint need to *prevail* in the network. Formally:

Definition 6 (Prevailing constraint) *A constraint c prevails in a constraint network $\mathcal{P} = \langle \mathcal{X}, \mathcal{C} \rangle$ iff any partial instantiation of $\mathcal{P}$ with respect to $var(c)$ that satisfies c can be extended into a model of $\mathcal{P}$.*

Thus, relaxing $\mathcal{P}$ by a more permissive constraint c must lead to a network $\mathcal{P}'$ such that any solution to c has been extended into a solution to $\mathcal{P}'$.

Definition 7 (Relaxed constraint network) *A network $\mathcal{P}' = \langle \mathcal{X}', \mathcal{C}' \rangle$ is a relaxed network of $\mathcal{P} = \langle \mathcal{X}, \mathcal{C} \rangle$ by c iff any solution to c can be extended into a solution to $\mathcal{P}'$.*

Note that requiring all solutions of c to be solutions of $\mathcal{P}'$ can lead to the existence of additional solutions related to variables outside the scope of c through a domino effect about relaxation within $\mathcal{P}$. Hence, $\mathcal{P}'$ can exhibit new solutions about variables not occurring in c in addition to those provided by $\mathcal{P}$. For a similar reason, we cannot require in the general case that any solution of $\mathcal{P}$ is a solution of $\mathcal{P}'$.

Several constraints in $\mathcal{P}$ can prevent solutions to c from prevailing in $\mathcal{P}$. A first approach provides the user will a full-fledged explanation of the reasons for this, under the form of the minimal sets of constraints that cause the problem. It is based on MUC-finding algorithms. The idea is as follows.

Whenever a single element of $rel(c)$ is introduced as a new constraint in $\mathcal{P}$ and when this leads $\mathcal{P}'$ to be unsatisfiable, this means that $\mathcal{P}$ does not authorize $rel(c)$. Accordingly, we compute and exhibit MUCs in $\mathcal{P}'$. A least one constraint per MUC is expelled (or relaxed) from $\mathcal{P}$. This is iterated until $\mathcal{P}'$ becomes satisfiable. The whole process is then iterated for all elements of $rel(c)$. Thereafter, c is just safely added since we are sure that all its solutions can be now accommodated. When we know which elements of $rel(c)$ cannot be extended into solutions to $\mathcal{P}$, it is sufficient for the procedure to consider these elements, only.

Accordingly, any detected MUC provides a minimal set of constraints preventing an additional solution given by c from prevailing in $\mathcal{P}$. The algorithm *Relax-MUC* describes the skeleton of an iterative approach finding out and "breaking" in an automatic way MUCs iteratively, until no MUC remains. Note that the depicted algorithm does not return the set of detected MUCs but expels one constraint per MUC and simply delivers the final resulting relaxed constraint network.

A dual algorithm relying on the computation of MSSes was also experimented. However, it often proved less efficient than *Relax-MUC*, since search does not stop when a falsified constraint is found but only when a given number of falsified constraints is reached.

5 Combining both Situations

We consider now situations where more permissive constraints must be introduced in $\mathcal{P}$, while some previous partial solutions must be preserved at the same time. *Relax-MUC* does this job: it enlarges the set of solutions of $\mathcal{P}$ and thus preserves all of them, including thus all partial ones.

By expelling constraints *Relax-MUC* can enlarge the sets of partial solutions related to the variables occurring in the additional constraint c. Note that it can also enlarge sets of partial solutions related to variables not occurring in c. In some situations this is acceptable, provided that the user is given the full explanation of this phenomenon. In the resource allocation problem, the manager might need to approve the choice of constraints that are to be expelled or amended to enforce all partial solutions of c. She has to be made aware that by relaxing $\mathcal{P}$ in this way, the problem is transformed into a new one that possibly could give rise to new solutions about variables outside $var(c)$.

Algorithm 2: Relax-MUC

```
   input: $\mathcal{P} = \langle \mathcal{X}, \mathcal{C} \rangle$: a constraint network ;
          $c$: the additional constraint ;
   output: $\langle \mathcal{X}', \mathcal{C}' \rangle$: a relaxed network of $\mathcal{P}$ by $c$ ;
1  $\mathcal{C}' \leftarrow \mathcal{C}$ ; $\mathcal{X}' \leftarrow \mathcal{X} \cup var(c)$ ;
2  foreach $\vec{t} \in rel(c)$ do
3      Let $c_t$ s.t. $rel(c_t) = \{\vec{t}\}$ and $var(c_t) = var(c)$ ;
4      while $\langle \mathcal{X}', \mathcal{C}' \cup \{c_t\} \rangle$ is unsat do
5          $\langle \mathcal{X}', \mathcal{C}'' \rangle \leftarrow$ MUC ($\mathcal{X}', \mathcal{C}' \cup \{C_t\}$) ;
6          select one constraint $c''$ in $\mathcal{C}''$ s.t. $c'' \neq c_t$ ;
7          $\mathcal{C}' \leftarrow \mathcal{C}' \setminus \{c''\}$ ;
8  $\mathcal{C}' \leftarrow \mathcal{C}' \cup \{c\}$ ;
9  return $\langle \mathcal{X}', \mathcal{C}' \rangle$ ;
```

In other situations, the user might require the more permissive constraint c to have a more limited impact on $\mathcal{P}$: specifically, it must not enlarge the sets of partial solutions that are not directly related to variables occurring in c. Let us go back to the parental benefits issue. As explained in the introduction, $\mathcal{P}$ there translates complex fiscal rules. Enforcing a new legal rule relaxing the conditions allowing for the parental benefits could also have an impact on other variables of $\mathcal{P}$, linked to other fiscal issues. However, it might be the case that the new rule has been envisioned as having no impact on any other fiscal issue. In these circumstances, it is not acceptable to allow the possibility of enlarging sets of partial solutions (e.g., increase other possible benefits) outside the relaxed rule. In other words, in some situations, we only want to extend the range of partial solutions *for some given variables*. Specifically, in the example, these partial solutions must only concern the values for the variable expressing the number of children and the variable expressing whether or not the person is entitled to get the parental benefits. On the contrary, we do not want any other aspects of $\mathcal{P}$ to be changed. In technical words, this amounts to enforce a more permissive constraint c, while safely keeping the set of all partial solutions to variables not occurring in c and not enlarging it.

Relax-and-Preserve performs that task. For each variable x_i occurring in $var(c)$, all its occurrences in $\mathcal{P}$ are renamed by a new additional variable y_i (l. 1-7). The idea is then to enforce all partial solutions related to the variables x_i's of $var(c)$ through an additional constraint c_{new} (l. 8). c_{new} also ensures adequate connections between values of x_i's and y_i's. Whenever a partial solution (i.e., a tuple of $rel(c)$) can be extended into a solution of $\mathcal{P}$, x_i's and y_i's are pairwise assigned identically using this partial solution, and the resulting tuple is added within $rel(c_{new})$ (l. 10-12). Whenever the partial solution cannot be extended into a solution to $\mathcal{P}$, a series of correspondences are created between the tuple of values for the variables x_i's that forms the partial solution, and all the tuples of solution values for the variables y_i's in $\mathcal{P}$. The resulting tuples of values for $(x_1, \ldots, x_m, y_1, \ldots, y_m)$ are added as new tuples within $rel(c_{new})$ (l. 13-15). Note that although the algorithm is depicted as manipulating tuples, c_{new} can be recorded and inserted in intentional form within $\mathcal{C}'$ (l.16). Accordingly, the additional partial solutions introduced by c do not alter the behavior of the network with respect to any variable occurring in $\mathcal{P}$, except those occurring in c. Let us give an example.

Example 1 *Let $\mathcal{P} = \langle \mathcal{X}, \mathcal{C} \rangle$ a constraint network such that $\mathcal{X} = \{x_1, x_2, x_3\}$ with $dom(x_1) = dom(x_2) = dom(x_3) = \{1, 2\}$ and $\mathcal{C} = \{(x_1 = x_2), (x_1 = x_3), (x_2 = x_3)\}$ and a more permissive constraint $c = (x_1 \leq x_2)$. The transformed network is $\mathcal{P}' = \langle \mathcal{X}', \mathcal{C}' \rangle$ where $\mathcal{X} = \{x_1, x_2, x_3, y_1, y_2\}$, $dom(y_1) = dom(y_2) = \{1, 2\}$ and $\mathcal{C} = \{(y_1 = y_2), (y_1 = x_3), (y_2 = x_3), c_{new}\}$ with $var(c_{new}) = \{x_1, x_2, y_1, y_2\}$ and $rel(c_{new}) = \{(1,1,1,1), (2,2,2,2), (1,2,1,1), (1,2,2,2)\}$.*

Algorithm 3: Relax-and-Preserve

```
input: $\mathcal{P} = \langle \mathcal{X}, \mathcal{C} \rangle$ a constraint network ;
       a constraint $c$ with $var(c) = \{x_1, \ldots, x_m\}$ ;
output: $\mathcal{P}' = \langle \mathcal{X}', \mathcal{C}' \rangle$ a constraint network s.t. the
        partial solutions to $\mathcal{P}$ w.r.t. $\mathcal{X} \setminus var(c)$ are
        preserved and $\mathcal{P}'$ extends any solution of $c$ ;
1  $\mathcal{X}' \leftarrow \mathcal{X} \cup \{y_1, \ldots, y_m\}$ where $dom(y_i) = dom(x_i)$;
2  $\mathcal{C}' \leftarrow \emptyset$;
3  foreach $c_j \in \mathcal{C}$ do
4      if $var(c_j) \cap \{x_1, \ldots, x_m\} = \emptyset$ then
5          $\mathcal{C}' \leftarrow \mathcal{C}' \cup \{c_j\}$;
6      else
7          $\mathcal{C}' \leftarrow \mathcal{C}' \cup \{c'_j\}$ where $c'_j$ is obtained from $c_j$ by
           replacing every occurrence of the variable $x_i$ by
           $y_i$ for all $x_i$ in $var(c_j) \cap \{x_1, \ldots, x_m\}$;
8  Let $c_{new}$ s.t. $var(c_{new}) = \{x_1, \ldots, x_m, y_1, \ldots, y_m\}$ ;
9  $rel(c_{new}) \leftarrow \emptyset$;
10 foreach $\vec{t} \in rel(c)$ do
11     if $\vec{t}$ can be extended into a model of $\mathcal{P}$ then
12         $rel(c_{new}) \leftarrow rel(c_{new}) \cup \{(\vec{t}, \vec{t})\}$;
13     else
14         foreach $\vec{t_1} \in \mathcal{S}_{(\mathcal{P},\mathcal{X})}$ do
15             $rel(c_{new}) \leftarrow rel(c_{new}) \cup \{(\vec{t}, \vec{t_1})\}$;
16 $\mathcal{C}' \leftarrow \mathcal{C}' \cup \{c_{new}\}$;
17 return $\mathcal{P}' \leftarrow \langle \mathcal{X}', \mathcal{C}' \rangle$ ;
```

Interestingly, the concept of partial solution allows forms of subsumption and equivalence *modulo a set of variables* between networks to be defined.

Definition 8 (Subsumption between constraint networks) *Let $\mathcal{P}' = \langle \mathcal{X}', \mathcal{C}' \rangle$, $\mathcal{P}" = \langle \mathcal{X}", \mathcal{C}" \rangle$ and $\mathcal{X}$ a set of variables. $\mathcal{P}'$ subsumes $\mathcal{P}"$ modulo $\mathcal{X}$ (noted $\mathcal{P}' \models_{\mathcal{X}} \mathcal{P}"$) iff for all models $\mathcal{A}_1$ of $\mathcal{P}'$ there exists a model $\mathcal{A}_2$ of $\mathcal{P}"$ such that $\mathcal{A}_1(\mathcal{X} \cap \mathcal{X}' \cap \mathcal{X}") \subseteq \mathcal{A}_2(\mathcal{X} \cap \mathcal{X}")$.*

Algorithm 4: Relax-and-Preserve-Naive

```
input: $\mathcal{P} = \langle \mathcal{X}, \mathcal{C} \rangle$: a constraint network, $c$ a constraint
output: $\langle \mathcal{X}', \mathcal{C}' \rangle$: a constraint network s.t. the partial
        solutions to $\mathcal{P}$ w.r.t. $\mathcal{X} \setminus var(c)$ are preserved
        and $c$ inserted
1  $\mathcal{S} \leftarrow$ AllSolutions($\mathcal{P}$) ;
2  Let $\mathcal{S}_{(\mathcal{P},\mathcal{X})}$ be the set of partial solutions obtained from
   $\mathcal{S}$ w.r.t. $\mathcal{X} \setminus var(c)$ ;
3  Let $c_{\mathcal{S}_{(\mathcal{P},\mathcal{X})}}$ be the constraint s.t.
   $var(c_{\mathcal{S}_{(\mathcal{P},\mathcal{X})}}) = var(\mathcal{S}_{(\mathcal{P},\mathcal{X})})$ and
   $rel(c_{\mathcal{S}_{(\mathcal{P},\mathcal{X})}}) = \{\mathcal{A} | \mathcal{A} \in \mathcal{S}_{(\mathcal{P},\mathcal{X})}\}$ ;
4  $\mathcal{C}' \leftarrow \{c_{\mathcal{S}_{(\mathcal{P},\mathcal{X})}}, c\}$ ; $\mathcal{X}' \leftarrow \mathcal{X} \cup var(c)$ ;
5  return $\langle \mathcal{X}', \mathcal{C}' \rangle$;
```

Definition 9 (Equivalence between constraint networks) *Let $\mathcal{P}' = \langle \mathcal{X}', \mathcal{C}' \rangle$, $\mathcal{P}" = \langle \mathcal{X}", \mathcal{C}" \rangle$ and $\mathcal{X}$ a set of variables. $\mathcal{P}'$ is equivalent to $\mathcal{P}"$ modulo $\mathcal{X}$ (noted $\mathcal{P}' \equiv_{\mathcal{X}} \mathcal{P}"$) iff $\mathcal{P}' \models_{\mathcal{X}} \mathcal{P}"$ and $\mathcal{P}" \models_{\mathcal{X}} \mathcal{P}'$.*

Accordingly, we have that:

Property 1 *When Relax-and-Preserve is run on $\mathcal{P} = \langle \mathcal{X}, \mathcal{C} \rangle$ with c being the more permissive constraint to enforce, it delivers $\mathcal{P}' = \langle \mathcal{X}', \mathcal{C}' \rangle$ s.t. $\mathcal{P}' \equiv_{(\mathcal{X} \cap \mathcal{X}') \setminus var(c)} \mathcal{P}$ and $\langle var(c), \{c\} \rangle \equiv_{var(c)} \mathcal{P}'$.*

We also provide a naive algorithm *Relax-and-Preserve-Naive* that computes solutions in extension and delivers them as forming $\mathcal{P}'$. As explained earlier, it does not meet our goal since a set of tuples is delivered and the structure of the problem under the form of constraints is lost for further explanations or possible evolutions of the network. Besides, it proved intractable on many instances, as it requires all complete solutions of $\mathcal{P}$ to be computed. Note that in its depicted form, *Relax-and-Preserve* creates a constraint in extension within $\mathcal{P}'$: however, this is a local phenomenon within $\mathcal{P}'$ and it can be rewritten in intention.

6 Experimental Computational Results

All results below have been obtained on a Quad-core Intel XEON X5550 32Gb. *Relax-MUC*, *Relax-and-Preserve* and *Relax-and-Preserve-Naive* are available from https://www.cril.univ-artois.fr/~mazure together will the experimental data. They have been run on 986 instances taken from the satisfiable benchmarks of the CSP competitions repository http://cpai.ucc.ie/08/ and http://cpai.ucc.ie/09/. In Table 1, a sample of typical results is provided. Time-out ("TO") was set to 30 mins. The full table is also available from https://www.cril.univ-artois.fr/~mazure. Although *Relax-MUC* has a slightly different general scope, we compared the three algorithms on the same instances. Columns of Table 1 provide the name of the instances $\mathcal{P}$ and their numbers of constraints and variables, successively. Then, it lists the number of variables and tuples of the additional constraint c. c was generated as follows: $var(c)$ was randomly selected as a subset of the variables $\mathcal{X}$ from $\mathcal{P}$. The cardinality of this subset ranged from 2 to 4. $rel(c)$ contained all

Instance			constraint c		*Relax-MUC*		*Relax-and-Preserve*			*Relax-and-Preserve-Naive*
name	$\#\mathcal{C}$	$\#\mathcal{X}$	$\#var(c)$	$\#rel(c)$	time (secs.)	$\#rm$	time (secs.)	$\#var(c_{new})$	$\#rel(c_{new})$	time (secs.)
aim-200-2-0-sat-2-ext	382	200	4	4	104.74	2	0.97	8	4	3.86
air04	1646	8904	3	8	483.51	0	487.9	6	8	TO
bdd-21-133-18-78-9-ext	133	21	4	5	TO	—	30.09	8	8	12.82
bf-0432-007-ext	1943	970	4	10	8.01	3	0.07	8	24	TO
bmc-ibm-02-08	14493	2810	4	1028	712.09	10	25.01	8	7154	TO
bqwh-15-106-53-ext	596	106	3	14	29.74	2	0.72	6	56	2.25
bqwh-15-106-82-ext	600	106	3	35	TO	—	2.35	6	243	11.83
circ-4-3	764	144	4	13	TO	—	0.23	8	24	5.77
composed-25-10-20-6-ext	620	105	4	5630	40.97	10	40.45	8	6726920	TO
crossword-m1c-lex-vg4-7-ext	11	28	3	4459	7.1	4	56.61	6	380451	23.21
crossword-m1-lex-15-02	944	191	2	187	488.01	6	71.63	4	4075	TO
david-11	406	87	3	1331	2.32	0	2.07	6	1331	TO
domino-800-100	800	800	2	2504	32.53	1	28.11	4	194000	1.15
driverlogw-05c-sat-ext	6173	351	3	21	TO	—	0.67	6	120	TO
e0ddr2-10-by-5-6	265	50	2	15654	62.95	2	151.37	4	1168500	TO
fapp01-0200-8	2053	200	2	5019	TO	—	TO	—	—	TO
frb30-15-1-ext	284	30	2	57	TO	—	1.2	4	112	1.89
games120-9	638	120	4	6014	18.1	1	19.04	8	1067256	TO
geo50-20-d4-75-47-ext	383	50	4	159481	TO	—	376.31	8	37264032	TO
geo50-20-d4-75-88-ext	354	50	4	158878	TO	—	598.69	8	135424854	TO
graceful–K4-P2	164	24	2	198	107.89	6	17.4	4	5208	72.96
graph2-f24	2245	400	3	1555	39.46	1	25.38	6	586240	TO
hanoi-3-ext	5	6	2	14	0	2	0	4	14	0.34
ii-32e1	1408	444	4	13	3.78	1	12.63	8	24	TO
jean-10	254	80	4	10000	9.27	0	8.87	8	10000	TO
jnh210-ext	799	100	4	5	90.85	1	0.12	8	8	188.30
jnh213-ext	797	100	4	7	1782.02	4	0.04	8	16	3.10
langford-2-8	128	16	3	1084	0.82	5	0.71	6	80400	1.07
lard-88-88	3828	88	2	7854	648.9	85	117.41	4	15422176	TO
mknap-1-0	11	6	4	4	0	7	0	8	4	0.35
mps-p0033	15	33	4	13	0.62	1	0.15	8	24	3.83
mps-p0033	15	33	4	13	0.62	1	0.15	8	24	3.83
myciel5-6	236	47	4	1134	0.63	1	0.71	8	59400	TO
os-taillard-4-100-6	48	16	2	6289	10.57	4	8.95	4	447696	436.64
par-16-1-c	1581	634	3	2	TO	—	4.78	6	2	55.39
primes-10-80-2-7	80	100	2	196	TO	—	27.57	4	196	1.37
qcp-10-67-5-ext	822	100	2	1	0	0	0.01	4	1	TO
queens-5-5-5-ext	160	25	4	261	29.02	34	0.13	8	17080	0.91
qwh-15-106-4-ext	607	106	4	22	TO	—	0.19	8	60	2.54
qwh-15-106-4-ext	2324	225	4	63	29.97	7	3.25	8	495	3.90
radar-30-70-4.5-0.95-98	12471	10990	4	16	136.59	0	110.08	8	16	TO
radar-8-30-3-0-32	64	180	4	199	67.51	2	29.56	8	3600	TO
ramsey-25-4	2300	300	3	64	1.84	0	1.73	6	64	TO
rand-2-40-16-250-350-36-ext	250	40	3	2907	1156.32	3	736.78	6	1825747	17.51
rand-2-30-15-306-230-23-ext	221	30	2	57	1412.45	12	0.76	4	57	3.23
renault-mod-36-ext	154	111	3	12	2.37	2	0.82	6	40	TO
route	24492	72	2	51	TO	—	254.35	4	420	TO
ruler-34-8-a4	350	8	2	307	1741.85	17	6.46	4	612	9.13
s3-3-3-4	616	228	3	3	TO	—	43.45	6	4	107.92
scen2-f24	1235	200	3	7084	43.17	1	43.07	6	134254	TO
scen5	2598	400	2	326	TO	—	20.36	4	972	TO
series-9	72	17	4	1063	1.14	14	0.56	8	53583	2.54
ssa-7552-158-ext	1985	1363	3	3	56.42	1	0.04	6	4	TO
super-jobShop-enddr1-10	845	100	2	6904	123.17	3	401.68	4	7889700	TO
tsp-20-2-ext	230	61	2	5010	113.07	3	36.67	4	35028	3.23
will199GPIA-7	7065	701	4	2401	205.43	0	196.62	8	2401	TO

Table 1: *Relax-MUC*, *Relax-and-Preserve*, *Relax-and-Preserve-Naive* results

tuples from the cartesian product of the domain of the variables of $var(c)$ that can be extended into solutions of $\mathcal{P}$, together with an additionally 25% of tuples selected randomly amongst the tuples from that cartesian product that cannot be extended into solutions of $\mathcal{P}$. When less than 4 tuples from the cartesian product were already satisfied, no new tuple was added and *Relax-MUC* did not remove any constraint (an interesting conclusion could however be drawn about the benchmark when all tuples in the cartesian product were satisfied: no variable in any pair of concerned variables does restrict the range of solutions for the other one). Finally, the computing time to deliver $\mathcal{P}$ is listed for each method, as well as the number of constraints dropped ($\#rm$) by *Relax-MUC* and the number of variables and tuples of c_{new} introduced by *Relax-and-Preserve*. Again, let us stress that the algorithms address different problems and the computational results are just intended to illustrate the actual viability of the approaches. Not surprisingly, *Relax-and-Preserve-Naive* appears often intractable from a practical point of view. *Relax-and-Preserve* is more efficient than *Relax-MUC* for most instances. In this respect, let us stress that *Relax-MUC* has been built in a way that it should be used in a step-by-step and interactive mode, giving the user the full-fledged knowledge and freedom of decision about the choice of constraints that will be dropped or weakened.

Conclusion

In the CSP-related literature, relaxation is a transformation paradigm to recover satisfiability for unsatisfiable networks [Jussien and Boizumault, 1996; Georget *et al.*, 1999; Amilhastre *et al.*, 2002; Nordlander *et al.*, 2003]. To the best of our knowledge, relaxation and preserving partial solutions have not been studied so far in the context of evolving satisfiable networks. This contribution is an attempt to fill this gap. More generally, we believe that the dynamics of constraint networks is an issue of importance that needs further practical and theoretical studies. As a promising perspective, we also believe that many concepts introduced in this paper could be reused for addressing the problem of merging constraint networks.

References

[Amilhastre *et al.*, 2002] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic CSPs—application to configuration. *Artificial Intelligence*, 135(1-2):199–234, 2002.

[Bakker *et al.*, 1993] René R. Bakker, F. Dikker, Frank Tempelman, and Petronella Maria Wognum. Diagnosing and solving over-determined constraint satisfaction problems. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, volume 1, pages 276–281, 1993.

[Beldiceanu *et al.*, 2012] Nicolas Beldiceanu, Narendra Jussien, and Eric Pinson, editors. *Proceedings of the Ninth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2012)*, volume 7298 of *Lecture Notes in Computer Science*. Springer, 2012.

[Belov and Marques-Silva, 2011] Anton Belov and João Marques-Silva. Accelerating MUS extraction with recursive model rotation. In Per Bjesse and Anna Slobodová, editors, *Proceedings of the International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA*, pages 37–40, 2011.

[Belov and Marques-Silva, 2012] Anton Belov and João Marques-Silva. MUSer2: An efficient MUS extractor. *JSAT*, 8(1/2):123–128, 2012.

[Eiter and Gottlob, 1992] Thomas Eiter and Georg Gottlob. On the complexity of propositional knowledge base revision, updates, and counterfactuals. *Artificial Intelligence*, 57(2-3):227–270, 1992.

[Georget *et al.*, 1999] Yan Georget, Philippe Codognet, and Francesca Rossi. Constraint retraction in CLP(FD): Formal framework and performance results. *Constraints*, 4(1):5–42, 1999.

[Grégoire *et al.*, 2007] Éric Grégoire, Bertrand Mazure, and Cédric Piette. MUST: Provide a finer-grained explanation of unsatisfiability. In Christian Bessière, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP'07)*, pages 317–331. Springer, 2007.

[Grégoire *et al.*, 2008] Éric Grégoire, Bertrand Mazure, and Cédric Piette. On finding minimally unsatisfiable cores of CSPs. *International Journal on Artificial Intelligence Tools (IJAIT)*, 17(4):745 – 763, 2008.

[Han and Lee, 1999] Benjamin Han and Shie-Jue Lee. Deriving minimal conflict sets by CS-Trees with mark set in diagnosis from first principles. *IEEE Transactions on Systems, Man, and Cybernetics*, 29(2):281–286, 1999.

[Hémery *et al.*, 2006] Fred Hémery, Chirstophe Lecoutre, Lakhdar Saïs, and Frédéric Boussemart. Extracting MUCs from constraint networks. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06)*, pages 113–117, 2006.

[Janota and Marques-Silva, 2011] Mikolás Janota and João Marques-Silva. cmMUS: A tool for circumscription-based MUS membership testing. In James P. Delgrande and Wolfgang Faber, editors, *Proceedings of Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada*, volume 6645 of *Lecture Notes in Computer Science*, pages 266–271. Springer, 2011.

[Junker, 2001] Ulrich Junker. QuickXplain: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints (CONS-1)*, 2001.

[Junker, 2004] Ulrich Junker. QuickXplain: Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI'04)*, pages 167–172, 2004.

[Jussien and Barichard, 2000] Narendra Jussien and Vincent Barichard. The PaLM system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP'00*, pages 118–133, 2000.

[Jussien and Boizumault, 1996] Narendra Jussien and Patrice Boizumault. Maintien de déduction pour la relaxation de contraintes. In Jean-Louis Imbert, editor, *Actes des Cinquièmes Journées Francophones de Programmation Logique et Programmation par Contraintes (JFPLC'96)*, pages 239–254. Hermes, 1996.

[Marques-Silva and Lynce, 2011] João P. Marques-Silva and Inês Lynce. On improving mus extraction algorithms. In Karem A. Sakallah and Laurent Simon, editors, *Proceedings of Theory and Applications of Satisfiability Testing - 14th International Conference, SAT 2011, Ann Arbor, MI, USA*, volume 6695 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2011.

[Milano, 2012] Michela Milano, editor. *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP'2012)*, volume 7514 of *Lecture Notes in Computer Science*. Springer, 2012.

[Nordlander *et al.*, 2003] Tomas Nordlander, Ken Brown, and Derek Sleeman. Constraint relaxation techniques to aid the reuse of knowledge bases and problem solvers. In *Proceedings of the Twenty-third SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence*, pages 323–335, 2003.

[Pesant, 2012] Gilles Pesant, editor. *Constraints (Journal)*, volume 17. Springer, 2012.

[Petit *et al.*, 2003] Thierry Petit, Christian Bessière, and Jean-Charles Régin. A general conflict-set based framework for partial constraint satisfaction. In *Proceedings of SOFT'03: Workshop on Soft Constraints held with CP'03*, 2003.

[Rossi *et al.*, 2006] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.