A Generic Framework for Learning Implied Constraints

Christian Bessiere[1], Rémi Coletta[1], and Thierry Petit[2]

[1] LIRMM (CNRS/University of Montpellier),
161 Rue Ada, 34392 Montpellier Cedex 5, France.
{bessiere,coletta}@lirmm.fr
[2] LINA (FRE CNRS 2729), École des Mines de Nantes,
4, Rue Alfred Kastler, FR-44307 Nantes Cedex 3, France.
thierry.petit@emn.fr

Abstract. Many models may exist to encode a problem as a constraint satisfaction problem. One lack of constraint programming is the strong expertise required to build a model that leads to an efficient solving. Hence, there is an increasing interest in automatic reformulation of models. This paper presents a technique consisting of *learning* implied constraints from allowed and forbidden assignments of values to sets of variables. Such constraints can be added to a model to improve the solving process. We propose a general framework to learn parameterized implied constraints. We apply this framework to the case of the global cardinality constraint and we show by experiments the interest of our approach.

1 Introduction

Constraint programming is a powerful framework for solving problems in various areas: scheduling, planning, design, configuration, etc. Using constraint programming requires a strong expertise. Unfortunately, not all development teams have constraint programming experts in their staff. Thus, automatic model reformulation is a key issue [9]. Firsch et al. [6] have proposed a pure reformulation technique. They present a system that refines an abstract specification of a problem into alternative constraint programs. Colton and Miguel [4] exploit a theory formation program to provide concepts and theorems which can be translated into *implied* constraints. Indeed, one way to improve a model consists of adding implied constraints. An implied constraint is not mandatory to express the problem but it can be very useful to solve it [15]. A good example of the use of implied constraints is sport scheduling problems [14].

In this paper, we assume that we have a model which expresses correctly the problem, but the solving time is not satisfactory. We wish to improve it by automatically adding implied constraints. Our idea is to define a learning algorithm that deduces new constraints from assignments of values to sets of variables. That is not reformulation in the usual meaning of the word. Notably, our technique should be able to learn constraints which are not deductible by an analysis of the structure and the semantics of the constraints. They may on the contrary strongly depend on initial domains, and become different if we modify those initial domains. Notice that machine learning techniques have already been applied to constraint programming in different contexts. In [3], a learning technique is used for acquiring an initial model from an interaction with a user. In [8], authors deal with the specific task of acquiring relaxed propagators of the constraints already belonging to the model, in order to improve the solving time.

Here, we aim at learning *parameterized* constraints. The set of tuples allowed by a parameterized constraint not only depends on the variables it involves and their domains, but also on some extra information provided by *parameters*, which are not necessarily part of the problem variables. For instance, the $\texttt{NValue}(k, [X_1, \ldots, X_n])$ constraint holds iff k is equal to the number of different values taken by $X_1, \ldots, X_n$. k can be a CSP variable (in its more general definition) or a parameter which leads to different sets of allowed tuples on the X_i's depending of the values it can take. A parameter can be a set of integers, a range, or a single integer value. In our context, a learning algorithm would try, for example, to learn the smallest range of possible values for k such that no solution exists outside this range. If the algorithm returns a lower bound for that range strictly greater than 0, or an upper bound strictly less than n, then the learned constraint can be of interest for improving the solving process. Following that idea, we propose a framework to learn automatically parameterized constraints in a given model. The principle is to exploit the information provided by solutions and non-solutions of a sub-problem to learn the parameters of a possibly implied constraint involving the variables of that sub-problem. If the learned constraint is of interest, then it can be added to the model to improve the solving efficiency. A lot of existing constraints [1] may be used as implied parameterized constraints: `AtLeast`, `AtMost`, `Change`, `Common`, `Count`, `Gcc`, `Max`, `Min`, `NValue`, etc. Note that all these constraints can involve sets of variables of any size. To validate the impact of our work, we apply our paradigm to the case of global cardinality constraints (`Gcc`). A `Gcc` constrains the number of times each value v is taken to be equal to a parameter p_v. When the possible values for the p_{v_i} are ranges, there exists an efficient arc consistency algorithm for this constraint [13]. In this case, `Gcc` corresponds to dealing globally with a a conjunction of `AtLeast` and `AtMost` constraints. That global constraint is thus a good candidate for being an implied constraint useful in various contexts.

This paper is organized as follows. Section 2 points out the interest of using implied constraints by a short scheduling example. Section 3 presents our generic framework for learning parameterized constraints. Section 4 discusses the set of variables and constraints to use in the learning process. Section 5 provides the application of our technique to global cardinality through two sets of experiments, respectively in the context of satisfaction and optimization. Finally we conclude in Section 6.

2 Implied Constraints

When a problem has been encoded with variables and constraints, some additional implied properties may be satisfied by any solution. In some cases it is interesting to express those properties explicitly by adding some new constraints to the model. Indeed, constraint programming decreases the search space by reducing domains of variables. New constraints may come up with efficient filtering algorithms leading to new reductions. In the literature, such constraints are called *implied* constraints [15, 14]. In the Ilog Solver 5.0. user's manual [7], p. 279, the concept of implied constraint is pointed out by adding a `Gcc` to improve the solving of a graph coloring problem.

Let us focus on another example. We consider a simple problem where m tasks have to be scheduled. All tasks have the same duration d and they are partitioned into two sets: m_1 tasks require 1 resource, $m_2 = m - m_1$ tasks require 2 resources. Time is represented by integer units starting at point 0. At a given point of time, the maximum usable resource is bounded by $maxi$. A task should start separated from each other by at least two units of time. Some random precedence constraints between start of tasks are added. Given those data and a makespan (maximum time) p equal to $2 * (m-1) + d$ (the best possible makespan w.r.t. the constraint on starting times), the question is to find a feasible schedule. We implemented with Choco [2] a naive model where the start of each task is represented by a variable, and resources are represented at each unit of time by an array of variables, where each index corresponds to the consumption of a given task at this point of time. Constraints on durations and separation of beginning of tasks are primitive ones, and sum constraints are used for the maximum allowed resource. We place us in the context of a non expert user, so we run the default search heuristic of Choco (the variable with minimum domain first, assigned with its smallest available value). We stop the search when it exceeds 60 seconds. Even with $m = 4$ and $d = 4$, and 2 precedence constraints, we obtain the bad results reported in Table 1.

Table 1. Results on the basic model.

m_1/m_2	$maxi$	#nodes	time (sec.)
4/0	4	—	> 60
4/0	3	42,129	7.4
4/0	2	85	0.12
3/1	4	45	0.030
3/1	3	33	0.041
3/1	2	7 (no sol)	0.030

A non-expert user could be very surprised by these results. Notably, restricting more $maxi$, the maximum available amount of resource, allows to find a solution faster. The explanation is that when resource constraints propagate, they reduce significantly the search space and compensate the lack of propagation of constraints related to the ordering of tasks.

It is necessary to improve the model. One way is to look if constraints on the ordering of tasks do not induce limits on the number of tasks at each point of time.[1] The value $p = 2(m-1) + d$ is exactly the minimal possible one for scheduling all the tasks if they are separated by at least two units. Therefore we have to place tasks as soon as possible. This means one task at time zero, one task at time 2, etc. This fact can be expressed explicitly in our problem with a `Gcc` constraint.

Definition 1. *A `Gcc` constraint C as defined in [13] involves a set of variables $var(C)$ and a sequence of parameters $\Delta = (p_{v_1}, \ldots, p_{v_k})$, where p_{v_i} is the number of times value v_i appears in a tuple. Each p_{v_i} takes its value in the range $[min(p_{v_i})..max(p_{v_i})]$. C is satisfied when each p_{v_i} belongs to its range.*

[1] A good search heuristic may have also helped to solve the problem, but the automatic generation of robust search heuristics is out from the topic of this paper.

Variables in $\mathcal{X}$ represent starts of tasks, and their values the points of time. For every value v, $max(p_v) = maxi$; if v is even and v is smaller than $2m - 1$, $min(p_v) = 1$ (i.e., one task has to start there), otherwise $min(p_v) = 0$. Other rules over cardinalities could be deduced, for instance no task begins at odd points (i.e., $max(p_v) = 0$ if v is odd). However, even the simple `Gcc` constraint defined above leads to significant improvements (see Table 2).

Table 2. Results when an implied `Gcc` is added to the basic model.

m_1/m_2	$maxi$	#nodes	time (sec.)
4/0	4	46	0.04
4/0	3	46	0.04
4/0	2	25	0.04
3/1	4	45	0.05
3/1	3	32	0.04
3/1	2	3 (no sol)	0.03

This example shows that for a non-expert in constraint programming, even very small problems can lead to surprisingly bad results that could be avoided by adding simple implied constraints. One can also cite the example of models generated from databases by computer programs, or web-based configuration systems. Therefore new methods for improving models in an automatic way are welcome.

3 Learning Parameterized Constraints

The set of tuples allowed by a classical (or ground) constraint is completely known once the variables it involves and their domains are set. A parameterized constraint may allow different sets of tuples depending on the possible values for its parameters. In this section, we first define and use parameterized constraints in the most general way to let our work be as general as possible. We then discuss the application of our technique to special types of representations of the parameters. (We suppose parameters take values in the set $\mathbb{Z}$ of integers.)

Definition 2. *Given a set of parameters $\Delta = \{p_1, ..., p_{|\Delta|}\}$ taking their values in $\mathbb{Z}$, a* parameterized constraint *C is a constraint which expresses a property on the variables it involves (denoted by $var(C)$) depending on the possible values for its set of parameters Δ. Given $s \in \mathbb{Z}^{|\Delta|}$, a assignment of values to the parameters, $C(s)$ refers to the set of allowed tuples of the constraint C when each p_i in Δ takes the i^{th} value in s, noted $s[p_i]$. Given $S \subseteq \mathbb{Z}^{|\Delta|}$, $C(S) = \bigvee_{s \in S} C(s)$.*

Definition 3. *Given a constraint C with parameters Δ, given a tuple e on $var(C)$ (or on a superset of $var(C)$), S_e denotes the set of combinations of values s in $\mathbb{Z}^{|\Delta|}$ such that $C(s)$ accepts e.*

Since our goal is to learn implied constraints on any type of constraint problem, we focus on those parameterized constraints where for any tuple e on $var(C)$, $S_e \neq \emptyset$. Note that all parameterized constraints mentioned in introduction verify this.

The basic idea in the rest of the paper is to learn the parameters of an implied constraint C by exploiting information provided by solutions and non-solutions of a subproblem extracted from the initial model. The selection of that subproblem, in terms of variables and constraints that should be considered is discussed in section 4. We assume that the constraint C for wich we will learn the parameters has been chosen.

Definition 4. *$\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ denotes the problem used to learn an implied parameterized constraint. Given a problem $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, an* instance *$e$ is an assignment of values to variables in $\mathcal{X}$. It is* positive *if e belongs to $Sol(\mathcal{P})$, otherwise it is* negative.

The goal is to use such positive and negative instances to learn a new implied parameterized constraint.

Definition 5. *An* implied constraint *C for $\mathcal{P}$ is a constraint such that $var(C) \subseteq \mathcal{X}$ and the set $Sol(\mathcal{P})$ of solutions of $\mathcal{P}$ is equal to $Sol((\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \{C\}))$; Namely, for any instance $e \in Sol(\mathcal{P}), e[var(C)]$ is allowed by C.*

We consider only constraints for which $C(\mathbb{Z}^{\Delta})$ is the universal constraint (i.e., for any tuple e, $S_e \neq \emptyset$). Thus, following Definition 5, $C(\mathbb{Z}^{\Delta})$ is an implied constraint for the problem $\mathcal{P}$. The objective of a learning algorithm will be to learn a 'target' set $T \subseteq \mathbb{Z}^{|\Delta|}$, as small as possible, such that $C(T)$ is still an implied constraint. Any $s \in Z^{\Delta}$ which is not necessary to accept some solutions of $\mathcal{P}$ can be removed from T. The tighter the learnt constraint is, the more promising its filtering power will be when this constraint will be added to the initial model. Analyzing positive (resp. negative) instances of $\mathcal{P}$, we show in the next subsection how to compute the target set T.

3.1 Deductions from Positive and Negative Instances

Notation 1 *Given a problem $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ and a parameterized constraint C, we denote by $required(T)$ the set of elements r of $\mathbb{Z}^{|\Delta|}$ such that $r \in required(T)$ iff $\forall S \subseteq \mathbb{Z}^{|\Delta|}$, $C(S \setminus \{r\})$ is not an implied constraint, namely r is compulsory in T if we want $C(T)$ to be an implied constraint in $\mathcal{P}$. $poss(T)$ denotes those r in $\mathbb{Z}^{|\Delta|}$ for which we do not know if they must belong to $required(T)$.*

In other words, $required(T)$ represents those combinations of values for the parameters that are necessary to preserve the set of solutions of $\mathcal{P}$, and $poss(T)$ those for which we have not proved yet that we would not lose solutions without them.

The fact that an implied constraint should not remove solutions when we add it to the problem leads to the following property w.r.t. positive instances.

Property 1. Let e^+ be a positive instance and $T \subseteq \mathbb{Z}^{|\Delta|}$. If $C(T)$ is an implied constraint for $\mathcal{P}$ then $T \cap S_{e^+} \neq \emptyset$.

Proof. e^+ is a solution of $\mathcal{P}$. At least one $s \in T$ should accept that instance to preserve the set of solutions of $\mathcal{P}$. So, $s \in S_{e^+}$ by Definition 3. □

Corollary 1. *Let e^+ be a positive instance of $\mathcal{P}$ and C a parameterized constraint. If there is a unique $s \in \mathbb{Z}^{|\Delta|}$ such that $e^+[var(C)] \in C(s)$ then $s \in required(T)$.*

Negative instances will help to remove from the set $poss(T)$ the combinations of parameters that can be removed without losing solutions. When receiving a negative instance e^-, we want to know if it is possible to reduce the target set T in order to reject e^- while preserving $Sol(\mathcal{P})$. It is of course not possible to remove directly from $poss(T)$ the elements of S_{e^-} since they can be necessary to 'cover' some of the solutions of $\mathcal{P}$. The following property gives us conditions under which we can do that.

Property 2. Let $\mathcal{P}$ be a problem, C a parameterized constraint, and $S \subset \mathbb{Z}^{|\Delta|}$. If $\mathcal{P}$ augmented with the constraint $C(S)$ has no solution then $C(\mathbb{Z}^{|\Delta|} \backslash S)$ is an implied constraint for $\mathcal{P}$.

Proof. By previous assumption, we know that $C(\mathbb{Z}^{|\Delta|})$ is an implied constraint for $\mathcal{P}$. Since $\mathcal{P} \cup \{C(S)\}$ is inconsistent, we know that $\nexists e \in Sol(\mathcal{P})$ such that $S_e \cap S \neq \emptyset$. Thus $Sol(\mathcal{P} + C(\mathbb{Z}^{|\Delta|})) = Sol(\mathcal{P} + C(\mathbb{Z}^{|\Delta|} \backslash S))$. □

Corollary 2. *Let $\mathcal{P}$ be a problem, C a parameterized constraint, $S \in \mathbb{Z}^{|\Delta|}$, and $poss(T)$ a set such that $C(poss(T))$ is implied on $\mathcal{P}$. If $\mathcal{P}$ augmented with the constraint $C(S)$ has no solution then $C(poss(T) \setminus S)$ is an implied constraint for $\mathcal{P}$.*

In Algorithm 1, we give a brute-force technique to learn implied constraints when no extra-information is known from the parameters. It uses corollaries 1 and 2. It takes as input a parameterized constraint and a set of instances E of $\mathcal{P}$. The output gives us a set of combinations of values for the parameters, such that we are guaranteed that no solution of the problem is lost if we add C to $\mathcal{P}$ with these parameters.

Algorithm 1: Learning Algorithm for Parameterized Constraints.

Input: C, Δ, $E = \{e_1, \ldots, e_k\}$
Output: $poss(T)$
$required(T) \leftarrow \emptyset$;
$poss(T) \leftarrow \emptyset$;
while $(E \neq \emptyset)$ ***and*** $(required(T) \neq poss(T))$ **do**
  Pick $e \in E$;
  if e *is positive* **then**
1   **if** $|S_e| = 1$ **then** $required(T) \leftarrow required(T) \cup S_e$; /* Corol. 1*/
  else
    for *some* $S \subseteq S_{e^-} \cap (poss(T) \setminus required(T))$ **do**
2   **if** $Sol(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup C(S)) = \emptyset$ **then** $poss(T) \leftarrow poss(T) \setminus S$; /* Corol. 2*/
      else put an element of $Sol(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup C(S))$ as positive instance in E

Checking whether $\mathcal{P}$ augmented with a constraint $C(S)$ is inconsistent (line 2) is of course an NP-hard problem. Even if $\mathcal{P}$ in this learning phase is not supposed to be the whole problem we want to reformulate but just a subpart, it is necessary to follow some heuristics to avoid huge numbers of NP-hard calls to a solver.

3.2 Using properties on the constraints and parameters

For a given parameterized constraint C, different representations of parameters may exist. The complexity of associated filtering algorithms generally differ according to the representation used. The more general case studied until now (Algorithm 1) consists of considering that allowed tuples of parameters for C are given in extension as a set T.

In this setting, corollary 1 may seem very weak. However, in practice, many parameterized constraints satisfy the following property: any two different combinations of parameters in $\mathbb{Z}^{|\Delta|}$correspond to disjoint sets of allowed tuples on $var(C)$. For instance, this is the case for the the Gcc: Each combination of parameters imposes a precise number of occurrences for each value.

Definition 6. *A constraint C is* parameter-partitioned *iff for any $s, s' \in \mathbb{Z}^{|\Delta|}$, if $s \neq s'$ then $C(s) \cap C(s') = \emptyset$. Given a tuple e on $var(C)$, s_e denotes the single element in S_e.*

Corollary 3. *Let e^+ be a positive instance. If C is a parameter-partitioned constraint, $s_{e^+} \in required(T)$.*

This corollary is a first way to improve Algorithm 1. It allows a faster construction of $required(T)$ since it is not necessary to check whether s_{e^+} is the only combination allowing e^+ (line 1 in Algorithm 1).

Secondly, as far as we know, existing parameterized constraints are defined by sets of possible values for their parameters taken separately, which is less expressive than considering directly any subset of $\mathbb{Z}^{|\Delta|}$. We will note $T[p_i]$ for the values of a parameter we wish to learn. Following this, $required(T[p_i]) = \{k_i \in \mathbb{Z}$ such that $\exists s \in required(T), s[p_i] = k_i\}$, namely the set of values for p_i that are required in $T[p_i]$. Similarly, $poss(T[p_i]) = \{k_i \in \mathbb{Z}$ such that $\exists s \in poss(T), s[p_i] = k_i\}$.

If the possible values for a parameter p_i are represented by a **set of integers**, the learning algorithm will use sets of integers to represent $required(T[p_i])$ and $poss(T[p_i])$. The parameterized constraint will be called with $poss(T[p_1]) \times \ldots \times poss(T[p_{|\Delta|}])$, which cannot be any subset of $\mathbb{Z}^{|\Delta|}$. Alternatively, the possible values for a parameter p_i can be represented by a **range of integers**. In this last case, the learning algorithm will use ranges of the form $[min(poss(T[p_i]))..max(poss(T[p_i]))]$ to represent $poss(T[p_i])$ (idem for $required(T[p_i])$). This case restricts even more the possibilities for the combinations of parameters allowed. The last possible case consists of using a simple value for each parameter. However in the context of implied constraints, the two interesting cases are sets and ranges. The general properties presented in Section 3.1 can be rewritten to fit the cases where we deal with these special types of representation of the parameters.

Sets of integers The corollaries 1 and 2 can be rewritten as property 3 and corollary 4.

Property 3. Let e^+ be a positive instance of $\mathcal{P}$ and C a parameterized constraint such that parameters are sets of integers. If there is a unique $s \in \mathbb{Z}^{|\Delta|}$ such that $e^+[var(C)] \in C(s)$ then for any $p_i \in \Delta$, $s[p_i] \in required(T[p_i])$.

Proof. From Corollary 1. □

Corollary 4. *Let $\mathcal{P}$ be a problem, C a parameterized constraint where parameters are represented by set of integers, and $poss(T)$ a set such that $C(poss(T))$ is implied on $\mathcal{P}$. Let p be a parameter of C, v a value in $poss(T[p])$, and $S = \{s \in poss(T)$ such that $s[p] = v\}$. If $\mathcal{P}$ augmented with the constraint $C(S)$ has no solution then v can be removed from $poss(T[p])$.*

Like in Algorithm 1, positive and negative instances will be used to determine which values $v \in poss(T[p])$ should be checked. When receiving a positive instance, Property 3 tells us when a value must go in $required(T[p])$. For each negative instance e^-, if the constraint is *parameter-partitioned*, a heuristic can be to apply corollary 4 with $v = s_{e^-}[p]$.

Ranges of integers Since ranges are a special case of sets, with no hole in their domain, the only possibility for restricting the sets $poss(T[p_i])$ and $required(T[p_i])$ of a parameter p_i is to shrink their bounds.

Property 4. Let e^+ be a positive instance of $\mathcal{P}$ and C a parameterized constraint such that parameters are ranges. If there is a unique $s \in \mathbb{Z}^{|\Delta|}$ such that $e^+[var(C)] \in C(s)$ then for any $p_i \in \Delta$, $min(required(T[p_i])) \leq s[p_i] \leq max(required(T[p_i]))$.

Proof. From Property 3. $\square$

Corollary 5. *Let $\mathcal{P}$ be a problem, C a parameterized constraint where parameters are represented by ranges, and $poss(T)$ a set such that $C(poss(T))$ is implied on $\mathcal{P}$.
Let p be a parameter of C, v a value in $poss(T[p])$, and $S = \{s \in poss(T)$ such that $s[p] \leq v\}$ (respectively: $S = \{s \in poss(T)$ such that $s[p] \geq v\}$).
If $\mathcal{P}$ augmented with the constraint $C(S)$ has no solution then $min(poss(T[p])) > v$ (respectively: $max(poss(T[p_i])) < v$).*

We will see in section 5 the specialization of Algorithm 1 involving Property 4 and Corollaries 3 and 5.

4 Constraint Problems Used in the Learning Phase

Section 3 presented a learning technique based on exploitation of positive and negative instances. If the current model is not good, obviously it should not provide easily (positive) instances. This section discusses what is the set of variables to use and what are the constraints to consider in the learning process.

4.1 Split the problem

It is probably not a good idea to learn a constraint over all the variables of the problem. It is more promising to *split* the initial problem into sub-problems holding only a subset of the variables. Sub-problems have more chances to be easily solvable, and to provide quickly instances useful to learn a constraint which is related to the variables of the sub-problem. By definition, if the learned constraint is valid in the sub-problem then it is still valid in the main problem: variables and constraints of the sub-problem are a subset of the initial one (we consider for the learning process the constraints involving only variables of the sub-problem). Generally, the smaller are the sub-problems, the easier is the learning process. But the learned constraint is less global (so it will prune less). For problems depending directly on a given size n, a derived way to proceed could be to learn implied constraints for small values of n and to deduce from them the corresponding implied constraints for problems of greater size. However in this case there is no formal proof of the validity of the new added constraint in the problem of greater size. Therefore such a process is not "automatic" and the implied constraint will have to be added by hand in the problem of greater size.

4.2 Relax the problem

One may learn implied constraints on relaxed versions of the initial problem. This case is close to the first one. We could have said that we split the problem with respect to constraints instead of with respect to variables. Furthermore we can also consider relaxed versions of sub-problems. For instance the `Gcc` constraint presented in the example of Section 2 is only related to variables representing tasks, and the restriction has nothing to do with resource constraints. We will see in Section 5 that we can learn such a constraint by considering a problem where resource constraints are ignored.

4.3 Optimization problems

In the context of a Branch and Bound algorithm, a way to proceed is to learn new implied constraints at each step of the optimization. They will by definition remain valid for better objective values. Indeed, when we try to find a solution with a better objective value, we consider the same problem augmented with an additional constraint on the objective. The incremental aspect is interesting because it may be easy to find a solution with a poor objective value. If the learned constraint improves the solving process in a significant way, we may observe a nice cooperation between the learning process and the solving process.

5 Experiments

This section deals with the `Gcc` constraint (see Definition 1 in Section 2). In our experiments, Δ is a set of *ranges*. This choice is motivated by two important facts:

1. Quimper et al. [10] proved that the general case is NP-Complete,
2. Powerful algorithms exist when parameters are ranges [13, 11, 12].

Therefore the case which makes sense is definitively ranges.

Notation 2 *Let e be an instance. $occ(v, e)$ denotes the number of times value v appears in e (i.e., the number of variables taking that value).*

The Gcc constraint is *parameter-partitioned* (see subsection 3.2). Given an instance e, the unique element accepting it is: $s_e = (occ(v_1, e), \ldots, occ(v_{|\Delta|}, e))$. Therefore, we specialized Algorithm 1 by applying Property 4 and Corollaries 3 and 5 presented in section 3.2.

Algorithm 2: Algorithm for *Parameter-partitioned* Constraints with **ranges**.

Input: C, Δ, $E = \{e_1, \ldots, e_k\}$
Output: $poss(T)$
foreach $p_i \in \Delta$ **do**
  $min(poss(T[p_i])) \leftarrow -\infty; max(poss(T[p_i])) \leftarrow +\infty$;
  $min(required(T)[p_i]) \leftarrow +\infty; max(required(T)[p_i]) \leftarrow -\infty$;
while $(E \neq \emptyset)$ ***and*** $(required(T) \neq poss(T))$ **do**
  Pick $e \in E$;
  if e *is positive* **then**
    foreach $p_i \in \Delta$ **do**
      if $s_e[p_i] < min(required(T)[p_i])$ **then** $min(required(T)[p_i]) \leftarrow s_e[p_i]$;
      if $s_e[p_i] > max(required(T)[p_i])$ **then**
      $max(required(T)[p_i]) \leftarrow s_e[p_i]$;
  else
    foreach $p_i \in \Delta$ **do**
      if $min(poss(T[p_i])) < s_e[p_i] < min(required(T)[p_i])$ **then**
        $S \leftarrow \{s \in poss(T)$ such that $s[p_i] < s_e[p_i]\}$;
1   **if** $Sol(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup C(S)) = \emptyset$ **then** $min(poss(T[p_i])) \leftarrow s_e[p_i] + 1$;
        else put a solution as positive instance in E
      if $max(poss(T[p_i])) > s_e[p_i] > max(required(T)[p_i])$ **then**
        $S \leftarrow \{s \in poss(T)$ such that $s[p_i] > s_e[p_i]\}$;
2   **if** $Sol(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup C(S)) = \emptyset$ **then** $max(poss(T[p_i])) \leftarrow s_e[p_i] - 1$;
        else put a solution as positive instance in E

The implementation was performed using the Choco constraints solver [2].

5.1 Speeding Up the Learning Phase

It is important to make the learning process as fast as possible and as effective as possible to increase its usability. In this subsection, we propose two simple techniques that permit some improvements.

Time Limit To avoid too long runs in the refutation steps, one can put a cutoff to limit the time spent in each CSP solved on the fly when dealing with negative instances (lines 1 and 2). If the solver has not determined if there is a solution or not with a given bound before the cutoff, the search stops and nothing is updated. This feature avoids the risk to lose a lot of time just for studying a few number of critical occurrences of values.[2]

[2] When using a time limit we no longer have the guarantee to learn exactly the same implied constraint if we change the order in which instances are taken into account.

Apply local consistency instead of solve Reducing the time spent in a run can be done in another way. Instead of running completely the solver until satisfaction or inconsistency, we can apply a relaxed form of resolution by just applying some level of local consistency. This way, when a failure is detected, we are guaranteed that the refutation cause we tried is valid (since local inconsistency implies inconsistency) while we cannot deduce anything from a successful local consistency enforcement. Let us go a bit more in the details of the kind of local consistency that can suit our learning process. The first idea is to use arc consistency (AC) of the CSP instead of complete solving. Unfortunately, in practice, AC is often not strong enough to detect inconsistency. We then have to look for stronger local consistencies. But, state of the art solvers do not provide local consistencies stronger than AC. We propose to use local consistencies inspired from singleton arc consistency (SAC) because they are defined only in terms of arc consistency on particular subproblems. (See [5].) Hence, they can directly be applied in any solver.

Definition 7 (SAC). *A constraint network* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ *is Singleton Arc Consistent iff:* $\forall X_i \in \mathcal{X}, \forall v_i \in D(X_i)$*, arc consistency doesn't fail on* $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \{X_i = v_i\})$*.*

We can easily define k-singleton arc consistency (k-SAC), a local consistency even stronger than *SAC*.

Definition 8 (k-SAC). *A constraint network* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ *is k-SAC iff:* $\forall \{X_{i_1}, \ldots, X_{i_k}\} \subseteq \mathcal{X}, \forall v_{i_1} \in D(X_{i_1}), \ldots, v_{i_k} \in D(X_{i_k})$*, arc consistency does not fail on* $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \{X_{i_1} = v_{i_1}, \ldots, X_{i_k} = v_{i_k}\})$*.*

Finally, we introduce a relaxation of k-SAC, which will be particularly well-suited to check refutation of bounds of `Gcc` constraints in a problem.

Definition 9 (k-SAC(v)). *A constraint network* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ *is k-SAC(v) iff:* $\forall X_i \in \mathcal{X}, v \in D(X_i), \exists \{X_{j_1} \cdots X_{j_{k-1}}\} \subseteq \mathcal{X}$ *such that arc consistency does not fail on* $(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup \{(X_{j_1} = v), \cdots, (X_{j_{k-1}} = v), (X_i = v)\}$

A practical exploitation of this consistency is presented in subsection 5.3.

5.2 Satisfaction Problem

This subsection presents experiments on the problem of Section 2. For all these tests, the learning algorithm was run on the problem without the constraints related to resources. Its simplicity leads to a learning part of only a few milliseconds when m=4 or m=5. Among the improvements of subsection 5.1 we implemented the time limit. We used 15 instances to learn each `Gcc`, and we fixed the time limit to 0.4 seconds. Table 3 shows the results obtained with exactly the problem of Section 2, with a model augmented with the learned `Gcc`. The first time number corresponds to the learning process and the second one to the solving time.

m_1/m_2	$maxi$	#nodes	time (sec.)
4/0	4	47	0.297 + 0.026
4/0	3	47	0.279 + 0.018
4/0	2	23	0.265 + 0.015
3/1	4	43	0.312 + 0.031
3/1	3	30	0.286 + 0.019
3/1	2	3 (no sol)	0.279 + 0.013

Table 3. Problems with $m = 4$ when using the learned `Gcc`.

The results in table 3 are almost the same as results of Table 2, proving that the learning algorithm was able to give us the robustness we needed in the solving process. Tables 4 and 5 show the results obtained with a number of random precedence constraints equal to 4 and $m = d = 5$. Again we stopped the search when it exceeded 60 seconds.

m_1/m_2	$maxi$	#nodes	time (sec.)
5/0	5	–	> 60
5/0	4	–	> 60
5/0	3	–	> 60
5/0	2	53,250 (no sol)	19
3/2	5	–	> 60
3/2	4	–	> 60
3/2	3	1,790 (no sol)	1.2
1/4	5	11,065	3.9
1/4	4	7,985 (no sol)	2.9

Table 4. Problems with $m = 5$ without using the learned `Gcc`.

We were able to solve any problem with size $m = 6$ in at most a few seconds (including the learning time) while most of them are not solvable without the implied `Gcc` even after several minutes. Our goal with that first bench was to take a naive model, with a naive search strategy, and to experiment how much our technique can improve it in terms of robustness. This bench is well suited to the use of implied `Gcc`'s. However our point here is that the user just needs to have the intuition "there is maybe some hidden `Gcc` related to the ordering of tasks", and then run the learning algorithm. The human effort is thus much lower than studying by hand the possible implied cardinality constraints according to each combination of parameters m, d, and precedence constraints.

5.3 Optimization Problem

At the Computer Science department of the Institute of Technology of University of Montpellier (IUT), students have to perform an individual project. They choose their

m_1/m_2	$maxi$	#nodes	time (sec.)
5/0	5	74	0.729 + 0.030
5/0	4	74	0.730 + 0.025
5/0	3	63	0.726 + 0.024
5/0	2	361 (no sol)	0.721 + 0.243
3/2	5	63	0.734 + 0.025
3/2	4	92	0.717 + 0.059
3/2	3	123 (no sol)	0.738 + 0.122
1/4	5	39	0.740 + 0.023
1/4	4	154 (no sol)	0.718 + 0.133

Table 5. Problems with $m = 5$ when using the learned `Gcc`.

project from a set of subjects proposed by their teachers. Students have to provide a list $(p_1, \ldots, p_m)$ of the m projects they prefer. Their preferences form a total order: project p_i is strictly preferred to project p_{i+1}. Since most of the students make generally very similar choices, it may happen that no solution exists in which all the students obtain a project of their list.[3] The goal of the "IUT problem" is thus to assign projects to students, such that two students do not share a same project, while maximizing the satisfaction of the students. A student is very satisfied if she obtains her first choice, less if she obtains the second one, and so on... until obtaining no project of her list, which represents her worst possible satisfaction. Until last year, teachers were doing assignments by hand, spending many hours to find a far from optimal solution. Two years ago, one of the proposed projects was to solve this affectation problem using constraint programming. The students who have worked on this project provided the following model : students are variables, and their domains are the set of choices made by the student plus the extra-value "0", meaning "no project assigned at this round". Constraints are set between each pair (X_i, X_j) of students to express "$(X_i \neq X_j) \vee (X_i = X_j = 0)$". Some preference-variables measure the satisfaction of each student. They are linked to the student-variables by constraints defined in extension $\{(p_1, m), \ldots, (p_m, 1), (0, 0)\}$. The objective is to maximize the sum of preference-variables. In the following, this model will be referred as the *initial model*. The solver is first launched on the initial model for a short period of time (1 sec.) in order to produce a first solution S_0 (the best found within these 1 seconds) and an upper bound to the quality of solutions. Before starting the learning phase, we add to the initial model a constraint accepting only instantiations with objective value at least as great as the one of S_0. Then the learning process is performed on a set E of instances, containing S_0 (labelled positive), and randomly generated instances (which are almost always negative in practice). We did not use the time limit improvement but the *k-SAC(v)* consistency (described in Section 5.1) instead of complete solves. At the end of the learning phase, we launch the solver on the problem augmented with the learned implied constraint.

[3] In this case, remaining students make new choices in unassigned projects and a new affectation is computed on this remaining problem.

Since the real size of the problem is too huge to solve it several times, in the following we perform tests on smaller problems stem from the real data of last year. Notice that the way these problems are generated preserves the ratio "number of projects/number of students" , and the particular distribution of the choices. The gain obtained on the real problem is given at the end of that section.

Table 6(a) summarizes the average time and number of explored nodes of the initial model according to the number of students n involved in the problem.

n	#nodes	time (sec.)
10	110	0.2
15	2,648	7.6
20	137,982	183.2

(a) Initial Model.

n	$\|E\|$	#nodes	time (sec.)
10	10	12	0.2 + 0.0
15	10	57	2.6 + 1.1
15	20	50	3.8 + 1.1
15	30	50	5.1 + 0.9
15	40	50	5.9 + 0.9
20	10	4,801	5.7 + 14.9
20	20	2,800	6.8 + 13.0
20	30	2,643	7.9 + 8.4
20	40	1,998	9.1 + 8.4

(b) Reformulated Model.

Table 6. Results on the *IUT* problem

Table 6(b) shows the results obtained on the same data as Table 6(a) when learning the `Gcc` on the student variables according to the number of students n involved in the problem and the number of instances $|E|$ used during the learning phase. The learning time includes the time devoted to the generation of the initial positive example.

The first observation is that, thanks to relaxation to the local consistency tests instead of complete solves, the learning process requires very reasonable time (9.1 seconds in the worst case). Secondly, we observe that the more we learn instances, the more precise the implied constraint we learn is, and then the more compact the search tree of the improved model is. On this problem it is always beneficial to improve the model before trying to solve it, since the sum of the learning time and the new model solving time is lower than the time required to solve directly the initial model.[4] That gain represents only few seconds for $n = 15$, but reaches a factor 10 when the problem size grows to $n = 20$.

On the real problem ($n = 60$), this approach allows to solve the problem in 43 minutes (including 90 seconds for the learning phase on 50 instances) while the solving of the initial model has been stopped after 12 hours. Thanks to the control of the cost of the learning process, this approach does not need any particular tuning. It is robust to the number of instances selected to be learned.

[4] for $n = 10$, the learning phase is not launched since the optimal solution has been found during the initial solution search time.

6 Conclusion

This paper presented a new generic framework for learning implied constraints from allowed and forbidden assignments of values to sets of variables. The objective was to improve automatically constraint models. To validate the impact of our work, we applied our technique to the case of implied global cardinality constraints. We proposed a few simple techniques that permit to speed up our algorithms. We validated the interest of our approach by two series of experiments.

References

1. N. Beldiceanu. Global constraints as graph properties on a structured network of elementary constraints of the same type. *Proceedings CP*, pages 52–66, 2000.
2. Choco. A Java library for constraint satisfaction problems, constraint programming and explanation-based constraint solving. *URL: http://sourceforge.net/projects/choco*, 2005.
3. R. Coletta, C. Bessiere, B. O'Sullivan, E.C. Freuder, S. O'Connell, and J. Quinqueton. Semi-automatic modeling by constraint acquisition. *Proceedings CP*, pages 812–816, 2003.
4. S. Colton and I. Miguel. Constraint generation via automated theory formation. *Proceedings CP*, pages 575–579, 2001.
5. R. Debruyne and C. Bessière. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.
6. A. M. Firsch, C. Jefferson, B. Martinez Hernandez, and Ian Miguel. The rules of constraint modelling. *Proceedings IJCAI*, to appear, 2005.
7. Ilog. Solver 5.0 User's Manual. *URL: http://www.ilog.fr*, 2000.
8. A. Legtchenko, A. Lallouet, and A. Ed-Dbali. Intermediate consistencies by delaying expensive propagators. *Procedings FLAIRS*, 2004.
9. J.F. Puget. Constraint programming next challenge: Simplicity of use. In *Proceedings CP*, pages 5–8, Toronto, Canada, 2004.
10. C.-G. Quimper. Enforcing domain consistency on the extended global cardinality constraint is np-hard. Technical Report CS-2003-39, School of Computer Science, University of Waterloo, 2003.
11. C.-G. Quimper, P. Van Beek, A. Lopez-Ortiz, A. Golynski, and S. B. Sadjad. An efficient bounds consistency algorithm for the global cardinality constraint. *Proceedings CP*, pages 600–614, 2003.
12. C.-G. Quimper, A. López-Ortiz, P. van Beek, and A. Golynski. Improved algorithms for the *global cardinality* constraint. *Proceedings CP*, 3258:542–556, 2004.
13. J-C. Régin. Generalized arc consistency for global cardinality constraint. *Proceedings AAAI*, pages 209–215, 1996.
14. J-C. Régin. Minimization of the number of breaks in sports scheduling problems using constraints programming. *DIMACS series in Discrete Mathematics and Theoretical Computer Science*, 57:115–130, 2001.
15. B. Smith, K. Stergiou, and T. Walsh. Modelling the golomb ruler problem. In J.C. Régin and W. Nuijten, editors, *Proceedings IJCAI'99 workshop on non-binary constraints*, Stockholm, Sweden, 1999.