

Using the Global Constraint Seeker for Learning Structured Constraint Models: a First Attempt

Nicolas Beldiceanu¹ and Helmut Simonis^{2*}

¹ TASC team (INRIA/CNRS), Mines de Nantes, France
Nicolas.Beldiceanu@mines-nantes.fr

² Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
h.simonis@4c.ucc.ie

Abstract. Considering problems that have a strong internal structure, this paper shows how to generate constraint models from a set of positive, flat samples (i.e., solutions) without knowing a priori neither the constraint candidates, nor the way variables are shared within constraints. We describe two key contributions to building such a model generator: (1) First, learning is modeled as a bi-criteria optimization problem over ranked constraint candidates returned by the Constraint Seeker, where we optimize both the compactness of the model, and the rank (or appropriateness) of the selected constraints. (2) Second, filtering out irrelevant candidate models is achieved by using meta data of the global constraint catalog that describe links between constraints. Some initial experiments on a proof-of-concept implementation show promising results.

1 Scope and Hypothesis

Global constraints were initially introduced [3] in order to more efficiently handle the filtering associated with some recurring structured constraint networks [1]. An inherent disadvantage of the approach is that the introduction of global constraints does not make using constraint programming any easier, since the growing number of global constraints presents confusing choices to most users. Based on this recognized concern about ease of use of constraint programming [19], this paper shows how global constraints can be, in the context of structured problems, an essential component to automatically learning models from example solutions. More precisely, this paper presents the sketch of a generic approach, as well as a proof of concept, for automatically extracting constraint models from a set of positive, flat samples (i.e., solutions provided as a flat list of integers), that relies both on global constraints and constraint programming. This work is done under the following five assumptions:

1. We assume that the samples directly correspond to solutions that one typically finds in standard magazines and/or a standard Operations Research problem compendium for the corresponding problems, i.e., we do not require that samples are solutions of special, reformulated models of the original problem.

* The second author is supported by Science Foundation Ireland (Grant Numbers 05/IN/I886 and 10/IN.1/I3032).

2. Many problems are not defined completely just by a solution, they also involve some kind of additional data or hints, which are required in order to interpret the solution. This is the case both for a number of puzzles, where hints are part of the problem statement, as well as for a number of Operations Research problems, where data (e.g., a cost matrix in optimization problems, or durations and resource use of activities in scheduling problems) are also part of the problem definition. Within this paper, we restrict ourselves to problems where, beside the positive samples, no extra hints are provided.
3. We assume that all positive samples are *correct* (i.e., there is no noise in the sample data).
4. All samples have the same size, i.e. we do not have to generalize the model found for arbitrary problem sizes.
5. We assume that we are looking at problems that have a *strong internal structure*, i.e., they can be represented in a very compact way. This is in fact the case for most problems considered by Constraint Programming, but is equally true for problem class repositories like Garey and Johnson [13].

We now discuss the relative importance of these five hypotheses:

1. In principle we could easily eliminate the first hypothesis by providing a limited number of standard transformations encoding common reformulations, e.g., transformation from 0/1 variables to non 0/1 variables [14]. We choose not to consider such reformulations in our first prototype.
2. Integrating hints within the samples could be done by coming up with a general way to describe hints. This would require some research but seems feasible for some classes of hints. In our current prototype we can already provide hints on the way variables are grouped together. This is required for some problems where the way variables are grouped in some constraints is not regular at all, and is therefore explicitly provided as part of the original problem description. This is for instance the case for some Jigsaw Sudoku where 3 by 3 blocks are replaced by ad hoc shapes.
3. It should be much harder to remove the third hypothesis about the correctness of samples. If samples are almost correct, e.g., only a very limited number of integer values are wrong, an idea worth investigating would be to use soft global constraints together with a very restricted violation cost [18]. It is unclear at this stage how much this would increase the number of candidate models, this line of work is not the focus of our current research.
4. We will investigate how to deal with samples of different sizes in future work.
5. As we will describe later in the paper, our approach for searching models very much relies on this last hypothesis, that is on the assumption that the problem has a strong internal structure, i.e., its constraint generator can be described in a compact way.

Our approach takes advantage of hypotheses 3. and 5., it uses the fact that our samples are reliable and that we restrict ourselves to structured problems. It also relies on the following key ingredients:

- First, it tries to express models as *a limited number of conjunctions of similar global constraints*. It uses the knowledge base describing various properties of global con-

straints provided by the global constraint catalog [2] in order to come up with constraints that are not only valid for the given samples, but also make sense for a human modeler. Considering *conjunction of similar global constraints* is a key point of our approach since:

- It seems to be the right degree of abstraction for expressing the space of learning hypothesis since it allows *describing very concisely models of structured problems*.
 - It follows one of key inductive criteria, namely the *minimum description length principle* that states that the best model wrt. a set of samples is the model that minimizes the sum of the size of the model plus the size of the samples when described by the model.
 - Conjunction of similar global constraints seems also comprehensible when it come to the point to provide an output that *must be intelligible to the user*, which may eventually modify or extend it (e.g., add symmetry breaking constraints or implied constraints).
- Second, it *relies on the global Constraint Seeker functionality* [4] for retrieving and ranking relevant candidate constraints that can match a given combination of parameters obtained from the positive samples.
- Third, it addresses the learning problem of a conjunction of constraints as a *bi-criteria optimization constraint search problem*, where a conjunction of constraints can both be represented in a *very compact way*, and consists of constraints that are *highly ranked* by the Constraint Seeker.

Section 2 provides an overview of the different components of our method. Section 3 explains how we propose alternative ways of combining variables that will be passed as constraint arguments. Section 4 describes the bi-criteria optimisation problem we solve in order find conjunctions of relevant constraints that have a very regular structure. Section 5 shows how to select relevant conjunctions for our model from the different conjunctions of constraint candidates found in Section 4. Section 6 will evaluate our method on some initial example problems, while Section 7 looks at related work.

2 Overview of the Learning Algorithm

The learning algorithm is decomposed into the following successive steps:

1. Given $\mathcal{V} = v_1, v_2, \dots, v_s$ variables, where s is the size of the samples, a *groups of variables generator* generates ordered sequences of variables of \mathcal{V} on which we will search for constraints. The aim of this generator is to systematically propose different ways of grouping variables together, which can be both described concisely and which matches the pattern found in typical constraint models. This first step will be described in Section 3.
2. The *instance generator* takes as input the samples as well as the ordered sequences of variables generated by the groups of variables generator. From this input it generates the ground (fully instantiated) parameters that will be passed to the Constraint Seeker [4] in order to retrieve the corresponding matching constraints. Since this part is quite straightforward it will not be detailed later on.

3. For each sequence of variables the *candidate generator* takes the corresponding ground parameters build by the instance generator and calls the Constraint Seeker to find relevant constraint candidates. The details of this operation are described in [4].
4. Once the candidate generator has generated a set of candidate constraints for each element of the ordered collection of sequences of variables, we call the *relevance optimizer* on each such collection. Its purpose is to find out for each ordered collection of sequences of variables one or several conjunctions of constraints that consist of both highly relevant and concisely described, structured sets of constraints. This step is done by solving a bi-criteria constraint optimization problem and will be described in Section 4.
5. Given a set of conjunctions of constraints found by the relevance optimizer, the *dominance checker* discards conjunctions that are dominated by other conjunctions. The idea here is to prefer reporting stronger conjunctions of constraints rather than weaker one, in the sense that we give preference to conjunctions that admit fewer solutions. This last step will be described in Section 5.

3 Grouping Variables Together for Generating Constraint Arguments

Given a sample of length n , a first question to address is how to group together the variables that will be passed to the global constraints of the conjunction. Of course groups of variables can be build in many different ways, but relying on our assumption about structured problems, we focus our attention to a limited number¹ of structured sequences of variables. Here, *structured sequence of variables* means sequences of variables for which the generator can be described in a very compact form.

Definition 1. (ordered sequences of variables) Given a sequence of variables S , an ordered collection of sequences of variables of S consist of sequences s_1, s_2, \dots, s_p such that:

- each sequence consists of a sequence of distinct variables of S ,
- no pair of sequences involve exactly the same set of variables,
- within a given sequence, variable indices are increasing.

Very often it turns out that an ordered collection of sequences of variables corresponds to a partition of the variables of S where all sequences in the partition involve the same number of variables. As we will see later on, the order of the sequences of variables matters since it reflects the order of the positions in a sample (since we assume that samples are given w.r.t. some natural order of the original problem).

Given n , the size of the sample, we now provide a list of common generators of ordered sequences of variables, where for each generator we give its parameters² as well as a short description:

¹ In our context, *limited number* means limited for a computer, that is less than a few thousands candidates.

² Parameters are assumed to be natural numbers.

- **matrix partition generator** $(m_1, m_2, size_1, size_2)$ where $n = m_1 \cdot m_2$, $size_1 + size_2 > 2$, and $\forall i \in [1, 2] : size_i \in [1, m_i - 1] \wedge m_i \bmod size_i = 0$, is a generator which decomposes the sample into a m_1 columns by m_2 rows matrix. Each row (respectively column) of that matrix is partitioned into blocks of size $size_1$ (respectively $size_2$). As an example, consider the sample v_1, v_2, \dots, v_{12} to which we apply the matrix partition generator matrix partition(6, 2, 3, 1). We obtain the ordered sequences of variables $\langle v_1, v_2, v_3 \rangle, \langle v_4, v_5, v_6 \rangle, \langle v_7, v_8, v_9 \rangle, \langle v_{10}, v_{11}, v_{12} \rangle$. This kind of generator is useful in the context of problems which are naturally expressed as matrix models, e.g., *Latin Squares*, *Orthogonal Latin Squares*, *Sudoku*, *sport scheduling* and *timetabling*. Similarly the 3-d matrix partition generator³ parametrized by $(m_1, m_2, m_3, size_1, size_2, size_3)$ where $n = \prod_{i=1}^3 m_i$, $\sum_{i=1}^3 size_i > 3$, and $\forall i \in [1, 3] : size_i \in [1, m_i - 1] \wedge m_i \bmod size_i = 0$, decomposes the sample into a m_1 by m_2 by m_3 matrix where each dimension is partitioned into blocks of size $size_1$, $size_2$ and $size_3$.
- **diagonal** which can be applied if $n = m^2$, and which extracts the two main diagonals of the m by m matrix associated with the sample. This kind of generator is useful both for magic squares and Sudoku X.
- **modulo partition generator** (d) where $d \leq \lfloor \sqrt{n} \rfloor$ is a generator which decomposes the sample into d sequences, corresponding respectively to the variables for which the remainder of the indices divided by d are all equal to a same value. As an example, consider the sample v_1, v_2, \dots, v_{12} to which we apply the modulo partition generator modulo(3). We obtain the ordered sequences of variables $\langle v_1, v_4, v_7, v_{10} \rangle, \langle v_2, v_5, v_8, v_{11} \rangle, \langle v_3, v_6, v_9, v_{12} \rangle$. This kind of generator is useful in the context where constraints relate variables that are located at a fixed distance from each other, e.g., all interval series.
- **block partition generator** $(size_1, size_2)$ where $size_1 > 0$, $size_2 > 0$, $n \bmod (size_1 + size_2) = 0$, and $size_1 + size_2 \leq 2 \cdot \lfloor \sqrt{n} \rfloor$ is a generator which successively creates sequences of sizes $size_1, size_2, size_1, size_2, \dots, size_1, size_2$. As an example, consider the sample v_1, v_2, \dots, v_{12} to which we apply the generator block_partition(4, 2). We obtain the ordered sequences of variables $\langle v_1, v_2, v_3, v_4 \rangle, \langle v_5, v_6 \rangle, \langle v_7, v_8, v_9, v_{10} \rangle, \langle v_{11}, v_{12} \rangle$. This kind of generator is useful in the context of *cyclic timetabling* problems where, for instance a first constraint applies for each working day (i.e., Monday to Friday), and a second constraint applies for the weekend (i.e., $size_1 = 5$ and $size_2 = 2$).
- **sliding window generator** $(size, d)$ where $size \in [1, n - 1]$, $d \in [1, size - 1]$, and $(n - size) \bmod d = 0$ is a generator which creates all sequences of consecutive variables of length $size$ such that the index minus one of the first variable of each sequence is divisible by d . As an example, consider the sample v_1, v_2, \dots, v_{12} to which we apply the generator sliding_window(6, 3). We obtain the ordered sequences of variables $\langle v_1, v_2, v_3, v_4, v_5, v_6 \rangle, \langle v_4, v_5, v_6, v_7, v_8, v_9 \rangle, \langle v_7, v_8, v_9, v_{10}, v_{11}, v_{12} \rangle$. These sliding windows are often encountered in timetabling problems.
- **triangular difference table generator** is a generator which creates $n - 1$ sequences, where the i^{th} sequence corresponds to the differences $v_{1+i} - v_1, v_{2+i} -$

³ This generator will be added later on in the system.

$v_2, \dots, v_n - v_{n-i}$. As an example, consider the sample v_1, v_2, \dots, v_5 to which we apply the triangular difference table generator. We obtain the ordered sequences $\langle v_2 - v_1, v_3 - v_2, v_4 - v_3, v_5 - v_4 \rangle$, $\langle v_3 - v_1, v_4 - v_2, v_5 - v_3 \rangle$, $\langle v_4 - v_1, v_5 - v_2 \rangle$, and $\langle v_5 - v_1 \rangle$. This kind of generator is useful in the context of problems like Costas Array.

After applying the generators on our positive samples we get sets of potential candidate sequences of different ways variables may occur in the constraints. For each such sequence we can try to find matching constraints. This is the topic of the next section.

4 Learning a Conjunction of Constraints as a Bi-criteria Optimization Problem

Given an ordered collection of sequences of variables $\mathcal{S} = vars_1, vars_2, \dots, vars_p$, obtained by one of the generators described in Section 3, we use the Constraint Seeker to associate a constraint ctr_i ($1 \leq i \leq p$) to each sequence of variables $vars_i$, so that the conjunction $ctr_1(vars_1) \wedge ctr_2(vars_2) \wedge \dots \wedge ctr_p(vars_p)$ holds for all given samples. As potential constraint candidates we consider those constraints of the global constraint catalog for which an evaluator is currently available, i.e. 70% of the constraints. Trying to learn global constraints rather than binary constraints introduces a bias, focusing our attention on structured problems. Since this is our basic hypothesis, this allows us to drastically reduce the number of potential constraint networks we have to consider. However, since there may still exist a lot of solutions, this section also shows how to measure the relevance of a conjunction of constraints as a candidate for our model. The relevance measures both the *compactness of the conjunction of constraints* found, as well as the *importance of the constraints* themselves. We now describe how to encode this problem as a bi-criteria constraint optimization problem.

- First, we associate to each sequence of variables $vars_i$ ($1 \leq i \leq p$), a variable C_i whose values $ctr_{i,1}, ctr_{i,2}, \dots, ctr_{i,q_i}$ correspond to constraints (and their additional parameters) that match all positive samples. For each $i \in [1, p]$ the possible set of candidate constraints is obtained by a single call to the Constraint Seeker, where we pass the projection of all positive samples on the corresponding variables $vars_i$. For each candidate constraint $ctr_{i,j}$ ($1 \leq i \leq p, 1 \leq j \leq q_i$) the seeker also returns its rank $rank_{i,j}$, where smaller rank indicates more relevant constraints [4].
- Second, we also associate to each sequence of variables $vars_i$ ($1 \leq i \leq p$), a variable R_i giving the rank of the constraint assigned to variable C_i . An *element*($C_i, \langle rank_{i,1}, rank_{i,2}, \dots, rank_{i,q_i} \rangle, R_i$) constraint links variables C_i and R_i . The *ranking cost RankCost* associated with constraints C_1, C_2, \dots, C_p is equal to the sum of the corresponding ranking variables, that is $R_1 + R_2 + \dots + R_p$. This is the first criteria we try to minimize.
- Third, we also associate to the constraints variables C_1, C_2, \dots, C_p a *compactness cost SizeCost* which is the minimum of the following two quantities:
 - The *number of changes* in the sequence C_1, C_2, \dots, C_p , i.e., the number of times $C_i \neq C_{i+1}$ ($1 \leq i < p$) holds.

- The *smallest period* of the sequence C_1, C_2, \dots, C_p , i.e., the smallest number e such that $C_i = C_{i+e}$ for all i in $[1, p - e]$.

The goal of *SizeCost* is to measure the regularity of the sequence of constraints C_1, C_2, \dots, C_p . This is the second criteria we try to minimize.

Since the ranking and the compactness costs are incomparable, we solve the following two optimizations problems:

- A first problem where we first minimize the compactness cost, and then the ranking cost.
- A second problem where we first minimize the ranking cost, and then the compactness cost.

We only retain Pareto-optimal solutions to this bi-criteria optimization problem. As a result, we get a set of conjunctions of constraints that can be further reduced by checking for implications between them. The next section discusses this task.

5 Selecting Relevant Models among Pareto Optimal Candidates

Given a set of conjunctions of constraints that are all candidates to be part of the final model, this section shows how to filter out irrelevant candidates. Even if we only keep the Pareto-optimal solutions in Section 4, we can still find a large number of candidates. We want to remove weak explanations which are implied by stronger candidates to reduce the size of our model. Deciding implication between sets of constraints in general is quite difficult, we therefore introduce the simpler notion of domination of a candidate by an other candidate. Intuitively, a candidate conjunction C_1 dominates a candidate conjunction C_2 if conjunction C_1 implies conjunction C_2 due to structural properties of the constraints. As an example, $C_1 = \text{alldifferent}(\langle v_1, v_2, v_3, v_4 \rangle)$ [2, page 410] implies $C_2 = \text{alldifferent}(\langle v_1, v_2 \rangle) \wedge \text{alldifferent}(\langle v_3, v_4 \rangle)$, consequently C_1 dominates C_2 . We will rely on some meta data in the catalog that describe relations between constraints to implement these domination checks efficiently. But we first need to introduce some definitions.

Definition 2. (contractible constraint w.r.t. one of its arguments) A constraint c is contractible w.r.t. one of its arguments that corresponds to a collection of items Arg if, for each ground instance which satisfies c , that instance is still satisfied when we remove any item from Arg .

A typical example of a contractible constraint is the *alldifferent* constraint. In addition, there is a slightly restricted definition of contractibility if we focus on the prefix or the suffix of a collection of items⁴ corresponding to one argument of the constraint. For instance, consider the constraint *among_seq*(*low*, *up*, *seq*, *variables*, *values*) [2, page 476] that enforces all sequences of *seq* consecutive variables of the collection *variables*

⁴ Given a collection of items $\mathcal{I} = \langle I_1, I_2, \dots, I_n \rangle$, a *prefix* of \mathcal{I} is defined by $\mathcal{I} = \langle I_1, I_2, \dots, I_m \rangle$, ($m \leq n$). Similarly, a *suffix* of \mathcal{I} is defined by $\mathcal{I} = \langle I_m, I_{m+1}, \dots, I_n \rangle$, ($m \geq 1$).

to take at least *low* values in *values* and at most *up* values in *values*. *among_seq* is not contractible w.r.t. *variables* since removing a value in the middle of *variables* may create a new sequence for which the constraint does not hold. However, if we remove a prefix or a suffix from *variables*, then the corresponding *among_seq* constraint still holds, since no new sequences are created.

Definition 3. (*p*-contractible and *s*-contractible constraint w.r.t. one of its arguments) A constraint *c* is *p*-contractible (respectively *s*-contractible) w.r.t. one of its arguments that corresponds to a collection of items *Arg* if, for each ground instance which satisfies *c*, that instance is still satisfied when we remove any prefix (respectively suffix) from *Arg*.

The *among_seq* and *pattern* [2, pages 476,1424] constraints are examples of *p*-contractible as well as *s*-contractible constraints, while the *int_value_precede* and the *int_value_precede_chain* [2, pages 1062,1064] constraints are examples of *s*-contractible constraints. Note that the notion of *s*-contractibility was initially introduced by M. J. Maher in the context of open constraints [15], where it was simply called contractibility. We now recall a dual notion, also introduced by M. J. Maher, called extensibility.

Definition 4. (*extensible* constraint w.r.t. one of its arguments) A constraint *c* is *extensible* w.r.t. one of its arguments that corresponds to a collection of items *Arg* if, for each ground instance which satisfies *c*, that instance is still satisfied when we add any ground item to *Arg*.

If we rather just restrict ourself on adding an item in *front* of *Arg* (respectively *after* *Arg*) we say that the constraint is *p*-*extensible* (respectively *s*-*extensible*).

An example of an *extensible* constraint is for instance *atleast*(*n*, *vars*, *val*) [2, page 504], i.e., at least *n* variables of variables *vars* are assigned to value *val*. Here the constraint *atleast*(1, $\langle v_1, v_2 \rangle$, 6) is stronger than the constraint *atleast*(1, $\langle v_1, v_2, v_3, v_4 \rangle$, 6). An example of *s*-*extensible* constraint is the *element* constraint (i.e., we can add items at the end of its table). We are now in position to introduce the notion of domination between two conjunctions of constraints.

Definition 5. (*domination of a conjunction candidate by another conjunction candidate*) A conjunction candidate \mathcal{C}_1 dominates a conjunction candidate \mathcal{C}_2 if for all constraints $ctr_2(vars_2)$ in \mathcal{C}_2 there exists at least one constraint $ctr_1(vars_1)$ in \mathcal{C}_1 such that at least one of the following conditions holds:

1. $(vars_2 = vars_1) \wedge (ctr_1 \Rightarrow ctr_2)$,
2. $(vars_2 \subseteq vars_1) \wedge (ctr_1 \Rightarrow ctr_2) \wedge contractible(ctr_2)$,
3. $(vars_2 \text{ is a prefix of } vars_1) \wedge (ctr_1 \Rightarrow ctr_2) \wedge s\text{-contractible}(ctr_2)$,
4. $(vars_2 \text{ is a suffix of } vars_1) \wedge (ctr_1 \Rightarrow ctr_2) \wedge p\text{-contractible}(ctr_2)$,
5. $(vars_1 \subseteq vars_2) \wedge (ctr_1 \Rightarrow ctr_2) \wedge expendible(ctr_2)$,
6. $(vars_1 \text{ is a prefix of } vars_2) \wedge (ctr_1 \Rightarrow ctr_2) \wedge s\text{-expendible}(ctr_2)$,
7. $(vars_1 \text{ is a suffix of } vars_2) \wedge (ctr_1 \Rightarrow ctr_2) \wedge p\text{-expendible}(ctr_2)$.

As an illustration of Definition 5 consider the conjunction \mathcal{C}_1

$$\left\{ \begin{array}{l} \text{alldifferent_consecutive_values}(\langle v_1, v_2, v_3, v_4 \rangle) \wedge \\ \text{alldifferent_consecutive_values}(\langle v_5, v_6, v_7, v_8 \rangle), \end{array} \right.$$

where *alldifferent_consecutive_values* enforces variables to take consecutive distinct values [2, pages 420], and the conjunction \mathcal{C}_2

$$\left\{ \begin{array}{l} \text{alldifferent}(\langle v_1, v_2 \rangle) \wedge \text{alldifferent}(\langle v_3, v_4 \rangle) \wedge \\ \text{alldifferent}(\langle v_5, v_6 \rangle) \wedge \text{alldifferent}(\langle v_7, v_8 \rangle). \end{array} \right.$$

Conjunction \mathcal{C}_1 dominates conjunction \mathcal{C}_2 since every constraint of \mathcal{C}_2 is implied by at least one constraint of \mathcal{C}_1 . For instance, *alldifferent*($\langle v_1, v_2 \rangle$) is implied by *alldifferent_consecutive_values*($\langle v_1, v_2, v_3, v_4 \rangle$) since:

1. First, $\text{alldifferent_consecutive_values}(\langle v_1, v_2, v_3, v_4 \rangle) \Rightarrow \text{alldifferent}(\langle v_1, v_2, v_3, v_4 \rangle)$. This implication is explicitly stated in the catalog.
2. Second, since *alldifferent*($\langle v_1, v_2, v_3, v_4 \rangle$) is contractible it implies *alldifferent*($\langle v_1, v_2 \rangle$). Again the fact that *alldifferent* is contractible is explicitly given in the catalog.

In the evaluation below, we use some additional dominance rules, which we can not detail here due to lack of space.

6 Evaluation

All experiments were done without providing and kind of hint to the system (e.g., without hints on the way variables are grouped together inside constraints, which means that all combinations of parameters for each generator are tried for generating candidate constraints). With the current generators, finding a model is typically done within a few seconds (i.e., 15 sec.) on a nowadays laptop.⁵

6.1 Magic Squares

We start our evaluation with a small puzzle, the Magic Square problem. In a Magic Square of size n , we find all numbers from 1 to n^2 in the cells of a quadratic matrix, and the sum of each row, each column and both main diagonals is the same and equal to $\frac{\sum_{i=1}^{n^2} i}{n}$. One of the most famous Magic Squares of size 4 was used in the engraving *Melencolia I* by Albrecht Dürer, it is shown in Figure 1. Each cell contains one of the numbers from 1 to 16, with the index indicating the position in the input vector, counted from 1. To use this example as input for our program, we flatten the structure and only keep the integer vector 16, 3, 2, 13, 5, 10, 11, 8, 9, 6, 7, 12, 4, 15, 14, 1 in the given order. Our program finds the constraints shown in Figure 2. We encounter a number of constraints: the *alldifferent_consecutive_values* constraint

⁵ In the future the user may be allowed to explicitly provide such hints, but with the current set of generators this is not required.

Fig. 1: Magic Square of Size 4x4

16 ₁	3 ₂	2 ₃	13 ₄
5 ₅	10 ₆	11 ₇	8 ₈
9 ₉	6 ₁₀	7 ₁₁	12 ₁₂
4 ₁₃	15 ₁₄	14 ₁₅	1 ₁₆

states that the values are pairwise distinct (`alldifferent`), but also range over a consecutive range of numbers (here 1 to 16). The `symmetric_alldifferent` enforces the usual `alldifferent` constraint together with the condition $x_i = j \Leftrightarrow x_j = i$. The `sum_ctr` constraint is the linear equality constraint $\sum x_i = 34$, using the additional parameters `=` (for equality) and 34, as a right-hand side. Finally, `strictly_decreasing` enforces binary `>` constraints between its arguments.

The constraints shown above are the only ones remaining after the dominance check for the given sequence generators and the constraints currently considered by our program. The program found 45 initial candidates, matching the positive example for each sequence element. 28 candidates were removed by the dominance check, for example `alldifferent` constraints on each row and column, dominated by the `alldifferent_consecutive_values` constraint on the complete set of variables. Of the remaining 17 candidates, 8 were removed as trivial, for example `no_peak` constraints on 2 variables, which are trivially satisfied.

Note that the constraints shown do not uniquely identify the given problem instance. We can write a constraint program based on the constraints found and search for solutions. Figure 3 shows all 5 solutions for this program, they nicely generalize the one example we provided as input.

Not all Magic Squares will satisfy the additional constraints shown above, they are only valid for a subset of all 4x4 Magic Squares and are therefore not redundant. This leads to the question on whether we can isolate the original constraints of the problem and how many samples we will need for that task. To answer this question we perform an experiment where we pick a random sample of k entries from the set of all solutions for the 4x4 Magic Square problem, and count how many constraint pattern are detected by our program. Table 1 shows the distribution of results over 100 runs for each sample size. The rows indicate the size of the sample, between 1 and 9. The columns indicate how many runs produced i pattern. The results seem to indicate that for the Magic Square problem even with only three or four samples a rather accurate identification of the core constraints of the problem is possible. But note that the random sample selection does not really match typical human behaviour, humans often have difficulty creating or selecting random problem instances.

6.2 BIBD

As a second example we consider the Balanced Incomplete Block Design (BIBD) generation, which is a standard combinatorial problem, often used as a benchmark problem in Constraint Programming. A BIBD is defined as an arrangement of v distinct objects

Fig. 2: Constraints Found for Magic Square

Partition Generator	Partition	Constraint(s)																
matrix(16,1,16,1)	original sequence of values	1×alldifferent_consecutive_values 1×symmetric_alldifferent, extra parameter [1, 2, ..., 16]																
matrix(4,4,1,4)	<table border="1"> <tr><td>16₁</td><td>3₂</td><td>2₃</td><td>13₄</td></tr> <tr><td>5₅</td><td>10₆</td><td>11₇</td><td>8₈</td></tr> <tr><td>9₉</td><td>6₁₀</td><td>7₁₁</td><td>12₁₂</td></tr> <tr><td>4₁₃</td><td>15₁₄</td><td>14₁₅</td><td>1₁₆</td></tr> </table>	16 ₁	3 ₂	2 ₃	13 ₄	5 ₅	10 ₆	11 ₇	8 ₈	9 ₉	6 ₁₀	7 ₁₁	12 ₁₂	4 ₁₃	15 ₁₄	14 ₁₅	1 ₁₆	4×sum_ctr, extra parameters =, 34
16 ₁	3 ₂	2 ₃	13 ₄															
5 ₅	10 ₆	11 ₇	8 ₈															
9 ₉	6 ₁₀	7 ₁₁	12 ₁₂															
4 ₁₃	15 ₁₄	14 ₁₅	1 ₁₆															
matrix(4,4,4,1)	<table border="1"> <tr><td>16₁</td><td>3₂</td><td>2₃</td><td>13₄</td></tr> <tr><td>5₅</td><td>10₆</td><td>11₇</td><td>8₈</td></tr> <tr><td>9₉</td><td>6₁₀</td><td>7₁₁</td><td>12₁₂</td></tr> <tr><td>4₁₃</td><td>15₁₄</td><td>14₁₅</td><td>1₁₆</td></tr> </table>	16 ₁	3 ₂	2 ₃	13 ₄	5 ₅	10 ₆	11 ₇	8 ₈	9 ₉	6 ₁₀	7 ₁₁	12 ₁₂	4 ₁₃	15 ₁₄	14 ₁₅	1 ₁₆	4×sum_ctr, extra parameters =, 34
16 ₁	3 ₂	2 ₃	13 ₄															
5 ₅	10 ₆	11 ₇	8 ₈															
9 ₉	6 ₁₀	7 ₁₁	12 ₁₂															
4 ₁₃	15 ₁₄	14 ₁₅	1 ₁₆															
matrix(4,4,2,2)	<table border="1"> <tr><td>16₁</td><td>3₂</td><td>2₃</td><td>13₄</td></tr> <tr><td>5₅</td><td>10₆</td><td>11₇</td><td>8₈</td></tr> <tr><td>9₉</td><td>6₁₀</td><td>7₁₁</td><td>12₁₂</td></tr> <tr><td>4₁₃</td><td>15₁₄</td><td>14₁₅</td><td>1₁₆</td></tr> </table>	16 ₁	3 ₂	2 ₃	13 ₄	5 ₅	10 ₆	11 ₇	8 ₈	9 ₉	6 ₁₀	7 ₁₁	12 ₁₂	4 ₁₃	15 ₁₄	14 ₁₅	1 ₁₆	4×sum_ctr, extra parameters =, 34
16 ₁	3 ₂	2 ₃	13 ₄															
5 ₅	10 ₆	11 ₇	8 ₈															
9 ₉	6 ₁₀	7 ₁₁	12 ₁₂															
4 ₁₃	15 ₁₄	14 ₁₅	1 ₁₆															
matrix(8,2,4,1)	<table border="1"> <tr><td>16₁</td><td>3₂</td><td>2₃</td><td>13₄</td></tr> <tr><td>5₅</td><td>10₆</td><td>11₇</td><td>8₈</td></tr> <tr><td>9₉</td><td>6₁₀</td><td>7₁₁</td><td>12₁₂</td></tr> <tr><td>4₁₃</td><td>15₁₄</td><td>14₁₅</td><td>1₁₆</td></tr> </table>	16 ₁	3 ₂	2 ₃	13 ₄	5 ₅	10 ₆	11 ₇	8 ₈	9 ₉	6 ₁₀	7 ₁₁	12 ₁₂	4 ₁₃	15 ₁₄	14 ₁₅	1 ₁₆	4×sum_ctr, extra parameters =, 34
16 ₁	3 ₂	2 ₃	13 ₄															
5 ₅	10 ₆	11 ₇	8 ₈															
9 ₉	6 ₁₀	7 ₁₁	12 ₁₂															
4 ₁₃	15 ₁₄	14 ₁₅	1 ₁₆															
matrix(2,8,2,2)	<table border="1"> <tr><td>16₁</td><td>3₂</td><td>2₃</td><td>13₄</td><td>5₅</td><td>10₆</td><td>11₇</td><td>8₈</td></tr> <tr><td>9₉</td><td>6₁₀</td><td>7₁₁</td><td>12₁₂</td><td>4₁₃</td><td>15₁₄</td><td>14₁₅</td><td>1₁₆</td></tr> </table>	16 ₁	3 ₂	2 ₃	13 ₄	5 ₅	10 ₆	11 ₇	8 ₈	9 ₉	6 ₁₀	7 ₁₁	12 ₁₂	4 ₁₃	15 ₁₄	14 ₁₅	1 ₁₆	4×sum_ctr, extra parameters =, 34
16 ₁	3 ₂	2 ₃	13 ₄	5 ₅	10 ₆	11 ₇	8 ₈											
9 ₉	6 ₁₀	7 ₁₁	12 ₁₂	4 ₁₃	15 ₁₄	14 ₁₅	1 ₁₆											
diagonal	<table border="1"> <tr><td>16₁</td><td>3₂</td><td>2₃</td><td>13₄</td></tr> <tr><td>5₅</td><td>10₆</td><td>11₇</td><td>8₈</td></tr> <tr><td>9₉</td><td>6₁₀</td><td>7₁₁</td><td>12₁₂</td></tr> <tr><td>4₁₃</td><td>15₁₄</td><td>14₁₅</td><td>1₁₆</td></tr> </table>	16 ₁	3 ₂	2 ₃	13 ₄	5 ₅	10 ₆	11 ₇	8 ₈	9 ₉	6 ₁₀	7 ₁₁	12 ₁₂	4 ₁₃	15 ₁₄	14 ₁₅	1 ₁₆	2×sum_ctr, extra parameters =, 34 2×strictly_decreasing
16 ₁	3 ₂	2 ₃	13 ₄															
5 ₅	10 ₆	11 ₇	8 ₈															
9 ₉	6 ₁₀	7 ₁₁	12 ₁₂															
4 ₁₃	15 ₁₄	14 ₁₅	1 ₁₆															

into b blocks such that each block contains exactly k distinct objects, each object occurs in exactly r different blocks, and every two distinct objects occur together in exactly λ blocks. Another way of defining a BIBD is in terms of its incidence matrix, which is a binary matrix with v rows, b columns, r ones per row, k ones per column, and scalar product λ between any pair of distinct rows. A BIBD is therefore specified by its parameters (v, b, r, k, λ) . We consider the $(7,7,3,3,1)$ design, with the sample, given as an incidence matrix, shown in Figure 4. Figure 5 shows the constraints found based on this one positive sample, together with the partition on which they apply. The constraints are expressed on the rows and columns, and an additional, spurious `no_peak` constraint set on the diagonals. We find the row and column sums (`sum_ctr`), which work on each row and column individually. But we also find the `scalar_product`

Fig. 3: Solutions to Generated Model

13	3	2	16	13	2	3	16	16	2	5	11	16	3	2	13	16	2	3	13
8	10	11	5	8	11	10	5	3	13	10	8	5	10	11	8	5	11	10	8
12	6	7	9	12	7	6	9	9	7	4	14	9	6	7	12	9	7	6	12
1	15	14	4	1	14	15	4	6	12	15	1	4	15	14	1	4	14	15	1

Table 1: Number of Pattern Found in 100 Runs, for Different Sample Sizes

Size	1	2	3	4	5	6	7	8	9	10	11
1	-	-	-	28	26	19	15	2	5	2	3
2	-	-	69	25	3	2	-	-	-	-	1
3	-	-	87	12	1	-	-	-	-	-	-
4	-	-	93	6	1	-	-	-	-	-	-
5	-	-	95	5	-	-	-	-	-	-	-
6	-	-	99	1	-	-	-	-	-	-	-
7	-	-	98	2	-	-	-	-	-	-	-
8	-	-	100	-	-	-	-	-	-	-	-
9	-	-	100	-	-	-	-	-	-	-	-

Fig. 4: Sample (7,7,3,3,1) Design

0	0	0	0	1	1	1
0	0	1	1	0	0	1
0	1	0	1	0	1	0
0	1	1	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	1	0	0	0	0	1

constraint, which is expressed on all pairs of rows resp. columns. For the given example we also find the double-lex constraints which impose the lexicographical order on rows and columns in our example. They are expressed in two ways: the `lex_chain_less` considers the matrix of all rows resp. columns, while the `lex_less` constraint expresses the same condition on all pairs of rows resp. columns.

6.3 Orthogonal Latin Squares

As a last example we consider Orthogonal Latin Squares of size n , which are two $n \times n$ matrices A and B containing numbers from 1 to n , which are Latin Squares (i.e. each

Fig. 5: Constraints Found for BIBD Example

Partition	Constraints
matrix(7,7,7,1)	all pairs: 21×scalar_product 7×sum_ctr, extra parameters =, 3 matrix: 1×lex_chain_less all pairs: 21×lex_less
matrix(7,7,1,7)	all pairs: 21×scalar_product 7×sum_ctr, extra parameters =, 3 matrix: 1×lex_chain_less all pairs: 21×lex_less
diagonal	2×no_peak

row and column is alldifferent), and where the tuples $\langle a_{ij}, b_{ij} \rangle$ are pairwise different. One finds a list of all low-order Latin Squares at <http://cs.anu.edu.au/~bdm/data/latin.html>, with samples given as vectors of integer values, e.g. 0, 2, 1, 3, 4, 6, 5, 6, 1, 4, 2, 5, 3, 0, 1, 0, 6, 5, 3, 4, 2, 2, 5, 0, 4, 6, 1, 3, 5, 3, 2, 6, 1, 0, 4, 3, 4, 5, 1, 0, 2, 6, 4, 6, 3, 0, 2, 5, 1, 0, 6, 5, 2, 1, 4, 3, 5, 0, 4, 3, 2, 1, 6, 2, 1, 3, 5, 0, 6, 4, 1, 4, 2, 0, 6, 3, 5, 6, 3, 0, 1, 4, 5, 2, 4, 5, 1, 6, 3, 2, 0, 3, 2, 6, 4, 5, 0, 1 as an example of a 7x7 Orthogonal Latin Square. Note that this does not indicate where we can find the matrices A and B, but just provides a vector of integer values, as we require as input format.

The constraints found by our program are shown in Figure 6, we identify the matrices A and B, find the key `alldifferent_consecutive_values` constraints on rows and columns, and find the `lex_alldifferent` constraint on all pairs $\langle a_{ij}, b_{ij} \rangle$, just by exploring different matrix generators. We also find weaker `lex_alldifferent` constraints and a `sum_ctr` constraint. Some of these may be implied by domination rules which have not been implemented yet.

Fig. 6: Constraints Found for Orthogonal Latin Squares Example

Partition	Constraints
matrix(14,7,7,1)	14xalldifferent_consecutive_values
matrix(17,7,1,7)	14xalldifferent_consecutive_values
matrix(2,49,2,1)	1xlex_alldifferent
matrix(7,14,7,7)	2xsum_ctr with extra parameters =, 147
matrix(7,14,7,1)	1xlex_alldifferent
matrix(17,7,7,7)	
matrix(49,2,7,1)	
matrix(7,14,7,2)	
matrix(14,7,2,7)	
matrix(14,7,1,7)	

7 Related Work

Our proposed method is a special, restricted case of Constraint Acquisition. Constraint Acquisition [17] is the process of finding a constraint network from a training set of positive and negative examples. The learning process operates on a library of allowed constraints, and a resulting solution is a conjunction of constraints from that library, each constraint ranging over a subset of the variables.

This area of research has attracted a fair bit of work over the last ten years [8, 24, 6, 16, 5, 12, 7, 11, 20, 22, 21]. A key idea for solving this problem is the use of version space learning from AI, which considers the set of all possible constraint networks which accept the training set.

In an interactive setting, the training set is not fixed, but will be derived incrementally. If the target model has not been identified, the system may suggest new training instances, which the user has to classify as either positive or negative. Ideally, these new examples are chosen to maximally reduce the version space that needs to be considered.

One of the challenges of constraint acquisition for a library of global constraints is that many global constraints have additional parameters which might not occur in the examples given, which only list the main decision variables describing the problem. The values of these parameters must be learned from the examples as well, this is considered in [9, 10].

Another issue is that in constraint acquisition we don't know over which subset of the decision variables a constraint will be expressed. When we consider only binary constraints, this does not matter, we can explore all binary combinations of variables in quadratic time. For a global constraint with k variables which ranges over a subset of n decision variables, we are faced with a combinatorial explosion, especially if the order of the variables in the constraint matters.

We address these two problems in the approach presented here:

- When considering structured variable sequences over multiple positive examples, we can quite systematically explore which global constraints, with sufficient additional parameters, may be consistent with all sequences given. In case of functional dependencies we do not have to guess these additional parameter values, we can derive them automatically from the instances.
- By systematically exploring structured variable partitions we avoid the potential combinatorial explosion of looking at every subset of the variables. These partitions lead to potential sequences of constraints, of which we only keep those with sufficient structure (periodic or with limited number of changes). This also avoids the problem of finding a multitude of candidate constraints over random subsets of the variables. At the same time we can, by considering solutions with higher cost, also find less structured sets of constraints over the considered sequences.

8 Conclusion

We have presented in this paper a first step towards building a practical constraint acquisition system which can automatically derive structured constraint models from small sets of positive examples, considering the global constraints in the global constraint catalog as basic building blocks. This work extends our previous results on a Constraint Seeker by partitioning given positive examples systematically into structured subsets, and applying the Constraint Seeker on each of these sequences. We then solve a multi-criteria constraint optimization problem to discover regular constraint structures over these sequences, and finally apply meta-data from the constraint catalog to filter dominated constraint candidates. Initial experiments indicate that this method can be applied to a variety of structured constraint problems.

References

1. N. Beldiceanu. Global Constraints as Graph Properties on a Structured Network of Elementary Constraints of the Same Type. In *CP'2000*, volume 1894 of *LNCS*, pages 52–66.
2. N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global Constraint Catalog, 2nd Edition. Technical Report T2010-07, Swedish Institute of Computer Science, 2010.

3. N. Beldiceanu and E. Contejean. Introducing Global Constraints in CHIP. *Mathl. Comput. Modelling*, 20(12):97–123, 1994.
4. N. Beldiceanu and H. Simonis. A Constraint Seeker: Finding and Ranking Global Constraints from Examples. In J. H.M. Lee, editor, *CP'2011*, LNCS, Perugia, Italy, 2011.
5. C. Bessière, R. Coletta, E. C. Freuder, and B. O'Sullivan. Leveraging the learning power of examples in automated constraint acquisition. In *CP'2004*, pages 123–137, 2004.
6. C. Bessière, R. Coletta, F. Koriche, and B. O'Sullivan. A SAT-based version space algorithm for acquiring constraint satisfaction problems. In J. Gama, R. Camacho, P. Brazdil, A. Jorge, and L. Torgo, editors, *ECML*, volume 3720 of *Lecture Notes in Computer Science*, pages 23–34. Springer, 2005.
7. C. Bessière, R. Coletta, F. Koriche, and B. O'Sullivan. Acquiring constraint networks using a SAT-based version space algorithm. In *AAAI*. AAAI Press, 2006.
8. C. Bessière, R. Coletta, B. O'Sullivan, and M. Paulin. Query-driven constraint acquisition. In Veloso [23], pages 50–55.
9. C. Bessière, R. Coletta, and T. Petit. Acquiring parameters of implied global constraints. In *CP'2005*, pages 747–751, 2005.
10. C. Bessière, R. Coletta, and T. Petit. Learning implied global constraints. In Veloso [23], pages 44–49.
11. J. Charnley, S. Colton, and I. Miguel. Automatic generation of implied constraints. In G. Brewka, S. Coradeschi, A. Perini, and P. Traverso, editors, *ECAI*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 73–77. IOS Press, 2006.
12. R. Coletta, C. Bessière, B. O'Sullivan, E. C. Freuder, S. O'Connell, and J. Quinqueton. Semi-automatic modeling by constraint acquisition. In *CP'2003*, pages 812–816, 2003.
13. M. R. Garey and D. S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W.H.Freeman and co., San Francisco, 1979.
14. B. M. Hernández. *The Systematic Generation of Channelled Models in Constraint Satisfaction*. PhD thesis, University of York, York, YO10 5DD, UK, 2007.
15. M. J. Maher. Open Constraints in a Boundable World. In W.-J. van Hove and J. N. Hooker, editors, *CPAIOR'09*, volume 5547 of *LNCS*, pages 163–177, Pittsburgh, PA, USA, 2009.
16. S. O'Connell, B. O'Sullivan, and E. C. Freuder. Timid acquisition of constraint satisfaction problems. In H. Haddad, L. M. Liebrock, A. Omicini, and R. L. Wainwright, editors, *SAC*, pages 404–408. ACM, 2005.
17. B. O'Sullivan. Automated modelling and solving in constraint programming. In M. Fox and D. Poole, editors, *AAAI*. AAAI Press, 2010.
18. T. Petit, J.-C. Régin, and C. Bessière. Specific Filtering Algorithms for Over-Constrained Problems. In T. Walsh, editor, *Principles and Practice of Constraint Programming (CP'2001)*, volume 2239 of *LNCS*, pages 451–463. Springer-Verlag, 2001.
19. J.-F. Puget. Constraint Programming Next Challenge: Simplicity of Use. In M. G. Wallace, editor, *CP'2004*, volume 3258 of *LNCS*, pages 5–8, 2004.
20. J. Quinqueton, G. Raymond, and C. Bessière. An agent for constraint acquisition and emergence. In V. Negru, T. Jebelean, D. Petcu, and D. Zaharie, editors, *SYNASC*, pages 229–234. IEEE Computer Society, 2007.
21. F. Rossi and A. Sperduti. Acquiring both constraint and solution preferences in interactive constraint systems. *Constraints*, 9(4):311–332, 2004.
22. K. M. Shchekotykhin and G. Friedrich. Argumentation based constraint acquisition. In W. Wang, H. Kargupta, S. Ranka, P. S. Yu, and X. Wu, editors, *ICDM*, pages 476–482. IEEE Computer Society, 2009.
23. M. M. Veloso, editor. *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, 2007.
24. X. Vu and B. O'Sullivan. Generalized constraint acquisition. In *SARA'07*, pages 411–412, 2007.