Learning Structured Constraint Models: a First Attempt

Nicolas Beldiceanu[1] 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. In this paper we give an overview of an early prototype which learns structured constraint models from flat, positive examples of solutions. It is based on previous work on a Constraint Seeker, which finds constraints in the global constraint catalog satisfying positive and negative examples. In the current tool we extend this system to find structured conjunctions of constraints on regular subsets of variables in the given solutions. Two main elements of the approach are a bi-criteria optimzation problem which finds conjunctions of constraints which are both regular and relevant, and a syntactic dominance check between conjunctions, which removes implied constraints without requiring a full theorem prover, using meta-data in the constraint catalog. Some initial experiments on a proof-of-concept implementation show promising results.

1 Scope and Assumptions

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 [13], 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, 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 standard Operations Research problems 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 ourself 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 [11].

Our approach takes advantage of assumptions 3. and 5., i.e., on the fact that our samples are reliable and that we restrict ourself 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 constraints 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.
- 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 will evaluate our method on some initial example problems, while Section 4 looks at related work.

2 Overview of the Learning Algorithm

The learning algorithm is decomposed into the following successive steps:

1. Given $\mathcal{V} = v_1, v_2, \ldots, v_s$ 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 systematicaly propose different ways of grouping variables together, which can be both described concisely

and which matches the pattern found in typical constraint models. One example is the *matrix partition generator* $matrix(p, q, \alpha, \beta)$ which interprets a sample of size $s = p * q$ as a $p \times q$ matrix, and extracts blocks of size $\alpha * \beta$ from α rows and β columns. Another example is the *diagonal* generator which, for $s = n * n$, extracts the two main diagonals of the $n \times n$ matrix. Other generators are described in `http://4c.ucc.ie/~hsimonis/modref11.pdf`, which also contains a more detailed description of the other components of our tool.

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 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 here.

3. For each sequence of variables the *candidate generator* takes the corresponding ground parameters built 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. One of the criteria is the regularity of the set of constraints. We prefer sequences which change only a few times, or which have a periodic structure with a short period. We can express this regularity by representing the constraint alternatives by integer indices, and then expressing `change` or `period` constraints over possible integer sequences. For each selected constraint we also know its rank in the Constraint Seeker output, which can be linked to the constraint index by `element` constraints. For a sequence of constraints, the rank is defined as the sum of the ranks of its constraints. These two criteria are incomparable, the resulting problem is therefore treated as a multi-criteria optimization problem.

5. Given a set of conjunctions of constraints found by the relevance optimizer, the *dominance checker* discards conjunctions that are implied by other conjunctions. As checking for implications between arbitrary sets of constraints is hard even for a full-featured theorem prover, we use a weaker domination criteria, based on meta-data in the constraint catalog. Key concepts besides implication are contractible and expendible. A constraint is *contractible* w.r.t. to one of its arguments, if we can remove any variable from that argument in a solution, and still obtain a solution. The `alldifferent` constraint for example is contractible. A constraint is *expendible* if we can add any value to an argument in a solution, and still obtain a solution. The `atleast` constraint is extensible. Used together with meta-data about implications, we can now check if one set of constraints is dominated (and therefore implied) by another one, based on a simple, syntactic check.

3 Evaluation

Instead of describing the details of our algorithm formally, we concentrate here on presenting some initial example uses, which show the potential of the proposed method.

3.1 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 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.

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}

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.

Fig. 2: Constraints Found for Magic Square

Partition Generator	Partition	Constraint(s)
matrix(16,1,16,1)	original sequence of values	$1\times$`alldifferent_consecutive_values` $1\times$`symmetric_alldifferent`, extra parameter $[1, 2, \ldots, 16]$
matrix(4,4,1,4)	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}	$4\times$`sum_ctr`, extra parameters $=$, 34
matrix(4,4,4,1)	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}	$4\times$`sum_ctr`, extra parameters $=$, 34
matrix(4,4,2,2)	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}	$4\times$`sum_ctr`, extra parameters $=$, 34
matrix(8,2,4,1)	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}	$4\times$`sum_ctr`, extra parameters $=$, 34
matrix(2,8,2,2)	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}	$4\times$`sum_ctr`, extra parameters $=$, 34
diagonal	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}	$2\times$`sum_ctr`, extra parameters $=$, 34 $2\times$`strictly_decreasing`

Fig. 3: Solutions to Generated Model

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

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

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

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

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

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

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	-	-	-	-	-	-	-

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.

3.2 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 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

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

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 paramters =, 3 matrix: 1×`lex_chain_less` all pairs: 21×`lex_less`
diagonal	2×`no_peak`

the row and column sums (`sum_ctr`), which work on each row and column individually. But we also find the `scalar_product` 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.

3.3 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 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 for the Latin Squares, 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)	14×`alldifferent_consecutive_values`
matrix(17,7,1,7)	14×`alldifferent_consecutive_values`
matrix(2,49,2,1)	1×`lex_alldifferent`
matrix(7,14,7,7)	2×`sum_ctr` with extra parameters =, 147
matrix(7,14,7,1) 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)	1×`lex_alldifferent`

4 Related Work

Our proposed method is a special, restricted case of Constraint Acquisition. Constraint Acquisition [12] 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 [6, 10, 5, 9, 14]. 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 [7, 8].

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.

5 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 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 R. Dechter, editor, *Principles and Practice of*

Constraint Programming (CP'2000), volume 1894 of *LNCS*, pages 52–66. Springer-Verlag, 2000. Preprint available as SICS Tech Report T2000-01.

2. N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global Constraint Catalog, 2nd Edition. Technical Report T2010-07, Swedish Institute of Computer Science, 2010. Available at `ftp://ftp.sics.se/pub/SICS-reports/Reports/SICS-T--2005-08--SE.pdf`.
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, *Principles and Practice of Constraint Programming (CP'2011)*, LNCS, Perugia, Italy, 2011. Springer-Verlag.
5. 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.
6. C. Bessière, R. Coletta, B. O'Sullivan, and M. Paulin. Query-driven constraint acquisition. In Veloso [15], pages 50–55.
7. C. Bessière, R. Coletta, and T. Petit. Acquiring parameters of implied global constraints. In P. van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 747–751. Springer, 2005.
8. C. Bessière, R. Coletta, and T. Petit. Learning implied global constraints. In Veloso [15], pages 44–49.
9. 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.
10. R. Coletta, C. Bessière, B. O'Sullivan, E. C. Freuder, S. O'Connell, and J. Quinqueton. Semi-automatic modeling by constraint acquisition. In F. Rossi, editor, *CP*, volume 2833 of *Lecture Notes in Computer Science*, pages 812–816. Springer, 2003.
11. 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.
12. B. O'Sullivan. Automated modelling and solving in constraint programming. In M. Fox and D. Poole, editors, *AAAI*. AAAI Press, 2010.
13. J.-F. Puget. Constraint Programming Next Challenge: Simplicity of Use. In M. G. Wallace, editor, *Principles and Practice of Constraint Programming (CP'2004)*, volume 3258 of *LNCS*, pages 5–8. Springer-Verlag, 2004.
14. F. Rossi and A. Sperduti. Acquiring both constraint and solution preferences in interactive constraint systems. *Constraints*, 9(4):311–332, 2004.
15. M. M. Veloso, editor. *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, 2007.