# An Introduction of FD-Complete Constraints

Sven Löffler(✉), Ke Liu, and Petra Hofstedt

Brandenburg University of Technology Cottbus-Senftenberg, Cottbus, Germany
Sven.Loeffler@b-tu.de, Hofstedt@b-tu.de

**Abstract.** The performance of solving a constraint problem can often be improved by converting a subproblem into a single constraint (for example into a regular membership constraint or a table constraint). In the past, it stood out, that specialist constraint solvers (like simplex solver or SAT solver) outperform general constraint solvers, for the problems they can handle. The disadvantage of such specialist constraint solvers is that they can handle only a small subset of problems with special limitations to the domains of the variables and/or to the allowed constraints. In this paper we introduce the concept of fd-complete constraints and fd-complete constraint satisfaction problems, which allow combining both previous approaches. More accurately, we convert general constraint problems into problems which use only one, respectively one kind of constraint. The goal is it to interpret and solve the converted constraint problems with specialist solvers, which can solve the transformed constraint problems faster than the original solver the original constraint problems.

**Keywords:** Constraint programming · CSP · Refinement · Optimizations · Regular membership constraint · Regular CSPs · Table constraint · FD-completeness

## 1 Introduction

Constraint programming (CP) is a powerful method to model and solve NP-complete problems in a declarative way. Typical research problems in CP are among others rostering, graph coloring, optimization, resource management, planning, scheduling and satisfiability (SAT) problems [12].

Because the search space of constraint satisfaction problems (CSPs) and constraint satisfaction optimization problems (CSOPs or COPs) is immensely big and the solution process often needs an extremely high amount of time we are always interested in improving the solution process. There are various ways to describe a CSP in practice and consequently, the problem can be modeled by different combinations of constraints, which results in the differences in resolution speed and behavior.

In particular, there is the possibility to represent a CSP with only one constraint or with constraints, which are of the same kind. Inspired by the concept of np-complete problems, we introduce the definition of fd-complete constraints.


Published by Springer Nature Switzerland AG 2020
I. Maglogiannis et al. (Eds.): AIAI 2020, IFIP AICT 584, pp. 27–38, 2020.
https://doi.org/10.1007/978-3-030-49186-4_3

A *finite domain complete constraint (fd-complete constraint)* is a finite domain constraint which can represent every other fd constraint. Thus an fd-complete constraint can be used to replace each other constraint in a CSP.

Possible representatives of fd-complete constraints are the *table* constraint or the *regular membership* constraint. Previous researches show how constraints can be transformed into *table* [2] and *regular membership* [8–11] constraints. These publications also showed, that already the transformations into a *regular membership* or *table* constraint can improve the solving speed of a CSP.

In this paper, we want to go a step further. We not only transform parts of a CSP into an fd-complete constraint, we transform the whole CSP into one, which contains only a set (of the same kind) of fd-complete constraints. Thus we have the possibility to use a solver, solver settings or search strategies which are optimized for the used constraints.

The rest of this paper is structured as follows. In Sect. 2, we explain the necessary definitions of constraint programming. In Sect. 3, we introduce our new concept of fd-complete constraint satisfaction problem. Section 4 shows the transformation of a regular CSP into a binary variable scalar CSP. In Sect. 5, a rostering example is given to underline the benefit of finite domain CSPs. Finally, Sect. 6 draws a conclusion and proposes research directions in the future.

*Remark 1:* We use the notation of a *regular* constraint as a synonym for *regular membership* respectively *regular language membership* constraint.

## 2 Preliminaries

In this section we introduce some basic definitions and concepts of constraint programming (CP) and show the relevant constraints, which are used in the rest of the paper. We consider *CSPs*, which are defined in the following way.

A *constraint satisfaction problem (CSP)* is defined as a 3-tuple $P = (X, D, C)$ where $X = \{x_1, x_2, \ldots, x_n\}$ is a set of variables, $D = \{D_1, D_2, \ldots, D_n\}$ is a set of finite domains where $D_i$ is the domain of $x_i$ and $C = \{c_1, c_2, \ldots, c_m\}$ is a set of constraints covering between one and all variables of $X$ [16].

A *constraint* $c_j = (X_j, R_j)$ is a relation $R_j$, which is defined over a set of variables $X_j \subseteq X$ [4].

The *scope* of a constraint $c_j = (X_j, R_j)$ indicates the set of variables, which is covered by the constraint $c_j$: $scope(c_j) = X_j$ [4].

The relation $R$ of a constraint $c = (X, R)$ is more clearly a subset of the Cartesian product of the domain values $D_1 \times ... \times D_n$ of the corresponding variables $X = \{x_1, ..., x_n\}$. Examples for constraints are amongst other things $c_1 = (\{x, y\}, (x > y)), c_2 = (\{R, U, I\}, (R = U/I))$ or $c_3 = (\{A, B, C\}, (A \wedge B \rightarrow C))$. For the reason that we only consider finite domains, it is always possible to enumerate all allowed tuples of a constraint, which we will use for the transformation into other constraints like *regular* or *table* constraints.

Based on the definition above, an fd-complete constraint can substitute all constraints $C$ of a given CSP $P = (X, D, C)$. For the reason that a CSP can only

have a limited number of solutions $c_{sol}$, because we have a limited number of variables $n$ and a limited number of values inside of the domains of the variables, it is possible to enumerate all of the solutions of a CSP.

It follows a list of definitions of the relevant constraints for this paper. Let a CSP $P = (X, D, C)$ and a subset $X'$ of variables $X$ of the CSP $P$ be given.

The *scalar* constraint guarantees for an ordered subset of variables $X' = \{x_1, \ldots, x_n\} \subseteq X$, a vector of integers $v = (v_1, ..., v_n)$, a relation $\Re$ and another variable $x_r \in X$ that the scalar product of the variables $X'$ with the vector $v$ is in relation $\Re$ to $x_r$.

$$scalar(X', v, \Re, x_r) = (\sum_{i=1}^{n} x_i * v_i) \ \Re \ x_r$$

Like we already figured out, the *table* and the *regular* constraints are representatives of fd-complete constraints. It follows a short description of these both constraints.

The *table* constraint is one of the most frequently used constraints in practice. For an ordered subset of variables $X' = \{x_1, \ldots, x_n\} \subseteq X$, a positive (negative) *table* constraint defines that any solution of the CSP $P$ must (not) be explicitly assigned to a tuple $t$ of a given tuple list $T$, which consists the allowed (disallowed) combinations of values for $X'$. For a given list of tuples $T$, we can state the positive *table* constraint as:

$$table(X', T) = \{(x_1, \ldots, x_n) \mid x_1 \in D_1, \ldots, x_n \in D_n\} \subseteq T$$

The *regular* constraint and its propagation [5,13,14] is based on deterministic finite automatons (DFAs) [7]. Thus, we briefly review the notion of a deterministic finite automaton (DFA) before we define the *regular* constraint.

A *deterministic finite automaton (DFA)* is a quintuple $M = (Q, \Sigma, \delta, q_0, F)$, where $Q$ is a finite set of states, $\Sigma$ is the finite input alphabet, $\delta$ is a transformation function $Q \times \Sigma \rightarrow Q$, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final or accepting states. A word $w \in \Sigma^*$ is accepted by $M$, if the corresponding DFA $M$ with the input $w$ stops in a final state $f \in F$. All accepting words of the DFA $M$ can be summarized to the accepting language $L(M)$ of the DFA $M$ [7].

The *regular constraint* is another of the most common constraints in practice. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA, let $X = \{x_1, \ldots, x_n\}$ be an ordered set of variables with domains $D = \{D_1, D_2, \ldots, D_n\}$, $\forall i \in \{1, \ldots, n\} : \ D_i \subseteq \Sigma$. The regular constraint is defined as [6]:

$$regular(X, M) = \\ \{(w_1, \ldots, w_n) \mid \forall i \in \{1, \ldots, n\}, w_i \in D_i, \ (w_1 w_2 \ldots w_n) \in L(M)\}$$

So the concatenation of the values $v_i$ of the variables $x_i \forall i \in \{1, ..., n\}$ must be accepted by the automaton $M$. The input DFA $M$ of a *regular* constraint is internally transformed into a directed acyclic graph (DAG) $M'$ [14].

We define a *directed acyclic graph (DAG)* as a DFA $M = (Q, \Sigma, \delta, q_{initial}, F)$, where the states $q$ in $Q$ are partitioned into levels $Q = \{Q_0, ..., Q_n | \ Q_i = \{q_{i,1}, q_{i,2}, ...\} \mid \forall i \in \{1, ..., n\}\}$, the initial state $q_{0,0}$ is the only element of $Q_0$, the final states are members of the last level of states ($F = Q_n$), and transitions are only allowed from a state $q_{i,j}$ in level $Q_i$ to a state $q_{t,k}$ in level $Q_t$, where $t \geq i$.

For our use, we only allow transitions from a state $q_{i,j}$ in level $Q_i$ to a state $q_{i+1,k}$ in level $Q_{i+1}$. We use the notation $q_{i,j} \xrightarrow{v} q_{i+1,k}$, if the state $q_{i,j}$ of level $Q_i$ have a transition with value $v$ to the state $q_{i+1,k}$ of level $Q_{i+1}$. A solution of a *regular* constraint is a path from the initial state $q_{0,0}$ to the final state $q_{n,0}$ with values $v_1 \in D_1, ..., v_n \in D_n$ $(q_{0,0} \xrightarrow{v_1} q_{1,j_1} \xrightarrow{v_2} \cdots \xrightarrow{v_n} q_{n,0})$.

Based on the definition of the regular constraint, we use the notion of a regular constraint satisfaction problem, and analogously the notation of a table CSP:

A *regular constraint satisfaction problem (regular CSP)* is defined as a 3-tuple $P = (X, D, C)$, where $X = \{x_1, x_2, \ldots, x_n\}$ is a set of variables, $D = \{D_1, D_2, \ldots, D_n\}$ is a set of finite domains where $D_i$ is the domain of $x_i$ and $C = \{c_1, c_2, \ldots, c_m\}$ is a set of *regular* constraints over variables of $X$.

A *table constraint satisfaction problem (table CSP)* is defined as a 3-tuple $P = (X, D, C)$, where $X = \{x_1, x_2, \ldots, x_n\}$ is a set of variables, $D = \{D_1, D_2, \ldots, D_n\}$ is a set of finite domains where $D_i$ is the domain of $x_i$ and $C = \{c_1, c_2, \ldots, c_m\}$ is a set of *table* constraints over variables of $X$.

## 3 FD-Complete CSPs

After we defined fd-complete constraints and showed two examples (the *table* and the *regular* constraint), we will extend this concept a little bit more. The possibility to substitute a constraint or a subset of constraints of a CSP with an fd-complete constraint can lead to a speed up in the solution process [2,8–11]. The problem is that it is mostly not useful to transform a whole CSP into another one with only one fd-complete constraint, which substitutes all constraints of the original CSP. In most cases, the transformation process will be more time consuming than solving the original problem.

Mostly, we can reduce this slow-down significantly and reach a speed-up, if we substitute only some of the constraints $c_i \in C$ of a given CSP $P = (X, D, C)$ with semantically equivalent **fd**-complete constraints $c_i^{\mathbf{fd}} \in C^{\mathbf{fd}}$ and combine them together (to constraints $c_j^{\text{c\_fd}} \in C^{\text{c\_fd}}$). Finding the best subset of constraints, which should be transformed and combined, depends on several things like the needed time for transformation or the size of the data structures of the transformed or the combined constraints.

Often, a CSP $P' = (X, D, C')$, which contains some of the original constraints $c_i \in C$ and some of the combined constraints $c_j^{\text{c\_fd}}$, which are semantically equivalent to a subset of the fd-complete constraints $C^{\text{fd}}$, has the fastest solution process.

In contrast to this, we will explain why it can be useful to create a CSP $P = (X, D, C)$ with only singleton $c_i^{\text{fd}}$ and combined fd-complete constraints $c_j^{\text{c\_fd}}$, even though the propagation of the original constraint $c_i$ is possibly faster than the propagation of the corresponding fd-complete constraint $c_i^{\text{fd}}$.

The advantage is, that we may be able to (create and) use specialist constraint solvers for CSPs which have only constraints of a special type (a popular example is the simplex algorithm, which allows only linear optimization under linear side conditions). Furthermore, we can create and use more efficient search strategies and solver settings if the used constraints have the same shape.

Also interesting for future researches is, that a CSP $P$ with only one kind of constraint may be able to be transformed easier into other models for example a SAT model and make it possibly simpler to transfer a constraint problem of a given language into another one. The last idea is for example used to translate MiniZinc [18] models into other models (like Gecode [17], Google OR-Tools [1] or Choco [15] models). Thus, many constraints, which were entered in MiniZinc, will be translated into the *table* constraint.

Based on the potential benefit of CSPs, which contain only fd-complete constraints, we created the definition of an fd-complete CSP.

A *finite domain complete constraint satisfaction problem (fd-complete CSP)* is a CSP $P = (X, D, C)$ which contains only fd-complete constraints of the same kind.

Examples for an fd-complete CSP are the *regular CSP* [9] or the *table CSP*. We call fd-complete CSPs with $n$ different kinds of constraints fd-complete CSPs of degree $n$. We differentiate between two kinds of fd-complete CSPs. On the one hand we have directly convertible fd-complete CSPs like table or regular CSPs. Every general CSP $P = (X, D, C)$ can be transformed into a **d**irectly **c**onvertible **fd**-complete CSP $P^{\textbf{dc\_fd}} = (X, D, C^{\text{fd}})$ by transformation of each single constraint $c \in C$ to the corresponding fd-complete constraint $c \in C^{\text{fd}}$. In particular, it is not necessary to transform the variables $X$ or domains $D$ of the original CSP $P$.

On the other hand, there are **i**ndirectly **c**onvertible **fd**-complete CSPs $P^{\textbf{ic\_fd}} = (X', D', C^{\text{fd}})$, where the variables and their domains must be transformed additionally to the constraints. We present a representative example of this group in the next section.

## 4 The Binary Scalar CSP as a Representative of FD-Complete CSPs

In this Section, we will introduce a description of how a general CSP $P$ can be transformed into a CSP $P^{\text{bvsc}}$, which contains only **b**inary **v**ariables and **s**calar **c**onstraints. Using only binary variables allows us the use of the very good researched SAT solvers, for solving $P^{\text{bvsc}}$. Furthermore, for the reason that the CSP $P^{\text{bvsc}}$ only contains scalar constraints, we expect that a possibly more accurate and more specialized solver can be created to solve it.

Traditionally, an input CSP $P$ for a SAT solver contains binary variables and logic constraints like *and*, *or*, *implication*, *negation* etc. It is also possible to transform a regular constraint into a set of logic constraints, but we will see, that the transformation into a set of *scalar* constraints is more compact. Thus, we expect that a specialized solver for *scalar* constraints over binary variables works faster than a usual SAT solver.

The transformation from a general CSP $P$ to a binary variable scalar CSP $P^{\text{bvsc}}$ can happen in two steps.

Step one: transforming the CSP $P = (X, D, C)$ into a regular CSP $P^{\text{reg}} = (X, D, C^{\text{reg}})$, which contains the same variables $X$, the same domains $D$, but only regular constraints $C^{\text{reg}}$ instead of the original constraints $C$. For example, it is possible to transform each original constraint $c \in C$ into a regular constraint $c^{\text{reg}} \in C^{\text{reg}}$. The algorithms and transformations presented in [8–11] can be used for this step.

Step two: transform the regular CSP $P^{\text{reg}} = (X, D, C^{\text{reg}})$ into a binary variable scalar CSP $P^{\text{bvsc}} = (X^{\text{b}}, D^{\text{b}}, C^{\text{sc}})$. We consider a regular CSP $P^{\text{reg}} = (X, D, C^{\text{reg}})$ with $n$ variables $X = \{x_1, ..., x_n\}$, the associated finite domains $D = \{D_1, ..., D_n\}$ and $m$ regular constraints $C^{\text{reg}} = \{c_1^{\text{reg}}, ..., c_m^{\text{reg}}\}$ and transform it into a binary variable scalar CSP $P^{\text{bvsc}} = (X^{\text{b}}, D^{\text{b}}, C^{\text{sc}})$ with $k$ variables $X^{\text{b}} = \{x_1^{\text{b}}, ..., x_k^{\text{b}}\}$, which have all the domain $\{0, 1\} = D_1^{\text{b}} = ... = D_k^{\text{b}}$, and $l$ scalar constraints $C^{\text{sc}} = \{c_1^{\text{sc}}, ..., c_l^{\text{sc}}\}$.

There are two kinds of variables in $X^{\text{b}}$. The first kind contains for each domain value of each variable of the regular CSP $P^{\text{reg}}$, respectively the original CSP $P$, a binary variable $x_{i,j}^{\text{b}} \forall i \in \{1, ..., n\}, j \in \{1, ..., |D_i|\}$. If such a variable $x_{i,j}^{\text{b}} \in x^{\text{b}}$ of $P^{\text{bvsc}}$ is set to 1, it represents that the corresponding variable $x_i \in X$ in the regular CSP $P^{\text{reg}}$ is set to the value $d_j$, where $d_j$ is the $j$-th value of $D_i$, otherwise, if a variable $x_{i,j}^{\text{b}} \in X^{\text{b}}$ of $P^{\text{bvsc}}$ is set to 0, it represents that the corresponding variable $x_i \in X$ in the regular CSP $P^{\text{reg}}$ cannot be set to the value $d_j$. Thus transform the original search space into a binary variable search space.

The second kind of variables represents the states of the underlying DAGs of the regular constraints $C^{\text{reg}}$. Each regular constraint $c_h^{\text{reg}} \in C^{\text{reg}}$ has a DAG $M_h$ which represents the regular expression of the *regular* constraint. For each state $q_{i,j}^{\text{h}}$ of each DAG $M_h$, a binary variable $x_{i,j}^{\text{h}}$ represents if the state $q_{i,j}^{\text{h}}$ is on a solution path ($x_{i,j}^{\text{h}}$ is set to one) or not ($x_{i,j}^{\text{h}}$ is set to 0).

After defining the variables and the domains (all {0,1}) we need to define our constraints, which must be semantically equivalent to the constraints $C^{\text{reg}}$ and realize that each binary variable set $\{x_{i,1}^{\text{b}}, ..., x_{i,|D_i|}^{\text{b}}\}$, which represents one original variable $x_i \in X$, contains only one variable which is instantiated to 1. This guarantees that we can directly conclude a solution of the original CSP $P$ from a solution of $P^{\text{bvsc}}$. Doing this, we have for each binary variable set $\{x_{i,1}^{\text{b}}, ..., x_{i,|D_i|}^{\text{b}}\}$ a scalar constraint, which set the number of 1s in $\{x_{i,1}^{\text{b}}, ..., x_{i,|D_i|}^{\text{b}}\}$ to one (see Eq. 1).

$$\forall i \in \{1, ..., n\} : scalar(\{x_{i,1}^{\text{b}}, ..., x_{i,|D_i|}^{\text{b}}\}, \{1, ..., 1\}, =, 1) \tag{1}$$

Analogously, we have constraints for each set of variables $x^{\mathrm{h}}_{i,j}$, which represents each one level $i$ of the DAG $M_h$. The path of each solution of a DAG contains only one state $q^{\mathrm{h}}_{i,j}$ of each level $i$. Thus, we can define the following *scalar* constraints (see Eq. 2).

$$\forall i \in \{0, ..., n\}, h \in \{1, ..., |C^{\mathrm{reg}}|\} : scalar(\{x^{\mathrm{h}}_{i,*}\}, \{1, ..., 1\}, =, 1) \tag{2}$$

To represent a DAG $M_h$, we need to describe the paths of the solutions of the DAG. The initial state $q^{\mathrm{h}}_{0,0}$ and the final state $q^{\mathrm{h}}_{n,0}$ are mandatory parts of a solution path. Thus, the variables $x^{\mathrm{h}}_{0,0}$ and $x^{\mathrm{h}}_{n,0}$ must be set to 1. A state $q^{\mathrm{h}}_{i,j_3}$ of a DAG $M_h$ is part of a solution path, if one of its predecessor states $q^{\mathrm{h}}_{i-1,j_1}$ and one of the values $d_{i-1,j_2} \in D_{i-1}$ of the edge between this both states is part of the solution path. This can be represented by the logical formula $q_{i-1,j_1} \wedge d_{i-1,j_2} \rightarrow q^{\mathrm{h}}_{i,j_3}$. See Fig. 1 for some example path relations.

Applied to our variable system it concludes the formula $x^{\mathrm{h}}_{i-1,j_1} \wedge x^{\mathrm{b}}_{i-1,j_2} \rightarrow x^{\mathrm{h}}_{i,j_3}$. Furthermore, this logical formula can be represented as *scalar* constraint (see Eq. 3).

$$\begin{gathered} \forall i \in \{1, ..., n\}, h \in \{1, ..., |C^{\mathrm{reg}}|\}, j_1, j_2, j_3, \\ \text{where } q^{\mathrm{h}}_{i-1,j_1} \xrightarrow{d_{i-1,j_2}} q^{\mathrm{h}}_{i,j_3} \text{is an edge in } M_h : \\ scalar(\{x^{\mathrm{h}}_{i-1,j_1}, x^{\mathrm{b}}_{i-1,j_2}, x^{\mathrm{h}}_{i,j_3}\}, \{1, 1, -2\}, \leq, 1) \end{gathered} \tag{3}$$

The equivalence of the statement $x^{\mathrm{h}}_{i-1,j_1} \wedge x^{\mathrm{b}}_{i-1,j_2} \rightarrow x^{\mathrm{h}}_{i,j_3}$ with the statement in Eq. 3 can be easily proved with the truth table shown in Table 1. The result of the logical formula is always true if the scalar product is smaller or equal to one, otherwise it is false.

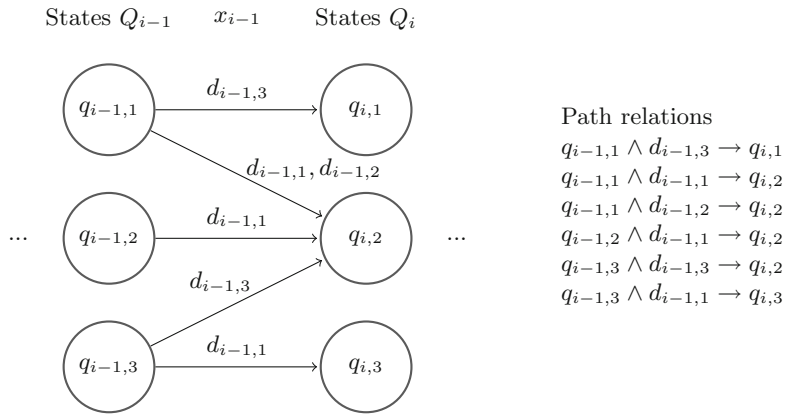


**Fig. 1.** Example path relations for an excerpt of a DAG.

**Table 1.** The truth table to show the equivalence of $x^{\mathrm{h}}_{i-1,j_1} \wedge x^{\mathrm{b}}_{i-1,j_2} \rightarrow x^{\mathrm{h}}_{i,j_3}$ and $scalar(\{x^{\mathrm{h}}_{i-1,j_1}, x^{\mathrm{b}}_{i-1,j_2}, x^{\mathrm{h}}_{i,j_3}\}, \{1, 1, -2\}, \leq, 1)$.

| $x^{\mathrm{h}}_{i-1,j_1} \wedge x^{\mathrm{b}}_{i-1,j_2} \rightarrow x^{\mathrm{h}}_{i,j_3}$ | | $\Leftrightarrow$ | $1 * x^{\mathrm{h}}_{i-1,j_1} + 1 * x^{\mathrm{b}}_{i-1,j_2} - 2 * x^{\mathrm{h}}_{i,j_3} \leq 1$ | |
|---|---|---|---|---|
| $True \wedge True \rightarrow True$ | True | $\Leftrightarrow$ | $0 \leq 1$ | $1 * 1 + 1 * 1 - 2 * 1 \leq 1$ |
| $True \wedge True \rightarrow False$ | False | $\Leftrightarrow$ | $2 \leq 1$ | $1 * 1 + 1 * 1 - 2 * 0 \leq 1$ |
| $True \wedge False \rightarrow True$ | True | $\Leftrightarrow$ | $-1 \leq 1$ | $1 * 1 + 1 * 0 - 2 * 1 \leq 1$ |
| $True \wedge False \rightarrow False$ | True | $\Leftrightarrow$ | $1 \leq 1$ | $1 * 1 + 1 * 0 - 2 * 0 \leq 1$ |
| $False \wedge True \rightarrow True$ | True | $\Leftrightarrow$ | $-1 \leq 1$ | $1 * 0 + 1 * 1 - 2 * 1 \leq 1$ |
| $False \wedge True \rightarrow False$ | True | $\Leftrightarrow$ | $1 \leq 1$ | $1 * 0 + 1 * 1 - 2 * 0 \leq 1$ |
| $False \wedge False \rightarrow True$ | True | $\Leftrightarrow$ | $-2 \leq 1$ | $1 * 0 + 1 * 0 - 2 * 1 \leq 1$ |
| $False \wedge False \rightarrow False$ | True | $\Leftrightarrow$ | $0 \leq 1$ | $1 * 0 + 1 * 0 - 2 * 0 \leq 1$ |

Thus, we have everything we need to represent the regular CSP $P^{\mathrm{reg}} = (X, D, C^{\mathrm{reg}})$ as a binary variable scalar CSP $P^{\mathrm{bvsc}} = (X^{\mathrm{b}}, D^{\mathrm{b}}, C^{\mathrm{sc}})$, with

$$
\begin{aligned}
X^{\mathrm{b}} &= \{x^{\mathrm{b}}_{i,j} \mid \forall i \in \{1, ..., n\}, j \in \{1, ..., |D_i|\}\} \\
&\quad \cup \{x^{\mathrm{h}}_{i,j} \mid \forall h \in \{1, ..., |C^{\mathrm{reg}}|\}, i \in \{0, ..., n\}, j \in \{1, ..., |Q^{\mathrm{h}}_i|\}\} \\
D^{\mathrm{b}} &= \{D_i = \{0, 1\} \mid \forall i \in \{1, ..., |X^{\mathrm{b}}|\}\} \\
C^{\mathrm{sc}} &= \{scalar(\{x^{\mathrm{b}}_{i,1}, ..., x^{\mathrm{b}}_{i,|D_i|}\}, \{1, ..., 1\}, =, 1) \mid \forall i \in \{1, ..., n\}\} \\
&\quad \cup \{scalar(\{x^{\mathrm{h}}_{i,*}\}, \{1, ..., 1\}, =, 1) \mid \forall i \in \{0, ..., n\}, h \in \{1, ..., |C^{\mathrm{reg}}|\}\} \\
&\quad \cup \{scalar(\{x^{\mathrm{h}}_{i-1,j_1}, x^{\mathrm{b}}_{i-1,j_2}, x^{\mathrm{h}}_{i,j_3}\}, \{1, 1, -2\}, \leq, 1)\} \mid \\
&\qquad \forall i \in \{1, ..., n\}, h \in \{1, ..., |C^{\mathrm{reg}}|\}, j_1, j_2, j_3, where \\
&\qquad q^{\mathrm{h}}_{i-1,j_1} \xrightarrow{d_{i-1,j_2}} q^{\mathrm{h}}_{i,j_3} \text{ is an edge in } \mathrm{M_h}\}
\end{aligned}
$$

*Remark 2:* The constraints in Eq. 1 and 2 are the reason, why we use scalar constraints instead of logic constraints, because such *count* constraints are just not compactly representable as logic constraints, but very simple as scalar constraints.

*Remark 3:* In the implementation we did some optimizations like merging different scalar constraints, which represent path relations between the same states $(q_{i-1,j_1}, q_{i,j_3})$, but with $k$ different values $(d_{i-1,j_{2,1}}, ..., d_{i-1,j_{2,k}})$ to one single scalar constraint.

## 5 Experimental Results

For demonstrating the benefit of fd-complete constraints and fd-complete CSPs, we use a rostering problem close to the presented one in Section 4.1 in [9]. Briefly summarized, consider the rostering problem four shifts (0 = day off, 1 = early,

2 = late, 3 = night shift), $m$ employees and $w$ weeks. Let $n$ be the number of days ($n = w * 7$). The goal is it to find a valid shift assignment for all employees, such that the following four restrictions are satisfied:

**$R_1$:** At each Saturday is the same shift as at the following Sunday.
**$R_2$:** A forward rotating system is used, so the following shift combinations are not allowed for two days in a row: (late, early), (night, early), (night, late).
**$R_3$:** There is a minimum number of two and a maximum number of four (for night shifts 3) consecutive days, with the same shift.
**$R_4$:** Between $\lfloor \frac{w}{4} \rfloor$ and $\lfloor \frac{w}{4} \rfloor + 1$ employees are needed for each shift and day.

The given restrictions cover some but not all German laws for shift planning, but it is possible to extend the problem for individual use cases. The difference to [9] is that we changed in restriction $R_4$ the upper bound from $\lceil \frac{w}{4} \rceil$ to $\lfloor \frac{w}{4} \rfloor + 1$ to guarantee that every problem has at least one solution.

As typical for many rostering problems, we just consider the plan of one person $P_1$ and assume that the plan for further staff is received by rotating $P_1$s plan by one week, two weeks, etc. For example given a shift plan $sol_{P_1} = (v_1, v_2, \ldots, v_n)$ for person $P_1$, the plan for a person $P_2$ would be $(v_8, v_9, \ldots, v_n, v_1, ..., v_7)$, for a person $P_3$ would be $(v_{15}, v_{16}, \ldots, v_n, v_1, ..., v_{14})$, and so on.

The following naive CSP $P_{naive}$ describes the given rostering problem. The constraints in $C_1$ satisfy the restriction $R_1$, the constraints in $C_2$ satisfy the restriction $R_2$, the constraints in $C_3, C_4$ and $C_5$ satisfy the restriction $R_3$ and the constraints in $C_6$ and $C_7$ satisfy the restriction $R_4$ (Fig. 2).

$$
\begin{aligned}
X &= \{x_1, ..., x_n\} \\
D &= \{D_1 = ... = D_n = \{0, 1, 2, 3\}\} \\
C &= C_1 = \{x_{i+6} = x_{i+7} \mid \forall i \in \{1, ..., w\}\} \\
&\quad \cup\, C_2 = \{(x_i, x_{i+1}) \notin \{(2,1), (3,1), (3,2)\} \mid \forall i \in \{1, ..., n\}\} \\
&\quad \cup\, C_3 = \{(x_i == x_{i+1}) \vee (x_i == x_{i-1})\} \mid \forall i \in \{2, ..., n-1\}\} \\
&\quad \cup\, C_4 = \{allEqual(x_i, x_{i+1}, x_{i+2}, x_{i+3}) \rightarrow (x_i \neq x_{i+4}) \mid \forall i \in \{1, ..., n-4\}\} \\
&\quad \cup\, C_5 = \{allEqual(x_i, x_{i+1}, x_{i+2}, 3) \rightarrow (x_i \neq x_{i+3}) \mid \forall i \in \{1, ..., n-3\}\} \\
&\quad \cup\, C_6 = \{\textstyle\sum_{i=1}^{w} x_{i+d} \geq \lfloor \frac{w}{4} \rfloor \mid \forall d \in \{1, ..., 7\}\} \\
&\quad \cup\, C_7 = \{\textstyle\sum_{i=1}^{w} x_{i+d} \leq \lfloor \frac{w}{4} \rfloor + 1 \mid \forall d \in \{1, ..., 7\}\}
\end{aligned}
$$

**Fig. 2.** The naive CSP $P_{naive}$, which describes the given rostering problem.

For our benchmark suite we computed different instances of the rostering problem with number of weeks respectively number of employees is equal to (4, 5, 6, 7 and 8) in different versions:

1. *Naive*: The CSP is directly solved as it is modeled in $P_{naive}$.
2. *Regular*: Each constraint was substituted by a regular version of it as described in [11].

3. *RegularIntersected*: Each constraint was substituted by a regular version of it as described in [11] and the regular versions of the constraints $C_1$ to $C_5$ are combined to one regular constraint.
4. *BVSC*: The CSP $P_{naive}$ was transformed into a binary variable scalar constraint CSP $P^{\text{bvsc}}$ as described above.
5. *BVSCO*: The binary variable scalar constraint CSP $P^{\text{bvsco}}$ was created and optimized by hand.

All the experiments are set up on a DELL laptop with an Intel i7-4610M CPU, 3.00 GHz, with 16 GB 1600 MHz DDR3 and running under Windows 7 Professional with Service Pack 1. The algorithms are implemented in Java under JDK version 1.8.0_191 and Choco Solver version 4.0.4 [15]. We used the *DowOverWDeg* search strategy which is explained in [3] and used as default search strategy in the Choco Solver [15].

**Table 2.** A comparison of the solution process of the given rostering problem with different sizes and different modelling versions. (*Time limit was reached.)

| Problem | #Solutions | *Naive* | *BVSC* | *BVSCO* | *Regular* | *RegularIntersected* |
|---|---|---|---|---|---|---|
| 4 weeks | 44 | 100% | 100% | 100% | 100% | 100% |
| | | 0.726 s | 5.815 s | 0.155 s | 2.139 s | 0.371 s |
| 5 weeks | 217339 | 100% | 100% | 100% | 100% | 100% |
| | | 96.622 s | 345.377 s | 8.87 s | 59.599 s | 5.697 s |
| 6 weeks | 9443633 | 17% | 2% | 100% | 34% | 100% |
| | | 600 s* | 600 s* | 396.619 s | 600 s* | 196.055 s |
| 7 weeks | 8463303 | 9% | 0.1% | 84% | 28% | 100% |
| | | 600 s* | 600 s* | 600 s* | 600 s* | 214.486 s |
| 8 weeks | 42979 | 11% | 0.002% | 93% | 76% | 100% |
| | | 600 s* | 600 s* | 600 s* | 600 s* | 25.649 s |

Table 2 shows a comparison of the different modelling versions. For four and five weeks, all approaches found all solutions (100%). The *RegularIntersected* and the *BVSCO* approach are the fastest with distance to the other approaches. Only the *RegularIntersected* approach found all solutions for the 6, 7 and 8 week problem. It is shown how many solutions every approach found in at maximum 600 s, which was the time out. It is visible that the *RegularIntersected* approach is always the fastest and the *BVSC* approach is always the worst. But we also can see, that the by hand optimized *BVSCO* is also always very good, much better than the original approach. In these experiments we did not use reduction methods, which are known from equality and inequality solving. We think that in the future we can find an automatic way from the original CSP to one which is very close to the *BVSCO*, and so one that is much better than the original, naive problem. It is clearly evident, those fd-CSPs like regular CSPs or binary

variable CSPs with scalar constraints can lead to significant improvement of the solution process of constraint problems.

*Remark 4:* The transformation time to create the responsible fd-CSPs from the original CSP was always included in the calculation time in Table 2.

*Remark 5:* The results presented in Table 2 show already the power of fd-complete constraints (the *RegularIntersected* and the *BVSCO* approach are partial more than 24 times faster as the original naive approach). Further more we may can improve the solution process much more if we use a specific solver for regular CSPs or binary variable CSPs with scalar constraints instead of the general fd-solver Choco.

## 6 Conclusion and Future Work

We have presented a new class of constraints, which allows describing every other CSP with a single constraint (fd-complete constraint). We extended this approach to fd-complete CSPs, which are also able to represent each other CSP. Furthermore, we introduced an example fd-complete CSP, the binary scalar CSP, and explained why we think, that this fd-complete CSP can improve the solution speed of general CSPs. In Sect. 5 was shown that the use of fd-CSPs like regular CSPs or binary variable CSPs with scalar constraints can improve the solution process of CSPs in a significant way.

Future work includes the creation of specialist constraint solvers, specialist search strategies and specialist constraint solver settings. We expect that there are a big number of fd-complete constraints and fd-complete CSPs and, therefore, also a high number of specialist constraint solvers and search strategies, which need to be discovered. Moreover, we need an algorithm to decide into which fd-CSP we should transform given problems.

## References

1. Google LLC: Google OR-Tools (2019). https://developers.google.com/optimization/. Accessed 22 Nov 2019
2. Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P., Salamon, A.Z.: Automatic discovery and exploitation of promising subproblems for tabulation. In: Hooker, J.N. (ed.) CP 2018. LNCS, vol. 11008, pp. 3–12. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98334-9_1
3. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: de Mántaras, R.L., Saitta, L. (eds.) Proceedings of the 16th European Conference on Artificial Intelligence (ECAI 2004), including Prestigious Applicants of Intelligent Systems (PAIS 2004), Valencia, Spain, 22–27 August 2004, pp. 146–150. IOS Press (2004)
4. Dechter, R.: Constraint Processing. Elsevier Morgan Kaufmann, Burlington (2003)
5. Hellsten, L., Pesant, G., van Beek, P.: A domain consistency algorithm for the stretch constraint. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 290–304. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30201-8_23

6. van Hoeve, W.J., Katriel, I.: Global constraints. In: [16], 1st edn. (2006). (chapter 6)
7. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Boston (1979)
8. Löffler, S., Liu, K., Hofstedt, P.: The power of regular constraints in CSPS. In: 47. Jahrestagung der Gesellschaft für Informatik (Informatik 2017), Chemnitz, Germany, 25–29 September 2017, pp. 603–614 (2017). https://doi.org/10.18420/in2017_57
9. Löffler, S., Liu, K., Hofstedt, P.: The regularization of CSPs for rostering, planning and resource management problems. In: Iliadis, L., Maglogiannis, I., Plagianakos, V. (eds.) AIAI 2018. IAICT, vol. 519, pp. 209–218. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92007-8_18
10. Löffler, S., Liu, K., Hofstedt, P.: A meta constraint satisfaction optimization problem for the optimization of regular constraint satisfaction problems. In: Rocha, A.P., Steels, L., van den Herik, H.J. (eds.) Proceedings of the 11th International Conference on Agents and Artificial Intelligence (ICAART 2019), Prague, Czech Republic, 19–21 February 2019, vol. 2, pp. 435–442. SciTePress (2019). https://doi.org/10.5220/0007260204350442
11. Löffler, S., Liu, K., Hofstedt, P.: The regularization of small sub-constraint satisfaction problems. CoRR abs/1908.05907 (2019). http://arxiv.org/abs/1908.05907
12. Marriott, K.: Programming with Constraints - An Introduction. MIT Press, Cambridge (1998)
13. Pesant, G.: A filtering algorithm for the stretch constraint. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 183–195. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45578-7_13
14. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 482–495. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30201-8_36
15. Prud'homme, C., Fages, J.G., Lorca, X.: Choco documentation. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. (2019). http://www.choco-solver.org/. Accessed 07 Nov 2019
16. Rossi, F., Beek, P.V., Walsh, T.: Handbook of Constraint Programming, 1st edn. Elsevier, Amsterdam (2006)
17. Schulte, C., Lagerkvist, M., Tack, G.: Gecode 6.2.0 (2019). https://www.gecode.org/. Accessed 22 Nov 2019 (2019)
18. Tack, G., Stuckey, P.J.: Minizinc 2.3.2. Monash University (2019). https://www.minizinc.org/. Accessed 22 Nov 2019 (2019)