

Reformulating Table Constraints using Functional Dependencies – An Application to Explanation Generation

Hadrien Cambazard and Barry O’Sullivan

Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
{h.cambazard|b.osullivan}@4c.ucc.ie

Abstract. We present a novel approach to automatically reformulating constraints defined as tables of allowed assignments to variables. Constraints of this form are common in a variety of settings. Specifically, we propose an approach in which a high arity table constraint is reformulated as a conjunction of lower arity constraints. The reformulation is logically equivalent to the original constraint. We demonstrate that by using functional dependencies from the field of database design such reformulations can be found. We apply the approach to the problem of generating explanations as minimal conflicts. We show that reformulations can be found that yield compact explanations of inconsistency by reducing both the number of variables required to explain inconsistency and the arity of the largest constraint involved in the explanation. We demonstrate our approach on real-world datasets with positive results.

1 Introduction

Constraint satisfaction techniques are ubiquitous in many practical problem-solving contexts [21]. The fundamental notion in constraint satisfaction is to separate the representation of a problem from the method used to solve it [8]. As the number of interactive systems built upon constraints technology increases, several issues become a concern. For example, how should we integrate information sources that are more naturally represented as tables, or simple databases, with more traditional constraint-based methods? Also, how can we support the generation of concise explanations when a set of constraints cannot be satisfied?

There are many contexts when explanations are required, particularly ones that are concise in the sense that they involve few variables and low arity constraints. For example, it is common in user-focused interactive applications to allow users to add unary constraints, i.e. assignments to variables, to a set of background constraints that define the general problem being solved. When such a set of constraints cannot be satisfied, one can compute a set-wise maximal subset of the user’s choices that are consistent, or a set-wise minimal subset of his constraints that are inconsistent. These are the standard notions of maximal (minimal) relaxation (conflict) that have been used in the artificial intelligence community [7, 12]. Similarly, when a knowledge-base is defined in terms of

constraints, it may contain conflicts that incorrectly forbid solutions. By using test-cases, e.g. possibly (partial) solutions that should be consistent with the knowledge-base, minimal conflicts can be very helpful in identifying those problematic constraints. In these conflicts, large table constraints may be presented in an explanation, but if they have high arity it can be difficult to identify the precise cause. It is, therefore, interesting to consider reformulating these constraints into an equivalent set of constraints of lower arity. The expectation is that minimal conflicts will only contain those interesting lower arity constraints, thus helping the knowledge engineer identify the source of the problem more effectively. Finally, in the most general setting where we wish to compute the minimal unsatisfiable core of a constraint problem, reformulating large arity table constraints can help find a more compact core involving fewer variables.

In the context of more general arity constraints, computing a minimal conflict can be more challenging. However, again, there is a significant literature on finding minimal conflicting sets of constraints, sometimes referred to as minimal unsatisfiable subproblems [17]. The duality between conflicting sets of constraints and relaxations is also well known [2, 6, 12, 20].

The *objective* of the research presented in this paper is to answer the following question: how do we compute *concise* explanations when we have problems involving constraints of very high arity, i.e. defined over many variables. We regard an explanation as being concise if it involves constraints of low arity over a small subset of the variables of the problem. In particular, we consider the case of positive table constraints [15] which are defined using tables of consistent assignments to a set of variables.

Our *approach* is to apply techniques used to normalise relational databases to automatically reformulate a large arity constraint into a conjunction of lower arity constraints. We can guarantee that the set of solutions to the reformulation is equivalent to that of the original table constraint. Since the reformulation involves constraints of lower arity, more concise explanations can be found. We *demonstrate* these characteristics on several real-world datasets.

The *contributions* of this paper are as follows:

1. We present a well-founded approach to automatically reformulating positively defined table constraints using techniques from database design.
2. We show how to compute a *best* reformulation in which, for example, the maximum constraint arity is minimised.
3. We prove a variety of properties of our approach, including: the equivalence of the amount of pruning on the reformulation and the original table constraint; the complexity class of finding a reformulation with a given maximum arity; and a uniqueness property on the reformulation for a given valid set of functional dependencies that yields an efficient algorithm to find the best reformulation.
4. We empirically demonstrate that our approach can find compact reformulations of real-world table constraints, and that it can help find more compact explanations. We also show that it is competitive with a proof-based method for computing explanations.

The remainder of the paper is organised as follows. In Section 2 we present the formal background required throughout the paper. We demonstrate the utility of constraint reformulation for generating explanations in Section 3. We present our approach to constraint reformulation in Sections 4 and 5. An empirical evaluation is presented in Section 6. A number of concluding remarks along with a summary of some interesting directions for future work are presented in Section 7.

2 Background

A constraint satisfaction problem (CSP) is defined by a set of variables, each of which must be assigned a value from its domain, subject to a set of constraints. Each constraint restricts the set of consistent assignments to a subset of the variables. We define the CSP formally as follows.

Definition 1 (Constraint Satisfaction Problem). *A constraint satisfaction problem \mathcal{P} is a triple $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where: $\mathcal{X} = \{x_1, \dots, x_{nv}\}$ is a finite set of variables; $\mathcal{D} = \{D(x_1), \dots, D(x_{nv})\}$ is a set of domains corresponding to the possible values of each variable; and $\mathcal{C} = \{c_1, \dots, c_{nc}\}$ is a set of constraints. Each constraint $c_i \in \mathcal{C}$ is defined by a scope and a relation. The scope denoted $\text{scope}(c_i)$ is an ordered subset of \mathcal{X} and the relation, $\text{rel}(c_i)$, is a set of tuples over $\text{scope}(c_i)$ that satisfy the constraint c_i . The number of variables constrained by c_i , i.e. $|\text{scope}(c_i)|$, is known as the arity of the constraint c_i .*

Definition 2 (Solution to a CSP). *Given a CSP $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ and given any constraint $c_i \in \mathcal{C}$, an assignment t to the variables in $\text{scope}(c_i)$ satisfies c_i if $t \in \text{rel}(c_i)$. We denote as $t[x_i]$ the value $v \in D(x_i)$ assigned to x_i in t . An assignment t of \mathcal{X} is a solution of \mathcal{P} if for every $c_i \in \mathcal{C}$, the projection of t to $\text{scope}(c_i)$ satisfies c_i . We denote as $\text{sol}(\mathcal{P})$ the set of all solutions to \mathcal{P} .*

Example 1 (A Simple CSP). We demonstrate the definitions given above on a simple example CSP in which $\mathcal{X} = \{x_1, x_2, x_3, x_4\}$, $\mathcal{D} = \{D(x_1) = \{0, 1, 2\}, D(x_2) = \{0, 2, 4\}, D(x_3) = \{0, 1, 2, 3\}, D(x_4) = \{2, 3, 4\}\}$, and the constraints \mathcal{C} are:

$$\begin{aligned} c_1(x_1, x_3) &\stackrel{\text{def}}{=} \{(0, 0), (1, 0), (2, 1), (0, 2), (2, 3)\}, \\ c_2(x_1, x_4) &\stackrel{\text{def}}{=} \{(0, 4), (1, 2), (2, 3), (2, 2)\}, \\ c_3(x_2, x_3) &\stackrel{\text{def}}{=} \{(0, 0), (4, 1), (4, 2), (2, 3)\}, \\ c_4(x_2, x_4) &\stackrel{\text{def}}{=} \{(0, 4), (0, 2), (4, 3), (4, 4), (2, 2)\}. \end{aligned}$$

The constraints scopes are $\{x_1, x_3\}$, $\{x_1, x_4\}$, $\{x_2, x_3\}$, and $\{x_2, x_4\}$, and the relations are the sets specified in the right-hand sides above. Using the natural subscript ordering for the variables in \mathcal{X} , the set of solutions to our example CSP are:

$$\{(0, 0, 0, 4), (0, 4, 2, 4), (1, 0, 0, 2), (2, 2, 3, 2), (2, 4, 1, 3)\}.$$

▲

In this paper we are interested in automatically computing *equivalent* reformulations of large arity table constraints. It is useful, therefore, to formally define the meaning of equivalence between CSPs.

Definition 3 (Equivalent CSPs). *Given a CSP $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ and a CSP $\mathcal{P}' \stackrel{\text{def}}{=} \langle \mathcal{X}, \mathcal{D}, \mathcal{C}' \rangle$ we say that \mathcal{P} is equivalent to \mathcal{P}' if both problems have equivalent sets of solutions, i.e. $\text{sol}(\mathcal{P}) \equiv \text{sol}(\mathcal{P}')$.*

When we reformulate a high arity constraint c we obtain a conjunction of lower arity constraints, c^1, \dots, c^k , where the scope of each c^j is a subset of $\text{scope}(c)$. To define the notion of restricting a tuple of allowed values to a subscope of the original constraint we use the projection operator from Codd's Relational Algebra.

Definition 4 (Scope Restriction [10]). *Let r be a relation over a set of variables Y and t , a tuple of r . The projection onto $Z \subseteq Y$ of t , denoted $t[Z]$, is the restriction of t to Z . The projection of r onto Z , denoted $\sigma_Z(r)$, is the set $\{t[Z] \mid t \in r\}$.*

Scope restriction is a form of constraint relaxation. Informally, by restricting the scope of a constraint we are reducing the number of variables that are constrained. Therefore, any tuple satisfying the original constraint also satisfies the relaxation.

Definition 5 (Relaxation of a Positive Table Constraint). *We say that constraint c^r with a scope $X = \text{scope}(c^r)$ is a relaxation of constraint c if $X \subseteq \text{scope}(c)$ and $\text{rel}(c^r) = \sigma_X(\text{rel}(c))$, i.e. the set of allowed tuples of the relaxation c^r are obtained by projecting the set of allowed tuples of c onto the set of variables constrained by c^r . The constraint c^r corresponding to the relaxation of c on X is also denoted $\Pi_X(c)$.*

Based on the notion of constraint relaxation, we can regard a reformulation of a constraint as a subset of its (irredundant) relaxations.

Definition 6 (Constraint Reformulation). *A reformulation $\Delta(c)$ of a positive table constraint c is a set of relaxations $\mathcal{R} \stackrel{\text{def}}{=} \{c^1, \dots, c^k\}$ of c such that $\forall c^a, c^b \in \mathcal{R}, a \neq b, \text{scope}(c^a) \not\subseteq \text{scope}(c^b)$.*

The notion of reformulation can be extended to scopes; a *scope reformulation* in the following is intended as a set of subsets of scopes defining the relaxations involved. As a notational convenience, we often denote a reformulation of a constraint c by its scope reformulation.

Example 2 (Constraint Reformulation). Consider the following constraint, over the same variables and domains as those in Example 1:

$$c_a(x_1, x_2, x_3, x_4) \stackrel{\text{def}}{=} \{(0, 0, 0, 4), (0, 4, 2, 4), (1, 0, 0, 2), (2, 2, 3, 2), (2, 4, 1, 3)\}.$$

A reformulation of this constraint, in terms of two relaxations of c_a , is as follows:

$$\begin{aligned} c_a^1(x_1, x_2, x_4) &\stackrel{\text{def}}{=} \{(0, 0, 4), (1, 0, 2), (2, 4, 3), (0, 4, 4), (2, 2, 2)\}, \\ c_a^2(x_1, x_3) &\stackrel{\text{def}}{=} \{(0, 0), (0, 2), (1, 0), (2, 1), (2, 3)\}. \end{aligned}$$

We will refer to this reformulation as $\Delta(c_a) = \{\{x_1, x_2, x_4\}, \{x_1, x_3\}\}$. Note that the set of solutions to the reformulation is not equivalent to those of c_a , since the tuple $(0, 4, 0, 4)$ is allowed by this reformulation. However, note that the set of constraints in Example 1 forms an equivalent reformulation of c_a . \blacktriangle

Reformulations that give rise to conjunctions of constraints that are equivalent, i.e. have the same set of solutions, are very important. We refer to such reformulations as *lossless*, in keeping with the standard terminology in databases¹.

Definition 7 (Lossless Reformulation). *Given a CSP $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{X}, \mathcal{D}, \{c\} \rangle$ involving a single constraint c . $\Delta(c)$ is a lossless reformulation of c if the CSP $\mathcal{P}' \stackrel{\text{def}}{=} \langle \mathcal{X}, \mathcal{D}, \Delta(c) \rangle$ is such that $\text{sol}(\mathcal{P}) \equiv \text{sol}(\mathcal{P}')$.*

We will focus on such lossless reformulations in the remainder of this paper.

3 An Application of Lossless Constraint Reformulation

In many real-world problems we have to deal with large arity table constraints. For example, in the well-known Renault Megane Car Configuration Problem [1], all constraints are represented as extensional table constraints, each defined by a set of allowed tuples of assignments. The constraints in this problem involve up to ten variables. When a set of constraints does not admit a solution, a common form of explanation is to generate a minimal conflicting set of constraints, i.e. a set of constraints that is inconsistent, but all proper subsets of this set are consistent. When our problem formulation contains many high arity constraints, such an explanation may not be concise. However, a lossless reformulation may exist that gives a very concise explanation.

Example 3 (Explanations from Lossless Reformulations). Consider the following set of constraints, each of arity four:

$$\begin{aligned} c_1(x_1, x_3, x_4, x_6) &\stackrel{\text{def}}{=} \{(2, 0, 0, 1), (1, 1, 1, 2), (2, 0, 2, 1), (1, 1, 0, 0)\}, \\ c_2(x_1, x_2, x_5, x_7) &\stackrel{\text{def}}{=} \{(0, 1, 0, 1), (1, 2, 2, 0), (1, 0, 1, 0), (2, 1, 0, 2)\}, \\ c_3(x_5, x_2, x_3, x_4) &\stackrel{\text{def}}{=} \{(1, 0, 2, 2), (2, 1, 1, 2), (0, 2, 0, 1), (2, 1, 2, 1)\}. \end{aligned}$$

This set of constraints does not admit a solution. A minimal conflict set that is sufficient to explain this inconsistency involves all three constraints. This is

¹ Note that in our context by selecting a reformulation that is not lossless the reformulation admits solutions that are inconsistent with the original constraint. In this setting we have “lost” information ruling out some combinations of values.

because by removing any of the constraints, the problem becomes consistent and a solution can be found.

However, a lossless reformulation provides a much more compact explanation. It can be shown that each of the constraints above is equivalent to the corresponding conjunctions of constraints given below:

$$\begin{aligned} c_1(x_1, x_3, x_4, x_6) &\equiv \{c_1^1(x_1, x_3), c_1^2(x_1, x_6), c_1^3(x_4, x_6)\}, \\ c_2(x_1, x_2, x_5, x_7) &\equiv \{c_2^1(x_1, x_7), c_2^2(x_5, x_2), c_2^3(x_1, x_2)\}, \\ c_3(x_5, x_2, x_3, x_4) &\equiv \{c_3^1(x_2, x_5), c_3^2(x_2, x_3, x_4)\}. \end{aligned}$$

On this reformulation, the inconsistency can be more compactly explained by showing the following set of constraints from the reformulation:

$$\{c_1^1(x_1, x_3), c_3^2(x_2, x_3, x_4), c_2^3(x_1, x_2)\}.$$

In fact, an even more compact explanation can be found since in c_3^2 , x_4 is not required to explain the inconsistency. However no lossless reformulations can highlight this fact². \blacktriangle

We now present how lossless reformulations can be computed automatically.

4 Reformulation of Positive Table Constraints

In contrast with earlier work, e.g. [10], our approach to computing lossless reformulations of positive table constraints exploits the concept of functional dependencies in a relation [11]. A *functional dependency* in a relation $rel(c)$ is written as $\mathcal{F} : X \rightarrow y$, where $X \cup \{y\} \subseteq scope(c)$. A functional dependency states that if a pair of tuples in the relation take the same values for the variables in X , they must also take the same value for variable y .

A functional dependency $\mathcal{F} : X \rightarrow y$ is minimal if y is not functionally dependent on any subset of X . It is said to be *trivial* if $y \in X$. Algorithms for finding the set of all minimal and non-trivial dependencies that hold in a given relation are known [11].

Example 4 (Functional Dependencies in Constraint Relations). Consider constraint c_a from Example 2, presented below:

$$c_a(x_1, x_2, x_3, x_4) \stackrel{\text{def}}{=} \{(0, 0, 0, 4), (0, 4, 2, 4), (1, 0, 0, 2), (2, 2, 3, 2), (2, 4, 1, 3)\}.$$

The following minimal functional dependencies (among the seven that exist for this relation) hold in c_a : $\mathcal{F}_1 : \{x_3\} \rightarrow x_2$, $\mathcal{F}_2 : \{x_1, x_2\} \rightarrow x_3$, and $\mathcal{F}_3 : \{x_1, x_2\} \rightarrow x_4$. The values of x_2 are uniquely determined by the value of x_3 and the values of x_4 and x_3 depend, similarly, only on the values taken by x_1 and x_2 . The dependency $\{x_2\} \rightarrow x_3$ does not hold because value 4 of x_2 does not determine the value of x_3 (2 or 1). \blacktriangle

² The approach mentioned in Section 6.2, based on proof-based explanation, could eventually find the conflict $\{c_1^1(x_1, x_3), c_3^2(x_2, x_3), c_2^3(x_1, x_2)\}$.

Based on a set of functional dependencies that hold on the relation of a constraint, we define a lossless reformulation of the constraint into a conjunction of constraints as follows.

Definition 8 (Constraint Reformulation using a Functional Dependency).

Let c be a positive table constraint, $\mathcal{F} : X \rightarrow y$ a functional dependency that holds on $rel(c)$ such that $X \cup \{y\} \subset scope(c)$. The reformulation of c , denoted $\Delta(c, \mathcal{F})$, using \mathcal{F} is defined by:

$$\Delta(c, \mathcal{F}) = \{\Pi_{scope(c)-\{y\}}(c), \Pi_{X \cup \{y\}}(c)\}.$$

Given a positively defined table constraint c and a functional dependency $\mathcal{F} : X \rightarrow y$ holding on $rel(c)$, constraint c can be decomposed into a pair of constraints. The first constraint is defined over the scope $scope(c) - \{y\}$, while the second constraint is defined over the scope $X \cup \{y\}$. Informally, a functional dependency allows us to split the scope of a constraint by eliminating the functionally dependant variable y . The notion of reformulation is again extended to scopes, with $\Delta(scope(c), \mathcal{F}) = \{scope(c) - \{y\}, X \cup \{y\}\}$.

Example 5 (Constraint Reformulation using a Functional Dependency). Consider again the constraint c_a and the functional dependencies presented in Example 4. The original scope of c_a is (x_1, x_2, x_3, x_4) . If we apply $\mathcal{F}_3 : \{x_1, x_2\} \rightarrow x_4$, this scope is split into (x_1, x_2, x_3) and (x_1, x_2, x_4) , in accordance with the procedure above. If we apply $\mathcal{F}_1 : \{x_3\} \rightarrow x_2$ on the scope (x_1, x_2, x_3) we can split this into (x_1, x_3) , (x_2, x_3) and the resulting lossless reformulation of c_a is made of (x_1, x_3) , (x_2, x_3) and (x_1, x_2, x_4) . \blacktriangle

Since we perform constraint reformulation using functional dependencies the following theorem immediately follows.

Theorem 1 (Lossless Reformulation). *The reformulation $\Delta(c, \mathcal{F})$ of constraint c using the functional dependency \mathcal{F} holding on $rel(c)$ is lossless.*

Proof. Follows from the definition of functional dependency. \blacksquare

While it is of paramount importance that decomposing a constraint into a conjunction of lower arity constraints does not unsoundly relax the problem, we may also be concerned with how constraint propagation is affected. Fortunately, we can show that achieving generalised arc consistency (GAC) [5, 15, 16] on the reformulation is equivalent to achieving GAC on the original constraint.

Theorem 2 (Constraint Propagation on the Reformulation). *Let $\Delta(c, \mathcal{F})$ be a reformulation of a constraint c using the functional dependency $\mathcal{F} : X \rightarrow y$ holding on $rel(c)$. Achieving GAC on each $c_i \in \Delta(c, \mathcal{F})$ is equivalent to achieving GAC on c .*

Proof. We show that given a variable $x \in scope(c)$, different from y , and a value from the domain of x , i.e. $v \in D(x)$, a pair (x, v) has a support in c if and only if it has a support in $c^1 = \Pi_{X \cup \{y\}}(c)$ and $c^2 = \Pi_{scope(c)-\{y\}}(c)$. In a similar way

we show that a pair (y, v) has a support in c if and only if it has one in c^1 . There are, thus, two cases to consider: (a) where $x \neq y$, and (b) where $x = y$. We first consider the case where $x \neq y$:

- \Rightarrow Let t be the support of (x, v) in c . Then, (x, v) is supported in both $t[X \cup \{y\}]$ and $t[V - \{y\}]$ in c^1 and c^2 , respectively.
- \Leftarrow Let t_2 be a support of (x, v) in c^2 then $t_1 = t_2[X]$ is a valid tuple of c^1 and there is only one possible value for $t_1[y]$ because y is determined by X . Thus, the tuple t such that $t[\text{scope}(c) - y] = t_2$ and $t[y] = t_1[y]$ is a valid tuple of c and supports (x, v) .

In the case where $x = y$, similar reasoning holds and for any support t in c of (y, v) , $t[X \cup \{y\}]$ is a support in c^1 and a support in c can be built similarly from a support in c^1 . \blacksquare

Our reformulation strategy is essentially the same as that used in database design to decompose relations into a normal form called Boyce-Codd Normal Form (BCNF). BCNF ensures a lossless reformulation, but does not preserve the dependencies (some dependencies that hold in the original scope may no longer be found in the resulting reformulation because their scope has been split by the use of another dependency). A relation is in BCNF if the left-hand side of all remaining functional dependencies determine every other attribute of the relation. However, we can often decompose relations further. Specifically, in this paper we focus on *minimal* reformulations³.

Definition 9 (Minimal Reformulation). *Given a constraint c , the reformulation $\Delta(c, \delta)$ obtained from a set of dependencies δ holding on $\text{rel}(c)$ is said to be minimal if each element of $\Delta(c, \delta)$ cannot be decomposed further and there does not exist a reformulation $\Delta'(c, \delta) \subset \Delta(c, \delta)$ such that $\Delta'(c, \delta)$ is lossless.*

Example 6 (Minimal Reformulation). Consider the constraint scope $c(x_1, x_2, x_3, x_4)$ on which $\mathcal{F}_1 : \{x_1, x_4\} \rightarrow x_3$, $\mathcal{F}_2 : \{x_1\} \rightarrow x_2$ and $\mathcal{F}_3 : \{x_3\} \rightarrow x_1$ hold. We use these functional dependencies in the order $\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3$, to produce the following sequence of reformulations:

- $\Delta_1(c, \mathcal{F}_1) = \{\{x_1, x_4, x_3\}, \{x_1, x_2, x_4\}\};$
- $\Delta_2(c, \{\mathcal{F}_1, \mathcal{F}_2\}) = \{\{x_1, x_4, x_3\}, \{x_1, x_2\}, \{x_1, x_4\}\};$
- $\Delta_3(c, \{\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3\}) = \{\{x_1, x_3\}, \{x_3, x_4\}, \{x_1, x_2\}, \{x_1, x_4\}\}.$

The final reformulation, $\Delta_3(c, \{\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3\})$ is not minimal. In $\Delta_2(c, \{\mathcal{F}_1, \mathcal{F}_2\})$, the scope $\{x_1, x_4, x_3\}$ contains the scope $\{x_1, x_4\}$ (the first constraint implies the other), therefore the latter can be removed while still ensuring that the resultant reformulation is lossless. In fact, the decomposition of $\{x_1, x_4, x_3\}$ that follows is itself lossless. A minimal decomposition would, therefore, be $\{\{x_1, x_3\}, \{x_3, x_4\}, \{x_1, x_2\}\}$. \blacktriangle

³ The notation $\Delta(c, \mathcal{F})$ is extended to sets of dependencies and $\Delta(c, \delta)$ refers to the reformulation obtained from a set of dependencies δ .

Notice that a minimal reformulation is in BCNF, but a BCNF might not be minimal. Also, it is sometimes possible to decompose relations beyond what is possible using functional dependencies alone, while still obtaining a lossless reformulation. However, Theorem 2 is no longer guaranteed to hold, so constraint propagation can be adversely affected. We will illustrate the point on two constraints introduced previously in this paper and presented in Table 1.

Table 1. Two examples of table constraints used in previous examples.

c_a				c_3			
x_1	x_2	x_3	x_4	x_1	x_2	x_3	x_4
0	0	0	4	1	0	2	2
1	0	0	2	2	1	1	2
2	4	1	3	0	2	0	1
0	4	2	4	2	1	2	1
2	2	3	2				

Consider the following reformulation of constraint c_a given in Example 5:

$$\Delta(c_a, \{\mathcal{F}_3, \mathcal{F}_1\}) = \{\{x_3, x_2\}, \{x_1, x_3\}, \{x_1, x_2, x_4\}\}.$$

At first glance, no further reformulation can be performed on the constraint defined over scope $\{x_1, x_2, x_4\}$ because no single variable determines any other among $\{x_1, x_2, x_4\}$. However the following reformulation is lossless:

$$\Delta^*(c_a) = \{\{x_3, x_2\}, \{x_1, x_3\}, \{x_1, x_4\}, \{x_2, x_4\}\}.$$

To see the reason that this reformulation is lossless, we will consider the relation of this constraint in more detail. For a relation $rel(c)$ over a set of variables $X \cup \{y\}$, we denote by $support(y, t[X])$ the set of values in $D(y)$ that can be used to extend t to a tuple in $rel(c)$. These values are called the *supports* of t in $D(y)$. We can decompose the subscope $\{x_1, x_2, x_4\}$ into $\{\{x_1, x_4\}, \{x_2, x_4\}\}$ because, firstly, the functional dependency $\mathcal{F} : \{x_1, x_2\} \rightarrow x_4$ holds in this relation and, secondly, the following property holds: $\forall v_1 \in D(x_1), v_2 \in D(x_2) : |support(x_4, x_1 = v_1) \cap support(x_4, x_2 = v_2)| = 1$. For example with $v_1 = 2$ and $v_2 = 4$ we get $\{3, 2\} \cap \{3, 4\} = \{3\}$. x_1 and x_2 determine x_4 *independently* of each other and do not need to be considered together. This is a much stronger property than that imposed by a functional dependency.

We now consider c_3 , with the two functional dependencies $\mathcal{F}_a : \{x_2, x_4\} \rightarrow x_3$ and $\mathcal{F}_b : \{x_2\} \rightarrow x_1$. Notice that, in this case, $support(x_3, x_2 = 1) = \{1, 2\}$ and $support(x_3, x_4 = 2) = \{2, 1\}$, so that $|\{1, 2\} \cap \{2, 1\}| > 1$. Therefore, x_2, x_4 *cannot independently* determine the value of x_3 but need to be considered together. The reformulation, therefore, remains $\Delta(c_3, \{\mathcal{F}_b\}) = \{\{x_2, x_1\}, \{x_2, x_3, x_4\}\}$. It is impossible to find a lossless reformulation of c_3 into binary constraints.

However, while we could decompose c_a into a set of binary constraints, achieving arc consistency on these is not equivalent to achieving GAC on c_a , even

though the reformulation is lossless. This is not surprising of course. Consider the example of $\text{PERMUTATION}(x_1, x_2, x_3)$ ($\text{ALLDIFFERENT}(x_1, x_2, x_3)$ where each variable has a domain of size 3). Functional dependencies alone cannot decompose this constraint into binary constraints, but it can be decomposed into $x_1 \neq x_2$, $x_1 \neq x_3$ and $x_2 \neq x_3$ using $\mathcal{F} : \{x_1, x_2\} \rightarrow x_3$ and the fact that $\forall v_1 \in D(x_1), v_2 \in D(x_2) : |\text{supports}(x_3, x_1 = v_1) \cap \text{supports}(x_3, x_2 = v_2)| = 1$. Such a reformulation is lossless, but it is also well known that it does not achieve GAC, unlike the ALLDIFFERENT global constraint [18].

Finally, note that the worst-case complexity of enforcing GAC on a reformulation $\Delta(c)$ with $|\Delta(c)| = k$ might be better than on the original constraint. The complexity of GAC relies on the maximum number of tuples involved in any constraint relation, $s = \max_{c_i \in \Delta(c)} (|\text{rel}(c_i)|)$, or the maximum arity, $a = \max_{c_i \in \Delta(c)} (|\text{scope}(c_i)|)$, depending on the GAC scheme used [5, 16]. However it is easy to see that s is always equal to $|\text{rel}(c)|$ in what remains of the initial relation. By using a dependency, a small table can be extracted but the number of tuples of the relation from which the variable is removed remains unchanged. This does not mean that the overall reformulation cannot be more compact. Specifically its memory size, computed as the number of entries in the tables $ms = \sum_{c_i \in \Delta(c)} (|\text{rel}(c_i)| \times |\text{scope}(c_i)|)$, can be smaller than $|\text{rel}(c)| \times |\text{scope}(c)|$; a and ms therefore provide a formal basis to characterise a *good* reformulation in terms of space and time complexity. More specifically, the time complexity of achieving GAC on the reformulation is either $\mathcal{O}(k \times a^2 \times d^a)$ or $\mathcal{O}(k \times a^2 \times d \times s)$, where a constraint check can be performed with cost equal to the arity, a , of the constraint. Moreover the complexity of the most recent algorithm for positive table constraints relies on both s and a [15]. In the following, we will focus on the problem of finding the reformulation where a is minimal since it matches our objective to find a reformulation that guarantees that more compact explanations of inconsistency can be found.

5 Finding Optimal Reformulations

The typical objective in reformulation is to find an equivalent formulation of a problem instance that is more efficient to solve than the original. In this paper we have a different objective, namely to reformulate in order to generate more compact explanations of inconsistency. In this section we will consider the problem of how to find an optimal reformulation of a problem using functional dependencies. The objective function here is to minimise the largest arity of the constraints in our reformulation. In Section 5.1 we will consider the notion of a valid reformulation sequence, since the order in which we apply functional dependencies is important. In Section 5.2 we prove the complexity class of finding a reformulation of a problem in which the maximum arity is bounded. In Section 5.3 we will present an approach to finding the reformulation with the minimum largest arity.

5.1 Valid Reformulation Sequences

A reformulation is obtained by applying a sequence of functional dependencies. However, as dependencies are applied, others may no longer be applicable to the reformulation we obtain. For example, in Example 5 we did not apply functional dependency $\mathcal{F}_2 : \{x_1, x_2\} \rightarrow x_3$. This is because having applied \mathcal{F}_1 and \mathcal{F}_3 it was no longer applicable, since no scope in our reformulation could be decomposed with it. Two dependencies cannot be applied in the same scope if they have, for example, the same right-hand side. Others have to be applied in a given order. For two dependencies $\mathcal{F}_i : X_i \rightarrow y_i$ and $\mathcal{F}_j : X_j \rightarrow y_j$ we will say that $\mathcal{F}_i \subset \mathcal{F}_j$ if and only if $X_i \cup \{y_i\} \subset X_j \cup \{y_j\}$.

Theorem 3 (Valid Ordering of Functional Dependencies). *Given a constraint c , let $\mathcal{F}_i : X_i \rightarrow y_i$ and $\mathcal{F}_j : X_j \rightarrow y_j$ be two minimal functional dependencies that hold in $\text{rel}(c)$ such that $y_j \in X_i \cup \{y_i\}$ and $\mathcal{F}_i \not\subset \mathcal{F}_j$ (i.e. $X_i \cup \{y_i\} \not\subset X_j \cup \{y_j\}$). Then, when \mathcal{F}_i and \mathcal{F}_j are applied on the same scope, \mathcal{F}_i can only be applied before \mathcal{F}_j , which we denote as $\mathcal{F}_i \prec \mathcal{F}_j$.*

Proof. Suppose that \mathcal{F}_i and \mathcal{F}_j are applied on the same scope for example $\text{scope}(c)$. Applying the functional dependency \mathcal{F}_j would give rise to two new constraints with the scopes $X_j \cup \{y_j\}$ and $\text{scope}(c) - \{y_j\}$ (by Definition 8). $X_i \cup \{y_i\}$ can neither be part of $\text{scope}(c) - \{y_j\}$ if $y_j \in X_i \cup \{y_i\}$ nor part of $X_j \cup \{y_j\}$ since $\mathcal{F}_i \not\subset \mathcal{F}_j$. Therefore, \mathcal{F}_i can no longer apply. ■

Given a set of dependencies δ , we denote by G_δ the directed graph of precedences between the dependencies in δ . The nodes of G_δ are the dependencies in δ . An edge $(\mathcal{F}_i, \mathcal{F}_j)$ is added if $\mathcal{F}_i \prec \mathcal{F}_j$, i.e. \mathcal{F}_i can only be applied before \mathcal{F}_j . For the sake of simplicity, we will denote by $\delta(S)$ the subset of δ applying on scope S : $\delta(S) = \{\mathcal{F}_i \in \delta \mid X_i \cup \{y_i\} \subseteq S\}$. To fully characterize a set of dependencies that can be used to give rise to a reformulation, we define the *root* of a set δ and a scope S as $\text{root}(\delta, S) = \{\mathcal{F}_i \in \delta(S) \mid \forall \mathcal{F}_j \in \delta(S) : \mathcal{F}_i \not\subset \mathcal{F}_j\}$, i.e. the subset of δ that applies to scope S and where no dependency is included in another.

Definition 10 (Valid Set of Dependencies). *A set of functional dependencies $\delta^* \subseteq \delta$ holding on a scope S is said to be valid if and only if $G_{\text{root}(\delta^*, S)}$ is acyclic and $\forall \mathcal{F}_i : X_i \rightarrow y_i \in \delta^*$, $G_{\text{root}(\delta^*, X_i \cup \{y_i\})}$ is acyclic.*

Example 7 (Valid Set of Dependencies). Consider the set of dependencies $\delta = \{\mathcal{F}_1, \dots, \mathcal{F}_4\}$ and their corresponding precedence graph presented in Figure 1. The set of functional dependencies $\delta^* = \{\mathcal{F}_1, \mathcal{F}_3, \mathcal{F}_4\}$ is a valid set, since it can be verified that $\text{root}(\delta^*, S) = \{\mathcal{F}_1, \mathcal{F}_4\}$, $\text{root}(\{\mathcal{F}_1\}, \{x_1, x_2, x_3\}) = \{\mathcal{F}_3\}$, $\text{root}(\{\mathcal{F}_4\}, \{x_1, x_4\}) = \emptyset$ and $\text{root}(\{\mathcal{F}_3\}, \{x_3, x_2\}) = \emptyset$ all correspond to acyclic subgraphs of G_δ . ▲

All dependencies in a valid set δ can be used to decompose the original scope by applying them in an order compatible with the precedences of G_δ . Moreover, the reformulation associated with the valid sequence of δ , using all elements of δ , is unique. This is a keystone result for the reformulation algorithm we propose, and we prove another observation related to the minimality of dependencies to help understand it more easily.

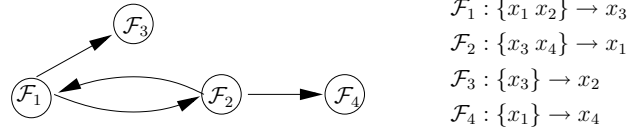


Fig. 1. An example of a graph of precedences.

Lemma 1. Let \mathcal{F}_i and \mathcal{F}_j be two minimal functional dependencies such that $\mathcal{F}_i \subseteq \mathcal{F}_j$, i.e. that $X_i \cup \{y_i\} \subseteq X_j \cup \{y_j\}$. Then $y_j \in X_i \cup \{y_i\}$.

Proof. If $y_j \notin X_i \cup \{y_i\}$ it means that $X_i \cup \{y_i\} \subseteq X_j$ and, therefore, that \mathcal{F}_j is not a minimal functional dependency because there is another dependency, namely \mathcal{F}_i , contained inside its left-hand side. ■

Consider, for example, the minimal dependency $\mathcal{F} : \{x_1, x_2, x_3\} \rightarrow y$, the only way to further decompose the scope obtained after applying this dependency is with a functional dependency of the form $\{y\} \rightarrow x_1$ or $\{x_2, y\} \rightarrow x_3$ but y is mandatory on the left-hand side.

Theorem 4 (Unique Minimal Reformulation). *The minimal reformulation obtained after applying a valid set of minimal functional dependencies δ on a scope S is unique and the dependencies of δ can be used only once.*

Proof. Let δ^* be the root set of δ and initial scope S , i.e. such that $\delta^* = \text{root}(\delta, S) = \{\mathcal{F}_i \in \delta \mid \forall \mathcal{F}_j \in \delta : \mathcal{F}_i \not\prec \mathcal{F}_j\}$. Let $k = |\delta^*|$, the number of functional dependencies in δ^* . The reformulation obtained using δ^* is unique. Since δ^* is acyclic (being the root of δ), the set of all y_i variables of the dependencies $\mathcal{F}_i : X_i \rightarrow y_i$ of δ^* are different; if \mathcal{F}_i and \mathcal{F}_j are such that $y_i = y_j$ then we would have both $\mathcal{F}_i \prec \mathcal{F}_j$ and $\mathcal{F}_j \prec \mathcal{F}_i$ (having $y_i \in X_j \cup \{y_j\}$ and $y_j \in X_i \cup \{y_i\}$ in Theorem 3) so δ^* would be cyclic and δ would not be valid. All $\{y_1, \dots, y_k\}$ are different, therefore, each \mathcal{F}_i can only be applied on the main scope S . The reformulation obtained by applying dependencies of δ^* in an order respecting their precedences is made of either $k + 1$ or k new scopes:

$$\left\{ \begin{array}{l} S - \{y_1, \dots, y_k\} \\ X_1 \cup \{y_1\} \\ \dots \\ X_k \cup \{y_k\} \end{array} \right\} \quad \text{or} \quad \left\{ \begin{array}{l} X_1 \cup \{y_1\} \\ \dots \\ X_k \cup \{y_k\}. \end{array} \right.$$

The scope $S - \{y_1, \dots, y_k\}$ might be removed if it is included in one of the $X_i \cup \{y_i\}$ scopes, in order to ensure that the reformulation we obtain is minimal. This does not affect the rest of the reformulation as no other dependencies hold on this scope (all other dependencies involve a y_i because of minimality of dependencies) and the reformulation of the scope that contains it will be lossless. The resulting reformulation after this first step is, therefore, unique and involves k or $k + 1$ new scopes.

Now consider a dependency $\mathcal{F}_i \in \delta - \delta^*$, \mathcal{F}_i has precedences with at least one dependency of δ^* and it holds on a single scope of the previous reformulation.

Indeed, there exists an \mathcal{F}_j of δ^* such that $\mathcal{F}_i \subset \mathcal{F}_j$ and this \mathcal{F}_j is unique because $y_j \in X_i \cup \{y_i\}$ (Lemma 1) and all $\{y_1, \dots, y_k\}$ are different.

The remaining dependencies are, therefore, partitioned amongst the previous scopes and this partition is also unique. (Notice also that $S - \{y_1, \dots, y_k\}$ cannot, therefore, be decomposed further as all other dependencies must involve one of the y_j variables.) This will result in potentially k new unique reformulations, by the same reasoning, showing that the overall reformulation of S by δ is unique and that every dependency can be used only once in the process. ■

Theorem 4 is a key result that underlies the reformulation algorithm and its $O(2^n)$ complexity. The reformulation of a valid set can be quickly computed because any order of the dependencies that respects the precedence graph produces the same result. One can, therefore, simply search for valid sets amongst the subsets of the original dependencies for the best reformulation without considering all possible sequences which would give an $O(n!)$ complexity.

A reformulation from a valid set δ of dependencies holding on an initial scope S is computed using Algorithm 1. Line 4 computes the root set of the dependencies holding on the current scope. This scope is then removed from the current reformulation (line 6) and all the new scopes obtained from δ^* are added to the reformulation (lines 7-8). Lines 9-10 ensure that the algorithm returns a minimal reformulation.

Algorithm 1 : DECOMPOSE($\delta = \{\mathcal{F}_1, \dots, \mathcal{F}_n\}, S = \{x_1, \dots, x_r\}$)

1. $DS \leftarrow \{S\}$;
 2. **While** $\delta \neq \emptyset$ **Do**
 3. **For each** $S_k \in DS$ **Do**
 4. $\delta^* \leftarrow \text{root}(\delta, S_k)$;
 5. **If** $\delta^* \neq \emptyset$
 6. $DS \leftarrow DS - S_k$; $c \leftarrow |\delta^*|$;
 7. **For each** $\mathcal{F}_i \in \delta^*$ **Do**
 8. $DS \leftarrow DS \cup \{X_i \cup \{y_i\}\}$;
 9. **If** $\nexists P \in DS, \{S_k - \{y_1, \dots, y_c\}\} \subseteq P$;
 10. $DS \leftarrow DS \cup \{S_k - \{y_1, \dots, y_c\}\}$;
 11. $\delta \leftarrow \delta - \delta^*$;
 12. **Return** DS ;
-

5.2 Complexity of Arity-Bounded Reformulation

Given a set of functional dependencies, we seek to use them to find a reformulation in which the maximum arity of the constraints is minimised. This problem can be shown to be NP-Hard, since the corresponding decision problem can be used to solve the Feedback Vertex Set problem, which is known to be NP-Complete. We define the basic decision problem as follows.

BOUNDED MAX-ARITY REFORMULATION

Instance: A set δ of minimal functional dependencies holding on a scope $S = (x_1, \dots, x_m)$, and an integer $1 \leq b \leq m$.

Question: Does there exist a subset $\delta_b \subseteq \delta$ with $a_{\Delta(S, \delta_b)} \leq b$ where $a_{\Delta(S, \delta_b)} = \max_{S_i \in \Delta(S, \delta_b)} |S_i|$ is the maximum arity of the scopes in the reformulation obtained using δ_b ?

Our complexity proof is based on a reduction from the FEEDBACK VERTEX SET, which is known to be NP-Complete [9].

FEEDBACK VERTEX SET

Instance: A directed graph $G = (V, E)$ and a positive integer $k \leq |V|$.

Question: Does there exist an $X \subseteq V$ with $|X| \leq k$ such that G with the vertices from X removed is acyclic?

Theorem 5. BOUNDED MAX-ARITY REFORMULATION *problem is NP-Complete.*

Proof. Clearly this problem is in NP. Given a valid set of functional dependencies, the maximum arity of the resultant unique reformulation can be computed in polynomial time (Theorem 4).

Consider an instance of the FEEDBACK VERTEX SET on a graph $G = (V, E)$ with $|V| = n$. We construct an instance of the BOUNDED MAX-ARITY REFORMULATION problem in the following way. A functional dependency $\mathcal{F}_k : \{\dots\} \rightarrow v_k$ is associated with each node of V . Then each v_k is instantiated to x_k and added to the left-hand side of all dependencies corresponding to the predecessors of v_k in G . Moreover, we add n new variables from $\{x_{n+1}, \dots, x_{2n}\}$, one to each left-hand side all functional dependencies. Figure 2 shows the construction resulting from the two previous steps on an example graph G . The size of these dependencies is at most $n + 1$ (a complete graph), they respect the precedences of G , they are minimal and can only apply on the main scope, i.e. they are not included in one another. Let δ be such a set of dependencies.

We then introduce as many variables as needed to ensure that the largest arity is always found on the main scope of any reformulation; this way, the two objective functions match perfectly and removing the fewest number of nodes to achieve acyclicity gives rise to the reformulation with the smallest maximum arity. The set of variables becomes $\{x_1, \dots, x_{2n}, \dots, x_{2n+2}\}$, so that $m = 2n + 2$. Finally we choose $b = m - (n - k)$.

The FEEDBACK VERTEX SET has a solution by removing at most k nodes if and only if there is a reformulation whose maximum arity is less than b :

\Rightarrow Let's assume that we have a solution of the FEEDBACK VERTEX SET with $|X| \leq k$. Then the remaining $n - k$ nodes define a set of dependencies without cycles. $n - k$ variables are removed from the main scope $\{x_1, \dots, x_{2n}, \dots, x_{2n+2}\}$ leading to an arity of $2n + 2 - (n - k) = n + 2 + k = b$. In the best case ($k = 0$) the arity is of size $n + 2$ which proves that it is the maximum arity because the size of the dependencies are at most $n + 1$ and thus $a_{\Delta(S, V-X)} \leq b$.

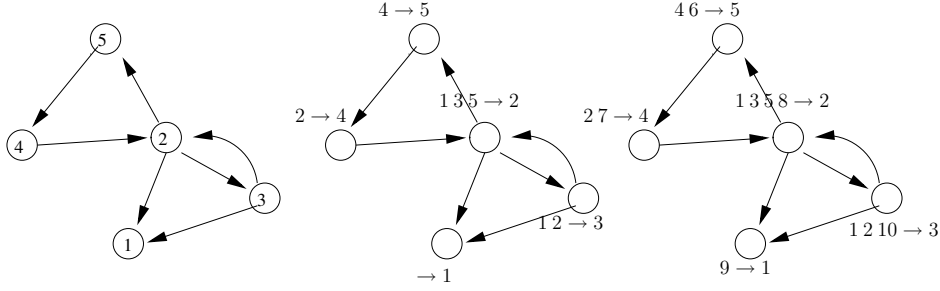


Fig. 2. An example set of dependencies built from a given graph.

⇐ A solution of BOUNDED MAX-ARITY REFORMULATION is an acyclic subset δ_b of δ such that $a_{\Delta(S, \delta_b)} \leq b$. It follows that at least $m - b$ variables have been removed from the main scope which means that at most $n - (m - b)$ corresponding nodes X have been removed from the graph and $|X| \leq n - (m - b) = k$. ■

Notice, finally, that for any set of minimal functional dependencies there always exists a relation in which only these dependencies hold (as well as all dependencies logically implied by them). Such relations are called Armstrong relations, and a proof of their existence can be found in [3]. The time complexity to generate an Armstrong relation is exponential in the number of functional dependencies required to hold in it. Fortunately, we do not need to build this relation. Our only assumption is that the set of functional dependencies is given explicitly, since the problem of finding all minimal functional dependencies is itself NP-Hard.

5.3 Computing the Reformulation with Smallest Maximum Arity

We describe a simple algorithm for computing the reformulation with the smallest maximum arity (Algorithm 2). We consider an initial scope S and a set of minimal dependencies δ that hold on S with $|\delta| = n$. The algorithm is, basically, a branch and bound over the subsets of δ . At each node it deals with the current partial set of dependencies (called PD) and the set of current candidate dependencies (called CD) to be included (or not) in PD . Algorithm 2 is invoked by calling $\text{BESTDECOMP}(\delta, \emptyset, S)$.

The following procedures are applied :

- if CD is empty, we compute the unique reformulation using Algorithm 1 and, therefore, its maximum arity (Line 3) and update the best known solution if needed. Notice that at this point PD is a valid set of dependencies.
- Otherwise, the method $\text{PRUNING}(CD, PD \cup \{\mathcal{F}_i\})$ removes all functional dependencies \mathcal{F}_j of CD such that $PD \cup \{\mathcal{F}_i\} \cup \{\mathcal{F}_j\}$ would not be a valid set (if a set of dependencies that have to be in the same root set contains a cycle).
- If the bound obtained for the new pair $(CD - \mathcal{F}_j, PD)$ is compatible with the best known solution, the algorithm branches.

Algorithm 2 BESTDECOMP($CD = \{\mathcal{F}_i, \dots, \mathcal{F}_n\}, PD, S$)

1. **If** $CD = \emptyset$
2. $DS \leftarrow \text{DECOMPOSE}(PD, S)$;
3. $\text{maxArity} \leftarrow \max_{S_k \in DS} |S_k|$;
4. **If** $\text{maxArity} < \text{bestKnownMaxArity}$
5. $\text{bestKnownMaxArity} \leftarrow \text{maxArity}$;
6. store DS as the new best known solution;
7. **Else**
8. $CD \leftarrow CD - \{\mathcal{F}_i\}$;
9. $FD \leftarrow \text{PRUNING}(CD, PD \cup \{\mathcal{F}_i\})$;
10. **If** $\text{BOUND}(CD - FD, PD \cup \{\mathcal{F}_i\}, S) < \text{bestKnownMaxArity}$
11. BESTDECOMP($CD - FD, PD \cup \{\mathcal{F}_i\}, S$);
12. **If** $\text{BOUND}(CD, PD, S) < \text{bestKnownMaxArity}$
13. BESTDECOMP(CD, PD, S);

We turn our attention to the bounding procedure (Algorithm 3). The following simple bound considers each dependency \mathcal{F}_i of PD separately and computes a lower bound for the maximum arity of the reformulation resulting from its use. One can bound the best way to reduce the arity of the scope S_i of \mathcal{F}_i by bounding the maximum number of dependencies that can actually be applied on S_i (lines 6). Indeed, each dependency decreases the arity of S_i by one and the biggest remaining scope after reformulation is often what remains from S_i itself. Line 8 takes the initial scope into account and perform the same reasoning. We use a naive lower bound for the size of the minimum vertex feedback set which involves partitioning the graph into cliques $P = \{C_1 \dots C_k\}$. In each clique, all nodes except one must be removed to break all the cycles. Any partition P , therefore, gives a lower bound as $\sum_{C_i \in P} (|C_i| - 1)$.

Algorithm 3 BOUND(CD, PD, S)

1. $lb \leftarrow 0$;
2. $AD \leftarrow CD \cup PD$;
3. **For** $\mathcal{F}_i \in PD$ **Do**
4. $S_i \leftarrow X_i \cup \{y_i\}$;
5. $\delta_i \leftarrow \{\mathcal{F}_j \in AD \mid \mathcal{F}_j \subset \mathcal{F}_i\}$;
6. $lbi \leftarrow |S_i| - (|\delta_i| - \text{lower bound on the min. vertex feedback set in } G_{\delta_i})$;
7. $lb \leftarrow \max(lb, lbi)$;
8. $lbs \leftarrow |S| - (|AD| - \text{lower bound on the min. vertex set feedback in } G_{AD})$;
9. **Return** $\max(lb, lbs)$;

Many other bounds could be designed, but our focus here is to show that the main complexity of this algorithm is $\mathcal{O}(2^n)$ and not $\mathcal{O}(n!)$. Further investigations remain to be carried out to improve this algorithm and, especially, the bounds.

6 Experiments

The objective of our experimental evaluation was twofold⁴. Firstly, we wished to evaluate the extent to which functional dependencies could be used to effectively reformulate real-world table constraints in order to reduce the maximum arity of the constraints in the reformulation. Secondly, we sought to show that using a reformulation of table constraints based on functional dependencies more compact minimal conflict explanations could be found and how they compare with those computed using a proof-based explanation technique [13]. We used a well-known approach to generating proof-based explanations to determine the subset of the variables in the problem that were used to compute a proof of insolubility. From this set of variables we computed the set of constraints in the corresponding explanation by projecting the original problem constraints onto the set of variables used in the proof. Our experiments achieved each of these objectives and support our claim that exploiting functional dependencies in constraint relations is a promising approach to automated reformulation of table constraints.

We have used a well-known library, called TANE [11], to compute the set of functional dependencies in a constraint relation. Four datasets were used in our experiments:

1. From the Renault Megane Car Configuration Problem [1] we used the two largest table constraints (R80 and R140). R80 involves 10 variables and contains 342 tuples, while R140 involves 9 variables and contains 164 tuples.
2. We used a dataset of digital cameras, defined using 11 variables and 112 tuples [19].
3. We used a dataset of laptops, defined using 13 variables and 403 tuples [19].
4. We used the AI-CBR travel case-base, defined using 9 variables and 1470 tuples [14].

For each dataset we removed any field that gave a unique identifier to each tuple, since these would have introduced many “artificial” dependencies. Specifically, this resulted in the arities of the digital camera and laptop datasets being reduced to 8 and 10 variables, respectively. All other datasets remained unchanged.

6.1 Experiment 1: Reformulation of Real-World Constraints

The objective of this experiment was to evaluate the extent to which a functional dependency-based reformulation could be used to reformulate the table constraints in each of our datasets. This experiment involved computing the set of all functional dependencies, and computing a reformulation with the smallest maximum arity. We summarise our results in Table 2. We report the number of tuples and arity of the initial tables. We also present the number of functional dependencies found by TANE, from which we find the best reformulation, i.e. the one that gives a reformulation whose maximum arity is smallest. For each

⁴ Experiments were implemented using Choco (<http://choco-solver.net>), and run on a MacBook with 2Gb of RAM and a dual core 2Ghz processor.

dataset we report the number of constraints, minimum and maximum arity, and the time (in seconds) taken to compute the optimal reformulation⁵.

Table 2. Lossless reformulations of the table constraints in our dataset.

Dataset	#tuples	arity	#dependencies	#constraints	min.arity	max.arity	time(s)
camera R0	112	8	41	4	4	5	0.051
laptop R0	403	10	54	6	5	6	0.022
renault R80	342	10	2	3	2	8	0.002
renault R104	164	9	11	6	2	4	0.025
travel R0	1470	9	7	4	3	6	0.004

There are several very positive results in this table. Firstly, the optimal reformulation can be found extremely quickly. Secondly, it is possible to find reformulations in which the maximum arity is significantly reduced, e.g. by 55% in the case of the Renault R104 constraint. Thirdly, it is interesting to see that the reformulations found can sometimes contain constraints of very low arity. Specifically, for both constraints from the Renault configuration problem we were able to find an equivalent reformulation involving some binary constraints.

6.2 Experiment 2: Comparing Compact Explanations

The objective of this experiment was to measure the compactness of the explanations of inconsistency found for each dataset using the original constraint, as well as a reformulation with the smallest maximum arity and those found using a proof-based technique. In each case we computed an explanation as a set of constraints that formed a conflict in the problem. It should be noted, however, that the proof-based explanations are not guaranteed to be minimal, and are composed of relaxations of the constraints. We include a comparison with the proof-based approach because it is the only other approach able to deal with large arities, and it is informative to see how it compares against the minimal explanations found on the reformulation. The key measurements taken in this experiment were the number of variables, along with the arity of the constraints involved in the explanations of inconsistency. A more compact explanation involves fewer variables and constraints of lower arity. We used the same datasets as before, but in the case of the Renault tables we combined both together in the same problem, giving us four scenarios in total.

The approach we adopted was, for each dataset, to add a set of randomly generated binary constraints to force inconsistency with a given (set of) table constraint(s), but are consistent without the table constraint. Therefore, in the case where the table constraints in our datasets were used in their original form, the table constraints were involved in all explanations of inconsistency. However,

⁵ This does not include the time needed by TANE to extract the dependencies, which was always less than 0.1 seconds on these tables.

using the reformulation, it was often possible to find a much more compact explanation, as our results illustrate.

To generate a variety of classes of inconsistent problems for which we computed explanations, we varied both the density of constraints to be combined with the constraints in our dataset, and their tightness. The *density of a set of constraints* is the percentage of all possible binary constraints that can be defined on a set of variables that are present in the constraint network. The *tightness* of a constraint is the percentage of all possible tuples that can be assigned to a pair of variables that violate the constraint.

Table 3. Constraint density and tightness settings used in our experiments.

Dataset	density (%)	low tightness (%)	high tightness (%)
camera	43	75	88
laptop	27	69	90
travel	33	72	94
renault	8	42	88

For each dataset, 20 problems are generated: 10 with a low tightness and 10 with high tightness. The density and tightness of the random binary problems, therefore, varied for each table as it is related to the size of the initial scope and the initial domains of the variables in the constraint. The *low tightness* setting was chosen to be the first tightness found before the problems generated began to be inconsistent and the *high tightness* setting is the last tightness found before all problems generated are inconsistent. Table 3 summarises the parameters used to generate the set of additional binary constraints used for each dataset to generate inconsistencies.

To fairly compare the compactness of explanations found using the reformulation against that of the original constraint, for each inconsistent problem we computed the set of *all minimal conflicts* using a well-known algorithm by Bailey and Stuckey [2]. Based on the set of all possible minimal conflicts, we measured both the smallest maximum arity and the total number of variables in that best explanation (we refer to this as **best**), and the average maximum arity and average total number of variables (which we refer to as **avg**).

In Figure 3 we present our results. For each dataset we show two figures: one for the cases where inconsistency was caused by the addition of low tightness constraints, while in the other high tightness constraints were used. In each case, we plot: (a) a point at a coordinate given by the average minimum number of variables in the explanation and the corresponding average smallest maximum arity (the **best** case); and (b) a point at a coordinate given by the average number of variables in the explanation and the corresponding average arity (the **avg** case). The former represents the best case, while the latter represents the average measurements for the explanations we compute. We also plot coordinates corresponding to the original constraint in each case.

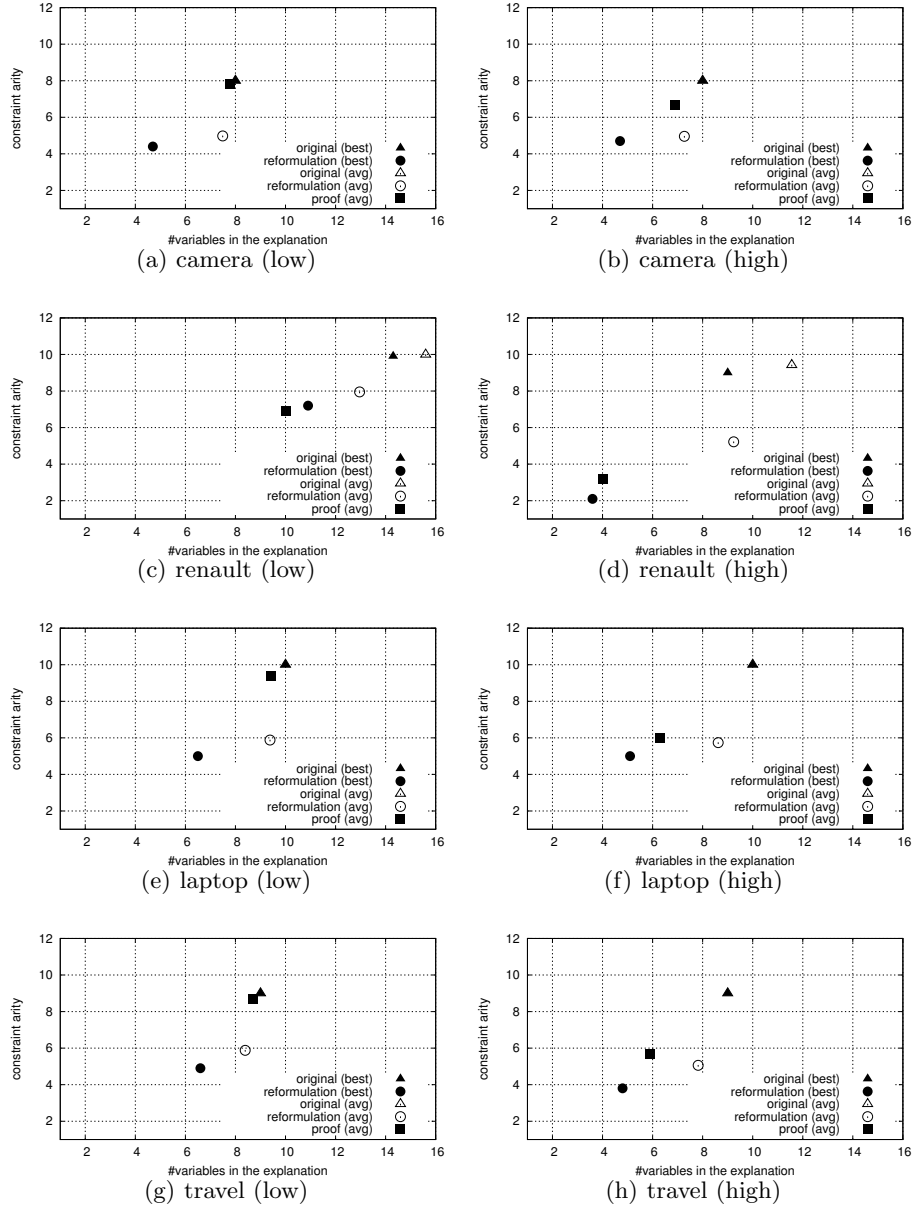


Fig. 3. Experimental results showing the compactness of the explanations found using reformulations of table constraints, as compared with the original case. Note that the **avg** and **best** values for the original constraint always coincide, except for the Renault dataset.

It is clear that, in each case, the minimal set of constraints sufficient to explain inconsistency in the case of the reformulation is always more compact than if the original table was used. It is often possible to find explanations that either reduce the arity of the largest constraint, or the number of variables involved in the explanation by almost half. When the constraints added are of high tightness, in the case of the Renault tables (Figure 3(d)), the number of variables can be reduced from nine to almost three, while the maximum arity constraint in the explanation is reduced from nine to two.

We also compared the compactness of proof-based explanations for each of the problems studied with the reformulation-based approach. The comparison was performed by considering the average arity of the constraints and the average number of variables in the explanations extracted from each insolubility proof. We present the results in Figure 3 as the data-points marked `proof (avg)`. With the exception of the results on the Renault instances, the best explanations found using the reformulation approach are more compact than those found using the proof-based approach; in the case of Renault the proof-based explanations are marginally more compact, but not convincingly so. In the average case, the arity of the constraints in explanations found from the reformulation is lower than those found using the proof-based approach. When the tightness of the user's constraints is low, both methods produce explanations in terms of a similar number of variables. However, when the tightness of these constraints is high, the proof-based method can find explanations in terms of fewer variables.

Our reformulation-based approach is based on the concept of lossless reformulation. Therefore, our explanations are not based on an unsound relaxation of the problem. The proof-based approach can find arbitrarily lossy relaxations of constraints, therefore, it can be regarded as finding compact explanations at the expense of soundness of the constraints that are used in the explanation. The fact that we find lossless explanations and remain competitive, often outperforming a proof-based method, is very encouraging.

Finally, proof-based explanations can only be generated using a constraint solver that maintains eliminating explanations. Most commercial solvers do not do this. However, our approach is based on a reformulation of the problem, so any minimal conflict generation algorithm, e.g. Junker's QUICKXPLAIN algorithm [12] which is used in a number of ILOG products⁶, can be used to generate compact explanations without any modification to the underlying solver.

7 Conclusions and Future Work

We have presented a novel approach to automatically reformulating constraints defined as a table of allowed assignments to variables. Constraints of this form are common in a variety of settings, such as product configuration or recommender systems for complex catalogues of products. We demonstrated that by using functional dependencies from the field of database design, reformulations of table

⁶ ILOG is a major supplier of optimisation technology, <http://www.ilog.com>.

constraints could be found that yield compact explanations of inconsistency by reducing both the number of variables required to explain inconsistency and the arity of the largest constraint involved in the explanation. We demonstrated our approach on real-world datasets with very positive results.

There are several extensions to this work we plan to pursue. Firstly, there are many more general forms of dependency that can be explored in the context of automated constraint reformulation. For example, approximate dependencies [11] hold if a given percentage of tuples are removed from a table. Multi-valued dependencies can also be useful [4].

Secondly, in this paper we considered optimal reformulations to be those in which the largest constraint arity in the reformulation was minimised. As discussed earlier, we can also consider the objective of minimising the total memory size of the constraints in the reformulation. When the motivation for reformulating constraints is to facilitate the generation of more compact minimal conflict explanations we prefer to minimise maximum arity.

Thirdly, while in this paper our focus was on reformulating positively defined table constraints, it is also interesting to consider the problem of reformulating negatively defined table constraints, i.e. those that are defined in terms of no-goods. In this setting, a reformulation that is not lossless corresponds to a restriction, or tightening, of the original constraint. This has interesting implications for a variety of reasons, each worthy of further study.

Finally, while we have used functional dependencies to reformulate table constraints, it is interesting to consider how they can improve search performance over a given instance. It is easy to see that functional dependencies can help partition a set of problem variables into those decision variables over which we search, and those that are set by these variables by propagation. In a setting where a specific problem instance will be solved many times over, a search heuristic that exploits functional dependencies might be very efficient. The general question of how to exploit functional dependencies during search is an interesting direction for further study.

Acknowledgements

This work was supported by Science Foundation Ireland under Grant Number 05/IN/I886. We thank Pearl Pu for providing the laptop and camera datasets. We are especially grateful to Christian Bessiere, Christophe Lecoutre and Igor Razgon for their very helpful and encouraging comments on an earlier draft of this paper. Finally, we would like to thank the reviewers for their very detailed and helpful critiques of the original version of this paper.

References

1. Jérôme Amilhastre, Hélène Fargier, and Pierre Marguis. Consistency restoration and explanations in dynamic CSPs – application to configuration. *Artif. Intell.*, 135:199–234, 2002.

2. James Bailey and Peter J. Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *Proceedings of PADL*, pages 174–186, 2005.
3. Catriel Beeri, Martin Dowd, Ronald Fagin, and Richard Statman. On the structure of armstrong relations for functional dependencies. *J. ACM*, 31(1):30–46, 1984.
4. Catriel Beeri, Ronald Fagin, and John H. Howard. A complete axiomatization for functional and multivalued dependencies in database relations. In Diane C. P. Smith, editor, *SIGMOD Conference*, pages 47–61. ACM, 1977.
5. Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks: Preliminary results. In *IJCAI (1)*, pages 398–404, 1997.
6. Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artif. Intell.*, 32(1):97–130, 1987.
7. Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.
8. Eugene C. Freuder. In pursuit of the holy grail. *ACM Computing Surveys*, 28(4):63, 1996.
9. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Series of Books in the Mathematical Sciences. W.H.Freeman, 1979.
10. Marc Gyssens, Peter Jeavons, and David A. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artif. Intell.*, 66(1):57–89, 1994.
11. Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, 42(2):100–111, 1999.
12. Ulrich Junker. QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In *AAAI*, pages 167–172, 2004.
13. Narendra Jussien. The versatility of using explanations within constraint programming. Research Report 03-04-INFO, École des Mines de Nantes, Nantes, France, 2003.
14. David B. Leake and Raja Sooriamurthi. When two case bases are better than one: Exploiting multiple case bases. In David W. Aha and Ian Watson, editors, *ICCB*, volume 2080 of *Lecture Notes in Computer Science*, pages 321–335. Springer, 2001.
15. Christophe Lecoutre and Radoslaw Szymanek. Generalized arc consistency for positive table constraints. In Frédéric Benhamou, editor, *CP*, volume 4204 of *Lecture Notes in Computer Science*, pages 284–298. Springer, 2006.
16. Olivier Lhomme and Jean-Charles Régin. A fast arc consistency algorithm for n-ary constraints. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 405–410. AAAI Press / The MIT Press, 2005.
17. Mark H. Liffiton and Kareem A. Sakallah. On finding all minimally unsatisfiable subformulas. In *SAT*, pages 173–186, 2005.
18. Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI*, pages 362–367, 1994.
19. James Reilly, Jiyong Zhang, Lorraine McGinty, Pearl Pu, and Barry Smyth. Evaluating compound critiquing recommenders: a real-user study. In *ACM Conference on Electronic Commerce*, pages 114–123, 2007.
20. Raymond Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987.
21. Mark Wallace. Practical applications of constraint programming. *Constraints*, 1(1/2):139–168, 1996.