Reformulating Positive Table Constraints using Functional Dependencies

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. Constraints that are defined by tables of allowed tuples of assignments are common in constraint programming. In this paper we present an approach to reformulating table constraints of large arity into a conjunction of lower arity constraints. Our approach exploits functional dependencies. We study the complexity of finding reformulations that either minimize the memory size or arity of a constraint using a set of its functional dependencies. We also present an algorithm to compute such reformulations. We show that our technique is complementary to existing approaches for compressing extensional constraints.

1 Introduction

Constraint satisfaction techniques are ubiquitous in many practical problem-solving contexts [20]. Constraints can either be defined intensionally, as expressions or global constraints, or extensionally as tables of allowed or forbidden tuples of values. In this paper we focus on table constraints that are defined in terms of *allowed* tuples, which are often referred to as *positive* table constraints. Most constraint toolkits, e.g. ILOG Solver and Choco, provide support for specifying such constraints. Table constraints occur "naturally" in many application domains, such as product configuration where data is available from databases, spreadsheets or catalogues [13]. Also, naive users of constraint toolkits often find it convenient to use table constraints rather than more appropriate models using global constraints and other advanced features.

While table constraints might be easy to specify, they suffer from three disadvantages from a computational point of view. Firstly, table constraints are propagated with algorithms such as GAC-SCHEMA [4] with running times that are proportional to the number of tuples allowed by the constraint, which is exponential in its arity. Secondly, when a set of table constraints is inconsistent, we may be interested in characterising the inconsistency by generating an explanation [5]. A typical form of explanation is a minimal set of conflicting constraints. Large arity table constraints do not lend themselves to giving compact explanations because they might involve too large a subset of the variables of the problem. Thirdly, the amount of space required to store a table constraint might be excessive since there can be redundant information copied many times in the tuples of the constraint. Therefore, there is considerable motivation for looking at techniques to automatically reformulate table constraints by either reducing their arity, the number of tuples that define them, or the amount of space they require.

In this paper we study such a technique that exploits functional dependencies in the constraint. In particular, we reformulate by eliminating functional dependencies and computing equivalent conjunctions of lower arity constraints. We study the complexity of the problem of finding a reformulation that either minimizes memory size or arity based on a set of functional dependencies that hold on the constraint. We also propose an algorithm for computing such a reformulation. We show that this reformulation technique is complementary to previous approaches to compactly representing table constraints [6, 8, 10, 12, 19] by capturing quite a different structure in their tuples.

2 Background

A constraint satisfaction problem $\mathcal{P}$ is a triple $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where: $\mathcal{X}$ is a finite set of variables; $\mathcal{D}$ is a set of domains corresponding to the possible values of each variable; and $\mathcal{C}$ is a set of constraints. Each constraint $c_i \in \mathcal{C}$ is defined by a scope and a relation. The scope denoted $scope(c_i)$ is an ordered subset of $\mathcal{X}$ and the relation, $rel(c_i)$, is a set of tuples over $scope(c_i)$ that satisfy the constraint. The number of variables constrained by c_i, i.e. $|scope(c_i)|$, is known as the *arity* of the constraint c_i. A solution to $\mathcal{P}$ is an assignment of all variables to a value of their domain that satisfy all the constraints of the problem. We will moreover denote by $sol(\mathcal{P})$ the set of all solutions of $\mathcal{P}$.

In order to avoid confusion between the notion of a database relation, and the relation of a constraint, we define the projection operator σ in the following. Let r be a relation over a set of variables X and t, a tuple of r. The projection onto $Z \subseteq X$ 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 $\{t[Z] | t \in r\}$. $\sigma_Z(c)$ denotes the corresponding constraint with a scope $Z \subseteq scope(c)$ and $rel(\sigma_Z(c)) = \sigma_Z(rel(c))$. $\sigma_Z(c)$ is a relaxation of c, i.e. its set of allowed tuples is obtained by projecting the set of allowed tuples of c onto Z.

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

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*.

Definition 2 (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 $sol(\mathcal{P}) = sol(\mathcal{P}')$.*

In contrast with earlier work, e.g. [9], 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. $\mathcal{F} : X \rightarrow y$ 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 1 (Functional Dependencies in Constraints). Consider the following constraint:

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

The following minimal functional dependencies (among the seven that exist) 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 for x_2 does not determine the value of x_3 (2 or 1). ▲

3 Reformulation based on dependencies

For the sake of simplicity, we will denote by S_i the scope $X_i \cup \{y_i\}$ of a dependency $\mathcal{F}_i : X_i \rightarrow y_i$. Moreover, we will say that $\mathcal{F}_i \subset \mathcal{F}_j$ if and only if $S_i \subset S_j$. Finally for a set of dependencies δ, we define $\delta(S) = \{\mathcal{F}_i \in \delta \mid S_i \subset S\}$, the restriction of δ to a scope S, *i.e* all the dependencies of δ that hold on this scope. A dependency can be used to reformulate a constraint in the following way.

Definition 3 (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: $\Delta(c, \mathcal{F}) = \{\sigma_{scope(c)-\{y\}}(c), \sigma_{X \cup \{y\}}(c)\}$.*

Informally, a functional dependency is used to split the scope of a constraint into two scopes by eliminating the functionally dependant variable y. For the sake of simplicity, the notion of reformulation is extended to scopes with $\Delta(scope(c), \mathcal{F}) = \{scope(c) - \{y\}, X \cup \{y\}\}$ and relations, $\Delta(rel(c), \mathcal{F}) = \{\sigma_{scope(c)-\{y\}}(rel(c)), \sigma_{X \cup \{y\}}(rel(c))\}$.

Example 2 (Constraint Reformulation using a Functional Dependency). Consider again the constraint c_a presented in Example 1. 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 it into (x_1, x_3), (x_2, x_3), giving the lossless reformulation: $\Delta(c_a, \langle \mathcal{F}_3, \mathcal{F}_1 \rangle) = \{\{x_3, x_2\}, \{x_1, x_3\}, \{x_1, x_2, x_4\}\}$. ▲

Our reformulation strategy is related to decomposing database relations into Boyce-Codd Normal Form (BCNF). However, we can often decompose relations further. Specifically, we focus on *minimal* reformulations.

Definition 4 (Minimal Reformulation). *Given a constraint c, the reformulation $\Delta(c, \delta)$ obtained from a set of dependencies δ holding on $rel(c)$ is said to be minimal if there does not exist a set $\delta' \subset \delta$ such that the reformulation $\Delta'(c, \delta') \subset \Delta(c, \delta)$ and $\Delta'(c, \delta')$ is lossless.*

Example 3 (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. The following sequence of reformulations is produced by $\mathcal{F}_1$, $\mathcal{F}_2$, $\mathcal{F}_3$:

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

The final reformulation, $\Delta_3(c, \langle \mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3 \rangle)$ is not minimal. In $\Delta_2(c, \langle \mathcal{F}_1, \mathcal{F}_2 \rangle)$, 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 $\{\{x_1, x_3\}, \{x_3, x_4\}, \{x_1, x_2\}\}$. ▲

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. Unfortunately, achieving generalised arc consistency (GAC) [4, 15, 16] on the reformulation is not equivalent to achieving GAC on the original constraint. Arc-consistency is preserved only in the case of dependencies between pairs of variables.

Theorem 1 (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.*

The previous theorem holds because it lead to a Berge-acyclic reformulation. It does not hold in the general case as shown on the following example :

Example 4 (Counter example). Constraint c_1 of table 1 is split into c_2 and c_3 using

Table 1.

c_1				c_2			c_3		
x_1	x_2	x_3	x_4	x_2	x_3	x_4	x_1	x_2	x_3
0	0	0	2	0	0	2	0	0	0
2	0	1	1	0	1	1	2	0	1
3	1	0	0	1	0	0	3	1	0
1	1	1	3	1	1	3	1	1	1

$\{x_1, x_2\} \rightarrow x_3$. At this stage, because of losslessness, a support for any pair of (variable,value) in c_1 can be built from supports in c_2 and c_3. This is however only true when no pruning has taken place. Imagine that values 2 and 3 are pruned from x_1 and x_3, there still exist supports in c_2 and c_3 for values 0 and 1 whereas c_1 is inconsistent. ▲

The presence of cycles also allows to build lossless decomposition beyond the use of dependencies such as the decomposition of the ALLDIFFERENT global constraint into a network of binary differences.

Example 5 (Lossless Reformulation beyond functional dependencies).

Table 2.

c_4				c_5		c_6		c_7		
x_1	x_2	x_3	x_4	x_3	x_2	x_1	x_3	x_1	x_2	x_4
0	0	0	4	0	0	0	0	0	0	4
1	0	0	2	1	4	1	0	1	0	2
2	4	1	3	2	4	2	1	2	4	3
0	4	2	4	3	2	0	2	0	4	4
2	2	3	2			2	3	2	2	2

Consider constraint c_4 of table 2. Applying $\mathcal{F}_3 : \{x_1, x_2\} \rightarrow x_4$ and $\mathcal{F}_1 : \{x_3\} \rightarrow x_2$ gives the lossless reformulation:

$$\Delta(c_1, \langle \mathcal{F}_3, \mathcal{F}_1 \rangle) = \{\{x_3, x_2\}, \{x_1, x_3\}, \{x_1, x_2, x_4\}\}.$$

By splitting c_7 into c_8 and c_9 respectively on scopes $\{x_1, x_4\}$ and $\{x_2, x_4\}$, we can obtain another lossless reformulation of c_1 :

$$\Delta(c_1) = \{\{x_3, x_2\}, \{x_1, x_3\}, \{x_1, x_4\}, \{x_2, x_4\}\}.$$

The reformulation of c_7 into c_8 and c_9 is not lossless itself as it allows the tuples $(2, 0, 2)$ and $(1, 2, 2)$ on (x_1, x_2, x_4). However, those tuples can not be extended on x_3 due to c_2 and c_3 and the overall reformulation of c_1 remains lossless. For example, the first tuple (2,0,2) assigned value 2 to x_1 and 0 to x_2 and the values of x_3 supporting them in c_2 and c_3 are disjoint (namely $\{1, 3\}$ in c_3 and $\{0\}$ in c_2). This refers to join-dependencies in database.

4 Characterizing a Good Reformulation

For a relation r over a set X of variables, the size of r is computed as the number of values in the table (the memory size) i.e. $|r| * |scope(r)|$. A reformulation $\Delta(r, \delta)$ is a conjunction of constraints that can be characterized by:

1. its maximum arity $a_{\Delta(r,\delta)} = max_{r_i \in \Delta(r,\delta)} |scope(r_i)|$;
2. its maximum number of tuples $l_{\Delta(r,\delta)} = max_{r_i \in \Delta(r,\delta)} |r_i|$;
3. its overall memory size $s_{\Delta(r,\delta)} = \sum_{r_i \in \Delta(r,\delta)} |r_i| * |scope(r_i)|$.

The complexity of GAC is determined by the maximum number of tuples involved in any constraint relation, $l_{\Delta(r,\delta)}$, or the maximum arity, $a_{\Delta(r,\delta)}$, depending on the GAC scheme used [4, 16]. However it is easy to see that $l_{\Delta(r,\delta)}$ is always equal to $|rel(c)|$ in what remains of the initial relation. The maximal number of tuples in the reformulation cannot be decreased. 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. Consider $X_i \rightarrow y_i$ holding on S, once y_i has been removed from S, no pair of tuples can be identical in the resulting table, otherwise they would be identical on X_i and,

therefore, would have been identical on y_i as well, by definition of a functional dependency. This does not mean that the reformulation cannot be more compact. Specifically $s_{\Delta(r,\delta)}$ can be smaller than $|rel(c)| \times |scope(c)|$; arity and memory size, therefore, provide a basis to characterise a *good* reformulation and we will focus our study on those two parameters. Notice however that if the size can be reduced, no exponential gain is possible.

5 Finding Optimal Reformulations

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 and this implies compatibilities between dependencies and valid orderings to compute the reformulation.

Theorem 2 (Valid Ordering of Functional Dependencies [5]). *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 $rel(c)$ such that $y_j \in S_i$ and $\mathcal{F}_i \not\subset \mathcal{F}_j$. 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 $\mathcal{F}_i \prec \mathcal{F}_j$.*

Example 6 (Valid Ordering Dependencies). Consider $F_1 : \{x_1, x_2\} \rightarrow x_3$ and $F_2 : \{x_1, x_3\} \rightarrow x_4$. F_2 can only be applied before F_1 ($\mathcal{F}_2 \prec \mathcal{F}_1$) because the use of $\mathcal{F}_1$ would remove x_3 from the scope and thus prevents $\mathcal{F}_2$ from applying on this scope. ▲

A set of dependencies that can be used together to reformulate a scope is called a valid set:

Definition 5 (Valid set of dependencies). *A set of dependencies δ holding on a scope S is said to be valid if all the depencencies of δ can be applied in a sequence to reformulate S.*

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 $\mathcal{F}_j$*. 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 $root(\delta, S) = \{\mathcal{F}_i \in \delta(S) \mid \forall \mathcal{F}_j \in \delta(S) : \mathcal{F}_i \not\subset \mathcal{F}_j\}$, i.e. the subset of δ that applies to scope S and where no dependency is included in another. A root set on S corresponds to the dependencies that will be used to remove variables from S. The precedence graph of such a set, therefore, needs to be acyclic and valid sets of dependencies are characterized as follows:

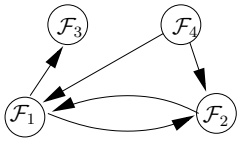
Theorem 3 (A Condition for Valid Set of Dependencies). *A set of functional dependencies $\delta^* \subseteq \delta$ holding on a scope S is valid if and only if $G_{root(\delta^*,S)}$ is acyclic and $\forall \mathcal{F}_i \in \delta^*$, $G_{root(\delta^*,S_i)}$ is acyclic.*

Example 7 (Valid Set of Dependencies). Consider $\delta = \{\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3, \mathcal{F}_4\}$, and their corresponding precedence graph presented in Figure 1. This is a valid set, even if G_δ is cyclic, since it can be verified that all roots correspond to acyclic subgraphs of G_δ

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

```
1.  DS ← {S};
2.  While δ ≠ ∅ Do
3.      For each Scope_k ∈ DS Do
4.          δ* ← root(δ, Scope_k);
5.          If δ* ≠ ∅
6.              DS ← DS − Scope_k;  c ← |δ*|;
7.              For each F_i ∈ δ* Do
8.                  DS ← DS ∪ S_i;
9.              If ∄P ∈ DS, {Scope_k − {y_1, ..., y_c}} ⊆ P;
10.                 DS ← DS ∪ {Scope_k − {y_1, ..., y_c}};
11.             δ ← δ − δ*;
12. Return DS;
```

: $root(\delta, S) = \{\mathcal{F}_1, \mathcal{F}_4\}$, $root(\delta, \{x_1, x_2, x_3\}) = \{\mathcal{F}_3\}$, $root(\delta, \{x_1, x_3, x_5, x_4\}) = \{\mathcal{F}_2\}$, $root(\delta, \{x_3, x_2\}) = \emptyset$ and $root(\delta, \{x_3, x_4, x_1\}) = \emptyset$. So, first of all x_3, x_4 will both be removed from S using $\mathcal{F}_1, \mathcal{F}_4$. Secondly, $\mathcal{F}_3$ and $\mathcal{F}_2$ will be used on the two independent subscopes produced by $\mathcal{F}_1, \mathcal{F}_4$. ▲



$\mathcal{F}_1 : \{x_1\, x_2\} \rightarrow x_3$
$\mathcal{F}_2 : \{x_3\, x_4\} \rightarrow x_1$
$\mathcal{F}_3 : \{x_3\} \rightarrow x_2$
$\mathcal{F}_4 : \{x_1\, x_3\, x_5\} \rightarrow x_4$

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

All dependencies in a valid set δ can be used to decompose the original scope by applying them root by root in an order compatible with the precedences of each root. Moreover, the reformulation associated with the valid sequence of δ, using all elements of δ, is unique. This result relies on the following observation that we will use later.

Lemma 1. *Let $\mathcal{F}_i$, $\mathcal{F}_j$ be two minimal dependencies s.t $\mathcal{F}_i \subseteq \mathcal{F}_j$, then $y_j \in X_i \cup \{y_i\}$.*

Lemma 1 can be used to show that the reformulation using a valid set is unique and the dependencies can only be used once in the process because they are partitioned between the scopes as the scopes are broken.

Theorem 4 (Uniqueness of the Reformulation). *The 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.*

Thus, the reformulation of a valid set can be quickly computed because any order of the dependencies that respects the precedence graph of each root produces the same result. Algorithm 1 computes the reformulation (as a set of scopes) obtained from a valid set δ applying on S. Notice that the resulting reformulation is not necessarily minimal.

Algorithm 2 : SIMPLIFYREFORMULATION($\Delta = \{S_1, \ldots S_p\}$, r)

```
1. DS ← Δ;
2. For each S_k ∈ Δ Do
3.     If ⋈_{S_i ≠ S_k ∈ Δ} (σ_{S_i}(r)) = r
4.         DS ← DS − S_k;
5. Return DS;
```

Example 8 (Uniqueness of Reformulation). Consider the set of four dependencies δ of Figure 1. After applying $\mathcal{F}_4$ on S, $\mathcal{F}_2$ can only be applied on the scope x_1, x_3, x_5, x_4 produced by $\mathcal{F}_4$ because, as we have $\mathcal{F}_2 \subset \mathcal{F}_4$, it necessarily involves x_4 (Lemma 1) in its left hand side and x_4 does not appear in the remainder of S. ▲

The uniqueness of the reformulation for a valid set underpins our claim that when finding an optimal reformulation one can search for valid sets amongst the subsets of the original dependencies ($\mathcal{O}(2^n)$ complexity) rather than consider all possible sequences, which would give an $\mathcal{O}(n!)$ complexity. Once an optimal reformulation has been found it can be rendered minimal easily by checking if each sub-scope is necessary for lossless-ness in a greedy manner. Algorithm 2 takes as input a reformulation and a relation to check that each subscope S_k of the reformulation is mandatory to have a lossless reformulation by performing the join (denoted $\bowtie$) of all the projections of the relation on each other subscope. It returns a subset of Δ which is a minimal reformulation. Notice that many minimal reformulation can exist and this process only returns one of them arbitrarily.

6 Complexity

Given a set of functional dependencies holding on a scope, we consider the complexity of finding a reformulation of minimum size or a reformulation in which the maximum arity of the constraints is minimized. Both problems can be shown to be NP-Hard, since their corresponding decision problems can be used to solve the Weighted Feedback Vertex Set (WFVS) problem, which is known to be NP-Complete.

6.1 Complexity of Arity-Bounded Reformulation

We define the basic decision problem as follows.

> BOUNDED MAX-ARITY REFORMULATION (BAR)
> **Instance:** A set δ of minimal functional dependencies holding on a scope $S = (x_1, \ldots, x_m)$, 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?

Notice that the BAR problem assumes that the set of functional dependencies is given explicitly in advance. The reason is that finding the dependencies from the relation

is itself an NP-Complete problem. However, given a set of functional dependencies δ, it can be shown that there always exists a relation on which only those functional dependencies hold; such relations are known as Armstrong relations [3].

Our complexity proof is based on a reduction from the FEEDBACK VERTEX SET, which is known to be NP-Complete [7].

FEEDBACK VERTEX SET (FVS)
Instance: A directed graph $G = (V, E)$ and a positive integer k.
Question: Does there exist an $X \subset V$ 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).

To prove completeness we show a reduction from the FEEDBACK VERTEX SET to the BAR problem. Consider an instance of the FVS on a graph $G = (V, E)$ with $|V| = n$. We construct an instance of the BAR problem in the following way. A functional dependency $\mathcal{F}_k : \{\ldots\} \rightarrow v_k$ is associated with each node of V. Then each v_k is instantiated to x_k and added to the left side of all dependencies corresponding to the predecessors of v_k in G. Moreover, we add n new variables from $\{x_{n+1}, \ldots, x_{2n}\}$, one to each left 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 $\{x_1, \ldots, x_{2n}, \ldots x_{2n+2}\}$, so that $m = 2n + 2$. Finally we choose $b = m - (n - k)$.

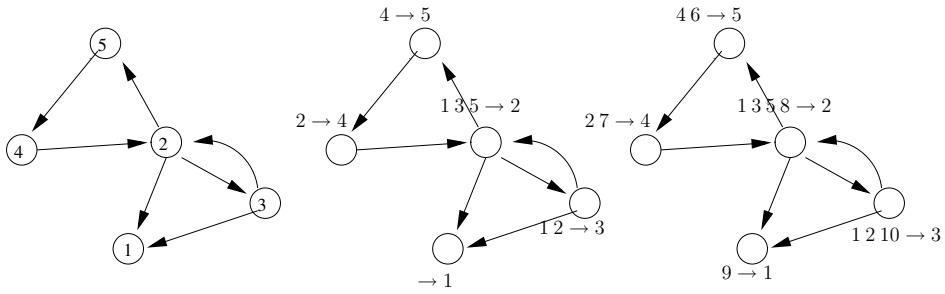


Fig. 2. An example set of dependencies built from a given graph.

The FVS has a solution by removing 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 $\{x_1, \ldots, x_{2n}, \ldots x_{2n+2}\}$ 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 dependencies are of size at most $n+1$ and thus $a_{\Delta(S,V-X)} \leq b$.

$\Leftarrow$ 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 $|X| \leq n-(m-b) = k$. ■

We recall, 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 [**?**]. 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, since the problem of finding all minimal functional dependencies is itself NP-Hard.

6.2 Complexity of Size-Bounded Reformulation

We start first by rewriting the size of a reformulation in $\Delta(r,\delta)$ only in terms of $|\sigma_{S_j}(r)|$ of each dependency $\mathcal{F}_j$ of δ. Notice that each relation r_i of $\Delta(r,\delta)$ is initially derived from a dependency $\mathcal{F}_j$ that produces a relation of scope S_j (with the exception of r_0 denoting here the remainder of the initial scope S). Then p_i variables might have been eventually removed from S_j by other dependencies to reach $scope(r_i)$ (p_i is null for at least one relation of $\Delta(r,\delta)$). The pair (S_j, p_i) is known for each $r_i \neq r_0$ and r_0 can be associated with the pair (S, p_0). The size of a reformulation can be written as:

$$s_{\Delta(r,\delta)} = \sum_{r_i:(S_j,p_i)\in\Delta(r,\delta)} (|S_j| - p_i) \times |\sigma_{S_j}(r)|.$$

The size can, therefore, be computed from the details of each dependency $\mathcal{F}_i$, in particular the cardinality of its projection onto scope S_i, i.e. $|\sigma_{S_i}(r)|$. We define the basic decision problem as follows:

BOUNDED SIZE REFORMULATION (BSR)
Instance: A set δ of minimal functional dependencies holding on a scope $S = (x_1, \ldots, x_m)$. A set $H = \{h_0\} \cup \{h_i | \mathcal{F}_i \in \delta\}$ of positive integers denoting the number of tuples, h_0, of a relation on S and the number of tuples of the relations obtained from S with each dependency of δ. A positive integer b.

Question: Does there exist a subset $\delta_b \subseteq \delta$ with $s_{\Delta(S,\delta_b)} \leq b$, where $s_{\Delta(S,\delta_b)} = \sum_{r_i:(S_j,p_i)\in\Delta(S,\delta_b)} (|S_j| - p_i) \times h_j$ is the size of the reformulation obtained using δ_b?

While constructing an Armstrong relation is exponential in the size of δ, upper and lower bounds on the minimal number of tuples in the relation are known [3]. The BSR problem contrary to BAR refers to a set of dependencies with the specific number of tuples in each projection of the relation on the scope of each dependency of δ. We show here that such a relation always exists for some specific set of dependencies and use it to prove theorem 6.

Lemma 2. *Let δ be a minimal set of dependencies defined on scope S such that one variable $x \in S$ does not appear in the scope of any dependency of δ. Assume that each $\mathcal{F}_i \in \delta$ has at least one element $e_i \in X_i$ that does not appear in the scope of any other dependency. Let h_α be an upper bound on the minimum number of tuples in an Armstrong relation for δ. Consider also a set of integers $h_i \geq h_\alpha$ for all $1 \leq i \leq |\delta|$. There is an Armstrong relation r for δ such that $|r| \geq \sum_i h_i + h_\alpha$ and $\forall i, h_i = |\sigma_{S_i}(r)|$.*

Proof. Consider the minimum Armstrong relation Λ for δ with at most h_α tuples and v_α different values (in the range $[1, v_\alpha]$). We show here how to add tuples to Λ without breaking any dependency in δ (no dependencies can be introduced by adding tuples) to achieve the proper size of each projection. For each dependency $\mathcal{F}_i$ of δ we add a set $T = \{t_j | 1 \leq j \leq h_i - |\sigma_{S_i}(\Lambda)|\}$ of tuples to Λ. Let t be an arbitrary tuple of Λ. Each t_j of T is identical to t except that $t_j[e_i] = v_\alpha + i + j$. It can be easily verified that T appears only in σ_{S_i} and in none of the σ_{S_j} for $i \neq j$. In addition, the only violated dependencies by the addition of T are of the form $V \rightarrow e_i$ that cannot be in δ. Finally $|r|$ can be made greater than $\sum_i h_i + h_\alpha$ by adding tuples equal to a tuple of Λ except on x where $t[x]$ is constructed using a new value for each tuple. ■

Our complexity proof is based on a reduction from the WEIGHTED FEEDBACK VERTEX SET, which is known to be NP-Complete [7].

WEIGHTED FEEDBACK VERTEX SET (WFVS)
Instance: A positive integer k and a weighted directed graph $G = (V, W, E)$ where $w_v \in W$ denotes a positive integer associated with each node $v \in V$.
Question: Does there exist an $X \subseteq V$ with $\sum_{x \in X} w_x \leq k$ such that G with the vertices from X removed is acyclic?

Theorem 6. *The* BOUNDED SIZE REFORMULATION *problem is NP-Complete.*

Proof. Clearly, this problem is in NP. To prove completeness we show a reduction from the WFVS problem. Consider an instance of the WFVS on a graph $G = (V, W, E)$ with $|V| = n$. We construct an instance of the BSR problem in the following way. A functional dependency $\mathcal{F}_k : \{\ldots\} \rightarrow v_k$ 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 *different* new variables to each left side of *all* functional dependencies and one more variable that does not appear in any dependency. The resulting scope S is equal to $\{x_1, \ldots, x_{n^2+n+1}\}$. Figure 3 shows the construction resulting from the two previous steps on an example graph G. These dependencies 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 and $m = n^2 + n + 1$. We denote by L, the least common multiple of all $|S_i|$ and $w_i^{'} = (\frac{h_0}{h_\alpha L} - w_i)$. We set h_i with $0 < i \leq n$,

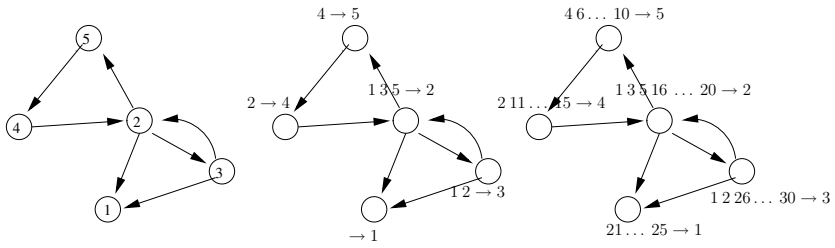


Fig. 3. An example set of dependencies built from a given graph.

b and h_0 to the following values: $h_0 = L \times max_i(w_i) \times h_\alpha, h_i = w_i^{'} \times \frac{h_\alpha L}{|S_i|}$ and $b = h_\alpha L(\sum_{i \in V} w_i^{'} + k) + h_0 \times (n^2 + 1)$, where h_α is an upper bound on the size of the minimum Armstrong relation for δ (computable in polynomial time). It can be verified that they are all positive integers, that $h_i \geq h_\alpha$ and that $h_0 \geq \sum_{i=1}^{n} h_i + h_\alpha$ because $\sum_{i=1}^{n} h_i + h_\alpha = \sum_{i=1}^{n} \frac{h_0 - h_a L w_i}{|S_i|} + h_\alpha = h_0 \sum_{i=1}^{n} \frac{1}{|S_i|} - h_a(\sum_{i=1}^{n} \frac{L w_i}{|S_i|} - 1)$ and that $\sum_{i=1}^{n} \frac{1}{|S_i|} \leq 1$ and $\sum_{i=1}^{n} \frac{L w_i}{|S_i|} > 1$. Indeed, all w_i are positive and for all i, $|S_i| > n$. This transformation is polynomial and such an Armstrong relation with the corresponding properties exists according to Lemma 2.

We show that G has a feedback vertex set of weight at most k if and only if S has a reformulation of size at most b. Assume that we have a solution to BSR with $s_{\Delta(S,\delta_b)} \leq b$ and denote by X the set of nodes corresponding to the complement of δ_b in V so that $\delta_b = V - X$. There is a one-to-one correspondence between the nodes of G and dependencies of δ by construction. It can be verified that $\sum_{i \in X} w_i \leq k$, in the following way starting from $s_{\Delta(S,\delta_b)} \leq b$; notice that all the p_i are null except p_0 because all dependencies apply on the main scope and thus :

$$\sum_{i \in \delta_b} |S_i| \times h_i + h_0(|S| - |\delta_b|) \leq b$$

$$h_\alpha L \sum_{i \in \delta_b} w_i^{'} + h_0(n^2 + n + 1 - (n - |X|)) \leq b$$

$$\sum_{i \in \delta_b} w_i^{'} + \frac{h_0}{h_\alpha L}|X| \leq \sum_{i \in V} w_i^{'} + k \Rightarrow -\sum_{i \in X} w_i^{'} + \frac{h_0}{h_\alpha L}|X| \leq k \Rightarrow \sum_{i \in X} w_i \leq k.$$

Conversely, it can be easily seen, using the same computation, that any solution X to the WFVS having a weight at most k gives a set of dependencies δ_b (the complement of X in V) that provides a reformulation of size at most b. ■

7 An Algorithm for Optimal Reformulation

We propose a complete algorithm to find optimal reformulations. We observe that many independent subproblems occur when trying to compute the optimal reformulation in terms of arity or size.

The rationale behind the algorithm is that once a dependency $\mathcal{F}_i$ is chosen, $\delta(S_i)$ can only apply on S_i and, therefore, two independent subproblems appear: the optimal reformulation of S_i using $\delta(S_i)$ on one side and the optimal reformulation of $S - \{y_i\}$ on the other. $\delta(S_i)$ can only apply on scope S_i because all the dependencies of $\delta(S_i)$ involve y_i according to Lemma 1, and y_i only appears in S_i.

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. This suggests that the optimal reformulation of the scope of each dependency can be computed first to avoid their redundant computations. We simply show here that all optimal solutions can be indeed mapped to optimal solutions based only on optimal reformulation of the scope of each dependency.

Theorem 7 (Independence of subproblems). *Let δ be a set of dependencies holding on S and $\delta' \subseteq \delta$ an optimal solution to the reformulation problem. Let's denote by $\delta^*_{\mathcal{F}_i}$ the optimal solution to the reformulation problem of S_i using $\delta(S_i)$. Then the set $\delta'' = \cup_{\mathcal{F}_i \in root(\delta', S)} \delta^*_{\mathcal{F}_i}$ is also optimal.*

This result simply relies on the observation that all subproblems on each S_i are independent and can be expressed by the following lemma first:

Lemma 3. *For any two minimal compatible dependencies $\mathcal{F}_i$ and $\mathcal{F}_j$,* i.e *such that $\mathcal{F}_i \prec \mathcal{F}_j$ and $\mathcal{F}_j \prec \mathcal{F}_i$ do not hold, then $\delta^*_{\mathcal{F}_i} \cap \delta^*_{\mathcal{F}_j} = \emptyset$.*

The reformulation algorithm is presented as Algorithm 3 and can be see as a dynamic programming algorithm.

Algorithm 3 OPTIMALREFORMULATION($\delta = \{\mathcal{F}_1, \ldots, \mathcal{F}_n\}, S$)

```
1. sort δ by increasing size of scopes;
2. For each F_i ∈ δ do
3.     (a*/s*_{F_i}, δ*_{F_i}) ← FindReducedOR(δ(S_i), ∅, S_i);
4. FindReducedOR(δ, ∅, S);
```

The arity/size of the optimal reformulation obtained by each $\mathcal{F}_i$ of $\delta(S_i)$ is known when calling Algorithm 4, i.e. FINDREDUCEDOR (line 3), as all such subproblems have been solved independently before due to the sorting of line 1. Their optimal value is denoted $a^*_{\mathcal{F}_i}$ or $s^*_{\mathcal{F}_i}$ and the set of corresponding dependencies by $\delta^*_{\mathcal{F}_i}$.

Example 9 (An Optimal Reformulation Problem). Figure 4 gives an example of a reformulation problem where the dependencies have been organized into independent subproblems. The arity of the initial scope is eight and the circled dependencies denote an optimal solution to get a reformulation of arity three. The optimal value of the subproblem associated to each dependency is indicated in parentheses. $1\ 4\ 6\ 8 \rightarrow 7$ leads, for example, to a subscope that can be reformulated with a maximum arity of three by using $7\ 1\ 4 \rightarrow 6$, $7\ 6 \rightarrow 4$ and $7\ 1 \rightarrow 8$. It can also be seen that $7\ 6 \rightarrow 4$, which is

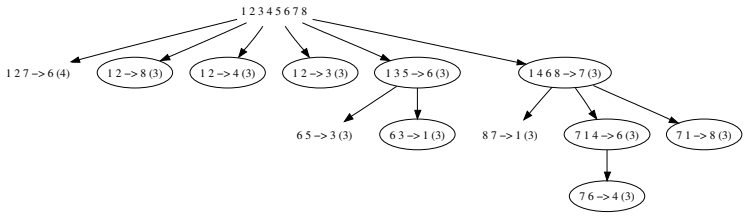


Fig. 4. An example of a hierarchy of subproblems.

included in $1\ 4\ 6\ 8\ \rightarrow 7$ and $7\ 1\ 4\ \rightarrow 6$, therefore, uses 7 and 6 in its left-hand side as stated by Lemma 1. Algorithm 3 will solve the subproblems from the leaves to the root of this tree by calling Algorithm 4 for each subproblem. ▲

Algorithm 4 FINDREDUCEDOR($CD = \{\mathcal{F}_i, \ldots \mathcal{F}_n\}$, PD, S)

```
// a*_{F_i}/s*_{F_i} and δ*_{F_i} are assumed to be known for all F_i ∈ CD
1. If CD = ∅
2.    REFORMULATE(PD ∪ {δ*_{F_i} | F_i ∈ PD}, S);
3.    If an improving solution has been found
4.      store it and update the upper bound
6. Else
7.    CD ← CD − {F_i}; // F_i is chosen for branching
8.    FD ← PRUNING(CD, PD ∪ {F_i}) ∪ δ(S_i) ∪ {F_k ∈ CD | S_i ⊆ S_k};
9.    If BOUND(CD − FD, PD ∪ {F_i}, S) < upperBound
10.     FINDREDUCEDOR(CD − FD, PD ∪ {F_i}, S);
11.   If BOUND(CD, PD, S) < upperBound
12.     FINDREDUCEDOR(CD, PD, S);
```

Algorithm 4 assumes that each dependency is labeled with its corresponding $a^*/s^*_{\mathcal{F}_i}$ and $\delta^*_{\mathcal{F}_i}$. It computes the optimal reformulation of S using such dependencies. It is, basically, a branch and bound over the subsets of subproblems defined by each dependency of δ. At each node it considers the current set of subproblems (called PD) chosen to reformulate S and the set of current candidate subproblems (called CD) to be included (or not) in PD. PD and CD are maintained as sets of dependencies and PD can be seen as the root set of S: $root(\delta, S)$. The algorithm proceeds as follows:

- If CD is empty, the reformulation is computed with Algorithm 1 and the best known solution is updated if needed. Notice that $PD \cup \{\delta^*_{\mathcal{F}_i} | \mathcal{F}_i \in PD\}$ is a valid set.
- Otherwise, the algorithm branches by selecting which subproblem will be used to remove a variable from the scope S. A dependency $\mathcal{F}_i$ is chosen and the algorithm branches on the corresponding subproblem (line 7). FD represents the forbidden dependencies and the method PRUNING($CD, PD \cup \{\mathcal{F}_i\}$) computes all the functional dependencies of CD that would create a cycle in the initial root set represented by PD; As a subproblem is selected for branching, all dependencies from $\delta(S_i)$ and the one containing S_i are pruned (line 8). Branching on $7\ 1\ 4\ \rightarrow 6$ on the example of Figure 4 would prune $1\ 4\ 6\ 8 \rightarrow 7$, $7\ 6 \rightarrow 4$ and all dependencies creating a cycle i.e. $1\ 3\ 5 \rightarrow 6$, $6\ 5 \rightarrow 3$, $6\ 3 \rightarrow 1$ and $1\ 2\ 7 \rightarrow 6$.

Algorithm 5 BOUNDARITY(CD, PD, S)

1. $lb \leftarrow max_{\mathcal{F}_i \in PD}\ a^*_{\mathcal{F}_i}$;
2. $lbs \leftarrow |S|$ - $|PD|$ - (|CD| - lower bound on the min. vertex feedback set in G_{CD});
3. **Return** $max(lb, lbs)$;

– If the bound obtained for the new pair $(CD - FD, PD \cup \{\mathcal{F}_i\})$ is compatible with the best known solution, the algorithm branches.

Finally, the reformulation computed by the algorithm is optimal but not necessarily minimal in the case of the arity and algorithm 2 has to be applied to get a minimal one. In the case of the size, the reformulation is necessarily minimal as it would not be optimal otherwise.

The algorithm relies on the partitioning of functional dependencies amongst subproblems as outline by Lemma 3, and specifically that no dependency is added twice (when completing PD with the dependencies of each subproblem – line 2).

The algorithm is sound because it computes only valid sets: adding $\delta^*_{\mathcal{F}_i}$ (line 2) cannot introduce any cycle in any root sets because each $\delta^*_{\mathcal{F}_i}$ is known to be valid already. The only root that needs to be checked is the initial one, PD, which is ensured by the pruning of the corresponding cycles (line 9). A heuristic can be applied (at line 7). A preprocessing step can also be applied when minimizing the size by removing all $\mathcal{F}_i$ such that $s^*_{\mathcal{F}_i} > h_0$. Finally simple bounds are used to prune the search. The proof of NP-Completeness showed the strong relationship between this problem and the WFVS. The bounding procedures relate to simple bounds for the WFVS.

The minimum arity expected from a current set of functional dependencies PD and the remaining candidates CD is simply computed here as the maximum over the minimal arity known for each dependency of PD and a lower bound on the arity expected from the remain of the original scope S (see Algorithm 5). The latter is computed by looking at the maximum number of variables that can still be removed from S without creating a cycle. This is the quantity: (|CD| - lower bound on the min. vertex feedback set in G_{CD}).

All bounds known for the FVS can be used. We use a simple lower bound that 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 $\sum_{C_i \in P}(|C_i| - 1)$. Consider the case of minimizing the size of the reformulation (Algorithm 6). Firstly, the optimal size of the reformulation associated with each dependency of PD is taken into account into the bound. Secondly, we consider the graph G_{CD} where a weight equal to $s^*_{\mathcal{F}_i} - h_0$ is associated with each node $\mathcal{F}_i$. The weight corresponds to the reduction in size (the gain) obtained by the use $\mathcal{F}_i$. A lower bound on the minimum weighted vertex feedback set gives an upper bound on the gain in size that we can expect due to the remaining dependencies.

A lower bound on the minimum WFVS can be based on the partition into cliques as well, by keeping in each clique the node of minimum weight $\sum_{C_i \in P} min_{\mathcal{F}_j \in C_i} s^*_{\mathcal{F}_i}$.

Algorithm 6 BOUNDSIZE(CD, PD, S)

```
1. lb ← ∑_{F_i ∈ PD} s*_{F_i} ;
2. lb ← lb + h_0 × (|S| − |PD|) − (∑_{F_i ∈ CD} (s*_{F_i} − h_0)) −
       lower bound on the min. weighted vertex feedback set in G_CD);
3. Return lb;
```

8 A Comparison with Other Compression Techniques

Several techniques have been introduced to improve the efficiency of reasoning over table constraints [6, 8, 12, 19]. A recent approach based on table compression [12] relies on a *cross-product* representation the tuples [10], e.g. the tuples $\{\langle 1,1,1\rangle, \langle 1,2,1\rangle\}$ can be represented as a single tuple $\langle (1)(1,2)(1)\rangle$. The authors of [12] state that *"The applicability of the representation is also reduced for tables where some of the variables are functionally dependent on some others"*. We believe that both techniques can strictly benefit from each other as breaking functional dependencies can only reduce the Hamming distance between the tuples by projecting them on sub-scopes. We show here that the technique in [12] is complementary to our reformulation approach.

Proposition 1. *The gain in size achieved by the approach presented in [12] and our functional dependency-based approach to reformulation are incomparable.*

Proof. Consider constraint c of Table 3 with $4kn$ tuples. For any pair of tuples in c, the Hamming distance is at least 2, thus preventing any compression using the representation from [12]. Notice that this table exhibits several dependencies and especially $x_2 \rightarrow x_3$ and $x_1 \rightarrow x_4$ corresponding, in this example, to two equality constraints. The reformulation $\Delta(c)$ is obtained from those two dependencies and its size, $2k+2n+2kn$ shows the following gain: $1 < \frac{s_c}{s_{\Delta(c)}} < 2$ if $k, n > 2$ (the gain would increase with the arity). Another observation is that c_3 can now be represented very efficiently by a cross-product $\langle (1,\ldots,n) \times (1,\ldots,k)\rangle$ whereas no dependencies hold[1], showing that both techniques are complementary. ■

Other representations of the tuples have been proposed such as [8], which relies on a "trie" data structure. A trie aims a factoring the shared prefixes of the tuples so it, essentially, captures the same kind of structure as the cross-product, i.e. the information stored in a redundant way in many tuples. The approach of [19] tries to achieve compression of the tuples by computing a minimal automaton whereas the CASE constraint corresponding to the table constraint in Sicstus [6] uses a DAG to get a representation similar to the cross-product. All these approaches rely on the idea that the tuples of the table share some information which is stored redundantly and can be compressed by using the appropriate data structure. Dependent variables only hinder their efficiency. Our reformulation approach is orthogonal to those techniques by capturing a very different kind of structure. It cannot, however, achieve by itself an exponential reduction in size, which is possible with the previous compression approaches. The bottleneck

[1] This is, in fact, a multi-valued dependency [11] as x_1 and x_2 are independent of each other and could also be detected as such.

Table 3. An example of a table constraint and its reformulation.

c				$\Delta(c)$					
				c_1		c_2		c_3	
x_1	x_2	x_3	x_4	x_2	x_3	x_1	x_4	x_1	x_2
1	1	1	1	1	1	1	1	1	1
2	1	1	2	2	2	2	2	2	1
.	.	.	.	.	.	.	.	.	.
n	1	1	n	k	k	n	n	n	1
1	2	2	1					1	2
.	.	.	.					.	.
n	2	2	n					n	2
.	.	.	.					.	.
1	k	k	1					1	k
.	.	.	.					.	.
n	k	k	n					n	k

lies in the fact that the maximum number of tuples cannot be reduced using functional dependencies alone. Typically, for a constraint of arity a with n tuples of original size an, the reformulation always contains, in the best case, a constraint of size $2n$.

9 Experimental Results

The objective of our experimental evaluation was to study how effectively our functional dependency-based approach to reformulation could reduce the worst-case (maximum) arity of constraints in our reformulation as well as its total memory size[2] . We considered two cases with respect to size: measuring the sum of the sizes of each table constraint in the reformulation, as well as the sum of the sizes of a REGULAR-based compilation of each table. We considered positive table constraints from the following five datasets: the Renault Megane Car Configuration Problem (we used the two largest table constraints, R80 and R140) [2]; a dataset of digital cameras [18]; a dataset of laptops [18]; the AI-CBR travel case-base [14]; and a dataset based on the crossword puzzle CSP benchmark [1]. We used a well-known library, called TANE [11], to compute the set of minimal functional dependencies for each table.

Table 4. Results on minimizing arity and memory size (time given in seconds).

Instance Details						Minimize Maximum Arity						Minimize Memory Size					Minimize DFA		
						Complete				Greedy		Complete			Greedy		Complete		
name	nbt	arity	size	ndep	time	max	gain	size	time	arity	time	size	gain	time	size	time	orig	refor	gain
camera	112	8	896	41	0.21	**5**	1.6	2220	0.07	5	0.05	**896**	1.0	0.02	896	0.01	421	**421**	1.0
laptop	403	10	4030	54	0.12	**6**	1.67	14393	0.03	7	0.0	**4030**	1.0	0.05	4030	0.05	1452	**1423**	1.02
rn R80	342	10	3420	2	0.25	**8**	1.25	2836	0.0	8	0.0	**2836**	1.21	0.0	2836	0.0	209	**137**	1.53
rn R104	164	9	1476	11	0.06	**4**	2.25	1026	0.0	5	0.0	**836**	1.77	0.01	1140	0.0	183	**100**	1.83
travel	1470	9	13230	7	0.14	**6**	1.5	27535	0.0	6	0.0	**13230**	1.0	0.01	13230	0.01	3693	**3021**	1.22
cw R10	1881	12	22572	26	0.63	**10**	1.2	60114	0.0	10	0.0	**22572**	1.0	0.06	22572	0.06	5138	**5138**	1.0
cw R11	1136	13	14768	128	0.65	**9**	1.44	45690	0.84	9	0.0	**14768**	1.0	0.13	14768	0.13	5426	**4773**	1.14
cw R12	545	14	7630	1211	0.62	**7**	2.0	33546	109.1	8	0.02	**7630**	1.0	0.56	7630	0.51	3585	**3503**	1.02
cw R13	278	15	4170	2243	0.41	6	2.5	14645	-	7	0.15	**4170**	1.0	0.83	4170	0.51	2095	**2046**	1.02
cw R14	103	16	1648	4268	0.45	5	3.2	5061	-	6	0.74	**1648**	1.0	1.65	1648	0.4	970	**942**	1.03
cw R15	57	17	969	5057	0.37	**4**	4.25	2611	10.78	5	1.13	**969**	1.0	2.04	969	0.29	674	**627**	1.07
cw R16	23	18	414	3514	0.4	**3**	6.0	945	1.77	4	0.32	**409**	1.01	1.58	414	0.12	287	**284**	1.01

[2] These experiments were run on a MacBook 2 GHz Intel Core Duo, 2 GB 667 Mhz DDR2.

Table 4 presents the results and is divided into four parts. Firstly, we present some information on the original tables including their number of tuples, arity, size, number of minimal functional dependencies and the time needed by Tane to extract them. Secondly, we show the results associated with finding the optimal reformulation that minimizes the maximum arity. Thirdly, we present similar results focused on minimizing memory size. Finally, we show results associated with minimizing the size of the automata used by a REGULAR-based reformulation. A time limit was put at 120 seconds and a '-' indicates that the time-out was reached while bold indicates when the result has been proven optimal. The columns "gain" present the ratios of the original measure divided by the measure from the reformulation. We also present the results for a simple greedy algorithm selecting first the dependency of smallest scope. We observe that we can always find a reformulation in which the maximum arity is reduced: the gains range from 1.2 to 6. When we focus on minimizing memory size, the results are less successful and only the two Renault configuration benchmark tables are reduced. The optimal reformulation algorithm is very efficient when there are fewer than 1000 dependencies, but the more practical greedy algorithm achieves excellent performance even if the optimal solution is not always found.

Compression techniques can also be used to further reduce the size of our reformulation. In the final three columns of Table 4 we present the size of the automaton of a REGULAR [17, 19] constraint for the original table, the sum of the sizes of each automata for the reformulation, and the corresponding gain in space we achieve through reformulation. The heuristic used to order the variables in the automaton is simply to put first the variable of minimum domain. We find that using a compilation of our reformulation, we reduce space in almost every case, and particularly so for the laptop, Renault, and travel datasets confirming the complementarity of the techniques.

10 Conclusion

Constraints that are defined by tables of allowed tuples of assignments are common in constraint programming. They are a very natural modeling tool for beginners in CP who often tend to enumerate the allowed tuples of a logical relation that does not fit perfectly into any of the intentional constraints provided by a constraint toolkit. In this paper we present an approach to reformulating table constraints of large arity into a conjunction of lower arity constraints. Our approach exploits functional dependencies that might hold on the relation. We summarized many issues on dependencies in the context of reformulation, presented the complexity of the reformulation problem, a dynamic programming algorithm for optimal reformulation, and evaluated it on real-world and academic datasets. The experiments show that the gain of size is not large enough for an improvement in performance during search on those benchmark but open many opportunities when combined with compression techniques which deserve further studies. The experiments stand here as a proof of concept, as this technique is intending as an automatic way to deal with naive models made of large arity constraints, thus making CP easier to use.

The future work on this topic would be to extend the reformulation schema to be able to remove several variables at the same time when functional dependencies can be

gathered to have the form $X \rightarrow Y$ (where Y is a set of variables). This can not improve the quality of the reformulation found when minimizing the max-arity but could eventually lead to better reformulations in size. However as functional dependencies can not bring an exponential gain in size, it seems more interesting to directly extend this work to multi-valued dependencies.

Acknowledgement

This work was supported by Science Foundation Ireland (Grant number 05/IN/5806).

References

1. J. Sillito A. Beacham, X. Chen and P. van Beek. Constraint programming lessons learned from crossword puzzles. In *14th Canadian Conference on Artificial Intelligence*, pages 78–87, 2001.
2. Jérôme Amilhastre, Hélène Fargier, and Pierre Marguis. Consistency restoration and explanations in dynamic CSPs – application to configuration. *Artif. Intell.*, 135:199–234, 2002.
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. Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks: Preliminary results. In *IJCAI (1)*, pages 398–404, 1997.
5. Hadrien Cambazard and Barry O'Sullivan. Reformulating table constraints using functional dependencies - an application to explanation generation. *Constraints*, 13, 2008.
6. Mat Carlsson. Filtering for the case constraint. In *Talk given at the Advanced School on Global Constraints*, Samos, Greece, 2006.
7. 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.
8. Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. In *AAAI*, pages 191–197, 2007.
9. Marc Gyssens, Peter Jeavons, and David A. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artif. Intell.*, 66(1):57–89, 1994.
10. Paul D. Hubbe and Eugene C. Freuder. An efficient cross product representation of the constraint satisfaction problem search space. In *AAAI*, pages 421–427, 1992.
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. George Katsirelos and Toby Walsh. A compression algorithm for large arity extensional constraints. In *CP*, pages 379–393, 2007.
13. François Laburthe and Yves Caseau. Using constraints for exploring catalogs. In *CP*, pages 883–888, 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, *ICCBR*, 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. G. Pesant. A regular language membership constraint for finite sequences of variables. In LNCS Springer, editor, *Tenth International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3258, 2004.
18. 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.
19. Guillaume Richaud, Hadrien Cambazard, Barry O'Sullivan, and Narendra Jussien. Automata for nogood recording in constraint satisfaction problems. In *CP06 Workshop on the Integration of SAT and CP techniques*, Nantes, France, 2006.
20. Mark Wallace. Practical applications of constraint programming. *Constraints*, 1(1/2):139–168, 1996.

Appendix

Proof of theorem 1.

Proof. The reformulation $\Delta(c, \mathcal{F})$ of constraint c is lossless since it relies on a functional dependency. Moreover, the left hand side of $\mathcal{F}$ is restricted to a single variable and the resulting constraint network is Berge-acyclic : $\Delta(c, \mathcal{F}) = \{\{x, y\}, \{x, scope(c) - y\}\}$. Arc-consistency for a Berge-acyclic constraint network is achieved by making each constraint of the corresponding network arc-consistent. ■

Proof of theorem 2.

Proof. Applying the functional dependency $\mathcal{F}_j$ would give rise to two new constraints with the scopes $X_j \cup \{y_j\}$ and $scope(c) - \{y_j\}$ (by Definition 3). $X_i \cup \{y_i\}$ can neither be part of $scope(c_i) - \{y_j\}$ if $y_j \in X_i \cup \{y_i\}$ nor of $X_j \cup \{y_j\}$ since $X_i \cup \{y_i\} \not\subset X_j \cup \{y_j\}$. Therefore, $\mathcal{F}_i$ can no longer apply. ■

Proof of theorem 3.

Proof. Consider first a set of dependencies not included in each other and applied on the same scope S. We first show that this set is a valid set for S if and only if the precedence graph associated to this set is acyclic. If the set is valid, then there is an order of the dependencies $< \mathcal{F}_1, \ldots, \mathcal{F}_i, \ldots, \mathcal{F}_k >$ such that the F_i do not contain one the variables removed by $\mathcal{F}_1, \ldots, \mathcal{F}_{i-1}$. In other words, (as no dependencies is included in another one) there is no $p < i$ such that $\mathcal{F}_i \prec \mathcal{F}_p$ and thus the precedence graph is acyclic. Now if the precedence graph is acyclic, any order satisfying the precedences can be used as a sequence (following a topological order). The general case is now as follow:
$\Rightarrow$ Assume δ^* is valid. For a sequence to exist, all dependencies removing variables from the same scope must define an acyclic graph as shown above. Thus all roots must be acyclic. A root is by definition a set of dependencies applying on the same scope.
$\Leftarrow$ If all the roots are acyclic, then the following sequence of dependencies can be used to reformulate S: Add first the dependencies of $root(\delta^*, S)$ ordered to respect the precedences of $G_{root(\delta^*,S)}$, then for each of the scope generated S_i, extend the sequence by adding the dependencies of $root(\delta^*, S_i)$ ordered to respect the precedences

of $G_{root(\delta^*,S_i)}$ and so on as long as $root(\delta^*, S_i)$ is not empty. ■

Proof of Lemma 1.

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 minimal because another dependency, namely $\mathcal{F}_i$, is contained in its left-hand side. ■

Proof of theorem 4.

Proof. Let $\delta_0 = root(\delta, S)$ be the root set of δ. Let $k = |\delta_0|$, the number functional dependencies in δ_0. The reformulation obtained by δ_0 is unique. Because δ_0 is a valid set of dependencies (being a subset of δ), the set of all y_i variables of the dependencies $\mathcal{F}_i : X_i \rightarrow y_i$ of δ_0 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 2) and δ_0 would not be valid. All $\{y_1, \ldots, y_k\}$ are different, therefore, each $\mathcal{F}_i$ can only be applied on the main scope S. The reformulation obtained by applying dependencies of δ_0 in an order respecting their precedences is made of $k + 1$ scopes:

$$\begin{cases} S_0 = S - \{y_1, \ldots, y_k\} \\ S_1 = X_1 \cup \{y_1\} \\ \ldots \\ S_k = X_k \cup \{y_k\} \end{cases}$$

Let's now consider a dependency $\mathcal{F}_a \in \delta - \delta_0$ and show that it holds on a single scope of the previous reformulation. $\mathcal{F}_a$ was contained in one of the dependencies of δ_0 and therefore holds on at least one of the scope, S_i, of the previous reformulation. Assume that it also holds on S_j then by lemma 1 y_i and y_j are in S_a and therefore $y_j \in S_i$ and $y_i \in S_j$ which would mean $\mathcal{F}_i \prec \mathcal{F}_j$ and $\mathcal{F}_j \prec \mathcal{F}_i$. This is impossible as δ is assumed to be valid. Thus, $\mathcal{F}_a$ is contained in **exactly** one scope of the previous reformulation.

The remaining dependencies are, therefore, partitioned amongst the previous scopes and this partition is also unique. 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. ■

Proof of Lemma 3.

Proof. We cannot have both $\mathcal{F}_i \prec \mathcal{F}_j$ and $\mathcal{F}_j \prec \mathcal{F}_i$. $\mathcal{F}_i$ and $\mathcal{F}_j$ must, therefore, have a different right hand side so $x_i \neq x_j$(we assume that $\mathcal{F}_i$ and $\mathcal{F}_j$ are not equal). Assume there is a dependency $\mathcal{F}_k : X_k \rightarrow x_k$ in both $\delta^*_{\mathcal{F}_i}$ and $\delta^*_{\mathcal{F}_j}$ then x_i and x_j are in X_k (by Lemma 1). That means that x_i is also in X_j (because we assumed $\mathcal{F}_k \in \delta^*_{\mathcal{F}_j}$) and similarly x_j is in X_i. In that case we have $\mathcal{F}_i \prec \mathcal{F}_j$ and $\mathcal{F}_j \prec \mathcal{F}_i$ which gives a contradiction. ■

Proof of theorem 7.

Proof. For any two dependencies $\mathcal{F}_i$ and $\mathcal{F}_j$ of δ' we have $\delta^*_{\mathcal{F}_i} \cap \delta^*_{\mathcal{F}_j} = \emptyset$ by Lemma 3. $\mathcal{F}_i \prec \mathcal{F}_j$ and $\mathcal{F}_j \prec \mathcal{F}_i$ cannot both hold as δ' is a valid set. We can therefore replace each $\delta'(S_i)$ by $\delta^*_{\mathcal{F}_i}$ and the objective can not be degraded in the process. Thus δ'' is also a valid and optimal reformulation. ■