Improving Relational Consistency Algorithms Using Dynamic Relation Partitioning*

Anthony Schneider[1], Robert J. Woodward[1,2], Berthe Y. Choueiry[1], and Christian Bessiere[2]

[1] Constraint Systems Laboratory, University of Nebraska-Lincoln, USA
{aschneid,rwoodwar,choueiry}@cse.unl.edu
[2] LIRMM, CNRS & University of Montpellier, France
bessiere@lirmm.fr

Abstract. Relational consistency algorithms are instrumental for solving difficult instances of Constraint Satisfaction Problems (CSPs), often allowing backtrack-free search. In this paper, we improve an algorithm for enforcing relational consistency by exploiting the property that the constraints of the dual encoding of a CSP are piecewise functional. This property allows us to partition a CSP relation into blocks of equivalent tuples at varying levels of granularity. Our new algorithm dynamically exploits those partitions. Our experiments show a significant improvement of the processing time for enforcing relational consistency.

1 Introduction

Algorithms for enforcing local consistency are a focal point of research in Constraint Programming because they are an efficient means to reduce the size of the search space and effort [1]. In recent years, new techniques for enforcing higher levels of consistency have been proposed. While most consider combinations of two constraints [3, 20–22], some operate on combinations of two or more constraints [2, 6, 15, 16, 23, 24]. In this paper, we improve the performance of the algorithm for enforcing the relational consistency property R(∗,m)C [15, 16] (originally known as m-wise consistency [10]). This property ensures that any tuple can be consistently extended over every combination of $m-1$ relations.

Samaras and Stergiou showed that the constraints of the dual encoding of a CSP are piecewise functional [11, 22]. Given two constraints that are adjacent in the dual graph, this property partitions the tuples of each relation into a set of *blocks*, i.e., equivalence classes of tuples. They exploited those partitions in an algorithm (PW-AC) for enforcing pairwise-consistency (i.e., R(∗,2)C), which is defined on *pairs* of relations. Extending the work of Samaras and Stergiou, we identify as *coarse* blocks those induced on a constraint's relation by one other adjacent constraint and as *fine* blocks those induced by all other adjacent

* This research was supported by NSF Grant No. RI-111795 and EU project ICON (FP7-284715). Woodward was supported by an NSF GRF Grant No. 1041000 and a Chateaubriand Fellowship. Experiments were conducted on the equipment of the Holland Computing Center at the University of Nebraska–Lincoln.

B. O'Sullivan (Ed.): CP 2014, LNCS 8656, pp. 688–704, 2014.

constraints. We modify the PERTUPLE[1] algorithm for enforcing R(∗,m)C into the PERFB algorithm, which exploits not only the fine and coarse blocks but also intermediate ones induced by a subset of the constraint's neighbors.

The contributions of this paper are as follows: *a*) The definitions of levels of relation partitions and the specification of data structures to store and manipulate the coarse and fine blocks; *b*) The design of an algorithm that utilizes those data structures to enforce R(∗,m)C; *c*) A complexity analysis of our data structures and algorithm; and, *d*) An empirical evaluation of PERFB comparing its performance to that of PERTUPLE.

In addition to the contribution of Samaras & Stergiou [22], our approach is related to the following research. Karakashian et al. propose a compact data structure, the index tree, which finds coarse blocks and stores them in the leaves of the tree [16]. However, they fall short of exploiting them to improve constraint propagation. Lecoutre et al. propose the algorithm STR3 to enforce GAC using the size of the blocks induced, on a relation, by a variable in its scope [19]. Further, Lecoutre et al. propose the algorithm eSTR to enforce pairwise-consistency using only the size of the coarse blocks [20]. Our work is also related to the computation of subproblem interchangeability [4, 5, 7, 18], where the variables' domains (instead of constraints' relations) are dynamically partitioned by the constraints of a specified subproblem.

This paper is organized as follows. Section 2 gives some background information. Section 3 discusses relation partitioning. Section 4 describes how to create relation partitions and the data structures for storing them. Section 5 gives the partition-based algorithms for enforcing R(∗,m)C. Section 6 discusses our experiments and results. Finally, Section 7 concludes this paper.

2 Background

A constraint satisfaction problem (CSP) is defined by $\mathcal{P}= (\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X}$ is a set of variables, $\mathcal{D}$ is a set of domains, and $\mathcal{C}$ is a set of constraints. A variable in $\mathcal{X}$ has a finite domain in $\mathcal{D}$, and is constrained by constraints in $\mathcal{C}$. The constraints restrict the acceptable combinations of values for variables. A solution to the CSP is an assignment to each variable of a value taken from its domain such that all the constraints are satisfied. Deciding the existence of a solution for a CSP is NP-complete.

Each constraint $C_i \in \mathcal{C}$ is defined by a relation R_i specified over the *scope* of the constraint, *scope*(C_i), which is the set of variables to which the constraint applies. The *arity* of a constraint is the cardinality of its scope. In this paper, we study table constraints, where a tuple $t_i \in R_i$ is a combination of allowed values for the variables in *scope*(C_i). We call the *subscope* of a constraint a subset of its scope, and use it to denote the set of variables common to two constraints: *subscope*(C_i, C_j) = *scope*(C_i)∩ *scope*(C_j). We use the relational operator project, π, to restrict a partial assignment (e.g., a tuple) to a particular set of variables.

[1] PERTUPLE was originally called PROCESSQUEUE [16] and later renamed PERTUPLE to contrast it to another algorithm, ALLSOL, that guarantees the same result [8, 14].

Several graphical representations of a CSP exist. In the *hypergraph*, the vertices represent the variables of the CSP, and the hyperedges represent the scopes of the constraints. Figure 1 shows the hypergraph of our running example. In the *dual graph*, the vertices represent the CSP constraints, and the edges connect vertices representing constraints whose scopes overlap (see Figure 2). Thus, two CSP constraints are *adjacent* or *neighbors* in the dual graph when their subscope is not empty. The constraints of the dual graph enforce the equality of the variables in the subscope of the two adjacent CSP constraints.

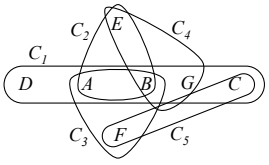


Fig. 1. Hypergraph of a CSP example

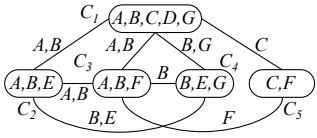


Fig. 2. Dual graph of CSP in Figure 1

Backtrack search is typically used to solve CSPs. To reduce the size of the search tree, CSPs are usually filtered by enforcing a given *local consistency property*. One common property is Generalized Arc Consistency (GAC). A CSP is GAC iff for every constraint, any value in the domain of any variable in the scope of the constraint can be extended to a tuple satisfying the constraint. While GAC is enforced by filtering the domains, other consistency properties are enforced by filtering the relations (which are then typically projected on the domains). Karakashian et al. proposed a relation-filtering algorithm that allows us to control the consistency level enforced while preserving the topology of the constraint network [16]. Their algorithm enforces R(∗,m)C, which guarantees that every relation is *minimal* in every combination of m relations.

Definition 1. *A set of m constraints $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$ with $m \geq 2$ is said to be R(∗,m)C iff every tuple in the relation of each constraint $C_i \in \mathcal{C}$ can be extended to the variables in $\bigcup_{C_j \in \mathcal{C}} scope(C_j) \setminus scope(C_i)$ in an assignment that satisfies all the constraints in $\mathcal{C}$ simultaneously. A network is R(∗,m)C iff every set of m constraints, $m \geq 2$, is R(∗,m)C.*

PerTuple, the algorithm for enforcing R(∗,m)C, ensures that each tuple in a relation appears in a solution of the dual CSP induced by the m relations by conducting a backtrack search on the tuples of the $m-1$ relations (see Figure 3).

Samaras and Stergiou showed that the constraints of the dual graph are piecewise functional [11, 22]. Given two CSP constraints with nonempty overlapping scopes, this property partitions the tuples of each relation into a set of *blocks*, equivalence classes of tuples, where each block is consistent with at most one block in the other relation (see Figure 4). PW-AC, their algorithm for enforcing pairwise-consistency, finds the partition induced on the relation of each constraint by each one of its neighbors in the dual graph. When a block of a

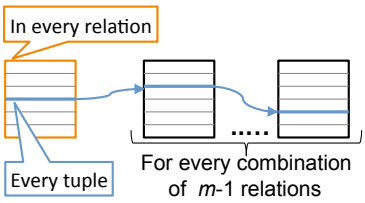


Fig. 3. Illustrating R(∗,m)C

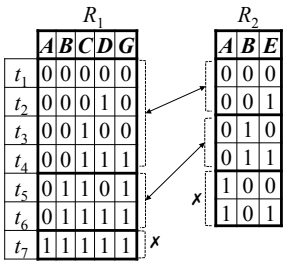


Fig. 4. Piecewise functional constraint

relation is not supported in one of the constraint's neighbors, the block's tuples are deleted. This operation may cause other blocks of the same relation to lose tuples, eventually becoming empty. Propagation stops when the network is pairwise-consistent or when a relation becomes empty. The effectiveness of PW-AC was established on sparse networks and other structured benchmarks [22].[2]

3 Relation Partitioning

We exploit the equivalence classes induced, on the relation R_i of a constraint C_i, by C_i's neighbors in the dual graph. We distinguish three types of such classes depending on the subset of neighbors considered: coarse (cb), fine (fb), and intermediate blocks (ib). Figures 5 and 6 illustrate the above for R_1. The notations and data structures used in the following sections refer to this example.

Coarse blocks: Any single neighbor of C_i in the dual graph partitions R_i into a set of coarse blocks. In Figure 5, $subscope(C_1,C_2) = \{A,B,C,D,G\} \cap \{A,B,E\} = \{A,B\} = o_1$. The tuples $t_{i\in[1,4]} \in R_1$ are equivalent for R_1 given o_1=00,[3] and consistent with (0,0,0) and (0,0,1) $\in R_2$. Indeed, $\pi_{o_1}(t_{i\in[1,4]} \in R_1) = (0,0)$ and $\pi_{o_1}((0,0,0) \in R_2) = \pi_{o_1}((0,0,1) \in R_2) = (0,0)$. Further, the above does not hold for any other tuple of R_1. Thus, $cb_1 = \{t_1,t_2,t_3,t_4\}$ is the coarse block of R_1 induced by o_1=00. The other two coarse blocks are $cb_2 = \{t_5,t_6\}$ and $cb_3 = \{t_7\}$. Similarly, $subscope(C_1,C_{j\in\{3,4,5\}})$ is $o_1 = \{A,B\}$, $o_2 = \{B,G\}$, and $o_3 = \{C\}$ respectively. Thus, o_1, o_2, and o_3 induce on R_1 the set of coarse blocks $\{cb_1,cb_2,cb_3\}$, $\{cb_4,cb_5,cb_6\}$, and $\{cb_7,cb_8\}$, respectively. Coarse blocks are the partitions identified and exploited by Samaras and Stergiou [22].

Fine blocks: When we consider all the constraints adjacent to C_i in the dual graph, they induce on R_i the finest possible partition, obtained by performing the intersections of all R_i's coarse blocks. As a result, they yield the (unique) set of R_i's fine blocks. In Figure 6, the set of fine blocks of R_1 is $\{fb_1,fb_2,\ldots,fb_5\}$.

[2] We suspect that PW-AC could be shown to be effective on dense networks had the redundant edges of the dual CSPs been removed [12,16].

[3] Abusing tuple/set assignment notation.

Intermediate blocks: Finally, the partition induced on a relation R_i by a given combination of m constraints depends on the neighboring constraints of C_i that are included in m. The granularity of that partition is intermediate: not finer than R_i's fine partition and not coarser than any of its coarse partitions. For example, $\{C_2, C_5\} \subset neighbors(C_1)$ induce the intermediate blocks $\{ib_1, ib_2, ib_3, ib_4\}$.

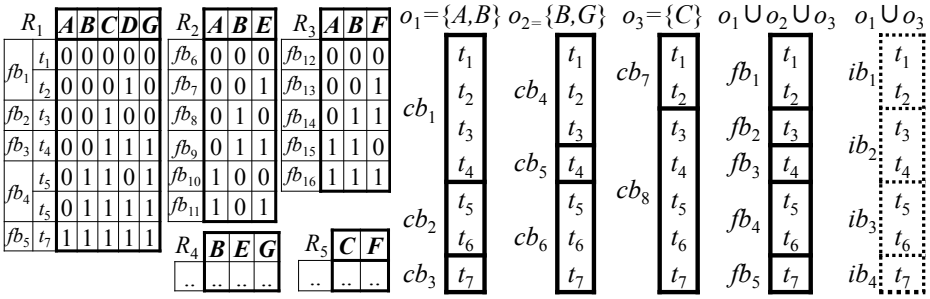


Fig. 5. Relations of CSP example **Fig. 6.** Coarse, fine, and intermediate blocks of R_1

We compute and store the fine and coarse blocks before preprocessing. PERFB, our new algorithm for enforcing R(∗,m)C, conducts backtrack search over the fine blocks, and uses the coarse blocks during lookahead. It does *not* permanently store any of the intermediate partitions.

4 Generating and Storing Partition Blocks

Because the partitions of a relation R_i are induced by an equivalence relation, any coarse or intermediate block of R_i is made up of a number of R_i's fine blocks (e.g., in Figure 6, $cb_1 = \{fb_1, fb_2, fb_3\}$ and $ib_2 = \{fb_2, fb_3\}$). Also, any fine block appears in exactly one block of a partition of a given granularity. For this reason, we first build the fine blocks of a relation, then we build its coarse blocks as sets of fine blocks. Below, we describe the data structures for storing the blocks (both fine and coarse) of a relation. Then, we describe how to generate them, and discuss their complexity.

4.1 Data Structures

Figure 7 partially depicts the data structures for the example of Figure 5. A coarse block is uniquely determined by three entities: a subscope, values of the subscope, and the relation that is being partitioned. For example, cb_1 of Figure 6 is determined by o_1, o_1=00, and R_1, and stored in $cb_{R1,o1=00}$.

The structure *rel-cb*$_{o_1=00}$ stores an entry for all relations R_i (i.e., R_1, R_2, R_3) with at least one tuple t_i where $\pi_{o_1}(t_i) = (0, 0)$. This entry points to the coarse

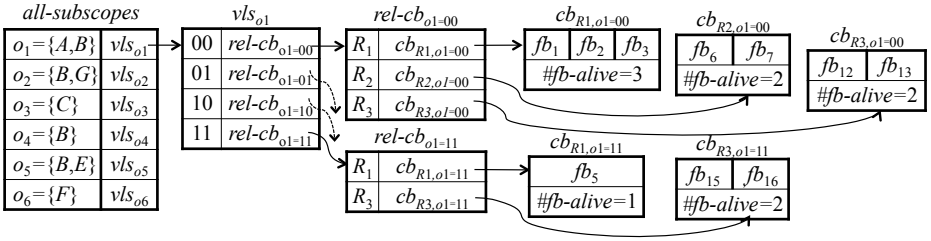


Fig. 7. Data structures for storing coarse and fine blocks for the CSP in Figure 5

blocks of R_i with o_1=00. Further, a back-pointer, not shown in Figure 7 for readability, links each such coarse block back to $rel\text{-}cb_{o_1=00}$. All the coarse blocks accessible from $rel\text{-}cb_{o_1=00}$ are pairwise consistent. If a relation R_i has the subscope o_1 but does not have an entry in $rel\text{-}cb_{o_1=00}$, or if R_i loses during constraint propagation all its tuples t_i where $\pi_{o_1}(t_i) = (0, 0)$, then all the coarse blocks accessible from $rel\text{-}cb_{o_1=00}$ are inconsistent. Thus, given any coarse block with o_1=00 (e.g., $cb_{R_2,o_1=00}$), $rel\text{-}cb_{o_1=00}$ gives us constant-time access to the coarse blocks that are pairwise consistent with it in all relations.

The structure vls_{o_1} gives access to the structures storing $rel\text{-}cb_{o_1=v_i}$ for each subscope value v_i of o_1 (i.e., 00, 01, 10, 11). For each v_i, there is one such entry.

The structure *all-subscopes* gives access to the structures storing the subscopes values vls_{o_i} for each subscope o_i in the problem. For example, for $o_1 = \{A, B\}$, *all-subscopes* gives access to vls_{o1}.

In addition to the above structures, we use two constant-time lookup tables. The table $fb\text{-}subscope\text{-}2\text{-}cb$ gives the coarse block in which a fine block of a relation appears given a subscope. For example, $fb\text{-}subscope\text{-}2\text{-}cb[fb_2, R_1, \{A, B\}]$ points to the coarse block $cb_{R_1,o_1=00}$ shown in Figure 7. Similarly, $tup\text{-}2\text{-}fb$ maps a tuple t_i in a relation to the fine block that contains t_i. In the example in Figures 5 and 6, it maps $t_5 \in R_1$ to the structure storing fb_4.

4.2 Fine Blocks

CreateFineBlocks (Algorithm 1) generates the fine blocks of a relation R_i. We use the following accessors and notations:

- The function FineBlocks(R_i) returns the set of fine blocks of R_i.
- The accessor $\#tuples\text{-}alive(fb_i)$ gives the count of living tuples of fb_i. This count is stored in the structure of fb_i, and updated during search whenever a tuple in fb_i is marked or unmarked as deleted.
- The accessor $\#fb\text{-}alive(R_i)$ gives the number of fine blocks alive in R_i.
- The function FindEquivFB(R_i, $subTuple$) is a relational selection operator: it iterates over the fine blocks in FineBlocks(R_i), and returns the fine block with the values assignment matching $subTuple$.

CreateFineBlocks (Algorithm 1) groups tuples with the same values assignments for the variables in $U_s = \bigcup_{C_j \in neighbors(C_i)} subscope(C_i, C_j)$. In addition to grouping tuples, it keeps a counter storing the total number of living tuples in the given fine partition. When a tuple is deleted during search, we use the array *tup*-2-fb to update the count of living tuples in the tuple's fine block.

Algorithm 1. Creating the fine partition of a relation R_i

```
1: function CreateFineBlocks(R_i)
2:     FineBlocks(R_i) ← ∅
3:     U_s ← ⋃_{C_j ∈ neighbors(C_i)} subscope(C_i, C_j)
4:     for each tuple τ ∈ R_i do
5:         subTuple ← π_{U_s}(τ)
6:         fb_curr ← FindEquivFB(R_i, subTuple)
7:         if fb_curr = nil then
8:             fb_curr ← create a new fine block
9:             FineBlocks(R_i) ← FineBlocks(R_i) ∪ {fb_curr}
10:            #fb-alive(R_i) ← #fb-alive(R_i) + 1
11:        fb_curr ← fb_curr ∪ {τ}
12:        #tuples-alive(fb_curr) ← #tuples-alive(fb_curr) +1
13:        tup-2-fb[τ] ← fb_curr
14:    return FineBlocks(R_i)
```

Complexity: We use the following parameters in the complexity analysis. t is the maximum number of tuples in a relation; k is the maximum constraint arity; $e = |\mathcal{C}|$ is the number of constraints in the CSP; and $|o_i|$ is the size of the largest subscope that a relation shares with a neighbor. The time complexity of each of FineBlocks(R_i) and *#fb-alive*(R_i) is $O(1)$. The creation of U_s on Line 3 of Algorithm 1 is $O(e \cdot |o_i| \cdot log(|o_i|))$ because a constraint may be adjacent to all other constraints in the dual graph, and each edge requires inserting at most $|o_i|$ variables into the set. In the worst case, the function FindEquivFB performs k comparisons on t fine blocks to find an existing equivalent fine block. Its complexity is thus $O(t \cdot k)$. Computing the subTuple in Line 5 is $O(k)$. Thus, CreateFineBlocks is $O(e \cdot |o_i| \cdot log(|o_i|) + (t \cdot (k + k \cdot t))) = O(k \cdot t^2 + e \cdot |o_i| \cdot log(|o_i|))$. When $arity(C_i) = |U_s|$, Lines 5–10 are bypassed. In this case, each fine block of R_i has a single tuple. Thus, while the time complexity is large, the cost of Lines 5–10 is incurred only when a fine block can potentially have more than one tuple.

The space complexity for storing the fine blocks of R_i is $O(t)$, incurred when each fine block in R_i has one tuple. The array mapping tuples to the fine blocks to which they belong is $O(t)$, making the total space complexity $O(t)$ per constraint.

4.3 Coarse Blocks

Below, we describe the creation of the coarse blocks, similar to the ones introduced by Samaras and Stergiou [22]. Our design improves on theirs in that *a*) a

coarse block stores fine blocks and not tuples, thus, potentially reducing the size of each coarse block, and *b*) we iterate over subscopes, rather than pairs of relations (e.g., in Figures 5 and 6, R_1 has four neighbors but only three subscopes).

CreateCoarseBlocks (Algorithm 2) generates the coarse blocks of a relation R_i for a subscope o_i out of R_i's existing fine blocks. It first retrieves all the values of o_i (Line 2). Then, it iterates over the fine blocks fb_i of R_i, adding each fb_i to the appropriate coarse partition. When a coarse block does not exist (Line 9), it is created (Line 10). The accessor function *#fb-alive*(cb_i) maintains the number of fine blocks alive in the coarse block cb_i. This function has the same name as *#fb-alive*(R_i) (function-name overload), which operates in a similar manner.

Algorithm 2. Creating the coarse partition of a relation R_i given a subscope o_i

```
1: function CreateCoarseBlocks(R_i, o_i)
2:     vls_{o_i} ← all-subscopes[o_i]
3:     for each fb_i ∈ FineBlocks(R_i) do
4:         if #tuples-alive(fb_i) = 0 then
5:             continue
6:         v_i ← π_{o_i}(fb_i)
7:         if vls_{o_i}[v_i] does not exist then
8:             vls_{o_i}[v_i] ← create new rel-cb table
9:         curr-rel-cb ← vls_{o_i}[v_i]
10:        curr-cb ← curr-rel-cb[R_i]
11:        if curr-cb does not already exist then
12:            curr-cb ← create new coarse block
13:        curr-cb ← curr-cb ∪ {fb_i}
14:        #fb-alive(curr-cb) ← #fb-alive(curr-cb) + 1
15:        fb-subscope-2-cb[fb_i, R_i, o_i] ← curr-cb
16:    return vls_{o_i}
```

Complexity: The table lookups on Lines 2 and 10 of CreateCoarseBlocks are $O(1)$. The table lookups for vls_{o_i} on Lines 6-9 are $O(\log(t) \cdot |o_i|)$ if vls_{o_i} is represented as a binary search tree that uses v_i as a key. Creating v_i on Line 6 is $O(|o_i|)$. The for-loop is executed $O(t)$ times. The complexity for CreateCoarseBlocks is thus $O(|o_i| \cdot t \cdot \log(t))$.

The time complexity to get the specific coarse block to which a particular fine block belongs provided a subscope is $O(1)$ thanks to the *fb-subscope-2-cb* table. The time complexity to query all fine blocks in a relation R_j that are consistent with a coarse block corresponding to a relation R_i is $O(1)$ thanks to the back-pointer to the *rel-cb* table stored in each coarse block.

The space complexity for a set of coarse blocks for a single subscope and relation is $O(t)$ because each fine block is in exactly one coarse block for a given subscope. The space complexity for all coarse blocks is then $O(k \cdot t \cdot e^2)$ because, in the worst case, each relation is partitioned by every other relation in the

problem, and each coarse block is identified by a subtuple of size k. Additionally, the *fb-subscope-2-cb* table requires $O(e^2 \cdot t)$ space, as each tuple is in exactly one coarse block.

Note that the coarse blocks have the same space complexity as the index-tree data structure [16], but can be more efficiently queried.

5 Consistency Algorithm: From PerTuple to PerFB

Below, we describe PerFB and FB-SearchSupport, which improve PerTuple and SearchSupport [16], respectively, for enforcing R(∗,m)C on a CSP. Like PerTuple, PerFB takes as input $\mathcal{Q}$ and Φ. Φ is the set of all combinations of m relations. The queue $\mathcal{Q}$ is initialized to all the combination-relations pairs $\langle \varphi, R_i \rangle$ such that $\varphi \in \Phi$ and $R_i \in \varphi$. PerFB iterates over all fine blocks of a relation R_i in a combination φ, calling FB-SearchSupport to ensure that a fine block can be extended to a solution in the dual CSP induced by φ by conducting a backtrack search that maintains support structures. In addition to the *static* fine and coarse blocks, PerFB and FB-SearchSupport make use of intermediate blocks, *dynamically* induced by the relations in φ. In this section, we abuse the notations and use *subscope*(R_i, R_j), *scope*(R_i), *neighbors*(R_i) to refer to *subscope*(C_i, C_j), *scope*(C_i), *neighbors*(C_i), respectively.

5.1 PerFB

PerFB (Algorithm 3) improves PerTuple [16] in two ways in order to reduce the number of costly calls to FB-SearchSupport:

1. PerFB ensures that all fine blocks, rather than all tuples, in a relation R_i can be extended to a solution over the relations of a combination φ of size m (Line 11). This difference can reduce the number of calls to FB-SearchSupport.
2. The number of calls to FB-SearchSupport can be further reduced by exploiting the dynamically induced intermediate blocks.

We use the following additional notations to describe how PerFB operates.

- The accessor $CB((R_i, fb_i), o_i)$ retrieves the coarse block of R_i containing the fine block fb_i given the subscope o_i. It uses the table *fb-subscope-2-cb*.
- The accessor *Support*$(R_j, CB(R_i, fb_i, o_{ij}))$ retrieves the coarse block of R_j containing the fine blocks consistent with fb_i of relation R_i given o_{ij} = *subscope*(C_i, C_j). To this end, it uses the back-pointer to the *rel-cb* structures from the coarse block $CB(R_i, fb_i, o_{ij})$ and accesses *rel-cb*$[R_j]$.
- The structure *shared-fvars*$[l]$ stores, at the search level l in FB-SearchSupport where R_i is 'assigned' a fine block, the variables in $\bigcup_{R_j \in \varphi}$ *subscope*(R_i, R_j).

In Figure 6, fb_2 and fb_3 are equivalent in $\varphi = \{R_1, R_2, R_5\}$ yielding $ib_2 = \{fb_2, fb_3\}$ for R_1 by $\{o_1 \cup o_3\}$. PerFB exploits such intermediate blocks. The key

Algorithm 3. Enforces R(∗,m)C using a queue $\mathcal{Q}$ and list Φ of combinations

```
 1: function PerFB(Q, Φ)
 2:     while Q ≠ ∅ do
 3:         ⟨φ, R_i⟩ ← Pop(Q)
 4:         deleted ← false
 5:         R_f ← φ \ R_i
 6:         for R_j ∈ φ do
 7:             equiv-FBs[R_j] ← ∅
 8:         for i = 1 to m do
 9:             shared-fvars[i] ← ∅
10:         shared-fvars[1] ← ⋃_{R_j ∈ R_f} subscope(R_i, R_j)
11:         for each living fb_i ∈ FineBlocks(R_i) do
12:             v_i ← π_{(shared-fvars[1])}(fb_i)
13:             if equiv-FBs[R_i, v_i] does not exist then
14:                 equiv-FBs[R_i, v_i] ← FB-SearchSupport(fp_i, ⟨R_i, R_f⟩, equiv-FBs)
15:             if equiv-FBs[R_i, v_i] = false then
16:                 for each tuple τ ∈ fb_i do
17:                     Delete(τ, R_i)
18:                 deleted ← true
19:                 #fb-alive(R_i) ← #fb-alive(R_i) − 1
20:                 if #fb-alive(R_i) = 0 then
21:                     return inconsistent
22:         if deleted then
23:             for each φ' ∈ (Φ \ {φ}), R_i ∈ φ' do
24:                 for each R' ∈ (φ' \ {R_i}) do
25:                     Q ← Q ∪ {⟨φ', R'⟩}
26:     return consistent
```

to dynamically identifying them is the table $equiv\text{-}FBs[R_i, v_i]$, which is created at each call to PerFB, and returns $true$ or $false$, given a relation $R_i \in \varphi$, and a subset of values v_i from a fine block fb_i. The subset v_i is determined by projecting fb_i over the variables in *shared-fvars*[1] (Lines 10–12 in Algorithm 3). Any other fine block of R_i with the same v_i is necessarily in the same intermediate block. Thus, before executing FB-SearchSupport, we check $equiv\text{-}FBs[R_i, v_i]$ to see if a result for this particular v_i was already found (Line 13). If so, the result is reused. Otherwise, FB-SearchSupport is called, and its result stored for future use (Line 14). Similar to PerTuple, when fb_i has no support, its tuples are marked as deleted, and the count of fine blocks alive in R_i is decremented (Lines 15–19). Inconsistency is detected when all fine blocks in R_i are deleted (Lines 20–21). The updates of $\mathcal{Q}$ are identical to those in PerTuple. The only relation used to access $equiv\text{-}FBs$ in PerFB is R_i. Other relations' entries in $equiv\text{-}FBs$ are discussed in Section 5.2.

When $|shared\text{-}fvars[1]| = arity(C_i)$, PerFB reduces to PerTuple because no two fine blocks in R_i are equivalent. In this case, the discovery of equivalent fine blocks is bypassed to save on CPU time and memory.

Finally, note that $m = 2$ does not require any calls to FB-SearchSupport. For this reason, for $m = 2$, we use PW-AC [22] during preprocessing and PerFB during search. Further, because, when $m = 2$, the intermediate blocks are exactly the stored coarse blocks, checking whether or not a *coarse* block is consistent can be done in constant time by checking the $\#fb\text{-}alive(R_j)$ of the coarse block returned by $Support(R_j, CB(R_i, fb_i, o_{ij}))$, where R_j is the other relation in the combination. Thus, intermediate blocks are not used.

5.2 FB-SearchSupport

FB-SearchSupport performs backtrack search with forward checking on the subproblem induced, on the dual of the CSP $\mathcal{P}$, by the relations in the combination φ, denoted as $\mathcal{P}_{D\varphi}$. The variables of $\mathcal{P}_{D\varphi}$ are the relations in the combination $\varphi = \{R_i\} \cup \mathcal{R}_f$. The 'variable' R_i is assigned the 'value' fb_i in the search. FB-SearchSupport is called with the argument $(fb_i, \langle R_i, \mathcal{R}_f \rangle, equiv\text{-}FBs)$. The

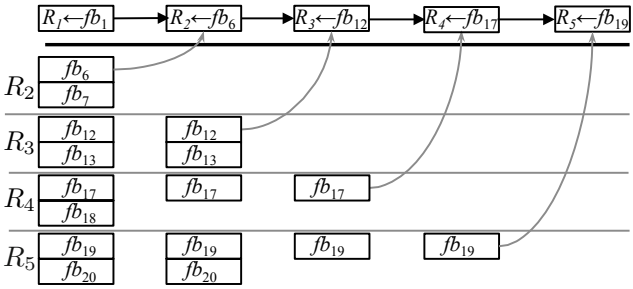


Fig. 8. Backtrack search on fine blocks using coarse and intermediate blocks

domain of a relation $R_j \in \mathcal{R}_f$ is the set of fine blocks (represented by their indices) in the coarse block returned by $Support(R_j, CB((R_i, fb_i), o_{ij}))$, where o_{ij} is the subscope of R_i and R_j. However, coarse blocks are not defined when R_i and R_j are not neighbors ($subscope(R_i, R_j) = \emptyset$). Thus, the 'domain' of R_j is either *a*) the set of living fine blocks from FineBlocks(R_j) when no relation adjacent to R_j has been instantiated, or *b*) the set of fine blocks in the coarse block $Support(R_j, CB((R_k, fb_k), o_{jk}))$, where R_k, a relation adjacent to R_j, was 'assigned' fp_k.

Figure 8 illustrates forward checking in FB-SearchSupport using the example from Figure 5. Assume R_4 contains only tuples $(0,0,0)$ and $(0,1,0)$, and R_5 only $(0,0)$ and $(0,1)$, denoted $fb_{17}, fb_{18}, fb_{19}, fb_{20}$, respectively. When $R_2 \leftarrow fb_6$, forward checking removes fb_{18} from the domain of the dual variable R_4. As mentioned above, each fine block has an accessor index. The set of fine blocks in a coarse block is represented by a sorted array of indices. Thus, the 'intersection' of the current domain of R_4 and $cb_x(R_4)$, where $cb_x(R_4) = Support(R_4, CB(R_2, fb_6), o_{24})$ is performed by iterating over the index of each

fine block in the current domain of R_4, performing a binary search on the fine block indices of $cb_x(R_4)$, and removing, from the current domain of R_4, the indices of the fine blocks not listed in $cb_x(R_4)$.

We further exploit the intermediate partitions in the subproblem $\mathcal{P}_{D\varphi}$ in FB-SearchSupport in order to *bypass the exploration of entire redundant subtrees during search*. While this mechanism did not yield significant savings in the number of nodes visited in our experiments for finding one solution, it may prove useful when we search for all solutions (i.e., AllSol). Fine blocks are passed over for instantiation by observing the following:

1. When instantiating a relation R_j at level l, we initialize $\mathit{shared\text{-}fvars}[l] \leftarrow \bigcup_{R_k \in \mathcal{R}_f} \mathit{subscope}(R_j, R_k)$.
2. Prior to instantiating $R_j \leftarrow fb_j$ at level l in search, we check in $\mathit{equiv\text{-}FBs}$ whether or not an equivalent fine block was already instantiated. That is, we check $\mathit{equiv\text{-}FBs}[R_j, v_j]$ where $v_j = \pi_{\mathit{shared\text{-}fvars}[l]}(fb_j)$. If the entry is $false$, fb_j need not be instantiated because an equivalent fine block in the same intermediate partition was already found inconsistent on a previous path in the same search. (Note that the entry cannot be *true* because search terminates after finding the first solution.) When the domain of a future 'variable' is annihilated during forward checking for v_j, $\mathit{equiv\text{-}FBs}[R_j, v_j]$ is marked as $false$.
3. When unlabeling a 'variable' R_j at a level l (upon backtracking), $\mathit{equiv\text{-}FBs}[R_j]$ and $\mathit{shared\text{-}fvars}[l]$ are set to $\emptyset$.

Complexity. When deleting a tuple during search, it is important to maintain the correct counts of fine and coarse blocks. Each tuple deletion costs $O(e^2)$ updates. Updates are constant time thanks to the fb-*subscope*-2-*cb* and *tup*-2-fp tables. The cost of these updates is, in practice, greatly dwarfed by that of FB-SearchSupport. The time complexity of PerFB is identical to that of PerTuple, and dominated by the $O(t^{m-1})$ search conducted in FB-SearchSupport [16]. Additionally, PerFB performs at most as many calls to FB-SearchSupport as PerTuple does, because $\bigcup_{R_j \in \varphi \setminus \{R_i\}} \mathit{subscope}(R_i, R_j)$ is the same as $\mathit{scope}(R_i)$ in the worst case, and all fine blocks have a single tuple. Insertion and retrieval of equivalent fine blocks for R_i is done in $O(k \cdot log(t))$ time. Indeed, the entry for $\mathit{equiv\text{-}FBs}[R_i]$ is a binary search tree with sub-tuples of values v_i as its keys, comparing each node in the tree is $O(k)$, and $O(log(t))$ comparisons may be required when each fine block has only one tuple.

At each level of search in FB-SearchSupport, $\mathit{equiv\text{-}FBs}$ holds $O(t)$ fine blocks, each represented by a sub-tuple of size $O(k)$. Thus, an additional $O(m \cdot t \cdot k)$ space is required for PerFB to store the equivalent fine blocks at each level of search in FB-SearchSupport.

6 Empirical Evaluations

We compare the performance of PerFB to that of PerTuple. We use the latest strategy for enforcing R(∗,m)C obtained after removing redundant edges from

the dual graph [12], localizing consistency propagation to the clusters of a tree decomposition of the CSP, and bolstering propagation between adjacent clusters by the addition of constraint projections to the clusters' separators [15]. (The corresponding consistency property is denoted cl+proj-wR($*$,m)C.) Although weakening the dual graph weakens consistency for $m > 2$, it also reduces the number of combinations and, thus, cost. Importantly, localization of the constraints to clusters is an excellent 'set up' for testing intermediate partitions. For both PerFB and PerTuple, we used $m = \{2, 3, 4, |\psi(cl)|\}$ where $m = |\psi(cl)|$ is the number of constraints in a cluster in the tree decomposition and corresponds to enforcing the minimality of each cluster .

In our experiments, we find the first solution of an instance by backtrack search, using the dynamic variable ordering dom/deg and doing full lookahead with relational consistency (i.e., cl+proj-wR($*$,m)C for $m = \{2, 3, 4, |\psi(cl)|\}$). For the evaluation, we use benchmarks from the CSP Solver Competition that are either hard to solve, thus requiring high levels of consistency, or are challenging for R($*$,m)C, thus demonstrating the effectiveness of partitioning.[4] We limit maximum processing time to 2 hours, and the maximum memory allocation to 8GB. All CPU times are reported in seconds and include *all* processing operations, including data-structure creation, preprocessing, and search.

Table 1 lists the min, max, and mean values of the fine block sizes averaged over all instances in a benchmark, as well as the size of the largest block in any instance in the benchmark. Table 2 lists similar results for the coarse blocks. Benchmarks not shown in Table 1 all have one tuple per fine block. While the average size tends to be fairly small, some benchmarks show rather large values (e.g., modifiedRenault and tightness0.9). Even though the block sizes may seem small, our technique remains beneficial because cluster-based R($*$,m)C (i.e., cl+proj-wR($*$,m)C) restricts the neighborhood of a relation by localization.

Table 1. Absolute and averaged size of fine blocks

	Absolute	Averages		
Benchmark	Max	Min	Max	Mean
geom	17	1.0	1.2	1.0
graphColoring-hos	3	1.0	2.0	1.0
graphColoring-sgb-book	12	1.0	7.7	1.1
hanoi	2	1.0	2.0	1.0
modifiedRenault	260	1.0	25.6	1.0
rand-10-20-10	2	1.0	1.3	1.0
renault	4	1.0	4.0	1.0
ssa	8	1.0	3.1	1.1
tightness0.9	38	1.0	28.1	1.0
varDimacs	16	1.0	3.4	1.1

[4] Aim-(50, 100, 200), composed-(25-1-2, 25-1-25, 25-1-40, 25-1-80, 25-10-20, 75-1-2, 75-1-25, 75-1-40, 75-1-80), dag-rand, dubois, geom, graphColoring-(hos, mug, register-mulsol, sgb-book, sgb-games, sgb-queen), hanoi, lexVg, modifiedRenault, pret, pseudo-aim, rand-(10-20-10, 3-20-20-fcd), renault, rlfapGraphsMod, rlfapScensMod, ssa, super-queens, tightness0.9, varDimacs.

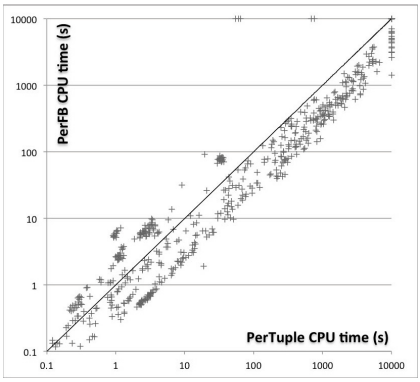
Table 2. Absolute and averaged size of coarse blocks

	Abs	Averages				Abs	Averages		
Benchmark	**Max**	**Min**	**Max**	**Mean**	**Benchmark**	**Max**	**Min**	**Max**	**Mean**
aim-50,100,200,pseudo	4	1.0	4.0	2.1	grCol-sgb-queen	17	10.3	10.3	10.3
cmpsed-25-1-2,25,40,80	10	1.0	10.0	8.0-8.4	hanoi	3	1.0	3.0	2.9
cmpsed-25-10-20	10	1.0	10.0	7.6	lexVg	875	1.0	484.7	3.6
cmpsed-75-1-2,25,40,80	10	1.0	10.0	8.3-8.5	modifiedRenault	48,720	1.0	48,720.0	7.9
dag-rand	108	1.0	91.6	2.9	rand-10-20-10	1,046	1.0	119.2	1.3
dubois,pret	2	1.0	2.0	1.5	rand-3-20-20-fcd	190	1.0	181.5	12.8
geom	20	6.4	20.0	15.0	renault	48,720	1.0	48,720.0	7.7
grCol-hos	6	1.0	3.3	3.3	rlfapGr/ScensMod	44,43	1.0	30.0,35.6	18.5,19.4
grCol-mug	3	1.0	2.5	2.4	ssa	31	1.0	14.7	2.1
grCol-register-mulsol	48	23.2	23.2	23.2	super-queens	49	15.6	17.6	16.4
grCol-sgb-book	12	1.0	7.7	7.5	tightness0.9	40	1.0	36.3	16.9
grCol-sgb-games	8	1.0	6.3	6.1	varDimacs	512	1.0	115.0	5.6

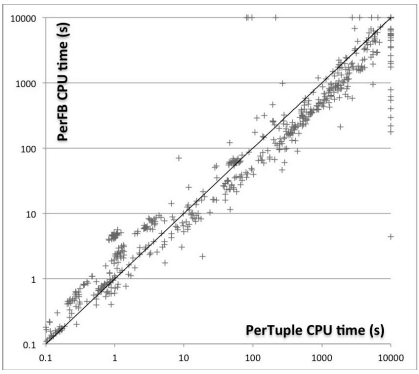
Table 3 summarizes our results. It reports the numbers of instances completed (#Completed) by each algorithm, those completed *only by* one algorithm, and those completed *by both* algorithms. It also reports the average CPU time, the number of calls to SearchSupport or FB-SearchSupport, and their ratio. For each value of m, the average CPU time is computed over instances completed by both algorithms. The best values are bolded. Note that the entry for SearchSupport calls for $m = 2$ is blank because PerFB does not call FB-SearchSupport in this case.

PerFB clearly wins across the board. While few instances are solved only by PerTuple, many more are solved only by PerFB. The discovery and exploitation of equivalent fine blocks during PerFB clearly greatly reduces the number of calls to find a support, with the poorest reduction ($m = 4$) still reducing the number of searches by over half. This saving is reflected in the reduction of CPU time because the cost of searching for a support in a combination of m relations is much larger than that of identifying and storing equivalent fine blocks (see Section 5). Although not shown here, the average percentages of nodes visited that were skipped in FB-SearchSupport thanks to the usage of intermediate partitions are .01%, .04%, and .10% for $m = 3, 4$, and ψ, respectively. Thus, the use of intermediate partitions during FB-SearchSupport is largely ineffectual when finding a single solution to the subproblem. (However, it may be useful for improving the performance of AllSol.)

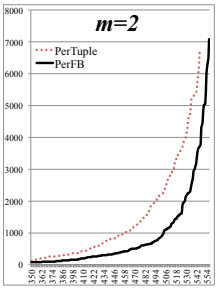
The scatter plots in Figure 9a and 9b compare the CPU time of PerFB and PerTuple for solving all 853 instances for $m = 2$ and $m = |\psi(cl)|$. Marks below the diagonal line represent instances where PerFB outperformed PerTuple. Marks on the right (top) border denote instances that timed out only for PerTuple (PerFB). Where PerTuple outperforms PerFB, the instances are 'easier' and the time difference is negligible for the majority of these (note the logarithmic scale). On the other hand, for hard instances, PerFB is faster. This difference is likely due to the cost of identifying the intermediate partitions in PerFB; easy instances tend to not make use of the intermediate partitions, but may still incur the cost of identifying them. The cumulative charts in Figures 9c, 9d, 9e, and 9f display the number of instances completed within a given time



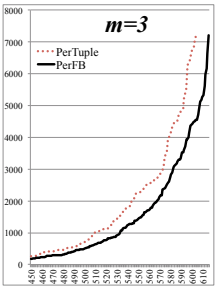
(a) $m = 2$



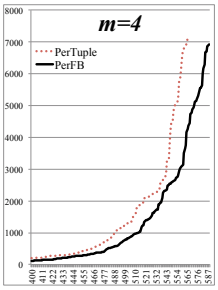
(b) $m = |\psi(cl)|$



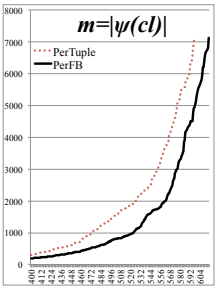
(c) $m = 2$



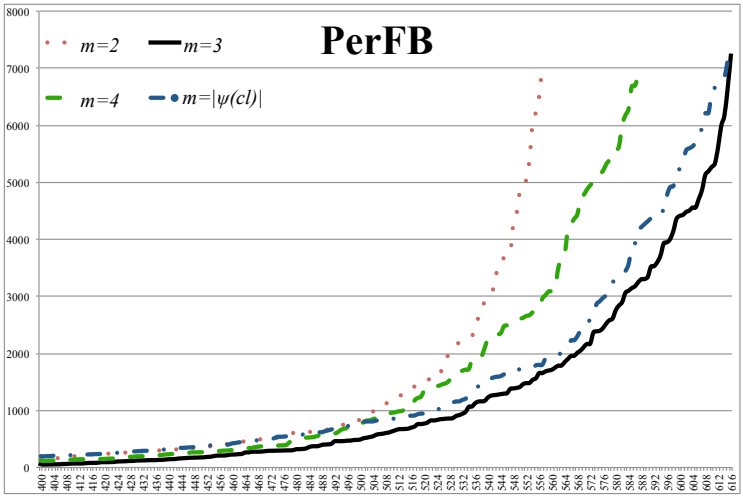
(d) $m = 3$



(e) $m = 4$



(f) $m = |\psi(cl)|$



(g) PerFB for $m = 2, 3, 4, |\psi(cl)|$

Fig. 9. Pairwise comparisons of PerFB and PerTuple for tested values of m

Table 3. Summary of results for each tested value of m over 853 instances

	$m=2$		$m=3$		$m=4$		$m=\|\psi(cl)\|$	
	PerTuple	PerFB	PerTuple	PerFB	PerTuple	PerFB	PerTuple	PerFB
#Completed	546	**557**	604	**616**	566	**589**	597	**615**
... only by	5	**16**	1	**13**	2	**25**	8	**26**
... by both	541		603		564		589	
Avg. CPU (sec)	538	**227**	521	**362**	472	**314**	669	**458**
SearchSupport calls (10^9)	86.4	0	88.1	26.1	52.7	19.6	24.7	8.1
ratio	–		3.37		2.69		3.06	

by each algorithm, and show that PerFB outperforms PerTuple for every m. Figure 9g compares PerFB for varying values of m. PerFB with $m = 3, |\psi(cl)|$ are the clear winners on the tested benchmarks.

We establish statistical significance by running a one-tailed paired t-test on instances completed by both PerFB and PerTuple for each value of m. The tests give $p < .01$ for each value of m. Thus, the two algorithms are extremely unlikely to have equivalent performances. This result and those in Table 3 and Figure 9 support our hypothesis that PerFB outperforms PerTuple.

7 Conclusion and Future Work

Given the importance of minimal CSPs in reasoning [9] and higher-level consistencies in solving difficult problems [13], it seems important to improve the performance of the techniques for enforcing them. In this paper, we extend the work of Samaras & Stergiou [22] to improve the initial algorithm of Karakashian et al. [16] for relational consistency by exploiting blocks of equivalent tuples at various levels of granularity, and we empirically validate our approach.

We need to evaluate the effectiveness of the approach on AllSol, the alternative algorithm for minimality [8,14]. We believe that applying the ideas explored in this paper to join computation in relational databases is a promising next step [17], potentially highly rewarding in practice.

References

1. Bessiere, C.: Constraint Propagation. In: Handbook of Constraint Programming, Elsevier (2006)
2. Bessiere, C., Cardon, S., Debruyne, R., Lecoutre, C.: Efficient Algorithms for Singleton Arc Consistency. Constraints 16(1), 25–53 (2011)
3. Bessière, C., Stergiou, K., Walsh, T.: Domain Filtering Consistencies for Non-Binary Constraints. Artificial Intelligence 172, 800–822 (2008)

4. Choueiry, B.Y., Davis, A.M.: Dynamic Bundling: Less Effort for More Solutions. In: Koenig, S., Holte, R. (eds.) SARA 2002. LNCS (LNAI), vol. 2371, pp. 64–82. Springer, Heidelberg (2002)
5. Choueiry, B.Y., Noubir, G.: On the Computation of Local Interchangeability in Discrete Constraint Satisfaction Problems. In: AAAI 1998, pp. 326–333 (1998)
6. Dechter, R., van Beek, P.: Local and Global Relational Consistency. Theor. Comput. Sci. 173(1), 283–308 (1997)
7. Freuder, E.C.: Eliminating Interchangeable Values in Constraint Satisfaction Problems. In: AAAI 1991, pp. 227–233 (1991)
8. Geschwender, D., Karakashian, S., Woodward, R., Choueiry, B.Y., Scott, S.D.: Selecting the Appropriate Consistency Algorithm for CSPs Using Machine Learning Techniques. In: Pre-PhD Student Abstract and Poster Program of AAAI 2013, pp. 1611–1612 (2013)
9. Gottlob, G.: On Minimal Constraint Networks. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 325–339. Springer, Heidelberg (2011)
10. Gyssens, M.: On the Complexity of Join Dependencies. ACM Trans. Database Systems 11(1), 81–108 (1986)
11. Van Hentenryck, P., Deville, Y., Teng, C.-M.: A Generic Arc Consistency Algorithm and its Specializations. Artificial Intelligence 57, 291–321 (1992)
12. Janssen, P., Jégou, P., Nougier, B., Vilarem, M.-C.: A Filtering Process for General Constraint-Satisfaction Problems: Achieving Pairwise-Consistency Using an Associated Binary Representation. In: IEEE Workshop on Tools for AI, pp. 420–427 (1989)
13. Jeavons, P., Petke, J.: Local Consistency and SAT-Solvers. JAIR 43, 329–351 (2012)
14. Karakashian, S.: Practical Tractability of CSPs by Higher Level Consistency and Tree Decomposition. PhD thesis, University of Nebraska-Lincoln (2013)
15. Karakashian, S., Woodward, R., Choueiry, B.Y.: Improving the Performance of Consistency Algorithms by Localizing and Bolstering Propagation in a Tree Decomposition. In: AAAI 2013, pp. 466–473 (2013)
16. Karakashian, S., Woodward, R., Reeson, C., Choueiry, B.Y., Bessiere, C.: A First Practical Algorithm for High Levels of Relational Consistency. In: AAAI 2010, pp. 101–107 (2010)
17. Lal, A., Choueiry, B.Y.: Constraint Processing Techniques for Improving Join Computation: A Proof of Concept. In: Kuijpers, B., Revesz, P.Z. (eds.) CDB 2004. LNCS, vol. 3074, pp. 143–160. Springer, Heidelberg (2004)
18. Lal, A., Choueiry, B.Y., Freuder, E.C.: Neighborhood Interchangeability and Dynamic Bundling for Non-Binary Finite CSPs. In: AAAI 2005, pp. 387–404 (2005)
19. Lecoutre, C., Likitvivatanavong, C., Yap, R.H.C.: A Path-Optimal GAC Algorithm for Table Constraints. In: ECAI 2012, pp. 510–515 (2012)
20. Lecoutre, C., Paparrizou, A., Stergiou, K.: Extending STR to a Higher-Order Consistency. In: AAAI 2013, pp. 576–582 (2013)
21. Paparrizou, A., Stergiou, K.: An Efficient Higher-Order Consistency Algorithm for Table Constraints. In: AAAI 2012 (2012)
22. Samaras, N., Stergiou, K.: Binary Encodings of Non-binary Constraint Satisfaction Problems: Algorithms and Experimental Results. JAIR 24, 641–684 (2005)
23. Woodward, R., Karakashian, S., Choueiry, B.Y., Bessiere, C.: Solving Difficult CSPs with Relational Neighborhood Inverse Consistency. In: AAAI 2011, pp. 112–119 (2011)
24. Woodward, R.J., Karakashian, S., Choueiry, B.Y., Bessiere, C.: Revisiting Neighborhood Inverse Consistency on Binary CSPs. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 688–703. Springer, Heidelberg (2012)