HAL
open science

Maintaining Arc Consistency with Multiple Residues

Christophe Lecoutre, Chavalit Likitvivatanavong, Scott Shannon, Roland Yap, Yuanlin Zhang

To cite this version:

Christophe Lecoutre, Chavalit Likitvivatanavong, Scott Shannon, Roland Yap, Yuanlin Zhang. Maintaining Arc Consistency with Multiple Residues. Constraint Programming Letters (CPL), 2008, 2, pp.3-19. hal-00868069

HAL Id: hal-00868069

https://hal.science/hal-00868069

Submitted on 1 Oct 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Constraint Programming Letters 2 (2008) 3–19 Submitted 4/2007; Published 1/2008

Maintaining Arc Consistency with Multiple Residues

Christophe Lecoutre LECOUTRE@CRIL.FR
CRIL-CNRS FRE 2499, Université d'Artois, Lens, France

Chavalit Likitvivatanavong CHAVALIT@COMP.NUS.EDU.SG
School of Computing, National University of Singapore, Singapore

Scott G. Shannon EVERSABRE@HOTMAIL.COM
Department of Computer Science, Texas Tech University, USA

Roland H. C. Yap RYAP@COMP.NUS.EDU.SG
School of Computing, National University of Singapore, Singapore

Yuanlin Zhang YZHANG@CS.TTU.EDU
Department of Computer Science, Texas Tech University, USA

Editor: M.R.C. van Dongen

Abstract

Exploiting residual supports (or residues) has proved to be one of the most cost-effective approaches for Maintaining Arc Consistency during search (MAC). While MAC based on optimal AC algorithm may have better theoretical time complexity in some cases, in practice the overhead for maintaining required data structure during search outweighs the benefit, not to mention the more complicated implementation. Implementing MAC with residues, on the other hand, is trivial.

In this paper we extend previous work on residues and investigate the use of multiple residues during search. We first give a theoretical analysis of residue-based algorithms that explains their good practical performance. We then propose several heuristics on how to deal with multiple residues. Finally, our empirical study shows that with a proper and limited number of residues, many constraint checks can be saved. When the constraint check is expensive or a problem is hard, the multiple residues approach is competitive in both the number of constraint checks and cpu time.

Keywords: Arc Consistency, Residual Supports, MAC

1. Introduction

Maintaining Arc Consistency (MAC) (Sabin and Freuder, 1994) has been considered one of the most efficient algorithm for solving large and hard constraint satisfaction problems. At its core is the Arc Consistency algorithm (AC), whose efficiency plays a vital role in the overall performance of MAC.

Lecoutre and Hemery (2007) and Likitvivatanavong et al. (2007) show that when arc consistency is enforced during search (ACS), a MAC3-like algorithm that simply reuses supports found earlier, called *residual supports* or *residues*, can outperform the optimal algorithm MAC2001/3.1 (Bessière et al., 2005). Since a single residue is good, this paper investigates the use of multiple residues. We give a theoretical explanation for the effectiveness of residues. We then investigate various heuristics for using multiple residues. Finally, we perform extensive experiments with different variations of multiple residues. Our results show that when the number of residues is small (1 to 5), the number of constraint checks decreases sharply with a moderate increase of extra cost like validity checks.

©2008 Lecoutre *et al.*.

2. Preliminaries

A (finite) Constraint Network (CN) P is a pair $(\mathscr{X}, \mathscr{C})$ where $\mathscr{X}$ is a finite set of n variables and $\mathscr{C}$ a finite set of e constraints. Each variable $X \in \mathscr{X}$ has an associated domain containing the set of values allowed for X. The initial domain X is denoted by $dom^{init}(X)$; the current one by $dom(X)$. Each constraint $C \in \mathscr{C}$ involves an ordered subset of variables of $\mathscr{X}$ called scope (denoted by *scp*(C)), and an associated relation (denoted by $rel(C)$). For each r-ary constraint C with *scp*(C) = $\{X_1, \ldots, X_r\}$, $rel(C) \subseteq \prod_{i=1}^{r} dom^{init}(X_i)$. For any $t = (a_1, \ldots, a_r)$ of $rel(C)$, called a tuple, $t[X_i]$ denotes a_i. With a total order on the domains, tuples can be ordered using a lexicographic order $\prec$. To simplify the presentation, we use two special values $\bot$ and $\top$ where $\bot \prec t \prec \top$ for any tuple t.

Let C be an r-ary constraint and *scp*(C) = $\{X_1, \ldots, X_r\}$, an r-tuple t of $\prod_{i=1}^{r} dom^{init}(X_i)$ is said to be: (1) *allowed* by C iff $t \in rel(C)$, (2) *valid* iff $\forall X_i \in$ *scp*(C), $t[X_i] \in dom(X_i)$, (3) *a support* in C iff it is allowed by C and valid, and (4) *a conflict* iff it is not allowed by C and valid. A tuple t is a *support of* (X_i, a) in C when t is a support in C and $t[X_i] = a$. A *constraint check* determines if a tuple is allowed. A *validity check* determines if a tuple is valid. A solution to a constraint network is an assignment of values to all the variables such that all the constraints are satisfied.

A pair (X, a), with $X \in \mathscr{X}$ and $a \in dom(X)$, is *generalized arc-consistent (GAC)* iff $\forall C \in \mathscr{C}$ where $X \in scp(C)$, there exists a support of (X, a) in C. P is GAC iff $\forall X \in \mathscr{X}$, $dom(X) \neq \emptyset$ and $\forall a \in dom(X)$, (X, a) is GAC. For binary constraint networks GAC is called AC. A *CN-value* is a triplet (C, X, a) where $C \in \mathscr{C}$, $X \in scp(C)$, $a \in dom(X)$.

3. Residual Supports

We illustrate the concept of residual support with GAC3. A *residue* for a CN-value is a support that has been previously found and stored for future use. Unlike the *last* structure in AC2001/3.1, a residue for a value might not be a lower bound of the current supports of the value. The concept of residue has been introduced under its multi-directional form by Lecoutre et al. (2003) and under its uni-directional form by Likitvivatanavong et al. (2004).

3.1 GAC3 with Residual Supports

GAC3 with residual supports is shown in Algorithm 1. The algorithm adds a three-dimensional array *res*, which is initialized to $\bot$. For a CN-value (C,X,a), *res*[C, X, a] stores the residue for (X, a) with respect to C.

Algorithm 1: GAC3^{rm} (P = $(\mathscr{X}, \mathscr{C})$: Constraint Network)

```
for each C ∈ 𝒞 ∧ X ∈ scp(C) ∧ a ∈ dom(X) do res[C, X, a] ← ⊥
Q ← {(C, X) | C ∈ 𝒞 ∧ X ∈ scp(C)}
while Q ≠ ∅ do
    extract (C, X) from Q
    if revise(C, X) then
        Q ← Q ∪ {(C', X') | C' ∈ 𝒞 ∧ C' ≠ C ∧ X' ≠ X ∧ {X, X'} ⊆ scp(C')}
```

In each revision of arc (C, X) by *revise* (Algorithm 2), the validity of the residue for each CN-value (C, X, a) is tested first (line 3). If it fails, a new support is searched from scratch (line 4). If a support t is found, *multi-directionality* is exploited to update the residues of all values of t (line 8), since t is also a support of $t[Y]$ for all $Y \in scp(C)$. Consequently, $r - 1$ residues (r is the arity of C) are obtained for other values in the tuple with no effort. The *uni-directional* form, in contrast, only updates the residue of the CN-value (C, X, a).

Algorithm 2: revise(C: Constraint, X: Variable): Boolean

```
1 nbElements ← |dom(X)|
2 for each a ∈ dom(X) do
3     if isValid(C,res[C,X,a]) then continue
4     t ← seekSupport(C, X, a)
5     if t = ⊤ then
6         remove a from dom(X)
7     else
8         for each Y ∈ vars(C) do addResidue(C, Y, t)
9 return nbElements ≠ |dom(X)|
```

Algorithm 3: isValid(C: Constraint, t: Tuple): Boolean

```
1 if t = ⊥ then return false
2 for each X ∈ scp(C) do
3     if t[X] ∉ dom(X) then return false
4 return true
```

Algorithm 4: seekSupport(C: Constraint, X: Variable, a: Value): Tuple

```
1 t ← ⊥
2 while t ≠ ⊤ do
3     if t ∈ rel(C) then return t
4     t ← setNextValid(C, X, a, t)
5 return ⊤
```

Function *addResidue*(C: Constraint, X: Variable, t: Tuple) (not listed), assigns t to $res[C, X, t[X]]$. Function *isValid* (Algorithm 3) determines whether or not the given tuple is valid ($\perp$ is not valid). Function *seekSupport* (Algorithm 4) determines from scratch a support for (X, a) in C. It uses function *setNextValid* (not listed) which returns either the smallest valid tuple t' built from C such that $t \prec t'$ and $t'[X] = a$, or $\top$ if it does not exist. $\top$ does not belong to any relation.

3.2 Complexity Results

To understand why residues work, we present some results for binary problems. In particular, we study the complexity of AC3^{rm} when used stand-alone and when embedded in MAC. Without any loss of generality, we assume that each domain contains exactly d values.

Proposition 1 *AC3^{rm} has a worst-case space complexity of $O(ed)$ and a worst-case time complexity of $O(ed^3)$.*

Proof We assume here that each constraint is represented in an intentional form with space complexity $O(1)$, and $n < e$ (otherwise, each connected component of the network can be analyzed similarly). The space required for a CN is $O(e + nd)$, the space for Q is $O(e * 2) = O(e)$, and the space for *res* is $O(e * 2 * d) = O(ed)$. Thus, space complexity of AC3^{rm} is $O(ed)$.
Like the optimality proof of AC3.1 (Bessière et al., 2005), the number of validity checks performed by AC3^{rm} will not exceed the number of constraint checks performed by AC3. As AC3 requires $O(ed^3)$ in the worst case, so is AC3^{rm}. ■

The analysis can be refined to consider the tightness of the constraints ($\frac{|\text{allowed tuples}|}{|\text{possible tuples}|}$).

Definition 2 *A constraint C is* tightness-bounded *iff for any CN-value involving C, either its number of supports is $O(1)$ or its number of conflicts is $O(1)$ when $d \to \infty$.*

Many common constraints are tightness-bounded. For example, $X = Y$ or $X \neq Y$. For equations, each value is supported at most once. For dis-equations, each value allows at most one conflict. In practice, we observe that AC3^{rm} behaves in an optimal way when applied to constraints of small or high tightness.

Proposition 3 *Applied to a constraint network involving tightness-bounded constraints, AC3^{rm} admits a worst-case time complexity of $O(ed^2)$, which is optimal.*

Proof In (Lecoutre and Hemery, 2007), it is shown that the worst-case accumulated time complexity of *seekSupport* for a CN-value (C, X, a) is $O(cs + d)$ where c is the number of conflicts of (X, a) in C and s the number of supports of (X, a) in C. If C is tightness-bounded, then either $c = O(1)$ and $s = O(d)$, or $c = O(d)$ and $s = O(1)$ since $c + s = d$. It implies that the worst-case accumulated time complexity of *seekSupport* for a CN-value (C,X,a) is $O(d + d) = O(d)$. The overall complexity of AC3^{rm} is then $O(ed^2)$. ■

Proposition 3 shows that AC3^{rm} behaves optimally when constraints are tightness-bounded. This suggests AC3^{rm} should be quite competitive, compared to optimal algorithms like AC2001/3.1, on highly structured problems. These results are confirmed when the state-of-the-art generic algorithm MAC (Sabin and Freuder, 1994) is considered. The following result is directly obtained from previous propositions and the fact that AC3 is an incremental algorithm so no maintenance of data structures is necessary when backtracking. MAC3^{rm} denotes MAC embedding AC3^{rm}.

Proposition 4 *MAC3^{rm} admits a worst-case space complexity of $O(ed)$, and for any branch of the search tree admits: (1) a worst-case time complexity of $O(ed^3)$, and (2) a worst-case time complexity of $O(ed^2)$ for a constraint network involving tightness-bounded constraints.*

These theoretical results partially justify the data in (Likitvivatanavong et al., 2004; Lecoutre and Hemery, 2007). Further, they hold for single as well as (a bounded number of) multiple residues. We will focus on the practical aspects of multiple residues from now.

4. Fundamentals of Multiple Residues

Given the results of the single residue approach (Lecoutre and Hemery, 2007; Likitvivatanavong et al., 2004), it is interesting to see if we can achieve better performance by using *multiple residues*. Using more than one residue allows us to record not only the latest support, but also a selected

subset of the past supports. The memory space for holding these residues for each CN-value is denoted *residue store*.

To find a support, an AC or ACS algorithm with multiple residues always considers the residue store first. If all residues fail the validity test, the algorithm looks for a new support which may then be added to the store. When the store is full, ideally the residue that has the least chance of being a support in the future is deleted. We approximate the idealized case using a number of heuristics. They can be used to evaluate each residue and return a value called *utility* that approximates the idealized probability. The utility also guides in identifying which residues to check first when looking for a support.

We present a generic algorithm and several policies that define the utility of a residue.

4.1 A Generic Multiple Residue Algorithm

The generic multiple residue algorithm is similar to the single one. The difference lies in how the residue store is updated and how the validity of residues is checked. Function *addResidue* is redefined in Algorithm 5 and *isValid* (used in Algorithm 2) is replaced by *existValid* (Algorithm 6). There are three algorithm variants differing in management of utilities. The *static* variant computes a utility score only when a new support is added to the residue store. The *dynamic* and *fully dynamic* variants update the utility of the first found valid residue (line 4 of Algorithm 6). For the fully dynamic variant, the score of a residue is updated whenever its validity is checked (line 6 of Algorithm 6). The different variants are differentiated by the global variable *updateType*.

Algorithm 5: addResidue(C: Constraint, X: Variable, t: Tuple) `(generic routine)`

```
1 u ← f(t)
2 if size(res[C, X, t[X]]) = maxR then
3     (t0, u0) ← min(res[C, X, t[X]])
4     if u0 < u then delete(res[C, X, t[X]], (t0, u0)))
5 if size(res[C, X, t[X]]) < maxR then insert(res[C, X, t[X]], (t, u))
```

Algorithm 6: existValid(C: Constraint, D: Dictionary): Boolean `(generic routine)`

```
1 (t, u) ← max(D)
2 while t ≠ ⊥ do
3     if isValid(C, t) then
4         if updateType ∈ {dynamic, fullyDynamic} then update(D, (t, f(t)))
5         return true
6     if updateType = fullyDynamic then update(D, (t, f(t)))
7     (t, u) ← pred(D, (t, u))
8 return false
```

The following data structures implement the ideas. Residue stores have fixed size $maxR$. The utility function is denoted by f. *res*$[C, X, a]$ is now a Dictionary S with the following operations. *max*(S) returns a pair (t, u) where the residue t has the highest utility score u; *pred*($S, (t, u)$) returns a pair (t', u') where t' is the residue of utility u' which is the next smaller score than u, and returns $\perp$ when there is none; *insert*($S, (t, u)$)/*delete*($S, (t, u)$) inserts/deletes the pair (t, u); *update*($S, (t, u')$) updates the tuple (t, u) in the store by replacing u with u' and updates the tuple's position in S.

size(S) returns the number of residues in S. In practice, a Dictionary can be implemented using a variety of data structures, for example balanced trees. We remark that the new support might not automatically become a residue when the residue store is full. For a new support to become a residue, its utility score must improve on the minimum score of the store (line 4 of Algorithm 5).

4.2 Policies Defining the Utility Function

We consider the following policies which are based on evaluating the probability (utility) of a support being valid in the future:

1. *Level of the search tree*. The rationale is that the support found at a deeper level should be more robust to change in the network caused by a new assignment or backtracking. When the search backtracks, previously removed values are restored, but the *current* support is still valid until the search tree branches off to a different path. LEVELMIN policy defines the utility of a new support or a residue in the store as the current level of the search tree when it was found or checked (for validity).

2. *Domain size*. The rationale is that the support found when the domain is smaller is more robust. DOMMIN policy defines the utility of a support (or residue) based on the size of the relevant domains when it was found (or its validity is checked). Specifically, if *revise*(C, X) is invoked and the tuple t is found as a new support of $a \in X$, then $f(t) = \sum_{Y \in scp(C)-X} |dom(Y)|$.

3. *Chronology*. The intuition is that the latest support should remain valid in the near future, where the network has not changed much. FIFO ("First In First Out") defines the utility of a value by the time stamp when it was found or its validity was checked.

4. *Frequency*. The expectation here is that the residue that has passed validity check frequently in the past will continue to do so in the future. FRQCYMIN initially sets the utility of a new support to one, then adds one to the score for every successful validity check for a residue and subtracts one for every unsuccessful check.

LEVELMIN, DOMMAX, FIFO and FRQCYMIN are called heuristic policies because they are based on the rationale for the corresponding factor. We also have the corresponding anti-heuristics which use the opposing rationale and whose anti-heuristic score is simply the negative of the heuristic score: LEVELMAX, DOMMAX, LIFO and FRQCYMAX.

In addition, we also consider a random replacement policy RANDOM as a baseline for comparison purpose in experiments.

5. Implementations of Different Policies

Intuitively, it makes sense that the residue store shouldn't be large — this is also borne out by our empirical study. Thus, we use an array for the residue store to speed up access. Other data structures may be more efficient with a larger residue store but have more overhead. Arrays also have better cache behavior (Mitchell (2005)).

We use the following convention. Array indices range from 0 to $maxR - 1$. Given an array D, $D.size$ (initialized to 0) is the current number of residues. In the routine *addResidue*, D represents $res[C, X, t[X]]$.

5.1 FIFO Policy

We use circular arrays to hold residues for static and fully dynamic FIFO. There is no need to record the utility score explicitly. Given an array D, $D.head$ gives the index of the residue with the highest score; it is initially set to zero.

To find a support, both static and fully dynamic variants simply search from $D.head$ until the end of the circular array (Algorithm 8). When a residue r is found to be valid in the circular array, fully dynamic FIFO sets $D.head$ to point to r, since it has now become the most recent support. Moreover, the invalid residues before r are automatically placed at the end of the circular array as a side-effect. To add a value to the store, we simply store it before (in a modular way) $D.head$ (Algorithm 7).

Algorithm 7: addResidue(C, X, t) (FIFO)

```
1 if D.size < maxR then D.size ← D.size + 1
2 if updateType = dynamic then
3     for p ← D.size − 1 down to 1 do D[p] ← D[p − 1]
4     D[0] ← t
5 else
6     D.head ← (D.head − 1 + maxR) mod maxR
7     D[D.head] ← t
```

The residue store for dynamic FIFO also uses arrays. To find a support, we search array from beginning to the end. Since the utilities of the invalid residues before the first valid residue are not updated, when the first valid residue is found, we simply move it to the front and shift the invalid residues one position to the right (Algorithm 9). To add a new residue, we shift the first $maxR - 1$ residues one position to the right and put the new one at the front (Algorithm 7). Array copying can be avoided but other alternative implementations may have higher overheads for the residue store sizes here.

Algorithm 8: existValid(C, D) (static and fully dynamic FIFO)

```
1 for i ← 0 to D.size − 1 do
2     p ← (D.head + i) mod D.size
3     if isValid(C, D[p]) then
4         if updateType = fullyDynamic then D.head ← p
5         return true
6 return false
```

Algorithm 9: existValid(C, D) (dynamic FIFO)

```
1 for p ← 0 to D.size − 1 do
2     if isValid(C, D[p]) then
3         temp ← D[p]
4         for i ← p down to 1 do D[i] ← D[i − 1]
5         D[0] ← temp
6         return true
7 return false
```

5.2 LEVEL/DOM Policy

Like dynamic FIFO, we use standard arrays to hold residues. However, for LEVEL and DOM policies, given array D and index i, $D[i]$ refers to the tuple (t, u) stored at the ith position, where t ($D[i].residue$) is the residue and u ($D[i].score$) is the utility. The residues are arranged in a descending order according to the utility, with ties broken by giving priority to the most recent support. *addResidue* and *existValid* are listed in Algorithm 10–11. LEVELMIN/MAX and DOMMIN/MAX are achieved by setting f as shown in Section 4.2.

Here, only static and dynamic variants are described. The fully dynamic variant involves updating the utility of invalid residues checked so far as well as the utility of the first valid residue found. All these residues need to be re-ordered afterward. The cost of sorting these residues every time *existValid* is called is too expensive for our purpose. In contrast, the dynamic variant needs only to shift a single element in the array to its appropriate place to maintain the ordering of residues (lines 3–13 of Algorithm 11).

Algorithm 10: addResidue(C, X, t) `(static and dynamic LEVEL/DOM)`

```
1  if D.size < maxR then
2      D.size ← D.size + 1
3  else if D[D.size − 1].score ≥ f(t) then return
4  p ← D.size − 2
5  while p ≥ 0 ∧ D[p].score ≤ f(t) do
6      D[p + 1] ← D[p]
7      p ← p − 1
8  D[p + 1] ← (t, f(t))
```

Algorithm 11: existValid(C, D) `(static and dynamic LEVEL/DOM)`

```
1  for p ← 0 to D.size − 1 do
2      if isValid(C, D[p].residue) then
3          if updateType= dynamic and f(t) ≠ D[p].score then
4              if f(t) > D[p].score and p > 0 then
5                  while p > 0 and D[p − 1].score ≤ f(t) do
6                      D[p] ← D[p − 1]
7                      p ← p − 1
8                  D[p] ← (t, f(t))
9              if f(t) < D[p].score and p < D.size − 1 then
10                 while p < D.size − 1 and D[p + 1].score > f(t) do
11                     D[p] ← D[p + 1]
12                     p ← p + 1
13                 D[p] ← (t, f(t))
14         return true
15 return false
```

Algorithm 12: addResidue(C, X, t) `(dynamic and fully dynamic FRQCYMIN)`

```
1 if D.size < maxR then
2     D.size ← D.size + 1
3 else if updateType = fullyDynamic ∧ D[D.size − 1].score > 1 then return
4 D[D.size − 1] ← (t, 1)
```

Algorithm 13: existValid(C, D) `(dynamic and fully dynamic FRQCYMIN)`

```
1 for p ← 0 to D.size − 1 do
2     if isValid(C, D[p].residue) then
3         D[p].score ← D[p].score + 1
4         re-order(D)
5         return true
6     else if updateType = fullyDynamic then D[p].score ← D[p].score − 1
7 return false
```

5.3 FRQCY Policy

We use arrays to hold residues in a descending order by their utility. It doesn't make sense to consider the static version since residues in the store would have a constant score of 1.

The function *addResidue* is shown in Algorithm 12. As the utility score for dynamic FRQCYMIN can only increase, it may be possible that all residues have a score greater than 1. Requiring the new support score to improve on the lowest score for fully dynamic FRQCYMIN would mean that the new support would never be recorded as its initial score is one. Rather, we replace the residue with the lowest score regardless of its value.

The function *existValid* is given in Algorithm 13. The principle is the following: when the residue at position p is found to be valid, its score is incremented, while, for fully dynamic FRQCYMIN, its score is decremented. When a residue is found to be valid, we have to re-order the array. It is performed by calling the routine *re-order*(D) (not described). Dynamic and fully dynamic FRQCYMAX are similar and have not been listed.

6. Experimental Results

We studied the performance of multiple residues with various store sizes and policies. The benchmarks are problems from the 2006 CSP solver competition[1].

6.1 Binary Problems

For binary benchmarks, we used a solver written in C++ that implements MAC embedding AC3^r (i.e., AC3 with uni-directional residues). It includes dom/deg variable ordering and the lexicographical value ordering. The results were obtained on a DELL PowerEdge 1850 (two 3.6GHz Intel Xeon CPUs) in Linux.

We first considered some classes of binary random instances situated at the phase transition for satisfiability. These are $(40, 8, 753, 100)$, $(40, 11, 414, 200)$, $(40, 16, 250, 350)$, $(40, 25, 180, 500)$, $(40, 40, 135, 650)$, $(40, 80, 103, 800)$, $(40, 180, 84, 900)$ (called R1–R7). A class is represented by (n, d, e, t)

1. See `http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/benchmarks.html`

with $t/1000$ denoting the probability of a tuple not allowed by a constraint. We used 21 instances for each class. The cpu time is the time (in seconds) it takes to solve 21 of them.

We also considered the following academic and real world problems: ehi-85-297-12, ehi-85-297-13, frb40-19-3, frb35-17-5, pigeons-10, pigeons-11, qa-5, qa-6, qk-20-0-5, qk-25-0-5, fapp01-0200-8, fapp01-0200-9, graph-10, graph-14, scen-11, scen-05 (called P1–P16).

For each problem, we collected the performance data using algorithms with different policies and residue store size from 1 to 10. Because it is obvious that an algorithm using larger number of residues requires fewer number of constraint checks, *we made the cost of each constraint check as small as possible in order to test the performance in the worst case*. Due to limited space, we can only present selected data, but *all the analysis and observations apply to all the data unless mentioned otherwise*.

6.1.1 HEURISTICS VERSUS ANTI-HEURISTICS

The experimental results on the effectiveness of heuristics and anti-heuristics are shown in Figure 1. The x-axis is an ordered sequence of pairs (*problem*, *residueNumber*) where *problem* is R1–R7 or P1–P16 and *residueNumber* is 1–10. We expect that the heuristics will always be better than the anti-heuristics. The results show that DOMMAX and LEVELMIN indeed performed better than DOMMIN and LEVELMAX on any data-point. The hypothesis is clearly supported by empirical evidence: the residue found when the domain size or the search level is smaller is more robust.

The result for FRQCY is surprising, however. This can be attributed to the fact that FRQCYMAX behaves in similar way to FIFO, which has been shown very effective and robust in our results. FRQCYMIN on the contrary favors new support less. This result indicates that the residues used most frequently might be less relevant in the future.

6.1.2 QUANTITATIVE EFFECT OF RESIDUE NUMBER ON PERFORMANCE

Having established that anti-heuristics are always worse, we will only consider the heuristics in the subsequent reports[2]. We study in this section how the number of residues affects the broad performance of algorithms. Our data shows some clear patterns. First, the number of constraint checks decreases sharply and quickly converges to a stable number as the residue number increases. Second, the extra cost reflected mainly by validity checks increases at most linearly. As an illustration, we give performance of FIFO on all problems with varying residue number in Figure 2. The x-axis is the same as in the previous subsection. The two observations above imply that for big residue number, the new algorithms will not pay off. However, for small numbers (say 1 to 5), the extra cost can be compensated by the savings over constraint checks. The cpu time in Figure 2 reflects this interaction. The cpu time saving is not obvious due to the cheap cost of constraint checks. The third observation is that as the problems become more difficult, i.e., more constraint checks are needed, the savings (both in terms of constraint checks and cpu time) is more significant, as shown in the figure. The last observation is that the number of operations needed to maintain the proper ordering of the residue store is much lower than the number of the validity checks.

2. The exception is FRQCY, for which the anti-heuristic (FRQCYMAX) is better.

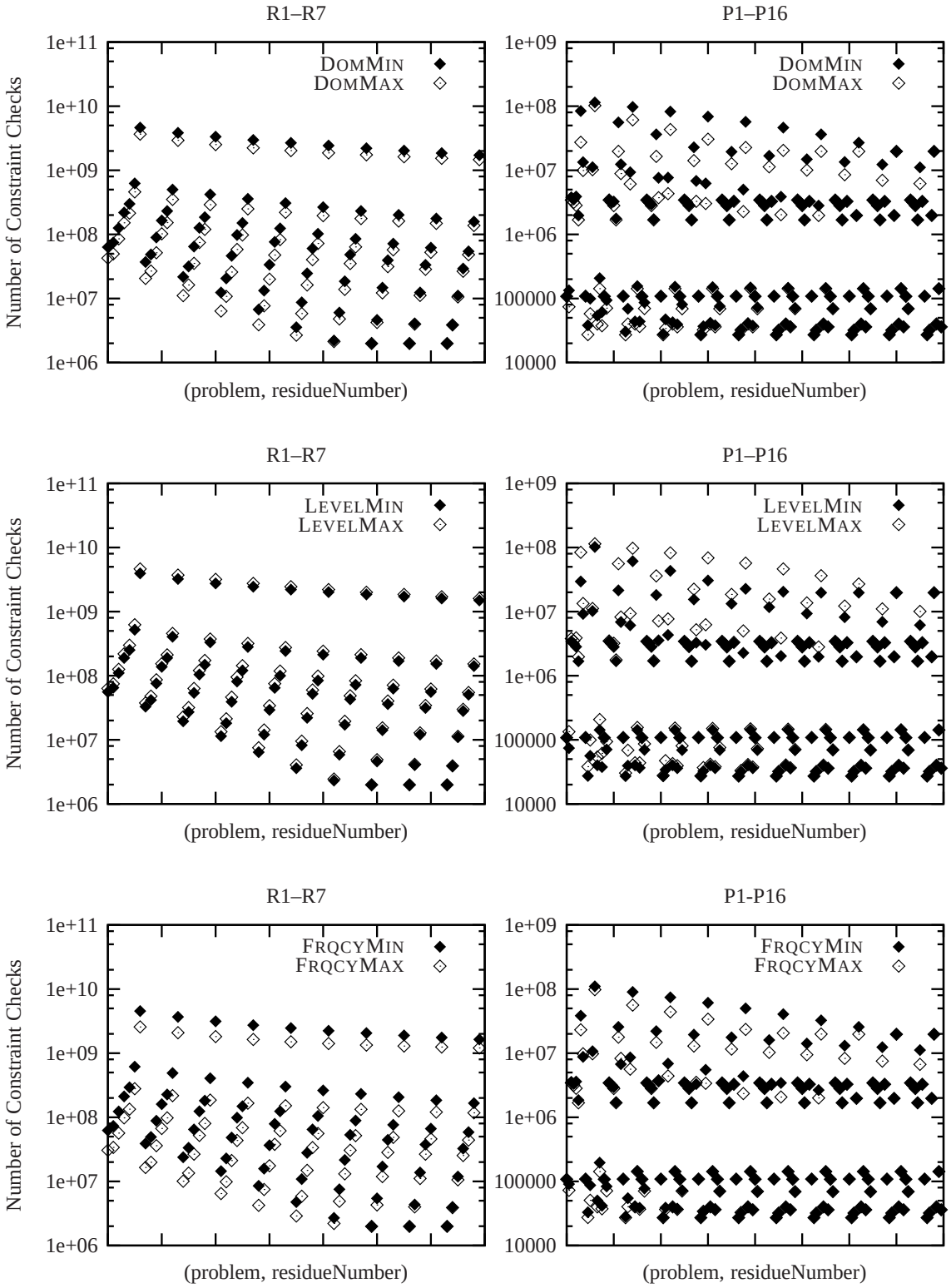

Figure 1: Comparisons of heuristics and anti-heuristics.

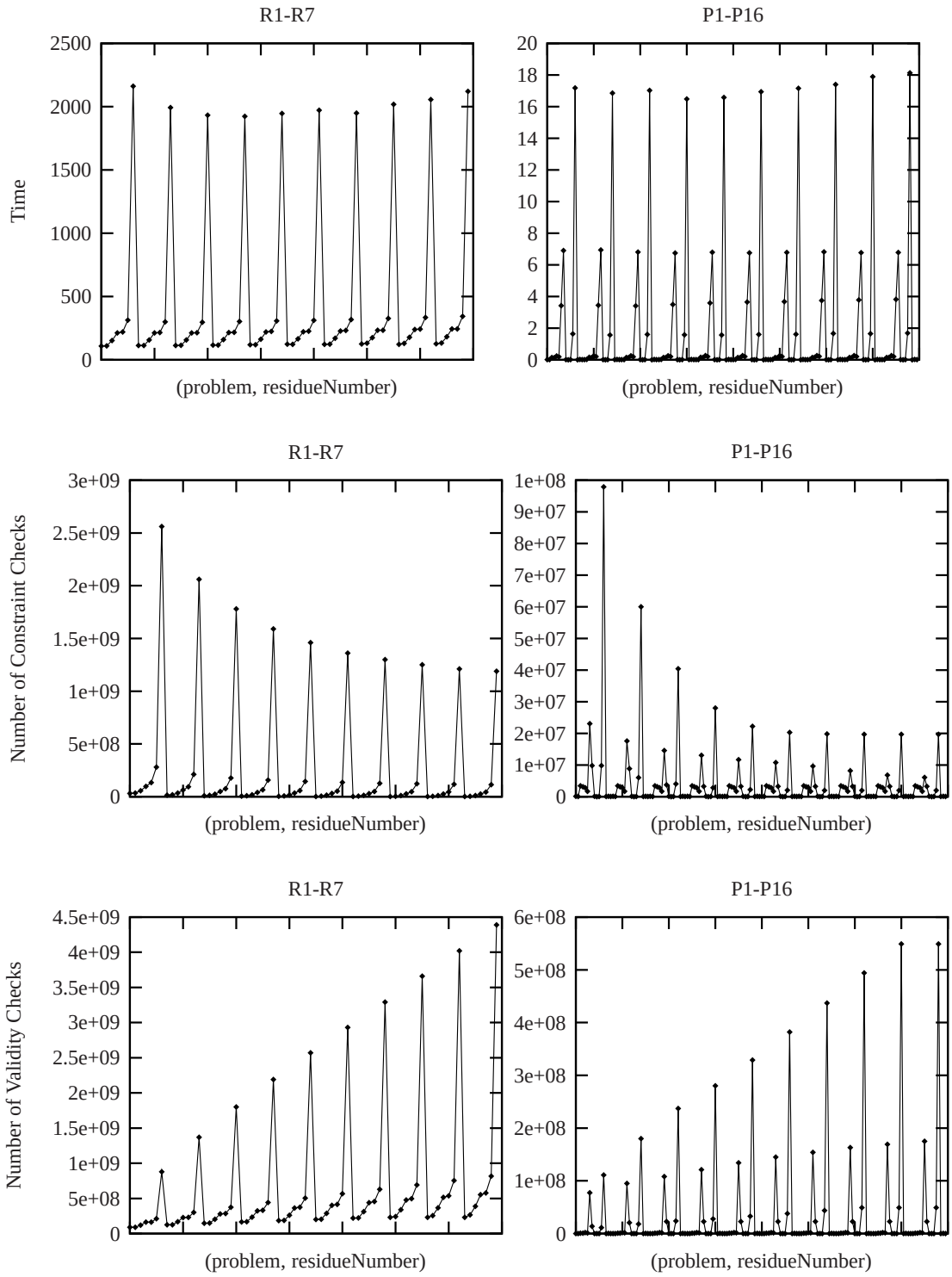

Figure 2: Trends for cpu time and the number of constraint and validity checks against the number of residues for FIFO.

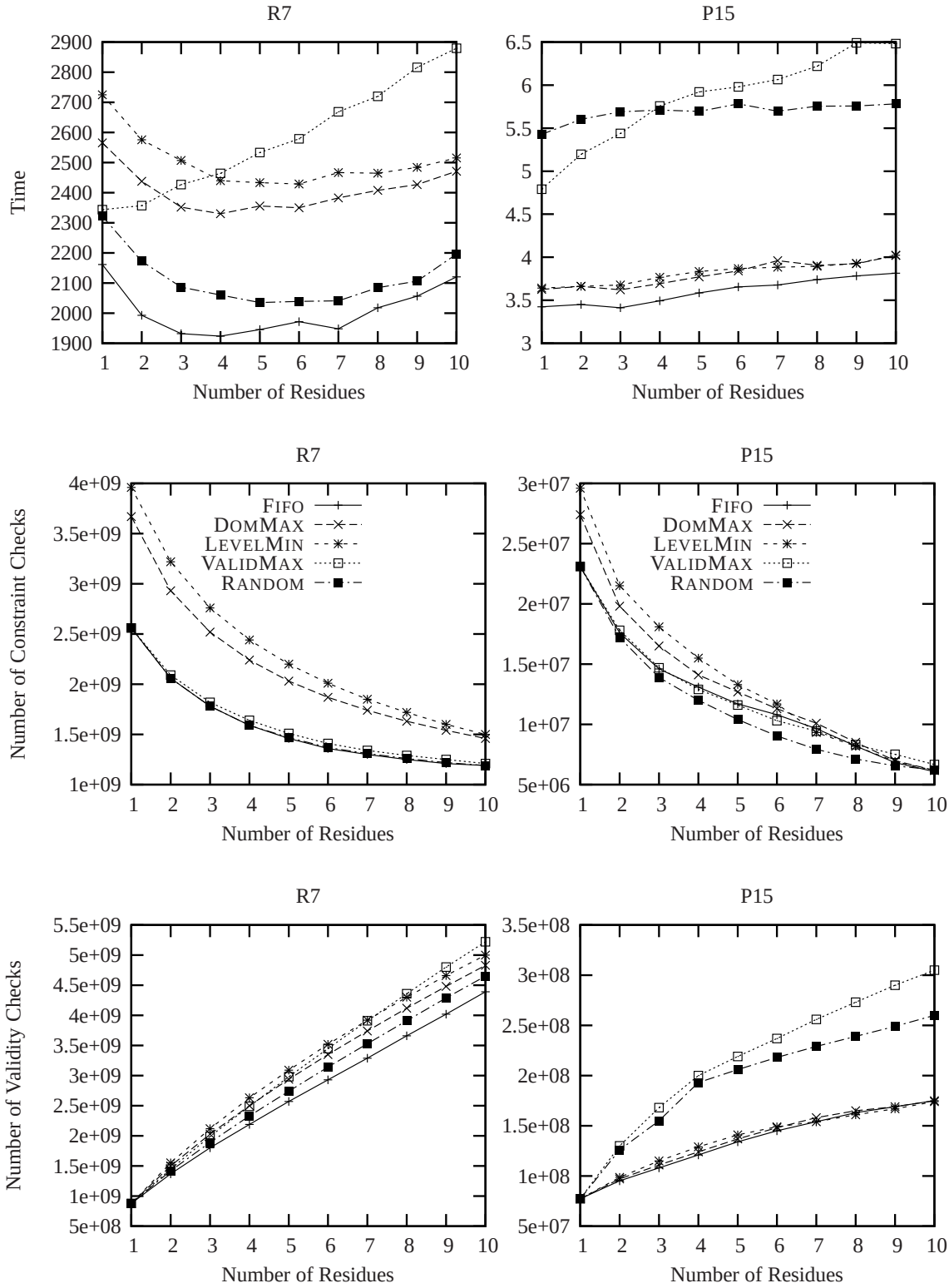

Figure 3: Comparisons of heuristic policies.

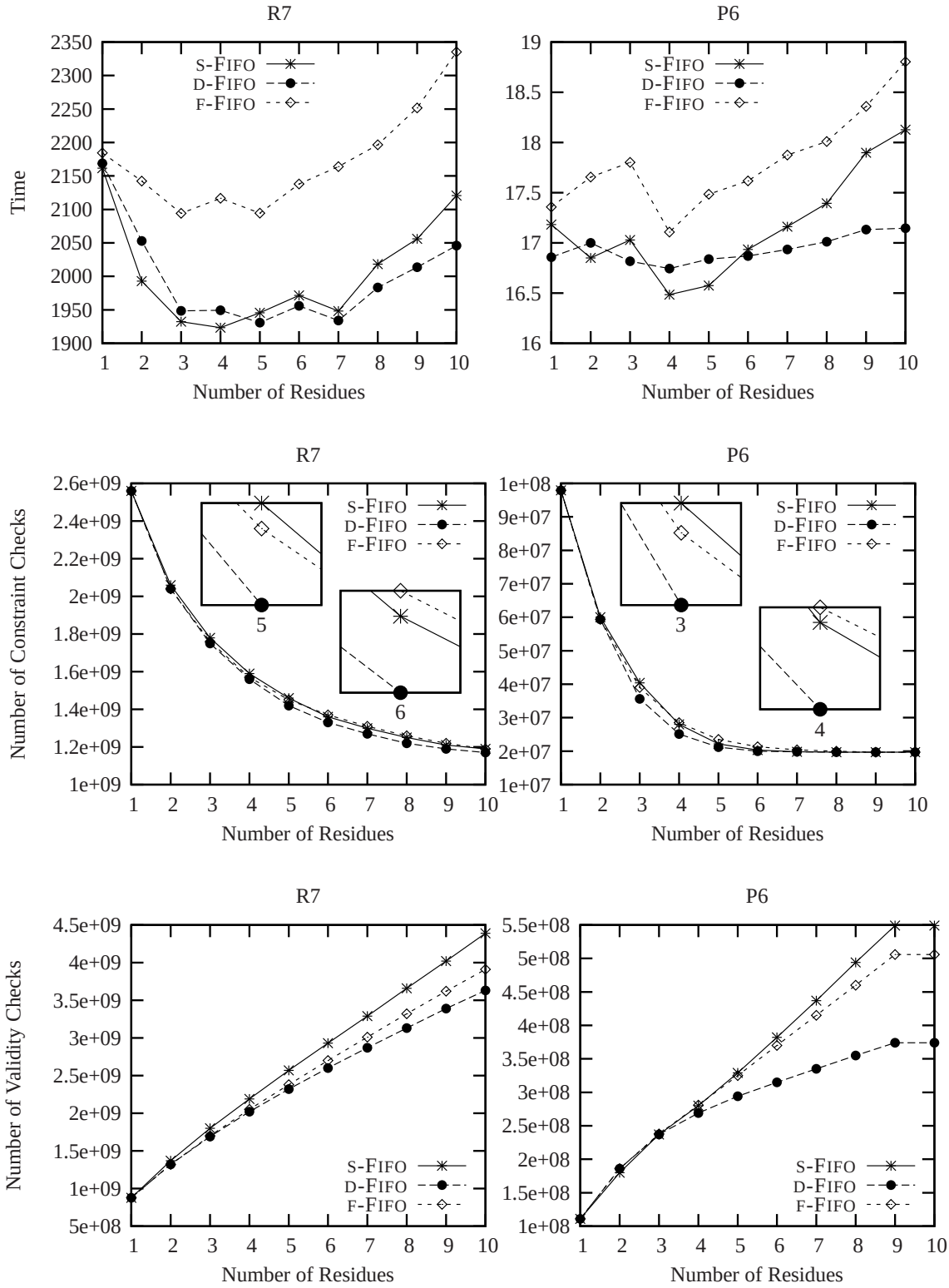

Figure 4: Comparisons of different updating strategies for FIFO.

6.1.3 COMPARISON OF VARIOUS POLICIES AND UPDATING STRATEGIES

We compare performance of the static approach for FIFO, DOMMAX, LEVELMIN and FRQCYMIN on each problem class. Selected results on R7 and P15 are shown in Figure 3. Results on other problem classes show similar trends. For comparison purpose, we also tested a random replacement policy (RANDOM), described as follows. The residue store of RANDOM is an array of length $maxR$. The kth new residue found is inserted at the kth position in the array, where $1 \leq k \leq maxR$. When the residue store is full ($k > maxR$), we pick an index within the range of the array randomly and overwrite the old residue with a new one. Searching for a support does not need to be undeterministic since the residue replacement already provides randomness for the policy. For simplicity, the search always starts from the lowest to the highest index in the array.

From the graphs, RANDOM is among the best policy for R7 while it is among the wost for P15. We notice that the relative performance of RANDOM also varies across different problem classes. Thus, the effect of RANDOM is dependent on many factors, which is not surprising. By contrast, the performance of other policies forms a clear and uniform ordering. Since its performance is unpredictable, we will no longer consider RANDOM in later analysis.

FIFO has the best results, followed by DOMMAX, LEVELMIN, and FRQCYMAX. DOMMAX beats LEVELMIN possibly because it is finer-grained: LEVELMIN assigns the same utility to residues found at the same level regardless of individual differences in their respective domain size. While FRQCYMAX is closer to FIFO in the number of constraint checks, the cpu time is out of proportion with other policies. This is due partly to the cost of residue store ordering when the residue number gets larger, and partly to the large number of validity checks. From these graphs, we see that the ability of FRQCYMAX to retain supports is very close to that of FIFO, but the much larger number of validity checks implies that these supports are positioned at the very end of the array.

Next, we compare the performance of different updating strategies for FIFO in Figure 4. In cpu time, static FIFO is faster than dynamic FIFO when residue numbers are small, but becomes gradually slower as the number of residues increases. Nonetheless, the best results are obtained with static FIFO using low number of residues. The graphs for cpu time show that the best residue number lies in the region where the saving in constraint checks has just begun to be outweighed by the increase in the number of validity checks.

We do not give details for other policies but in general the difference between static and dynamic policy is small. For harder problems like P6 and P15, dynamic approach requires fewer number of constraint checks for DOMMAX and LEVELMIN while the converse is true for the anti-heuristics. However, as for cpu time, the dynamic approach is slower due to cheap constraint checks and higher cost in maintaining residue store dynamically.

6.2 Non-binary Problems

For non-binary problems, we used the Abscon solver. It implements MGAC embedding GAC3^{rm} with $dom/wdeg$ variable ordering and lexicographical value ordering. Experiments were done in Linux on a cluster of 93 nodes, each with two Intel Xeon 3GHz and 2GB RAM. The results we present are for some classical hard non-binary instances from the *Dimacs aim*, *Chessboard Coloration*, *Dubois*, *Schurr's Lemma*, *Dimacs Pret*, *Golomb's Ruler*, *All Interval Series*, and *Traveling Salesman Problem*.

$Instances$		$MGAC2001$	$MGAC3$	$MGAC3^{rm}$	$MGAC3^{rm2}$	$MGAC3^{rm3}$
aim-200-3-4-2	cpu	14.23	9.71	10.04	9.59	9.99
	$ccks$	$2146K$	$2196K$	$1033K$	$881K$	$880K$
aim-200-3-4-3	cpu	402.82	249.07	267.14	242.97	277.47
	$ccks$	$107M$	$110M$	$53M$	$45M$	$45M$
cc-20-20-2	cpu	16.42	11.25	12.28	13.61	14.36
	$ccks$	$129K$	$157K$	82059	53839	50539
cc-25-25-2	cpu	45.24	24.23	26.0	28.55	34.48
	$ccks$	$253K$	$305K$	$152K$	92975	87453
dubois-23-ext	cpu	646.4	634.04	559.37	550.92	592.25
	$ccks$	$222M$	$576M$	$202M$	$148M$	$148M$
dubois-24-ext	cpu	1272.38	1243.97	1147.67	1103.96	1125.77
	$ccks$	$429M$	$1122M$	$393M$	$290M$	$290M$
lemma-15-9-mod	cpu	33.03	32.759	27.58	22.63	25.17
	$ccks$	$47M$	$66M$	$33M$	$25M$	$24M$
lemma-20-9-mod	cpu	62.93	65.94	44.15	43.36	44.71
	$ccks$	$105M$	$142M$	$68M$	$52M$	$49M$
pret-60-60-ext	cpu	80.22	76.92	83.88	76.83	73.41
	$ccks$	$30M$	$78M$	$30M$	$21M$	$21M$
pret-60-75-ext	cpu	97.2	80.36	83.77	82.14	80.21
	$ccks$	$31M$	$82M$	$31M$	$22M$	$22M$
ruler-44-9-a3	cpu	12.83	22.86	13.47	12.75	15.2
	$ccks$	$33M$	$80M$	$36M$	$30M$	$29M$
ruler-44-10-a3	cpu	34.34	60.12	32.18	30.97	34.64
	$ccks$	$95M$	$242M$	$97M$	$80M$	$77M$
series-14	cpu	140.48	168.76	125.93	112.51	132.07
	$ccks$	$298M$	$508M$	$248M$	$200M$	$188M$
series-15	cpu	713.22	974.25	646.56	608.37	694.07
	$ccks$	$1624M$	$2774M$	$1351M$	$1091M$	$1024M$
tsp-25-681-ext	cpu	65.459	93.41	68.51	56.64	64.08
	$ccks$	$107M$	$219M$	$85M$	$72M$	$69M$
tsp-25-715-ext	cpu	113.69	167.75	95.18	89.42	97.01
	$ccks$	$195M$	$458M$	$156M$	$135M$	$131M$

Here, we have focused our attention to fully dynamic FIFO policy.[3] We only considered a limited number of residues ($GAC3^{rmk}$ is $GAC3^{rm}$ with k residues associated with each CN-value) as it appears to be the right approach. The results in the table further verify our observations on the impact of residue number over performance. Specifically, the number of constraint checks drops sharply and converges quickly. We observe that MGAC3^{rm2} is a good compromise between saving constraint checks and improving cpu time.

7. Conclusion

We have generalized the existing work on single residue to multiple residues. We have a thorough investigation of the multiple residue approach including complexity analysis, the policies to manage the residue store, and an extensive empirical study of the effectiveness of the policies and the impact of store size on the performance. For heuristic policies, a key observation is that the number of constraint checks decreases quickly and converges to a stable number as the number of residues increases — indeed, in our experiments the harder the problem is, the larger the saving in the number of constraint checks. However, the extra cost reflected by validity checks also increases steadily as the number of residues increases. This suggests that the optimum number of residues

3. Unlike in the previous section, fully dynamic FIFO is better than both dynamic and static FIFO on these non-binary problems. We do not consider other policies for space reasons, as it is clear that FIFO is the best policy.

should be small (say 1 to 5) since the total cost would be dominated by the cost of validity checks as more and more residues are in use.

References

Christian Bessière, Jean-Charles Régin, Roland H. C. Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

Christophe Lecoutre and Fred Hemery. A study of residual supports in arc consistency. In *Proceedings of IJCAI'07*, pages 125–130, 2007.

Christophe Lecoutre, Frederic Boussemart, and Fred Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *Proceedings of CP'03*, pages 480–494, 2003.

Chavalit Likitvivatanavong, Yuanlin Zhang, James Bowen, and Eugene C. Freuder. Arc consistency in MAC: a new perspective. In *Proceedings of CP'04 Workshop on Constraint Propagation And Implementation*, pages 93–107, 2004.

Chavalit Likitvivatanavong, Yuanlin Zhang, Scott Shannon, James Bowen, and Eugene C. Freuder. Arc consistency during seach. In *Proceedings of IJCAI'07*, pages 137–142, 2007.

David G. Mitchell. A sat solver primer. *EATCS Bulletin (The Logic in Computer Science Column)*, pages 112–133, 2005.

Daniel Sabin and Eugene C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of CP'94*, pages 10–20, 1994.