Constraint Programming Letters 1 (2006) 1-48 Submitted 6/2006; Published 7/2006

Maintaining Arc Consistency with Multiple Residues

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

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

Scott 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

Yuanlin Zhang Y.ZHANG@TTU.EDU
Department of Computer Science, Texas Tech University, USA

Editor:

Abstract

Exploiting residual supports (or residues) has proved to one of the most cost-effective approach 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 works on residue and investigate the use of multiple residues during search. We study various factors that could affect performance, such as the number of residues, their arrangement during search, and replacement policies for ineffective residues. Experimental results indicate that the best performance is archived by using just a few residues and the most basic replacement policy.

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 [cite]. At its core is the Arc Consistency algorithm (AC), whose efficiency plays a vital role in the overall performance of MAC.

The development AC can be dated back several decades to the original AC algorithm, AC3 (Mackworth (1977a)). AC3 has the time complexity of $O(ed^3)$, where e is the number of constraints in the problem instance and d the maximum domain size. Although optimal AC algorithms — whose time complexity are $O(ed^2)$ — have been reported recently, AC3 is still used widely in MAC due to the simplicity of implementation and its competitive performance in practice. Indeed, data structure used by optimal AC algorithms must be managed carefully in backtracking environment, and this has a non-negligible cost.

We consider MAC with binary-branching rather than d-way branching as it was shown theoretically superior (Mitchell (2003)). In the case of binary branching, it should be noted that although MAC based on an optimal AC algorithm admits an optimal worst-case time complexity on any branch involving only positive decisions (i.e variable assignments), this is not true in presence of negative decisions.

Optimal AC algorithm (Bessière et al. (2005)) employs the concept of *last support*, a lower-bound indicating that no smaller value in an can be a support. During constraint propagation, AC2001/3.1 is then able to look for a support starting from this bound rather than from the beginning of the domain. In the context of MAC however, the data structure associated with the last-support needs to be maintained. That is, during search its value before and after backtrack must be identical. MAC2001/3.1 thus requires additional data structure in order to keep track of all the previous supports, therefore incurring overhead. It is possible, however, not to deal with this extra complexity by resetting the last-support at each node in the search tree. But this means information from the invocation of AC of the previous node cannot be passed on to the next node.

Nonetheless, it may not be immediately obvious that a third option is possible: the last data structure can be used simply as a cache. While this reduces the algorithm to AC3, the residual data can be carried from one node to the next with ease and needs no maintainenance after backtrack. This residual information (or residue) has been shown one of the most important factors in the performance of MAC (Lecoutre and Hemery (2007); Likitvivatanavong et al. (2007)), so much so that MAC using residue alone has better CPU time than MAC2001/3.1, whose overhead outweighing its advantage of being AC-optimal on positive branches. Coupled with multi-directionality, the use of MAC with residue even outperforms MAC2001/3.1 in term of the number of constraint checks.

In this paper we study the use of multiple residues in MAC. The rational is clear: more residues means there are more candidates for a support, and so a higher chance of cache hit. However, we are faced with several issues not relevant to a single residue: for instance how to add, remove, and rearrange residues during search. We study these factors and compare the results on both randomly generated problems and benchmark problems. It turns out that in many cases the most effective strategy is to use just one or two residues while rearranging and replacing residue in queue-like order.

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, denoted $dom(X)$, which contains the set of values allowed for X. Each constraint $C \in \mathscr{C}$ involves an ordered subset of variables of $\mathscr{X}$, called scope and denoted $scp(C)$, and has an associated relation, denoted $rel(C)$, which contains the set of tuples allowed for the variables of its scope. The initial (resp. current) domain of a variable X is denoted $dom^{init}(X)$ (resp. $dom(X)$). of X is denoted dom(X). For each r-ary constraint C such that $scp(C) = \{X_1, \ldots, X_r\}$, we have: $rel(C) \subseteq \prod_{i=1}^{r} dom^{init}(X_i)$ where $\prod$ denotes the Cartesian product. Also, for any element $t = (a_1, \ldots, a_r)$, called tuple, of $\prod_{i=1}^{r} dom^{init}(X_i)$, $t[X_i]$ denotes the value a_i. It is also important to note that, assuming a total order on domains, tuples can

be ordered using a lexicographic order $\prec$. To simplify the presentation of some algorithms, we will use two special values $\perp$ and $\top$ such that any tuple t is such that $\perp \prec t \prec \top$.

Definition 1 *Let C be a r-ary constraint such that $scp(C) = \{X_1, \ldots, X_r\}$, an r-tuple t of $\prod_{i=1}^{r} dom^{init}(X_i)$ is said to be:*

- *allowed by C iff $t \in rel(C)$,*
- *valid iff $\forall X_i \in scp(C)$, $t[X_i] \in dom(X_i)$,*
- *a support in C iff it is allowed by C and valid.*

A tuple t will be said to be a support of (X_i, a) in C when t is a support in C such that $t[X_i] = a$. A tuple that is not a support will be called a conflict. Determining if a tuple is allowed is called a constraint check and determining if a tuple is valid is called a validity check. A solution to a constraint network is an assignment of values to all the variables such that all the constraints are satisfied. A constraint network is said to be satisfiable iff it admits at least one solution. The Constraint Satisfaction Problem (CSP) is the NP-complete task of determining whether a given constraint network is satisfiable. A CSP instance is then defined by a constraint network, and solving it involves either finding one (or more) solution or reporting the problem non-soluble.

To solve a CSP instance, one can apply inference or search methods (Dechter (2003)). Usually, domains s are reduced by removing *inconsistent* values — values that cannot occur in any solution. Generalized Arc Consistency (GAC) is a principal property of consistent constraint networks and establishing GAC on a given network P involves removing all values that are not generalized arc-consistent. GAC on binary constraint networks is called AC.

Definition 2 *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN. A pair (X, a), with $X \in \mathscr{X}$ and $a \in dom(X)$, is generalized arc-consistent (GAC) iff for all $C \in \mathscr{C}$ such that $X \in scp(C)$, there exists a support of (X, a) in C. P is GAC iff for all $X \in \mathscr{X}$, $dom(X) \neq \emptyset$ and (X, a) is GAC for all $a \in dom(X)$.*

Finally, we introduce the useful notion of cn-value.

Definition 3 *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN. A cn-value is a triplet of the form (C,X,a) where $C \in \mathscr{C}$, $X \in scp(C)$ and $a \in dom(X)$.*

3. Residual Supports

For some (global) constraints, it is possible to define efficient specific filtering algorithms to enforce GAC. However, when the semantics of the constraints is unknown or cannot be exploited, it is necessary to adopt a generic approach. One simple and well-known generic algorithm for enforcing (G)AC is called (G)AC3 (Mackworth (1977a,b)). In this section, we illustrate the concept of residual support (or more simply, residue) with GAC3.

A residue is a support that has been stored during a previous execution of the procedure which determines if a value is supported by a constraint. A crucial difference with the data structure last-support used in AC2001/3.1 is that a residue is not guaranteed to represent

a lower bound of the smallest current support of a value. The concept of residue has been introduced under its multi-directional form in Lecoutre et al. (2003) and under its uni-directional form in Likitvivatanavong et al. (2004). We explain below the distinction between the two forms.

3.1 GAC3 with residual supports

The new algorithm that we obtain from GAC3 when exploiting residues is denoted GAC3^{rm} (Algorithm 1) and it requires the introduction of a three-dimensional array, denoted *res*, which is used to store for any cn-value (C,X,a) its associated residue — the last found support of (X,a) in C. Initial values of *res* are set to $\perp$ and all pairs (C, X), called arcs, are put in a set Q. Once Q has been initialized, each arc is revised in turn; when a value has been removed, the effect must be propagated by updating the set Q (Line 6).

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

1 **for** each $C \in \mathscr{C}, X \in scp(C), a \in dom(X)$ **do** $res[C, X, a] \leftarrow \perp$
2 $Q \leftarrow \{(C, X) \mid C \in \mathscr{C} \wedge X \in scp(C)\}$
3 **while** $Q \neq \emptyset$ **do**
4 extract (C, X) from Q
5 **if** $revise(C, X)$ **then**
6 $Q \leftarrow Q \cup \{(C', X') \mid C' \in \mathscr{C}, C' \neq C, X' \neq X \wedge \{X, X'\} \subseteq scp(C')$

A revision is performed by a call to the function *revise* described by Algorithm 2. A revision involves a validity check for each value's residue (Line 3). If the check fails, a new support is searched from scratch (Line 4). If a support is found, residues are updated; otherwise the value is removed.

Residues are updated (Line 6 of Algorithm 2) by exploiting *multi-directionality*: the fact that the tuple found as support of (X, a) in C is also a support for the other value of this tuple. As a result, we obtain $r - 1$ residues for other values in the tuple with no effort. The uni-directional form on the other hand records for (X, a) the tuple found as support of (X, a) in C (the entire for-loop in line 6 of Algorithm 2 is replaced by $addResidue(C, X, t)$.

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

1 $nbElements \leftarrow |dom(X)|$
2 **for** *each* $a \in dom(X)$ **do**
3 **if** $isValid(C, res[C, X, a])$ **then** continue
4 ingle statement $t \leftarrow seekSupport(C, X, a)$
5 **if** $t = \top$ **then** remove a from $dom(X)$
6 **else** **for** *each* $Y \in vars(C)$ **do** addResidue(C, Y, t)
7 **return** $nbElements \neq |dom(X)|$

Function $addResidue$(C:Constraint, X:Variable, t:Tuple) simply assigns t to $res[C, X, t[X]]$. Function $isValid$ (Algorithm 3) determines whether or not the given tuple is valid. Function $seekSupport$ (Algorithm 4) determines from scratch whether or not there exists a support of (X, a) in C. It uses $setNextValid$ (not detailed) which returns either the smallest valid

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

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

tuple t' built from C such that $t \prec t'$ and $t'[X] = a$, or $\top$ if it does not exist. Note that $\bot$ is considered invalid and $\top$ does not belong to any relation.

3.2 Theoretical foundations of residues

In this section we present some results for algorithms that exploit residues in binary constraint networks. In particular, we study the complexity of AC3^{rm} when invoked as a stand-alone algorithm and when embedded in MAC. Without loss of generality, we assume that each domain contains exactly d values in order to simplify some results.

Proposition 4 *AC3^{rm} admits a worst-case space complexity in $O(ed)$ and a worst-case time complexity in $O(ed^3)$.*

Proof We assume here that each constraint is represented in intention (i.e. by a predicate) in $O(1)$ and that $n < e$ (otherwise, the network contains several connected components). The space required to represent the constraint network is $O(e + nd)$, the space required by Q is $O(e*2) = O(e)$ and the space required by the data structure res is $O(e*2*d) = O(ed)$. We then obtain $O(ed)$. The number of operations (validity or constraint checks) performed by AC3^{rm} is bounded by the number of operations performed by AC3. As AC3 is $O(ed^3)$, AC3^{rm} is also $O(ed^3)$. ■

Interestingly, it is possible to refine the analysis when considering the tightness of the constraints (the ratio of the number of allowed tuples to the total number of possible tuples in a constraint). For random constraints, we can precisely determine the cost of seeking a support.

Proposition 5 *If C is a random binary constraint of tightness $t < 1$, the expected number of constraint checks performed in seekSupport is asymptotically (when d tends to infinity) equal to $1/(1-t)$.*

Proof The expected number of constraint checks is equal to $E(X) = 1 * (1-t) + 2 * t * (1-t) + 3 * t^2 * (1-t) + \ldots$ since, when *seekSupport* is called, the probability of making only one constraint check is $1-t$, the probability of making exactly two constraint checks

is $t * (1-t)$, etc. We then obtain $E(X) = (1-t) * \sum_{i=1}^{\infty} i * t^{i-1}$. As $t < 1$, we can show that $\sum_{i=1}^{\infty} i * t^{i-1} = 1/(1-t)^2$. We deduce $E(X) = 1/(1-t)$. ■

The first significant result is the following.

Proposition 6 *Applied to a random constraint network, $AC3^{rm}$ admits an average time complexity in $O(ed^2)$.*

Proof The number of validity checks is clearly $O(ed^2)$ in the worst-case since for each cn-value, at most d validity checks can be performed. In the worst-case, for a given cn-value (C, X, a), the number of calls to *seekSupport* is bounded by $s+1$ with s denoting the number of supports of (X, a) in C and we know from Proposition 5 that each call to *seekSupport* requires on average $1/(1-t) = d/s$ constraint checks. We then obtain an average time complexity in $O(e * 2 * d * (s+1) * d/s) = O(ed^2)$. ■

We can obtain another interesting result when considering tightness-bounded constraints defined as follows.

Definition 7 *A constraint C is said to be tightness-bounded iff for any cn-value of C, either its number of supports is $O(1)$ or its number of conflicts is $O(1)$ (when d, the domain size of each variable, tends to infinity).*

There are some usual constraints that are tightness-bounded. For example, a constraint of equality $X = Y$ or a constraint of difference $X \neq Y$ is tightness-bounded. In the former case, each value is supported at most one time and in the latter case, each value admits at most one conflict. In practice, the following result indicates that, when applied to constraints of small or high tightness, AC3^{rm} behaves in an optimal way. The proof of this proposition can be derived from results that can be found in Lecoutre and Hemery (2007).

Proposition 8 *Applied to a constraint network involving tightness-bounded constraints, $AC3^{rm}$ admits a worst-case time complexity in $O(ed^2)$, which is optimal.*

Propositions 6 and 8 indicate that AC3^{rm} has an optimal behavior both when constraints are random (on average) and when constraints are highly structured. This suggests that AC3^{rm} should be quite competitive, compared to an optimal algorithm such as AC2001/3.1. These results are confirmed when the state-of-the-art generic algorithm MAC Sabin and Freuder (1994) is considered. MAC is the algorithm that maintains arc consistency during the search of a solution and that can embed any of the generic AC algorithms proposed in the literature (for example, MAC3^{rm} is MAC embedding AC3^{rm}). The following results are directly obtained from previous propositions, the fact that no maintenance of data structures is required upon backtracking and the fact that arc consistency is an incremental property.

Proposition 9 *$MAC3^{rm}$ admits a worst-case space complexity in $O(ed)$ and a worst-case time complexity in $O(ed^3)$ for any branch of the search tree.*

Proposition 10 *Applied to a random constraint network, $MAC3^{rm}$ admits an average time complexity in $O(ed^2)$ for any branch of the search tree.*

Proposition 11 *Applied to a constraint network involving tightness-bounded constraints, $MAC3^{rm}$ admits a worst-case time complexity in $O(ed^2)$ for any branch of the search tree.*

All these theoretical results have been confirmed by the experimental results obtained in Likitvivatanavong et al. (2004); Lecoutre and Hemery (2007). Besides, they still hold if we consider a bounded number of residual supports to be associated with each cn-value, instead of a single one. In this paper, we will then focus our attention to the practical interest of using multiple residues.

4. Fundamental of Multiple Residues

Given excellent performance of a single residue in Maintaining (Generalized) Arc Consistency algorithm (Lecoutre and Hemery (2007); Likitvivatanavong et al. (2004)), it is interesting to see whether better performance could be achieved just by increasing the number of residues. Unlike single residue, we not only record just the latest support, but some of the past supports found as well. However, as soon as multiple residues are in use we must be concerned with the follow questions:

1. When will the current support become a residue?
2. How to choose a residue to be replaced?
3. In which order are residues examined?
4. What is the appropriate number of residues?

In general, if all residues fail validity check the algorithm should look for a new support and this support will become one of the residues recorded. If there is no more space, the residue that has the least chance of being a support in the future will be removed to make room for a new one. Although there is no precise method for determining this probability, a number of heuristics can be employed to evaluate the utility of each residue. The algorithm should then give higher priority to residues with better ranking and perform validity check on them first. In addition, these residues should be arranged so that examining them in order of utility be as fast as possible. Good heuristics are therefore crucial in keeping likely supports in the residue store and lowering the number of validity check performed.

Specifically, we adopt the following scheme. To determine the utility of a residue, a numerical value will be computed for each support found, where the utility function varies depending on the residue replacement policy used. Each cn-value will be associated with a residue store, which contains tuples (t, u), where t is a residue and u its utility score. The algorithm searches through residue store in descending order according to the utility score. When the residue store is full and none of the residues is valid, we pick the tuple that contains the residue with the lowest utility score, compare it with that of the current support, and retain the one with higher score.

4.1 A generic algorithm

Algorithms that use multiple residues have almost the same structure as those for single residue. The difference lies in the routines that update residues and check their validity. We change the code for *addResidue* in Algorithm 2 to the one in Algorithm 5 and replace *isValid* in Algorithm 2 with *existValid* (Algorithm 6).

We use the following convention. $maxR$ is the maximum number of residues. f represents the utility function. $res[C, X, a]$ is now a Dictionary (a dynamic data structure) S that allows the following operations:

- $max(S)$, $min(S)$. Return the tuple (t, u) where u is the highest (lowest) utility score.
- $predecessor(S, s)$, $successor(S, s)$. Return the tuple (t, u) where u is the next smaller (larger) score than that of s, return $\perp$ ($\top$) when there is none.
- $insert(S, (s, w))$, $delete(S, (s, w))$. Insert and delete the tuple (s, w) where s is a residue and w its utility score.
- $update(S, s, w)$. Replace the score of s with w and update the tuple's position in S.
- $size(S)$. Return the number of tuples in S.

In practice, Dictionary can be implemented using a variety of data structure. Implementation that employs balanced tree, for instance (e.g. Cormen et al. (2001)), requires $O(\lg n)$ in the worst-case for each of the operations.

It should also be emphasized that the current support should not automatically become a residue when the residue store is full. Although the existence of the support is the reason a value is kept in its domain for the *current* revision, the tuple may lose its validity and its status as the support for the *next* revision. For a support to become a residue, its utility score of a support should improve upon the minimum score (Line 3 of Algorithm 5).

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

```
1 if size(res[C, X, t[X]]) = maxR then
2     (t_0, u_0) ← min(res[C, X, t[X]])
3     if u_0 < f(t) then delete(res[C, X, t[X]], (t_0, u_0))
4 if size(res[C, X, t[X]]) < maxR then  insert(res[C, X, t[X]], (t, f(t)))
```

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         update(D, t, f(t))          /* dynamic and fully dynamic utility update only */
5         return true
6     update the utility score of t                          /* fully dynamic update only /*
7     (t, u) ← predecessor(D, t)
8 return false
```

5. Residue Replacement Policies

We describe some policies for residue replacement in this section. Each of them may have further three classifications based on the way utility scores are updated: static, dynamic, and fully dynamic. Static policy computes utility score only once for each support when it becomes a residue. Dynamic policy recomputes the score for valid residues as well (Line 4 of Algorithm 6). In addition, fully dynamic policy updates the score of a residue whenever a validity check is performed, regardless of the result (Line 6 of Algorithm 6).

For each factor described below we derived two opposite policies:

1. *Level of the search tree.* The utility function returns the level of the search tree when the residue was found. The rational is that a support found at a deeper level should be more robust to change in the network. When the search backtracks, the support found at a deeper level remains valid since the only change to the domains is the restoration of pruned values and it does not affect values already present in the domains. Only after the search tree branches off to a different path may the support lose its validity. LevelMin policy replaces the residue with the shallowest level first when the residue store is full. Moreover, a residue with deeper level is checked for validity before another with shallower level. LevelMax performs the opposite.

2. *Domain size.* The utility function returns a value based on the size of the relevant domains when the residue was found. We use the following function: if $revise(C, X, a)$ is invoked and the tuple t is found as a support, then $f(t) = \sum_Y |dom(Y)|$ for all $Y \in scp(C)$ such that $Y \neq X$. DomMin policy replaces the residue with the smallest domains first and performs validity check on a residue with larger domains before another with smaller domains. DomMax performs the opposite.

3. *Chronology.* The utility function returns the time stamp when the residue was found. Fifo("First In First Out") policy replaces the oldest residue first and check residues from the latest to the oldest. Lifo ("Last In First Out") performs the opposite.

4. *Frequency.* The utility function adds one to the score for every successful validity check for a residue, and subtracts one for every unsuccessful check. ValidMin replaces the residue with the lowest score first and check a residue with higher score before another with lower score. ValidMax performs the opposite.

6. Specific Implementations

Although conceptually the Dictionary used in order to store residues can accommodate any finite number of them, there is no guarantee that more residues would yield better performance. Indeed, our experimental results has shown that large number of residues has an adverse affect on the performance.

In this section we provide specific implementations for some policies described in previous section. We use array, a fast-accessed data structure, to hold residue since the practical number of residues is expected to be small. Data structure such as balanced tree has much larger overhead although it may be more efficient when the number of residues is large enough. Still, array has an advantage of being cache-awared (Mitchell (2005)).

We use the following convention. Array indices range from 0 to $maxR - 1$ where $maxR$ is the array size (the maximum number of residues). Given an array D, $D.size$ is the current number of residues; $size$ are set to zero initially. In the routine *addResidue*, we presuppose the following statement: $D \leftarrow res[C, X, t[X]]$.

6.1 FIFO policy

We use circular array to hold residues for static and fully dynamic FIFO. There is no need to record the utility score since it is already implicit in the ordering of residues. Given array D, $D.head$ refers to the index to the first residue to be checked; $head$ is set to zero initially.

When a residue r is found to be valid in the circular array $(q_1, \cdots, q_i, r, s_1, \cdots, s_j)$, where $head$ points to q_1, fully dynamic FIFO will set $head$ to point to r, since it has now become the most recent support. Moreover, the i invalid residues q_1 to q_i are automatically placed at the end of the circular array as a side-effect. Consequently the j residues s_1 to s_j will be checked for validity before q_1 to q_i.

By contrast, the utilities of the i invalid residues q_1 to q_i are not updated for dynamic FIFO. We use standard array to hold residues for dynamic FIFO and simply move the residue r to the front and shift the i invalid residues one position to the back. The result is $(r, q_1, \cdots, q_i, s_1, \cdots, s_j)$, where $head$ points to r. Linked-list can be used as an alternative implementation to avoid array copying in line 2 of Algorithm 9 and line 4 of Algorithm 10, although it is not cache-awared.

Algorithm 7: addResidue(C, X, t) `(static and fully dynamic FIFO)`

```
1 if D.size < maxR then D.size ← D.size + 1
2 D.head ← (D.head − 1 + maxR) mod maxR
3 D[D.head] ← t
```

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 maxR
3     if isValid(C, D[p]) then
4         D.head ← p                    /* fully dynamic FIFO only */
5         return true
6 return false
```

Algorithm 9: addResidue(C, X, t) `(dynamic FIFO)`

```
1 if D.size < maxR then D.size ← D.size + 1
2 for p ← D.size − 1 down to 0 do D[p] ← D[p − 1]
3 D[0] ← t
```

Algorithm 10: existValid(C, D) (dynamic FIFO)

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

6.2 Static and dynamic LEVEL/DOM policy

Routines for LEVEL and DOM policy are given in Algorithm 11 and 12. Like dynamic FIFO, we use standard array to hold residues and shift them around to maintain the ordering. Residues are recorded together with their utility scores; given array D, $D[i]$ refers to the tuple (t, u) stored at the ith position, where t is the residue and u its utility score. We also use $D[i].tuple$ to refer to t and $D[i].score$ to refer to u. The residues are arranged in descending order according to the utility, with ties broken by giving priority to the most recent support (Line 3 of Algorithm 11, and Line 8, 13 of Algorithm 12).

LEVELMIN and DOMMIN can be achieved by setting the utility function as described in section 5. LEVELMAX and DOMMAX can be achieved by setting the utility functions to be the reverse. For instance, if $f(t)$ is the utility function of LEVELMIN then the utility function of LEVELMAX would be $-f(t)$.

Only static and dynamic policy are described. Fully dynamic policy updates invalid residues with new utility scores and it requires full sorting to re-establish the ordering. This is too expensive for our purpose. On the other hand, dynamic policy needs only to shift a single element to its appropriate place (Lines 6 to 16 of Algorithm 12).

Algorithm 11: addResidue(C, X, t) (static and dynamic LEVEL/DOM)

```
1  (foundPos ← false) and (p ← 0)
2  while not foundPos and p < D.size do
3      if D[p].score ≤ f(t) then  foundPos ← true  else  p ← p + 1
4  if foundPos or D.size < maxR then
5      q ← D.size
6      if q = maxR then q ← maxR − 1  else  D.size ← D.size + 1
7      while q > p do
8          D[q] ← D[p − 1]
9          q ← q − 1
10     D[p] ← t
```

6.3 VALID policy

We use standard array for VALIDMIN and maintain the descending order. The order is opposite for VALIDMAX. Unlike the case for LEVEL and DOM, the utility function is unchanged for the two VALID policies and we cannot use the same code for both.

Algorithm 12: existValid(C, D) (static and dynamic LEVEL/DOM)

```
1  (found ← false) and (p ← 0)
2  while not found and p < D.size do
3      (t, u) ← D[p]
4      if isValid(C, t) then found ← true else p ← p + 1
5  /* The following if-block is for dynamic policy only */
6  if found and f(t) ≠ u then
7      if f(t) > u and p > 0 then
8          while p > 0 and D[p − 1].score ≤ f(t) do
9              D[p] ← D[p − 1]
10             p ← p − 1
11         D[p] ← (t, f(t))
12     if f(t) < u and p < D.size − 1 then
13         while p < D.size − 1 and D[p + 1].score > f(t) do
14             D[p] ← D[p + 1]
15             p ← p + 1
16         D[p] ← (t, f(t))
17 return found
```

We only describe dynamic and fully dynamic VALIDMIN since static VALIDMIN is uninteresting: every residue has a fixed score of 1 at any time. *addResidue* for dynamic and fully dynamic VALIDMIN are shown in Algorithm 13. Note that the utility score for dynamic VALIDMIN always increases so it may be possible that all residues have score of more than one. In this case, if we insist that the score of the new support must at least as good as the the lowest score (as done in fully dynamic VALIDMIN) the new support would never be recorded since its initial score is one.

The code of *existValid* for fully dynamic VALIDMIN is given in Algorithm 14. If the residue at position p is found to be valid its score would be increased by one while those of the residues in position $0, \ldots, p-1$ would be decreased by one. Afterward, we re-order the array by performing insertion sort, calling the routine *insertion-sort*(elt, A) where A is a sorted array and *elt* the element to be inserted. Because residues from position p onward remain in descending order, it forms a sorted sub-array. We insert the residue in position $p-1$ into the sub-array so that the size of the sorted sub-array increases by one. Subsequently, the residues in position $p-2$ down to 0 are inserted one by one until the size of the sub-array reaches its maximum, at which point the whole array is sorted.

We do not provide the code for dynamic VALIDMIN since it is simpler: there is only a single residue whose position needs to be adjusted. Dynamic and fully dynamic VALIDMAX can be done in a similar fashion although the ordering is ascending rather than descending.

7. Experimental Results

An extensive experiment is conducted to study the performance of various cache size and policies. Our solver was compiled by g++ 3.4.3, and the experiment was carried out on a DELL PowerEdge 1850 (two 3.6GHz Intel Xeon CPUs) with Linux 2.4.21. We use dom/deg variable ordering and lexicographical value ordering.

Algorithm 13: addResidue(C, X, t) `(dynamic and fully dynamic ValidMin)`

```
if D.size < maxR then
    D[D.size] ← (t, 1)
    D.size ← D.size + 1
else
    D[D.size − 1] ← (t, 1)                                  /* dynamic ValidMin only */
    if D[D.size − 1].score ≤ 1 then  D[D.size − 1] ← (t, 1)  /* fully dynamic ValidMin only */
```

Algorithm 14: existValid(C, D) `(fully dynamic ValidMin)`

```
found ← false
for p ← 0 to D.size − 1 do
    if isValid(C, D[p].tuple) then
        D[p].score ← D[p].score + 1
        for q ← p − 1 down to 0 do  insertion-sort(D[q], D[q + 1, ..., D.size − 1])
        return true
    else  D[p].score ← D[p].score − 1
return false
```

The bencharmarks we use are summarized below. They are classified into two categories: random problems, academic and real world problems **?** where the problem names like ehi-85-297-12 are from.

Random problems. We select a set of prblems with varying domains, and all of them are in the phase transition region. A class of random problems can be characterised by $(n, d, e, tightness)$ where n is the number of variable, d the maximum domain size, e the number of constraints, and *tightness/1000* is the probability of a tuple not allowed by a constraint. Specifically, the problems used are R1=$(40, 8, 753, 100)$, R2=$(40, 11, 414, 200)$, R3=$(40, 16, 250, 350)$, R4=$(40, 25, 180, 500)$, R5=$(40, 40, 135, 650)$, R6=$(40, 80, 103, 800)$, and R7=$(40, 180, 84, 900)$.

Academic and real world problems are list below. P1 (ehi-85-297-12) and P2 (ehi-85-297-13) are unsatisfiable 3-SAT instances converted to binary CSP instances; P3 (frb40-19-3) and P4 (frb35-17-5), forced instances of model RB; P5 (pigeons-10) and P6 (pigeons-11), pigeon hole problems; P7 (qa-5) and p8 (qa-6), queen attacking problem; P9 (qk-20-0-5) and P10 (qk-25-0-5), queens-knight problems; P11 (fapp01-0200-8) and P12 (fapp01-0200-9), frequency assignment problems with polarization constraints; P13 (graph-10), P14 (graph-14), P15 (scen-11) and P16 (scen-05), radio link frequency assignment problems (RLFAP).

We have implemented both static and dynamic versions for policies of FIFO, levelMin, levelMax, domMIN, domMAX; both dynamic and full dynamic version of validMin and validMax; and full dynamic verion of FIFO. In our system, every policy is euipped with a residue store of any given size. For each problem from R1 to R7 and from P1 to P16, and each policy, we collect the performance data of the policy against the store size varying from 1 to 10. *All the analysis and observation apply to this data unless it is mentioned otherwise.* The performance of the algorithms is measured by the number of cosntraint checks, the number of domain checks and cpu time that are used to solve a problem. Clearly, all the algorithms will be more effiecient than the traditional ones when the constraint checks are

expensive. To test their performance in the worst case, the constraint check in our system is made extremely cheap.

The objective of the experiment is to test how the cache size and policies affect the effectiveness of a CSP solver. The analysis of the experimental data is presented in the following dimensions: 1) How the cache size affect the performance of a policy? 2) How the policies perform in solving a problem? and 3) How different versions of a policy perform against each other?

7.1 Policies

In this sub section, we study which policies are more effective in solving a problem. The first observation is made on the relation of the opposite heuristics like domMin vs. domMax. We all heuristics of the policies of FIFO, domMax, levelMin and validMin and anti-heuristics the their opposites.

We did not implement LIFO because the "old" residues in the store will never be discarded. It's expected to perform badly most of the time.

The experimental data shows that domMAX, levelMIN performs better than domMin, levelMax. The results are presented in Figure 5 and 6. X-axis is a sequence of all pairs of (pi, c) where pi is a problem and c a cache size. Our explanation is that the support found when the relevant domain is smaller should be "stronger". The similar reason is applicable to levelMIN.

It is first a surprise to see that validMax is beaten clearly by validMin in the data in Figure 7 and 8 where x-axis has the same meaning as that of Figure 5 and 6. One explanation is that validMax is in fact very close to FIFO which has been shown very effective and robust in our data. validMin less favor the new (i.e., the most recent) support. The residues used most frequently might be less relevant in the future.

New we look at top performers of the policies. The performance (both time and ccks) of FIFO, domMAX, levelMIN, and maxUSE is quite close for the academic and real world problems (Figure 10).

For random problems, FIFO approach is better than the other methods in terms of both cpu time and constraint checks (Figure 9).

static vs. dynamic policies For easier problems, the difference between static policy and dynamic one is small. For the harder problems like P6 and P15, dynamic approach has less ccks for FIFO, maxDomain, and minLevel while the converse is true for anti-heuristics policies. However, as for cpu time, the dynamic approach is slower in our experiment due to cheap constraint checks and higher cost in maitaining residue store for dynamic method.

7.2 Residue Store Size

It is clear that, as the cache size increases, the number of constraint checks will be decreased and the cost of managing residues could increase. We show how they changed in our benchmark problems. In this section, we *will not* consider the polycies associated with anti-heuristics.

The residue store size is also called cache size here for short. From the experimental data, the contraint checks decrease very quickly when the cache size increases, converging very quickly to a stable number (as one can see when the cache size is big enough, the constraint

check of a given tuple will be made only once). The domain checks increase linearly to the cache size. The observation applies to all our benchmark problems. In Figure 1 and 2, we list the constraint checks against cache sizes for problems P15 and R7 for wich the number of constraint checks converges the slowest. The bottom chart of Figure 1 shows the number of constraint checks for each (c, p) where c is cache size and p an algorithm. X-axis represents pairs (c, p) which are odered in a way their number of constrait checks are increasing. The X-axisis of the top and bottom chart uses the same order of the (c, p) pairs. The curve of the middle chart represents the cache size corresponding to a pair (c, p), and that of the top one reprsents the valid checks needed for each pair (c, p). It can be observed that as the cache size increases (from right to left in the middle chart), the number of constrait checks decreases rapidly in the bottom chart. However, the number of domain checks (in the top chart) is linear to the cache size. A remark here is that Figure 1 is just an *approximate* visualization of the huge amount of data. Our manual check of the data also verifies the relationship between number of constraint checks and cache sizes. The meaning of Figure 2 is similar to Figure 1 while the former is on a real world problem.

To illustrate the observation, we list in Figure 11 and 12 the performance of one algorithm (FIFO static) agaist all cache size and problems. Specifically, the pairs (cachesize, problem) of the x-axis are ordered first in cache size and then problem numbers. For example, the first period (with two peaks) of the curve is for all problems with cachesize being 1, the second period is that with cache size of 2 etc.

As a more detailed sample, in Figure 3, we list the constraint checks, domain checks, and cpu time used by FIFO algorithms, represented by 10, 11, and 12, to solve a hard problem P6 with varying cache sizes. Figure 4 is for the same set of data for a hard random problem class R7.

As for CPU time, the gain by increasing cache size is not very significant. Given the fact that we use an extrem check constraint check, as the cost of constraint check increases, the grain will be quite clear. The time gain caused by larger cache size is quite clear for the hard random problem R7.

From Figrue 3 and Figure 4, it is also clear that as cachesize increases, the cpu time will also increase. This can be explained by the following facts. The constraint checks begin to converge to a constant number after cache size is bigger than 5. However, the domain checks increases steadily across the whole range of cache sizes from 1 to 10. From our experiment, 2-5 are good choices for the cache size.

Next, we carried out experiments using Abscon, one of the best solvers from the Second International CSP Solver Competition (see results at http://www.cril.univ-artois.fr/CPAI06/). Results on selected 10 binary instances and 10 non-binary instances are shown in two tables on the following pages. Columns named 2001 and 3 are the results for M(G)AC2001 and M(G)AC3. The rest are results for M(G)AC3 with varying number of residues and whether multi-directionality is exploited. Table legend: *cpu* CPU time in second, *mem* the maximum amount of memory used, *ccks* the number of constraint checks, *vcks* the number of validity checks. For each instance, the three best results in CPU time are shades in gray, with the best one being the darkest.

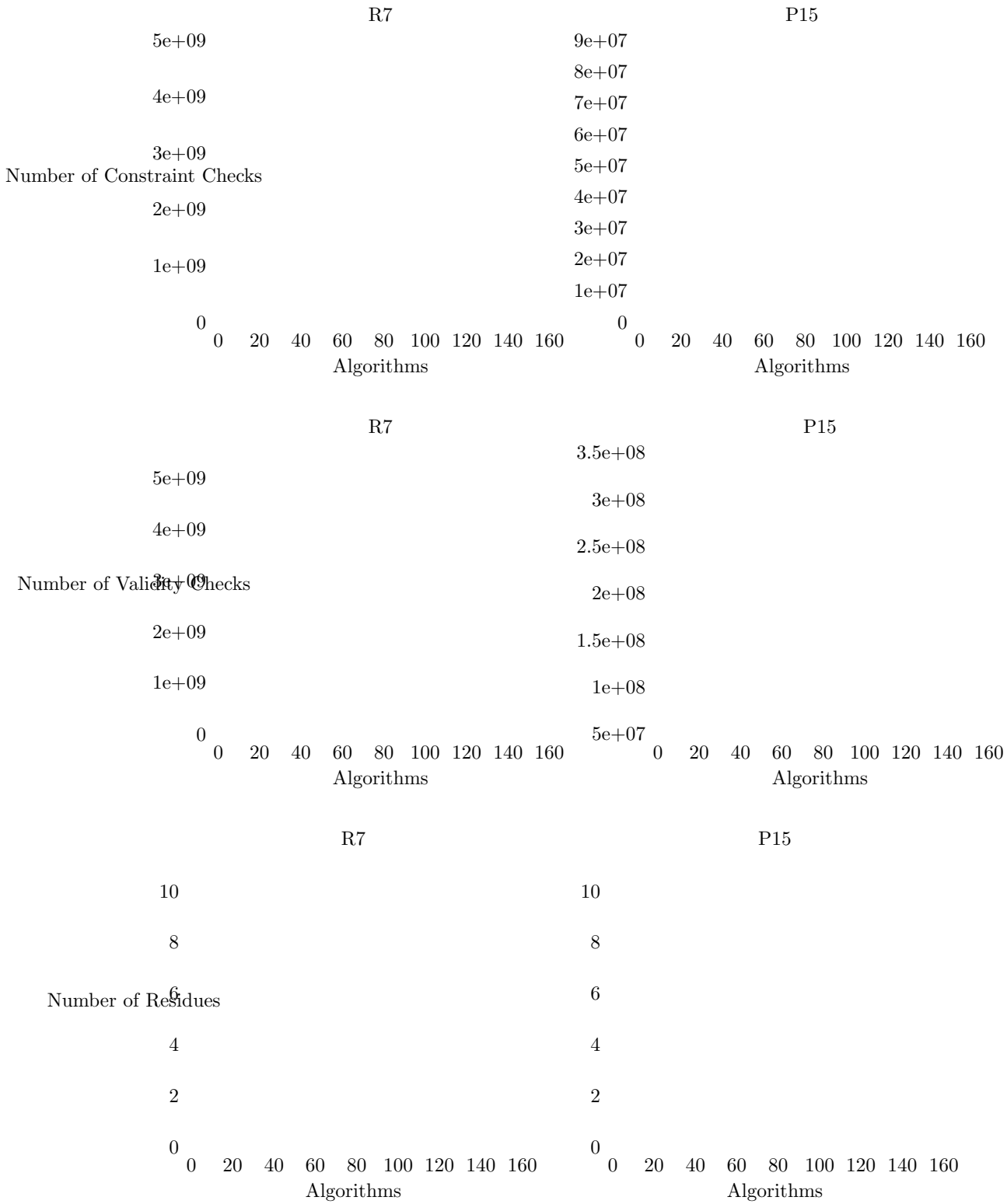


Figure 1: Trends for the number of constraint checks, the number of validity checks, and the number of residues.

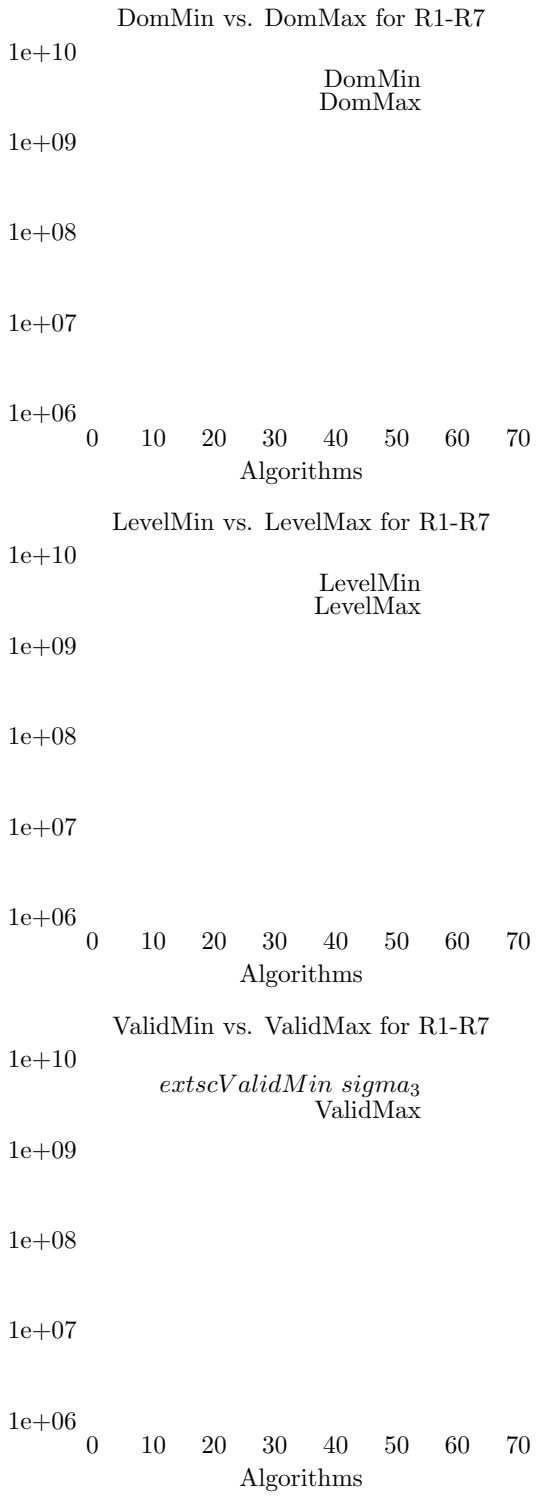
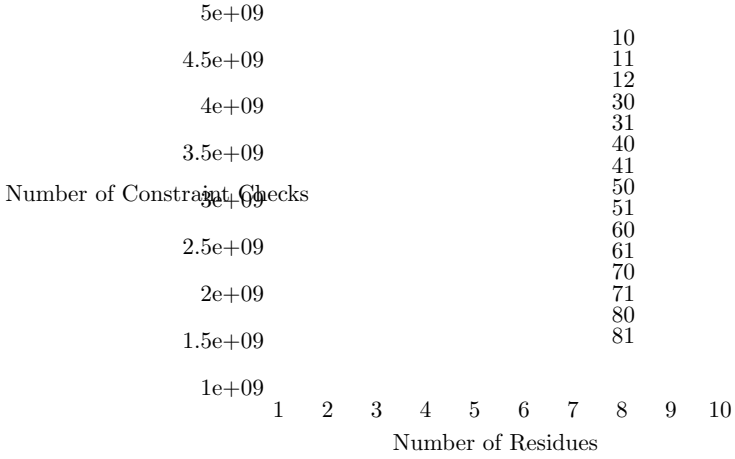


Figure 2: Comparisons between heuristics and anti-heuristics.



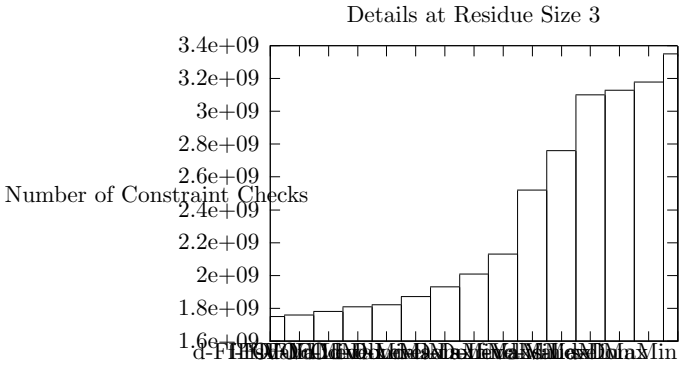


Figure 3: Experiments:

Binary Instances		2001	3	1*r*	1*rm*	2*r*	2*rm*	3*r*	3*rm*	4*r*	4*rm*
bqwh-18-141-0-extall100	*cpu*	3.142	2.991	3.018	3.099	3.067	3.098	3.138	3.205	3.184	3.218
	mem	12M	10M	15M	15M	19M	19M	20M	20M	20M	20M
	ccks	2192K	2192K	1117K	1127K	748K	748K	665K	666K	651K	650K
	vcks	0	0	2183K	2185K	3291K	3296K	4176K	4182K	5043K	5048K
e0ddr1-10-by-5-10	*cpu*	13520.64	17021.33	10967.15	10875.8	10012.09	9259.86	10566.25	10002.35	10018.3	10791.03
	mem	11M	10M	11M	11M	11M	11M	17M	17M	17M	17M
	ccks	79M	171M	75M	75M	63M	63M	62M	62M	61M	61M
	vcks	33M	0	46M	46M	66M	66M	80M	80M	94M	94M
enddr1-10-by-5-2	*cpu*	85.52	146.39	67.22	70.88	62.8	60.71	70.0	61.37	70.27	65.03
	mem	9284K	7546K	8177K	8177K	8704K	8704K	18M	18M	18M	18M
	ccks	154K	188K	161K	161K	161K	161K	161K	161K	161K	161K
	vcks	5361	0	10754	10951	13736	14103	16718	17242	19700	20381
fapp05-0350-0all11	*cpu*	48.641	47.934	47.551	47.7	47.812	48.212	49.03	49.023	49.093	46.39
	mem	73M	43M	65M	64M	84M	84M	180M	179M	180M	180M
	ccks	2076K	3103K	1980K	1873K	1905K	1795K	1904K	1794K	1904K	1794K
	vcks	66842	0	104K	108K	135K	139K	159K	164K	184K	189K
frb50-23-1-mgd-extall5	*cpu*	3462.238	3044.222	2332.218	2374.424	2190.786	2290.472	2658.74	2849.956	2784.47	2857.532
	mem	13M	12M	13M	13M	17M	17M	18M	18M	18M	18M
	ccks	7062M	12330M	4710M	4619M	2875M	2818M	2174M	2217M	1783M	1863M
	vcks	5872M	0	9297M	9297M	12835M	12817M	15080M	15069M	16965M	16981M
geo50-20-d4-75-100-extall100	*cpu*	12.312	9.973	8.826	9.385	8.933	8.679	10.075	10.986	10.461	10.254
	mem	11M	10M	11M	11M	13M	13M	17M	17M	17M	17M
	ccks	25M	40M	18M	18M	12M	12M	9341K	9207K	7696K	7688K
	vcks	19M	0	32M	32M	46M	46M	56M	56M	64M	64M
qcp-20-187-0-ext	*cpu*	538.97	432.0	451.41	490.09	506.47	505.0	582.13	570.9	532.76	563.98
	mem	34M	30M	57M	57M	77M	77M	81M	81M	81M	81M
	ccks	457M	457M	190M	194M	86M	87M	60M	62M	54M	56M
	vcks	0	0	457M	457M	647M	648M	773M	774M	892M	892M
qwh-20-166-0-extall10	*cpu*	254.823	195.843	239.482	244.324	239.184	249.216	266.803	259.3	270.763	269.16
	mem	33M	21M	48M	48M	70M	70M	83M	84M	84M	83M
	ccks	204M	204M	85M	86M	32M	32M	17M	18M	14M	14M
	vcks	0	0	204M	204M	289M	289M	340M	340M	388M	388M
rand-2-40-25-180-500-0-extall100	*cpu*	30.02	29.34	22.552	22.562	22.344	21.605	23.926	22.368	22.999	23.65
	mem	8484K	7955K	8224K	8224K	8354K	8353K	9314K	9314K	9315K	9314K
	ccks	89M	169M	64M	63M	43M	42M	35M	35M	30M	31M
	vcks	70M	0	100M	100M	137M	137M	163M	163M	186M	186M
scen1all11	*cpu*	1.398	1.36	1.387	1.423	1.54	1.49	1.58	1.635	1.766	1.537
	mem	37M	22M	44M	44M	60M	60M	88M	88M	87M	88M
	ccks	502K	949K	473K	410K	419K	356K	403K	340K	390K	326K
	vcks	189K	0	340K	345K	535K	541K	689K	694K	833K	838K

Non-binary Instances		2001	3	1*r*	1*rm*	2*r*	2*rm*	3*r*	3*rm*	4*r*	4*rm*
aim-200-2-0-1	*cpu*	240.35	192.86	182.29	195.13	178.92	189.99	194.21	181.4	194.25	208.55
	mem	25*M*	25*M*	29*M*	29*M*	33*M*	33*M*	33*M*	33*M*	34*M*	34*M*
	ccks	56*M*	59*M*	30*M*	30*M*	27*M*	27*M*	27*M*	27*M*	27*M*	27*M*
	vcks	2042*K*	0	58*M*	58*M*	88*M*	88*M*	116*M*	116*M*	144*M*	144*M*
air04	*cpu*	41.74	44.68	41.79	40.68	40.83	43.16	42.63	42.27	41.85	42.63
	mem	304*M*	136*M*	290*M*	291*M*	321*M*	324*M*	398*M*	398*M*	480*M*	480*M*
	ccks	31521	321*K*	31521	17712	31521	17712	31521	17711	31521	17711
	vcks	290*K*	0	290*K*	304*K*	290*K*	308*K*	290*K*	311*K*	290*K*	315*K*
blast-tlan3	*cpu*	187.09	237.09	196.04	196.98	176.97	206.11	205.7	196.73	200.69	191.11
	mem	487*M*	402*M*	508*M*	508*M*	480*M*	457*M*	505*M*	534*M*	574*M*	575*M*
	ccks	7328*K*	28*M*	9979*K*	8604*K*	9979*K*	7991*K*	9979*K*	7991*K*	9979*K*	7990*K*
	vcks	1222*K*	0	1227*K*	1306*K*	1238*K*	1363*K*	1249*K*	1378*K*	1260*K*	1392*K*
dubois-24-ext	*cpu*	1272.38	1243.97	1140.3	1147.67	1121.4	1103.96	1224.07	1125.77	1206.98	1168.23
	mem	21*M*	21*M*	25*M*	25*M*	25*M*	25*M*	25*M*	25*M*	25*M*	25*M*
	ccks	429*M*	1122*M*	466*M*	393*M*	290*M*	290*M*	290*M*	290*M*	290*M*	290*M*
	vcks	1010*M*	378*M*	1366*M*	1336*M*	1747*M*	1747*M*	2117*M*	2117*M*	2487*M*	2487*M*
lemma-50-9-mod	*cpu*	599.62	529.43	521.62	393.97	552.27	496.69	525.81	545.48	490.17	530.63
	mem	25*M*	21*M*	29*M*	29*M*	33*M*	33*M*	33*M*	33*M*	33*M*	33*M*
	ccks	1171*M*	1402*M*	1154*M*	719*M*	946*M*	621*M*	660*M*	597*M*	574*M*	551*M*
	vcks	233*M*	0	504*M*	504*M*	868*M*	759*M*	1126*M*	980*M*	1347*M*	1199*M*
mps-enigma	*cpu*	1516.46	1564.36	1497.12	1609.82	1537.26	1465.45	1611.4	1564.51	1466.81	1566.76
	mem	21*M*	21*M*	25*M*	25*M*	25*M*	25*M*	25*M*	25*M*	25*M*	25*M*
	ccks	137*M*	137*M*	137*M*	137*M*	137*M*	137*M*	137*M*	137*M*	137*M*	137*M*
	vcks	200*K*	0	259*K*	260*K*	316*K*	307*K*	350*K*	340*K*	378*K*	373*K*
pret-60-75-ext	*cpu*	97.2	80.36	83.44	83.77	76.92	82.14	84.59	80.21	79.12	80.82
	mem	21*M*	21*M*	25*M*	25*M*	25*M*	25*M*	25*M*	25*M*	25*M*	25*M*
	ccks	31*M*	82*M*	36*M*	31*M*	22*M*	22*M*	22*M*	22*M*	22*M*	22*M*
	vcks	74*M*	27*M*	101*M*	99*M*	131*M*	131*M*	159*M*	159*M*	188*M*	188*M*
radar-30-70-4.5-0.95-98	*cpu*	52.04	44.65	44.71	44.83	44.4	44.42	45.06	49.46	44.1	48.54
	mem	207*M*	184*M*	302*M*	302*M*	216*M*	216*M*	231*M*	231*M*	237*M*	238*M*
	ccks	225*K*	551*K*	230*K*	111*K*	230*K*	107*K*	230*K*	105*K*	230*K*	104*K*
	vcks	206*K*	0	206*K*	223*K*	297*K*	269*K*	387*K*	310*K*	478*K*	350*K*
ruler-55-11-a4	*cpu*	996.52	956.13	688.03	749.12	743.59	857.13	766.72	1003.35	771.89	989.01
	mem	33*M*	21*M*	33*M*	33*M*	41*M*	41*M*	49*M*	49*M*	53*M*	53*M*
	ccks	929*M*	1431*M*	684*M*	639*M*	548*M*	511*M*	499*M*	502*M*	468*M*	491*M*
	vcks	1071*M*	0	1314*M*	1314*M*	1951*M*	1914*M*	2448*M*	2483*M*	2902*M*	2963*M*
tsp-25-715-ext	*cpu*	113.69	167.75	171.83	95.18	97.14	89.42	101.26	97.01	103.63	97.53
	mem	29*M*	25*M*	29*M*	29*M*	33*M*	33*M*	37*M*	37*M*	41*M*	41*M*
	ccks	195*M*	458*M*	167*M*	156*M*	138*M*	135*M*	129*M*	131*M*	124*M*	127*M*
	vcks	266*M*	372*M*	270*M*	259*M*	294*M*	291*M*	327*M*	330*M*	361*M*	366*M*

8. Conclusions

References

C. Bessière, J.-C. Régin, R. H. C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2), 2005.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.

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

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

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

C. Likitvivatanavong, Y. Zhang, S. Shannon, J. Bowen, and E. C. Freuder. Arc consistency during seach. In *Proceedings of IJCAI'07*, pages 137–142, 2007.

A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977a.

A. K. Mackworth. On reading sketch maps. In *Proceedings of IJCAI'77*, pages 598–606, 1977b.

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

D. G. Mitchell. Resolution and constraint satisfaction. In *Proceedings of CP-03*, pages 555–569, Kinsale, Ireland, 2003.

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