Handling Heterogeneous Constraints in Revision Ordering Heuristics

Julien Vion and Sylvain Piechowiak

Université de Valenciennes et du Hainaut Cambrésis,
LAMIH CNRS FRE 3304,
59313 Valenciennes Cedex 9, France.
{julien.vion|sylvain.piechowiak}@univ-valenciennes.fr

Abstract. Most constraint solvers use the general AC-5 scheme [17] to handle constraint propagation. AC-5 generalizes the concept of *constraint revision*. Each constraint type can thus be shipped with its own revision algorithm, with various complexities and performances.
Previous papers showed that the order in which constraints are revised have a non-negligible impact on performances of propagation [20,6,1]. However, most of the ideas presented on these papers are based on the use of homogeneous propagators for binary constraints defined in extension. This paper give ideas to handle heterogeneous constraints in a general revision schedule.

1 Introduction

The constraint satisfaction problem (CSP) consists in deciding whether a solution to a discrete constraint network (CN) exists. A CN consists in a set of discrete variables and constraints. Each constraint has one or more variables in its scope, and defines which instantiations of these variables are allowed. A solution to the CN is an instantiation of all variables which satisfies all constraints. The CSP is a standard NP-complete problem and generalizes very naturally many real-life industrial problems such as scheduling, rostering, etc.

Standard techniques for solving CSP instances use interleaved *decision*, *propagation* and *backtrack* steps. Most often, the propagation phase consists in establishing arc consistency by pruning all values that are inconsistent (i.e. cannot appear in any solution of the CSP according to the current decisions) from the point of view of a single given constraint. The resulting search algorithm is called `MAC` (see Page 2). The `AC` function performs the propagation step (see below), and returns **false** if the CN is inconsistent (e.g., one variable domain is empty). $\mathcal{N} = \top$ means that the CN is trivially consistent (e.g., there are no constraints, or all variable domains are singletons). In the given algorithm, δ should not be trivially implied by $\mathcal{N}$, and the final disjunction has a short-circuit behavior. This algorithm is of course very schematic. In particular, users are often interested in obtaining a solution, not simply knowing that one exists. Obtaining a consistent solution from the given algorithm is trivial. The nature of the decisions δ is a major issue in Constraint Programming. Standard all-purpose algorithms

Function MAC($\mathcal{N}$): boolean

```
1 N' ← N ;
2 if ¬AC(N') then return false;
3 if N' = ⊤ then return true;
4 Let δ be some logical decision;
5 return MAC(N'|δ) ∨ MAC(N'|¬δ);
```

usually make variable assignements (reduce the domain of a chosen variable to a singleton). The choice of the variable is important. Common decision heuristics are the *dom/ddeg* or *dom/wdeg* variable assignment ordering heuristics [5].

Removing values that are inconsistent w.r.t. a given constraint is called a *constraint revision* and propagation algorithms are designed to perform such revisions until some fix-point is reached. As constraint revisions are NP-hard in the general case (i.e. without any clue on the semantics of the constraint), for a long time, CSP were limited so as only to involve binary constraints (two variables per constraint). The binary CSP is NP-complete, but AC can be enforced on any binary CN in polynomial space and time.

Dozens of algorithms have been proposed during the last 40 years for performing the propagation step: AC-1 to AC-8, numerous variants of AC-3, etc. Most of these algorithms are actually the combination of a *propagation* and a general *constraint revision* algorithms (either NP-hard or limited to binary constraints). However, two propagation algorithms, AC-5 [17] and GAC-schema [3], consider a generic revision process, and thus generalize most other algorithms. In particular, AC-5 opened enormous perspectives to constraint programming. The main idea of AC-5 is that each constraint in the CN has semantic properties, that can be exploited by a specific (often polynomial) algorithm to perform the constraint revision. Thus, CSP solvers provide a "toolbox" of known useful constraints (*less than*, *not equal*, *sum*, *all different*...), each of which is shipped with its own *propagator*. This scheme gave birth to *global constraints* [2], which permitted to reuse powerful graph theory, artificial intelligence or operational research algorithms for performing constraint revision, and are now of primary importance to handle industrial-class problems. Most modern constraint solvers are based on AC-5.

AC-5, as all AC algorithms since AC-3 [12], are based on a *propagation queue*. When a variable loses a value, constraints involving this variable may no longer be AC, and requires to propagate the removed value to other variables. The propagation queue is used to keep track of such modifications, from which the propagation algorithms deduce which constraint revisions must be performed. The nature of the data stored in the queue (values, constraints and/or variables), as well as the order in which the different constraint revisions are processed, have a significant impact on the performance of the propagation. Works have been dedicated to devise a "good" ordering of the revisions [20,6,1], but they are usually focused on binary CSP with general propagators. Applying these techniques to heterogeneous propagators may lead to trivial pathological cases.

Note: in this paper, we distinguish *generic* from *general* concepts. A *generic* scheme can be *specialized* to the most efficient technique for the sought problem. A *general* algorithm works on any problem without specialization.

The contributions of this paper are :

1. establish a clear state-of-the-art on coarse-grained, generic propagation algorithms (Section 3),
2. define a generic, constraint-based revision ordering heuristic (Section 4.2),
3. survey data structures proposed in the algorithmic literature and show experimentally how they can improve the performance of propagation algorithms (Section 4.3).

2 Background

Definition 1 (Constraint Network, Variable, Domain, Constraint, Instantiation). *A* Constraint Network $\mathcal{N}$ *is a pair* $(\mathscr{X}, \mathscr{C})$ *which consists of :*

- *a set of* n variables $\mathscr{X}$*; a* domain $\mathrm{dom}(X)$ *is attached to each variable* $X \in \mathscr{X}$ *and denotes the finite set of at most* d *values that the variable* X *can be instantiated to, and*
- *a set of* e constraints $\mathscr{C}$*; each constraint* $C \in \mathscr{C}$ *involves at most* k *variables* $\mathrm{vars}(C) \subseteq \mathscr{X}$*; the constraint specifies the allowed* instantiations *for these variables.*

The set of constraints with a given variable X in scope is denoted $\mathrm{ctr}(X)$.

A constraint can be defined in *extension* (i.e., an exhaustive list of allowed or forbidden instantiations), or in *intention* (i.e., using some application $f_C : \prod_{X \in \mathrm{vars}(C)} \mathrm{dom}(X) \rightarrow \mathbb{B}$). *Global constraints* define some property of arbitrary arity that the values of the variables in its scope must verify (e.g., *all different*).

The *Constraint Satisfaction Problem* (CSP) consists in deciding whether a solution to a CN (i.e., an instantiation of all variables satisfying all constraints of the CN) exists. A *constraint check* consists in testing whether a constraint allows a given instantiation of variables. When all constraints can be checked in polynomial space and time, the CSP is NP-complete.

Definition 2 (Arc consistency). *Let* C *be a constraint and* $X \in \mathrm{vars}(C)$*. Value* $v \in \mathrm{dom}(X)$ *is* Arc-Consistent *(AC) w.r.t.* C *iff there exists an instantiation of* $\mathrm{vars}(C)$*, allowed by* C*, which instantiates* X *to* v *(such an instantiation is called a* support *of* v *w.r.t.* C*).* C *is AC iff* $\forall X \in \mathrm{vars}(C)$*,* $\forall v \in \mathrm{dom}(X)$*,* v *is AC.*

$\mathcal{N} = (\mathscr{X}, \mathscr{C})$ *is AC iff* $\forall C \in \mathscr{C}$*,* C *is AC.*

In the literature, the definition of Arc Consistency is often restricted to binary CNs, and the extension of AC to non-binary CNs is called Generalized AC, Hyper-AC, or Domain Consistency. In this paper we refer to Arc Consistency for both binary and non-binary CNs.

Definition 3 (Closure). *Let $\mathcal{N} = (\mathscr{X}, \mathscr{C})$ be a constraint network.* $\mathrm{AC}(\mathcal{N}, C)$ *is the* closure *of $\mathcal{N}$ for AC on C, i.e. the CN obtained from $\mathcal{N}$ where $\forall X \in \mathrm{vars}(C)$, all values $v \in \mathrm{dom}(X)$ that are not AC w.r.t. C have been removed.*

$\mathrm{AC}(\mathcal{N})$ *is the closure of $\mathcal{N}$ for AC, i.e. the CN obtained from $\mathcal{N}$ where $\forall C \in \mathscr{C}$, C have been made AC by closure.*

For any CN $\mathcal{N}(\mathscr{X}, \mathscr{C})$, $\mathrm{AC}(\mathcal{N}, C)$ for any $C \in \mathscr{C}$ and $\mathrm{AC}(\mathcal{N})$ are unique. In the general case, computing the closure for AC on a CN is NP-hard. Optimal algorithms such as GAC-schema are in $O(ekd^k)$ [3].

Definition 4 (Propagator). *Given a CN $\mathcal{N} = (\mathscr{X}, \mathscr{C})$, the* propagator *for a given constraint $C \in \mathscr{C}$ is the algorithm that computes* $\mathrm{AC}(\mathcal{N}, C)$.

3 A generic, coarse-grained propagation algorithm

This section does not intend to bring out innovative propagation algorithms, but instead aims to establish a clear state-of-the-art of coarse-grained, generic propagation techniques.

The main difference between AC algorithms lies in the way the general constraint propagator works. However, independently of the general propagator, these algorithms are often sorted in two families, depending on the nature of the data stored into the propagation queue. So-called *fine-grained* algorithms store every single *value* that have been removed from the domain of the different variables in the propagation queue, and try to exploit this information to avoid unecessary work. *Coarse-grained* algorithms only store the *variable* where a value have been removed, and/or constraints involving them, that thus must be revised. Although using fine-grained propagation queues is essential in designing optimal algorithms, the theoretical difference is at best marginal (the coarse-grained GAC-2001 algorithm is in $O(ek^2d^k)$ [4]), and the simpler data structures used by coarse-grained algorithms usually make them as much efficient in practice.

Mackworth's original AC-3 algorithm [12] was *arc-oriented*, that is, the propagation queue was composed of (*Variable*, *Constraint*) pairs. The *Variable* part of the pair identifies a variable which is not guaranteed to be AC w.r.t. the *Constraint*. McGregor showed in [13] that a similar behavior could be obtained by simply storing the *modified variables* in the queue. However, when working with non-binary constraints, variable-oriented propagation is not informative enough to avoid all useless revisions: when two variables involving the same non-binary constraint are in the queue, the domain of each variable involved by the constraint should be controlled for arc-consistency only once.

Boussemart *et al.* proposed in [6] to introduce an auxiliary data structure we call *modified*$[C]$ to emulate the benefits of an arc-oriented propagation scheme in a variable-oriented propagation algorithm. It is used to keep track of which variables have been actually modified since the last revision of a constraint. Interestingly enough, this auxiliary data structure can also be used to devise a purely constraint-oriented propagation algorithm which avoids these useless

Algorithm 1: AC-5$^{\text{v}}(\mathcal{N} = (\mathscr{X}, \mathscr{C}))$: CN

```
1  Q ← 𝒳;
2  foreach C ∈ 𝒞 do modified[C] ← vars(C);
3  while Q ≠ ∅ do
4      Pick X from Q;
5      foreach C ∈ ctr(X) s.t. modified[C] ≠ ∅ do
6          Δ ← C.revise(modified[C]) ;
7          if Δ = ⊥ then return false ;
8          Q ← Q ∪ Δ;
9          modified[C] ← ∅;
10         foreach Y ∈ Δ do
11             foreach C' ∈ ctr(Y)\C do
12                 modified[C'] ← modified[C'] ∪ {Y};
13 return true;
```

revisions. The version presented here is slightly optimized (with $O(k)$ overhead in Algorithm 3 against $O(k^2)$ in the original version).

The original AC-5 algorithm was fine-grained, so we propose our coarse-grained variants.

3.1 Variable-oriented propagation

In AC-5$^{\text{v}}$ (Algorithm 1), the propagation queue Q contains recently modified variables, which require the revision of the constraints involving them (loop starting on Line 5). Initially, all variables are put in Q, however, when using the `MAC` procedure, only variables involved by the decisions δ are concerned.

The call to C.`revise`($modified[C]$) on Line 6 calls C's propagator, which may remove values from vars(C). The propagator returns a set $\Delta \subseteq \text{vars}(C)$ of modified variables,[1] or $\bot$ if an inconsistency has been detected (e.g., the domain of a variable has been emptied).

3.2 Constraint-oriented propagation

In this variant, called AC-5$^{\text{c}}$ (Algorithm 2), constraints yet to be revised are stored in the queue. This leads to a somewhat simpler algorithm and finer queue, but the *modified* data structure is even more important to avoid unecessary work: when a constraint C is put in the queue due to some removals in the domain of a variable X involved by C, C's propagator only needs to control the domains of the *other* variables for arc consistency.

Algorithm 2: AC-5^{c}($\mathcal{N} = (\mathcal{X}, \mathcal{C})$) : CN

```
1  Q ← C;
2  foreach C ∈ C do  modified[C] ← vars(C);
3  while Q ≠ ∅ do
4      Pick C from Q;
5      Δ ← C.revise(modified[C]) ;
6      if Δ = ⊥ then return false ;
7      modified[C] ← ∅;
8      foreach Y ∈ Δ do
9          foreach C' ∈ ctr(Y)\C do
10             Q ← Q ∪ {C'};
11             modified[C'] ← modified[C'] ∪ {Y};
12 return true;
```

Algorithm 3: revise^{rm}(*modified*: {Variable}): {Variable}

```
1  Δ ← ∅;
2  foreach X ∈ vars(this) s.t. modified ≠ {X} do
3      foreach v ∈ dom(X) s.t. this.res[X][v] is not valid do
4          τ ← this.findSupport(X, v) ;
5          if τ = ⊥ then
6              remove v from dom(X);
7              if dom(X) = ∅ then return ⊥;
8              Δ ← Δ ∪ {X};
9          else
10             foreach Y ∈ vars(this) do  this.res[Y][τ[Y]] ← τ;
11 return Δ;
```

3.3 The AC-3^{rm} propagator

To illustrate the use of our AC-5^{v}/AC-5^{c} scheme, we give a sample general propagator, called `revise`^{rm}, extracted from the AC-3^{rm} algorithm [9] and extended to handle non-binary constraints (Algorithm 3). *this* denotes the current constraint. τ is a tuple containing a value, denoted $\tau[X]$, for every variable $X \in \text{vars}(C)$. τ is said to be *valid* iff $\forall X \in \text{vars}(C), \tau[X] \in \text{dom}(X)$. The `findSupport` method seeks for an allowed, valid tuple supporting the given value for the current constraint, and returns $\perp$ if no such tuple can be found. If a support is found, it is recorded as a *residue* [11], exploiting the *multidirectionality* of the constraints. A most interesting feature of residues is that they are *stable on backtrack*, that is, when using the MAC procedure, residues that are found at some point of the search tree will also be valid after a backtrack. No update of the data structures is thus necessary upon backtracking.

[1] Many solvers use events to avoid the management of Δ sets.

revise^{rm} may be considered as the state-of-the-art algorithm to propagate, within MAC and using coarse-grained propagation queues, constraints defined in extension.[2] It can be used as a "fallback" propagator when no better algorithm exists or is implemented yet. Many efficient propagators may also be built on this algorithm, simply by specializing the `findSupport` method: although the standard behavior consists in iterating over all the $O(d^{k-1})$ valid tuples, checking the constraint (in $O(k)$) until an allowed tuple is found, better methods may be devised when working with known constraints. For example, with the $X = Y + Z$ constraint, a support for a value $x \in \text{dom}(X)$ can be found in $O(d)$: the algorithm iterates over the values $y \in \text{dom}(Y)$, and checks whether the value $z = x - y \in \text{dom}(Z)$.

4 Managing the propagation queue

Note: In this paper, all heuristics and sorting algorithms are min-based (minimum value first). Of course, it is perfectly feasible to reverse all comparisons to obtain max-based heuristics and sortings, without any impact on the algorithms and complexities.

4.1 Related work on ordering heuristics

The order in which the constraints are revised has an important impact on the performance of the propagation, and several works have been devoted to devise a good heuristic to know which constraint to propagate first. The original work is by Wallace & Freuder [20]. In their work, they study the impact of various ordering heuristics in an arc-oriented AC-3 propagation algorithm, restricted to binary CSPs. The heuristics devised by Wallace & Freuder follow this principle: for an efficient propagation, values should be filtered as soon as possible, so most constraining constraints should be propagated first.

Of course, it is very difficult to predict how strong a constraint is beforehand. Wallace & Freuder use the tightness (proportion of instantiations forbidden by the constraint) as an heuristic to estimate the strength of the constraint, which is reasonable when working on small binary CSP. However, computing the tightness of a general constraint is $\#P$-hard. Proposed less time-consuming alternatives consider the domain size (we call this the *dom* heuristic) or the degree of the variable in the arc. Note that even when working on tiny binary CSP, Wallace & Freuder's best results were obtained by applying an heuristic before the first propagation (using a pigeonhole sort algorithm), and rely on simple queues or stacks afterwards. An interesting alternative, proposed by Balafoutis & Stergiou in [1], is to exploit the constraint weights obtained from the *dom/wdeg* variable assignment heuristic [5] to devise the most interesting constraints. Moreover, this strategy seems to interact positively with the variable assignment heuristic. Both Wallace & Freuder and Balafoutis & Stergiou works are primarily oriented towards binary CSPs, using plain AC-3 for propagation.

[2] For binary constraints, one can refer to [10] for the revise^{bit} propagator.

Boussemart *et al.* study and experiment in [6] different revision ordering strategies, using either arc, variable or constraint-oriented propagation queues. As in previously cited works, Boussemart *et al.* perform their experiments on binary CSPs and use an homogeneous propagation algorithm, an improved variant of AC-3. The main result of their work is that the best variant in this context is the variable-oriented propagation scheme with the *dom* variable revision ordering heuristic, a result quite close to Wallace & Freuder's. Indeed, although constraint or arc-oriented revision ordering heuristics (using the product of the size of the domains of the variables in the scope of a constraint, an heuristic we call $\Pi\, dom$) successfully reduces the number of constraint checks compared to variable-oriented heuristics, they require a high overhead to compute the heuristics. However, the data structures used by Boussemart *et al.* can be greatly improved.

Another work of interest is [15] by Schulte & Stuckey. The authors explain the propagation scheme implemented at the core of the Gecode solver [14]. The technique is based on another folklore knowledge: since we cannot predict whether a constraint will filter values or not, let us minimize lost time by propagating the fastest constraints first. This technique may only be used with an arc or constraint-based propagation queue. An small integer identifier is associated to each constraint: 0 for very fast constraints (i.e., constant-time or $O(k)$ propagators), 1 for fast constraints (i.e., $O(d)$ propagators), up to 7 for the slowest constraints (i.e., NP-hard propagators). The propagation queue is divided in 8 FIFOs, and the integer identifies the queue in which the constraint is assigned. When picking a constraint for revision, the first FIFO is polled first, then the second if the first is empty, and so on. Moreover, Schulte & Stuckey propose to adapt the identifier dynamically, as even a NP-hard propagator can be applied quite quickly if most variables in the scope of the constraint are assigned.

Interestingly enough, the most successful ordering heuristics devised by Wallace & Freuder or Boussemart *et al.* (dom or $\Pi\, dom$) also cover the "fastest constraints first" principle: the AC-3-based propagations algorithms used in their experiments use propagators whose time complexities are highly correlated with the size of the domains.

4.2 Fine, constraint-based revision ordering heuristics

Firstly, we give a simple example showing the limits of the variable-based propagation scheme when the CSP include large arity constraints. Let $\mathcal{N}$ be a CSP with n variables X_1 to X_n, $\mathrm{dom}(X_i) = \{1, \ldots, n\}$, and the constraints $X_i \leq X_{i+1}$ $\forall i \in \{1, \ldots, n-1\}$ and alldifferent$(X_1, \ldots, X_n)$. The *alldifferent* constraint is implemented using an easy algorithm, which filters out all values present in singleton domains, and checks whether $\left|\bigcup_{X \in \mathrm{vars}(C)} \mathrm{dom}(X)\right| \leq |\mathrm{vars}(C)|$. This propagator does not establish (G)AC but is idempotent, detects trivial pigeonhole cases and has a quite low complexity ($O(kd)$).

Let us remove the lowest value from the domain of X_1 and propagate. Using a variable-based propagation scheme with any heuristic, or a constraint-based

Constraint	Evaluator
$X\{<,\leq,>,\geq,\neq\}Y$	2
$\bigvee(\dots)$	$\log_2(\lvert\text{vars}(C)\rvert)$
$\sum_{X\in\text{vars}(C)} X \leq k$	$\lvert\text{vars}(C)\rvert$
alldifferent$(\dots)$	$\lvert\text{vars}(C)\rvert^2$
$a \times X + b = Y$	$\min(\lvert\text{dom}(X)\rvert, \lvert\text{dom}(Y)\rvert)$
$X = Y\{+,\times\}Z$	$\lvert\text{dom}(X)\rvert\,\lvert\text{dom}(Y)\rvert + \lvert\text{dom}(X)\rvert\,\lvert\text{dom}(Z)\rvert + \lvert\text{dom}(Y)\rvert\,\lvert\text{dom}(Z)\rvert$
$X \iff C(\dots)$	$\mathit{evaluator}(C) + \mathit{evaluator}(\neg C)$
positive table	table size $\times\ \lvert\text{vars}(C)\rvert$
revise$^{\text{rm}}$	$\prod_{X\in\text{vars}(C)} \lvert\text{dom}(X)\rvert$
revise$^{\text{bit}}$ [10]	$\lvert\text{dom}(X)\rvert\,\lvert\text{dom}(Y)\rvert \div 10$

Table 1. Evaluators for various constraints.

propagation scheme with simple FIFO behavior, the propagator for $X_1 \leq X_2$ is called, removing the lowest value from X_2, then the propagator for *alldifferent*, then the propagator for $X_2 \leq X_3$, then *alldifferent* again, etc. With a constraint-based propagation scheme and a simple heuristic that prioritizes the stronger and faster $\leq$ constraints over *alldifferent*, the propagator for *alldifferent* would be called only once, hence a much faster propagation.

As a reference, our implementation requires 4 s to propagate the above scenario with $n = 1{,}000$ using a variable-based propagation scheme and 50 ms with a prioritized constraint-based propagation scheme.

We define a constraint-based heuristic as follows: each constraint type must implement an *evaluator*, i.e., a method that returns a float number. The number gives an estimation of the time required to propagate the constraint. In our implementation, we use either the average-case complexity if available, or the worst-case complexities of the propagators to compute the estimation. For the general-purpose revise$^{\text{rm}}$ propagator, we fallback to the $\Pi\,dom$ heuristic. Table 1 summarizes the evaluators we use for the various constraints implemented in our constraint solver. In the remaining of this paper, we will call this constraint revision ordering heuristic *eval*.

We combine our scheme with the ideas from Balafoutis & Stergiou [1], by dividing the value computed by the evaluator by the constraint weight, leading to the so-called $eval/w$ constraint revision ordering heuristic.

4.3 Data structures for priority queues: a survey

When using a heuristic for extracting the variable/constraint with the highest/lowest score, the queue is basically a *priority queue*. Various data structures have been proposed in the algorithmic literature for handling these. This section surveys a few of them.

AC-5$^{\text{v}}$ and AC-5$^{\text{c}}$ require two basic operations: inserting an object in the priority queue, and extracting the "best" object, that minimizes the score computed by some heuristic. Upon insertion, if the object is already in the queue, its

Data structure	Insert	Update	Remove min	Heuristics	Plot
m linked lists	$O(1)$	$O(1)$	$O(m)$	FIFO + m levels priority	
Bit vector	$O(1)$	$O(1)$	$\Theta(\lambda)$	Any	
Binary heap	$O(\log \lambda)$	$O(\log \lambda)$	$O(\log \lambda)$	Any	
Binomial heap	$O(1)$*	$O(\log \lambda)$	$O(\log \lambda)$	Any	
Fibonacci heap	$O(1)$	$O(1)$*	$O(\log \lambda)$*	Any	
Soft heap	$O(1)$*	N/A	$O(1)$*	Any (approximated)	

Table 2. Various data structures for implementing priority queues. * denotes an amortized complexity.

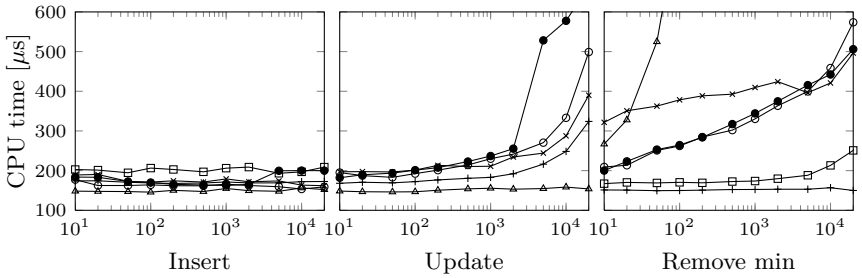

Fig. 1. Actual performance of our implementations: time to insert, update or remove minimum in a set of λ elements. Notice the semilog scale: straight plots are actually log-like.

position is updated. All heuristics devised so far may only evolve during propagation when a variable domain is modified. Thus, it is perfectly sound to update the heuristic score of a constraint only when such an event occur.

In order to experiment the various data structures, all queues implement the generic `Queue` interface as defined in the Java 1.6 API. All queues must be backed by a Set implementation in order to support the Update operation and preventing the same object to be inserted twice.

Here follows the list of data structures we implemented and experimented. Table 2 give the worst-case time complexities for both three basic operations on a structure containing λ elements. Space complexity is $\Theta(\lambda)$ for all structures except multiple linked lists which are in $\Theta(m + \lambda)$. Figure 1 shows the performance of our implementations. We implemented the data structures in Java and benchmarked them using Sun's Java 1.6u21 64-bit HotSpot Virtual Machine for Linux, running on a Intel Core 2 Duo processor @ 2.53 GHz. The JiP 1.2 profiler [21] was used to measure the performance of the various operations, using the following experimental protocol: λ random integer values are inserted in the priority queue. Then the three following operations are performed 1,000,000 times: an additional random integer value is inserted, the minimum integer is extracted from the queue, then an element is randomly updated. The profiler is

used to measure the time consumed by each of these three operations. Reported times are for inserting/updating/removing one single element.

Linked lists are the most common way to implement queues in propagation algorithms. Actually, FIFO queues are in the core of most solvers.[3] However, linked lists are not designed to be sorted, and picking the smallest element requires to parse all elements. Using multiple linked lists is an efficient way to implement very coarse heuristics. The heuristic computes an small integer number that identifies a FIFO queue in which the variable/constraint is stored. Our benchmarks used 8 FIFO linked lists. To try to emulate the behavior described in [15], the appropriate FIFO is chosen based on the result of the operation $\lfloor \log_3(h) \rfloor$ (h is the score computed by the heuristic). Indeed, the main factor Schulte & Stuckey use to choose the appropriate FIFO is the arity of the constraint, and the score h computed by the traditional $\Pi\, dom$ heuristic is in $O(d^k)$. The base 3 was chosen in order to normalize the use of all 8 FIFOs in the average case. When an update is requested, the variable/constraint is moved to the tail of the appropriate FIFO if needed.

Bit vectors can replace linked lists when static or heuristic ordering is used. One bit is associated to each variable or constraint. The bit is set upon insertion, and cleared upon removal. This ensures very fast insertion and update operations, but finding the minimal element still requires to parse all elements. Boussemart *et al.* used this scheme in their paper [6].

Binary heap is a well known data structure, and can be used for implementing priority queues. A binary heap is a naturally balanced binary tree in which each parent node is smaller than its children. The smallest element is thus always at the root of the tree. Inserting, updating and removing an element requires $O(\log \lambda)$ *sift* operations to maintain the heap property.

Binomial heap [19] use a special tree structure (a "forest" of heap trees) to achieve fast insertions, although removing the smallest element has the same performance as insertion. Updates are basically performed by removing and reinserting the element.

Fibonacci heap [7] is a "lazy" variant of binomial heaps, in which most computations are delayed until the remove min operation is called. Insertions and updates are thus very fast, although the remove min operation is not much slower than with binomial heaps, and even more robust for large amounts of data.

Soft heap [8] is a variant of binary heap in which the heap property is only maintained on the top of the tree. The root of the tree is thus no longer guaranteed to be the smallest element. However, the "corruption" is minimal and can be parameterized (we used $\epsilon = 10\,\%$). This permits constant $O(\log \frac{1}{\epsilon})$ complexities for both insert and remove min operations. However, insertion is noticeably slower than with other data structures, and update is not supported by our implementation (the element is simply left in place upon updating).

[3] Simple experiments show that LIFO strategies are almost always worse than FIFO.

			AC-5^v			AC-5^c		
	n	e	Inserts	Updates	Remvs	Inserts	Updates	Remvs
bqwh-18-141-0-ext	141	879	4.0 M	647 k	2.2 M	21 M	4.3 M	14 M
bqwh-18-141-47-glb	141	36	29 M	6.2 M	15 M	28 M	7.7 M	20 M
frb40-19-1	40	410	2.3 M	1.4 M	1.3 M	19 M	16 M	14 M
series-18	69	36	3.1 M	963 k	2.5 M	3.5 M	2.1 M	3.3 M
ruler-44-9-a3	45	74	2.9 M	1.5 M	2.0 M	4.5 M	6.0 M	3.3 M
langford-3-13	65	27	11 M	6.0 M	8.3 M	9.0 M	15 M	7.4 M
bmc-ibm-02-02	50 k	48 k	91 k	23 k	91 k	94 k	23 k	94 k
crossword-m1-lex-15-04	4.4 k	7.9 k	12 M	569 k	11 M	167 M	8.0 M	167 M
lemma-24-3	552	924	20 M	0	14 M	57 M	2.8 M	47 M
os-taillard-5-100-3	625	500	66 M	6.1 M	59 M	125 M	15 M	103 M
scen4	8.3 k	7.6 k	55 k	24 k	53 k	95 k	47 k	92 k
bigleq-70	70	70	18 M	16 M	10 M	30 M	36 M	19 M

Table 3. Number of operations required to solve various problems with the *eval* heuristic.

Choosing the best data structure may depend on the number of elements it will contain, as well on the relative importance of the insert, update and remove operations. As a reference, the problems used as benchmarks during the CPAI'08 Itl Solver Competition [16] had on average 863 explicit variables (from 2 to 62,704, std dev is 3,100, median 120) and 5,129 explicit constraints (from 1 to 546,105, std dev is 20,065, median 458). Moreover, using techniques such as constraint decomposition, symmetry breaking, implicit constraint detection, second-order consistencies or nogood learning can increase the number of variables and/or constraints significatively.

Table 3 gives an idea of the relative number of operations required to solve some well-known benchmark problems using the *eval* revision heuristic and the *dom/ddeg* decision heuristic (the more efficient *dom/wdeg* heuristic was not used to avoid any interference with the revision heuristic). The second and third columns, n and e, respectively show the number of variables and constraints actually present in the problem once the solver has performed appropriate decompositions. The number of removals is usually less than the number of inserts because the propagation is interrupted (and the priority queues cleared) when an inconsistency is encountered.

4.4 Note on Set implementation

Several data structures can be used to implement sets: hashtables, ordered trees, etc. For best, constant-time performance, we rely on simple arrays. An contiguous integer identifier is associated to each object upon creation, which identifies the index of the array where the structures will be stored. A basic set implementation can thus use an array of booleans (or a bit vector).

Clearing the sets is also an operation that can have a non negligible impact on the performances of the resolution. In some of our experiments, an $O(\lambda)$

	Bit vector	8 FIFOs	Bin Heap	Binom H	Fib Heap	Soft Heap
bqwh-18-141-0-ext	20.0 s	11.6 s	26.3 s	11.4 s	12.0 s	14.8 s
bqwh-18-141-47-glb	35.4 s	32.6 s	35.6 s	34.3 s	34.8 s	35.4 s
frb40-19-1	27.8 s	11.8 s	23.0 s	11.0 s	12.4 s	13.7 s
series-18	5.7 s	5.0 s	0.4 s	4.8 s	4.9 s	4.9 s
ruler-44-9-a3	8.3 s	7.2 s	27.1 s	6.8 s	7.0 s	6.8 s
langford-3-13	15.1 s	13.7 s	1.9 s	14.2 s	14.3 s	13.7 s
bmc-ibm-02-02	339.0 s	20.2 s	20.3 s	20.5 s	20.8 s	20.0 s
crossword-m1-lex-15-04	2,204.3 s	210.1 s	433.0 s	265.0 s	273.8 s	283.0 s
lemma-24-3	85.0 s	74.4 s	113.0 s	82.9 s	87.8 s	96.2 s
os-taillard-5-100-3	1,579.6 s	205.7 s	113.3 s	188.9 s	199.0 s	291.7 s
scen4	11.3 s	6.2 s	7.1 s	6.4 s	6.7 s	6.9 s
bigleq-70	318.9 s	45.8 s	50.3 s	42.9 s	43.8 s	42.0 s

Table 4. Time to solve the problems using AC-5^c and *eval* heuristic with various priority queues.

clear operation could take more than 90 % of the CPU time required to solve the problem! Set clearing can be performed in $O(1)$ by using integer counter, as proposed in [6] for the *modified* data structure. An integer number i is associated to the set, and is incremented when clearing is requested. When an object O is put in the set, the number $i_O = i$ is stored in the structure representing the object. The object is considered to be present in the set iff $i_O = i$.

5 Experiments

These experiments are performed in the same conditions as before (Sun's Java 1.6u21 64-bit HotSpot Virtual Machine for Linux, running on a Intel Core 2 Duo processor @ 2.53 GHz), but without the use of a profiler. The constraint solver used is CSP4J [18]. We selected representative problem instances from the CPAI'08 competition, that could be solved between 2 and 300 s using the *dom/ddeg* decision heuristic. Our experiments are still preliminar: our solver only implements a few constraint types, and the problem base of CPAI'08 lacks challenging problems with global constraints, which reduces the "heterogeneity" of the selected problems. In particular, few of them use global constraints at all.

A first set of experiments, summarized on Table 4, compares the different data structures using the plain *eval* constraint revision ordering heuristic. These results tend to show that either multiple FIFOs or Binomial heaps are the most efficient data structures for handling constraint revision ordering heuristics (and that linear-time data structures such as bit vectors definitively are not, despite their very fast insert and update operations).

Finally, Table 5 compares the different variable- and constraint-based heuristics. AC-5^v uses a Binomial heap in these experiments. The *dom/ddeg* decision heuristic was used, but constraint weights are still computed as for the *dom/wdeg* decision heuristic. These weights can thus be used for the *dom/wdeg* variable

	AC-5^v		AC-5^c/8 FIFOs				AC-5^c/Binomial heap			
	dom	$\frac{dom}{wdeg}$	$\Pi\, dom$	$\frac{\Pi\, dom}{w}$	$eval$	$\frac{eval}{w}$	$\Pi\, dom$	$\frac{\Pi\, dom}{w}$	$eval$	$\frac{eval}{w}$
bqwh-18-141-0-ext	9.0	9.2	8.9	11.7	10.8	10.9	10.4	11.4	10.5	11.5
bqwh-18-141-47-glb	32.5	32.3	33.8	33.8	33.4	33.7	35.1	33.8	36.2	33.0
frb40-19-1	7.8	8.7	9.7	12.7	12.0	13.2	12.1	14.2	12.2	14.2
series-18	4.9	5.3	5.2	5.1	5.0	5.2	5.1	5.0	5.5	5.5
ruler-44-9-a3	8.9	11.2	7.6	9.1	8.4	10.2	7.2	7.3	7.3	7.8
langford-3-13	17.1	18.1	15.3	15.3	14.4	16.2	15.4	15.9	14.6	16.7
bmc-ibm-02-02	19.6	20.5	19.8	20.0	19.7	19.1	19.6	19.5	19.3	19.2
crosswd-m1-lex-15-04	152.0	139.3	190.6	175.2	220.8	190.9	219.5	187.0	262.6	204.4
lemma-24-3	80.2	85.6	97.7	96.0	91.8	88.6	86.8	89.1	87.0	85.9
os-taillard-5-100-3	206.4	224.1	249.5	190.4	212.6	232.0	257.1	197.9	211.4	205.1
scen4	7.0	6.7	7.4	7.7	7.8	6.4	6.2	6.7	7.0	6.6
bigleq-70	92.4	100.9	34.5	34.5	28.2	97.8	30.8	31.4	27.6	91.2

Table 5. Time (in seconds) to solve the problems with the $dom/ddeg$ decision heuristic and different revision ordering heuristics.

and for $\Pi\, dom/w$ or $eval/w$ constraint revision ordering heuristics. Following Balafoutis & Stergiou results described in [1], these ordering heuristics are more senseful when combined with the $dom/wdeg$ decision heuristic.

Although we are aware that these experiments still fail to demonstrate a clear superiority of constraint-based heuristics, we are convinced that (1) constraint-based propagation is actually competitive w.r.t. variable-based propagation, (2) it successfully avoids pathological cases (our *bigleq* problem), and (3) opens a new field of research to devise better heuristics.

6 Conclusion & Perspectives

In this paper, we devised AC-5^v and AC-5^c, generic coarse-grained propagation algorithms using respectively variable- and constraint-based propagation queues. After recalling why the management of the propagation queue is important, we surveyed a few data structures that can be used to control the order in which variables or constraints will be revised.

We proposed a new, generic way to control the order in which the constraints are revised using the constraint-based propagation scheme, and showed experimentally that using clever data structures, this way of controlling the propagation can be competitive w.r.t. variable-based propagation, and can avoid pathological cases. These cases will occur frequently when using heavy global constraints, such as NP-hard constraints introduced by Lazy Clause Generation or algorithm hybridization. Variable-based propagation will then no longer be a viable alternative.

Although our heuristics are still not clearly better than standard general heuristics, we hope to open the perspectives to devise new techniques, either adaptative or by taking into account the strength of the constraints.

References

1. T. Balafoutis and K. Stergiou. Exploiting Constraint Weights for Revision Ordering in Arc Consistency Algorithms. In *Proceedings of the ECAI-2008 workshop on Modeling and Solving Problems with Constraints*, 2008.
2. N. Beldiceanu and E. Contejean. Introducing Global Constraints in CHIP. *Mathl. Comput. Modelling*, 20(12):97–123, 1994.
3. C. Bessière and J.-C. Régin. Arc Consistency for General Constraint Networks: Preliminary Results. In *Proceedings of IJCAI'97*, pages 398–404, 1997.
4. C. Bessière, J.-C. Régin, R.H.C. Yap, and Y. Zhang. An Optimal Coarse-Grained Arc Consistency Algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.
5. F. Boussemart, F. Hemery, C. Lecoutre, and L. Saïs. Boosting Systematic Search by Weighting Constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.
6. F. Boussemart, F. Hemery, and C. Lecoutre. Revision Ordering Heuristics for the Constraint Satisfaction Problem. In *Proceedings of CPAI'04 workshop held with CP'04*, pages 29–43, 2004.
7. M.L. Fredman and R.E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM*, 34(3):596–615, 1987.
8. H. Kaplan and U. Zwick. A Simpler Implementation and Analysis of Chazelle's Soft Heaps. In *Proc. of the 19th ACM-SIAM Symposium on Discrete Algorithms*, pages 477–485, 2009.
9. C. Lecoutre and F. Hemery. A Study of Residual Supports in Arc Consistency. In *Proceedings of IJCAI'2007*, pages 125–130, 2007.
10. C. Lecoutre and J. Vion. Enforcing Arc Consistency using Bitwise Operations. *Constraint Programming Letters*, 2:21–35, 2008.
11. C. Likitvivatanavong, Y. Zhang, J. Bowen, and E.C. Freuder. Arc Consistency in MAC: a New Perspective. In *Proceedings of CPAI'04 workshop held with CP'04*, pages 93–107, 2004.
12. A.K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.
13. J.J. McGregor. Relational Consistency Algorithms and their Application in Finding Subgraph and Graph Isomorphisms. *Information Sciences*, 19:229–250, 1979.
14. C. Schulte et al. Generic Constraint Development Environment (Gecode). `http://www.gecode.org/`, 2005-2009.
15. C. Schulte and P.J. Stuckey. Efficient Constraint Propagation Engines. *ACM Transactions on Programming Languages and Systems*, 31(1):1–43, 2008.
16. M. van Dongen, C. Lecoutre, and O. Roussel. Third International CSP Solvers Competition. `http://www.cril.univ-artois.fr/CPAI08`, 2008.
17. P. van Hentenryck, Y. Deville, and CM. Teng. A Generic Arc-Consistency Algorithm and its Specializations. *Artificial Intelligence*, 57:291–321, 1992.
18. J. Vion. Constraint Satisfaction Problem for Java. http://cspfj.sourceforge.net/, 2006.
19. J. Vuillemin. A Data Structure for Manipulating Priority Queues. *Communications of the ACM*, 21:309–314, 1978.
20. R.J. Wallace and E.C. Freuder. Ordering Heuristics for Arc Consistency Algorithms. In *Proceedings of NCCAI'92*, pages 163–169, 1992.
21. A. Wilcox and P. Hudson. Java Interactive Profiler. `http://jiprof.sourceforge.net/`, 2005–2010.