An Optimal Coarse-grained Arc Consistency Algorithm * †

Christian Bessiere
LIRMM-CNRS (UMR 5506)
161 rue Ada,
34392 Montpellier Cedex 5, France

Jean-Charles Régin
ILOG
1681 route des Dolines,
06560 Valbonne, France

Roland H.C. Yap
School of Computing,
Natl. Univ. of Singapore,
3 Science Dr. 2, Singapore

Yuanlin Zhang
Department of
Computer Science,
Texas Tech University, USA

Abstract

The use of constraint propagation is the main feature of any constraint solver. It is thus of prime importance to manage the propagation in an efficient and effective fashion. There are two classes of propagation algorithms for general constraints: fine-grained algorithms where the removal of a value for a variable will be propagated to the corresponding values for other variables, and coarse-grained algorithms where the removal of a value will be propagated to the related variables. One big advantage of coarse-grained algorithms, like AC-3, over fine-grained algorithms, like AC-4, is the ease of integration when implementing an algorithm in a constraint solver. However, fine-grained algorithms usually have optimal worst case time complexity while coarse-grained algorithms don't. For example, AC-3 is an algorithm with non-optimal worst case complexity although it is simple, efficient in practice, and widely used. In this paper we propose a coarse-grained algorithm, AC2001/3.1, that is worst case optimal and preserves as much as possible the ease of its integration into a solver (no heavy data structure to be maintained during search). Experimental results show that AC2001/3.1 is competitive with the best fine-grained algorithms such as AC-6. The idea behind the new algorithm can immediately be applied to obtain a path consistency algorithm that has the best-known time and space complexity. The same idea is then extended to non-binary constraints.

*Preliminary versions of this paper appeared in [BR01, ZY01].

†During that work, Christian Bessiere was supported by ILOG under a research collaboration contract ILOG/CNRS/University of Montpelier II, Yuanlin Zhang by the Science Foundation Ireland under Grant 00/PI.1/C075, and Roland Yap and Yuanlin Zhang by the Academic Research Fund, National Univ. of Singapore.

1 Introduction

Constraint propagation is a basic operation in constraint programming. It is now well-recognized that its extensive use is necessary when we want to efficiently solve hard constraint satisfaction problems. All the constraint solvers use propagation as a basic step. Thus, each improvement to a constraint propagation algorithm has an immediate effect on the performance of the constraint solving engine. In practical applications, many constraints are of well-known types for which specific algorithms are available. These algorithms generally receive a set of removed values for one of the variables involved in the constraint, and propagate these deletions to the other variables of the constraint. They are usually as cheap as one can expect in cpu time. This state of things implies that most of the existing solving engines are based on a constraint-oriented propagation scheme (ILOG Solver, CHOCO, etc.). We call the algorithms using this scheme *coarse-grained* algorithms. AC-3 [Mac77a, McG79] is a generic constraint propagation algorithm which fits the best this propagation scheme. Its successors, AC-4, AC-6, and AC-7, indeed, were written with a value-oriented propagation where the deletion of a value in the domain of a variable will be propagated only to the affected values in the domains of other variables. Algorithms using this propagation are called *fine-grained* algorithms here. The coarse-grained characteristics of AC-3 explain why it is the algorithm which is usually used to propagate those constraints for which nothing special is known about the semantics (and then for which no specific algorithm is available). When compared to AC-4, AC-6 or AC-7, this algorithm has a second strong advantage, namely, its independence with respect to specific data structure which should be maintained if used during a search procedure. Thus, it has ease of implementation. Fine-grained algorithms, on the other hand, have more complex implementation with possibly higher overheads as there is a need to maintain some complex data structures.

Unfortunately, the worst case time complexity of AC-3 is $\mathcal{O}(ed^3)$, where e is the number of constraints and d is the size of the maximum domain in a problem. Fine grained algorithms on the other hand enjoy optimal worst case complexity $\mathcal{O}(ed^2)$ [MH86]. The fine-grained algorithms are also more efficient when applied to networks where much propagation occurs [BFR95, BFR99] while AC-3 is better when there is very little propagation.

In this paper, we present a new algorithm, AC2001/3.1, which is the first worst case optimal coarse-grained arc consistency algorithm. This result is somewhat surprising since due to the non-optimality result of AC-3 [MF85] from 1985, it is widely held that only fine-grained algorithms have worst case optimality. AC2001/3.1preserves the simplicity of AC-3 while improving on AC-3 in efficiency both in terms of constraint checks and in terms of cpu time. In our experiments, AC2001/3.1 leads to substantial gains over AC-3 both on randomly generated and real-world instances of problems and comparable to AC-6, the fastest fine-grained algorithm.

The idea behind the new algorithm can be applied immediately to obtain a new simple path consistency algorithm, PC2001/3.1, which has the same time and space complexity as the best known theoretical results. We show how to use the same idea for arc consistency on non-binary constraints with a new algorithm, GAC2001/3.1. We also give a detailed comparison of coarse-grained and fine-grained algorithms.

The paper is organized as follows. The preliminaries are given in section 2 before the

presentation of AC2001/3.1 and its complexity analysis in section 3. Section 5 extends the idea to path consistency and and generalised arc consistency. Experimental results to benchmarking the performance of the new algorithm with respect to AC-3 and AC-6 are shown in section 4. We compare and contrast most propagation algorithms in section 6 before concluding in section 7.

2 Preliminaries

In this section we give some background material and notations used herein.

Definition 1 *A finite binary* constraint network *(N, D, C) consists of a finite set of variables $N = \{x_1, x_2, \cdots, x_n\}$, a set of domains $D = \{D_1, D_2, \cdots, D_n\}$, where the domain D_i ($i \in 1..n$) is a finite set of values that variable x_i can take, and a set of constraints $C = \{c_1, \cdots, c_e\}$, where each constraint c_k ($k \in 1..e$) is a binary relation on two variables. A constraint on x_i and x_j is usually denoted by c_{ij}. $(a, b) \in c_{ij}$ means that the constraint c_{ij} holds when $x_i = a$ and $x_j = b$. For the problem of interest here, we require that $\forall a, b \;\; a \in D_i, b \in D_j, (a, b) \in c_{ij}$ if and only if $(b, a) \in c_{ji}$. Verifying whether a tuple (a, b) where $a \in D_i$ and $b \in D_j$ is in c_{ij} is called a* constraint check. *A* solution *of a constraint network is an assignment of a value to each variable such that all constraints in the network are satisfied.*

For simplicity, in the above definition we consider only binary constraints, omitting the unary constraint on any variable [Mac77a]. Without loss of generality we assume there is only one constraint between each pair of variables.

Throughout this paper, n denotes the number of variables, d the size of the largest domain, and e the number of constraints in a constraint network. (x_i, a) denotes a value $a \in D_i$.

Definition 2 *Given a constraint network (N, D, C), the* support *of a value $a \in D_i$ under c_{ij} is a value $b \in D_j$ such that $(a, b) \in c_{ij}$. The value a is* viable *with respect to c_{ij} if it has a support in D_j. A constraint c_{ij} is* consistent from x_i to x_j*, that is along the arc (x_i, x_j), if and only if every $a \in D_i$ has a support in D_j. A constraint c_{ij} is* arc consistent *if and only if it is consistent along both arcs (x_i, x_j) and (x_j, x_i). A constraint network is* arc consistent *if and only if every constraint in the network is arc consistent.*

From the definition, we know that a constraint network is arc consistent if and only if every value is viable with respect to every constraint on its variable.

Before presenting and analyzing the new algorithm, let us briefly recall the AC-3 algorithm which is given in Fig. 1 as AC$\mathcal{X}$. The presentation follows [Mac77a, MF85] with a slight change in notation, and node consistency removed. The name of the algorithm AC$\mathcal{X}$ is parameterized by $\mathcal{X}$. For AC-3, the $\mathcal{X}$ is "-3" and thus the procedure REVISE$\mathcal{X}$ is "REVISE-3". For the new algorithm, AC2001/3.1, the $\mathcal{X}$ is "2001/3.1" and thus the procedure REVISE$\mathcal{X}$ is "REVISE2001/3.1".

To enforce arc consistency in a constraint network, a key task of AC-3 is to check the viability of a value with respect to any related constraint. REVISE-3(x_i, x_j) in Fig 2 is to remove those values in D_i without any support in D_j under c_{ij}. If any value in D_i is removed when revising (x_i, x_j), all binary constraints (or arcs) pointing to x_i, except

```
algorithm AC𝒳
    begin
1.      Q ← {(x_i, x_j) | c_ij ∈ C or c_ji ∈ C, i ≠ j}
        while Q not empty do
            select and delete any arc (x_i, x_j) from Q
2.          if REVISE𝒳(x_i, x_j) then
3.              Q ← Q ∪ {(x_k, x_i) | c_ki ∈ C, k ≠ j}
    end
```

Figure 1: A schema for coarse-grained arc consistency algorithms

```
procedure REVISE-3(x_i, x_j)
    begin
        DELETE ← false
        for each a ∈ D_i do
1.          if there is no b ∈ D_j such that c_ij(a, b) then
                delete a from D_i
                DELETE ← true
        return DELETE
    end
```

Figure 2: Procedure REVISE for AC-3

c_{ji}, will be revised (line 2 and 3 in Fig 1). A queue Q is used to hold these arcs for later processing. It can be shown that this algorithm is correct.

Proposition 1 ([Mac77a]) *Applying algorithm AC-3 to a constraint network makes it arc consistent.*

The traditional derivation of the complexity of AC-3 is given by the following theorem whose proof from [MF85] is modified in order to facilitate the presentation in Section 3.

Theorem 1 ([MF85]) *Given a constraint network (N, D, C), the time complexity of AC-3 is $\mathcal{O}(ed^3)$.*

Proof. Each arc (x_i, x_j) is revised if and only if it enters Q. The observation is that arc (x_i, x_j) enters Q if and only if some value of D_j is deleted (line 2–3 in Fig 1). So, arc (x_i, x_j) enters Q at most d times and thus is revised d times. Given that the number of arcs is $2e$, REVISE(x_i, x_j) is executed $\mathcal{O}(ed)$ times. The complexity of REVISE(x_i, x_j)

in Fig 2 is at most d^2. Hence, the result follows. □

The reader is referred to [Mac77a, MF85] for more details and motivations concerning arc consistency.

Remark. In implementing the queue, to reduce the number of queue operations, one way is simply enqueue the variable whose domain has shrunk, instead of enqueue all relevant arcs. When we dequeue a variable from the queue, we just revise all constraints pointing to this variable. The method is also called variable oriented propagation. This idea appeared in [McG79] and in [CJ98]. In this method, for each variable, one more constraint is revised than in the original algorithm AC-3. However, it seems that the savings from enqueue operations well compensates this cost in our experiments.

3 The New Algorithm

The worst case time complexity of AC-3 is based on a naive implementation of line 1 in Fig. 2 in which b is always searched from scratch. However, from the analysis we know a constraint (x_i, x_j) may be revised many times. The key idea to improve the efficiency of the algorithm is that we need to find from scratch a support for a value $a \in D_i$ *only in the first revision* of the arc (x_i, x_j), and store the support in a structure $Last((x_i, a), x_j)$. When checking the viability of $a \in D_i$ in the subsequent revisions of the arc (x_i, x_j), we only need to check whether its stored support $Last((x_i, a), x_j)$ is still in the domain D_j. If it was removed (because of the revision of other constraints), we would just have to explore the values in D_j that are "after" the support since its "predecessors" have already been checked before.

Assume without loss of generality that each domain D_i is associated with a total ordering $<_d$. The function *succ*(a, D_j), where D_j denotes the current domain of x_j during the procedure of arc consistency enforcing, returns the first value in D_j that is after a in accordance with $<_d$, or NIL, if no such an element exists. We define NIL as a value not belonging to any domain but preceding any value in any domain.

As a simple example, let the constraint c_{ij} be $x_i = x_j$, with $D_i = D_j = [1..11]$. The removal of value 11 from D_j (say, after the revision of some arc leaving x_j) leads to a revision of (x_i, x_j). REVISE-3 will look for a support for every value in D_i, for a total cost of $1 + 2 + \ldots + 9 + 10 + 10 = 65$ constraint checks, whereas only $(x_i, 11)$ had lost support. The new revision procedure makes sure that for each $a \in [1..10]$, $Last((x_i, a), x_j)$ still belongs to D_j, and finds that $Last((x_i, 11), x_j)$ has been removed. Looking for a new support for 11 does not need any constraint check since D_j does not contain any value greater than $Last((x_i, 11), x_j)$, which was equal to 11. It saves 65 constraint checks compared to AC-3.

The new algorithm, AC2001/3.1, is the main algorithm AC$\mathcal{X}$ augmented with the initialization of $Last((x_i, a), x_j)$ to be NIL for any constraint c_{ij} and any value $a \in D_i$. The corresponding revision procedure, REVISE2001/3.1 is given in Fig. 3. In Fig. 3, line 1 checks if the support in $Last$ is still valid and otherwise line 2 makes use of the domain ordering to find the first support after the old one. We now show the correctness of AC2001/3.1.

```
procedure REVISE2001/3.1(x_i, x_j)
    begin
        DELETE ← false
        for each a ∈ D_i do
            b ← Last((x_i, a), x_j)
1.          if b ∉ D_j then
                b ← succ(b, D_j)
2.              while (b ≠ NIL) and (¬c_ij(a, b)) do
                    b ← succ(b, D_j)
                if b ≠ NIL then
                    Last((x_i, a), x_j) ← b
                else
                    delete a from D_i
                    DELETE ← true
        return DELETE
    end
```

Figure 3: Procedure REVISE for AC2001/3.1

Theorem 2 *Applying algorithm AC2001/3.1 to a constraint network makes it arc consistent.*

Proof. AC-3 and AC2001/3.1 have exactly the same initialization phase except that AC2001/3.1 stores $Last((x_i, a), x_j)$, the support found for each a on each c_{ij}. It is sufficient to show that REVISE-3(x_i, x_j) and REVISE2001/3.1(x_i, x_j) are equivalent given that D_i and D_j are the same when either procedure is called. In other words, a values is deleted by REVISE-3 iff it is deleted by REVISE2001/3.1. Obviously the return value would also be the same for both procedures. Without loss of generality, we can assume that REVISE-3 visits the same values as the ordering used in $<_d$.

Suppose a value a is deleted by REVISE-3. Line 1 in REVISE-3 tells us that a has no support. Consequently, line 1 in REVISE2001/3.1 is also true and the while loop in line 2 will not find any support. Hence a will be deleted.

Now consider REVISE2001/3.1 deleting a value a. Let b be the previous support, Last$((x_i, a), x_j)$. Since line 1 will be true, b is not a support for a. The while loop at line 2 also doesn't find for a any support after b. Now suppose there is a support b' such that $b' <_d b$. It must also be a support for a in all the previous domains of x_j. Hence, Last$((x_i, a), x_j) \leq_d b'$, which contradicts $b =$ Last$((x_i, a), x_j)$. Thus, a has no support in D_j and will also be deleted by REVISE-3.

Proposition 1 implies that AC2001/3.1 achieves arc consistency on a constraint network. □

Next, we show that AC2001/3.1 has optimal worst case time complexity.

Theorem 3 *The worst case time complexity of AC2001/3.1 is $\mathcal{O}(ed^2)$ with space complexity $\mathcal{O}(ed)$.*

Proof. Here it is helpful to regard the execution of AC2001/3.1 on an instance of a constraint network as a sequence of calls to REVISE2001/3.1(x_i, x_j).

Consider the total time spent on an arc (x_i, x_j). From the proof in Theorem 1, the arc (x_i, x_j) will be revised at most d times.

In the $l^{th}(1 \leq l \leq d)$ revision of (x_i, x_j), let t_l be the time for searching a support for a value $a \in D_i$. t_l can be considered as 1 if $Last((x_i, a), x_j) \in D_j$ (see line 1 in Fig. 3) and otherwise it is s_l which is simply the number of elements in D_j checked after $Last((x_i, a), x_j)$ and before the next support is found (the **while** loop in line 2). So, the total time of the algorithm spent on $a \in D_i$ with respect to (x_i, x_j) is

$$\sum_1^d t_l \leq \sum_1^d 1 + \sum_1^d s_l$$

where $s_l = 0$ if $t_l = 1$. Observe that REVISE2001/3.1(x_i, x_j) checks an element in D_j at most once when looking for a support for $a \in D_i$. Therefore, $\sum_1^d s_l \leq d$ which results in $\sum_1^d t_l \leq 2d$.

To revise (x_i, x_j), we need to find a support for each value of D_i. For there are up to d values in D_i, at most $\mathcal{O}(d^2)$ time will be spent on revising the arc (x_i, x_j).

Hence, the complexity of AC2001/3.1 is $\mathcal{O}(ed^2)$ since the number of arcs in the constraint network is $2e$ (one constraint is regarded as two arcs).

The space complexity of AC2001/3.1 is bounded above by the size of Q, and the structure $Last$. Q can be of complexity in $O(n)$ or $O(e)$, depending on the the implementation of the queue. The size of $Last$ is in $O(ed)$ since each value $a \in D_i$ needs a space in $Last$ with respect to each constraint involving x_i. This gives a $O(ed)$ overall space complexity. □

4 Experimental Results: AC2001/3.1 versus AC-3 and AC-6

We presented AC2001/3.1, a refinement of AC-3 with optimal worst case time complexity. It remains to see whether it is effective in saving constraint checks and/or cpu time when compared to AC-3. As we said previously, the goal is not to compete with AC-6/AC-7, which have very subtle data structure for the propagation phase. However, we will see in the experimental results that AC2001/3.1 is often competitive with these fine-grained algorithms.

There have been many experimental studies on the performance of general arc consistency algorithms [Wal93, Bes94, BFR99]. Here, we take problems used in [BFR99], namely some random CSPs and Radio Link Frequency Assignment Problems (RLFAPs). Given the experimental results of [BFR99], AC-6 is chosen as a representative of a state-of-the-art algorithm because of its good runtime performance over the problems of concern. In addition, a new artificial problem, DOMINO, in the same vein as the problem in Fig. 5 in [DP88], is designed to study the worst case performance of AC-3.

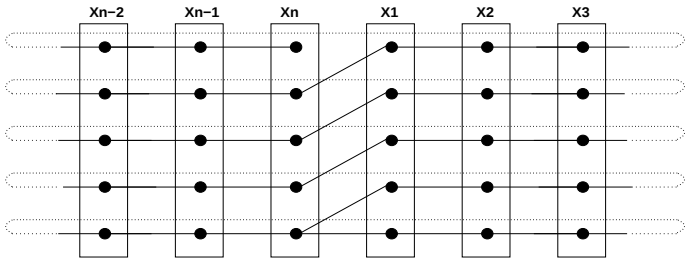

Figure 4: The domino problem

Randomly generated problems. For the random instances, we used a model B generator [Pro96]. The parameters are $\langle N, D, C, T\rangle$, where N is the number of variables, D the size of the domains, C the number of constraints (the *density* $p1$ is equal to $2C/N \cdot (N-1)$), and T the number of forbidden tuples per constraint (the *tightness* $p2$ is equal to T/D^2). We used the generator available in [FBDR96]. For each class of problems tested, we ran the first 50 instances generated using the initial seed 1964 (as in [BFR99]).

RLFAP. The radio link frequency assignment problem (RLFAP) is to assign frequencies to communication links to avoid interference [CdGL$^+$99]. We use the CELAR instances of RLFAP which are real-life problems available in the FullRLFAP archive[1] at `ftp://ftp.cs.unh.edu/pub/csp/archive/code/benchmarks`.

DOMINO. Informally the DOMINO problem is an undirected constraint graph with n variables and a cycle of constraints. The domain of any variable x_i is $D_i = \{1, 2, \ldots, d\}$. The constraints are $C = \{c_{i(i+1)} | \forall i \in 1..n-1\} \cup \{c_{1n}\}$ where $c_{1n} = \{(d, d)\} \cup \{(v, v+1) \mid v < d\}$ is called the *trigger constraint* and the other constraints in C are identity relations. (See the value based constraint graph in Fig. 4.) A DOMINO instance is thus fully characterized by the pair of parameters $\langle n, d\rangle$. The trigger constraint will make one value invalid during arc consistency and that value will trigger the domino effect on the values of all domains until each domain has only one value d left. So, each revision of an arc in coarse-grained algorithms removes one value while fine-grained algorithms only do the necessary work.

Some details of our implementation of AC2001/3.1 and AC-3.0 are as follows. We implemented domains and related operations by double-linked lists. The Q in AC-3 is implemented as a queue with a FIFO policy. For AC-6, we noted that using a single currently supported list per value is faster than using multiple lists with respect to related constraints as needed for AC-7. This may be one reason why AC-7 is slower than AC-6 in [BFR99]. Our implementation of AC-6 adopts a single currently supported list. The code is written in C++ with *g++*. The experiments are run on a PC Pentium II 300MHz processor with Linux. The performance of arc consistency algorithms here is measured along two dimensions: running time and number of constraint checks (#ccks).

[1]We thank the Centre d'Electronique de l'Armement (France).

	AC-3		AC2001/3.1		*AC-6*$^{(*)}$
	#ccks	time	#ccks	time	*time*
$P1$ (under-constrained)	100,010	0.04	100,010	0.05	*0.07*
$P2$ (over-constrained)	507,783	0.18	487,029	0.16	*0.10*
$P3$ (phase transition of AC)	2,860,542	1.06	688,606	0.34	*0.32*
$P4$ (phase transition of AC)	4,925,403	1.78	1,147,084	0.61	*0.66*
SCEN#08 (arc inconsistent)	4,084,987	1.67	2,721,100	1.25	*0.51*

Table 1: Arc consistency results in mean number of constraint checks (#ccks) and mean cpu time in seconds (time). (*) The number of constraint checks performed by AC-6 is similar to that of AC2001/3.1, as discussed in Section 6.

4.1 Arc Consistency as a Preprocessing Step

The first set of experiments shows the efficiency of AC2001/3.1 when arc consistency is used for preprocessing (without search). In this case, the chance to have some propagations is small on real instances. As such, we also choose problems falling in the phase transition of arc consistency (see [GMP$^+$97]). To see the different behaviours, we present results for randomly generated instances with different characteristics (those presented in [BFR99]) and on a RLFAP instance where enforcing arc consistency was not trivial:

- P1= $\langle 150, 50, 500, 1250 \rangle$, under-constrained CSPs, where all generated instances are already arc consistent;
- P2= $\langle 150, 50, 500, 2350 \rangle$, over-constrained CSPs, where all generated instances are *arc inconsistent*, which means that the instances are not satisfiable and this can be detected by enforcing arc consistency;
- P3=$\langle 150, 50, 500, 2296 \rangle$ and P4=$\langle 50, 50, 1225, 2188 \rangle$, problems in the phase transition of arc consistency;
- the RLFAP instance SCEN#08, which is arc inconsistent.

Table 1 presents the results. For the randomly generated instances, the number of constraint check (#ccks) and time are averaged over the 50 instances in each class. The under-constrained (P1) and over-constrained (P2) problems. represent cases where there is little or no propagation needed to reach the arc consistent or arc inconsistent state. This is the best case for AC-3. The AC2001/3.1 algorithm still gives comparable runtimes, which indicates that the overhead incurred by AC2001/3.1 is not significant since in P1 there are no savings in constraint checks and P2 only saves about 4% of the checks.

The P3 instances are sparse problems (with a density of 4.5%) at the phase transition of arc consistency. The P4 instances are dense problems (with a complete graph) also at the phase transition of arc consistency. Usually much propagation is needed on these problems to make the network arc consistent. We see that here the runtime of AC2001/3.1 is significantly faster than AC-3 due to large savings in the number of constraint checks.

The final experiment reports the results for a real-life problem, SCEN#08. Here, AC2001/3.1 also saves a significant amount of constraint checks and is also faster.

	MAC-3		MAC2001/3.1		*MAC6*
	#ccks	time	#ccks	time	*time*
SCEN#01	5,026,208	2.33	1,983,332	1.62	*2.05*
SCEN#11	77,885,671	39.50	9,369,298	21.96	*14.69*
GRAPH#09	6,269,218	2.95	2,127,598	1.99	*2.41*
GRAPH#10	6,790,702	3.04	2,430,109	1.85	*2.17*
GRAPH#14	5,503,326	2.53	1,840,886	1.66	*1.90*

Table 2: Results for search of the first solution with a MAC algorithm in number of constraint checks (#ccks) and cpu time in seconds (time).

In order to compare AC2001/3.1 to AC-6, it is necessary to first understand that they perform the same number of constraint checks (see Section 6). Here the runtimes show that for most of the problems AC2001/3.1 and AC-6 are comparable and we will return again to this comparison with the DOMINO problem.

4.2 Maintaining Arc Consistency during Search

The second set of experiments we present in this section shows the behaviour of AC2001/3.1 when arc consistency is maintained during search (MAC algorithm [SF94]) to find the first solution. We present results for all the instances contained in the FullRLFAP archive for which more than 2 seconds were needed to find a solution or to prove that none exists. It has to be noticed that the original objective in these instances is to find the "best" solution under some criteria. This is of course out of the scope of this paper.

Table 2 contains the results. From these instances we can see a significant gain for AC2001/3.1 on AC-3, with up to 9 times less constraint checks and twice less cpu time on SCEN#11. As for the experiments performed on random instances at the phase transition of arc consistency, this tends to show that the trick of storing the $Last$ data structure significantly pays off. In addition, we see that in spite of its simple data structures, AC2001/3.1 is faster than AC-6 on all instances except the difficult SCEN#11. The reason why AC-6 takes more time can be explained as follows. The main contribution to the slow down of AC-6 is the maintenance of the currently supported list for each value of each variable. Our experiments show that the overhead of maintaining the list does not usually compensate for the savings, at least under the assumption that constraint checking is cheap.

4.3 The DOMINO Problem

The last set of experiments we made shows the extreme case where the arc consistency process converges after a long propagation that removes all values in all domains but those belonging to the unique solution. The DOMINO problem is designed to exhibit this behaviour. What we can expect from such a pathological case is to show us the deep properties of non optimal coarse-grained, optimal coarse-grained and fine-grained algorithms.

The results are in Table 3. We can see clearly the effect of the non-optimal worst case complexity of AC-3. The number of constraint checks and cpu time increase dramatically

	AC-3		AC2001/3.1		*AC6*
	#ccks	time	#ccks	time	*time*
$\langle 1000, 10\rangle$	319,964	0.19	155,009	0.13	*0.16*
$\langle 500, 100\rangle$	90,845,149	25.70	7,525,099	3.18	*2.66*
$\langle 300, 300\rangle$	1,390,485,449	381.25	40,545,299	15.40	*12.16*

Table 3: Results on the DOMINO problem in number of constraint checks (#ccks) and cpu time in seconds (time).

	AC2001/3.1 domain checks	*AC6* list checks
$\langle 1000, 10\rangle$	53,991	17,999
$\langle 500, 100\rangle$	2,524,401	88,999
$\langle 300, 300\rangle$	13,544,401	179,399

Table 4: Results on the DOMINO problem in the number of domain versus list checks

with the size of the domains. As we already saw, AC2001/3.1 and AC-6 perform exactly the same number of constraint checks. However, as in the most difficult problem of Section 4.2, the fine-grained feature of AC-6 pays off with respect to AC2001/3.1 especially when the domain size increases. This can be explained by the way AC2001/3.1 and AC-6 propagate the deletions of values. If we look more closely at the operations performed by these two algorithms when a value (x_j, b) is deleted, we note that they achieve optimality in two different ways. For each (x_i, a) such that x_i shares a constraint with x_j, AC2001/3.1 checks $Last((x_i, a), x_j)$ against the domain D_j to know whether (x_i, a) still has support (see line 1 in Fig. 3). The $Last$ indicates where to start the new search for support. AC-6 on the other hand, maintains for each (x_j, b) the list of the values a for x_i with $Last((x_i, a), x_j) = b$. When b is deleted from D_j, thanks to these lists of supported values, AC-6 directly knows which values in D_i need to seek a new support, and where to start the new search.

By counting the number of such operations they perform (membership test of a $Last$ in a domain for AC2001/3.1 and list operations on supported lists for AC-6) we obtain the following interesting information. While they don't perform any such tests during the initialization phase, the number of tests they perform during the propagation as shown in Table 4 differs quite significantly. As domain size increases (and thus propagation becomes longer), the cost of AC2001/3.1 propagation increases faster than that of AC-6.

5 An Application to Path Consistency and Non-binary Constraints

5.1 Path Consistency

Notation. In this subsection, to simplify the presentation a variable x_i is represented by its index i.

Assume there is a constraint between any pair of variables in a given constraint network (N, D, C). If it is not the case, we add a special constraint between the uncon-

strained pairs of variables. This constraint allows the constrained variables to take any values. The network is *path consistent* if and only if for any $c_{ij} \in C$, any tuple $(a, b) \in c_{ij}$, and any variable $k \in N$, there exists a value $v \in D_k$ such that the values a, b, and v satisfy the constraints among variables i, j, and k.

The same idea behind AC2001/3.1 applies here. Specifi cally, in order to fi nd a new support for each $(a, b) \in c_{ij}$ with respect to a variable, say k, it is not necessary to start from scratch every time. We can start from where we stopped before. $Last((i, a), (j, b), k)$ is used to remember that point.

The path consistency algorithm, which we have named PC2001/3.1, partially motivated by the algorithm in [CJ98], is shown in Fig 5. It includes two parts: initialization (INITIALIZE(Q)) and propagation (the **while** loop on Q). During the initialization, a fi rst support is searched for each pair of values $((i, a), (j, b))$ on each third variable k. This support is stored in $Last((i, a), (j, b), k)$. When a tuple (a, b) is removed from c_{ij}, we enqueue $((i, a), j)$ and $((j, b), i)$ into Q. Later, when $((i, a), k)$ is popped from Q, REVISE_PATH($((i, a), k, Q$) (in Fig 6) will check every constraint c_{ij} where $j \in N - \{i, k\}$ to see if any tuple in c_{ij} is affected by the modifi cation of c_{ik}. For each constraint c_{ij}, REVISE_PATH tries to fi nd in D_k a support not from scratch but from its support in the previous revision (line 1 and line 2 in Fig 6) for only those tuples starting with a.

---

algorithm PC2001/3.1
  begin
    INITIALIZE(Q)
    while Q not empty **do**
      Select and delete any $((i, a), j)$ from Q
      REVISE_PATH($((i, a), j, Q)$)
    endwhile
  end
procedure INITIALIZE(Q)
  begin
    for any $i, j, k \in N$ **do**
      for any $a \in D_i, b \in D_j$ such that $(a, b) \in c_{ij}$ **do**
        if there is no $v \in D_k$ such that $(a, v) \in c_{ik} \wedge (v, b) \in c_{kj}$
        then
          $c_{ij}(a, b) \leftarrow$ **false**; $c_{ji}(b, a) \leftarrow$ **false**
          $Q \leftarrow Q \cup \{(i, a), j\} \cup \{(j, b), i\}$
        else
          Let $v \in D_k$ be the fi rst value satisfying
            $(a, v) \in c_{ik} \wedge (v, b) \in c_{kj}$
          Last($(i, a), (j, b), k$) $\leftarrow v$
  end

Figure 5: Algorithm to enforce path consistency

```
procedure REVISE_PATH( (i,a), k, Q)
    begin
        for any j ∈ N, j ≠ i, j ≠ k do
            for any b ∈ D_j such that (a,b) ∈ c_ij do
1.              v ← Last((i,a),(j,b),k)
2.              while (v ≠ NIL) ∧ ((a,v) ∉ c_ik ∨ (v,b) ∉ c_kj) do
                    v ← succ(v, D_k)
                if v = NIL then
                    c_ij(a,b) ← false; c_ji(b,a) ← false
                    Q ← Q ∪ {((i,a),j} ∪ {((j,b),i)}
                else Last((i,a),(j,b),k)) ← v
            endfor
    end
```

Figure 6: Revision procedure for PC algorithm

For this algorithm, we have the following result.

Theorem 4 *The time complexity of the algorithm PC2001/3.1 is $\mathcal{O}(n^3d^3)$ with space complexity $\mathcal{O}(n^3d^2)$.*

Proof. The complexity of the algorithm PC depends on the procedure REVISE_PATH whose second loop is to find a support for the tuple $((i,a),(j,b))$ with respect to k. The **while** loop in line 2 (Fig 6) either takes constant time if the condition is not satisfied (the support stored in $Last$ is still valid), or skips values in D_k otherwise. For the second case, no matter how many times we try to find a support for $((i,a),(j,b))$, at most we skip d values since totally we have only d values in D_k.

We know that it is necessary to find a support for $((i,a),(j,b))$ with respect to k if and only if some tuple (a,v) is removed from c_{ik}. So we need to find such a support d times. From first paragraph, for these d times we have at most d constant checks and d skips in total. As a result, to find a support for $((i,a),(j,b))$ with respect to k we need $2d$ steps. Given that i,j,k can be any variable from N and a,b any value from D_i and D_j respectively, we have n^3d^2 possible $((i,a),(j,b))$'s and k's. Hence, the total time cost is $n^3d^2 \times 2d$, that is $\mathcal{O}(n^3d^3)$.

The main working space is for the structure $Last((i,a),(j,b),k)$. The size of this structure is the number of combinations of possible choices for i,j,k,a,b, that is $\mathcal{O}(n^3d^2)$. □

The PC2001/3.1 has time complexity of $\mathcal{O}(n^3d^3)$ and space complexity of $\mathcal{O}(n^3d^2)$ which is the same bounds as the best known results obtained in [Sin96]. The algorithm in [Sin96] employs a supported list for each value of a variable and propagates the removal of values in a fashion of AC-6. Compared with the supported list, the structure

$Last((i,a),(j,b),k)$ is easier to maintain. This makes the PC2001/3.1 algorithm both simpler to understand and to implement.

5.2 Non-binary Constraints

AC2001/3.1 can be extended to GAC2001/3.1 to deal with non-binary constraints. The definition of arc consistency for non binary constraints is a direct extension of the binary one [Mac77b, MH86]. Let us denote by $var(c_j) = (x_{j_1}, \ldots, x_{j_q})$ the sequence of variables involved in a constraint c_j, by $rel(c_j)$ the set of tuples allowed by c_j, and by $D^{var(c_j)}_{|x_i=a}$ the set of the tuples τ in $D_{j_1} \times \cdots \times D_{j_q}$ with $\tau[x_i] = a$ (where $i \in \{j_1, .., j_q\}$). A tuple τ in $D^{var(c_j)}_{|x_i=a} \cap rel(c_j)$ is called a support for (x_i, a) on c_j. The constraint c_j is *arc consistent* (also called *generalized arc consistent*, or *GAC*) iff for any variable x_i in $var(c_j)$, every value $a \in D_i$ has a support on c_j. Tuples in a constraint c_j are totally ordered with respect to the lexicographic ordering obtained by combining the ordering $<_d$ of each domain with the ordering of the sequence $var(c_j)$ (or with respect to any total order used when searching for support). Once this ordering is defined, a call to REVISE2001/3.1(x_i, c_j) (see Fig. 7) checks for each $a \in D_i$ whether $Last((x_i, a), c_j)$, which is the smallest support found previously for (x_i, a), still belongs to $D^{var(c_j)}$. If not, it looks for a new support for a on c_j. If such a support τ exists, it is stored as $Last((x_i, a), c_j)$, otherwise a is removed from D_i. The function *succ*$(\tau, D^{var(c_j)}_{|x_i=a})$ returns the smallest tuple in $D^{var(c_j)}_{|x_i=a}$ greater than τ.

---

procedure REVISE2001/3.1(x_i, c_j)
  begin
    DELETE $\leftarrow$ **false**
    for each $a \in D_i$ **do**
      $\tau \leftarrow$ Last$((x_i, a), c_j)$
      if $\exists k / \tau[x_{j_k}] \notin D_{j_k}$ **then**
        $\tau \leftarrow$ *succ*$(\tau, D^{var(c_j)}_{|x_i=a})$
        while $(\tau \neq NIL)$ **and** $(\neg c_j(\tau))$ **do**
          $\tau \leftarrow$ *succ*$(\tau, D^{var(c_j)}_{|x_i=a})$
        if $\tau \neq NIL$ **then**
          Last$((x_i, a), c_j) \leftarrow \tau$
        else
          delete a from D_i
          DELETE $\leftarrow$ **true**
    return DELETE
  end

Figure 7: Procedure REVISE for GAC2001/3.1

In Fig. 8, we present a version of the main algorithm based on the one proposed in [Mac77b]. It is a brute force propagation schema that does not take into account the fact that in practice some of the constraints may have *ad hoc* propagators. Thus the algorithm may have to be adapted depending on the architecture of the solver in which it is used. Standard techniques are described in [ILO99, Lab00].

---

algorithm GAC$\mathcal{X}$
  begin
    $Q \leftarrow \{(x_i, c_j) \mid c_j \in C, x_i \in var(c_j)\}$
    while Q not empty **do**
      select and delete any pair (x_i, c_j) from Q
      if REVISE$\mathcal{X}(x_i, c_j)$ **then**
        $Q \leftarrow Q \cup \{(x_k, c_m) \mid c_m \in C, x_i, x_k \in var(c_m), m \neq j, i \neq k\}$
  end

Figure 8: A non binary version of coarse-grained arc consistency algorithm

Complexity. The worst-case time complexity of GAC2001/3.1 depends on the arity of the constraints involved in the constraint network. The greater the number of variables involved in a constraint, the higher the cost to propagate it. Let us first limit our analysis to the cost of enforcing GAC on a single constraint, c_j, of arity $r = |var(c_j)|$. For each variable $x_i \in var(c_j)$, for each value $a \in D_i$, we look for supports in the space $D^{var(c_j)}_{|x_i=a}$, which can contain up to d^{r-1} tuples. If the cost of constraint checks[2] is in $\mathcal{O}(r)$ this gives a cost in $\mathcal{O}(rd^{r-1})$ for checking viability of a value. Since we have to find support for rd values, the cost of enforcing GAC on c_j is in $\mathcal{O}(r^2 d^r)$. If we enforce GAC on the whole constraint network, values can be pruned by other constraints, and each time a value is pruned from the domain of a variable involved in c_j, we have to revise c_j. So, c_j can be revised up to rd times. Fortunately, additional calls to REVISE2001/3.1 do not increase its complexity since, as in the binary case, $Last((x_i, a), c_j)$ ensures that the search for support for (x_i, a) on c_j will never check twice the same tuple. Therefore, in a network involving constraints of arity bounded by r, the total time complexity of GAC2001/3.1 is in $\mathcal{O}(er^2 d^r)$.

6 Related Work and Discussion

Many arc consistency algorithms have been designed since the birth of the first such algorithm. In this section we present a systematic way to view these algorithms including

[2]The cost of a constraint check is sometimes considered as constant time while it is natural to assume its cost be linear to its arity.

AC-3, AC-4, AC-6 , AC-7 and AC2001/3.1. We also present an analysis of the performance of these algorithms, especially AC2001/3.1 and AC-6.

6.1 A Classification and Comparison of AC algorithms

Arc consistency algorithms can be classified by their methods of propagation. So far, two approaches are employed in known efficient algorithms: arc oriented and value oriented. Arc oriented propagation originates from AC-1 and its underlying computation model is the constraint graph where we have only variables and topological relationship between variables derived from constraints.

Definition 3 *The* constraint graph *of a constraint network* (N, D, C) *is the graph* $G = (\mathcal{V}, \mathcal{E})$ *where* $\mathcal{V} = N$ *and* $\mathcal{E} = \{(i, j) \mid \exists c_{ij} \in C\}$.

Value oriented propagation originates from AC-4 and its underlying computation model is the value based constraint graph where each constraint is also represented as a (sub)graph. For example, the graph in Fig 4 is a value based graph where a vertex is a value and an edge is an allowed tuple by the corresponding constraint.

Definition 4 *The* value based constraint graph *of a constraint network* (N, D, C) *is* $G = (\mathcal{V}, \mathcal{E})$ *where* $\mathcal{V} = \{(i, a) \mid x_i \in N, a \in D_i\}$ *and* $\mathcal{E} = \{((i, a), (j, b)) \mid a \in D_i, b \in D_j, c_{ij} \in C, (a, b) \in c_{ij}\}$.

The value based constraint graph is also known under the names *consistency graph* or *microstructure*. A more specific name for the traditional constraint graph may be *variable based constraint graph*. The key idea of value oriented propagation is that once a value is removed only the viability of those values depending on it will be checked. Thus it is more fine-grained than arc oriented propagation. Algorithms working with variable and value based constraint graphs can be classified respectively as coarse-grained algorithms and fine-grained algorithms.

An immediate observation is that compared with variable based constraint graph, the time complexity analysis in value based constraint graph is straightforward. That is, the total number of operations during the execution of a fine-grained algorithm will be of the same order as the number of edges in the value based constraint graph: $\mathcal{O}(ed^2)$. As far as we know, Perlin [Per92] is the first to make value based constraint graph explicit in arc consistency enforcing algorithm.

Given a computation model of propagation, the algorithms differ in the implementation details. Under variable based constraint graph, AC-3 [Mac77a] can be thought of as an open algorithm, as suggested by our notation AC$\mathcal{X}$. Its time complexity analysis in [MF85] can be regarded as a realized implementation where a very intuitive revision procedure is employed. The new algorithm AC2001/3.1 presented in this paper uses a new implementation of the revision procedure, leading to the optimal worst case time complexity. Our new approach simply remembers the support obtained in the previous revision of an arc while in the old one, the choice is to be lazy, forgetting previous computation. There are also some approaches to improve the space complexity of AC-3 in [McG79, CJ98].

For value based constraint graphs, AC-4 is the first AC implementation and AC-6 is a lazy version of AC-4. AC-7 exploits the bidirectionality on the basis of AC-6. Bidirectionality states that given any c_{ij}, c_{ji}, and any $a \in D_i, b \in D_j$, $(a, b) \in c_{ij}$ if and only if $(b, a) \in c_{ji}$.

Another observation is that the general properties or knowledge of a constraint network can be isolated from a specific arc consistency enforcing algorithm. In fact the idea of *metaknowledge* [BFR99] can be applied to algorithms for either computation model. For example, to save the number of constraint checks, the bidirectionality can be employed also in coarse-grained algorithm, e.g., in [Gas78, LBH03]. Other propagation heuristics [WF92] such as propagating deletion first [BFR99] are also applicable to the algorithms of both models.

We have delineated the AC algorithms which shows that AC2001/3.1 and AC-6 are methodologically different. From a technical perspective, the time complexity analysis of AC2001/3.1 is different from that of AC-6 where the worst case time complexity analysis is straightforward. The point of commonality between AC2001/3.1 and AC-6 is that they face the same problem: the domain may shrink during the process of arc consistency enforcing and thus the recorded support may not be valid in the future. This makes some portions of the implementation of AC2001/3.1 similar to AC-6. We remark that the proof technique in the traditional view of AC-3 does not directly lead to AC2001/3.1 and its complexity results.

6.2 Analysis of the Performance of AC Algorithms

The time complexity of AC-3 is in $\mathcal{O}(ed^3)$ while that of AC-4, AC-6, AC-7 and AC2001/3.1 is in $\mathcal{O}(ed^2)$. As for space complexity, AC-3 uses as little as $\mathcal{O}(e)$ for its queue, AC-4 has a complexity of $\mathcal{O}(ed^2)$, and AC2001/3.1, AC-6 and AC-7 have $\mathcal{O}(ed)$. When dealing with non-binary constraints, GAC3 [Mac77b] has a $\mathcal{O}(er^3d^{r+1})$ time complexity, GAC2001/3.1 is in $\mathcal{O}(er^2d^r)$, while GAC4 [MM88] and GAC-schema [BR97] are in $\mathcal{O}(erd^r)$. GAC4 is a factor r better than GAC2001/3.1 because it computes the d^r possible constraint checks on a constraint once and for all at the beginning, storing the information in lists of supported values. For GAC-schema, the reason is that the use of *multidirectionality* (i.e., bidirectionality for non-binary constraints) prevents it from checking a tuple once for each value composing it.

AC-4 does not perform well in practice [Wal93, BFR99] because it *reaches* the worst case complexity both theoretically and in actual problem instances when constructing the value based constraint graph for the instance. Other algorithms like AC-3 and AC-6 can take advantage of some instances where the worst case doesn't occur. In practice, both artificial and real life problems rarely make algorithms behave in the worst case except for AC-4.[3]

The number of constraint checks is also used to evaluate practical time efficiency of AC algorithms. In theory, applying bidirectionality to all algorithms will result in better performance since it decreases the number of constraint checks. However, if the cost of constraint checks is cheap, the overhead of using bidirectionality may not be compensated by its savings as demonstrated by [BFR99].

[3]However, the value based constraint graph induced from AC-4 provides a convenient and accurate tool for studying arc consistency.

AC-6 and AC2001/3.1 have the same worst-case time and space complexities. So, an interesting question here is "What are the differences between AC2001/3.1 and AC-6 in terms of constraint checks?".

Let us first briefly recall the AC-6 behavior [Bes94]. AC-6 looks for one support (the *first* one or *smallest* one with respect to the ordering $<_d$) for each value (x_i, a) with respect to each constraint c_{ij} to prove that a is currently viable. When (x_j, b) is found as the smallest support for (x_i, a) wrt c_{ij}, (x_i, a) is added to $S[x_j, b]$, the list of values currently having (x_j, b) as their smallest support. If (x_j, b) is removed from D_j, it is added to the $DeletionSet$, which is the stream driving propagations in AC-6. When (x_j, b) is picked from the $DeletionSet$, AC-6 looks for the *next* support, greater than b, in D_j for each value (x_i, a) in $S[x_j, b]$.

To allow a closer comparison, we will suppose in the following that the $S[x_i, a]$ lists used in AC-6 are split with respect to each constraint c_{ij} involving x_i, leading to a structure $S[x_i, a, x_j]$, as in AC-7.

Property 1 *Given a constraint network (N, D, C). If we suppose AC2001/3.1 and AC-6 follow the same ordering of variables and values when looking for supports and propagating deletions, then, enforcing arc consistency on the network with AC2001/3.1 performs the same constraint checks as with AC-6.*

Proof. Since they follow the same ordering, both algorithms perform the same constraint checks in the initialization phase: they stop search for support for a value (x_i, a) on c_{ij} as soon as the first $b \in D_j$ compatible with a is found, or when D_j is exhausted (then removing a from D_i). During the propagation phase, both algorithms look for a new support for a value (x_i, a) with respect to c_{ij} only when a value b removed from D_j was the current support for a (i.e., $a \in S[x_j, b, x_i]$ for AC-6, and $b = Last((x_i, a), x_j)$ for AC2001/3.1). Both algorithms search in D_j for a new support for a immediately greater than b. Thus, they will find the same new support for a with respect to c_{ij}, or will remove a, at the same time, and with the same constraint checks. And so on. □

From property 1, we see that the difference between AC2001/3.1 and AC-6 cannot be characterized by the number of constraint checks they perform. We will then focus on the way they find which values should look for a new support. For that, both algorithms handle their specific data structure. Let us characterize the number of times each of them checks its own data structure when a set $\Delta(x_j)$ of deletions from D_j is propagated with respect to a given constraint c_{ij}.

Property 2 *Let c_{ij} be a constraint in a constraint network (N, D, C). Let $\Delta(x_j)$ be a set of values removed from D_j that have to be propagated with respect to c_{ij}. If,*

- $d_A = |\Delta(x_j)| + \sum_{b \in \Delta(x_j)} |S[x_j, b, x_i]|$,
- $d_B = |D_i|$, *and*
- d_C = # *constraint checks performed on c_{ij} to propagate $\Delta(x_j)$,*

then, $d_A + d_C$ and $d_B + d_C$ represent the number of operations AC-6 and AC2001/3.1 will respectively perform to propagate $\Delta(x_j)$ on c_{ij}.

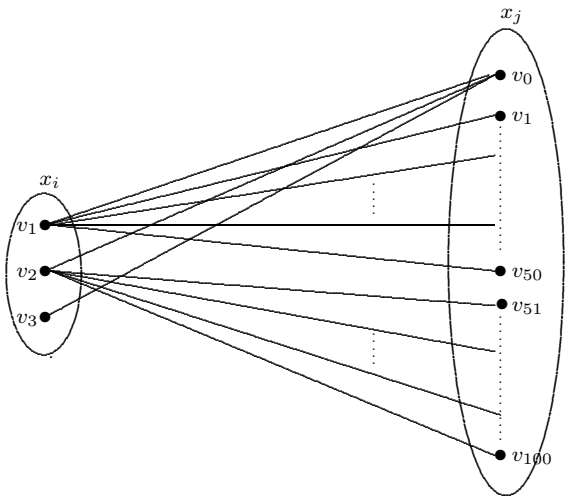

Figure 9: The constraint example

Proof. From property 1 we know that AC-6 and AC2001/3.1 perform the same constraint checks. The difference is in the process leading to them. AC-6 traverses the $S[x_j, b, x_i]$ list for each $b \in \Delta(x_j)$ (i.e., d_A operations), and AC2001/3.1 checks whether $Last((x_i, a), x_j)$ belongs to D_j for every a in D_i (i.e., d_B operations). □

We illustrate this on the extreme case presented in Fig. 9. In that example, the three values of x_i are all compatible with the first value v_0 of x_j. In addition, (x_i, v_1) is compatible with all the values of x_j from v_1 to v_{50}, and (x_i, v_2) with all the values of x_j from v_{51} to v_{100}. Imagine that for some reason, the value v_3 has been removed from D_i (i.e., $\Delta(x_i) = \{v_3\}$). This leads to $d_A = 1$, $d_B = 101$, and $d_C = 0$, which is a case in which propagating with AC-6 is much better than with AC2001/3.1, even if none of them needs any constraint check. Indeed, AC-6 just checks that $S[x_i, v_3, x_j]$ is empty,[4] and stops. AC2001/3.1 takes one by one the 101 values b of D_j to check that their $Last((x_j, b), x_i)$ is not in $\Delta(x_i)$. Imagine now that instead of the value v_3 of D_i these are the values v_1 to v_{100} of D_j that have been removed (i.e., $\Delta_j = \{v_1, \ldots, v_{100}\}$). Now, $d_A = 100$, $d_B = 3$, and $d_C = 0$. This means that AC2001/3.1 will clearly outperform AC-6. Indeed, AC-6 will check for all the 100 values b in $\Delta(x_j)$ that $S[x_j, b, x_i]$ is empty,[5] while AC2001/3.1 just checks that $Last((x_i, a), x_j)$ is not in $\Delta(x_j)$ for every value (totally 3) $a \in D_i$.

Finally, given that both variable and value based constraint graphs can lead to worst case optimal algorithms, we consider their strength on some special constraints: functional, monotonic and anti-functional. For more details, see [VDT92] and [ZY00]. Coarse grained algorithms can be easily adapted to process *monotonic* and *anti-monotonic* constraints in a time complexity of $\mathcal{O}(ed)$ (e.g., using AC2001/3.1). Fine grained algorithms

[4]The only value compatible with (x_i, v_3) is (x_j, v_0), which is currently supported by (x_i, v_1).

[5]Indeed, (x_j, v_0) is the current support for the three values in D_i since it is the smallest in D_j and it is compatible with every value in D_i.

(e.g., AC-4 and AC-6) can deal with *functional* constraints efficiently with complexity $\mathcal{O}(ed)$. We remark that the particular distance constraints in RLFAP can be enforced to be arc consistent in $\mathcal{O}(ed)$ by using a coarse-grained algorithm. It is difficult for coarse-grained algorithm to deal with functional constraints and tricky for fine grained algorithms to handle monotonic constraints. That is why AC-5 [VDT92] is introduced. In fact AC-5 uses both graphs.

By showing that coarse-grained algorithms can be made worst case optimal, this paper opens opportunities to construct new efficient algorithms through reexamining in the context of coarse-grained algorithms those techniques (e.g., bidirectionality and other heuristics or meta knowledge) mainly employed in fine-grained algorithms.

Detailed experiments in [Wal93] show the advantage of AC-3 over AC-4. Our work complements this by providing a way to make coarse-grained algorithms to be worst case optimal.

7 Conclusion

This paper presents AC2001/3.1, a coarse-grained algorithm that improves AC-3. AC2001/3.1 uses an additional data structure, the $Last$ supports, which should be maintained during propagation. This data structure permits a significant improvement on AC-3, and decreases the worst case time complexity to the optimal $O(ed^2)$. AC2001/3.1 is the first algorithm in the literature achieving optimally arc consistency while being free of any lists of supported values. While worst case time complexity gives us the upper bound on the time complexity, in practice, the running time and number of constraint checks are the prime consideration. Our experiments show that AC2001/3.1 significantly reduces the number of constraint checks and the running time of AC-3 on hard arc consistency problems. Furthermore, the running time of AC2001/3.1 is competitive with the best known algorithms, based on the benchmarks from the experiments in [BFR99]. Its behavior is analysed, and compared to that of AC-6, making a contribution to the understanding of the different AC algorithms. The paper shows how the technique used in AC2001/3.1 directly applies to non binary constraints. In addition, this technique can also be used to produce a new algorithm for path consistency. We conjecture from the results of [CJ98] that this algorithm can give a practical implementation for path consistency.

Acknowledgments

The first author would like to thank Philippe Charman who pointed out to him the negative side of fine-grained algorithms. The first author also wants to thank all the members of the OCRE team for the discussions they had about the specification of the CHOCO language.

References

[Bes94] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.

[BFR95] C. Bessière, E. C. Freuder, and J. C. Régin. Using inference to reduce arc consistency computation. In *Proceedings IJCAI'95*, pages 592–598, Montréal, Canada, 1995.

[BFR99] C. Bessière, E.C. Freuder, and J.C. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107:125–148, 1999.

[BR97] C. Bessière and J.C. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings IJCAI'97*, pages 398–404, Nagoya, Japan, 1997.

[BR01] C. Bessière and J.C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings IJCAI'01*, pages 309–315, Seattle WA, 2001.

[CdGL+99] C. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio link frequency assignment. *Constraints*, 4:79–89, 1999.

[CJ98] A. Chmeiss and P. Jégou. Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7(2):121–142, 1998.

[DP88] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[FBDR96] D. Frost, C. Bessière, R. Dechter, and J.C. Régin. Random uniform csp generators. URL: http://www.ics.uci.edu/~dfrost/csp/generator.html, 1996.

[Gas78] J. Gaschnig. Experimental case studies of backtrack vs waltz-type vs new algorithms for satisficing assignment problems. In *Proceedings CCSCSI'78*, pages 268–277, 1978.

[GMP+97] I.P. Gent, E. MacIntyre, P. Prosser, P. Shaw, and T. Walsh. The constrainedness of arc consistency. In *Proceedings CP'97*, pages 327–340, Linz, Austria, 1997.

[ILO99] ILOG. *User's manual*. ILOG Solver 4.4, ILOG S.A., 1999.

[Lab00] F. Laburthe. *User's manual*. CHOCO, 0.39 edition, 2000.

[LBH03] C. Lecoutre, F. Boussemart, and F. Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *Proceedings CP'03*, pages 480–494, Kinsale, Ireland, 2003.

[Mac77a] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mac77b] A.K. Mackworth. On reading sketch maps. In *Proceedings IJCAI'77*, pages 598–606, Cambridge MA, 1977.

[McG79] J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphism. *Information Science*, 19:229–250, 1979.

[MF85] A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.

[MH86] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[MM88] R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings ECAI'88*, pages 651–656, Munchen, FRG, 1988.

[Per92] M. Perlin. Arc consistency for factorable relations. *Artificial Intelligence*, 53:329–342, 1992.

[Pro96] P. Prosser. An empirical study of phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81(1–2):81–109, 1996.

[SF94] D. Sabin and E. Freuder. Contradicting conventional wisdosdom in constraint satisfaction. In *Proceedings of the Second Workshop on Principles and Practice of Constraint Programming*, pages 10–20, Rosario, Orcas Island, Washington, 1994.

[Sin96] M. Singh. Path consistency revisited. *Int. Journal on Art. Intelligence Tools*, 6(1&2):127–141, 1996.

[VDT92] P. Van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[Wal93] R.J. Wallace. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs. In *Proceedings IJCAI'93*, pages 239–245, Chambéry, France, 1993.

[WF92] R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *Proceedings Ninth Canadian Conference on Artificial Intelligence*, pages 163–169, Vancouver, Canada, 1992.

[ZY00] Y. Zhang and R.H.C. Yap. Arc consistency on n-ary monotonic and linear constraints. In *Proceedings CP'00*, pages 470–483, Singapore, 2000.

[ZY01] Y. Zhang and R.H.C. Yap. Making AC-3 an optimal algorithm. In *Proceedings IJCAI'01*, pages 316–321, Seattle WA, 2001.