Making AC-3 an Optimal Algorithm

Yuanlin Zhang and **Roland H.C. Yap***
School of Computing, National University of Singapore
Lower Kent Ridge Road, 119260, Singapore
{zhangyl, ryap}@comp.nus.edu.sg

Abstract

The AC-3 algorithm is a basic and widely used arc consistency enforcing algorithm in Constraint Satisfaction Problems (CSP). Its strength lies in that it is simple, empirically efficient and extensible. However its worst case time complexity was not considered optimal since the first complexity result for AC-3 [Mackworth and Freuder, 1985] with the bound $\mathcal{O}(ed^3)$, where e is the number of constraints and d the size of the largest domain. In this paper, we show suprisingly that AC-3 achieves the optimal worst case time complexity with $\mathcal{O}(ed^2)$. The result is applied to obtain a path consistency algorithm which has the same time and space complexity as the best known theoretical results. Our experimental results show that the new approach to AC-3 is comparable to the traditional AC-3 implementation for simpler problems where AC-3 is more efficient than other algorithms and significantly faster on hard instances.

1 Introduction

Arc consistency is a basic technique for solving Constraint Satisfaction Problems (CSP) and variations of arc consistency are used in many AI and constraint applications. There have been many algorithms developed for arc consistency such as AC-3 [Mackworth, 1977], AC-4 [Mohr and Henderson, 1986], AC-6 [Bessiere, 1994] and AC-7 [Bessiere *et al.*, 1999]. The AC-3 algorithm was proposed in 1977 [Mackworth, 1977]. The first worst case analysis of AC-3 [Mackworth and Freuder, 1985] gives a complexity of $\mathcal{O}(ed^3)$, where e is the number of constraints and d the size of largest domain. This result is deeply rooted in the CSP literature (eg. [Wallace, 1993; Bessiere *et al.*, 1999]) and thus AC-3 is typically considered to be non-optimal. Other algorithms such as AC-4, AC-6, AC-7 are considered theoretically optimal, with time complexity $\mathcal{O}(ed^2)$. As far as we are aware, there has not been any effort to improve the theoretical bound of AC-3 to be optimal. Here, we re-examine AC-3 for a number of reasons. Firstly, AC-3 is one of the simplest AC algorithms and is known to be practically efficient [Wallace, 1993]. The simplicity of arc revision in AC-3 makes it convenient for implementation and amenable to various extensions for many constraint systems. Thus while AC-3 is considered as being sub-optimal, it often is the algorithm of choice and can outperform other theoretically optimal algorithms.

In this paper, we show that AC-3 achieves worst case optimal time complexity of $\mathcal{O}(ed^2)$. This result is surprising since AC-3 being a coarse grained "arc revision" algorithm [Mackworth, 1977], is considered to be non-optimal. The known results for optimal algorithms are all on fine grained "value revision" algorithms. Preliminary experiments show that the new AC-3 is comparable to the traditional implementations on easy CSP instances where AC-3 is known to be substantially better than the optimal fine grained algorithms. In the hard problem instances such as those from the phase transition, the new AC-3 is significantly better and is comparable to the best known algorithms such as AC-6. We also show that the results for AC-3 can be applied immediately to obtain a path consistency algorithm which has the same time and space complexity as the best known theoretical results. [1]

2 Background

In this section we give some background material and notation used herein. The definitions for general CSP follow [Montanari, 1974; Mackworth, 1977].

Definition 1 *A* Constraint Satisfaction Problem *(N, D, C) consists of a finite set of variables $N = \{1, \cdots, n\}$, a set of domains $D = \{D_1, \cdots, D_n\}$, where $i \in D_i$, and a set of constraints $C = \{c_{ij} \mid i, j \in N\}$, where each constraint c_{ij} is a binary relation between variables i and j. For the problem of interest here, we require that $\forall x, y \;\; x \in D_i, y \in D_j, (x, y) \in c_{ij}$ if and only if $(y, x) \in c_{ji}$.*

For simplicity, in the above definition we consider only binary constraints, omitting the unary constraint on any variable [Mackworth, 1977]. Without loss of generality we assume there is only one constraint between each pair of variables.

Definition 2 *The* constraint graph *of a CSP (N, D, C) is the graph $G = (V, E)$ where $V = N$ and $E = \{(i, j) \mid \exists c_{ij} \in C$ or $\exists c_{ji} \in C\}$.*

*The Logistics Institute Asia-Pacific

[1] A related paper by Bessiere and Regin appears in this proceedings.

The arcs in CSP refer to the directed edges in G. Throughout this paper, n denotes the number of variables, d the size of the largest domain, and e the number of constraints in C.

Definition 3 *Given a CSP (N, D, C), an arc (i, j) of its constraint graph is* arc consistent *if and only if $\forall x \in D_i$, there exists $y \in D_j$ such that $c_{ij}(x, y)$ holds. A CSP (N, D, C) is arc consistent if and only if each arc in its constraint graph is arc consistent.*

The AC-3 algorithm for enforcing arc consistency on a CSP is given in figure 2. The presentation follows [Mackworth, 1977; Mackworth and Freuder, 1985] with a slight change in notation and node consistency removed.

```
procedure REVISE((i, j))
  begin
    DELETE ← false
    for each x ∈ D_i do
1.    if there is no y ∈ D_j such that c_ij(x, y) then
          delete x from D_i;
          DELETE ← true
      endif
    return DELETE
  end
```

Figure 1: procedure REVISE for AC-3

```
algorithm AC-3
  begin
1.  Q ← {(i, j) | c_ij ∈ C or c_ji ∈ C, i ≠ j}
    while Q not empty do
      select and delete any arc (k, m) from Q;
2.    if REVISE((k, m)) then
3.      Q ← Q ∪ {(i, k) | (i, k) ∈ C, i ≠ k, i ≠ m}
    endwhile
  end
```

Figure 2: The AC-3 algorithm

The task of REVISE((i, j)) in Fig 1 is to remove those invalid values not related to any other value with respect to arc (i, j). We will show in section 3 that different implementations of line 1 lead to different worst case complexities. As such, we argue that it is more useful to think of AC-3 as a framework rather than a specific algorithm. In AC-3, a CSP is modeled by its constraint graph G, and what AC-3 does is to revise all arcs $(*, i) = \{(k, i) | (k, i) \in E(G)\}$ (line 3 in Fig 2) except some special arc if the domain of variable i is modified by REVISE((i, j)). A queue Q is used to hold all arcs that need to be revised.The traditional understanding of AC-3 is given by the following theorem whose proof from [Mackworth and Freuder, 1985] is modified in order to facilitate the presentation in Section 3.

Theorem 1 *[Mackworth and Freuder, 1985] Given a CSP (N, D, C), the time complexity of AC-3 is $\mathcal{O}(ed^3)$.*

Proof Consider the times of revision of each arc (i, j). (i, j) is revised if and only if it enters Q. The observation is that arc (i, j) enters Q if and only if some value of j is deleted (line 2–3 in fig 2). So, arc (i, j) enters Q at most d times and thus is revised d times. Given that the number of arcs are $2e$, REVISE(i, j) is executed $\mathcal{O}(ed)$ times. The complexity of REVISE((i, j)) in Fig 1 is at most d^2. □

The reader is referred to [Mackworth, 1977; Mackworth and Freuder, 1985] for more details and motivations concerning arc consistency.

3 A New View of AC-3

The traditional view of AC-3 with the worst case time complexity of $\mathcal{O}(ed^3)$ (described by theorem 1) is based on a naive implementation of line 1 in Fig 1 that y is always searched from scratch. Hereafter, for ease of presentation, we call the classical implementation AC-3.0. The new approach to AC-3 in this paper, called AC-3.1, is based on the observation that y in line 1 of Fig 1 needn't be searched from scratch even though the same arc (i, j) may enter Q many times, The search is simply resumed from the point where it stopped in the previous revision of (i, j). This idea is implemented by procedure EXISTy($(i, x), j$) in Fig 3.

Assume without loss of generality that each domain D_i is associated with a total ordering. ResumePoint($(i, x), j$) remembers the first value $y \in D_j$ such that $c_{ij}(x, y)$ holds in the previous revision of (i, j). The *succ*(y, D_j^0) function, where D_j^0 denotes the domain of j before arc consistency enforcing, returns the successor of y in the ordering of D_j^0 or NIL, if no such element exists. NIL is a value not belonging to any domain and precedes all values in any domain.

```
procedure EXISTy((i, x), j)
  begin
    y ← ResumePoint((i, x), j);
1:  if y ∈ D_j then % y is still in the domain
      return true;
    else
2:      while ((y ← succ(y, D_j^0) and (y ≠ NIL))
          if y ∈ D_j and c_ij(x, y) then
            ResumePoint((i, x), j) ← y;
            return true
          endif;
        return false
    endif
  end
```

Figure 3: Procedure for searching y in REVISE(i, j)

Theorem 2 *The worst case time complexity of AC-3 can be achieved in O(ed^2).*

Proof Here it is helpful to regard the execution of AC-3.1 on a CSP instance as a sequence of calls to EXISTy($(i, x), j$). Consider the time spent on $x \in D_i$ with respect to (i, j). As in theorem 1, an arc (i, j) enters Q at most d times. So, with respect to (i, j), any value $x \in D_i$ will be passed to EXISTy($(i, x), j$) at most d times . Let the complexity of each execution of EXISTy($(i, x), j$) be t_l ($1 \leq l \leq d$). t_l can be considered as 1 if $y \in D_j$ (see line 1 in fig 3) and

otherwise it is s_l which is simply the number of elements in D_j skipped before next y is found (the while loop in line 2). Furthermore, the total time spent on $x \in D_i$ with respect to (i, 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 in EXIST$y((i, x), j)$ the while loop (line 2) will skip an element in D_j at most once with respect to $x \in D_i$. Therefore, $\sum_1^d s_l \leq d$. This gives, $\sum_1^d t_l \leq 2d$. For each arc (i, j), we have to check at most d values in D_i and thus at most O(d^2) time will be spent on checking arc (i, j). Thus, the complexity of the new implementation of AC-3 is O(ed^2) because the number of arcs in constraint graph of the CSP is $2e$. □

The space complexity of AC-3.1 is not as good as the traditional implementation of AC-3. AC-3.1 needs additional space to remember the resumption point of any value with respect to any related constraint. It can be shown that the extra space required is O(ed), which is the same as AC-6.

The same idea behind AC-3.1 applies to path consistency enforcing algorithms. If one pair $(x, y) \in c_{kj}$ is removed, we need to recheck all pairs $(x, *) \in c_{ij}$ with respect to $c_{kj} \circ c_{ik}$ (the composition of c_{ik} and c_{kj}), and $(*, y) \in c_{lk}$ with respect to $c_{jk} \circ c_{lj}$. The resumption point $z \in D_k$ is remembered for any pair (x, y) of any constraint c_{ij} with respect to any intermediate variable k such that $c_{ik}(x, z), c_{kj}(z, y)$ both hold. ResumePoint($(i, x), (j, y), k$) is employed to achieve the above idea in the algorithm in fig 4 which is partially motivated by the algorithm in [Chmeiss and Jegou, 1996].

```
algorithm PC
  begin
    INITIALIZE(Q);
    while Q not empty do
        Select and delete any ((i,x), j) from Q;
        REVISE_PC((i,x), j, Q))
    endwhile
  end
procedure INITIALIZE(Q)
  begin
    for any i, j, k ∈ N do
      for any x ∈ D_i, y ∈ D_j such that c_ij(x,y) do
        if there is no z ∈ D_k such that c_ik(x,z) ∧ c_kj(z,y)
        then
            c_ij(x,y) ← false;
            c_ji(y,x) ← false;
            Q ← Q ∪ {(i,x), j} ∪ {(j,y), i}
        else ResumePoint((i,x), (j,y), k) ← z
  end
```

Figure 4: Algorithm of Path Consistency Enforcing

By using a similar analysis to the proof of theorem 2, we have the following result.

Theorem 3 *The time complexity of the algorithm PC is O(n^3d^3) with space complexity O(n^3d^2).*

The time complexity and space complexity of the PC algorithm here are the same as the best known theoretical results [Singh, 1996].

```
procedure REVISE_PC( (i,x), k, Q)
  begin
    for any j ∈ N, k ≠ i, k ≠ j do
      for any y ∈ D_j such that c_ij(x,y) do
        z ← ResumePoint((i,x), (j,y), k);
        while not ((z ≠ NIL) ∧ c_ik(x,z) ∧ c_kj(z,y)
          do    z ← succ(z, D_k^0);
        if not ((c_ik(x,z) ∧ c_kj(z,y)) then
          Q ← Q ∪ {((i,x), j} ∪ {((j,y), i)}
        else ResumePoint((i,x), (j,y), k)) ← z
      endfor
  end
```

Figure 5: Revision procedure for PC algorithm

4 Preliminary experimental results

In this paper, we present some preliminary experimental results on the efficiency of AC-3. While arc consistency can be applied in the context of search (such as [Bessiere and Regin, 1996]), we focus on the performance statistics of the arc consistency algorithms alone.

The experiments are designed to compare the empirical performance of the new AC-3.1 algorithm with both the classical AC-3.0 algorithm and a state-of-the-art algorithm on a range of CSP instances with different properties.

There have been many experimental studies on the performance of general arc consistency algorithms [Wallace, 1993; Bessiere, 1994; Bessiere *et al.*, 1999]. Here, we adopt the choice of problems used in [Bessiere *et al.*, 1999], namely some random CSPs, Radio Link Frequency Assignment problems (RLFAPs) and the Zebra problem. The zebra problem is discarded as it is too small for benchmarking. Given the experimental results of [Bessiere *et al.*, 1999], AC-6 is chosen as a representative of a state-of-the-art algorithm because of its good timing performance over the problems of concern. In addition, an artificial problem DOMINO is designed to study the worst case performance of AC-3.

Randomly generated problems: As in [Frost *et al.*, 1996], a random CSP instance is characterized by n, d, e and the tightness of each constraint. The *tightness of a constraint* c_{ij} is defined to be $|D_i \times D_j| - |c_{ij}|$, the number of pairs NOT permitted by c_{ij}. A randomly generated CSP in our experiments is represented by a tuple (n, d, e, *tightness*). We use the first 50 instances of each of the following random problems generated using the initial seed 1964 (as in [Bessiere *et al.*, 1999]): (i) $P1$: under constrained CSPs $(150, 50, 500, 1250)$ where all generated instances are already arc consistent; (ii) $P2$: over constrained CSPs $(150, 50, 500, 2350)$ where all generated instances are *inconsistent* in the sense that some domain becomes empty in the process of arc consistency enforcing; and (iii) problems in the phase transition [Gent *et al.*, 1997] $P3$: $(150, 50, 500, 2296)$ and $P4$: $(50, 50, 1225, 2188)$. The $P3$ and $P4$ problems are further separated into the arc consistent instances, labeled as *ac*, which can be made arc consistent at the end of arc consistency enforcing; and inconsistent instances labeled as *inc*. More details on the choices for P1 to P4 can be found in [Bessiere *et al.*, 1999].

RLFAP: The RLFAP [Cabon *et al.*, 1999] is to assign frequencies to communication links to avoid interference. We use the CELAR instances of RLFAP which are real-life problems available at `ftp://ftp.cs.unh.edu/pub/csp/archive/code/benchmarks`.

DOMINO: Informally the DOMINO problem is an undirected constraint graph with a cycle and a trigger constraint. The domains are $D_i = \{1, 2, \cdots, d\}$. The constraints are $C = \{c_{i(i+1)} \mid i < n\} \cup \{c_{1n}\}$ where $c_{1n} = \{(d, d)\} \cup \{(x, x+1) \mid x < d\}$ is called the *trigger constraint* and the other constraints in C are identity relations. A DOMINO problem instance is characterized by two parameters n and d. 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 AC-3 algorithms can only remove one value while AC-6 only does the necessary work. This problem is used to illustrate the differences between AC-3 like algorithms and AC-6. The results explain why arc revision oriented algorithms may not be so bad in the worst case as one might imagine.

		AC-3.0	AC-3.1	AC-6
P1	#ccks	100,010	100,010	100,010
	time(50)	0.65	0.65	1.13
P2	#ccks	494,079	475,443	473,694
	time(50)	1.11	1.12	1.37
P3(ac)	#ccks	2,272,234	787,151	635,671
	time(25)	2.73	1.14	1.18
P3(inc)	#ccks	3,428,680	999,708	744,929
	time(25)	4.31	1.67	1.69
P4(ac)	#ccks	3,427,438	1,327,849	1,022,399
	time(21)	3.75	1.70	1.86
P4(inc)	#ccks	5,970,391	1,842,210	1,236,585
	time(29)	8.99	3.63	3.54

Table 1: Randomly generated problems

RFLAP		AC-3.0	AC-3.1	AC-6
#3	#ccks	615,371	615,371	615,371
	time(20)	1.47	1.70	2.46
#5	#ccks	1,762,565	1,519,017	1,248,801
	time(20)	4.27	3.40	5.61
#8	#ccks	3,575,903	2,920,174	2,685,128
	time(20)	8.11	6.42	8.67
#11	#ccks	971,893	971,893	971,893
	time(20)	2.26	2.55	3.44

Table 2: CELAR RLFAPs

Some details of our implementation of AC-3.1 and AC-3.0 are as follows. We implement domain and related operations by employing a double-linked list. The Q in AC-3 is implemented as a queue of nodes on which arcs incident will be revised [Chmeiss and Jegou, 1996] . A new node will be put at the end of the queue. Constraints in the queue are revised in a FIFO order. The code is written in C++ with *g++*. The experiments are run on a Pentium III 600 processor with Linux.

d		AC-3.0	AC-3.1	AC-6
100	#ccks	17,412,550	1,242,550	747,551
	time(10)	5.94	0.54	0.37
200	#ccks	136,325,150	4,985,150	2,995,151
	time(10)	43.65	2.21	1.17
300	#ccks	456,737,750	11,227,750	6,742,751
	time(10)	142.38	5.52	2.69

Table 3: DOMINO problems

For AC-6, we note that in our experiments, using a single currently supported list of a values is faster than using multiple lists with respect to related constraints proposed in [Bessiere *et al.*, 1999]. This may be one reason why AC-7 is slower than AC-6 in [Bessiere *et al.*, 1999]. Our implementation of AC-6 adopts a single currently supported list.

The performance of arc consistency algorithms here is measured along two dimensions: running time and number of constraint checks (#ccks). A *raw constraint check* tests if a pair (x, y) where $x \in D_i$ and $y \in D_j$ satisfies constraint c_{ij}. In this experiment we assume constraint check is cheap and thus the raw constraint and additional checks (e.g. line 1 in Figs 3) in both AC-3.1 and AC-6 are counted. In the tabulated experiment results, *#ccks* represents the average number of checks on tested instances, and *time(x)* the time in seconds on x instances.

The results for randomly generated problems are listed in Table 1. For the under constrained problems $P1$, AC-3.1 and AC-3.0 have similar running time. No particular slowdown for AC-3.1 is observed. In the over constrained problems $P2$, the performance of AC-3.1 is close to AC-3.0 but some constraint checks are saved. In the hard phase transition problems $P3$ and $P4$, AC-3.1 shows significant improvement in terms of both the number of constraint checks and the running time, and is better than or close to AC-6 in timing.

The results for CELAR RLFAP are given in Table 2. In simple problems, RLFAP#3 and RLFAP#11, which are already arc consistent before the execution of any AC algorithm, no significant slowdown of AC-3.1 over AC-3.0 is observed. For RLFAP#5 and RLFAP#8, AC-3.1 is faster than both AC-3.0 and AC-6 in terms of timing.

The reason why AC-6 takes more time while making less checks can be explained as follows. The main contribution to the slowdown of AC-6 is the maintenance of the currently supported list of each value of all domains. In order to achieve space complexity of $\mathcal{O}(ed)$, when a value in the currently supported list is checked, the space occupied in the list by that value has to be released. Our experiment shows that the overhead of maintaining the list doesn't compensate for the savings from less checks under the assumption that constraint checking is cheap.

The DOMINO problem is designed to show the gap between AC-3 implementations and AC-6. Results in Table 3 show that AC-3.1 is about half the speed of AC-6. This can

be explained by a variation of the proof in section 3, in AC-3.1 the time spent on justifying the validity of a value with respect to a constraint is at most $2d$ while in AC-6 it is at most d. The DOMINO problem also shows that AC-3.0 is at least an order of magnitude slower in time with more constraint checks over AC-3.1 and AC-6.

In summary, our experiments on randomly generated problems and RLFAPs show the new approach to AC-3 has a satisfactory performance on both simple problems and hard problems compared with the traditional view of AC-3 and state-of-the-art algorithms.

5 Related work and discussion

Some related work is the development of general purpose arc consistency algorithms AC-3, AC-4, AC-6 , AC-7 and the work of [Wallace, 1993]. We summarize previous algorithms before discussing how this paper gives an insight into AC-3 as compared with the other algorithms.

An arc consistency algorithm can be classified by its method 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. Value oriented propagation originates from AC-4 and its underlying computation model is the value based constraint graph.

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

Thus a more rigorous 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 those values related to it will be checked. Thus it is more fine grained than arc oriented propagation. Algorithms working with variable and value based constraint graph are also called *coarse grained algorithms* and *fine grained algorithms* respectively. An immediate observation is that compared with variable based constraint graph, time complexity analysis in value based constraint graph is straightforward.

Given a computation model of propagation, the algorithms differ in the implementation details. For variable based constraint graph, AC-3 [Mackworth, 1977] is an "open implementation". The approach in [Mackworth and Freuder, 1985] can be regarded as a realized implementation. The new view of AC-3 presented in this paper can be thought of as another implementation with optimal worst case complexity. Our new approach simply remembers the result obtained in previous revision of an arc while in the old one, the choice is to be lazy, forgetting previous computation. Other approaches to improving the space complexity of this model is [Chmeiss and Jegou, 1996]. For value based constraint graph, AC-4 is the first implementation and AC-6 is a lazy version of AC-4. AC-7 is based on AC-6 and it exploits the *bidirectional property* that given c_{ij}, c_{ji} and $x \in D_i, y \in D_j$, $c_{ij}(x, y)$ if and only if $c_{ji}(y, x)$.

Another aspect is the general properties or knowledge of a CSP which can be isolated from a specific arc consistency enforcing algorithm. Examples are AC-7 and AC-inference. We note that the idea of *metaknowledge* [Bessiere *et al.*, 1999] can be applied to algorithms of both computing models. For example, in terms of the number of *raw* constraint checks, the bidirectionality can be employed in coarse grained algorithm, eg. in [Gaschnig, 1978], however it may not be fully exploited under the variable based constraint graph model. Other propagation heuristics [Wallace, 1992] such as propagating deletion first [Bessiere *et al.*, 1999] are also applicable to algorithms of both models. This is another reason why we did not include AC-7 in our experimental comparison.

We have now a clear picture on the relationship between the new approach to AC-3 and other algorithms. AC-3.1 and AC-6 are methodologically different. From a technical perspective, the time complexity analysis of the new AC-3 is different from that of AC-6 where the worst case time complexity analysis is straightforward. The point of commonality between the new AC-3 and AC-6 is that they face the same problem: the domain may shrink in the process of arc consistency enforcing and thus the recorded information may not be always *correct*. This makes some portions of the new implementation of the AC-3.1 similar to AC-6. We remark that the proof technique in the traditional view of AC-3 does not directly lead to the new AC-3 and its complexity results.

The number of raw constraint checks is also used to evaluate practical efficiency of CSP algorithms. In theory, applying bidirectionality to all algorithms will result in a decrease of raw constraint checks. However, if the cost of raw constraint check is cheap, the overhead of using bidirectionality may not be compensated by its savings as demonstrated by [Bessiere *et al.*, 1999].

It can also be shown that if the same ordering of variables and values are processed, AC-3.1 and the classical AC-6 have the same number of raw constraint checks. AC-3.0 and AC-4 will make no less *raw* constraint checks than AC-3.1 and AC-6 respectively.

AC-4 does not perform well in practice [Wallace, 1993; Bessiere *et al.*, 1999] 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 being simpler 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. However, the value based constraint graph induced from AC-4 provides a convenient and accurate tool for studying arc consistency.

Given that both variable and value based constraint graph can lead to worst case optimal algorithms, we consider their strength on some special constraints: functional, monotonic and anti-functional. For more details, see [Van Hentenryck *et al.*, 1992] and [Zhang and Yap, 2000].

For coarse grained algorithms, it can be shown that for *monotonic* and *anti-monotonic* constraints arc consistency can be done with complexity of $\mathcal{O}(ed)$ (eg. using our new view of AC-3). With fine grained algorithms, both AC-4 and AC-6 can deal with *functional* constraints. 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 algorithm to

monotonic constraints.

In summary, there are coarse grained and fine grained algorithms which are competitive given their optimal worst case complexity and good empirical performance under varying conditions. In order to further improve the efficiency of arc consistency enforcing, more properties (both general like bidirectionality and special like monotonicity) of constraints and heuristics are desirable.

[Wallace, 1993] gives detailed experiments comparing the efficiency of AC-3 and AC-4. Our work complements this in the sense that with the new implementation, AC-3 now has optimal worst case time complexity.

6 Conclusion

This paper presents a natural implementation of AC-3 whose complexity is better than the traditional understanding. AC-3 was not previously known to have worst case optimal time complexity even though it is known to be efficient. Our new implementation brings AC-3 to $\mathcal{O}(ed^2)$ on par with the other optimal worst case time complexity algorithms. Techniques in the new implementation can also be used with path consistency algorithms.

While worst case time complexity gives us the upper bound on the time complexity, in practice, the running time and number of constraint checks for various CSP instances are the prime consideration. Our preliminary experiments show that the new implementation significantly reduces the number of constraint checks and the running time of the traditional one on hard arc consistency problems. Furthermore, the running time of AC-3.1 is competitive with the known best algorithms based on the benchmarks from the experiment results in [Bessiere *et al.*, 1999]. Further experiments are planed to have a better comparison with typical algorithms. We believe that based on the CELAR instances, the new approach to AC-3 leads to a more robust AC algorithm for real world problems than other algorithms.

We also show how the new AC-3 leads to a new algorithm for path consistency. We conjecture from the results of [Chmeiss and Jegou, 1996] that this algorithm can give a practical implementation for path consistency.

For future work, we want to examine the new AC-3 in maintaining arc consistency during search.

7 Acknowledgment

We are grateful to Christian Bessiere for providing benchmarks and discussion. We acknowledge the generosity of the French Centre d'Electronique de l'Armement for providing the CELAR benchmarks.

References

[Bessiere, 1994] C. Bessiere 1994. Arc-consistency and arc-consistency again, *Art. Int.*65 (1994) 179–190.

[Bessiere *et al.*, 1999] C. Bessiere, E. C. Freuder and J. Regin 1999. Using constraint metaknowledge to reduce arc consistency computation, *Art. Int.*107 (1999) 125–148.

[Bessiere and Regin, 1996] C. Bessiere and J. Regin 1996. MAC and combined heuristics: two reasons to forsake FC(and CBJ?) on hard problems, *Proc. of Principles and Practice of Constraint Programming*, Cambridge, MA. pp. 61–75.

[Cabon *et al.*, 1999] B. Cabon, S. de Givry, L. Lobjois, T. Schiex and J.P. Warners 1999. Radio link frequency assignment, *Constraints* 4(1) (1999) 79–89.

[Chmeiss and Jegou, 1996] A. Chmeiss and P. Jegou 1996. Path-Consistency: When Space Misses Time, *Proc. of AAAI-96*, USA: AAAI press.

[Frost *et al.*, 1996] D. Frost, C. Bessiere, R. Dechter and J. C. Regin 1996. Random uniform CSP generators, `http://www.lirmm.fr/~bessiere/generator.html`.

[Gaschnig, 1978] J. Gaschnig 1978. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisfying assignment problems, *Proc. of CCSCSI-78*, 1978.

[Gent *et al.*, 1997] J. P. Gent, E. Maclntyre, P. Prosser, P. Shar and T. Walsh 1997. The constrainedness of arc consistency, *Proc. of Principles and Practice of Constraint Programming 1997* Cambridge, MA, 1996, pp. 327–340.

[Van Hentenryck *et al.*, 1992] P. van Hentenryck, Y. Deville, and C. M. Teng 1992. A Generic Arc-Consistency Algorithm and its Specializations, *Art. Int.*58 (1992) 291–321.

[Mackworth, 1977] A. K. Mackworth 1977. Consistency in Networks of Relations,, *Art. Int.*8(1) (1977) 118–126.

[Mackworth and Freuder, 1985] A. K. Mackworth and E. C. Freuder 1985. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems,, *Art. Int.*25 (1985) 65–74.

[Mohr and Henderson, 1986] R. Mohr and T. C. Henderson 1986. Arc and Path Consistency Revisited, *Art. Int.*28 (1986) 225–233.

[Montanari, 1974] U. Montanari 1974. Networks of Constraints: Fundamental Properties and Applications, *Information Science* 7(2) (1974) 95–132.

[Singh, 1996] M. Singh 1996. Path consistency revisited, *Int. Journal on Art. Intelligence Tools* 5(1&2) (1996) 127–141.

[Wallace, 1993] Richard J. Wallace 1993. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs , *Proc. of IJCAI-93* Chambery, France, 1993.

[Wallace, 1992] R. J. Wallace and E. Freud 1992. Ordering heuristics for arc consistency algorithms , *Proc, of Canadian Conference on AI* Vancouver, BC, 1992, pp. 163-169.

[Zhang and Yap, 2000] Y. Zhang, R. H. C. Yap 2000. Arc consistency on n-ary monotonic and linear constraints, *Proc. of Principles and Practice of Constraint Programming* Singapore, 2000, pp. 470–483 .