AC-5*: An Improved AC-5 and Its Specializations

Bing Liu

Department of Information Systems and Computer Science
National University of Singapore
Lower Kent Ridge Road, Singapore 119260
liub@iscs.nus.sg

Abstract

Many general and specific arc consistency algorithms have been produced in the past for solving Constraint Satisfaction Problems (CSP). The important general algorithms are AC-3, AC-4, AC-5 and AC-6. AC-5 is also a generic algorithm. It can be reduced to AC-3, AC-4 and AC-6. Specific algorithms are efficient specializations of the general ones for specific constraints. Functional, anti-functional and monotonic constraints are three important classes of specific constraints. AC-5 has been specialized to produce an O(*ed*) algorithm (in time) for these classes of constraints. However, this specialization does not reduce the space requirement. In practical applications, both time and space requirements are important criteria in choosing an algorithm.

This paper makes two contributions. First, we propose an improved generic arc consistency algorithm, called AC-5*. It can be specialized to reduce both time and space complexities. Second, we present a more efficient technique for handling an important subclass of functional constraints, namely *increasing functional constraints*. This technique is significant because in practice almost all functional constraints are actually *increasing functional constraints*.

1. Introduction

Arc consistency techniques are the key techniques for solving Constraint Satisfaction Problems (CSP). Past research has produced many general and specific arc consistency algorithms. The most important general algorithms are AC-3 [15], AC-4 [18], AC-5 [25] and AC-6 [1]. AC-5 is also a generic algorithm, and it can be reduced to AC-3, AC-4 and AC-6. Among these algorithms, AC-3 has the optimal space complexity of $O(e + nd)$ [18, 15], while AC-4 and AC-6 have the optimal time complexity of $O(ed^2)$, where n is the number of variables, e is the number of arcs, and d is the size of the largest domain.

Specific algorithms are efficient specializations of the general ones for specific constraints [6, 25]. These methods typically exploit the semantics of individual constraints in consistency check, and they are more widely used in practical applications than the general ones. Functional (FC), anti-functional (ATFC) and monotonic constraints (MC) are three important classes of specific constraints.

In recent years, the CSP model has been implemented in constraint programming languages, such as CHIP [24], Charme [3], Ilog Solver [10], etc., for solving practical combinatorial problems, such as scheduling, sequencing and resource allocations [7, 24, 10]. The basic constraints used in these languages are special cases of FC, ATFC and MC.

In [25], AC-5 is specialized to produce an algorithm running in O(ed) (which is the optimal time complexity [25]) for these three classes of constraints. However, due to its fixed queue element representation, it cannot be specialized to reduce the space complexity. It still requires O($ed + nd$) space, which is worse than that of AC-3.

In practical applications, both time and space requirements are important in selecting an algorithm. Then, the question is: can we have a generic algorithm that can be specialized for these classes of constraints to achieve both the optimal time complexity of O(ed) and the optimal space complexity of O($e + nd$) (at the same time)? The answer is almost yes.

This paper makes two contributions. First, we propose an improved (over AC-5) generic arc consistency algorithm, called AC-5*. It is parametrized on two procedures and a queue element representation. Its key feature is that it can be specialized to produce efficient specific algorithms in terms of both time complexity and space complexity. It can also be reduced to AC-3, AC-4, AC-5 and AC-6 by proper implementation of the two procedures and the queue element.

Second, we introduce a more efficient technique for checking an important sub-class of functional constraints, namely, *increasing function constraints* (IFCs). In this method, IFCs need to be checked only once, rather than many times as in a typical consistency check process. This technique produces two important results:

1. Although the new technique is still O(ed) (in time) the same as that of AC-5, experiment results show that it outperforms the existing techniques substantially.
2. Together with AC-5*, we can produce an arc consistency algorithm for IFCs, ATFCs and MCs that has both the optimal time and the optimal space complexities. This cannot be achieved with existing techniques.

These results are significant in practice because the basic functional constraint used in current constraint programming languages [24, 3, 10] is actually an IFC.

The paper is organized as follows. Section 2 gives some definitions and conventions. Section 3 describes AC-5*. Section 4 discusses a specific implementation of AC-5*, where our new method for IFCs is presented. Section 5 shows the test results. Section 6 discusses the related work and Section 7 concludes the paper.

2. Preliminaries

A CSP is defined as follows: (1) variables - a finite set of n variables $\{1, 2, ..., n\}$, and (2) domains - each variable i takes its values from an associated finite domain D_i, and (3) constraints - a set of binary constraints C between variables. We assume, for simplicity, that there is only one constraint, denoted as C_{ij}, between a pair of variables i and j.

A graph G can be associated with a CSP, where each node represents a variable i, and each edge between two variables i and j represents a constraint, which is expressed as two directed arcs, (i, j) and (j, i). We denote by e the number of arcs in G, by d the size of the largest domain, and by $arc(G)$ the set of arcs in G.

We now recall the standard definitions of arc consistency for an arc and a graph.

Definition 1. An arc $(i, j) \in arc(G)$ is arc consistent with respect to D_i and D_j iff for all $v \in D_i$, there exists $w \in D_j$ such that $C_{ij}(v, w)$ holds.

Definition 2. A CSP is arc consistent iff for all $(i, j) \in arc(G)$, (i, j) is arc consistent with respect to D_i and D_j.

Our techniques require a total ordering on the domain. It is defined as follows:

Definition 3. A domain $D_i = \{v_1, ..., v_a\}$ is totally ordered iff $v_k < v_{k+1}$.

We now define functional (FC) and increasing functional (IFC), anti-functional (ATFC), and monotonic (MC) constraints (see also [25, 13]).

Definition 4. Given two variables i and j, we denote $i \rightarrow j$ iff for all $v \in D_i$ there exists at most one $w \in D_j$ such that $C_{ij}(v, w)$ holds. If it exists, then $w = f_{ij}(v)$.
A constraint is functional iff $i \rightarrow j$ and $j \rightarrow i$.

Definition 5. Given two variables, we denote $i \uparrow j$ iff (1) $i \rightarrow j$, and (2) for all $u, v \in D_i$ such that both $f_{ij}(u)$ and $f_{ij}(v)$ exist in D_j then $u < v$ implies $f_{ij}(u) < f_{ij}(v)$.
Observe that if $i \uparrow j$ (or equivalently $j \uparrow i$) then the constraint C_{ij} must be functional, and we call such a constraint *increasing functional constraint*.

Definition 6. A constraint C_{ij} is anti-functional iff $\neg C_{ij}$ is functional.

Definition 7. A constraint C_{ij} is monotonic iff there exists a total ordering on D_i and D_j such that, for all values $v \in D_i$ and $w \in D_j$, $C_{ij}(v, w)$ holds implies $C_{ij}(v', w')$ holds for all values $v' \in D_i$ and $w' \in D_j$ such that $v' \leq v$ and $w' \geq w$.

An example of an IFC is $x = y + 5$. An example of an ATFC is $x \neq y$, and an example of a MC is $x \leq y + 1$.

The basic constraints in the current constraint programming languages are special cases of FC, ATFC and MC [25, 24, 3, 10]. In fact, they are binary equations ($aX = bY + c$), inequalities ($aX \leq bY + c$ and $aX \geq bY + c$) and disequations ($aX \neq bY$), where a, b and c are constants and $a, b \geq 0$. Domain values are natural numbers. According to Definition 5, the equation $aX = bY + c$, which is a special case of FC, is also a special case of IFC.

Most of the earlier algorithms do not use the semantics of constraints to achieve better efficiency. AC-5 is different as the implementation of its two procedures ArcCons and LocalArcCons are left open. This means that for different constraints different procedures could be used. ArcCons checks an arc when it is first encountered, while LocalArcCons rechecks it if its consistency is broken. [25] provides the special ArcCons and LocalArcCons procedures for checking FCs, ATFCs and MCs in time O(ed). In this paper, we call these two procedures the initial check procedure and the recheck procedure respectively for intuitive reasons. This separation is important because the methods for recheck for certain constraints (e.g., FCs) may be different from their initial check methods.

3. AC-5* Algorithm

Since AC-5* is intended to improve AC-5, before presenting AC-5*, we make some observations about AC-5. Take note that all arc consistency algorithms work with a queue Q containing elements to be rechecked.

- The key feature of AC-5 is that its Q contains elements $((i, j), w)$, where (i, j) is an arc and w is a value that has been removed from D_j and justifies the need to recheck (i, j).
- Due to this queue element representation, AC-5 achieves the time complexity of O(ed), and the space complexity of O($ed + nd$) (because the size of Q is O(ed)) for FCs, ATFCs and MCs. The space complexity is worse than that of AC-3, which is O($e + nd$) (because the Q size of AC-3 is only O(e)).
- From AC-5 and its specializations, it can be observed that for ATFCs and MCs, value w is not used in its LocalArcCons implementations. Only the minimum and maximum domain values and the domain size are employed. Using w for these constraints (or arcs) is a waste of space. However, for FCs, using w is important. If we know a value

w that has been removed from *j*, we can remove its corresponding value in *i* in constant time. This results in an O(*ed*) algorithm (in time). If *w* is not known, only an O(ed^2) algorithm (in time) can be achieved for FCs. In general, we can say that knowing *w* may be useful for certain classes of constraints, but not for all.

We now present AC-5* (Figure 1). Like AC-5, AC-5* also has two steps, the initial check step (line 2 and 3), and the recheck step (line 4-8). In initial check, each arc in *arc*(*G*) is checked once to enforce arc consistency. In recheck, *ReCheck* is applied to each element of the queue *Q*, possibly generating new elements in *Q*.

```
    Algorithm AC-5*
      begin
1       Q ← ∅;
2       for each (i, j) ∈ arc(G) do
3         InitialCheck((i, j), Q)
4       while Q not empty do
5         begin
6           delete an element ELT from Q;
7           ReCheck(ELT, Q)
8         end
      end
```

Figure 1. AC-5* algorithm

The algorithm is parametrized on two procedures, *InitialCheck* and *ReCheck*, and the representation of the queue element *ELT*. Their implementations are left open. While the novelty of AC-5 as described in [25] is that its queue *Q* contains elements ((*i*, *j*), *w*), the novelties of AC-5* are (1) that it leaves the queue element representation open, and (2) that the elements in *Q* do not have to have the same representation. Since different queue elements can have different representations in the same algorithm, it allows AC-5* to be specialized to save space in addition to time. This is the key difference between AC-5* and AC-5. In fact, all the previous algorithms have fixed queue element representations.

In AC-5*, we classify arcs (or constraints) into two classes:

1. Arcs that require *w* to produce an efficient algorithm, e.g., FC arcs. We call this set of arcs *W_arc*s.
2. Arcs that do not require *w* to produce an efficient algorithm, e.g., ATFC and MC arcs. We call this set of arcs *NW_arc*s.

We are now in the position to present the specifications for the two abstract procedures *InitialCheck* and *ReCheck*, and the queue element *ELT*. In AC-5*, a queue element could either be (*i*, *j*) or ((*i*, *j*), *w*). If arc (*i*, *j*) is a *W_arc*, ((*i*, *j*), *w*) is used. If arc (*i*, *j*) is a *NW_arc*, (*i*, *j*) is used. Figure 2 shows *InitialCheck* and *ReCheck* (see also [25]).[1] Notice that there are two *ReCheck* procedures (I and II) because of the two queue element representations. *Enqueue* is given in Figure 3.[2] *Remove* removes the values in Δ from D_i.

It is clear that *W_arc*s can also be handled in the same way as *NW_arc*s (and vice versa) except that it is less efficient. As for which implementation to use, it depends on the problem. For *W_arc*s, it is basically a time and space tradeoff.

1 In the second *ReCheck* in Figure 2, Δ_2 may not be useful in AC-5* because if Δ_2 can be achieved, it means that knowing *w* could be unnecessary. Then, (*i*, *j*) should be classified as a *NW_arc*, rather than a *W_arc*.

2 Strictly speaking, arc (*j*, *i*) is not enqueued when (*i*, *j*) has been checked. This optimization could be added in AC-5* by including *j* as an argument in *Enqueue* and adding the conditions $k \neq j$ and $m \neq j$ to its definition.

AC-5* may be seen as an integration of AC-3 and AC-5 because of its two *ELT* representations. Its correctness follows directly from that of AC-5 and AC-3. All the general properties of AC-5 still apply, even in the case of space complexity because w may be required by every constraint.

With proper implementation of the three items, AC-5* can be reduced to AC-3, AC-4, AC-5, and AC-6. In these algorithms, *W_arc*s and *NW_arc*s are not distinguished.

- AC-3 is clearly a particular case of AC-5* where the implementations of *InitialCheck* and *ReCheck* are the same. The queue element *ELT* is simply (i, j).

Procedure InitialCheck (**in** (i, j), **inout** Q)
  begin
    $\Delta \leftarrow \{v \in D_i \mid \forall w \in D_j: \neg C_{ij}(v, w)\}$;
    Remove(Δ, i);
    Enqueue(i, Δ, Q)
  end

I **Procedure** ReCheck (**in** (i, j), **inout** Q)
  begin
    $\Delta \leftarrow \{v \in D_i \mid \forall w \in D_j: \neg C_{ij}(v, w)\}$;
    Remove(Δ, i);
    Enqueue(i, Δ, Q)
  end

II **Procedure** ReCheck(**in** $((i, j), w)$, **inout** Q)
  begin
    $\Delta_1 \subseteq \Delta \subseteq \Delta_2$ with $\Delta_1 \leftarrow \{v \in D_i \mid C_{ij}(v, w) \text{ and } \forall w' \in D_j: \neg C_{ij}(v, w')\}$
    $\Delta_2 \leftarrow \{v \in D_i \mid \forall w' \in D_j: \neg C_{ij}(v, w')\}$
    Remove(Δ, i);
    Enqueue(i, Δ, Q)
  end

Figure 2. *InitialCheck* and *ReCheck* procedures

Procedure Enqueue (**in** i, Δ, **inout** Q}
  if $\Delta \neq \emptyset$ **then begin**
    $Q \leftarrow Q \cup \{((k, i), v) \mid (k, i) \in W\_arc(G) \text{ and } v \in \Delta\}$;
    $Q \leftarrow Q \cup \{(m, i) \mid (m, i) \in NW\_arc(G)\}$
  end

Figure 3. *Enqueue* procedure

- AC-4 is also a particular case of AC-5* where *ELT* is $((i, j), w)$, but *ReCheck*'s implementation does not use i.[3] *ReCheck* prunes inconsistent values by maintaining a data structure S of size $O(ed^2)$. *InitialCheck* initializes S.
- AC-5 is also a special case of AC-5* where *ELT* is $((i, j), w)$. Note that AC-5* puts *Enqueue* and *Remove* in *InitialCheck* and *ReCheck*. This gives more flexibility in specializing the two procedures.
- Like AC-4, AC-6 is again a special case of AC-5*.[4] Unlike AC-4, *ReCheck* prunes inconsistent values by finding successive value supports and maintaining this information in a data structure S of size $O(ed)$. *InitialCheck* initializes S.

[3, 4] Using i in *ELT* is also a waste here as AC-4 and AC-6 uses only (j, w). This idea is not explored further in this paper because when using AC-4 or AC-6 there is no need to refer to AC-5 or AC-5*.

Since O(ed^2) is the optimal time complexity, which is achieved by AC-4 and AC-6, there is no way to reduce it other than considering particular classes of constraints.

Like AC-5, AC-5* can also be easily specialized to produce an O(ed) algorithm (in time) for FCs, ATFCs and MCs. The space complexity is, however, reduced.

- For ATFCs and MCs, the space complexity is O(e + nd) since we can use (i, j) as *ELT*, instead of O(ed + nd) as AC-5.
- For FCs, the space complexity is still O(ed + nd), which is the same as AC-5, since we use ((i, j) w) as *ELT*.

The key advantage of AC-5* is that it can achieve the space complexity of O(e + nd) for ATFCs and MCs. We will not present all the specific procedures for *InitialCheck* and *ReCheck* here as they can be easily obtained by modifying those in AC-5.

In the next section, we present a new technique for IFCs that does not need w to produce an O(ed) algorithm in time. This means that the space complexity becomes O(e + nd) for IFCs. Note that this new technique is not tied to AC-5*, although it is described in the AC-5* framework. It may be embedded in other general arc consistency algorithms.

4. An Implementation of AC-5*

We may view AC-3, AC-4, AC-5 and AC-6 as special implementations of AC-5*. In this section, we present another implementation with its specializations. For easy reference, we call this implementation, AC-5*$_3$, as it is similar to AC-3. The main difference is that AC-3 does not distinguish initial check and recheck.

Here, we also describe our new technique for checking IFCs. With this technique, we can achieve the optimal space complexity of O(e + nd) and the optimal time complexity of O(ed) for IFCs, ATFCs, and MCs. Although the time complexity for IFCs is still the same as that in AC-5, test results show that the new technique runs substantially faster than the existing ones. In this section, we will not describe the specialized procedures for ATFCs and MCs as they can be easily obtained by modifying those in AC-5.

In AC-5*$_3$, the queue element *ELT* is only (i, j). ((i, j), w) is not used because for these three classes of constraints, having w does not help. Thus, all the arcs are assumed to be *NW_arc*s (or simply *arc*s). Hence, the space complexity is O(e + nd) as AC-3.

Due to our new method for IFCs, in line 2 of AC-5*, each IFC in *arc*(G) is expressed as one arc (or edge), either (i, j) or (j, i), rather than two arcs. This simplifies the technique.

4.1 Domain representation

The domain data structures for AC-5*$_3$ are shown in Figure 4. This representation is similar to that in AC-5. However, there are some key differences, which will be made clear from the description below. AC-5's domain data structure cannot be used in our new technique for IFCs. Like AC-5 and AC-6, we assume a total ordering on the domain.

Figure 4(A) shows the top level structure. The field *merge* tells whether this domain has merged with another domain. Its meaning will become clear later. The field *element* is a set of pairs (v, *loc*) organized as a hash table on key v, where $v \in V$, and *loc* is a structure (Figure 4(B)) with its field *index* holding the array index of v in D_i*.values* (when a value is not even in the original D_i, D_i*.element*(v) will give 0). This field is different from that in AC-5, where v directly points to the index rather than a structure *loc* that indirectly points to the same index. This change is important for the new technique for IFCs. The field *values* contains the real values of the domain. The *info* field gives all the information about the current domain. Its data structure is shown in Figure 4(C). This is also different from AC-5 as AC-5 keeps all the information in the domain data structure in Figure 4(A). This modification is crucial to our new technique for IFCs.

Regarding the *info* structure, the field *size* gives the domain size. The fields *min* and *max* are used to access the minimum and maximum values in the domain. The fields *pred* and *succ* allow accessing in constant time the successor and predecessor of a value in the domain. This representation allows the algorithm to reason about array indices rather than values. These fields are the same as those in AC-5. The extra field is *arcs* storing the arcs (which are kept elsewhere in AC-5) related to the variable except those IFC arcs because our new technique checks them only once. Then, there is no need to store them.

Let $V = \{v_1, ..., v_a\}$ with $v_k < v_{k+1}$ be the original domain of i;

$D_i = \{v_{p_1}, ..., v_{p_m}\} \subseteq V$ with $p_k < p_{k+1}$ and m > 0

Syntax

$D_i.merge$: integer $\in \{0, ..., n\}$, where n is the number of variables in the CSP

$D_i.element$: set of pairs (v, loc) with $v \in V$ and loc is a structure, organized as a hash table on key v.

$D_i.values$: array [1..a] of elements $\in V$

$D_i.info$: an *info* structure

Semantics

$D_i.merge \neq 0$, if this domain has been merged with another domain

$= 0$, otherwise

$loc.index(D_i.element(v)) = p$ (with $v = v_p$), if $v \in D_i$

$= 0$, otherwise

$D_i.values[p] = v_p$

$D_i.info$ points to an *info* structure (see below)

(**A**). Domain data structure

Syntax

$loc.index$: integer

Semantics

$loc.index = p \in \{0, ..., a\}$

(**B**). *loc* data structure

Syntax

$info.size$: integer

$info.min$: integer $\in \{1, ..., a\}$

$info.max$: integer $\in \{1, ..., a\}$

$info.succ$: array [1..a] of integers $\in \{1, ..., a\}$

$info.pred$: array [1..a] of integers $\in \{1, ..., a\}$

$info.arcs$: a set of relevant arcs.

Semantics

$info.size = m$

$info.min = p_1$

$info.max = p_m$

$info.succ[p_k] = p_{k+1}$ $(1 \leq k < m)$, $info.succ[p_m] = +\infty$

$info.pred[p_{k+1}] = p_k$ $(1 \leq k < m)$, $info.pred[p_1] = -\infty$

$info.arcs = \{(l, i) \mid (l, i) \in arc(G)$ and (l, i) is not an IFC$\}$

(**C**). *info* data structure

Figure 4. Domain data structures

4.2 Check IFCs

We now present the new technique for checking IFCs in AC-5*$_3$. In a normal process, a constraint needs to be checked many times to maintain consistency. Recheck is necessary when its previously established consistency is broken by other constraints. In the new method, IFCs only need to be checked once. The main idea is to exploit the fact that consistent values in the domains of an IFC are one-to-one correspondent and in an increasing order. So, in initial check, we can merge the two domains by making them share some key information. Later on, domain change of one variable will be felt automatically by the other. In this way, rechecks of the constraint can be avoided.

Before presenting the detailed technique, we distinguish two types of CSPs, static CSPs and incremental CSPs. A static CSP refers to a CSP whose constraint network is completely available before arc consistency is enforced. An incremental CSP refers to a CSP whose variables and constraints may coming incrementally during the problem solving process. Most real-life CSPs are incremental ones.

The new technique has different properties for the two types of CSPs. For a static CSP, we can arrange the sequence of the arcs to be checked so that all the IFCs can be merged in initial check. Then, rechecking them will not be needed. However, for an incremental CSP, such ordering may not be possible. Then, the new technique does not guarantee that every IFC will need only one check. We shall discuss this further in Section 4.2.2.

4.2.1 Procedure *InitialCheck* for IFCs

We now present *InitialCheck* for IFCs (Figure 5). This procedure can be used for both static and incremental CSPs. Two sub-procedures are used. The first one is *MergeCheck* (Figure 6), which merges the two variables in (i, j). The second one is *NonMergeCheck* (Figure 7), which does not merge the two variables.

Procedure InitialCheck(**in** (i, j), **inout** Q)
  if $D_j.merge = 0$ **then** $MergeCheck((i, j), Q)$
  elseif $D_i.merge = 0$ **then** $MergeCheck((j, i), Q)$
  else $NonMergeCheck((i, j), Q)$

Figure 5. *InitialCheck* for IFCs

InitialCheck states that if $D_j.merge = 0$ or $D_i.merge = 0$, then merge i and j, otherwise do not merge them. Take note that not both $D_j.merge$ and $D_i.merge$ have to be 0 before they can be merged. This means that a variable can be merged with any number of other variables. When $D_j.merge = 0$, we merge j to i and when $D_i.merge = 0$, we merge i to j.

MergeCheck merges D_i and D_j by making them share the same *info* and some information in their *element* fields. For example, suppose we have the variables X and Y. X has the domain {1 2 4 6 7 9} and Y has the domain {5 8 9 10 11 12 13}. The IFC is $Y = X + 1$. The initial domains for X and Y are as follows (only two fields are shown):

$D_x.values$	=	1	2	4	6	7	9	
$loc.index(D_x.element(v))$	=	1	2	3	4	5	6	
$D_y.values$	=	5	8	9	10	11	12	13
$loc.index(D_y.element(w))$	=	1	2	3	4	5	6	7

After initial check, the domains become:

$D_x.values$	=	1	2	4	6	7	9
$loc.index(D_x.element(v))$	=	0	0	3	0	5	6
$D_y.values$	=	u	u	5	u	8	10
$loc.index(D_y.element(f_{xy}(v)))$	=			3		5	6

```
   Procedure MergeCheck(in (i, j), inout Q)
   begin
1     DELETE_i ← false; DELETE_j ← false;
2     newValues ← make an array of the same size as D_i.values;
3     for each v (= D_i.values[index_i]) ∈ D_i do
4         if f_ij(v) ∉ D_j then
5            begin
6               delete v from D_i;
7               DELETE_i ← true
8            end
9         else  begin
10                 newValues[index_i] ← f_ij(v);
11                 D_j.element(f_ij(v)) ← D_i.element(v);
12                 delete f_ij(v) from D_j without modifying D_j.element
13              end
14    if DELETE_i then Q ← Q ∪ {arc | arc ∈ D_i.info.arcs};
15    for each w ∈ D_j do
16       begin
17          loc.index(D_j.element(w)) ← 0;
18          DELETE_j ← true
19       end
20    if DELETE_j then Q ← Q ∪ {arc | arc ∈ D_j.info.arcs};
21    if D_i.merge = 0 then D_i.merge ← i;
22    D_j.merge ← D_i.merge;
23    D_i.info.arcs ← D_i.info.arcs ∪ D_j.info.arcs;
24    D_j.values ← newValues;
25    D_j.info ← D_i.info
   end
```

Figure 6. Procedure *MergeCheck* for IFCs

Note that u stands for undefined, and consistent values in the new $D_y.values$ are still in the original order. After initial check, for each pair of matching values $v \in D_x$ and $f_{xy}(v) \in D_y$, $D_x.element(v)$ and $D_y.element(f_{xy}(v))$ have the same *loc*. For those values in the initial D_y but not in the new D_y, their *loc.index* all have 0.

Here are the key points to show the correctness of *MergeCheck*.

- The array *newValues* (line 2) created with the same size as $D_i.values$ is used to store D_i's corresponding values in D_j. Since IFCs are functional, the number of values in D_j after the initial check cannot be more than the size of $D_i.values$. In line 10, whenever a value $v \in D_i$ has a matching value $f_{ij}(v) \in D_j$, $f_{ij}(v)$ is stored in *newValues* in the $index_i$th cell. Thus, after line 13, *newValues* has all the consistent values in D_j with regard to D_i. Consistent values in D_i and *newValues* also have the same array indices. Due to the $i \uparrow j$ property, values in *newValues* are still totally ordered.

  Line 12 deletes those values in D_j with matching values in D_i without modifying $D_j.element$. This operation can be done in $D_j.succ$ that is subsequently replaced in line 25. So, deleting those matching values in D_j does no harm. Line 15-19 removes those values in D_j having no matching values in D_i by updating only D_j's hash table. Other information on D_j requires no updating as it is subsequently replaced (line 25).

  In line 24, the original $D_j.values$ is replaced with *newValues*. It is clear no consistent value is lost and no inconsistent value is kept in the new $D_j.values$. All the inconsistent values in D_i are removed in line 6.

- Line 11 makes $D_j.element(f_{ij}(v))$ points to the same *loc* as $D_i.element(v)$ (as $v \in D_i$ and $f_{ij}(v) \in D_j$ are consistent). This ensures that when a value x is deleted from D_i (or D_j) by setting $loc.index(D_i.element(x))$ (or $loc.index(D_j.element(x))$ to 0 in the later process, the corresponding value in D_j (or D_i) is also deleted.

  Line 23 concatenates the related arcs of i and j and assigns it to $D_i.info.arcs$. This and the operation in line 25 ensure that when D_i (or D_j) is modified later (after initial check), related arcs of both i and j are activated for recheck.

  Line 25 makes D_j share the same *info* with D_i. It is clear that according to the definition of IFC, all the information in $D_i.info$ and $D_j.info$ should be the same after the initial check. The merging of the domains is complete. Later on, anything happens to D_i (or D_j), D_j (or D_i) feels it automatically.

NonMergeCheck is given in Figure 7, which uses the sub-procedure *Check* (also in Figure 7). It consists of two parts, the first part (line 2-17) is used when $D_i.merge = D_j.merge$, and the second part (line 18-23) is used otherwise.

```
   Procedure NonMergeCheck(in (i, j), inout Q)
1     if Di.merge = Dj.merge then
2        begin
3           DELETE ← false;
4           for each v ∈ Di do
5               if fij(v) ∉ Dj then
6                  begin
7                     delete v from Di;
8                     DELETE ← true;
9                  end
10              elseif Dj.element(fij(v)) and Di.element(v) do not share the same loc then
11                    begin
12                       delete v from Di;
13                       delete fij(v) from Dj;
14                       DELETE ← true;
15                    end
16          if DELETE then Q ← Q ∪ {arc | arc ∈ Di.info.arcs};
17       end
18    else  begin
19             if Check((i, j)) then Q ← Q ∪ {arc | arc ∈ Di.info.arcs};
20             if Check((j, i)) then Q ← Q ∪ {arc | arc ∈ Dj.info.arcs};
21             Di.info.arcs ← Di.info.arcs ∪ {(j, i)};
22             Dj.info.arcs ← Dj.info.arcs ∪ {(i, j)};
23          end

   Procedure Check(in (i, j))
      begin
         DELETE ← false;
         for each v ∈ Di do   if fij(v) ∉ Di then
                                 begin
                                    delete v from Di;
                                    DELETE ← true;
                                 end
         return DELETE;
      end
```

Figure 7. Procedure *NonMergeCheck* for IFCs

The correctness of the first part can be shown as follows. When D_i*.merge* = D_j*.merge,* it means that both i and j have been merged to the node in D_i*.merge*, possibly through some other nodes (refer to line 21 and 22 in *MergeCheck*). Then, both i and j share the same *info* and the same *loc*s in their *element* fields (for their matching values). There is no need to merge them again. Intuitively, this means that a cycle is encountered among the IFCs. Now we look at the three possible cases (line 5-15), taking note that removing a value in D_i (or D_j) automatically means removing its matching value in D_j (or D_i).

1. If $v \in D_i$ has no $f_{ij}(v) \in D_j$, v is removed from D_i. This also means that v's original matching value (which is not $f_{ij}(v)$) in D_j is removed. Thus, one value is removed from D_i, and one value is removed from D_j.
2. If $v \in D_i$ has $f_{ij}(v) \in D_j$ but D_i*.element*(v) and D_j*.element*($f_{ij}(v)$) do not share the same *loc* (line 10), which means that the already merged IFCs are violated, then both v and $f_{ij}(v)$ need to be removed (line 12 and 13). Automatically, their matching values in D_j and D_i are also removed respectively. Thus, two values are removed from D_i, and two values are removed from D_j.
3. Otherwise, v is consistent. Then, $f_{ij}(v) \in D_j$ must also be consistent.

It is clear that after line 15, both D_i and D_j are fully covered and are consistent since a constraint is always assumed to be symmetric. So, there is no need to check arc (j, i), and only one enqueue operation (line 16) is required because D_i*.info.arcs* = D_j*.info.arcs*.

The correctness of the second part is clear as *Check* is actually *REVISE* in AC-3. i and j are not merged. Both arcs (i, j) and (j, i) are checked in the procedure. Note that the two arcs are also attached to j and i (line 22 and 21) respectively, which are not there originally. They may require rechecks as they did not merge. Procedure *ReCheck* is given in Figure 8, which obviously meets its specification.

Procedure ReCheck(**in** (i, j), **inout** Q)
    if Check((i, j)) **then** $Q \leftarrow Q \cup \{arc \mid arc \in D_i.info.arcs\}$

Figure 8. Procedure *ReCheck* for not merged IFCs

4.2.2 Time complexity

Due to the use of hash table, each value in D_i (or D_j) needs to be checked only once in *MergeCheck* or *NonMergeCheck*. Thus, the consistency check in *InitialCheck* is O(d). Now, the question is: can we avoid going to the second part (line 18-23) of *NonMergeCheck*? If we can, then every IFC can be merged, and no recheck is needed. Hence, AC-5*$_3$ is O(ed) for IFCs, which is the same as that in AC-5. But AC-5*$_3$ eliminates the rechecks needed in AC-5, not to mention AC-5*$_3$'s superior space complexity. However, if we cannot avoid going there, those not merged IFCs still need recheck, and the consistency check in *ReCheck* is O(d), which is not as good as that in AC-5.

From the above discussion, we can derive a condition for checking all the IFCs in a CSP only once. Take note that every IFC in *arc*(G) is expressed as one edge (or arc).

Single Check Condition: For each IFC $(i, j) \in arc(G)$, when (i, j) is checked, it satisfies

(a) D_i*.merge* = 0 or D_j*.merge* = 0, or;
(b) D_i*.merge* = D_j*.merge*, with D_i*.merge* $\neq 0$.

It turns out that whether the condition is guaranteed to be satisfied depends on if a CSP is a static one or an incremental one.

Theorem: For a static CSP, we can always fix an order of arcs to be checked initially so that the above condition is satisfied.

The theorem can be proven quite easily. Here is the sketch of the proof. Without loss of generality, we only consider a connected network of IFCs. We first find a spanning tree for the network. Then, we order the IFCs in the spanning tree by traversing the tree (starting from any node). Thus, every IFC in the spanning tree satisfies (a) of the above condition when it is checked. The remaining IFCs can follow those in the spanning tree (in any order). Then, each IFC in this remaining set satisfies (b) when it is checked.

Let us have an example. See the network of IFCs in Figure 9. One way to order the edges is like this: (1, 2), (1, 3), (2, 4), (2, 3) and (3, 4). Then, when checked, (1, 2), (1, 3), (2, 4) satisfy (a) of the above condition, and (2, 3) and (3, 4) satisfy (b).

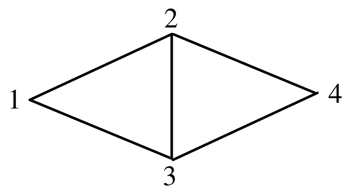

Figure 9. An example network of IFCs

For an incremental CSP, the situation is different. We may no longer have the freedom to order the constraints to fully satisfy the above condition. However, it can be partially satisfied. In this case, those IFCs that do not satisfy the condition still need recheck.

Fortunately, from our experience in building practical systems, we find that there are normally many clusters of IFCs in a real-life CSP, and each cluster typically has only 2 to 3 variables. Then, the above condition is always satisfied. The O(*ed*) result still stands.

5. Experimental Results

In this section, we compare the performances of the specializations of AC-3, AC-5 and AC-5*. The focus is on showing how the new technique for IFCs performs compared to the existing techniques. The constraints involved in the comparison are equations, inequalities and disequations, which are the basic constraints of the current constraint programming languages [25]. Equations are special cases of IFC (see Section 2). All the algorithms are implemented in CMU Common Lisp on SPARC-2.

We implemented AC-3, AC-5, AC-5^*_0, AC-5^*_{IFC}, and AC-5^*_3 with specialized techniques for checking those classes of constraints. AC-3 uses the specialized techniques in [24], and AC-5 uses the specialized techniques in [25]. They both use the same domain data structure in [25] (in [13], AC-3 uses the data structure in [24]). AC-5^*_0 also uses the specialized techniques in [25], but in the AC-5* framework, i.e., for *NW_arc*s, *w* is not used. AC-5^*_{IFC} uses the new technique for IFCs, and the techniques in [25] for the other constraints and also those not merged IFCs. AC-5^*_3 uses the technique described in Section 4 for IFCs and similar techniques to those in [25] for other constraints.

We report two sets of tests. One set uses CSPs with only IFCs, in particular, equations as discussed in Section 2. This set of tests is to show the effects of the new technique on IFCs alone. The other set uses typical scheduling problems. Scheduling problems are used here because most applications of constraint programming are in scheduling, sequencing and other similar domains.

For the first set of tests, variables, domains and constraints are randomly generated. The constraints are also randomly ordered. This is to reflect the incremental nature of practical CSPs. The number of variables ranges from 40 to 80, the domain sizes range from 10 to 100, and the number of constraints ranges from 45 to 90. Figure 10 shows the performance comparison of AC-3 and AC-5*$_3$ in percentage terms over 30 problems. The performance of AC-3 is taken as 100% (shown as the dash line). AC-5*$_3$ takes only 55% to 80% (73% on average) of the time for AC-3 for arc consistency.

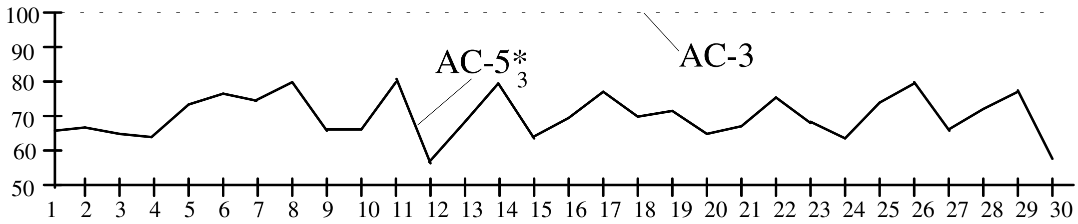

Figure 10. AC-3 and AC-5*$_3$ on IFCs

Figure 11 shows the comparison of AC-5 and AC-5*$_{\mathrm{IFC}}$ on the same set of IFC CSPs. The performance of AC-5 is taken as 100%. AC-5*$_{\mathrm{IFC}}$ only uses 60% to 82% (72% on average) of the time taken by AC-5.

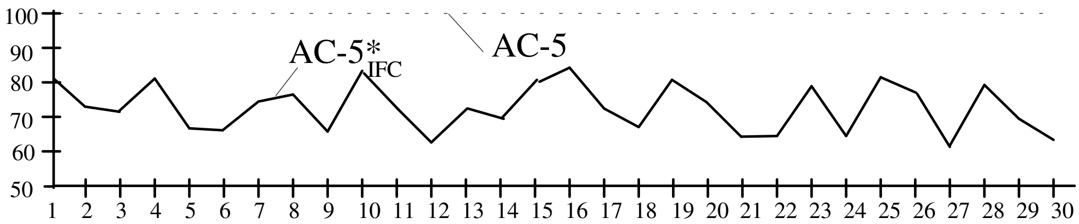

Figure 11. Comparison of AC-5 and AC-5*$_{\mathrm{IFC}}$ on IFCs

For this and the following set of tests using scheduling problems, we will not show the comparison of AC-5*$_0$ and AC-5. Their performances are quite similar. We will not compare AC-3 and AC-5 either because our purpose here is to show the effects of the new technique for IFCs. We believe the data we used is insufficient for a general performance comparison of AC-3 and AC-5. However, in our testing, we find that none dominates the other, although AC-5 has a better time complexity (O(ed)) for IFCs than AC-3 (O(ed^2)). The performances of AC-5*$_{\mathrm{IFC}}$ and AC-5*$_3$ are almost the same.

Our second set of tests is intended to show what we may see when the new technique is used in real applications. The test problems are typical job scheduling problems. The following representation is just one of the ways to represent a scheduling problem. In the scheduling domain, we need to schedule a number of jobs $J = \{J_1, J_2, ..., J_n\}$, and each job consists of a number of operations $J_i = \{Op_{i_1}, Op_{i_2}, ... Op_{i_m}\}$. The operations in each job have to be performed in a fixed sequence. Thus, the start time S_{i_k} of one operation must be greater than or equal to the end time $E_{i_{k\text{-}1}}$ of its previous operation, i.e., $S_{i_k} \geq E_{i_{k\text{-}1}}$. Each operation also has a duration Du_{i_k}. Then, we have the constraint $E_{i_k} = S_{i_k} + Du_{i_k}$ (which is an IFC). Each job has a due time Dt_i, by which it must be finished, i.e., $E_{i_m} \leq Dt_i$. There are also other constraints such as certain operations from different jobs may need to start at the same time or finished no later than certain time, etc. This by no means describes a full scheduling problem. A real problem also needs to consider resources, and requires a good solution. However, these are more to do with heuristic search strategies rather than consistency check, and they are out of the scope of this work. The results reported here are only for achieving consistency.

For this set of tests, we fix the number of operations to be 5 for each job, which means 10 variables per job. Each domain has 100 values. However, the number of jobs is randomly generated for each problem, ranging from 4 to 8. The constraints are also randomly sequenced to reflect the incremental nature of real-world CSPs. In each problem, around half of the constraints are IFCs. Figure 12 and 13 show the performance comparisons of AC-3 and AC-5*$_3$, and AC-5 and AC-5*$_{IFC}$ respectively (over 30 problems). It can be seen that the new technique produces substantial saving. AC-5*$_3$ takes 55% to 70% (64% on average) of the time for AC-3. AC-5*$_{IFC}$ only takes 54% to 71% (65% on average) of the time for AC-5.

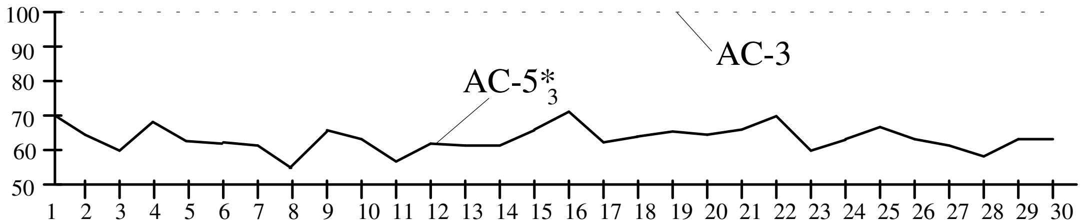

Figure 12. AC-3 and AC-5*$_3$ on scheduling problems

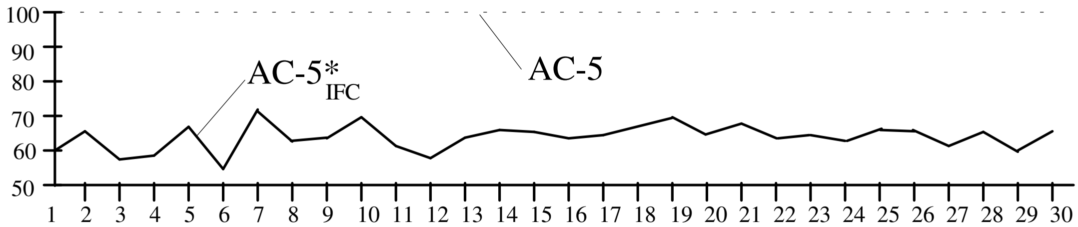

Figure 13. Comparison of AC-5 and AC-5*$_{IFC}$ on scheduling problems.

It can be seen from the figures that the new technique saves more with the second set of tests. This is because in the first set, many IFCs are not merged due to the *Single Check Condition* (in the second set, all the IFCs are merged), and IFCs requires more processing than the other constraints. For example, for an IFC (or FC) in AC-5, each domain needs to be traversed twice (once in the initial check step and once in the recheck step), while for the other constraints, it needs to be traversed at most once.

Finally, from all these comparisons, we can safely say that the new technique consistently produces considerable saving.

6. Related Work

Many general arc consistency algorithms have been developed in the past. AC-3 was perhaps the simplest. It has the optimal space complexity of O(e + nd), but its time complexity is O(ed^3). AC-4, working with domain values rather variables as in AC-3, has the optimal time complexity of O(ed^2), but its space complexity is O(ed^2). AC-6, which works in a similar way as AC-4, improves the space complexity of AC-4 from O(ed^2) to O(ed). The idea of working on domain values was also exploited by Perlin [22] to achieve a better bound for some classes of constraints through factorization.

AC-5 provides an unified and generic algorithm that can exploit the structure of the domain and also the structure of the constraints to produce more efficient algorithms for specific constraints. These efficient specific methods play a more important role than general algorithms in practical applications. The most commonly used constraints are FCs, ATFCs and MCs. Their subcases (i.e., equations, disequations and inequalities) form the core of the current constraint programming languages, e.g., CHIP [24], Charme [3], Ilog Solver [10], etc. In [25], AC-5 is specialized to achieve an O(ed) algorithm in time for these constraint classes. However, due to its fixed queue element representation, it cannot be specialized to reduce the space complexity, which is O(*ed* + *nd*).

AC-5* is an improvement over AC-5. It is also a generic algorithm. But because of its two queue element representations, it can be specialized to produce efficient algorithms for specific constraints not only in time, but also in space.

The new technique for IFCs is motivated by AC-5's technique for FCs. Although our technique is still O(*ed*) in time, experiments have shown it is more efficient than that in AC-5. More importantly, it is now possible to have an arc consistency algorithm for IFCs, ATFCs and MCs, which is both optimal in time and also optimal in space.

Mohr and Masini [19] discovered independently that binary equations, inequalities, and disequations can be solved in O(*ed*). Earlier work on constraint solvers (e.g., ALICE [14]) and constraint programming languages (e.g., CHIP [24]) also presented special algorithms for these classes of constraints. However, equations in all these methods need to be checked many times.

Recently, [2] proposed another general arc consistency algorithm AC-7. The main idea of AC-7 is to take advantage of the bidirectionality of constraints to reduce the number of consistency checks. Its space requirement is still O(*ed*) like AC-6 [1]. [2] did mention that the semantics of constraints can be used to infer or reduce constraint checks. However, it is not clear how AC-7 could be specialized to reduce both the time and space complexities for IFCs, ATFCs and MCs.

7. Conclusion

In this paper, we have proposed an improved generic arc consistency algorithm AC-5*. It can be specialized for specific constraints to produce efficient algorithms not only in time but also in space. This is significant because for practical applications, time and space complexities are equally important. In addition, we have also presented a new consistency technique for IFCs. It checks almost all IFCs only once rather than many times. Although this technique does not reduce the time complexity (compared to that in AC-5), our experiments have shown it outperforms the existing methods substantially. With this technique and AC-5*, we can have an arc consistency algorithm that is both optimal in time and in space for IFCs (almost), ATFCs and MCs. The main application of these techniques will be in constraint programming for solving practical scheduling, planning, sequencing and resource allocation problems.

References

[1] C. Bessiere and M. Cordier, Arc-consistency and arc-consistency again, *AAAI-93*, 108-113.

[2] C. Bessiere, E. Freuder and J-C. Regin, Using inference to reduce arc consistency computation, *IJCAI-95*, 592-598, 1995.

[3] *Charme Reference Manual*, Artificial Intelligence Development Centre, Bull, 1990.

[4] P. David, When functional and bijective constraints make a CSP polynomial, *IJCAI-93*, 224-229, 1993.

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

[6] Y. Deville and P. Van Hentenryck, An efficient arc consistency algorithm for a class of CSP problems, *AAAI-91*, 325-330.

[7] M. Dincbas, H. Simonis and P. Van Hentenryck, Solving large combinatorial problems in logic programming, *Journal of Logic Programming*, 8:75-93, 1990.

[8] E. C Freuder, Synthesizing constraint expressions, *Commun. of ACM*, 21: 958-966, 1978.

[9] R.M. Haralick and G.L. Elliot, Increasing tree search efficiency for constraint satisfaction problems, *Artificial Intelligence*, 14:263-313, 1980.

[10] Ilog Solver Reference Manual, ILOG, 1993.

[11] S. Kasif, On the parallel complexity of discrete relaxation in constraint satisfaction networks, *Artificial Intelligence*, 45:275-286, 1990.

[12] B. Liu and Y.W. Ku, ConstraintLisp: an object-oriented constraint programming language. *SIGPLAN Notices*, 27(11):17-26, 1992.

[13] B. Liu, Increasing functional constraints need to be checked only once, To appear in *IJCAI-95*, Montreal, Canada, August 19-25, 1995.

[14] J. Lauriere, A language and a program for stating and solving combinatorial problems, *Artificial Intelligence*, 10:29-127, 1978.

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

[16] 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.

[17] R. Mohr, Good old discrete relaxation, ECAI-88, Munich, Germany, 1988.

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

[19] R. Mohr and G. Masini, Running efficiently arc consistency, Springer, Berlin, 1988, 217-231.

[20] U. Montanari and F. Rossi, An efficient algorithm for the solution of hierarchical networks of constraints, *Workshop on Graph Grammars and Their Applications in Computer Science*, Warrenton, 1986.

[21] B. Nadel, Constraint satisfaction algorithms, *Computational Intelligence*, 5:288-224, 1989.

[22] M. Perlin, Arc consistency for factorable relations, *Artificial Intelligence*, 28:329-342, 1992.

[23] P. Van Beek, On the minimality and decomposability of constraint networks, *AAAI-92*, 447-452, 1992.

[24] P. Van Hentenryck, *Constraint satisfaction in logic programming*, MIT Press, 1989.

[25] P. Van Hentenryck, Y. Deville and C-M. Teng, A generic arc-consistency algorithm and its specifications, *Artificial Intelligence*, 27:291-322, 1992.

[26] P. Van Hentenryck, V. Saraswat and Y. Deville, *Constraint processing in cc*(*FD*), Tech. Rept. CS-93-02, Brown University, 1982.

[27] D. Waltz, *Generating semantic descriptions from drawings of scenes with shadows*, Tech Rept. AI271, MIT, Cambridge, MA, 1972.