An Improved Generic Arc Consistency Algorithm 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. This paper makes two contributions. First, it proposes an improved generic arc consistency algorithm, called AC-5*, which can be specialized to reduce both time and space complexities. Second, it presents a more efficient technique for handling an important subclass of functional constraints, namely *increasing functional constraints* (IFC).

1. Introduction

Arc consistency techniques are the key techniques for solving CSPs. Past research has produced many general and specific arc consistency algorithms. The most important general algorithms are AC-3 [6], AC-4 [8], AC-5 [10] 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($e + nd$) [8], while AC-4 and AC-6 have the optimal time complexity of O(ed^2) [8, 1], where n is the number of variables, e is the number of arcs, and d is the size of the domain.

Specific algorithms are efficient specializations of the general ones for specific constraints [9, 10]. These algorithms 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 [9] and Ilog Solver [3], for solving practical combinatorial search problems, such as scheduling and resource allocations [3, 10]. The basic constraints used in these languages are special cases of FC, ATFC and MC.

In [10], AC-5 is specialized to produce an algorithm running in O(*ed*) (the optimal time complexity [10]) 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 to check 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. The main idea of this technique was first introduced in [5]. This paper enhances the technique and presents two implications of it.

1. Although the new technique is still O(ed) (in time) for IFCs, the same as that of AC-5, experiments 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 the current constraint programming languages [9, 3] is actually an IFC.

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 among 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.

In this paper, we assume the standard definitions of arc consistency [7, 10]. We now define functional (FC) and increasing functional (IFC) constraints. We will not define anti-functional and monotonic constraints as they can be found in [10]. Since our techniques require a total ordering on the domain, let us define it here first.

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

Definition 2. 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 3. 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 $u \prec v$ implies $f_{ij}(u) \prec f_{ij}(v)$.

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*.

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 languages are special cases of FC, ATFC and MC [10, 3]. In fact, they are equations ($aX = bY + c$), inequalities ($aX \leq bY + c$) 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 3, 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 can be used. ArcCons checks an arc when it is first encountered, and LocalArcCons rechecks it if its consistency is broken. [10] 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 of efficiency reasons [4].

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.
- Due to this queue element representation, AC-5 achieves the time complexity of O(ed), and the space complexity of O($ed + nd$) for FCs, ATFCs and MCs. The space complexity is worse than that of AC-3, which is O($e + nd$).
- From AC-5, 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 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. If w is not known, only an O(ed^2) algorithm (in time) can be achieved for FCs.

We now present AC-5* (Figure 1). Like AC-5, AC-5* also has two steps, the initial check step (line 2-3), 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.

The algorithm is parametrized on two procedures, *InitialCheck* and *ReCheck*, and the representation of the queue element *ELT*. Their implementations are left open. 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.

```
    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

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 now 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*. Notice that there are two *ReCheck* procedures (I and II) because of the two queue element representations. *Enqueue* is given in Figure 3. *Remove* removes the values in Δ from D_i.

```
    Procedure InitialCheck (in (i, j), inout Q)
1       Δ ← {v ∈ D_i | ∀w ∈ D_j: ¬C_ij(v, w)};
2       Remove(Δ, i);
3       Enqueue(i, Δ, Q)

I   Procedure ReCheck (in (i, j), inout Q)
1       Δ ← {v ∈ D_i | ∀w ∈ D_j: ¬C_ij(v, w)};
2       Remove(Δ, i);
3       Enqueue(i, Δ, Q)

II  Procedure ReCheck(in ((i, j), w), inout Q)
1       Δ1 ⊆ Δ ⊆ Δ2  with  Δ1 ← {v ∈ D_i | C_ij(v, w) and ∀w' ∈ D_j: ¬C_ij(v, w')}
                           Δ2 ← {v ∈ D_i | ∀w' ∈ D_j: ¬C_ij(v, w')}
2       Remove(Δ, i);
3       Enqueue(i, Δ, Q)
```

Figure 2. *InitialCheck* and *ReCheck* procedures

```
    Procedure Enqueue (in i, Δ, inout Q}
        if Δ ≠ ∅ then   begin
                            Q ← Q ∪ {((k, i), v) | (k, i) ∈ W_arc(G) and v ∈ Δ};
                            Q ← Q ∪ {(m, i) | (m, i) ∈ NW_arc(G)}
                        end
```

Figure 3. *Enqueue* procedure

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. With proper implementation of the three items, AC-5* can be reduced to AC-3, AC-4, AC-5, and AC-6.

- 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).
- 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*. 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.

Like AC-5, AC-5* can also be 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 achieves 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 Section 4, we present a technique for IFCs that does not need w to produce an $O(ed)$ algorithm in time. This results in the space complexity of $O(e + nd)$ for IFCs.

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 will also describe our new technique for checking IFCs, which is an improvement over that in [5]. The main improvement is in the handling of cycles of IFCs, which were not handled satisfactorily [5]. 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. We will not describe the specialized procedures for ATFCs and MCs as they are almost the same as 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 *NW_arc*s. Hence, the space complexity is $O(e + nd)$ as AC-3. Due to the new method for IFCs, in line 2 of AC-5*, each IFC 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. 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 contains 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. The field *arcs* stores 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.

4.2 Check IFCs

This subsection presents the 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 so that all the IFCs can be merged in initial check. Then, recheck 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.

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

4.2.1 Procedure *InitialCheck* for IFCs

We now present *InitialCheck* for IFCs (Figure 5). Two sub-procedures are used. The first one is *MergeCheck* (Figure 6), which merges the two variables. The second one is *NonMergeCheck* (Figure 7), which does not merge the two variables.

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

Figure 5. *InitialCheck* for IFCs

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

Figure 6. Procedure *MergeCheck* for IFCs

MergeCheck merges D_i and D_j by making them share the same *info* and some information in their *element* fields. See [5] for an example and the proof of its correctness. *NonMergeCheck* given in Figure 7 consists of two parts, the first part (line 2-16) is used when $D_i.merge = D_j.merge$, and the second part (line 17-22) is used otherwise. Procedure *ReCheck* is obvious and omitted.

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, *InitialCheck* is O(d). Now, the question is: can we avoid going to the second part (line 17-22) 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 rechecks, 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 is expressed as one edge (or arc).

Single Check Condition: For each IFC (i, j), 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 the CSP is a static one or an incremental one.

```
Procedure NonMergeCheck(in (i, j), inout Q)
1     if D_i.merge = D_j.merge then
2         begin
3             DELETE ← false;
4             for each v ∈ D_i do
5                 if f_ij(v) ∉ D_j then  begin
6                                          delete v from D_i;
7                                          DELETE ← true;
8                                        end
9                 elseif D_j.element(f_ij(v)) and D_i.element(v) have different locs then
10                        begin
11                            delete v from D_i;
12                            delete f_ij(v) from D_j;
13                            DELETE ← true;
14                        end
15            if DELETE then Q ← Q ∪ {arc | arc ∈ D_i.info.arcs};
16        end
17    else  begin
18            if Check((i, j)) then Q ← Q ∪ {arc | arc ∈ D_i.info.arcs};
19            if Check((j, i)) then Q ← Q ∪ {arc | arc ∈ D_j.info.arcs};
20            D_i.info.arcs ← D_i.info.arcs ∪ {(j, i)};
21            D_j.info.arcs ← D_j.info.arcs ∪ {(i, j)};
22        end
  Procedure Check(in (i, j))
      DELETE ← false;
      for each v ∈ D_i do  if f_ij(v) ∉ D_i then  begin
                                                    delete v from D_i;
                                                    DELETE ← true;
                                                  end;
      return DELETE;
```

Figure 7. Procedure *NonMergeCheck* for IFCs

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. Interested readers, please refer to [4].

For an incremental CSP, we no longer have the freedom to order the constraints to satisfy the above condition. However, it can be partially satisfied. Thus, those IFCs that do not satisfy the condition still need recheck. Fortunately, our experiences of building practical systems show 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. Then 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 languages [10]. 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*$_{IFC}$, and AC-5*$_3$ with specialized techniques for checking those classes of constraints. AC-3 uses the specialized techniques in [9], and AC-5 uses the specialized techniques in [10]. They both use the same domain data structure in [10] (in [5], AC-3 uses the data structure in [9]). AC-5*$_{IFC}$ uses the new technique for merging IFCs, and the techniques in [10] for the other constraints and also for those not merged IFCs. AC-5*$_3$ uses the techniques (initial check and recheck) described in Section 4 for IFCs and similar techniques to those in [10] for other constraints.

We report two sets of tests. One set uses CSPs with only IFCs, in particular, equations as discussed in Section 2. 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, which is to reflect the incremental nature of the 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 8 shows the time 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% of the time for AC-3 for arc consistency.

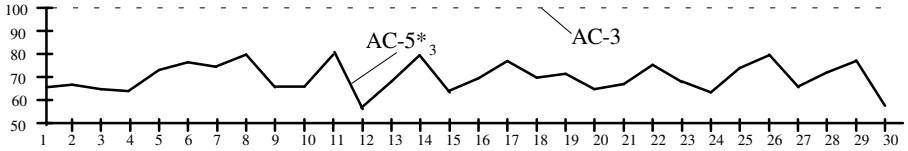

Figure 8. AC-3 and AC-5*$_3$ on IFCs

Figure 9 shows the comparison of AC-5 and AC-5*$_{IFC}$ on the same set of IFC CSPs. AC-5*$_{IFC}$ only uses 60% to 82% of the time taken by AC-5.

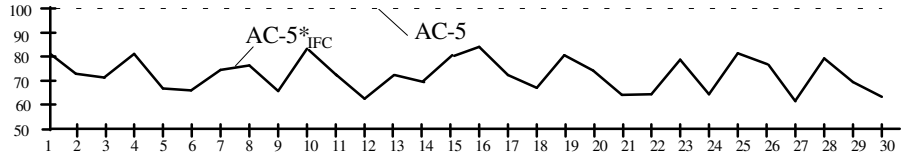

Figure 9. Comparison of AC-5 and AC-5*$_{IFC}$ on IFCs

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. 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. So the start time S_{i_k} of one operation and the end time $E_{i_{k-1}}$ of its previous operation must satisfy $S_{i_k} \geq E_{i_{k-1}}$. 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 some other constraints not listed here. These by no means describe a full scheduling problem. A real problem also needs to consider resources, and requires a good solution. However, they are more to do with heuristic search strategies rather than consistency check, and they are out of the scope of this paper. 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. In each problem, around half of the constraints are IFCs. Figure 10 and 11 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% of the time for AC-3. AC-5*$_{IFC}$ only takes 54% to 71% of the time for AC-5.

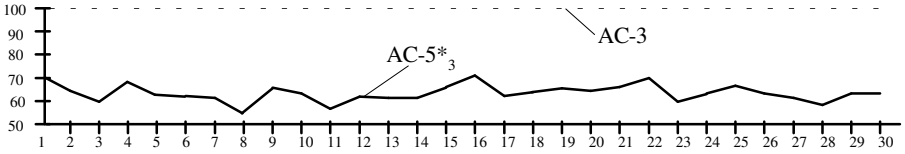

Figure 10. AC-3 and AC-5*$_3$ on scheduling problems

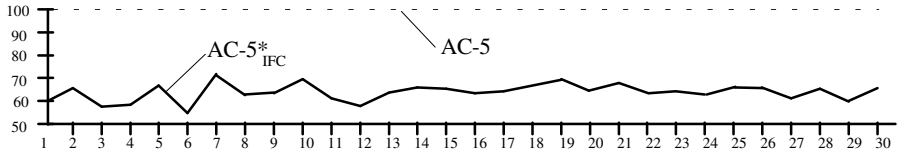

Figure 11. Comparison of AC-5 and AC-5*$_{IFC}$ on scheduling problems.

These comparisons show that the new technique consistently produces considerable saving.

6. Related Work

Many general arc consistency algorithms have been developed in the past. AC-3 has the optimal space complexity of O($e + nd$) [7, 8], but its time complexity is O(ed^3) [7]. AC-4 has the optimal time complexity of O(ed^2) [8], but its space complexity is O(ed^2) [8]. AC-6 reduces the space complexity of AC-4 to O(ed) from O(ed^2) [1].

AC-5 provides an generic algorithm that can exploit the structure of the domain and the structure of the constraint to produce more efficient algorithms for specific constraints. The most commonly used specific constraints are FCs, ATFCs and MCs. Their subclasses (i.e., equations, disequations and inequalities) form the core of the current constraint languages, e.g., CHIP [9], Ilog Solver [3], etc. In [10], AC-5 is specialized to achieve an O(ed) algorithm in time for these constraint classes. However, AC-5 cannot be specialized to reduce the space complexity.

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 both in time and in space.

Although the new technique for IFCs is still O(*ed*) in time, experiment results have shown that 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.

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

This paper proposed an improved generic arc consistency algorithm AC-5*. It can be specialized to produce efficient algorithms for specific constraints not only in time but also in space. This is significant because for practical applications, both time and space complexities are important. Furthermore, the paper presented a new consistency technique for IFCs. It checks almost all IFCs only once rather than many times. With this technique and AC-5*, we can have an arc consistency algorithm that is both optimal in time and optimal in space for IFCs, ATFCs and MCs. The main application of these techniques will be in constraint programming.

Acknowledgments: I thank Kim-Heng Teo for his valuable comments on various parts of this paper.

References

[1] C. Bessiere and M. Cordier, "Arc-consistency and arc-consistency again," *Artificial Intelligence,* 65:179-190, 1994.

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

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

[4] B. Liu, *AC-5*: an improved AC-5 and its specializations*, 1996. DISCS Technical Report. 1996.

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

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

[7] 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.

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

[9] P. van Hentenryck, *Constraint satisfaction in logic programming*, 1989.

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