# Increasing Functional Constraints Need to Be Checked Only Once

Bing Liu
Department of Information Systems and Computer Science
National University of Singapore
Lower Kent Ridge Road, Singapore 0511
Republic of Singapore

## Abstract

Central to solving Constraint Satisfaction Problem (CSP) is the problem of consistency check. Past research has produced many general and specific consistency algorithms. 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. They form the basis of the current constraint programming languages. This paper proposes a more efficient method for checking an important subclass of functional constraints, increasing functional constraints. Rather than checking them many times as in a typical consistency check process, in the new method they (almost all of them) only need to be checked once. This results in a substantia] saving in computation.

## 1. Introduction

The key technique in solving CSPs is consistency check. Arc consistency algorithms that work on a network representation of binary CSPs are perhaps the most important class of consistency algorithms. In this paper, we are also concerned with arc consistency of binary constraints, in particular, increasing functional constraints.

Over the past two decades, a number of general arc consistency algorithms have been proposed, e.g., AC-3 [Mackworth, 1977], AC-4 [Mohr and Henderson, 1986], AC-5 [Hentenryck et a/., 1992] and AC-6 [Bessiere and Cordier, 1993]. Apart from the general algorithms, many specific methods were also designed which are specializations of the general ones for specific constraints [Lauriere, 1978; Mohr and Masini, 1988, Hentenryck et a/., 1992]. These methods typically exploit the semantics of individual constraints, and are more efficient in checking these constraints. For instance, in [Hentenryck et a/., 1992], AC-5 is specialized for functional, anti-functional and monotonic constraints, and their piecewise generalizations (see their formal definitions in [Hentenryck et a/., 1992]).

Recent years, the CSP model has been implemented in constraint programming languages, such as CHIP [Hentenryck, 1989], Charme [Charme, 1990], Hog Solver [Hog, 1993], etc., for solving practical combinatorial problems, such as scheduling, sequencing and resource allocations [Dincbas et a/., 1990; Hentenryck, 1989]. The basic constraints used in these languages are special cases of functional, anti-functional and monotonic constraints.

In this paper, we propose a more efficient consistency technique for a frequently used subclass of functional constraints, namely, increasing functional constraints (IFC). This technique allows this subclass of constraints (almost all of them) to be checked (or considered) only once in an arc consistency check process.

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. This is achieved by merging the domains of the two variables in a IFC in the initial check such that rechecks will not be necessary. This technique may be embedded in any general arc consistency algorithm. This paper only presents a modified AC-5 algorithm, called AC-S"', to incorporate this technique.

Experimental results have shown the new technique outperforms the existing techniques substantially. This saving is important in practice because from our experiences in using two constraint languages, Charme and Hog Solver to build practical systems, many constraints are IFCs, in particular, equations of the form aX = bY + c, where X and Y are variables, and a and b are positive constants and c a constant.

Section 2 gives definitions and conventions. Section 3 describes AC-$5^+$. Section 4 presents the new method for IFCs. Section 5 shows the test results. Section 6 discusses the related work and Section 7 concludes the paper.

## 2. Preliminaries

A binary 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. If i and j are variables (i < j), we assume, for simplicity, that there is at most one constraint relating them, denoted $C_{ij}$.. A solution to a CSP is an assignment of values to variables such that constraints are satisfied.

A graph G can be associated with a binary CSP, where each node represents a variable /, and each edge between two variables i and y, 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 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 $(i, j) \in arc(G)$ such that $(i, j)$ is arc consistent with respect to $D_i$ and $D_j$.

Our new technique requires a total ordering on the variable domain. This total ordering 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 a functional constraint and an *increasing functional constraint*.

**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 $C_{ij}(v, w)$. 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* (IFC). An example of an IFC is $X = Y + 5$.

Most earlier algorithms (e.g., AC-3 [Mackworth, 1977], AC-4 [Mohr and Henderson, 1986] and also AC-6 [Bessiere and Cordier, 1993]) do not use the semantics of constraints to achieve better efficiency. AC-5 [Hentenryck *et al.*, 1992] is different as the implementations of its two procedures ArcCons and LocalArcCons are left open. This means that for different constraints different algorithms could be used. ArcCons checks an arc when it is first encountered, while LocalArcCons rechecks it if its consistency is broken by consistency check of other arcs. [Hentenryck *et al.*, 1992] provides the special ArcCons and LocalArcCons procedures for checking functional, anti-functional and monotonic constraints in $O(ed)$.

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 algorithm for recheck for certain constraints (e.g., functional constraints) may be different from its initial check algorithm.

In most situations, rechecks take a considerable amount of computation. In this paper, we will try to eliminate the rechecks for IFCs.

## 3. AC-5$^+$ Algorithm

Before discussing the new technique for IFCs, we first present AC-5$^+$, which provides the data structures and the framework for the discussion of our technique. The intention here is not to propose a new algorithm, but to do minimum modifications to AC-5 so that the new technique can be embedded in it. The main modification is in the domain data structures. In fact, we also modified AC-3 for our technique, which is much easier to modify than AC-5. Due to the space limitation, it is not discussed here.

AC-5$^+$ works with the domain data structures in Figure 1. Like AC-5, we assume a total ordering on the domain.

---

Let $S = \{v_1, ..., v_a\}$ with $v_k < v_{k+1}$ be the original domain of $i$;
$D_i = \{v_{p1}, ..., v_{pm}\} \subseteq S$ with $p_k < p_{k+1}$ and $m > 0$

**Syntax**
$D_i.merge$: boolean
$D_i.element$: set of pairs $(v, loc)$ with $v \in S$ and $loc$ is a structure, organized as a hash table on key $v$.
$D_i.values$: array $[1..a]$ of elements $\in S$
$D_i.info$: an *info* structure

**Semantics**
$D_i.merge$ = *true* if this domain has been merged with another domain
= *false* 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 arcs.

**Semantics**
$info.size = m$
$info.min = p_1$
$info.max = p_m$
$info.succ[p_k] = p_{k+1}$ $(1 \le k < m)$, $info.succ[p_m] = +\infty$
$info.pred[p_{k+1}] = p_k$ $(1 \le k < m)$, $info.pred[p_1] = -\infty$
$info.arcs = \{(k, i) \mid (k, i) \in arc(G)$, $(k, i)$ is not a IFC$\}$

(C). *info* data structure

---

Figure 1. Domain data structures

Figure 1(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 $v \in S$, and $loc$ is a structure (Figure 1(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 return 0). This 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 *info* field gives all the information about the current domain. Its data structure is shown in Figure 1(C). This is also different from AC-5 as AC-5 keeps all the information in the domain data structure in Figure 1(A). This modification is also crucial to our new technique.

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.

We now present AC-5$^+$ (Figure 2). It has two parts, the initial check part (line 2 and 3), and the recheck part (line 4-8). Note that in line 2, we use arc(G') instead of arc(G) as in AC-5. arc(G) is the same as arc(G) except that each IFC is expressed as one arc, either (i, j) or (j, i), rather than two arcs. The reason for this will be clear later. Q in AC-5$^+$ has elements of the form <(i, j), w> (line 6), where (/, j) is an arc and w is the array index of a real value (in Devalues) removed from Dj Only in one special case, w is the real value itself (see next section). In AC-5, w is always a real value removed from Dj. This change is also important.

```
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 <(i, j), w> from Q;
7                   recheck(i, j, w, Q)
8               end
        end
```

Figure 2. The AC-5$^+$ algorithm

The implementations of the two procedures initialCheck and recheck are left open. Their general definitions are almost the same as those for ARCCONS and LOCALARCCONS in AC-5. Due to the space limitation, we will not describe them here. The major difference is that AC-5$^+$ embeds two procedures remove and enqueue inside initialCheck and recheck as it gives more flexibility in implementing specialized initialCheck and recheck procedures. In AC-5, these procedures are in the main AC-5 algorithm, remove removes those inconsistent domain values from Di and enqueue adds elements of the form <(k, /), w> to Q, where (k, /) is a related arc of i and w is either the Di values array index of a real value or a real value.

AC-5$^+$ inherits all the properties and complexity results of AC-5 as the key operations and data structures of them are almost the same. The differences are non-essential.

## 4. Merge Variable Domains in a IFC

We now describe the proposed technique that checks (almost all) IFC only once. The main idea is to exploit the fact that consistent values in the variable domains of a IFC is one-to-one correspondent and in an increasing order. In initial check, we can make the two variables of a IFC share some key information. Then, later on, domain change of one variable will be felt automatically by the other. In this way, rechecks of the constraint can be avoided.

As indicated, our technique does not guarantee that every IFC will be checked once. Let us describe the condition under which every IFC needs only one check. Take note that every IFC in arc(G) is expressed as one arc (or edge).

Condition: each IFC in arc(G) to be checked (in the initial check process) must have no more than one variable that has appeared in a previously checked IFC (also in the initial check process).

For example, we have IFC arcs (1, 2), (3, 4), and (2, 3). If they are checked in this order, the above condition is not satisfied. It may be rearranged as (1, 2), (2, 3), and (3,4) to satisfy the above condition.

In practical problem solving, this condition may not be fully satisfied but only partially because the sequencing of constraints may depend on the problem, and the IFCs may also form cycles. In these cases, those IFCs that do not met the condition still need recheck. However, from our experience in building practical systems, there are normally many clusters of IFCs in a practical CSP, and each cluster typically has only 2 to 3 variables. Then, the above condition is always satisfied. That is why we say almost all IFCs need no recheck in our method.

Figure 3 shows the procedure initialCheck for IFCs. Two sub-procedures are used. The first one is mergeCheck (Figure 4), which is used when the arc (i, j) satisfies the above condition. The second one is nonMergeCheck (Figure 6), which is used when the arc (i,j) does not satisfy the condition. These two procedures use enqueue (Figure 5) mentioned in the last section.

```
    Procedure initialCheck(in i, j, out Q)
        begin
1           if not D_i.merge then
2               mergeCheck(i, j, Q)
3           elseif D_i.merge then
4                   nonMergeCheck(i, j, Q)
5           else    mergeCheck(j, i, Q)
        end
```

Figure 3. Procedure *initialCheck* for IFCs

*mergeCheck* merges $D_i$ and $D_j$ by making them share the same *info* and some information in their *element* fields. Notice that a domain can be merged with any number of other domains. Let us have an example of this merging. 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 |
| $loc.index(D_y.element(f_{x-y}(v)))$ | = | | | 3 | | 5 | 6 |

Note that $u$ stands for undefined, and consistent values in the new $D_i.values$ 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.

```
Procedure mergeCheck(in i, j, out Q)
    begin
1       Δ1 ← { };
2       newValues ← make an array of the same size
                    as Di.values;
3       for each v (= Di.values[indexi]) ∈ Di do
4           if fij(v) ∉ Dj then
5               delete v from Di;
6               Δ1 ← Δ1 ∪ {indexi}
7           else  indexj ← the array index of the value in
                            Dj.values supporting v;
8               newValues[indexi]← Dj.values[indexj];
9               Dj.element(fij(v)) ← Di.element(v);
10              delete fij(v) from Dj without modifying
                            Dj.element.
11      enqueue(i, Δ1, Q)
12      Δ2 ← { };
13      for each w ∈ Dj do
14          begin
15              Δ2 ← Δ2 ∪ {|w|};
16              loc.index(Dj.element(w)) ← 0
17          end
18      enqueue(j, Δ2, Q);
19      Di.merge ← true;
20      Dj.merge ← true;
21      Di.info.arcs ← Di.info.arcs ∪ Dj.info.arcs;
22      Di.values ← newValues;
23      Dj.info ← Di.info
    end
```

Figure 4. Procedure for merging two variables in a IFC

```
procedure enqueue(in i, Δ, inout Q)
        Q ← Q ∪ {<arc, w>) | arc ∈ Di.info.arcs, w ∈ Δ}.
```

Figure 5. Procedure for adding elements to $Q$ for recheck

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

- The array *newValues* created with the same size as $D_i.values$ is used to store $D_j$'s corresponding values in $D_j$. Because IFCs are functional, the number of values in $D_j$ after the initial check cannot be more than the size of the array in $D_i.values$. In line 8, whenever a value $v \in D_i$ has a corresponding value $w \in D_j$, $w$ is stored in *newValues* in the $index_i$th cell. $index_j$ (line 7) is obtained when testing the membership of $f_{ij}(v)$ in $D_j$ using the hash table in $D_j.element$. Thus, after line 10, *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 10 deletes those values in $D_j$ with matching values in $D_i$ without modifying $D_j.element$. This operation can be done in $D_j.succ$ which is subsequently replaced in line 23. So, deleting those matching values in $D_j$ does no harm. Line 13-17 removes those values in $D_j$ that do not have 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 23).

  In line 22, the original $D_i.values$ is replaced with *newValues*. It is clear no consistent value is lost and no inconsistent value is kept in the new $D_i.values$. All the inconsistent values in $D_i$ are removed in line 5.
- Line 9 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 21 concatenates the related arcs of $i$ and $j$ and assign it to $D_i.info.arcs$. This and the operation in line 23 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 23 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 domains is complete. Later on, anything happens to $D_i$ (or $D_j$), $D_j$ (or $D_i$) feels it automatically.
- The use of indices in $\Delta_1$ is crucial. When some values are removed from $D_i$ (or $D_j$) later on, the corresponding values in those domains merged with $D_i$ (or $D_j$) are also removed due to our merging operations. Indices in $\Delta_1$ reflects all these removals. Real domain values cannot be used in $\Delta_1$ because they only reflect the value changes in $D_i$ but not the others merged with it.

  However, $\Delta_2$ does contain the actual values removed from $D_j$ (the only case in AC-5$^+$) because $D_j.values$ is replaced with *newValues* so that using indices will be incorrect. As $D_j$ does not have any variable merged with it, real values can be used. Note that for a real value, we use $|w|$ as an indicator (line 15) (in a real implementation another indicator may be used). Procedure *recheck* uses this indicator to identify the value type, an array index or a real domain value.

*nonMergeCheck* is given in Figure 6. Its correctness is clear. It is almost the same as ARCCONS in AC-5 for functional constraints except that both arcs $(i, j)$ and $(j, i)$ are checked in *nonMergeCheck*. Note that in line 13 and 14, two arcs for the constraint are attached to two variables. These two arcs are not there originally, and they may require rechecks as they did not merge.

Since some IFCs may still need recheck, the procedure *recheck* is given in Figure 7, which is almost the same as that in AC-5 except in line 1, where we need to test whether $w$ is a real domain value or an array index.

Due to the use of hash table, each value in $D_i$ needs to be checked only once in *mergeCheck* and *nonMergeCheck*. Thus, *initialCheck* is $O(d)$. *recheck* is $O(1)$ as only one value needs to be checked. Hence, algorithm AC-5$^+$ is $O(ed)$ for IFCs, which is the same as for functional constraints in AC-5. However, our algorithm eliminates most of the rechecks needed in AC-5.

```
Procedure nonMergeCheck(in i, j, out Q)
    begin
1     Δ ← { };
2     for each v (= D_i.values[index_i]) ∈ D_i do
3         if f_ji(v) ∉ D_j then
4             delete v from D_i;
5             Δ ← Δ ∪ {index_i}
6     enqueue(i, Δ, Q);
7     Δ ← { };
8     for each w (= D_j.values[index_j]) ∈ D_j do
9         if f_ji(w) ∉ D_j then
10            delete w from D_j;
11            Δ ← Δ ∪ {index_j}
12    enqueue(j, Δ, Q);
13    D_i.info.arcs ← D_i.info.arcs ∪ {(j, i)};
14    D_j.info.arcs ← D_j.info.arcs ∪ {(i, j)}
    end
```

Figure 6. Procedure *nonMergeCheck* for a IFC

```
Procedure recheck(in i, j, w, out Q)
    begin
1     if w = {index} then w ← D_j.values[index]
2     if f_ji(w) ∈ D_i then
3        delete f_ji(w) (= D_i.values[index_i]) from D_i;
4        Δ ← {index_i}
5     else Δ ← { }
6     enqueue(i, Δ, Q);
    end
```

Figure 7. Procedure *recheck* for a IFC

*initialCheck* and *recheck* methods for non-IFC functional, anti-functional and monotonic constraints can be easily devised for AC-5$^+$. They are basically the same as those in AC-5.

## 5. Experimental Results

Here, we first compare the performances of AC-5$^+$ and AC-5 over equations, inequalities and disequations, which are the basic constraints of the current constraint languages. Equations, inequalities and disequations are special cases of functional, monotonic and anti-functional constraints respectively. Increasing equations, such as $X = aY + b$, where $X$ and $Y$ are variables, $a$ and $b$ are constants and $a$ is positive, are special cases of IFCs.

Since AC-3 may be preferable in practice [Hentenryck *et al.*, 1992] because of its space efficiency, we also compare the performance of our technique with the existing one in the context of AC-3. In particular, we compare two versions of AC-3: AC-3S and AC-3$^+$. AC-3S uses the special handling techniques in [Hentenryck, 1989] (it cannot use those in AC-5 because of the representation limitation of AC-3) for equations, inequalities and disequations. AC-3$^+$ uses our new method for IFCs without changing the techniques for non-IFC equations, inequalities and disequations. All the algorithms are implemented in CMU Common Lisp on SPARC-2.

We report two sets of tests. One set uses CSPs with only IFCs, in particular equations of the type: $X = aY + b$, where $X$ and $Y$ are variables, $a$ and $b$ are constants and $a$ is also positive. 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 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 performance comparison of AC-5 and AC-5$^+$ in percentage terms over 18 problems. The performance of AC-5 is taken as 100% (shown as the dash line). AC-5$^+$ takes only 52% to 80% of the time for AC-5 for arc consistency.

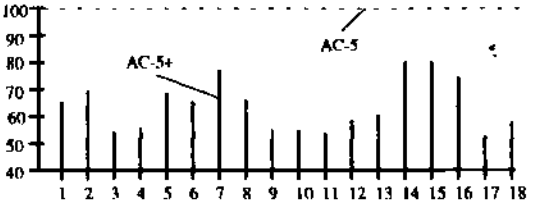


Figure 8. Comparison of AC-5 and AC-5$^+$ on IFCs

Figure 9 shows the comparison on the same set of IFC CSPs with AC-3S and AC-3$^+$. The performance of AC-3S is taken as 100%. AC-3$^+$ only uses 42% to 62% of the time taken by AC-3S. From the two figures, we can see that AC-3$^+$ improves more than AC-5$^+$. This is because AC-5 has a more efficient algorithm for functional constraints.

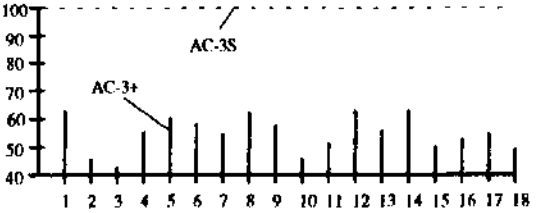


Figure 9. AC-3S and AC-3$^+$ on IFCs

The 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 this 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 $J_i = (Op_{i1}, Op_{i2}, ... Op_{im})$. The operations in each job have to be performed in a fixed sequence. Thus, the start time $S_{ik}$ of one operation must be greater than or equal to the ending time $E_{ik-1}$ of its previous operation, i.e., $S_{ik} \geq E_{ik-1}$. Each operation also has a duration $Du_{ik}$. Then, we have the constraint $E_{ik} = S_{ik} + Du_{ik}$ (which is a IFC). Each job has a due time $Dt_i$, by which it must be finished, i.e., $E_{im} \leq Dt_i$. 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. Figure 10 and 11 show the performance comparisons of AC-5 and AC-5$^+$, and AC-3S and AC-3$^+$ respectively (over 18 problems). It can be seen that the new technique produces substantial saving. AC-5$^+$ takes only 55% to 80% of the time for AC-5. AC-3$^+$ takes 42% to 57% of the time for AC-3S. On the surface, it seems that our technique should save less with the second set of tests because other types of constraints are involved. But from the figures, the savings are simmilar. This is because in the first set of tests, many IFCs may not be merged due to the condition in Section 4.

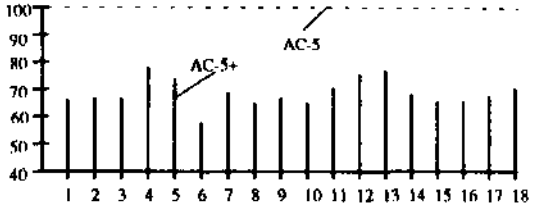

Figure 10. AC-5 and AC-5$^+$ on scheduling problems.

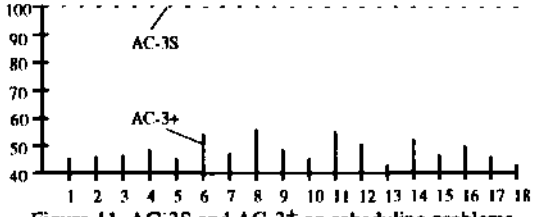

Figure 11. AC-3S and AC-3$^+$ on scheduling problems

The above saving is resulted from only a single arc consistency check. When applying the technique to constraint programming even more saving could be achieved due to backtrack search which requires consistency check to be performed many times.

## 6. Related Work

Many general and specific arc consistency techniques have been developed in the past. In real applications, specific methods perhaps play a more important role than general ones because of their efficiency. Most commonly used specific consistency methods are for functional, anti-functional, and monotonic constraints [Hentenryck, 1989; Hentenryck *et al.*, 1992; Mohr and Masini, 1988]. Their special cases also form the core of the current constraint programming languages, such as CHIP [Hentenryck, 1989], Charme [Charme, 1990], Ilog Solver [Ilog, 1993], etc.

AC-5 [Hentenryck *et al.*, 1992] is a generic arc consistency algorithm, i.e., it allows both general and specific checking of constraints. In [Hentenryck *et al.*, 1992], it is specialized for functional constraints, anti-functional constraints, monotonic constraints, and their piecewise constraints. The algorithm achieves O(*ed*) for these constraint classes. Our work is motivated by this algorithm. It is an improvement to AC-5's special technique for functional constraints. Although our technique is still *O(ed),* experiments have shown it is more efficient than that in AC-5.

Mohr and Masini [Mohr and Masini, 1988] discovered independently that binary equations, inequalities, and disequations can be solved in *O(ed)*. Earlier work on constraint solvers (e.g., ALICE (Lauriere, 1978]) and constraint programming languages (e.g., CHIP [Hentenryck, 1989]) also presented special algorithms for these types of constraints. However, equations in all these methods need to be checked many times.

## 7. Conclusion

We have proposed a new consistency technique for IFCs. It checks most IFCs only once rather than many times. Although this technique does not reduce the complexity, our experiments have shown it outperforms substantially the existing methods. The main application of this technique will be in constraint programming languages.

Acknowledgments: I would like to thank Kim-Heng Teo, Chee-Kit Looi and anonymous IJCAI reviewers for their helpful comments and suggestions.

## References

[Bessiere and Cordier, 1993] C. Bessiere and M Cordier. Arc-consistency and are-consistency again. *AAAI-93,* pages 108-113,1993.

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

[Dincbas *et al,* 1990] M. Dincbas, *et al.* Solving large combinational problem in logic programming. *Journal of Logic Prggramming,* 8:75-93,1990.

[Hog, 1993] *Hog Solver Reference Manual* ILOG, 1993.

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

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

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

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

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

[Perlin, 1991] M. Perlin. Arc consistency for factorable relataions. In *Proceedings of Third International Conf on Tools for AI,* pages 340-345,1991.

[Hentenryck, 1989] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, 1989.

[Hentenryck *et al,* 1992] P.V. Hentenryck, Y. Deville and C-M. Teng. A generic are-consistency algorithm and its specifications. *Artificial Intelligence,* 27:291-322. 1992.

[Waltz, 1972] D. Waltz. *Generating Semantic Descriptions from Drawings of Scenes with Shadows,* Tech Rept AI271, MIT, Cambridge, MA, 1972.