Improving Forward Checking with Delayed Evaluation§

Camilo Rueda and Frank D. Valencia
Grupo AVISPA,
Universidad Javeriana de Cali
Research Team AVISPA
Pontificia Universidad Javeriana Cali
Calle 18 Num. 118-250 Via a Pance
Cali, Colombia
Phone:(57) 2-5552175
Fax:(570 2-5552823
[crueda,fvalenci]@atlas.univalle.edu.co

Abstract

Several algorithms for solving constraint satisfaction problems (CSP) have been proposed in recent years. One of the most successful approaches involves using arc consistency as a pre-processing step followed by a form of backtrack search called *forward checking* (FC). Efficient algorithms for arc consistency based on the notion of *first support* have been proposed recently. We consider here this notion to be a natural consequence of delayed evaluation of constraint checks and apply it to FC. We present a specific algorithm, *first-found-forward-checking* (F3C), which takes advantage of the delayed evaluation of constraints. We prove that F3C can be no worst than FC, and show substantial reductions of constraint checks in practice. F3C has been effectively used to solve complex CSP's in a music composition system.

Keywords: constraints, constraint reasoning techniques, arc consistency, backtrack search, lazy evaluation.

1. Introduction

The use of simple forms of constraint reasoning has proved to be very effective in most practical applications. These techniques amount to a pre-processing step using arc consistency schemes (Mackworth[77]) to reduce variable domains followed by a form of back track search called *forward checking* (Haralick[80]). The standard formalism in constraint-based reasoning is a constraint graph defined by a set of variables (the nodes), a domain of values for each variable, and a set of constraints between the variables (the arcs). Efficient arc consistency algorithms (Mohr[86]) rest on the idea of computing for each value of a variable domain a set of supporting values. Supporting values are witnesses that constraints between variables hold. In Bessiere[94a,b] improvements in the time and space complexity of previous arc consistency algorithms are shown to be obtained by computing the set of supports incrementally. With a judicious choice of data structures it can be efficiently guaranteed that a particular element of the set of supports is generated only when it is really needed. The principle involved in this strategy, called *delayed evaluation*, makes part of the standard trade of functional programming (Abelson[85]). We use it to improve the standard forward checking (FC) algorithm.

FC attempts to reduce backtrack search by eliminating from the domain of some variables those values that are inconsistent with those already chosen for the other variables. It turns out that this process can also be seen as a set of supports computation in such a way that the idea of incrementallity applies. In fact, delayed evaluation of data structures can be profitably employed in several aspects of constraint reasoning for improving the time and/or space complexity of search algorithms.

Kondrak[94] proposes a characterization allowing to define theoretically an ordering of FC and various others backtracking algorithms according to their efficiency. We propose in this paper F3C (*first found forward checking*), a modification of FC with the incremental computation of supports and show that it ranks better than FC in that characterization.

---

§ This work has been partially supported by grant 12-51-14-041-95 from Colciencias-BID

The paper is organized as follows. Section 2 contains some preliminary definitions and properties on constraints networks. Section 3 describes F3C, section 4 proves the improvement over FC and gives experimental results, section 5 discusses the use of F3C and delayed evaluation in a system for computer aided composition. and section 6 contains conclusions.

2. Background

A *constraint network* (CN) is a triple $< V, D, C >$ where $V = \{x_1, x_2, \ldots, x_n\}$ is a set of variables, D is a set of values (the domain) and C is a set of constraints. We assume variables are identified by indexes in a set X where index $i \in X$ identifies variable $x_i \in V$. A given $CN =< X, D, C >$ is assumed throughout the paper. A *domain assignment* is a function $\delta: X \rightarrow P(D)$. $\delta(i)$ denotes the domain of values for variable x_i. Given assignments δ_1, δ_2, we say $\delta_1 \subseteq \delta_2$ when for all $i \in X$, $\delta_1(i) \subseteq \delta_2(i)$. Constraints are subsets of the products of their domains (that is, $R_{(i_1,i_2,\ldots,i_u)} \subseteq \delta(i_1) \times \ldots \times \delta(i_u)$). We say that values $v_{i_1} \in \delta(i_1), \ldots, v_{i_u} \in \delta(i_u)$, satisfy constraint $R_{(i_1,i_2,\ldots,i_u)}$ when $(v_{i_1}, \ldots, v_{i_u}) \in R_{(i_1,i_2,\ldots,i_u)}$. Given a domain assignment δ, a *value assignment set* (VAS) is a function $S: I \subseteq X \rightarrow D$ such that $S(i) \in \delta(i)$. A VAS represents the instanciation of variables with values in their corresponding domains. A VAS S is *consistent* if for each constraint involving variables instanciated by S the corresponding assignments satisfy the constraint. A *solution* of a CN is a consistent VAS instanciating all variables.

A constraint network is binary when all its constraints are binary relations. A binary network $< V, D, C >$ is usually represented as a directed graph $CG =< N, E >$ with $N = V$ and $(x_i, x_j) \in E$ iff $R_{(i,j)} \in C$. Arc consistency techniques are mostly used in binary constraint networks. The notion of *support* is fundamental in these techniques (Mohr[86]). Given a domain assignment δ, for each constraint $R_{(i,j)}$ we say that value $z \in \delta(j)$ is a *support* for value $v \in \delta(i)$ if $(v, z) \in R_{(i,j)}$. A value v_i is *viable* if for every $j \in X$ such that $R_{(i,j)} \in C$, v_i has support in $\delta(j)$. $\delta(i)$ is viable when all of its elements are viable. A domain D in a network is *arc consistent* when for all $i \in X$, $\delta(i)$ is viable.

Extending the notion of support to a VAS will allow us to neatly characterize backtrack search algorithms. We consider only binary networks to simplify notation, but the definitions can easily be extended to n-ary constraints. Having a constraint $R_{(i,j)} \in C$ and given a VAS S such that $j \notin dom(S)$ and a pair $(i, v_i) \in S$, v_j is a *support* for S if $(v_i, v_j) \in R_{(i,j)}$. A set A is a support for S if each element of A is a support for S.

Definition (viability and S-consistency).
A VAS S is *viable* in a domain assignment δ if for each $i \in dom(S)$ and for all variables $j \in X$ such that $j \notin dom(S)$ and $R_{(i,j)} \in C$, S has a support in $\delta(j)$. We assume VAS $\varnothing$ is viable. A domain assignment δ is *S-consistent* if for all $j \in X$, $\delta(j)$ is a support for S. A domain assignment δ is *maximal* S-consistent if it is S-consistent and for all S-consistent δ', $\delta' \subseteq \delta$ .Given a viable VAS S such that $j \notin dom(S)$ and $v_j \in \delta(j)$ a support for S, the VAS $S' = S \cup \{(j, v_j)\}$ is called an *extension* of S in δ.

Several backtrack search algorithms, including FC, work by extending consistent viable VAS's. Extending a VAS opens up at least three choice possibilities (we assume a given δ):

1. An element must be chosen from the set of candidate variables indexes $I(S) = \{j \notin dom(S) | j \in X\}$.
2. A support in the domain of the chosen variable must be selected from the set $\sigma_j(S) = \{v \in \delta(j) | v$ is a support for $S\}$.
3. When no extension is possible (i.e. S is not viable), a previously extended VAS $S' \subset S$ must be reconsidered.

Different backtrack search algorithms can be identified by the way in which they handle these sets. Chronological backtracking (BT, Bitner[75]) assumes predefined static orderings both on variables and on domains. The candidates chosen in $I(S)$ and in $\sigma_j(S)$ are the first elements according to their respective orderings. Similarly, when S is

not viable, a backtrack VAS $S' = S - \{(j, v_j)\}$ is selected, where x_j is the *last* variable in the ordering among those instanciated by S. Forward Checking performs essentially the same choices as BT does. The difference lies in that FC computes for a given VAS S *all* sets $\sigma_j(S)$, for every $j \in I(S)$. This allows to find out earlier whether S is viable or not and, additionally, reduces the sets of values to inspect for supports in subsequent extensions of S.

```
FCforward(S)
if |S|= n then output(S)  /*S is a solution */
                FCback(S)
else  repeat for j ∈ I(S) such that R(k,j) ∈ C
        σj(S) ← computeSupports(S, σj(S'))
      until σj(S) = ∅
      if ∃j ∈ I(S) such that σj(S) = ∅ then FCback(S)
      else j ← first(I(S))
           vj ← first(σj(S))
           σj(S) ← σj(S) - {vj}
           FCforward(S ∪ {(j,vj)})
FCback(S)
if S = ∅ then return()
else (j,vj) ← last(S)
     S' ← S - {(j,vj)}
     if σj(S') = ∅ then FCback(S')
     else v'j ← first(σj(S'))
          σj(S') ← σj(S') - {v'j}
          FCforward(S' ∪ {(j,vj)})
```

Figure 1: FC algorithm

FC (see Figure 1) computes a sequence of viable VAS's and corresponding sequence of domain assignments maintaining the following property:

Property (FC invariant).
Let $S_0 = \emptyset \subseteq S_1 \subseteq ... \subseteq S_k$ be the sequence of VAS's computed by FC when extending S_0 to S_k and let $\delta_0, \delta_1, ..., \delta_k$ be domain assignments such that for all $j \in X$, $\delta_i(j) = \sigma_j(S_i)$. For all $i = 1, ..., k-1$ the following holds:

1. S_i is viable. S_k is consistent.
2. δ_i is maximal S_i-consistent.
3. $\delta_{i-1} \supseteq \delta_i$

FC thus computes consistent *viable* VAS's by maintaining maximal S-consistent domain assignments. This extra reduction of domains is both simple and very effective (Nadel[85]). In an efficient implementation of FC, keeping all sets $\sigma_j(S)$ for every S can be avoided by a judicious choice of data structures but we do not consider these details here. In FC, *computeSupports* insures maximal S-consistency of domain assignments (we assume $\sigma(\emptyset) = \delta_\emptyset$). This is a costly operation. It takes $O(d)$ time for d the size of a variable domain. *computeSupports* is invoked for each constraint involving the last variable instanciated in the current VAS. This gives an $O(ed)$ computation, where $e = |C|$. It turns out that the necessary information provided by this computation (i.e. the viability of a VAS) can be obtained more efficiently with delayed exploration of the above mentioned sets. This optimization is in the spirit of that proposed by Bessiere[94a] for arc consistency computations.

3. First Found Forward Checking (F3C).

The new algorithm is based on the notion of *delayed-evaluation structure* (DES) which is essentially a *stream* in the sense of Abelson[85]. A DES can be seen as a sequence having an explicit representation of its first element and an

implicit representation of the other elements (it can thus be infinite). DES's are quite convenient to efficiently represent large data structures. We consider a DES as a data type with two operations: $first{:}DES \rightarrow element$, returns the first element of a DES. This operation is assumed to take constant time. $move{:}DES \rightarrow DES$ returns the DES resulting after discarding its first element. We use a DES to represent domain assignments. The key idea is to remark that the three properties of FC (see above) can be maintained without explicitly computing domain assignments. We only keep carefully chosen *witnesses* of each $\delta(j)$.

Definition (Witness set)
A *witness set* of a domain assignment δ is a function $\omega^{\delta}: X \rightarrow D \cup \{\bot\}$ (where $\bot \notin D$) such that for all $j \in dom(\delta)$ either $\omega^{\delta}(j) \in \delta(j)$ or $\omega^{\delta}(j) = \bot$. We extend the notion of S-consistency to witness sets. Given a VAS S, a witness set of an S-consistent domain assignment δ_S, written ω_S^{δ}, is such that $\omega_S^{\delta}(i) = S(i)$, for $i \in dom(S)$.

Given an ordering relation $\leq_X$ on X, a witness set can be expressed as a tuple $(\omega_S^{\delta}(i_1),\ldots,\omega_S^{\delta}(i_n))$ where $i_{k-1} \leq_x i_k$, $1 < k \leq n$. We also assume an ordering relation, $\leq_D$, on $D \cup \{\bot\}$ such that for all $d \in D, \bot \leq_D d$, and we order witness tuples lexicographically in the obvious way. For an S-consistent domain assignment δ_S, a useful S-consistent witness set of δ_S is $\varpi_S^{\delta}(j) = first(\delta_S(j))$, where *first* denotes the first element of each set in the domain assignment (we define $first(\varnothing) = \bot$). Given a VAS S, F3C uses ω_S^{δ} both to assert the viability of S and to provide a *delayed* representation of δ_S. As elements of δ_S are expanded they are inserted in a separate structure $\sigma(S)$. The following properties establish how this is done (when no confusion arises we drop superscripts for witness sets).

Proposition (F3C invariant)
Let $S_0 \subset \ldots S_k \subset S$ be the sequence of consistent VAS's and let $\omega_{S_0},\ldots,\omega_{S_k}$ and $\sigma(S_0),\ldots,\sigma(S_k)$ be corresponding witness tuples and support sets computed by F3C in extending S_0 to S. Let $\delta_{S_0} \supseteq \delta_{S_1} \supseteq \ldots \supseteq \delta_{S_k}$ be domain assignments such that δ_{S_i} is maximal S_i-consistent. The following holds:

1. $\sigma(S_i) \subseteq \delta_{S_i}$
2. ω_{S_i} is a witness set (tuple) of $\sigma(S_i)$ and $\omega_{S_{i-1}} \geq \omega_{S_i}$.
3. $\forall j \in X: \omega_{S_i}(j) = \max[\sigma_j(S_i)] > \bot$, where $\max[A]$ denotes the maximum element of set A in the pre-defined ordering (we define $\max[\varnothing] = \bot$).
4. If $d \in \delta_{S_i}(j)$ is such that $d \notin \sigma_j(S_i) \wedge d < \omega_{S_i}(j)$ then either every extension S' of S_i such that $S'(j) = d$ is not viable or else it has been output as a solution.
5. $S(k+1) = first(\sigma_{k+1}(S_k))$ is a support of S.

Property 1 says that F3C maintains witnesses of the viability of computed VAS's. Properties 2 and 3 assert that witness sets act as lexicographically moved *markers* of the known portion of delayed-evaluated consistent domain assignments. Property 4 guarantees that no viable element less than a witness is missed by an explicit part of a domain assignment. Property 5 asserts that extensions are consistent.

As mentioned above, domain assignments are represented "lazily". $\delta_{\varnothing}$ is the initial DES. Given a VAS S, the S-consistent domain assignment is represented by an explicit and an implicit part. The explicit part $\sigma(S)$ contains elements of $\delta_{\varnothing}$ already known to be S-consistent. The implicit part $\delta_S \subseteq \delta_{\varnothing}$ is a DES of currently untested potential supports for S. Both structures are related by $\delta_S(j) = \varnothing \vee successor_j^{\delta_{\varnothing}}(\omega_S(j)) = first(\delta_S(j))$, where $successor_j^{\delta}(x)$ is the element of $\delta(j)$ following x in the domain ordering. Procedure *nextSupport* (see figure 3) updates these structures when a viable VAS S is extended (*F3CForward*) or when non viability has been proved and a different extension of an $S' \subset S$ VAS must be tried (*F3CBack*), thus checking viability of consistent extensions. The following specification captures this behaviour of *nextSupport* more precisely.

Proposition (*nextSupport* postcondition)

Let $\sigma_j(S), \omega_S(j), \delta_S(j)$ be the input and $\sigma'_j(S), \omega'_S(j), \delta'_S(j)$ the computed structures in an invocation of *nextSupport*. Let $S_0 \subset ... S_i \subset ... S$ be the sequence of extensions leading to S. The following properties hold:

1. For all S_i, $\omega'_{S_i}(j) = last(\sigma'_j(S_i))$.
2. For all $S_i \subset S$, the set $\{v \in \sigma'_j(S_i) | \omega_{S_i}(j) \leq v < \omega'_{S_i}(j)\}$ is not a support for S.
3. If there is no $v > first(\sigma_j(S))$ in $\delta_\varnothing(j)$ such that v is support for S then $\sigma'_j(S) = \varnothing$.
4. Either $\sigma_j(S) \neq \varnothing$ and then $\sigma'_j(S) = \sigma_j(S) - \{first(\sigma_j(S))\}$, or $\sigma_j(S) = \varnothing$ and then we have $\{v | first(\delta_S(j)) < v < first(\delta'_S(j)) \wedge v \text{ is support of } S\} = \varnothing$, $\omega'_S(j)$ is the first support of S in $\delta_S(j)$, and $\sigma'_j(S) = \{\omega'_S(j)\}$.

```
F3Cforward(S)
1 if |S|= n then output(S)  /*S is a solution */
2              F3Cback(S)
3 else repeat for j ∈ I(S) such that R(k,j) ∈ C
4        ω_S(j), σ_j(S) ← nextSupport(S, j)
5      until ω_S(j) = ⊥
6      if ∃j ∈ I(S) such that ω_S(j) = ⊥ then F3Cback(S)
7      else j ← first(I(S))
8           F3Cforward(S ∪ {(j, first(σ_j(S)))})
F3Cback(S)
1 if S = ∅ then done()
2 else (j, v_j) ← last(S)
3      S' ← S − {(j, v_j)}
4      ω_S'(j), σ_j(S') ← nextSupport(S', j)
5      if ω_S'(j) = ⊥ then F3Cback(S')
6      else F3Cforward(S' ∪ {(j, first(σ_j(S')))})
```

Figure 2. F3C algorithm

```
nextSupport(j,S): returns ω(j), σ_j
v ← moveToNext(σ_j(S))
found ← false
while v ≠ ⊥ ∧ ¬found do
  isSupport ← true
  for (i,x) ∈ S, i = 0 to |S|-1 while isSupport do
    if v > ω_Si(j) then if (x,v) ∈ R(i,j)
                          then σ_j(S_i) ← σ_j(S_i) ∪ {v}
                               ω_Si(j) ← v
                          else isSupport ← false
                   else skip
    if isSupport then found ← true else v ← moveToNext(σ_j(S))
return (σ_j(S), ω_S(j)))
```

Figure 3. Procedure *nextSupport*

```
moveToNext(σ_j(S)): returns v
 v ← first(σ_j(S))
 if v = ⊥ then v ← first(δ_S(j))
                 if v ≠ ⊥ then δ_S(j) ← move(δ_S(j))
 else σ_j(S) ← σ_j(S) − {v}
 return (v)
```

Figure 4. Procedure *moveToNext*.

Procedure *moveToNext* (see figure 4) performs a *move* operation on the DES representing the appropriate domain assignment, when this is needed.

We discuss next some properties supporting the correctness of F3C. We only sketch proofs. We say a VAS S is *considered* in step 1 of *F3Cforward* or step 4 of *F3Cback* (see Figure 2). A VAS $S' \subset S$ is said *visited* . By the properties mentioned before, a visited VAS is viable and a considered VAS is consistent.

Lemma 1.
Let $S_0 \subset ... \subset S_k$ be a sequence of VAS's such that S_k is the currently considered VAS. If $\sigma_{k+1}(S_k) \neq \varnothing$ then S_k is viable, has been visited, and the extension $S_k \cup \{(k+1, first(\sigma_{k+1}(S_k)))\}$ has been considered.
proof : Only an invocation of $nextSupport(k+1, S_k)$ can add values to $\sigma_{k+1}(S_k)$. This cannot have been in the current invocation of *F3Cforward* or *F3Cback* since S_k is currently considered, so there must has been a previous consideration of S_k. Only *F3Cback* reconsiders VAS's, so step 4 of *F3Cback* is being performed. But then $S_k = S - \{(k+1, first(\sigma_{k+1}(S_k)))\}$, S is consistent and S_k is viable $\Diamond$.

Lemma 2.
If S is visited by F3C then all consistent direct extensions are considered.
proof : Let $S_0 \subset ... S_{k-1} \subset S$ be the extensions computed by F3C such that S_{k-1} is viable (visited), S is consistent and not visited by F3C. There are three reasons for not considering S:
1. $v = S(k)$ is not a support for S_{k-1}. But then S cannot be consistent.
2. Extension $(k,v) \in S$ is a support but $v < first(\sigma_k(S_{k-1}))$. Since v is a support, the only reason $v \notin \sigma_k(S_{k-1})$ is that $nextSupport(k, S_{k-1})$ moved passed it. But then $\sigma_k(S_{k-1}) \neq \varnothing$ and by lemma 1 extension $S_{k-1} \cup \{(k,v)\} = S$ would have been considered before, contradicting the assumption.
3. Extension $(k,v) \in S$ is a support but $v > \omega_{S_{k-1}}(k)$. F3C can only terminate in a recursive invocation of *F3Cback* since *F3Cforward* never changes a VAS before invoking *F3Cback* and the empty VAS is viable by definition. Since S_{k-1} is visited, a call $F3Cback(S_{k-1})$ from *F3Cback* must be performed. But this can only happen when $nextSupport(k, S_{k-1})$ returns $\omega_{S_{k-1}}(k) = \perp$, which means that $\delta_{S_{k-1}}(k)$ is the empty DES. But then, by the property linking a DES and its explicit part, *nextSupport* must have moved passed every support for S_{k-1} in $\delta_{S_{k-1}}(k)$. By lemma 1, S would have been considered, which is a contradiction $\Diamond$.

Corollary.
F3C visits all (and only) viable VAS's.
proof : follows from the lemmas above.

Theorem 1.
F3C finds all (and only) solutions.
proof: ($\Rightarrow$) Let S be a solution. Then S is a consistent extension of an S', which must be viable. By lemma 1, S is considered. Since $|S| = n$, it is output as a solution by *F3Cforward*. ($\Leftarrow$) Let S be a VAS output by *F3Cforward*. Let $S_0 \subset ... S_{k-1} \subset S$ be the extensions leading to S. Then $|S| = n$, by the corollary S_{k-1} is viable and by lemma 1 S is consistent. Thus S is a solution $\Diamond$.

4. Efficiency of F3C.

The characterization proposed in Kondrak[94] considers VAS's as nodes in a tree. This tree represents the search space of a backtrack algorithm. Using as metrics the number of tree nodes (i.e. VAS's) visited and the number of constraints checked on each visit, Kondrak[94] is able to define a precise hierarchy for the efficiency of several backtrack search algorithms: Chronological Backtrack (BT), back jumping (BJ), back marking (BM), conflict-directed back jumping (CBJ), forward checking (FC), really full look-ahead (RFL) and some hybrids (e.g. Backmarking + Backjumping, BMJ). Given algorithms P and Q, P appears below Q (i.e. it is better) in the hierarchy if Q visits all VAS's that P visits and P performs less constraint checks than Q in each visited VAS. F3C can be placed below FC in this hierarchy. In Kondrak[94] it is shown that FC *visits* a node (*considers* a VAS) iff its father (the VAS it is an extension of) is consistent with all the variables (is viable). The next theorem follows immediately.

Theorem 2.
FC considers all VAS's that F3C considers.

Theorem 3.
FC performs on each considered VAS at least as much constraint checks as F3C does.
proof: Let $S_0 \subset ... S_{k-1} \subset S$ be the sequence of extensions computed by F3C, with S_{k-1} viable and S being considered. Let $(v,w) \stackrel{?}{\in} R_{(i,j)}$, $i \le k < j \le n$ be a constraint checked in an invocation of $nextSupport(j,S)$. Constraint $R_{(i,j)}$ is checked only once in a given invocation since w is eliminated from $\sigma_j(S_i)$ If $i = k$, then $w \in \delta_S(j)$, a potential support of v, is being tested. Since FC tests *all* potential supports in (the explicit set) $\delta_S(j)$, it must have checked $R_{(i,j)}$. If $i < k$ a potential support of a previously considered VAS S_i is tested. We must also have $w > \omega_{S_i}(j)$, so (v,w) cannot have been checked before in the current sequence of extensions leading to S. Also, since $nextSupport$ considers constraints in the underlying variable ordering, w must be a support for all $S_j \subset S_i$. This means w would be a candidate for FC's $\sigma_j(S_i)$ if FC considered S_i. By theorem 2, this must be the case. But then FC must have also checked constraint $R_{(i,j)}$, which proves the theorem. We may further notice that FC would have *always* checked (v,w) , while in F3C this only happens if $w > \omega_{S_i}(j)$. Thus F3C might perform *less* constraint checks than FC $\Diamond$.

Figure 5 shows F3C in the hierarchy of Kondrak[94]. RFL was added to this hierarchy in Valencia[95]. Dashed lines express our conjecture regarding F3C+CBJ.

In our experiments we have found that the performance of F3C is quite stable, scoring consistently better than the other algorithms in most of our tests. We tested F3C on two bench mark problems and on a set of randomly generated problems. As benchmarks we used the puzzle problem ZEBRA and a combinatorial theory problem (CTP), both described in Vanhent[89]. For the randomly generated problems a set of parameters are defined to control different aspects of the constraint graph (Sabin[94], Bessiere[93]). $0 \le pc \le 1$ denotes the probability of having a constraint between a pair of variables, $0 \le pu \le 1$ is the probability that some constraint hold between a given pair of values, d is the domain size and n is the number of variables.

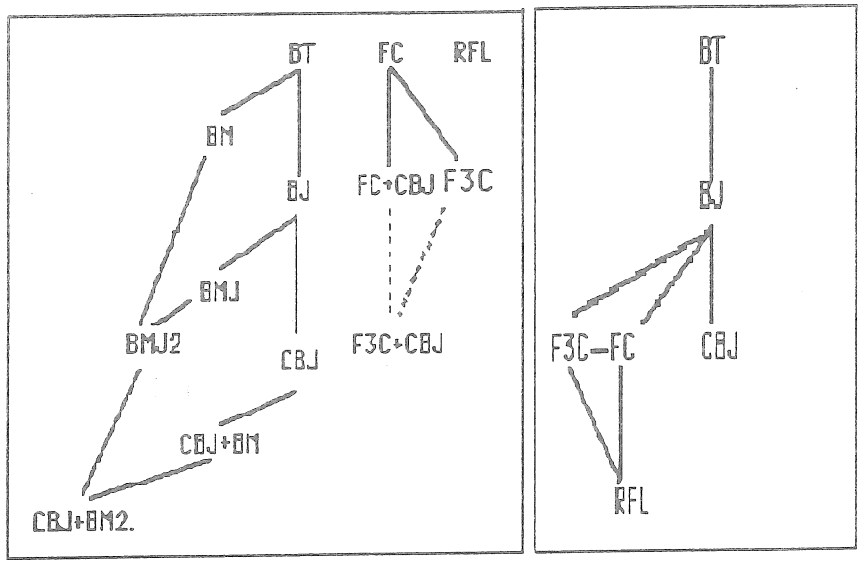

Figure 5. Algorithm hierarchy based on constraint checks (left) and considered VAS's (right) .

For our experiments we used an implementation of F3C with an explicit representation of δ_S (i.e. no DES's) and a matrix of markers pointing to witnesses of each $\delta_S(j)$. We also avoided creating copies of domains by maintaining explicitly the set of deleted (i.e. non supports) values in each domain and used it to reinstall them as

needed on backtracking. Results are shown in tables 1-3. We analyzed FC and F3C, both using the min-domains heuristic, i.e. the variable having the smallest current domain is the one selected next for extension. Results of these tests are given in tables 4 and 5.

ALGORITHM	Constraint Checks	Number of VAS's considered
CBJ	939	127
F3C	1037	144
BMJ	1227	267
FC	1329	144
BJ	2066	267
BT	2530	339

Table 1. Constraint checks and VAS's considered for the problem ZEBRA .

ALGORITHM	Constraint Checks	Number of VAS's considered
F3C	124858	8626
BMJ	127015	14582
FC	164557	8626
CBJ	620824	14487
BJ	624377	14582
BT	6668616	15733

Table 2. Constraint checks and VAS's considered, for CTP

Pu	FC	F3C	CBJ	BJ	BMJ	BT
5	30	28	50	583	583	708
10	48	36	272	351	264	771
15	65	46	48	52	47	52
20	131	97	195	267	188	967
25	103	67	232	311	242	2420
30	228	150	482	565	271	1245
35	284	188	293	1401	359	4608
40	991	604	3788	10548	3590	24960
45	860	515	3808	8230	3653	11757
50	2093	1418	4722	13477	4195	27158
55	3211	2169	15481	32635	5972	176578
60	8461	5667	36540	132939	29623	544724
65	15067	9573	38846	133502	26479	345203
70	503	357	620	2396	577	10542
75	808	716	557	6277	712	15108
80	190	106	436	436	209	1071
Total	33073	21737	106370	343970	76964	1167872

Table 3. Random problems with $pc = 30\%, n = 20, d = 5$ and $0.05 \leq pu \leq 0.8$.

Pu	FC(Mindom)	F3C(Mindom)	FC/F3C
5	3849	2050	0.53
10	6068.7	3386	0.56
15	10696.13	4912.5	0.46
20	15804.4	7498.6	0.47
25	35814.7	17536.2	0.49
30	44856.9	25409.8	0.57
35	103653	67919.1	0.66
40	194373.5	119939.5	0.62

45	1478361.3	1147081.6	0.78
50	3297041	3235978.3	0.98
55	13085	2658	0.20
60	3973.1	1627.2	0.41
65	1446.2	876.6	0.61
70	1590	530.6	0.33
75	1658.8	350	0.21
80	1851	385	0.21
85	2015.6	156	0.08
90	2526	160	0.06
95	2495.6	121.8	0.05
100	2940	98	0.03
Total	5224099.93	4638674.8	

Table 4. Constraint checks for FC and F3C with min-domain heuristics. Parameters are: $n = 20$, $d = 30$, $pc = 0.5$.

It can be seen in the tables that F3C outperforms FC both with or without the min-domains heuristics. An increase of about 12% in the first case and 25% in the second is observed. If we ignore the instances laying just at the border between under-constraint and over- constraint problems ($pu \approx 0.5$), the performance increase with min-domains is about 40%.

5. F3C in a musical composition system.

F3C has been used to improve the constraint engine of a visual environment for computer aided musical composition, called *Niobé* (Assayag[93]). The musical domain is particularly demanding for constraint satisfaction techniques: The number of variables and the domain size tend to be very large (e.g. a variable might denote a chord in a section of a piece), progressions from under-constraint to over-constraint subsequences might be *precisely* what the composer looks for and constraints are usually considered to have different degrees of importance. *Niobé* applies different constraint satisfaction techniques. Solution searching used originally FC with an embedded technique for handling *soft* constraints (Schiex[92]). Delayed evaluation is extensively used in *Niobé* : Domains are represented implicitly, and all pruning is performed "lazily". In fact, the lazily evaluated *AC-5* in *Niobé* can be straightforwardly adapted to behave as *AC-6* (Bessière[94]).

6. Conclusions.

We presented F3C, a new backtrack search algorithm based on the notion of *first support* defined in Bessière[94] for arc consistency computation. We described the key properties of FC and argued how they can be maintained more efficiently by keeping only a *witness* from each domain. We showed how this concept can be formally adapted to FC by representing domains as *streams* and considering delayed constraint filtering. This characterization allowed us to place F3C in the hierarchy proposed in Kondrak[94] and thus to prove the improvement of F3C over FC. In our experiments the improvement over FC was clearly demonstrated in terms of a smaller number of constraint checks, both with or without the min-domains heuristics. Comparison of F3C (*without* min-domains) with several other well known backtrack search algorithms shows that F3C consistently ranks first in most cases. We described a system for music composition in which the idea of delayed evaluation is profitably used in several aspects of the underlying constraint reasoning engine.

7. References

Abelson[85] Abelson, H., Sussman, G. J. *Structure and Interpretation of Computer Programs*. MIT PRESS, 1985.

Assayag[93] Assayag, G., Rueda, C. The Music representation Project at IRCAM. *Proc. icmc 93*. Tokyo, Japan, 1993.

Bessière[94a] Bessière, C., Regin, J. C. using Biderectionality to Speed up Arc Consistency Processing. *Proc.* ECAI'94.

Bitner[75] Bitner, J. R., Reingold, E. Backtrack Programming techniques. Comm. ACM, 18(1975)651-656.

Deville[91] Deville, Y., Van Hentenryck, P. An Efficient Arc-Consistency Algorithm for a Class of CSP Problems. *Proc.* IJCAI'91.

Haralick[80] Haralick, R. M. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Information Sciences* 14 (1978) 263-314.

Kondrak [94] A Theoretical Evaluation of Selected Backtracking Algorithms. TR94-10, University of Alberta, June 1994.

Mackworth[77] Mackworth, A. K. Consistency in Networks of Relations. *Artificial Intelligence* 28 (1986) 128-233.

Mohr[86] Mohr, R., Henderson, T. C. Arc and Path Consistency Revisited. Artificial *Intelligence* 28 (1986) 128-233.

Nadel[89] Nadel, B. Constraint Satisfaction Algorithms. *Computation Intelligence* 5 (1989) 188-224.

Sabin[94] Sabin, D., Freuder, E. Contradicting Conventional Wisdom in Constraint Satisfaction. *Proc. ECAI'94.*

Schiex[92] Schiex, T. Possibilistic Constraint Satisfaction Problems or "How to handle soft constraint?". Personal communication, Paris, 1992.

Valencia[95] Valencia, F., Castaño, G. Problemas de satisfacción de Restricciones: Unificación Formal y Nuevos Algoritmos. AV-95-03, Grupo AVISPA, Universidad Javeriana de Cali, 1995.

Vanhent[89] Van Hentenryck, P. Constraint satisfaction in Logic programming. MIT PRESS, 1989.