Extending Forward Checking

Fahiem Bacchus[1]

Department. of Computer Science, 6 Kings College Road, University Of Toronto, Toronto, Ontario, Canada, M5S 1A4, fbacchus@cs.toronto.edu ⋆

Abstract. Among backtracking based algorithms for constraint satisfaction problems (CSPs), algorithms employing constraint propagation, like forward checking (FC) and MAC, have had the most practical impact. These algorithms use constraint propagation during search to prune inconsistent values from the domains of the uninstantiated variables. In this paper we present a general approach to extending constraint propagating algorithms, especially forward checking. In particular, we provide a simple yet flexible mechanism for pruning domain values, and show that with this in place it becomes easy to utilize new mechanisms for detecting inconsistent values during search. This leads to a powerful and uniform technique for designing new CSP algorithms: one simply need design new methods for detecting inconsistent values and then interface them with the domain pruning mechanism. Furthermore, we also show that algorithms following this design can proved to be correct in a simple and uniform way. To demonstrate the utility of these ideas five "new" CSP algorithms are presented.

1 Introduction

Many of the most useful backtracking based (systematic) CSP algorithms utilize constraint propagation to detect inconsistent values during search. These inconsistent values can then be temporarily pruned from the domains of those variables, and restored when the search backtracks and nullifies the reason the values became inconsistent. In terms of simplicity and historical precedence the most basic algorithm in this class is Haralick and Elliott's forward checking algorithm (FC) [1].

Although conceptually simple, pruning values during search has rather profound computational effects. Different amounts of computation can be devoted to detect inconsistent values. But, once a value is known to be inconsistent it is easy to prune it (e.g., it can be delinked from a linked list containing the variable's current values), and this can yield exponential savings in the search of the subtree below— if we had not pruned it we might have had to, e.g., process this value an exponential number of times in subtree below. Perhaps an even more important benefit is that domain pruning causes the domain sizes of the uninstantiated variables to vary dynamically, and this provides invaluable input to dynamic variable order (DVO) heuristics [2].

However, popular algorithms like FC and MAC employ domain pruning in a fairly restricted way. In both of these algorithms pruning is done in lock step with search. That is, every time a new assignment $\mathcal{A}$ is made these algorithms perform constraint propagation, detect some set of inconsistent values, and prune those values to the *current* level. Hence, when we undo $\mathcal{A}$, all of these values are restored. As a result the domains

⋆ This research was supported by the Canadian Government through their NSERC program.

of the uninstantiated variables are in the same state when we backtrack from the current level as when we first entered it.

In this paper we show that there are other opportunities for detecting and pruning inconsistent values, and we use these new opportunities to develop an interesting range of extensions to the standard algorithms. It is very simple to implement a more general domain pruning and restoring mechanism that allows one to prune a value to an arbitrary level of the search tree. With this extra flexibility in place, one can set to the task of developing ways in which to use it. And as we will demonstrate in this paper there are a number of ways to do so. That is, there are ways of detecting that a value is inconsistent with levels higher than the current level, and thus pruning it above the current level.

In the sequel we present a generic template for extending constraint propagation algorithms. This template has an abstract interface to a domain pruning mechanism (that mechanism must do at least as much pruning as would FC). By designing and implementing new pruning mechanisms we can combine them with this template to obtain new CSP algorithms that extend FC. We then present a theorem which shows that as long as the new pruning mechanism satisfies a simple soundness criteria, the algorithm that arises from combining it with the template is both sound and complete. This means we can easily verify the correctness of our new algorithms (or any future algorithms that employ the same design technique). We then present five "new" CSP algorithms, all of which extend forward checking by taking advantage of various insights into when we can detect that values have become inconsistent. We are still working on evaluating these new algorithms empirically, but we present some preliminary results indicating potential in terms of improved search efficiency.

Given the intensity of research into new CSP algorithms, it should not be too surprising that some of these new algorithms end up being similar to previous proposals. However, an important contribution of our approach is that it provides a unification and simplification of a number of these previous proposals. This in itself has important practical significance: methods for cleanly designing and implementing CSP algorithms have as much of a role to play in practice as improved algorithmic efficiency. Furthermore, despite the relationship with previous proposals, some of the algorithms we present are novel and they all have novel features. Our approach also opens the door for the discovery of further improvements: any sound new method for discovering value inconsistencies can easily be plugged into our template to yield a new CSP algorithm.

2 Notation and Background

A CSP consists of a set of variables $\{V_1, \ldots, V_n\}$ and a set of constraints $\{C_1, \ldots, C_m\}$. Each variable V has a domain of values $\mathit{Dom}[V]$, and can be assigned any value $v \in \mathit{Dom}[V]$, indicated by $V \leftarrow v$.

Let $\mathcal{A}$ be any set of assignments. A variable can only be assigned a single value, hence the cardinality of $\mathcal{A}$ is at most n, $\|\mathcal{A}\| \leq n$. When $\|\mathcal{A}\| = n$ we call it a *complete* set of assignments. Associated with $\mathcal{A}$ is a set $\mathit{VarsOf}(\mathcal{A})$, the set of variables assigned values in $\mathcal{A}$.

Each constraint C is over some set of variables $\mathit{VarsOf}(C)$, and its arity is $\|\mathit{VarsOf}(C)\|$. A constraint is a set of sets of assignments: if the arity of C is k, then each element of C is a set of k assignments, one for each of the variables in $\mathit{VarsOf}(C)$. We say that a set of assignments $\mathcal{A}$ *satisfies* a constraint C if $\mathit{VarsOf}(C) \subseteq \mathit{VarsOf}(\mathcal{A})$ and there

```
FC+Prune(level)
 1. Var := picknextVar();
 2. if(Var == NIL)
 3.    processSolution();
 4.    if(FINDALL)
 5.        Prune::FoundSoln(level);
 6.       unassignlevels(level-1,level-1);
 7.       return(level-1);
 8.    else
 9.       return(0);
10. foreach val ∈ CurDom[Var]
11.    assign(Var,val,level);
12.    Prune::Asgn(Var,val,level);
13.    jbl := FC+Prune(level+1);
14.    if(jbl != level)
15.       return(jbl);
16. jbl := maxprlevel(Var);
17. Prune::Backup(jbl,Var,level);
18. unassignlevels(level,jbl);
19. return(jbl);

picknextVar()
 1. if(FutureVars == {})
 2.    Var := NIL;
 3. elseif ∃ V.V ∈ FutureVars ∧ CurDom[V] == {}
 4.    Var := pick V ∈ FutureVars such that
                CurDom[V]={}
 5. else
 6.    Var := pick some V ∈ FutureVars;
 7. return(Var);

assign(Var,val,level)
 1. FutureVars := FutureVars - {Var};
 2. asgnVal[Var] := val;
 3. VarAtLevel[level] := Var;

unassignlevels(l1,l2)
 1. for i := l1 to l2
 2.   Var := VarAtLevel[i];
 3.   VarAtLevel[i] := NIL;
 4.   asgnVal[Var] := NIL;
 5.   FutureVars := FutureVars ∪ {Var};
 6.   foreach v ∈ PrunedVals[i]
 7.     CurDom[v.Var] := CurDom[v.Var] ∪ {v};
 8.     v.prlevel = NIL;
 9.     v.CF = {0};
10.   PrunedVals[i] := {};

maxprlevel(Var)
 1. m := 0
 2. foreach v ∈ Dom[Var]
 3.    m := max(m,v.prlevel);

prune(v,level)
 1. CurDom[v.Var] := CurDom[v.Var] - {v};
 2. PrunedVals[level] := PrunedVals[level] ∪ {v};
 3. v.prlevel := level;
```

Fig. 1. A Template for Extending Forward Checking

exists an element of C that is a subset of $\mathcal{A}$. Furthermore, we say that $\mathcal{A}$ is *consistent* if it satisfies all constraints C such that *VarsOf*$(C) \subseteq$ *VarsOf*$(\mathcal{A})$. That is, it satisfies all constraints it *fully instantiates*. A *solution* to a CSP is a complete and consistent set of assignments.

An assignment $V \leftarrow v$ is *consistent* with a set of assignments $\mathcal{A}$ if $\mathcal{A} \cup \{V \leftarrow v\}$ is consistent. A *value* v (of some variable V) is consistent with $\mathcal{A}$ when $V \leftarrow v$ is consistent with $\mathcal{A}$, otherwise it is *inconsistent*. Finally, if we have a constraint C with $\{V, V'\} \subseteq$ *VarsOf*(C) then two values $v \in$ *Dom*$[V]$ and $v' \in$ *Dom*$[V']$ are said to be *compatible* (given some set of assignments $\mathcal{A}$ to the other variables of C) if $\{V \leftarrow v, V' \leftarrow v'\} \cup \mathcal{A} \in C$, in this case we also say that v *supports* v' (given $\mathcal{A}$) and vice versa.

3 A Template for Extending Forward Checking

FC+Prune is template for constraint propagating algorithms that find solutions by searching in a tree of variable assignments. It utilizes the following data structures. *FutureVars*, the set of uninstantiated variables. *asgnVal*, a map from a variable to its currently assigned value. *VarAtLevel*, a map from a level of the search tree to the variable assigned at that level. *CurDom*, a map from a variable to a set containing all of the values still available for it at this point of the search (i.e., the set of unpruned values of the variable). *Dom*, a map from a variable to its original domain of values. *PrunedVals*, a map from a level of the search tree to a set of values that have been pruned to this level.

In order to more flexibly prune and restore values at different levels of the tree we represent each value, v, by a structure that includes the fields $v.val$, the actual numeric or symbol value associated the value, $v.Var$, the variable the value is for, $v.prlevel$, the level to which v has been pruned back to, and $v.CF$ the conflict associated with v. How these data structures are used during search is explained below.

Search is initiated by the call **FC+Prune**(1). At every level the next variable to assign is selected (heuristically) by `picknextVar`. `picknextVar` returns `NIL` if there are no more future variables. Given the conditions specified below, this will mean that the current set of assignments, *CurAsgns*, is a solution, the last assignment of which was made at the previous level. The solution will be enumerated by `processSolution` (line 3). If we want to enumerate all solutions `FINDALL` is set to be true which will cause a return to the previous level to search for more solutions. Otherwise 0 will be returned (line 9) causing the recursion to unwind and terminate.

If we haven't found a solution, we proceed to examine every value in the current domain of the selected variable (line 10). For each value `val` we augment *CurAsgns* by making the assignment `Var` $\leftarrow$ `val` using `assign`. (`assign` updates the *FutureVars*, *asgnVal*, and *VarAtLevel* data structures). We then recursively examine the search tree under this assignment. The recursion will return a jump back level, and if that level is above the current level, we return to a previous invocation of **FC+Prune** higher up the search tree. Otherwise we continue with the ***foreach*** loop to test `Var`'s next value.

Once we have exhausted all possible values, we compute the deepest level to which we can backtrack so as to restore at least one value of `Var`'s domain (line 16),[1] undo

[1] The code maintains the invariant that if a value is a member of *PrunedVals*$[i]$ its *.prlevel* field has been set to i (all values are pruned by calling the function `prune`). Thus if we jump back

all of assignments in between these levels (`unassignlevels` updates the same data structures as `assign`), and then jump back. One important feature of our version of tree search is that this method of computing the level to backtrack to is uniformly applied even when sophisticated forms of backjumping are used. [2]

Flexible domain pruning is implemented by the simple device of maintaining the sets *PrunedVals* and *CurDom*. Given that we have detected that a value has become inconsistent with the assignments made at some prior level ℓ, we can prune that value by removing it from the *CurDom* of its variable, and placing it in the set *PrunedVals*$[\ell]$. When search backtracks to level ℓ (thus nullifying the reason the value became inconsistent) it restores all of the values in *PrunedVals*$[\ell]$ by moving these values back into the *CurDom* set of their associated variable. This processing is done by the functions `prune` and `unassignlevels`. It can also be noted that in some cases we might detect that a value has become inconsistent back to level 0 (the level above the first assignment). In this case we put the value in *PrunedVals*$[0]$ and it will never be restored.

One simplification used in **FC+Prune** is that it does not immediately check the result of pruning the future domains. If some future variables has its domain entirely deleted by **Prune::Asgn** (line 12), i.e., if a DWO has occurred, **FC+Prune** will detect this in its next recursive invocation: `picknextVar` always returns a variable with an empty domain if one exists. By doing this we simplify the code and the interface to the domain pruning mechanism.

FC+Prune allows for an interaction with an arbitrary domain pruning mechanism in three different locations: when we have found a solution **Prune::FoundSoln** (line 5), when we assign a value **Prune::Asgn** (line 12) and when we backtrack after having exhausted the domain of a variable **Prune::Backup** (line 17). At these three locations new inconsistent values can be discovered and pruned. All of the algorithms we present can be implemented by combining the **FC+Prune** template with an instantiation of these three subroutines.

3.1 Soundness and Completeness

Subject to two simple conditions we can show that any CSP algorithm generated by combining **FC+Prune** with an instantiation of the domain pruning subroutines is both sound and complete.

1. **Prune::Asgn** must remove all values of the future variables that are inconsistent with *CurAsgns*. This means we can consistently extend *CurAsgns* by assigning any future variable any value from its current domain. [3]
2. The domain pruning process must be *sound*. That is, at every stage of the search if a value v from the domain of variable V is in *PrunedVals*$[\ell]$ for some ℓ, then there can be no *unenumerated* solution in the subtree below level ℓ containing $V \leftarrow v$.

---

to the maximum of these values (computed by `maxprlevel`), we will restore at least one value.

[2] Sophisticated backwards moves are handled by sophisticated ways of doing domain pruning.

[3] For binary CSPs this means that the domain pruning mechanism must do at least as much pruning as FC. For n-ary CSPs, it must do at least as much pruning as the version of FC defined by van Hentenryck [3].

That is, pruning can only eliminate a value from a subtree when it is certain that the value cannot participate in any further solutions in that subtree. [4]

Theorem 1. *Subject to these conditions on the pruning subroutines* **FC+Prune** *is sound, i.e., all solutions it reports are in fact solutions, and complete, i.e., it will enumerate all solutions if* `FINDALL` *is true and it will enumerate one solution if* `FINDALL` *is false and there exists a solution. It will not enumerate any solutions if and only if none exist.*

All of the algorithms we present here satisfy the first condition (they all extend forward checking), and thus all we have to do to show them to be sound and complete is to demonstrate the second condition.

4 Some new CSP Algorithms

All of our new algorithms are developed by insights into new sound ways of detecting inconsistent values and detecting the level at which they became inconsistent (which might be above the current level). Inconsistent values can then be pruned back to their level of inconsistency. These algorithms can all be implemented by simply specifying the three pruning routines. In this section we present five new CSP algorithms designed in this way. All of these algorithms can be formalized to deal with n-ary CSPs. However, we will restrict our presentation to binary versions of the algorithms. The binary versions are easier to understand and more concise to present.

4.1 Extending FC

The first two algorithms we present are simple extensions of FC. After a new assignment $V' \leftarrow v'$ is made at level ℓ, FC prunes all values of the future variables that are inconsistent with this assignment. In FC these inconsistent values are pruned to level ℓ (i.e., placed in *PrunedVals*$[\ell]$). Thus these values will be restored as soon as we backtrack to level ℓ. One way of extending FC is illustrated by the following example.

Example 1. Say that we have two variables A and B, each with the domain of values $\{1, 2, 3\}$ and the constraint $A \geq B$. Further, say that along the current path of the search tree we instantiate A at level 5, that the value 3 of A has already been pruned by a prior assignment at level 1, and that no other values of A have been pruned. Hence at level 5 *CurDom*$[A] = \{1, 2\}$, $3 \in$ *PrunedVals*$[1]$, and $3.prlevel = 1$.

Say that we next make the assignment $A \leftarrow 1$, and then forward check the unassigned variables. When we forward check B we find that its values 2 and 3 are both inconsistent with $A \leftarrow 1$. Forward checking would prune both of these values to level 5. However, closer examination shows that $B \leftarrow 3$ became inconsistent at level 1, four levels above the current level. We cannot make the assignment $B \leftarrow 3$ until we can make the assignment $A \leftarrow 3$, which we cannot do until we restore that value by backtracking to level 1: all of the supports $B \leftarrow 3$ has on the domain of A were pruned away at level 1.

The following routine computes the value at which it is safe to prune a future value `v` given that we have just instantiated `Var`. In this routine *compat*(`val`,`v`) is true if and only the two values `val` and `v` are compatible, i.e., they satisfy the constraint between `val.Var` and `v.Var` or there is no constraint between their variables.

[4] Note that we define soundness with respect to the set of unenumerated solutions. By doing this we obtain a uniform way of treating the case where the algorithm is searching for all solutions.

```
findDS(v,Var)
1. ds := 0;
2. foreach val ∈ Dom[Var] ∧ compat(val,v)
3.     ds := max(ds,val.prlevel);
4. return(ds);
```

In the above example, when we call `findDS` with `v` equal to 3 of B and `Var` equal to A, it will correctly return the level 1: only 3 of A is compatible with 3 of B, and this value was pruned at level 1 ($3.prlevel = 1$).

What about $B \leftarrow 2$? This value is also inconsistent with the assignment $A \leftarrow 1$, but unlike $B \leftarrow 3$ it is consistent with the as yet unpruned value 2 of A. Hence, when we undo $A \leftarrow 1$ we must also restore 2 of B so that it is available when we try $A \leftarrow 2$. That is, it is only sound to prune 2 of B to level 5, the current level. We can get `findDS` to compute a sound pruning level in this case also by simply pruning all of the other values in *CurDom*$[A]$ to the current level whenever we make an assignment to A. Clearly we cannot use these other values until we undo the current assignment to A, so it is sound to prune them to the current level. With this modification, `findDS` will find that A's compatible values for $B \leftarrow 2$, 2 and 3, have been pruned at levels 5 and 1 respectively, and thus that it is sound to prune 2 of B to level 5.

EFC—Extended Forward Checking Our first algorithm EFC is based on using `findDS` when doing pruning after an assignment. In particular, it uses the following instantiation:

```
Prune::Asgn(Var,val,level)
1. foreach v ∈ CurDom[Var] ∧ v != val
2.     prune(v,level);
3. foreach V ∈ FutureVars ∧ constrained(Var,V)
4.     foreach v ∈ CurDom[V];
5.         if(¬compat(val,v));
6.             prune(v,findDS(v,Var));
```

This routine is called whenever we assign variable `Var` the value `val` at the current level `level`. It operates just like FC except (1) it prunes all other values in *CurDom*`[Var]` to `level` and (2) instead of pruning inconsistent future values to `level` it prunes them to the perhaps higher level returned by `findDS`. Hence, when we backtrack back to `level` we can save work by not having to reconsider some of these future values until we backtrack to an even higher level.

We can apply the same insight when we backtrack. Extending our previous example, say that at level 5 in addition to *CurDom*$[A] = \{1, 2\}$ and $3.prlevel = 1$ we also have that 1 of B has been pruned by a previous assignment at level 2. Hence, *CurDom*$[B] = \{2, 3\}$ and value 1 of B has $1.prlevel = 2$. Now we make the assignment $A \leftarrow 1$ at level 5 as before. The remaining values of B are inconsistent with the new assignment, and thus $A \leftarrow 1$ will cause a wipeout of B. B will then be the next variable selected at level 6 (variables with empty domains must be selected first by `picknextVar`), and at that level the values of B will have $1.prlevel = 2$, $2.prlevel = 5$, and $3.prlevel = 1$ (with 3 having been pruned back to level 1 by **Prune::Asgn**). This will cause a backtrack to level 5 to try a different assignment to A. Say that search continues, eventually backtracking to level 4 and then descending again. On the new descent we might again explore the assignment $A \leftarrow 1$. However, it is obvious that this assignment cannot succeed, in fact it cannot succeed until we ascend to level 2 and there restore the value 1 of B: $A \leftarrow 1$ lost its last support on B at level 2.

Hence, at the moment we backtrack from level 6, we can detect that the assignment $A \leftarrow 1$ made at the level we are backtracking to (level 5) is in fact inconsistent back to level 2 and prune this value to that level. This yields the following version of

```
Prune::Backup(jbl,Var,level)
1. jbval := asgnVal[VarAtLevel[jbl]]
2. prlevel := min(findDS(jbval,Var),jbl-1);
3. prune(jbval,prlevel);
```

In this routine, `Var` is the variable causing the backtrack—the variable whose domain has been exhausted, `level` is the current level, and `jbl` is the level we are about to jumpback to. The routine prunes the value assigned at `jbl`, i.e., `jbval`: we find the deepest support `jbval`, has on the exhausted variable `Var`, and prune it back to that level. In our example, this routine will be called with `jbl` equal to 5, `Var` equal to B, `level` equal to 6, thus 1 of A (`jbval`), will be pruned back to level 2.

Note however, that we have just completed the search of the subtree below `jbval`, so we also know that we do not need to try `jbval` again until we undo at least the previous assignment made at level `jbl-1`. Hence, it is always sound to prune `jbval` to the previous level `jbl-1`. Hence, we can prune it to the minimum of `jbl-1` and its deepest support.

Finally, **FC+Prune** delays checking whether on not it has found a solution until it recurses to the next level. Thus when a solution is enumerated its last assignment was made at the previous level `level-1`. **Prune::FoundSoln** simply prunes the last value assigned *asgnVal*`[`*VarAtLevel*`[level-1]]` back to the previous level, `level-2`. This pruning is sound: no *unenumerated* solution can contain this value until at least one other assignment in the current solution is undone (i.e., the assignment at `level-2`):

```
Prune::FoundSoln(level)
1. val := asgnVal[VarAtLevel[level-1]];
2. prune(val,level-2);
```

It is not difficult to turn the discussion above into a proof that the domain pruning used by EFC is *sound*, and thus to show by Theorem 1 that EFC is sound and complete.

EFC- It is often not worth the extra computation to backprune forward checked values (line 6 of **Prune::Asgn**).[5] So we could restrict ourselves to the extra pruning done by **Prune::Backup**. Thus we can define a new algorithm EFC-, by substituting on line 8 of **Prune::Asgn** the call `prune(v,level)`, which simply prunes forward checked values to the current level exactly like FC does.

Observations and Some Empirical Results Both algorithms have the ability to prune values back beyond the current level, whereas FC only prunes values to the current level. This means that there will be cases where at level i we find that all of the values of the current variable have been pruned above level $i-1$. This will cause a backjump as we always backtrack to a level were we can restore at least one value of the exhausted variable. Furthermore, by pruning back the assignment we backtrack to, it is possible that we might backtrack to a variable and there discover that all of its values have been pruned back even further. This will generate another backjump. That is, multiple backjumps are possible. Both algorithms also have the ability to discover arc-inconsistent

[5] In particular, there are often a large number of values removed by forward checking, and computing the deepest support for all of them can be more costly than is worthwhile.

values and prune those values back to the level they became arc-inconsistent (including pruning values that were initially arc-inconsistent back to level 0 where they will never be restored).[6]

For binary CSPs it can be shown that if EFC is able to prune a value v back to level i, then MAC would have pruned v at level i (or less). This means that on binary CSPs, except for heuristic reasons, EFC/EFC- cannot offer a savings in the number of nodes explored over MAC. Hence the potential for savings in these algorithms over MAC on binary CSPs is limited. In our experiments with random binary CSP problems generated by the CT model we have found EFC/EFC- to be mostly inferior to MAC. EFC/EFC- may still have some potential on n-ary CSPs.[7]

Nevertheless, in the binary case both EFC and EFC- can be, like conflict directed backjumping (CBJ) [6], of considerable assistance to standard FC. Table 1 shows a typical example using 50 random binary CSPs. Each CSP is generated using the standard random CT model, and has 200 variables each having 10 possible values, and 200 constraints with 76 incompatible pairs.[8] In the experiment (and the experiment reported in the next section) we are searching for the first solution using a 500MHz Pentium III machine. We utilize dynamic variable ordering with the fail-first heuristic (minimum remaining values), and using the variable's current degree as a tie-breaker.[9]

MAC on this problem suite outperforms EFC/EFC-. It also performs better on many other parameter settings the CT model. But interestingly we have not found a case where there are orders of magnitude difference in performance, as can occur when we compare MAC with FC (or even FCCBJ). MAC's superiority on these problems is not surprising. Achlioptas et al. have shown that the random CT model generates problems that are highly biased in favor of MAC [8], especially as the number of constraints grows (irrespective of tightness as long as the number of incompatible pairs is greater than the domain size). With 200 constraints and a high tightness of 76% the problems being generated are either initially arc-inconsistent or become arc-inconsistent after only a few variables have been assigned (MAC visits only a average of 29.6 nodes on these problems). Furthermore these problems have a large number of arc-inconsistent values, so the fail-first heuristic works particularly well for MAC. Nevertheless, the results (and many other similar parameter settings of the CT model) do serve to demonstrate that there exist classes of problems for which EFC is a significant assist to FC.

One final point is that EFC- is identical to the algorithm FC-BM described by Prosser in [9]. In particular, Prosser identified that when backing up from a DWO the gains of **Prune::Backup** could be achieved. However, the algorithm he described uti-

[6] Prosser's FC-D2C algorithm [4] uses a special test to recognize the case when a value can be pruned back to level 0. This special case is achieved automatically in EFC/EFC-.

[7] It is not hard to define EFC for n-ary CSPs, and it should be feasible to extend all of the new n-ary versions of FC defined in [5] using these ideas. Further empirical evaluation is needed to determine how useful such extensions would be.

[8] The particular, in the random CT model the 200 constraints are chosen at random from the (200*199/2) possible binary constraints, and the 76 incompatible pairs are chosen at random from the 100 possible pairs.

[9] We also tried the current domain size divided by degree as a heuristic [7]. FC showed somewhat better performance with this heuristic but it still failed on many problems, otherwise the results were very similar with both EFC and EFC- still performing better than FCCBJ.

	Time in CPU sec.		Nodes Visited	
	Ave.	Max.	Ave.	Max.
MAC	0.012	0.030	29.64	275
EFC	0.033	0.190	1507	12584
EFC-	0.274	5.910	31306	66121
FCCBJ	5.105	194.680	404308	15819402
FC	FC is able to solve only some of these problems within a 250 sec. time bound			

Table 1. Performance on 50 <200,10,200,76> random binary CSPs.

lized backmarking style data structures as well as domain pruning data structures. The result was very complex as it was difficult to keep these two data structures synchronized. The relative simplicity of EFC- helps to demonstrate the advantages of our design.[10]

4.2 Conflict Directed Pruning

Our other new CSP algorithms are based on using conflicts or no-goods to detect inconsistent values. Conflicts have appeared many times before in CSP algorithms, e.g., in Conflict Directed Backjumping (CBJ) [6] and in no-good learning [11]. One difference here is that we maintain conflicts for values rather than variables.

A *conflict* C for the value v is a set of assignments $\{V_1 \leftarrow v_1, \ldots, V_k \leftarrow v_k\}$ such that no *unenumerated* solution contains both $v.Var \leftarrow v$ and C.[11] All of the conflicts manipulated by our algorithms will be subsets of *CurAsgns*, and thus they can be represented as a set of levels: the set of assignments made at those levels is the conflict proper. For a conflict CF, $\max(CF)$ will denote its maximum level.

The three algorithms we present are all based on a pruning mechanism that prunes the value v back to $\max(CF)$ once we detect that CF is a conflict for v. It is clear that such a pruning mechanism is *sound*: $V \leftarrow v$ cannot participate in any solution until the assignment at $\max(CF)$ is undone. Hence, all that we have to do to prove these algorithms sound and complete is to ensure that the sets they identify as being conflicts are in fact conflicts.

Say we have a value $v \in Dom[V]$, we can compute a conflict for v given conflicts for the values of another variable V'. Let $\{v'_1, \ldots, v'_k\}$ be the values of V' that are *compatible* with v, then it can be shown that the union of the conflicts for the v'_i is a conflict for v. The following algorithm computes a conflict for a value v given that we have conflicts for the values of the variable Var.[12]

```
computeCF(v,Var)
1. cf := {0};
2. foreach val ∈Dom[Var] ∧ compat(val,v)
3.    cf := cf ∪ val.CF;
4. return(cf);
```

[10] That backmarking (BM) type savings can be a achieved with a domain pruning mechanism is not so surprising given the close relationship between BM and FC demonstrated in [10].

[11] We are using the set of *unenumerated* solutions to define conflicts. This again allows us to provide a uniform treatment of the case when our algorithms are searching for all solutions.

[12] The initial 0 in the conflict facilitates pruning values back to level 0 by allowing $\max(CF)$ to take on the value 0. For example, if v has no compatible values on Var, the conflict $\{0\}$ will be computed. In this case we can permanent prune v by pruning it back to level 0.

CFFC—Conflict Based Forward Checking The algorithm CFFC uses the same structure as EFC. When we make a new assignment $A \leftarrow a$, we prune all other values in *CurDom*$[A]$, setting their conflict set to be the current level. Then we do forward checking. When we find an inconsistent future value, we compute a conflict for it by unioning all the conflicts of its compatible values on A, and prune that future value back to the maximum of its new conflict set. This yields **Prune::Asgn**

```
Prune::Asgn(Var,val,level)
1. foreach v ∈ CurDom[Var] ∧ v != val
2.     v.CF := {level};
3.     prune(v,max(v.CF));
4. foreach V ∈ FutureVars ∧ constrained(Var,V)
5.     foreach v ∈ CurDom[V];
6.       if(¬compat(val,v));
7.           v.CF := computeCF(v,Var);
8.           prune(v,max(v.CF));
```

Similarly, when we backtrack to level `jbl` because we have exhausted all the values of `Var` at level `level`, we can compute a conflict for `jbval`, the value assigned at the jumpback level. This conflict is the union of the conflicts of the values of the exhausted variable `Var` that are compatible with `jbvar`, except that in this case we can remove `jbl` from these conflicts. We can then prune `jbval` back to the maximum of its conflict.[13]

```
Prune::Backup(jbl,Var,level)
1. jbval := asgnVal[VarAtLevel[jbl]]
2. jbval.CF := computeCF(jbval,Var) - {jbl};
3. prune(jbval,max(jbval.CF));
```

Finally, when we find a solution and we wish to enumerate more, we have discovered that all of the previous levels are a conflict for the last assigned value (assigned at level `level-1`).

```
Prune::FoundSoln(level)
1. val := asgnVal[VarAtLevel[level-1]];
2. val.CF := {1,...,level-2};
3. prune(val,max(val.CF));
```

The basic algorithm CFFC has some similar features to conflict directed backjumping (CBJ) [6], dead-end driven learning [12], and the no-goods used in dynamic backtracking [13]. The conflicts used here are, however, more fine grained: they are value specific conflicts.

In addition to CFFC we can define CFFC-. In analogy with EFC-, CFFC- is identical to CFFC except that instead of computing a conflict for each value that is pruned by forward checking in **Prune::Asgn**, we simply set its conflict to be the single level of the current assignment (much like FC). Specifically, we replace line 7 of the routine with line `v.CF := {level}`. CFFC- is generally faster that CFFC as computing the conflict sets for all values pruned by forward checking can take more time than it saves. We have also found that on some problems the extra pruning performed by CFFC can seriously degrade backjumping [4].

[13] Since 0 might be the maximum of a conflict set, this process automatically achieves the special case permanent pruning of Prosser's CBJ-DkC algorithm [4].

Finally we can define our fifth algorithm CFMAC. In this routine instead of doing forward checking we enforce arc-consistency. As in the forward checking case, whenever we discover that a value v can be pruned because it has lost all of its support on some variable V' we set v's conflict to the union of the conflicts of its supports on V' (again using the routine `computeCF`) and prune it to the max of this conflict set. In standard MAC, v would only be pruned to the current level. (In CFMAC we prune on backup just as with CFFC).

Observations and Some Empirical Results CFFC is a very powerful CSP algorithm. In particular, its backjumping ability is more powerful than CBJ. CBJ maintains one conflict per variable. This conflict is simply the union of the conflicts maintained by CFFC over all the values of the variable.[14] The higher level of detail maintained by CFFC allows for larger backjumps.

For simplicity, assume that we are using CFFC- (i.e., no backpruning of forward checked values occurs). Say that we have two variables A and B both with domain $\{a, b, c\}$ and the constraint $A = B$. And that at level 7 we have that *CurDom*$[B] = \{c\}$, with conflicts for its other two values $a.CF = \{1\}$, and $b.CF = \{6\}$. Furthermore, say that *CurDom*$[A] = \{a\}$, with the conflicts for its other two values $b.CF = \{1\}$ and $c.CF = \{2\}$. If at level 7 we next attempt to assign A we have only value to try: $A \leftarrow a$. This causes a DWO of B.

Search will descend to level 8 where it will examine B. At this point we have for the values of B, $a.CF = \{1\}$, $b.CF = \{6\}$, and $c.CF = \{7\}$ (the last having been computed by forward checking from $A \leftarrow a$). These conflicts cause a backtrack to level 7 to try a different value for A. Standard CBJ will at this point union $\{1, 6\}$ into A's conflict set: i.e., all of the conflicts of B except level 7. Since there are no more values for A, the search will then backstep to level 6. CFFC, on the other hand, will set a value specific conflict for the value a of A: a of A will only inherit the conflict set associated with a of B due to the constraint between A and B. Thus, on backtrack to level 7, the individual value conflicts associated with A will be $a.CF = \{1\}$, $b.CF = \{1\}$ and $c.CF = \{2\}$, and the search will backjump all the way to level 2. This kind of behavior multiplies. On backtrack we can pass back a shorter conflict, and this in turn can pass back shorter conflicts and generate better backtracks at the higher levels.

Another feature of CFFC is its ability to detect and prune values that have become i-inverse inconsistent [14] for arbitrary i. Thus, CFFC has the potential to achieve exponential savings over an algorithm that continually enforces k-inverse consistency on the values of the future variables for any fixed k. However an algorithm that enforces k-inverse consistency does the work required to discover all k-inverse inconsistent values prior to continuing its search, whereas CFFC might not discover some of these inconsistencies until after it has performed a search exponential in j for some $j > k$. Thus, the potential for exponential savings exists the other way around as well. On the other hand, CFFC is "getting" on with the search while it discovering these k-inverse inconsistent

[14] Of course CFFC requires more storage, in the worst case $N^2 * D$ space to store all of the conflicts (where N is the number of variables and D is the maximum size of their domains). However, in practice, we can store conflicts as lists of elements, and we can reuse these elements when we backtrack and empty these sets (line 9 of `unassignlevels`). In our implementation we have never found space to be a critical issue.

	Size	Ave CPU time sec.	Ave. Nodes		Size	Ave CPU time sec.	Ave. Nodes
MAC	85	160.523	2241218	MAC	90	328.825	4824044
MACCBJ	85	44.130	263010	MACCBJ	90	66.305	352838
CFFC-	85	0.078	3318	CFFC-	90	0.067	2719
CFFC	85	4.702	137418	CFFC	90	4.343	120536
CFMAC	85	4.242	24546	CFMAC	90	5.818	28966

Table 2. Performance on 100 3-SAT instances with an embedded unsatifiable subproblem.

values.[15] CFMAC, takes the approach that it is useful to discover all 2-inconsistencies prior to continuing search by always enforcing arc-consistency. Conflict based pruning can then be used to take advantage of any higher order inconsistencies discovered during search.

CFFC and its variants do not perform very well on problems drawn from the random CT class. It is almost always outperformed by either MAC or on smaller problems by FC. Again this is to be expected from the results of Achlioptas et al. [8]. The problems generated by the random CT model are likely to contain many arc-inconsistent values (thus MAC has an advantage on these problems), and more telling, these problems are unlikely to contain any values that are k-inverse inconsistent for large k while being $k-1$-inverse consistent.[16] Thus the expense of using conflicts to detect k-inconsistent values for larger k is hardly ever beneficial: simple arc-consistency will immediately detect almost all inconsistent values. Furthermore, this also means that conflicts are unlikely to generate powerful backtracks. Thus, CBJ hardly helps (over MAC) on these problems either [17].

Nevertheless, there are problems on which CFFC is superior. One example, are problems that have a small group of variables participating in an unsolvable subproblem. On these kinds of problems the CFFC algorithms perform exponentially better than MAC and MACCBJ. This behavior arises from the fact that the CFFC algorithms are able to prune away the values of these inconsistent variables so that it does not need to keep on trying them, and is also able to generate more powerful backjumps than CBJ. Although a perfect heuristic could instantiate these "bad" variables at the top of the tree any heuristics can be foiled by the structure of the problem.

Table 2 shows one type of embedded problem. These problems originated in the work of Bayardo and Schrag and were used to illustrate their CBJ based satisfiability solver RelSat [18]. They took relatively easy random 3-sat problems and embedded in them a small unsatisfiable sat problem. We took the random SAT problems produced by their generator and converted them to binary CSPs so that we could test them with our (currently binary) algorithms. The results are shown in Table 2.[17] One surprising

[15] In an interesting recent paper, it has been shown that search is a generic way of implementing k-inverse consistency checking [15]. CFFC can be viewed as an algorithm that does both at the same time.

[16] This follows from Mitchell's recent results showing that these problems have short refutation resolutions [16].

[17] In more detail, the RelSat generator took easy random 3-SAT problems with N variables (85 and 90 in our experiment) and $3.5N$ clauses and added an unsatisfiable subproblem with 10 variables and 40 clauses. We took each 3-clause and converted it to a variable with domain

result is the performance of CFFC-, which explores significantly fewer nodes than the CFFC and CFMAC both of which do more pruning than CFFC-. In these problems the extra pruning is significantly degrading backtracking [4]. In the experiment we used the same hardware as before, and DVO with the fail-first heuristic and the variable's current degree as a tie-breaker. The MAC algorithm we used was based on AC3, and in our benchmarking it ran at about 65% of the speed of the more sophisticated AC7 based implementation of J.C. Regin [20][18]

We have also experimented with a preliminary implementation of n-ary versions of CFFC, CFFC-, and CFMAC algorithms. The implementation was built on top of van Beek's CPLAN system [21]. In these experiments we found that on harder logistics, blocks, and grid world planning problem CFMAC out performs the base GAC-CBJ (generalized arc consistency with conflict directed backtracking) implemented in CPLAN (CFMAC can run 2 to 5 times faster on some of the harder problems). Plain GAC is completely outclassed by these two algorithms—it can take more than two hours of CPU time on some problems that are solved in less than 10 seconds by CFMAC and GAC-CBJ. In comparision with CFFC- and CFFC, plain GAC is sometimes inferior sometimes superior. We also tried an n-ary version of the Golomb ruler problem (using quaternary constraints). In contrast with the planning problems on the Golomb ruler CFFC- is the fastest algorithm, being about 10 times faster than CFMAC and GAC-CBJ on the 10 marks ruler problem (length 55). We plan to report in more detail on these experiments in the near future.

In conclusion, the work presented here makes the following contributions: (1) it unifies a number of ideas that have appeared previous work, (2) it provides a clean design for a range of extensions to constraint propagating algorithms, and (3) it provides a clean way to implement and prove correct these new extensions. We have also presented some sample extensions that demonstrate the approach, and some evidence that these extensions might have potential for practical use. More work needs to be done to get a true picture of the empirical properties of these extensions.

References

1. R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
2. Fahiem Bacchus and Paul van Run. Dynamic variable reordering in CSPs. In *Principles and Practice of Constraint Programming (CP95)*, number 976 in LNCS, pages 258–275. Springer-Verlag, New York, 1995.

size 7. Each value for this clause/variable corresponds to a truth assignment to the 3 variables in the clause (there are 7 satisfying truth assignments for a 3-clause), and two clauses sharing variables became variables constrained so that they have compatible truth assignments. This is the dual construction [19]. Although this is not a particularly effective way of solving SAT problems, it does have the nice feature that it preserves the small unsatisfiable subproblem: the subproblem generates a set of variables in the CSP for which there is no consistent set of values.

[18] Thus using the AC7 version of MAC would not have altered our results. We did not use Regin's implementation because it only ran on SUN SPARCs, and our main computational resource was an Intel based PC. We also tried the current domain size divided by degree heuristic. In this case all of the algorithms performed slightly worse, but the relative performance was the same.

3. Pascal van Hentenryck. *Constraint Satisfaction for Logic Programming*. MIT Press, 1989.
4. Patrick Prosser. Domain filtering can degrade intelligent backtracking search. In *Procceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 262–267, 1993.
5. Christian Bessière, Pedro Meseguer, Eugene C. Freuder, and Javier Larrosa. On forward checking for non-binary constraint satisfaction. In *Principles and Practice of Constraint Programming (CP99)*, number 1713 in LNCS, pages 88–102. Springer-Verlag, New York, 1999.
6. P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3), 1993.
7. C. Bessiere and J.-C. Regin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *Principles and Practice of Constraint Programming (CP96)*, number 1118 in LNCS, pages 61–75. Springer-Verlag, New York, 1996.
8. L. M. Achlioptas, L. Kirousis, E. Kranakis, D. Krizanc, M. Molloy, and Y. Stamatiou. Random constraint satisfaction: A more accurate picture. In *Principles and Practice of Constraint Programming (CP97)*, number 1330 in LNCS, pages 107–120. Springer-Verlag, New York, 1997.
9. P. Prosser. Forward checking with backmarking. In M. Meyer, editor, *Constraint Processing*, LNCS 923, pages 185–204. Springer-Verlag, New York, 1995.
10. Fahiem Bacchus and Adam Grove. On the Forward Checking algorithm. In *Principles and Practice of Constraint Programming (CP95)*, number 976 in LNCS, pages 292–309. Springer-Verlag, New York, 1995.
11. R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.
12. Daniel Frost and Rina Dechter. Dead-end driven learning. In *Proceedings of the AAAI National Conference*, pages 294–300, 1994.
13. Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.
14. E. Freuder and C. D. Elfe. Neighborhood inverse consistency preprocessing. In *Proceedings of the AAAI National Conference*, pages 202–208, 1996.
15. Gérard Verfaillie, David Martinez, and Christian Bessière. A generic customizable framework for inverse local consistency. In *Proceedings of the AAAI National Conference*, pages 169–174, 1999.
16. David Mitchell. Some random csps are hard for resolution. http://http://www.cs.toronto.edu/m̃itchell/papers/some.ps, 2000.
17. S. A. Grant and B. M. Smith. The phase transition behaviour of maintaining arc consistency. Technical report, University of Leeds, School of Computer Studies, 1995. Technical Report 95:25, available at http://www.scs.leeds.ac.uk/bms/papers.html.
18. R. J. Jr. Bayardo and R. C. Schrag. Using csp look-back techniques to solve exceptionally hard sat instances. In *Principles and Practice of Constraint Programming (CP-96)*, pages 46–60, 1996.
19. Fahiem Bacchus and Peter van Beek. On the conversion between non-binary and binary constraint satisfaction problems. In *Proceedings of the AAAI National Conference*, pages 311–318, 1998.
20. J.-C. Regin. *Developpement d'outils alogorithmiques pour l'Intelligence Artificielle. Application a la chimie*. PhD thesis, Universite Montpellier II, France, 1995.
21. Peter van Beek and Xinguang Chen. A constraint programming approach to planning. In *Proceedings of the AAAI National Conference*, pages 585–590, 1999.