Looking Forward in Constraint Satisfaction Algorithms*

Fahiem Bacchus
Dept. of Computer Science
University of Waterloo
Waterloo, Ontario
Canada, N2L 3G1
fbacchus@logos.uwaterloo.ca
416-978-7390, 416-978-1455 (FAX)

Adam Grove
44 Murray Place
Princeton NJ 08540
grove@pobox.com

January 22, 1999

Abstract

Empirical evidence indicates that the most successful backtracking algorithms for solving constraint satisfaction problems do some level of forward constraint propagation. That is they check constraints against the as yet uninstantiated variables in the search tree. Two popular examples are the *forward checking* and *maintaining arc-consistency* algorithms. Although there is a growing body of empirical evidence supporting the superiority of these algorithms, relatively little is known about the theoretical relationship between these forward looking algorithms and backtracking algorithms that do not employ any constraint propagation, like generic backtracking and backmarking. In this paper we demonstrate that there is in fact a strong relationship between these different types of algorithms.

1 Introduction

Constraint satisfaction problems (CSPs) [Mac87] are typical of the NP-complete combinatorial problems that are so pervasive in AI. A number of techniques have been proposed in the literature for solving CSPs, but among the class of systematic algorithms (i.e., algorithms that guarantee finding a solution if one exists) backtracking algorithms remain the most popular technique.

In this paper we will restrict our attention to binary CSPs, where every constraint is between only two variables (see [BvB98] for more on the relationship between binary and general CSPs),

*Fahiem Bacchus's research was supported by the Canadian Government through their IRIS project and NSERC programs. Some of the results of this paper were previously presented at the First International Conference on Principles and Practice of Constraint Programming 1995 and appear in [BG95]. Adam Grove was at the NEC Research Institute, Princeton, New Jersey, U.S.A., during that time.

and we consider five basic algorithms, generic backtracking (BT), backjumping (BJ), backmarking (BM), forward checking (FC), and maintaining arc consistency (MAC).

Generic backtracking is the oldest of these algorithm, having been known for at least a century [BE75]. However, it is far from the best. Backjumping and backmarking are two orthogonal improvements to BT [Gas78, Gas77]. The first, BJ, avoids visiting some nodes that cannot possibly lead to a solution, while the second, BM, avoids making some redundant consistency checks at the nodes it visits. Both of these algorithms can never do worse than BT [KvB95], and generally do much better. These three algorithms BT, BJ, and BM do all of their consistency checking backwards. That is, they wait until they instantiate a new variable before checking to see if that instantiation is consistent with the previous instantiations.

The forward checking [HE80] and maintaining arc consistency algorithms [Gas79] (more recently improved and popularized in [SF94, BR96, Reg95]) operate quite differently. They do all of their consistency checking forwards, checking every new instantiation against the future (as yet uninstantiated) variables. FC and MAC differ in that MAC performs more forward constraint checking than FC. Both of these algorithms are able to detect domain wipeouts (DWO) of future variables. DWO detection is useful as there can be no solution below a node where a future variable has a DWO. These algorithms can detect thus detect deadends higher up in the search tree and avoid searching nodes that BT must search.

In a recent paper Kondrak and van Beek [KvB95] presented a insightful analysis of a number of backtracking algorithms (including four of the five we consider here). They were able to demonstrate a number of dominance results between these algorithms in terms of the number of nodes visited and the number of constraint checks performed. However, although they were able to demonstrate that FC visits fewer nodes than BT, BJ or BM, they were not able to provide any results concerning the number of constraint checks performed.

This is an important gap to fill, as the number of constraint checks is really the most important metric, particularly for forward looking algorithms like FC and MAC. Most backtracking algorithms have complexity that is Θ(number of constraint checks) (although there can be non-trivial differences in the constant factors involved).[1] However, for algorithms that look forward there is no simple relationship between the number of nodes visited and the efficiency of the algorithm. In particular, for these algorithms there is a tradeoff between the amount of forward constraint checking they do at each node and the number of nodes they visit. For example, MAC may do more total constraint checks than FC on a particular problem even though it is guaranteed to visit no more nodes than FC.

Because of the different way BT, BJ and BM organize their computation, and the ability of FC and MAC to detect a domain wipeout higher up in the search tree, a formal comparison seems to be hard to provide. In particular, there are easily constructed examples where FC performs

[1] It can be difficult to measure constraint checks for some implementations of MAC. Nevertheless, it is possible to view the data structures built by some versions of MAC as being cached constraint checks. Hence, with MAC implementations based on algorithms like AC-4 [MH86], AC-6 [Bes94] or AC-7 [BFR95] we can count an access to these data structures as being a constraint check. Under this interpretation, these algorithms still have complexity O(number of constraint checks).

more consistency checks than even the worst algorithm, generic BT. Nevertheless, in this paper we demonstrate that there is a precise way of separating the computation performed by FC and MAC into two distinct components. It turns out that in the case of FC one of these components is identical to BM, while the other component can be precisely characterized as domain wipeout detection. Thus we will see that there is in fact a close theoretical relationship between these five CSP algorithms.

Furthermore, once we understand these two computational components of MAC and FC, we can use that to our advantage by finding more efficient ways of implementing each component, and by utilizing each component to its maximum. We will return to this point in Section 4.

This paper is organized as follows. In Section 2 we briefly present some of the necessary background. Then, in Section 3 we demonstrate the close relationship between FC and BM and show that this relationship can be extended to understand MAC as well. We also briefly discuss the orthogonal improvements that can be obtained from BJ style extensions. We close with some conclusions.

2 Preliminaries

A *binary* CSP consists of a finite set of variables, each with a finite domain of potential values, and constraints between pairs of variables. The goal is to assign a value to each variable so that all of the constraints are satisfied. Depending on the application we may want to find all consistent assignments, or to find just one.

Definition 2.1 A binary constraint satisfaction problem, $\mathcal{P}$, consists of:

- A finite collection of N *variables*, $V_1, \ldots, V_N$.
- For each variable V_i, a finite domain of k_i *values*, $D_i = \left\{v_1^i, v_2^i, \ldots, v_{k_i}^i\right\}$.
- For some set of pairs of variables $\{V_i, V_j\}$, a *constraint* $C_{\{i,j\}}$ between V_i and V_j which is simply a subset of $D_i \times D_j$. If $(v_l^i, v_m^j) \in C_{\{i,j\}}$ we say that the assignment $\{V_i \leftarrow v_l^i, V_j \leftarrow v_m^j\}$ is *consistent*. If there is no constraint between V_i and V_j then all assignments $\{V_i \leftarrow v_l^i, V_j \leftarrow v_m^j\}$ are *consistent*.

A *solution* to $\mathcal{P}$ is an assignment $\left\{V_1 \leftarrow v_{s_1}^1, \ldots, V_i \leftarrow v_{s_i}^i, \ldots, V_N \leftarrow v_{s_N}^N\right\}$ such that for all i, j, $\left\{V_i \leftarrow v_{s_i}^i, V_j \leftarrow v_{s_j}^j\right\}$ is consistent.

A number of algorithms have been developed for solving CSPs, most of which are based on backtracking search in a tree of variable assignments (each node in the tree is an assignment of a value to a variable). See Tsang [Tsa93] for a good description of many of these algorithms.

Backtracking algorithms examine partial solutions which are assignments to a subset of the variables, and try to extend these until all variables are assigned. BT is the simplest such algorithm. At every node of the search tree it assigns a value to an as yet uninstantiated variable, and then

checks the consistency of that new assignment with all of the past assignments on the path from the root of the tree to the current node. If any of these consistency checks fail, it tries to assign a different value to the variable. When no more values remain to be tried, it backtracks to the node's parent and tries to assign a different value to the variable there. BM is a variant of BT that saves a number of redundant consistency checks by keeping track of information about past consistency checks. It is explained in more detail below. BJ is another variant of BT that like BM keeps track of information about past consistency checks. With this information it is able to improve over BT in a manner that is orthogonal to BM, by skipping some parts of the tree that cannot contain any solutions.

FC and MAC, on the other hand order their consistency checks in a different way than BT, BJ, and BM. When a new instantiation is made they check that instantiation against the possible values of the future (as yet uninstantiated) variables, pruning all values inconsistent with the new instantiation. If a future variable has all of its values pruned a *domain wipeout* occurs (DWO), and the algorithms backtrack. Since all the consistency checks have been done in advance, these algorithms do not need to check the new instantiation against previous instantiations—if the new instantiation was inconsistent that value would have already been pruned.

FC does the "minimal" amount of forward constraint checking. In particular, it does just enough to insure that any new instantiation is consistent with the past variables. MAC on the other hand does additional forward constraint checking.

2.1 Forward Checking

The forward checking algorithm [HE80] constructs solutions by considering assignments to variables in a particular order, which for concreteness we take to be $V_1, V_2, V_3, \ldots, V_N$.[2] Suppose that we have found a consistent assignment to the first $i-1$ variables, which means that all constraints involving only these $i-1$ variables are satisfied. At this point, we call $V_1, \ldots, V_{i-1}$ the *past* variables, V_i the *current variable*, and the others the *future* variables. The characteristic data structure of the FC algorithm is a two dimensional array `Domain`. The idea is that $\texttt{Domain}^j_m$ will contain 0 if and only if the assignment $V_j \leftarrow v^j_m$ is consistent with the assignments chosen for all the past variables. Otherwise, it contains the index of the first assigned variable (i.e., the shallowest node in the search tree) with which $V_j \leftarrow v^j_m$ is inconsistent.

It follows that, when we are considering a possible value v^i_l for the current variable V_i, it is sufficient to look for a zero in $\texttt{Domain}^i_l$. Any such value is guaranteed to be consistent with all past choices; we do not need to do the backwards consistency checks that are characteristic of BT and its variants. The price, of course, is that when we make a successful assignment to the current variable, we must check it against all outstanding values of the future variables, updating there `Domain` arrays as necessary. Figure 1 gives the important parts of the algorithm in more detail; after initialization, the call FC(1) will print all solutions. Note that the current partial assignment

[2]For simplicity we assume a static ordering. Nevertheless, our results can be extended to dynamic orderings in a straightforward way, see Section 3.6.

```
procedure FC(i)
%Tries to instantiate V_i, then recurses
    for each v_l^i ∈ D_i
        s_i ← v_l^i
*       if Domain_l^i = 0
            if i = N
                print s_1, ..., s_N
            else
                if Check-Forward(i)
                    FC(i+1)
                Restore(i)

procedure Restore(i)
%Returns Domain to previous state
    for j = i + 1 to N
        if C_{i,j} exists
            for each v_m^j ∈ D_j
                if Domain_m^j = i
                    Domain_m^j ← 0

function Check-Forward(i)
%Checks s_i against future variables
    for j = i + 1 to N
        if C_{i,j} exists
            dwo = true
            for each v_m^j ∈ D_j
                if Domain_m^j = 0
                    if (s_i, v_m^j) ∈ C_{i,j}
                        dwo = false
                    else
                        Domain_m^j ← i
            if dwo
            return(false)
    return(true)
```

Figure 1: Pseudo-code for Forward Checking

is remembered in program variables $s_1, \ldots, s_N$. An assignment to s_i fails if there is a "domain wipe-out" (DWO), which means that we have discovered that every value of some future variable is inconsistent with our choices so far. DWO means that no solution can exist in the subtree below this assignment. Note also that, after we finish considering a choice for s_i, we must undo any changes made to the `Domain` array before continuing.[3]

It may seem as if FC can end up doing many redundant checks, as it checks against future variables that may never be visited. For example, checking V_1 against V_N is wasted work if the assignment to V_1 ends up forcing V_2 and V_3 to be completely inconsistent with each other. But balanced against this is the chance that it can detect domain wipe-outs and avoid parts of the search tree explored by the backward checking algorithms.

There is strong empirical evidence [HE80, Nad89, Pro93, vR94], that the work expended by FC to perform DWO detection results in a net gain.[4] Another major advantage of FC is that the

[3]We have designed this code for clarity; there are many alternatives that are more efficient. The Restore procedure could be improved, and a common presentation of the FC algorithm [HE80] uses a loop over V_i's current domain, i.e., the elements of $\texttt{Domain}^i$ that are zero, instead of over D_i. (The latter "improvement" may or may not be more efficient in practice as it requires maintaining a data structure containing the current domain.)

[4]In fact, there is a growing body of evidence that it is profitable to expend even more work. An number of recent studies have indicated that MAC (which does much more work than FC) performs better than FC on very hard problems [BR96, Reg95, Liu98, GS95].

number of remaining consistent values for each of the future variables can be computed without any additional constraint checks—this is simply the number of entries in $\texttt{Domain}^i$ that are zero. This means that the highly effective minimal remaining values (MRV) heuristic, in which we instantiate next that variable with fewest remaining values (also known as the fail-first (FF) heuristic), can be used "for free" to perform dynamic variable ordering [BvR95].

2.2 MAC

The MAC (maintain arc consisency) algorithm [Gas79, SF94] works in essentially the same manner as FC. It considers the variables in sequence and chooses a value for each variable just like FC. Once a variable is instantiated we reduce the domains of the future variables to ensure that they are all consistent with this new instantiation. MAC goes beyond FC in that we then also check the future variables against each other removing any arc-inconsistent values from their domains. Thus a future variable may have values removed either because that value is inconsistent with a past variable, or because it has no support in the domain of a future variable. Figure 2 gives a more detailed presentation of MAC-3,[5] i.e., MAC that uses the AC-3 algorithm [Mac77].[6]

2.3 Backmarking

Gashnig's backmarking algorithm [Gas77] is an improvement over generic backtracking. In explores exactly the same search tree as BT, doing all of its checks backwards just like BT. However, it offers a considerable improvement over backtracking by eliminating many redundant consistency checks. Recall that when an assignment $V_i \leftarrow v_l^i$ of the current variable V_i is made, BT checks the consistency of this assignment against all of the previous assignments $\left\{V_1 \leftarrow v_{s_1}^1, \ldots, V_{i-1} \leftarrow v_{s_{i-1}}^{i-1}\right\}$. If any of these consistency checks fail, BM takes the additional step of remembering the first point of failure in an array $\texttt{Mcl}_l^i$ ("maximum check level"). This information is used to save later consistency checks. Say that later we backtrack from V_i up to the variable V_j, assign V_j a new value, and then progress down the tree, once again reaching V_i. At V_i we might again attempt the assignment $V_i \leftarrow v_l^i$. The assignments $\left\{V_1 \leftarrow v_{s_1}^1, \ldots, V_{j-1} \leftarrow v_{s_{j-1}}^{j-1}\right\}$ have not changed since we last tried $V_i \leftarrow v_l^i$, so there is no point in repeating these checks. Furthermore, if $V_i \leftarrow v_l^i$ had failed against one of these assignments, we need not make any checks at all; the assignment will fail again, so we can immediately reject it. To realize these savings, BM uses an additional array $\texttt{Mbl}$ ("minimum backtrack level") which for each variable keeps track of how far we have backtracked since trying to instantiate this variable. (In our example $\texttt{Mbl}^i$ will store the information that we have only backtracked to V_j since last visiting V_i.) Figure 3 gives the backmarking algorithm in more detail.

In empirical tests BM is a considerable improvement over BT [vR94, Nad89].

[5] Again this code is designed for clarity not efficiency. Many optimizations are possible, including specialized treatment of single valued variables (e.g., the variable instantiated at the current level). See Liu [Liu98] for a good discussion of MAC implementation issues, and Sabin and Freuder [SF97] for additional ways of improving MAC.

[6] There are many different choices of arc-consistency algorithm that could be used by MAC, e.g., Sabin and Freuder [SF94] used the AC-4 algorithm [MH86] while Regine [Reg95] used the AC-7 algorithm [BFR95].

```
procedure MAC(i)
%Identical to FC(i), (Figure 1)
%except we call Check-Forward-AC(i)
%rather than Check-Forward(i).
%Restore(i) remains identical.

procedure Revise(j,k,i)
%Remove all domain elements of V_j
%that lack support on V_k. Revision is being
%performed at level i in the search tree.
    updated = false; dwo = true
    for each v_l^j ∈ D_j
        if Domain_l^j = 0
            notfound = true
            for each v_m^k ∈ D_k while notfound
                if Domain_m^k = 0 and (v_l^j, v_m^k) ∈ C_{j,k}
                    notfound = false
            if notfound
                Domain_l^j ← i; updated ← true
            else
                dwo ← false

function Check-Forward-AC(i)
%Checks s_i against future variables
%and maintains arc-consistency
    for j = i + 1 to N
        if C_{i,j} exists
            enqueue((j,i),RevQueue)

    while notempty(RevQueue)
        (j,k) ← dequeue(RevQueue)
        Revise(j,k,i)
        if dwo
            return(false)
        if updated
            for a = i + 1 to N
                if a ≠ k and C_{a,j} exists
                    enqueue((a,j),RevQueue)
return(true)
```

Figure 2: Pseudo-code for MAC

3 Formal Relationships

3.1 Bounded Inferior Performance

On some problems (e.g., N-Queens) BM can actually perform as well as FC, however the differences can be enormous (particularly on harder problems). It is easy to see that FC can do much better. For example, if the first and last variables are incompatible with each other FC will realize this almost immediately, whereas BM might search the entire search tree—which can be exponentially large—before declaring failure. Kondrak and vanBeek [KvB95] have shown that FC always explores a subset (not necessarily proper) of the nodes that BM, BT, or backjumping (BJ) visit. Nevertheless, since FC can perform more checks per node, FC may in fact perform more consistency checks.

Example 1 Suppose V_2 and V_3 are mutually inconsistent. The backward checking algorithms can discover this quickly, searching only three variables deep (thus making at most $k_1k_2 + k_1k_3 + k_2k_3$ consistency checks). Forward checking can take much longer. For each assignment to V_1, it checks against all subsequent variables, so that it does as many as $k_1 \sum_{i=4}^{N} k_i$ additional checks over the

```
procedure BM(i)
%Tries to instantiate V_i, then recurses
    for each v_l^i ∈ D_i
        s_i ← v_l^i
        if Mcl_l^i ≥ Mbl^i
            ok ← true
            for j = Mbl^i to i − 1 while ok
                Mcl_l^i ← j
                if (s_j, s_i) ∉ C_{i,j}
                    ok ← false
*           if ok
                if i = N
                    print s_1, ..., s_N
                else
                    BM(i+1)
    Mbl^i ← i−1
    Restore(i)

procedure Restore(i)
%Updates Mbl
    for j = i + 1 to N
        if Mbl^j = i
            Mbl^j ← i−1
```

Figure 3: Backmarking

backward checking algorithms. These extra checks do not reveal the inconsistency between V_2 and V_3 and hence are wasted work. ■

The problem is, of course, that FC delves deeply into the search tree to find DWOs, and this does not always pay off. But the cost is never exponential in the size of the problem. The following simple corollary of Kondrak and van Beek's result is worth making explicit.

Remark 3.1 Let K be the largest domain size. FC never performs more than NK times as many consistency checks as BT, BJ or BM (where N is the number of variables), but the performance loss can be arbitrarily close to this bound. On the other hand, there are families of problems in which BT, BJ and BM all perform $e^{\Omega(N \ln K)}$ more checks than FC.

Proof: Reasoning as in Example 1, we see that forward checking from a node costs at most NK checks. The example can be arranged (by choosing N large enough, $k_2 = k_3 = 1$, and $k_i = K$ for $i > 3$) so that the actual number of checks divided by NK is arbitrarily close to one. Our first claim now follows from Kondrak's result showing that FC explores no more nodes than BT, BM or BJ. There is a slight subtlety in the case of BM, as at some nodes BM does not perform any consistency checks. However, these nodes correspond to inconsistent assignments to the current variable so FC never visits these nodes.

For our second claim, simply consider a CSP problem in which BM, BJ, and BT do an exponential amount of work without finding a solution. Modify the problem by adding a new variable V_{N+1} that is incompatible with all assignments to V_1. In the new tree FC will detect that no solution

exists in no more than NK^2 checks, while BT, BM and BJ will still require an exponential amount of work. ■

It is easy to extend this observation to MAC.

Remark 3.2 Let MAC be implemented with an arc-consistency algorithm that is order $O(N^iK^j)$, where N is the number of variables and K is the largest domain size. Then MAC never performs more than N^jK^i times as many consistency checks as BT, BJ or BM. On the other hand, as above, BT, BJ and BM can perform exponentially more checks than MAC.

Proof: The previous proof applies *mutatis mutandis* . ■

For example, if MAC uses AC-4 we would have N^2K^2 (there is a $O(N^2)$ bound on the number of edges in the constraint graph) as a bound on how much more work MAC could do over BT, BJ or BM.

It is obvious that the important feature of FC and MAC is that they can use DWO detection to prune large parts of the search space, and in doing so save themselves a considerable amount of work over BT (as well as over BJ and BM). However, it turns out that FC improves over BT in another way as well. In particular, FC avoids many redundant checks in a manner that is *exactly* the same as BM.[7]

Taking account of the similarity between FC and BM allows us to separate the computation FC performs into two distinct components, a component that is identical to BM and the remaining component, which turns out to be exactly domain wipeout detection.

3.2 FC and BM

BM is often characterized as obtaining two types of savings over BT.

A. When $\texttt{Mcl}^i_\ell < \texttt{Mbl}^i$ we have previously checked the assignment $V_i \leftarrow \ell$ only down to a level $\texttt{Mcl}^i_\ell$ that is shallower than the level where we have changed the past assignments, $\texttt{Mbl}^i$. Thus this assignment is in conflict with past assignments that have not yet been changed, and we need not consider this assignment again.

B. When $\texttt{Mcl}^i_\ell \geq \texttt{Mbl}^i$ we have previously checked the assignment $V_i \leftarrow \ell$ down to a level $\texttt{Mcl}^i_\ell$ that was deeper than the level where we have changed the past assignments, $\texttt{Mbl}^i$. Thus we need only check this assignment against the new assignments, from $\texttt{Mbl}^i$ down to the current level.

An important fact is that FC includes both of these types of savings. Intuitively, the reason that FC includes type B savings is that FC does its constraint checking incrementally. Say we previously checked the assignment $V_i \leftarrow \ell$ down to a level h_1 (the value of $\texttt{Mcl}^i_\ell$), then we backed up to level

[7] We first reported this connection in [BG95] but did not provide a proof in that paper.

h_0 (the value of $\texttt{Mbl}^i$) and then came back down to the assignment $V_i \leftarrow \ell$. As we come back down FC would only check $V_i \leftarrow \ell$ against the new assignments we make as we move down from h_0 to the current level (i.e., from $\texttt{Mbl}^i$ down to the current level just as in BM).

Type A savings are similar. Say we previously checked the assignment $V_i \leftarrow \ell$ only down to a level h_1 (the value of $\texttt{Mcl}^i_\ell$), then we backed up to level h_0 (the value of $\texttt{Mbl}^i$), with $h_1 < h_0$ ($\texttt{Mcl}^i_\ell < \texttt{Mbl}^i$). FC would have pruned ℓ from the domain of V_i at level h_1 and ℓ would not be restored by backing up only to level h_0. Thus as we come back down to the level where V_i would be assigned we would never make any checks against the assignment $V_i \leftarrow \ell$, as ℓ is not in V_i's domain during this descent.

To make these intuitions precise requires a considerably deeper analysis. However, our analysis is also able to show that the converse also holds: although FC incorporates BM savings it does not incorporate any additional savings beyond BM. That is, in the sense made precise below, FC is the BM algorithm with an added component that turns out to be exactly DWO detection. To formalize our argument we first define the concept of a tree-check.

Definition 3.3 A particular consistency check $(s_i, v^j_\ell) \in C_{\{i,j\}}$ performed during the execution of the FC algorithm (Figure 1) is called a *tree-check* if the algorithm later attempts the partial assignment $s_j \leftarrow v^j_\ell$,[8] while $s_1, \ldots, s_i$ remain the same as at the time the check was made. (That is, $s_j \leftarrow v^j_\ell$ is later attempted in the subtree below the node where the check was make).

Example 2 Consider the example shown in Figure 4. The diagram shows a backtracking tree explored by FC. In the CSP there are four variables each with the two element domain $\{a, b\}$. At the top level of the tree the variable V_1 is instantiated with the value a. Then the domains of the future variables are checked against this instantiation. The checks performed at this stage are shown in the box below the assignment statement. Six constraints are checked: all possible values of the future variables against the assignment $V_1 \leftarrow a$. The search tree shows that a node that makes an assignment to V_4 is never visited by FC. Hence, the checks against $V_4 = a$ and $V_4 = b$ are non-tree checks. The other four checks are, on the other hand, all tree checks, as is indicated by the label "T" that follows them. In all cases a node that "attempts" that instantiation is later visited by the search process.

It should be noted that the node $V_3 = b$ is in a sense never visited by the search process, since this value for V_3 has been pruned prior to arriving at this node. Nevertheless, we consider FC to have visited that node as soon as it executes the assignment $s_3 \leftarrow v^3_l$ (line 4 in FC procedure) even if the subsequent line of code discovers that the value v^3_l has already been pruned. We show this node as being connected via a dotted line, and we count the checks against $V_3 = b$ above this node to be tree checks. Even though we are counting these phantom nodes[9] we are only counting *real*

[8] We consider an assignment to be attempted as soon as FC executes the code $s_i \leftarrow v^i_l$ (line 4 in FC procedure), even if the subsequent code immediately discovers that this value has already been pruned from the current domain of V_i.

[9] In contrast Kondrak and van Beek [KvB95] do not count such nodes in their analysis of the nodes visited by FC.

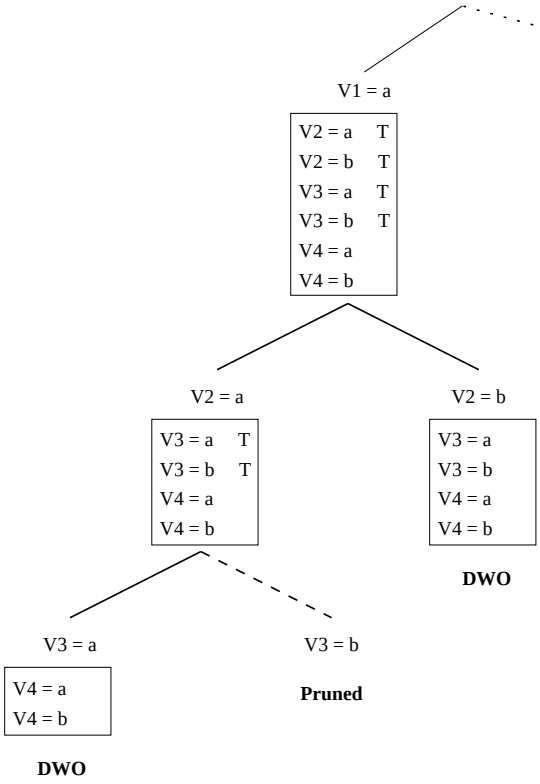

Figure 4: Tree Checks: only the checks labeled with "T" are tree checks.

consistency checks. That is, the set of consistency checks we consider are exactly those performed by any implementation of the FC algorithm. Counting these phantom nodes does not change the shape of the tree searched by FC. The main purpose in including such nodes is to allow us to more easily characterize tree-checks.

Many of the checks are non-tree checks. For example, all the checks performed at a node where DWO is detected are non-tree checks. More generally, all checks against a variable that is never visited in the subtree below are non-tree checks. ■

The intuition behind tree-checks and non-tree-checks is that non-tree-checks have the *sole* purpose of looking for DWO's, and if no DWO is found they are, in a sense, wasted. A tree-check may help in finding a DWO, but it is also used at least once in evaluating the correctness of a proposed instantiation to the current variable (in a sense, it directly helps in "building" the search tree). This distinction is an natural one, although it can only be made after the fact.[10]

[10]That is, we do not know whether or not a particular consistency check is a tree-check until somewhat later in the search process. However, should one wish to, it is quite easy to code the FC algorithm so that it keeps an accurate count of the tree-checks.

The concept of tree-checks is not the only idea we need. The other issue that makes a direct comparison between BM and FC impossible is simply that BM explores a different set of nodes: it does not detect DWO. To overcome this, we must imagine that BM is supplied with an DWOFC "oracle" that, given any partial assignment, can tell whether FC would detect a DWO at some future variable given the current set of assigned variables. We can imagine line $(*)$ in Figure 3 being replaced by:

$*$ **if** ok **and** not(DWOFC(i))

where DWOFC(i) is a call to an oracle testing for DWO (with the same power as FC constraint propagation) at some variable in the future of V_i. This change now makes the comparison between BM and FC fair. The surprising result is that, after accounting for this change, FC and BM are essentially identical algorithms. In particular, we have the following theorem.

Theorem 3.4 *BM, supplied with an oracle as described above, explores exactly the same nodes as FC. Furthermore, the number of consistency checks it makes is the same as the number of tree-checks FC makes.*

Proof: It is easy to see that an oracle supplied BM and FC explore the same nodes. FC will visit a node (including the phantom nodes mentioned in Example 2) if and only if (1) all constraints between the past variables (i.e., the already instantiated variables) are satisfied, and (2) no DWO was detected after any of these past variables were instantiated. Similarly BM will visit a node if and only if (1) all the constraints between the past variables are satisfied, and (2) since we call DWOFC after every instantiation, no DWO was detected after any of these past variables were instantiated.

To show the correspondence in the number of checks we construct a one-to-one onto mapping between the set of FC tree-checks and the set of BM (with an DWOFC oracle) checks. Suppose that at node n_1 BM checks the assignment $A_i = V_i \leftarrow \ell_i$ against the past assignment $A_j = V_j \leftarrow \ell_j$ made at node n_0. n_0 must lie on the path from the root of the tree down to n_1. Since FC visits the same nodes it also must visit n_0 and then descend to n_1. We map the BM check of A_i against A_j performed at n_1 to the FC tree-check of A_j against A_i performed at n_0.

(1) this mapping is well defined. That is, the FC tree-check we map to exists. Since BM checks A_i against the past assignments from the first assignment downwards (some of these prior checks could have been made at other nodes where we had previously attempted the instantiation A_i), we know that if it checked A_i against A_j, A_i must have been consistent with all assignments made before A_j. Thus, when FC visits n_0 we must have that ℓ_i is a member of the domain of V_i (i.e., it was not pruned by any prior assignment) and that FC must check it against the current assignment A_j. Furthermore, since FC later visits n_1 where we attempt the assignment A_i, its check of A_j against A_i is a tree-check.

(2) this mapping is one-to-one. Say that the check A_i against A_j is made by BM at two different nodes n_1 and n_2. Furthermore, say that both of these checks map to the same check of A_j against A_i made at node n_0 by FC. By the way in which the mapping is defined only checks made at nodes

in the subtree under n_0 can map to the FC tree-check made at n_0. Without loss of generality, let n_1 be the leftmost node and let n_2 be the second leftmost node in the subtree below n_0 where BM checks A_i against A_j. Thus when BM visits n_2 and checks the assignment A_i the previous time it checked this assignment would be at n_1. But then, given that n_0 is at level j we have that $\mathtt{Mbl}^i > j$ at the time BM visits n_2 (since n_1 and n_2 are in the subtree below n_0 BM cannot have backtracked to the level of n_0 inbetween). Hence, at n_2 BM would not have checked A_i against A_j as it does not check any assignments made above $\mathtt{Mbl}^i$. This contradicts the existence of two BM checks mapping to the same FC check.

(3) this mapping is onto. Say that FC makes a tree-check of A_j against A_i at node n_0. We claim that there is a BM check made at a node n_1 below n_0 that maps to this FC tree-check. Since it is a tree check we know that FC visits a node where we attempt the assignment A_i. Let n_1 be the leftmost node in the subtree below n_0 at which FC attempts the assignment A_i. BM also visits n_1, and we claim that at n_1 it performs a check of A_i against A_j. That is, it performs a check that maps to the FC tree-check in question. There are only two possible reasons why BM would not perform the check A_i against A_j at n_1. First, it could be that BM avoids this check because of condition A. Since n_1 is the leftmost node under n_0 where the assignment A_i is tested BM must have backed up above n_0 since last testing A_i (BM tests all assignments to a variable when it tests a variable). That is, if n_0 is at level j we must have $\mathtt{Mbl}^i < j$. But condition A requires that $\mathtt{Mcl}^i_{\ell_i} < \mathtt{Mbl}^i$. Hence, the assignment A_i must have failed at a level $< j$, i.e., above n_0. So when FC reaches n_0 the assignment A_i is inconsistent with an assignment made above n_0. This cannot be the case as then FC would not have made the check against A_i either—ℓ_i would be pruned from the domain of V_i prior to reaching n_0. If condition A does not hold condition B, must hold (this includes the case where we have never tested the assignment A_i before). As before since n_1 is the leftmost node under n_0 testing A_i we must have that $\mathtt{Mbl}^i < j$ the level of n_0. But then under condition B A_i would be tested against all assignments from $\mathtt{Mbl}^i$ down to the current level. None of the assignments that lie above n_0 can generate a failure—if they did FC would have pruned ℓ_i from the domain of V_i and would not have made any checks against A_i at n_0. Thus BM will reach and then perform the check of A_i against A_j as claimed. ■

That BM, when equipped with a DWO oracle should visit exactly the same nodes as FC is not surprising. However, that BM does exactly the same number of checks as the number of tree-checks performed by FC is. It must be remembered that BM utilizes some subtle bookkeeping in order to eliminate many redundant checks over BT. FC, on the other hand, simply prunes its future domains, a process that is, on the surface, quite distinct from BM's bookkeeping. Our theorem shows that these seemingly distinct processes actually coincide.

3.3 MAC and BM

As mentioned above the MAC algorithm invests more work than FC in detecting domain wipeouts. In particular, as in FC some values of the future variables might be pruned because they are inconsistent with the past variables, but MAC might also prune some of these values because they are

arc-inconsistent with the domains of the as yet uninstantiated variables. Hence, MAC can detect domain wipeouts earlier than FC—MAC's domain wipeout "oracle" is more powerful than FC's. An immediate consequence of this is that MAC visits a subset of the nodes visited by FC. In this section we restrict our attention to the MAC-3 algorithm, which uses AC-3 as its arc-consisency algorithm (this is the algorithm sketched in Figure 2). AC-3 explicitly uses consistency checks, and thus we can compare these checks against the checks performed by BM and FC.[11]

The following theorem shows that BM equipped with such an oracle will visit exactly the same nodes as MAC. However, it may perform more consistency checks than the tree-checks that MAC performs. This means that wipeout detection still "determines" the MAC's search tree, but that MAC is able to build its search tree using fewer consistency checks than BM.

As with FC we can separate the checks MAC performs into tree-checks and non-tree-checks. Our definition of a tree-check is unchanged from Definition 3.3.[12] It should be noted that in this definition only checks against already assigned variables are candidates for being counted as tree-checks[13]. In FC this includes all checks, but in MAC there will be many checks that are not of that category. No check of a future variable against a future variable can be counted as a tree-check.

The reason why MAC is able to save itself some tree-checks over an oracle equipped BM is that some domain values will have been pruned by arc-consistency processing. For example, suppose that the assignment made at level k causes the value v_ℓ^j of variable V_j to be pruned by arc-consistency processing. That is, $V_j \leftarrow v_\ell^j$ is consistent with all of the instantiated variables (so it would not be pruned by FC), however it is pruned because it lost all support on the domain of another uninstantiated variable. Furthermore, suppose that MAC subsequently descends to a level where V_j is instantiated without undoing the assignment made at level k. As MAC descends it never performs any checks against $V_j \leftarrow v_\ell^j$ as this value has been pruned. Given that an oracle equipped BM visits the same nodes it too will reach the level where it tries to instantiate V_j. However, at this point BM might check the assignment $V_j \leftarrow v_\ell^j$ against assignments that were made below level k—the wipeout oracle cannot inform it that this value was pruned by arc-consistency at level k.

Formally, we provide BM with an arc-consistency domain wipeout oracle by replacing line $(*)$ in Figure 3 with:

$*$ **if** ok **and** not(DWOMAC(i))

where DWOMAC(i) is a call to an oracle testing for DWO at some future variable, where the oracle has the same domain wipeout detection power as AC constraint propagation.

[11]Other arc-consistency algorithms, AC-4 and higher [MH86, Bes94, BFR95], utilize support sets. However, it is not difficult to see that such support sets are simply cached lists of successful consistency checks. Ever time these algorithms access these support sets (or decrement support counts) they are doing something equivalent to a consistency check. In particular, one could extend the results below to such algorithms by finding an appropriate mechanism for counting accesses to the support sets as consistency checks.

[12]Like FC we regard MAC as having visited a node whenever it attempts the assignment $s_j \leftarrow v_\ell^j$ (line 4 in the FC/MAC procedure). For MAC there will be a larger set of such assignments that will subsequently be discovered (on the next line of code) to be impossible due to the fact that the value has already been pruned from the domain of V_j.

[13]The definition only considers checks of the form $(s_i, v_\ell^j) \in C_{\{i,j\}}$, where s_i is the assigned value for V_i

Theorem 3.5 *BM, supplied with an oracle as described above, explores exactly the same nodes as MAC. Furthermore, the number of consistency checks it makes is at least as great as, and may be more than, the number of tree-checks MAC makes.*

Proof: That an oracle equipped BM visits the same nodes MAC is not as immediate as with the FC algorithm. We prove it by induction on the nodes visited. The base case is obvious. The first node visited is the same for both algorithms.[14] By induction assume that we have visited exactly the same sequence of nodes up to node n_i. We need to prove that both algorithms visit the same the next node n_{i+1}. MAC will descend to the next level after visiting n_i iff n_i was consistent with the past variables, and arc-consistency did not detect a DWO among the future variables. An oracle equipped BM will descend to the next level under exactly the same conditions. Hence in this case both algorithms will then visit the same next node—they both will attempt to assign the same next variable its first value. On the other hand MAC can fail to descend to the next level only because (1) n_i is an assignment that is immediately discovered to be illegal because the value used has already been pruned or (2) the assignment made at n_i generates a DWO among the future variables. In both cases the next node visited by MAC will be an attempt to assign the deepest instantiated variable that has a next value, its next value (this could be the same variable assigned at n_i). Since, by inductive assumption, all of the previous nodes are the same, if BM fails to descend at n_i it also will try to assign this same variable its next value. That is, BM will visit the same next node as MAC, if it fails to descend at n_i. Clearly in case (2) the oracle will inform BM of the DWO and BM will not descend at n_i. In case (1) the fact that the assignment made at n_i was previously pruned by MAC must mean that this assignment lost all of its support on some variable's domain. This implies that the assignment at n_i will generate a DWO of that variable. As we move down the tree variable domains can only decrease in size. Hence, when BM tries this assignment further down the tree, at n_i, it will still generate a DWO and the oracle will inform BM of this fact. In summary, BM will always visit the same next node n_{i+1} as MAC, and by induction our claim holds.

To show that BM performs at least as many checks as MAC performs tree-checks we construct a one-to-one mapping from the set of MAC tree-checks to the set of BM (with an DWOMAC oracle) checks. Suppose that at node n_0 MAC checks the assignment $A_i = V_i \leftarrow \ell_i$ against the assignment $A_j = V_j \leftarrow \ell_j$, and that this check turns out to be a tree-check. This means that in the subtree below n_0 MAC later attempts the assignment A_j at some node n_1. Furthermore, let n_1 be the leftmost node in this subtree where A_j is attempted. By the above BM also visits n_1. Since MAC tested A_j at n_0, A_j must be consistent with the assignments made above n_0. Furthermore, since n_1 is the leftmost node below n_0 where A_j is attempted, the search must have backtracked above n_0 since it previously tested A_j. Putting these two facts together the argument made in case (3) of Theorem 3.4 applies and we see that BM will check A_i against A_j at n_1. We map the MAC

[14] We are assuming a static variable ordering. In fact, a dynamic variable ordering is fine, as long as both algorithms make identical choices. Furthermore, as we have written them both algorithms need to iterate over the domain of the first variable even if the problem is originally arc-inconsistent. If one wishes to code MAC so that it does an initial arc-consistency pruning of the domains, one can code BM so that it does an initial call to DWOMAC prior to entering level 1. The theorem will then continue to hold.

tree check of A_i against A_j to the BM check of A_j against A_i made at n_1. Since n_1 is specified to be the leftmost node in the subtree under n_0 where MAC attempts A_j it is clearly unique. Hence, the mapping maps each MAC tree-check to a unique BM check—it is a one-to-one map.

It could be that MAC visits n_0 and n_1, but at n_0 it does not check A_i against A_j. The value ℓ_j in V_j's domain might have been pruned. In particular, it can be that A_j is consistent with all of the assignments made above n_0, but was pruned because it lost support in the domain of some future variable. In this case it is not difficult to see that BM will still check A_j against A_i at node n_1, even though MAC did not make the reciprocal check at n_0. Using this, concrete examples are easy to construct which demonstrate that BM may make more checks that MAC makes tree checks. The above mapping, however, shows that the set of MAC tree checks can be no larger that the set of BM checks. ■

3.4 FC and MAC

To compare FC and MAC we can again synchronize them in terms of the number of nodes each visits. We can imagine line $(*)$ in Figure 1 being replaced by

$*$ **if** $\texttt{Domain}_l^i = 0$ **and** not(DWOMAC(i))

where DWOMAC(i) is a call to an oracle testing for DWO with the same power as AC constraint propagation. It is not difficult to show (using very similar arguments at those used in Section 3.2) that FC equipped with a DWOMAC oracle is identical to BM with this oracle. That is, FC will visit the same number of nodes and perform the same tree-checks as BM performs checks. The results of the previous section then imply that FC equipped with a DWOMAC oracle explores the same nodes as MAC, performs at least as many tree-checks as MAC, and sometimes performs more tree-checks than MAC.

3.5 Backjumping

Besides optimizing its tree-checks in a manner that is at least as efficient as BM, DWO detection allows FC and MAC to achieve backjumping (BJ) savings. This observation is essentially a corollary of a result proved in [KvB95], but we present a different proof that serves to make the connection with DWO detection more clearly.

Theorem 3.6 *Let n be a node visited by BT that is skipped by BJ. Then any algorithm that uses* DWO *detection would never visit n either.*[15]

Proof: BJ[16] can skip nodes only when it backjumps from some node n_1 (which is a partial assignment to the variables $V_1, \ldots, V_i$), to a higher node n_0 (which is a partial assignment to the variable

[15]Another corollary of this is the fact that any algorithm that utilizes dynamic variable ordering using the popular minimal remaining values heuristic also achieves all BJ savings [BvR95].

[16]Here we assume some familiarity with details of the BJ algorithm see [Pro93] for more details on BJ and CBJ.

$V_1, \ldots, V_j$, $j < i$). Any skipped node n will be a node in the subtree under some such n_0. For the backjump to have occurred from n_1 directly to n_0, every value of V_i must have been inconsistent with the assignments made above and at n_0. But then DWO detection would have discovered at n_0 that V_i had a domain wipe-out. Hence, any algorithm that used DWO detection would never have explored *any* node in the subtree under n_0, and in particular would not have visited n. ■

The more sophisticated version of BJ, conflict directed backjumping (CBJ [Pro93]) can, on the other hand, generate savings that are not subsumed by any obvious extension to FC. To be precise we can imagine algorithms that extend FC beyond the arc-consistency processing of MAC. Enforcing K-consistency as defined by Freuder [Fre78] requires adding $k-1$ arity constraints. Hence for $K > 3$ (i.e., beyond path-consistency) it is not clear how K-consistency can be maintained by a CSP tree search algorithm geared towards binary constraints. However, there is an alternative extension of FC that is motivated by the following observation about MAC.

Observation 3.7 *When MAC completes its processing at a node n and does not find a* DWO *we have the following condition. For every unpruned value v in the domain of any variable V, and for every variable V' not equal to V, there exists an unpruned value v' in the domain V' such that the assignments $V \leftarrow v$ and $V' \leftarrow v'$ are consistent.*

MAC removes any values v that fail to satisfy this condition. Using this idea we can define MIkC—maintaining inverse k-consistency. Inverse k-consistency first appeared in [Fre85] and was explored more fully in [FE96].

Definition 3.8 For any $k \leq N$, where N is the number of variables in the CSP, the MIkC algorithm is a tree search algorithm that backtracks when a DWO is detected. After every new instantiation MIkC performs FC constraint propagation, furthermore it removes all values v from the domain all future variables V that fail to satisfy the following condition. Given any set of $k-1$ future variables $\{V_1, \ldots, V_{k-1}\}$ not including V, there must exist a set of unpruned values $\{v_1, \ldots, v_{k-1}\}$ (with $v_i \in D_i$) such that the set of assignments $\{V \leftarrow v, V_1 \leftarrow v_1, \ldots, V_{k-1} \leftarrow v_{k-1}\}$ is consistent with every constraint over the set $\{V, V_1, \ldots, V_k\}$.

MIkC continually removes domain values that fail to satisfy the above condition until it has either detected a DWO or all unpruned values satisfy the condition. It is not difficult to see that M2IC is precisely MAC, and that if $k = N$, MIkC will have a backtrack free search.[17]

Now we can state our assertion about CBJ more precisely.

Proposition 3.9 *For any fixed k there exist CSPs where CBJ will allow MIkC to skip nodes it would otherwise explore.*

Proof: We demonstrate such a CSP. Let V_0 be a binary variable, $V_1, \ldots, V_{m+1}$ be variables with $m+1$ values in their domains, and X be some other variable not equal to any of the V_i ($0 \leq i \leq m$)

[17] Unfortunately, to achieve a backtrack free search in general would require an exponential amount of work at the root of the search tree!

and not constrained with any of the variables $V_1, \ldots V_{m+1}$. For each of the variables $V_1, \ldots, V_{m+1}$ the value $m+1$ is a special "don't care" value. In particular, $V_i \leftarrow m+1$ is consistent with every other assignment to every other variable. Besides this "don't care" value, $V_1, \ldots, V_{m+1}$ have a all-different constraint (expressed as $m(m+1)/2$ binary constraints) over their other values. That is, if one of the members of the set $\{V_1, \ldots, V_{m+1}\}$ takes on the value $h < m+1$ no other member of the set can take on this value. Finally, we impose a constraint between V_0 and each of the V_1 thru V_{m+1}. The don't care value $m+1$ for each of these variables is incompatible with the assignment $V_0 \leftarrow \mathbf{false}$.

Say that MIkC, with $k < m$, first makes the assignment $V_0 \leftarrow \mathbf{false}$. In the subtree below the variables $V_1, \ldots, V_{m+1}$ are now reduced to having m possible values for $m+1$ variables all of which have to be different. That is, the subtree below contains no solution. Nevertheless, since $m > k$, every value of these variables can be consistently extended to any $k-1$ other variables. Hence MIkC will not prune any values from these variables.

Say that next MIkC instantiates X and then instantiates the first of the V_i variables, V_i. At some point (when $m-k$ of the V_i have been instantiated) it will detect a DWO, eventually backtracking to try the next value of X. If equipped with CBJ, however, once it has tried all assignments to V_1 CBJ will allow it to jump back to V_0 skipping any further assignments to X. (Clearly, V_1's conflict set at the time we have tried all of its values cannot include X, since X is not constrained with any of the V_i variables below it.) ■

3.6 Summary of the Formal Relationships

Our results are summarized in Figures 5 and 6. Figure 5 shows a hierarchy between the algorithms in terms of number of nodes visited.

Figure 6 gives the hierarchy in terms of checks. In this diagram the checks performed by the algorithms are divided into components. For most of the nodes we only have arrows between components of these algorithms. As discussed above, checks used to look for DWOs can have the benefit of allowing the algorithm to visit fewer nodes. Hence the total number of checks performed by these algorithms will vary—there will be a tradeoff between the number of nodes eliminated because of DWO detection and the extra work performed at each node to do DWO detection. Each of algorithms may perform more or fewer total checks dependent on the problem being solved.

CBJ does not affect the checks performed at a node, it simply reduces the number of nodes visited. Hence, for some of the algorithms that include CBJ we draw a thick arrow to indicate a relationship between the *total* number of checks performed by the linked algorithms.

We have developed these results using the simplifying assumption of a static variable ordering. In fact, these result still hold even when a dynamic variable ordering is used. Clearly, if two algorithms use *distinct* variable orderings (dynamic or static) no comparisons can be made—it is easy to develop examples where one ordering yields a backtrack free search while another takes exponential time. However, if at each node the two algorithms choose the same variable to instantiate next, then all of our proofs continue to hold. In particular, the algorithms may pursue different

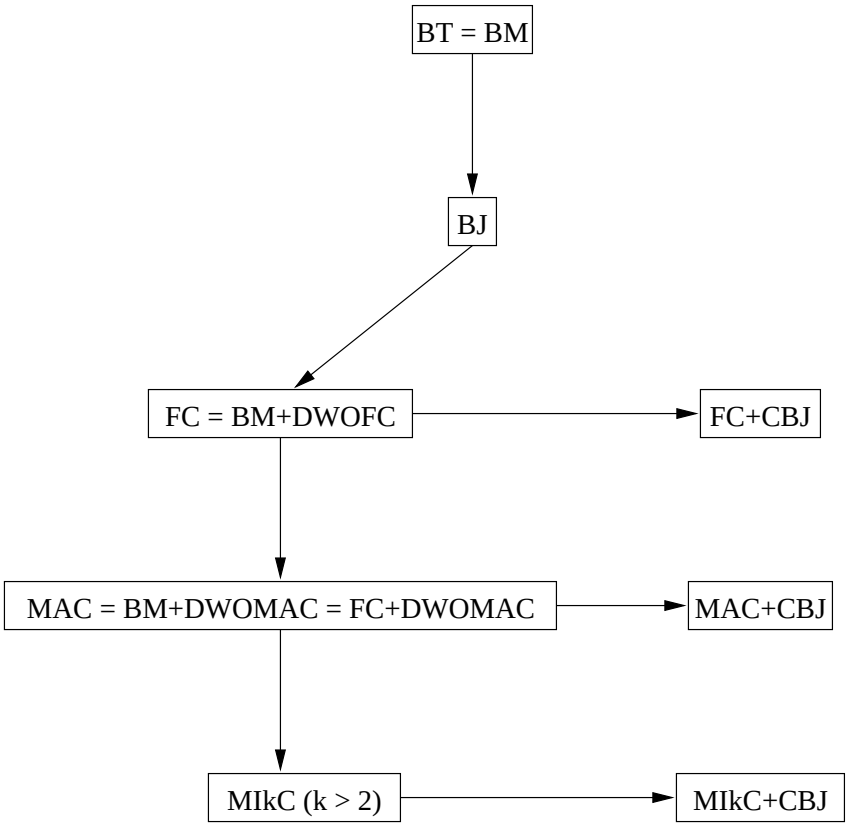

Figure 5: Hierarchy with respect to number of nodes visited. An arrow from A to B indicates that algorithm B visits a subset (not necessarily proper) of the nodes visited by algorithm A.

variable orderings along different branches (i.e., they may use a dynamic variable ordering) as long as they are both using the same dynamic ordering.

4 Conclusion

Strictly speaking, forward looking algorithms like FC and MAC and backwards looking algorithms like BT, BJ, and BM, are incomparable by worst-case complexity measures. This, coupled with the seemingly radical difference between looking forward and looking backward, might lead to the view that no interesting formal comparison is possible.

We have shown, however, that there is in fact a great deal of similarity in the processing done by these algorithms. In particular, we have shown that the processing performed by the forward looking algorithms we have considered can be broken up into two distinct components. One of these components is a tree-search procedure that is essentially the BM algorithm, while the other is a component that performs DWO detection.

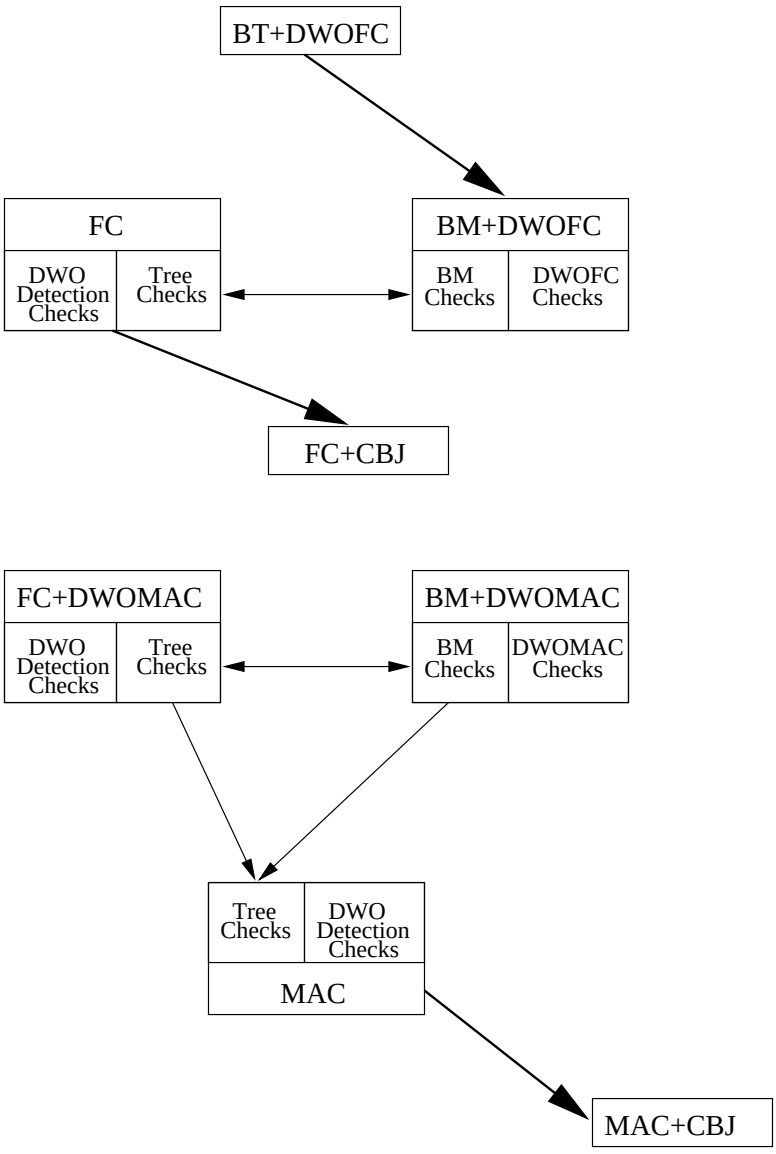

Figure 6: Hierarchy with respect to the number of checks. An arrow from box A to box B indicates that the algorithm component in B makes no more checks than the algorithm component in box A. A thick arrow indicates that this relationship holds for the total number of checks, not just for a component of the algorithm.

Besides increasing our understanding of these algorithms, this insights can also lead to practical improvements. In closing we mention three possible ways in which these insights might be exploited.

First, we might look for more efficient ways to implement each component. In particular, we can look for more efficient ways to do DWO detection. We have previously shown in [BG95], that applying this principle leads naturally to the "minimal forward checking" (or lazy forward checking) algorithm developed by Dent and Mercer [DM94]. This algorithm does not forward check every value of all of the future variables—this is more work than is required to detect a DWO. Rather, it simply ensures that every future variable has at least one value consistent with the current set of assignments. A change to the DWO component also requires a change to the "tree-search" component so that (1) it can fully utilize the work performed by the new DWO component without introducing redundant computation, and (2) it performs some extra computation to make up for the fact that the more efficient DWO component cannot provide it with as much information as before. It may also be possible to apply this principle of minimal DWO detection to MAC by utilizing the minimal version of arc-consistency developed by Schiex et al. [SRGV96].

Second, we can use the insight that FC's domain pruning is accomplishing the same thing as BM to develop more efficient versions and implementations of other CSP algorithms. In particular, Prosser [Pro95] has developed an algorithm he called forward-checking with backmarking. However, we have shown that FC already achieves all BM savings. In examining Prosser's proposal it becomes clear that he has identified some *additional* improvements to FC. He develops a method of taking advantage of these improvements by using BM type data structures, i.e., using `Mcl` and `Mbl` arrays. The resulting algorithm is quite cumbersome, as there are considerable complications involved in keeping FC's `Domain` data structure synchronized with these additional BM data structures.[18] We have shown that the domain pruning processing maintained in the `Domain` data structure achieves the same effect as the backmarking processing maintained in the `Mcl` and `Mbl` data structures. In a forthcoming paper [Bac99] it is shown how Prosser's improvements to FC (and more) can be achieved without resort to additional data structures, considerably simplifying the resulting algorithm.

Finally, another practical direction this work points to is the development of analytical and empirical tools to help predict what level of forward constraint propagation would be best for a particular problem. There is a cost/benefit tradeoff in doing DWO detection. It may be possible to develop a greater understanding of the relationship between the tightness of the constraints and the optimal level of effort to invest in DWO detection. In fact, it may be the case that an adaptive algorithm that does varying amounts of work at different nodes (dependent on an analysis of the remaining sub-problem under that node) is the most efficient way of utilizing forward constraint checking.

[18] And in fact Prosser's algorithm does not achieve all of the potential savings due to these synchronization problems.

References

[Bac99] Fahiem Bacchus. Enhanced forward constraint checking CSP algorithms. Technical report, University of Waterloo, 1999.

[BE75] J. R. Bitner and Reingold E. Backtracking programming techniques. *Communications of the ACM*, 18(11):651–656, 1975.

[Bes94] C. Bessiere. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, 1994.

[BFR95] C. Bessiere, E. C. Freuder, and J.-C. Regin. Using inference to reduce arc consistency computation. In *Procceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 224–230, 1995.

[BG95] Fahiem Bacchus and Adam Grove. On the Forward Checking algorithm. In *First International Conference on Principles and Practice of Constraint Programming (CP95)*, number 976 in Lecture Notes in Computer Science, pages 292–309. Springer-Verlag, New York, 1995.

[BR96] C. Bessiere and J.-C. Regin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *Second International Conference on Principles and Practice of Constraint Programming—CP96*, number 1118 in Lecture Notes in Computer Science, pages 61–75. Springer-Verlag, New York, 1996.

[BvB98] 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.

[BvR95] Fahiem Bacchus and Paul van Run. Dynamic variable reordering in CSPs. In *First International Conference on Principles and Practice of Constraint Programming (CP95)*, number 976 in Lecture Notes in Computer Science, pages 258–275. Springer-Verlag, New York, 1995.

[DM94] M. J. Dent and R. E. Mercer. Minimal forward checking. In *6th IEEE International Conference on Tools with Artificial Intelligence*, pages 432–438, New Orleans, 1994. Available via anonymous ftp from ftp://csd.uwo.ca/pub/csd-technical-reports/374/tai94.ps.Z.

[FE96] E. Freuder and C. D. Elfe. Neighborhood inverse consistency preprocessing. In *Proceedings of the AAAI National Conference*, pages 202–208, 1996.

[Fre78] E. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, 1978.

[Fre85] E. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32(4):755–761, 1985.

[Gas77] J. Gaschnig. A general Backtracking algorithm that eliminates most redundant tests. In *Procceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, page 457, 1977.

[Gas78] J. Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proceedings of the Canadian Artifical Intelligence Conference*, pages 268–277, 1978.

[Gas79] J. Gaschnig. *Performance measurement and analysis of certain search algorithms*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, U.S.A., 1979. Tech. Rept. CMU-CS-79-124.

[GS95] 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.

[HE80] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[KvB95] Grzegorz Kondrak and Peter van Beek. A theoretical evaluation of selected backtracking algorithms. In *Procceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, 1995.

[Liu98] Zhe Liu. Algorithms for constraint satisfaction problems CSPs. Master's thesis, Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1998. Available via anonymous ftp at "logos.uwaterloo.ca" in the file "/pub/bacchus/liu.ps.gz".

[Mac77] A. K. Macworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mac87] A. K. Mackworth. Constraint satisfaction. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*. John Wiley and Sons, New York, 1987.

[MH86] R. Mohr and T Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[Nad89] Bernard A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.

[Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3), 1993.

[Pro95] P. Prosser. Forward checking with backmarking. In M. Meyer, editor, *Constraint Processing*, LNCS 923, pages 185–204. Springer-Verlag, New York, 1995.

[Reg95] J.-C. Regin. *Developpement d'outils alogorithmiques pour l'Intelligence Artificielle. Application a la chimie*. PhD thesis, Universite Montpellier II, France, 1995.

[SF94] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the European Conference on Artificial Intelligence*, pages 125–129, 1994.

[SF97] D. Sabin and E. C. Freuder. Understanding and improving the MAC algorithm. In *Third International Conference on Principles and Practice of Constraint Programming (CP97)*, number 1330 in Lecture Notes in Computer Science, pages 67–81. Springer-Verlag, New York, 1997.

[SRGV96] T. Schiex, J.-C. Regin, C. Gaspin, and G. Verfaillie. Lazy arc consistency. In *Proceedings of the AAAI National Conference*, pages 216–221, 1996.

[Tsa93] E. Tsang. *Foundations of Constraint Satisfaction Algorithms*. Academic Press, New York, 1993.

[vR94] Paul van Run. Domain independant heuristics in hybrid algorithms for CSPs. Master's thesis, Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1994. Available via anonymous ftp at "logos.uwaterloo.ca" in the file "/pub/bacchus/vanrun.ps.gz".