On The Forward Checking Algorithm

Fahiem Bacchus[1] and Adam Grove[2]

[1] Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1,
(fbacchus@logos.uwaterloo.ca)
[2] NEC Research Institute, 4 Independence Way, Princeton NJ 08540, USA,
(grove@research.nj.nec.com)

Abstract. The *forward checking* algorithm for solving constraint satisfaction problems is a popular and successful alternative to backtracking. However, its success has largely been determined empirically, and there has been limited work towards a real understanding of why and when forward checking is the superior approach.

This paper advances our understanding by showing that forward checking is closely related to *backmarking*, which is a widely used improvement of ordinary backtracking. This result is somewhat surprising, because (as their names suggest) forward checking is superficially quite different from backtracking and its variants. The result may also help in predicting when forward checking will be the best method.

Finally, the paper shows how the relationship to backmarking helps understand a recently introduced improvement to the forward checking algorithm, known as *minimal forward checking*. We argue that the new algorithm is best viewed as a hybrid combination of backmarking and forward checking.

1 Introduction

Constraint satisfaction problems (CSPs) [Mac87] are typical of the NP-complete combinatorial problems that are so pervasive in AI. Plain *backtracking* (BT) is an algorithm for solving CSPs that has been known for at least a century [BE75], but it is far from the best. There are easy improvements to backtracking such as backjumping (BJ) [Gas78] and backmarking (BM) [Gas77], which never do worse than backtracking [Kon94], and generally do much better. There are also simple alternatives to backtracking, notably *forward checking* (FC) and its variants [HE80].

Our main topic in this paper is to further our understanding of forward checking, which has extensive empirical but limited theoretical support as one of the very best among the class of simple, general, CSP algorithms [Nad89]. Because of its demonstrated practical success, it is important to discover as much as possible about when and why FC is superior to other approaches.

It can be argued that since the general class of CSPs are NP-complete, there are unlikely to be any major distinctions between the various algorithms. But this is too pessimistic. While any NP-complete family must contain impractically hard problems, it is also likely to contain large subclasses of simpler problems. Indeed, the "region" of truly hard classes can be rather small [CKT91], and the particular instances we encounter in practice may well be outside of this region. Ideally we would identify the

simple classes and develop special purpose fast algorithms to solve them, but this is usually impractical. Instead, we can search for better general techniques that less often display exponential behavior. Backmarking, backjumping, and forward checking have all shown themselves to be practical improvements over backtracking: they are all able to solve problems that defeat plain backtracking.

All these CSP algorithms examine partial solutions, which are assignments to a subset of the variables, and try to extend these until all variables are assigned. BM is a variant of BT that saves a number of redundant consistency checks by some straightforward bookkeeping. BJ is another variant of BT that saves a distinct set of consistency checks from BM, this time by detecting and avoiding parts of the search tree that cannot contain any solutions. FC, on the other hand, is quite different from BT. In particular, it orders its consistency checks in a completely different way. BT and its variants do all of their checks backwards: whenever a new assignment is make, it is checked for compatibility against all previous assignments. FC does all of its checks forward: whenever a new assignment is made, it checks that assignment against all future, as yet uninstantiated, variables, keeping track of the implications of these checks. In Section 2 we discuss the FC algorithm in detail and present some additional background. FC can do exponentially better than BT and its variants, but it can also do worse. However, in Section 3 we observe that there is a tight polynomial bound on how much worse it can do. The seemingly very different nature of FC, coupled with the known fact that it can do worse, appears to have hindered a serious study of its relationship to the other algorithms. However, as we show, there are in fact important connections between FC and BT, BM, and BJ.

There seem to be a couple of different ways in which FC can be beaten by BT and its variants, and both lead to important insights about their relationship. We examine one example in Section 3, where we present the first of our main results. Roughly speaking, this is as follows. FC achieves its success (or otherwise) over the backward checking algorithms in two ways. First, it optimizes checks over its search tree in *exactly* the same way that BM does. And second, it performs other checks that work as an "early-warning-system" of inconsistency. Our result requires some care in formulation, but it essentially says that, were one to provide BM with an oracle that provides the same warnings, and simultaneously ignore the cost that FC incurs in finding these warnings, the two algorithms would have equivalent complexity.

So we see that there is a particular cost/benefit tradeoff that must be considered when comparing FC to BM, and in specific types of problem the net effect of this tradeoff may be apparent. While it is hard to make many further generalizations about the value of FC, another interesting result is due to Kondrak [Kon94]. He shows that every node that BJ avoids searching (over BM or BT) is also avoided by FC. In a sense, then, FC always includes the specific savings of *both* BM and BT, but it also incurs some additional cost in the hope of being even more efficient than either. The connection between BM and FC that we derive here has, to the best of our knowledge, never been noticed before. Prosser [Pro93a] has previously presented a modification of FC that allows it to realize some *additional* BM-type savings. But his modification is distinct from the results we prove here. We will discuss Prosser's work in more detail in Section 3.

In Section 4 we briefly discuss an improved version of forward checking first pro-

posed by Zweben and Eskey [ZE89], and subsequently presented independently, and in greater detail, by Dent and Mercer [DM94]. Dent and Mercer call their improved algorithm *minimal forward checking* (MFC). MFC was proposed as a "lazy" version of forward checking which avoids doing checks until they are absolutely necessary. MFC provably never does worse than the original FC algorithm. Its gains over plan FC are typically modest (10-40% in our own experiments, which seems to be in fair agreement with [DM94]), but are nevertheless worthwhile.

We re-discovered this algorithm not by thinking about lazy evaluation, but instead as a corollary of our results connecting backmarking and forward checking. As we show, there is a strong sense in which minimal forward checking is a logical hybrid of regular forward checking and backmarking that benefits from the advantages of both. Thus the algorithm that Zweben and Eskey, and Dent and Mercer, present as essentially a clever optimization of regular forward-checking has what is arguably quite a deep foundation.

In this section we also point out that MFC can be combined with dynamic variable reordering heuristics in a manner that preserves its performance edge over plain FC (with variable reordering). Thus showing that some of the conclusions of Dent and Mercer [DM94] on this issue are overly pessimistic. In fact, MFC in combination with dynamic variable reordering showed itself to be one of the most effective algorithms when tested against a wide range of other algorithms.

2 Preliminaries

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

Definition 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 each pair of variables $\{V_i, V_j\}$, a *constraint* $C_{\{i,j\}}$ between D_i and D_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 $\left\{V_i \leftarrow v_l^i, V_j \leftarrow v_m^j\right\}$ is *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. ∎

The forward checking algorithm [HE80] constructs solutions by considering assignments to variables in a particular order, which for concreteness we take to be $V_1, V_2, V_3, \ldots, V_N$.[3] Suppose that we have found a consistent assignment to the first $i-1$ variables, which means that all pairwise comparisons 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

[3] That is, we assume a static ordering in this paper. But see Section 4 for a discussion of dynamic variable orderings.

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 $\texttt{Domain}_m^j$ will contain 0 if and only if the assignment $V_j \leftarrow v_m^j$ is consistent with the assignments chosen for all the past variables. Otherwise, it contains the index of the first (i.e., the lowest) assigned variable with which $V_j \leftarrow v_m^j$ is inconsistent.

```
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 then
            if i = N then
                print s_1, ..., s_N
            else
                if Check-Forward(i) then
                    FC(i+1)
                Restore(i)

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

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

Fig. 1. Pseudo-code for Forward Checking

It follows that, when we are considering a possible value v_l^i for the current variable V_i, it is sufficient to look for a zero in $\texttt{Domain}_l^i$. Any such value is guaranteed to be consistent with all past choices. Hence, 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 `Domain` 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 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, of course, 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.[4]

[4] 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.)

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.

Our own experiments and those of Haralick and Elliot [HE80], Nadel [Nad89], Prosser [Pro93b] and van Run [vR94], have shown that the work expended by FC to perform DWO detection generally results in a net gain. In the n-Queens problem FC is slightly outperformed by BM, but in Nadel's confused queens, Prosser's version of the Zebra problem, and in a number of random tests (including extensive tests from Frost and Dechter's table of 50% solvable classes [FD94]) FC outperforms BM, is generally much better than BJ, and always totally out classes BT. In most of these tests the measure of complexity was taken to be the number of consistency checks. Several other complexity measures have been considered in the literature, the most popular being the number of nodes visited and CPU time. Counting the nodes visited is not an appropriate measure for comparing the performance of FC and the backward checking algorithms, as the amount of work FC does at each node is completely different from the other algorithms.[5] CPU time, on the other hand, is a very difficult measure to evaluate correctly as it is extremely implementation dependent. For this reason we also will focus on counting the number of consistency checks in our theoretical and experimental results.

Another major advantage of FC is that the number of remaining consistent values for each of the future variables can be computed without any additional constraint checks. 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 reordering. Bacchus and van Run [BvR94] have shown that FC and its variant FC-CBJ, when equipped with dynamic variable reordering using the MRV heuristic, outperform a wide range of similarly equipped backwards checking algorithms.

3 Forward checking—some theoretical results

Gashnig's backmarking algorithm [Gas77] improves backtracking by eliminating some 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

[5] Nodes visited can be a useful measure for comparing features other than performance.

any checks at all; the assignment will fail again, so we can immediately reject it. To realize these savings, the `Mcl` array is not quite enough by itself, because its entries are not necessarily up to date. Thus BM uses an additional array `Mbl` ("minimum backtrack level") which for each variable keeps track of how far we have backtracked since trying to instantiate this variable. (In the example above, $\texttt{Mbl}^i$ will store the information that we have only backtracked to V_j since last visiting V_i. Thus we know to ignore any information in $\texttt{Mcl}_l^i$ that pertains to variable V_j or later.) Figure 2 gives the backmarking algorithm in more detail.

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

Fig. 2. Backmarking

As we have mentioned, in empirical tests BM and FC often vie for top honors. But the differences can be enormous. 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 [Kon94] has shown that FC always explores a subset (not necessarily proper) of the nodes (i.e., partial instantiations) that BT, BM, and BJ visit. Nevertheless, since FC can perform more checks per node, FC may perform more consistency checks.

Example 1. Suppose V_2 and V_3 are mutually inconsistent. The backward checking algorithms can discover this quickly, only searching 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 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 DWO, 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's result is worth making explicit.

Remark. Let K be the largest domain size. FC never performs more than NK times as many consistency checks as BT, BJ or BM, 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 does no work at these nodes either.

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 obvious that the important feature of FC is that it can use DWO detection to prune large amounts of the search space, and in doing so save itself 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. This connection between FC and BM has, to our knowledge, never been made before, and the main result in this section is to make precise this connection. Taking account of the similarity between FC and BM allows us to make the difference—which is exactly DWO detection—clear. Our first definition partitions the work (i.e., the consistency checks) that FC performs into two groups.

Definition 2. A particular consistency check $(s_i, v_\ell^j) \in C_{\{i,j\}}$ performed during the execution of the FC algorithm (Figure 1) is called a *tree-check* if the algorithm later attempts the partial assignment $s_j \leftarrow v_\ell^j$,[6] while $s_1, \ldots, s_i$ remain the same as at the time the check was made. (That is, $s_j \leftarrow v_\ell^j$ is later attempted in the subtree below the node where the check was make). ∎

Example 2. Consider the example shown in Figure 3. 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

[6] We consider an assignment to be attempted as soon as FC executes the code $s_i \leftarrow v_\ell^i$ (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.

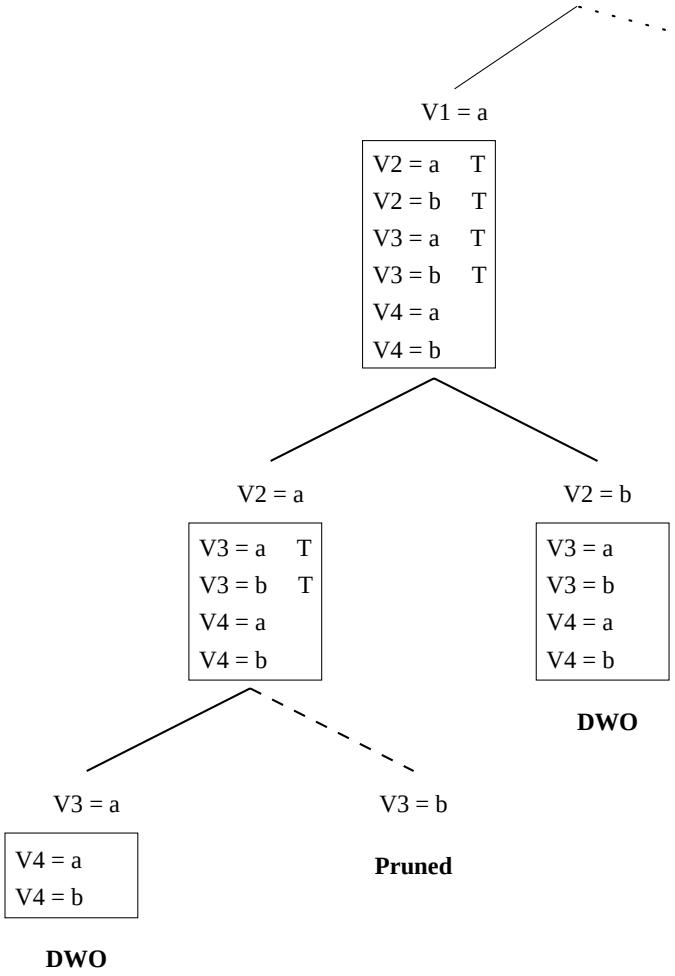

Fig. 3. Tree Checks: only the checks labled with "T" are tree checks.

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 trees show 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 ariving at this node. Nevertheless,

we consider FC to have visited that node, as it executes the assignment $s_3 \leftarrow v_l^3$ (line 4 in FC procedure) prior to checking to see if the value v_l^3 has already been pruned. Hence, we show this node as being connected via a dotted line, and we count the check against $V_3 = b$ to be a tree check.

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.

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 has the practical disadvantage that it can only be made after the fact.[7]

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 "oracle" that, given any partial assignment, can tell whether there is DWO at some future variable. We can imagine line $(*)$ in Figure 2 being replaced by:

$*$ **if** ok **and** not(DWO(i)) **then**

where DWO(i) is a call to the oracle testing for DWO 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, whose proof is omitted due to space limitations.

Theorem 3. *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.*

That BM, when equipped with a DWO oracle should visit exactly the same nodes as FC is not surprising. However, that BM does no fewer checks than the tree-checks of FC is. It must be remembered that BM utilizes some subtle bookeeping 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 bookeeping. It would seem that except for DWO detection FC would simply be doing its checks in the same manner as BT. Our theorem show that this is not the case. FC's domain pruning allows it to achieve all of the savings of BM's sophisticated bookeeping; it is doing its checks in a much more "intelligent" manner than plain BT.

This theorem aids our understanding of these algorithms by getting to the heart of their similarities and differences. It also has some practical implications. For instance,

[7] 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.

for some CSPs it may be clear that few DWO's will occur for distant future variables, in which case we now *know* (as contrasted to merely having a vague intuition) that FC will be outperformed by BM: BM does its checks as efficiently as FC without expending extra checks on the gamble of DWO detection. Perhaps deeper analysis will help us get a more quantitative understanding of the cost/benefit tradeoff of DWO detection in various problems. This result also helps us explain the empirical effectiveness of FC; BM's savings over BT can be very substantial in practice, so FC ability to capture all of these savings can make it very efficient even in cases where not much is saved by its DWO detection. This suggests that on some problems, the "forward looking" aspect of FC might be a misleading explanation for its success and that what is really important is its embedding of backmarking savings. Finally, we present another practical application of our result in the next section, where it is used to motivate an improvement to FC. Not coincidentally, the improvement makes the connection to BM more apparent.

Besides optimizing its tree-checks in exactly the same manner as BM, FC's DWO detection allows FC to achieve BJ savings. This observation is essentially a corollary of Kondrak's result, but we present a different proof that serves to make our point more clearly.

Theorem 4. *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.*

Proof. [8] BJ 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 lower level node n_0 (which is a partial assignment to the variable $V_1, \ldots, V_j$, $j < i$). 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 partial assignment at n_0. But then DWO detection would have discovered that V_i had a domain wipe-out at n_0. 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. ∎

A very similar argument can be used to show that, if all algorithms are supplied with a DWO oracle, then BT becomes equivalent to BJ, and BM becomes equivalent to BMJ (Prosser's [Pro93b] backmark-jumping algorithm). So Theorem 3 also holds for BMJ supplied with an oracle.

Theorem 3 demonstrates that FC optimizes its tree-checks in the same manner as BM. In particular, it realizes all of the BM savings in doing these checks. Prosser, in [Pro93a], has developed a technique whereby FC can achieve even more BM-type savings by maintaining information gathered whenever a domain wipe-out occurs. This information and the savings that can be realized from it are orthogonal to the BM savings already embedded with in FC. That FC already embeds the savings of standard BM was not noticed by Prosser. Interestingly, it would appear that Prosser's technique can easily added to the minimal forward checking technique described below. Such a combination would be worth exploring.

[8] Here we assume some familiarity with details of the BJ algorithm which, due to space limitations, we are unable to provide in the text.

4 Minimal Forward checking

The simplest example for which plain backtracking outperforms FC is where the assignments $V_i \leftarrow v_1^i$ (i.e., each variable is assigned the first value in its domain) are consistent, and where we are content to find a single solution. In this case, backtracking can descend immediately to a solution (the leftmost branch of the backtrack tree), while forward checking is held up checking each variable on that branch against all possible values of future variables. This example of BT outperforming FC is often mentioned in the literature, e.g., [Pro93b].

However, it is possible to improve the behavior of forward checking in such cases. The technique has been noted by Zweben and Eskey in [ZE89] and in more detail by Dent and Mercer in [DM94]. The idea motivating this technique is that of lazy evaluation, a notion from functional programming languages [ASJ85]. This is the notion of delaying computing things until absolutely necessary.

Standard FC checks every value in the remaining domains of the future variables, pruning those values that are inconsistent with the current assignment. If during this pruning phase it detects DWO it retracts the current assignment. Hence, we see that to apply the notion of lazy evaluation, all we need to compute immediately is whether or not DWO occurs. We do not need to check every value in the future domains at this stage, and we can delay these checks until we absolutely need it. By delaying these checks it may turn out that a DWO is detected before we need to perform them, hence a lazy version of FC can avoid performing some of the checks standard FC performs.

In the example, we only need to check forward against the first value of each subsequent variable's domain. In general, it is enough to find a single consistent value in the domain of each future variable to determine whether or not DWO occurs. In delaying the other consistency checks, however, when we backtrack, and need to re-instantiate a variable, we are no longer guaranteed that all relevant backward consistency checks have been performed. But whenever this happens, we can simply "catch up" by performing the appropriate checks. That is, we reach a stage where we must complete the delayed computations. If these tree checks are performed only "on demand" we can end up doing far fewer than if all *potentially* useful checks had been done during the forward checking phase. The overhead needed to keep track of what checks have been done is negligible, and so this idea leads to worthwhile gains.

The results of Section 3 led us to rediscover the MFC algorithm, but from notions quite different from lazy evaluation. Consider Theorem 3, which we can roughly paraphrase as saying that FC is essentially BM with DWO detection added. Our idea was to take this literally; i.e., implement BM and then enhance it with DWO detection. The point of this is that it becomes quite clear that we have the freedom to implement DWO detection as efficiently as possible—in particular, we can stop checking a future variable as soon as we have found a consistent value. In the regular presentation of forward checking, the forward consistency checks are doing several different things at once, and so it takes more insight to notice safe optimizations. Given our theorem, the situation is much more obvious.

Admittedly, some care is necessary. If we were to implement BM and DWO detection entirely independently, we may end up repeating checks. In particular, the DWO detection phase will perform some checks that BM also needs to perform. One feature of FC is

that it can avoid repeating these checks. Nevertheless, it is easy to can arrange things so that relevant checks in the DWO (forward checking) stage are remembered, and made available to the backmarking component. With this modification the hoped for gains do appear.

In our implementation of this idea, the main step is to modify the interpretation and use of the `Domain` array slightly. Now, we will use a *negative* value to note that a domain value was inconsistent with some past assignment. That is, $\texttt{Domain}_l^i = -j$ will signify that $V_i \leftarrow v_l^i$ was consistent with the current assignments up to s_{j-1}, but failed against s_j. We do this because the algorithm no longer checks every future value right up to the current assignment, even if it is consistent to that point. Furthermore, we wish to use positive values to remember the latest variable successfully checked against. Thus, $\texttt{Domain}_l^i = j$ means that $V_i \leftarrow v_l^i$ is consistent up to *and including* s_j. In particular, the initial value $\texttt{Domain}_l^i = 0$ means that $V_i \leftarrow v_l^i$ has not yet been checked against anything at all. We note that this is a different "accounting scheme" than that used by Dent and Mercer, who keep track of exactly which checks have been performed against earlier variables (rather than just the (signed) index of the last variable). Our scheme is more efficient as it requires $O(NK)$ space as compared to the $O(NK^2)$ space required by Dent and Mercer's scheme (where N is the number of variables and K is the size of the largest domain). More importantly, however, it makes the connection to BM much clearer. Nevertheless, we emphasize that this is a relatively minor implementation detail and the algorithm we were lead to by Theorem 3 is identical with Dent and Mercer's in all important respects.

Figure 4 shows the pseudo-code for the algorithm. The changes over FC have been marked and in addition the Update procedure is completely new. Procedure Check-Forward implements DWO, in the efficient fashion discussed above. In the main procedure MFC, we can no longer automatically assign values to s_i, as we did in FC. Rather, we must first catch up on any of the consistency checks that have been omitted. Catch up occurs in Update, which performs backwards checks. Note that it only checks the value v_m^j against the assignments it has not yet been checked against (i.e., the assigned values after the $\texttt{Domain}_m^j$'th). Furthermore, if $\texttt{Domain}_m^j$ is initially negative we perform no checks at all, as we know that this value is still inconsistent. This corresponds precisely to BM rejecting a value without incurring any checks when $\texttt{Mcl}_m^j < \texttt{Mbl}^j$. The only real difference between Update and BM is that the forward checking phase may have done more work than plain BM. That is, $\texttt{Domain}_m^j$, which plays a similar but more flexible role to the `Mbl` array in BM, may have a value greater than $\texttt{Mbl}^j$ in BM. This allows MFC to avoid repeating checks performed during the forward checking phase. Our purpose in presenting this pseudo-code is to highlight the Update procedure and its connection to BM. Of course, our whole point is that this is not a coincidence: the update code essentially *is* doing backmarking.

Clearly MFC performs no more checks than regular FC. Furthermore, as might be expected from the argument we gave, the relation that FC has to BM still holds:

Theorem 5. *BM with an oracle as in Section 3 explores the same nodes as MFC, and the number of consistency checks it makes is equal to the number of tree checks MFC makes.*

Thus, as should be expected from our description of the algorithm, MFC gains solely

```
procedure MFC(i)
%Tries to instantiate V_i
    for each v_l^i ∈ D_i
        s_i ← v_l^i
!       if Update(i,l,i-1) then
            if i = N then
                print s_1, ..., s_N
            else
                if Check-Forward(i) then
                    MFC(i+1)
                Restore(i)

procedure Restore(i)
%Returns Domain to previous state
    for j = i + 1 to N
        for each v_m^j ∈ D_j
!           if Abs(Domain_m^j) = i then
                Domain_m^j ← i−1

function Check-Forward(i)
%Checks s_i against future variables
%Only does enough work to find DWO
    for j = i + 1 to N
        DWO = TRUE
!       for v_m^j ∈ D_j and while DWO
!           if Update(j,m,i) then
                DWO = false
        if DWO then return(false)
    return(true)

function Update(j,m,i)
%Checks v_m^j against s_1 to s_i
%Updates Domain_m^j appropriately
    if Domain_m^j ≥ 0 then
        ok ← true
    else
        ok ← false
    for p = Domain_m^j + 1 to i and while ok
        if (s_p, v_m^j) ∉ C_{p,j} then
            ok ← false
            Domain_m^j ← −p
    if ok then
        Domain_m^j ← i
    return(ok)
```

Fig. 4. "Minimal" Forward Checking

by performing fewer non-tree checks.

We have tested MFC on various problems, including n-Queens, Prosser's version of the Zebra problem [Pro93b], and many random CSPs drawn from Frost and Dechter's table of 50% solvable classes [FD94]. As can be seen in Figure 5, MFC results in modest gains over FC, typically in the range of 10–30%. But in view of the fairly small changes over forward checking this is surely worthwhile. Figure 6 gives a comparison of the number of non-tree checks for both FC and MFC, to give some idea of the size of the savings here (which are surprisingly uniform across problem type). These results are in agreement with those of [DM94].

For the 10, 11, and 12 Queen problems we see that FC and MFC perform fewer tree checks than the number of checks performed by plain BM. Subtracting these numbers of tree checks from the corresponding number of checks performed by plain BM gives us the amount of savings that DWO detection yields. The data indicates that for these problems we do get savings from DWO detection, but looking at the non-tree checks we see that these savings are approximately equal to their costs. Hence, there is very little benefit, for n-Queens, in doing DWO detection. The main reason then, that FC performs about as well as BM in n-Queens is that FC optimizes its tree-checks just like BM.

We have also tested MFC using dynamic variable reordering, (MFCvar in the tables).

	MFC	FC	BM	BT	MFCvar[9]	FC-CBJvar[10]
10 Q[11]	220	242	220	1298	199	204
11 Q	1038	1155	1027	7417	904	935
12 Q	5298	5959	5225	—	4471	4635
Zebra[12]	129	182	1608	16196	2.8	2.9
R1[13]	11.6	15.1	73.8	511.1	0.84	0.86
R2	439	685	2447	—	23.6	27.7
R3	311	469	5753	—	3.6	3.9
R4	54	80	115	908	12	14

Fig. 5. Number of consistency checks (thousands) for various algorithms

	10 Queens	11 Queens	12 Queens	Zebra	R1	R2
Tree checks	134	616	3127	30	2.9	82.2
FC non-tree	108	539	2832	152	12.2	602
MFC non-tree	86	422	2171	99	8.7	357

Fig. 6. Tree and non-tree checks (1000's)

The reordering heuristic we used was the minimum remaining values heuristic whereby the next variable instantiated is that variable with fewest remaining consistent values (a fail-first heuristic). Because MFC delays performing consistency checks on the future variables, it does not provide an accurate count of the real number of remaining consistent values. Some of the values in the future domains will have been eliminated in MFC's search for the first consistent value, so the number of unpruned values can be used as a rough estimate of the true number of remaining values. This is the estimate used by Dent and Mercer [DM94]. However, they found that since they were only using an estimate, MFCvar was often beat by FCvar (which has the exact values). They concluded that MFCvar was inappropriate for large problems.

This conclusion is, however, overly pessimistic. Although MFC does not know the exact numbers of remaining values, it does have information that can be used to optimize the computation of which variable has the smallest domain. That is, it is not necessary to find the sizes of all future domains (as FC does) but only to find the smallest domain. To compute this we need to perform some additional consistency

[9] On demand forward check with dynamic variable reordering.

[10] FC with conflict directed backjumping and dynamic variable reordering.

[11] 10, 11, and 12 Queen problems. Number of checks to find *all* solutions.

[12] Prosser's Zebra problem. The number given is for finding all solutions, averaged over Prosser's original set of 450 different orderings of the problem variables [Pro93b].

[13] R1, R2, R3, and R4 are averages computed over 30 randomly generated problems drawn from Frost and Dechter's table of 50% solvable classes [FD94]. R1 consists of $N = 25$ (number of variables), $k = 3$ (domain size for each variable), $C = 89$ (number of variable pairs among the $N(N-1)/2$ possible pairs that are non-trivially constrained), and $T = 2$ (the number of incompatible value pairings among the K^2 possible pairings in each of the nontrivial constraint matrices, i.e., the tightness of each constraint). R2 is $N = 25$, $k = 6$, $C = 165$ and $T = 8$. R3 is $N = 25$, $k = 6$, $C = 65$, and $T = 16$. R4 is $N = 15$, $k = 9$, $C = 79$ and $T = 27$.

checks, but not as many as FC will. Furthermore, these additional checks performed can be noted so they do not have to be repeated. We omit further details, because they are straightforward. But the important point is that in *all* of our experiments, this version of MFCvar performed the fewest checks when compared with 24 other algorithms; see [BvR94] for details. The second best algorithm (which is very close in performance) was FC-CBJvar, forward checking with conflict directed backjumping [Pro93b] using dynamic variable reordering. It appears feasible to combine MFC with conflict directed backjumping, and with Prosser's additional BM savings [Pro93a]. The combination would be an interesting algorithm to examine.

5 Conclusion

Strictly speaking, forward checking and backtracking, or backmarking, are incomparable by worst-case complexity measures. This, coupled with the seemingly radical difference between looking forward and looking into the past, might lead to the view that no interesting formal comparison is possible.

On the contrary, there is as much similarity as there is difference. To stretch a point, we may say that forward-checking *is* backmarking augmented by a scheme to detect certain "obvious" wastes of time. The relative merits of the two depends only on whether or not the scheme pays off enough to be worth its cost, and simply knowing this may be sufficient to determine which will be better in a particular application.

Ordinary forward checking is not as efficient as it might be, and removing the inefficiency reveals its connection to backmarking even more clearly. This leads directly to the previously known idea of "minimal forward checking". Our reconstruction of MFC is interesting for two reasons, however. First, it emphasizes that MFC is a natural algorithm—a mixture of BM and FC—rather than an *ad hoc* optimization. Second, it is a good illustration of how greater understanding of the connections between various CSP algorithms can lead directly to even better techniques.

References

[ASJ85] H. Abelson, G. J. Sussman, and Sussman J. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

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

[BvR94] Fahiem Bacchus and Paul van Run. Dynamic variable reordering in CSPs. Submitted to this conference, 1994.

[CKT91] Peter Cheeseman, Bob Kanefsky, and Willian M. Taylor. Where the really hard problems are. In *Procceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 331–337, 1991.

[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.

[FD94] Daniel Frost and Rina Dechter. In search of the best constraint satisfaction search. In *Proceedings of the AAAI National Conference*, pages 301–306, 1994.

[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.

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

[Kon94] G. Kondrak. A theoretical evaluation of selected backtracking algorithms. Master's thesis, Dept. of Computer Science, University of Alberta, Edmonton, Alberta, Canada, 1994. Technical Report TR94-10.

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

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

[Pro93a] P. Prosser. Forward checking with backmarking. Technical Report Technical Report AISL-48-93, University of Strathclyde, Glasgow G1 1XH, Scotland, U.K., 1993. Available via anonymous ftp from ftp://ftp.cs.strath.ac.uk/research-reports/aisl-48-93.ps.Z.

[Pro93b] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3), 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.Z.

[ZE89] Monte Zweben and Megan Eskey. Constraint satisfaction with delayed evaluation. In *Procceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 875–880, 1989.