Dynamic Variable Ordering In CSPs*

Fahiem Bacchus[1] and Paul van Run[2]

[1] Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1,
(fbacchus@logos.uwaterloo.ca)
[2] ID-Direct

Abstract. We investigate the *dynamic variable ordering* (DVO) technique commonly used in conjunction with tree-search algorithms for solving constraint satisfaction problems. We first provide an implementation methodology for adding DVO to an arbitrary tree-search algorithm. Our methodology is applicable to a wide range of algorithms including those that maintain complicated information about the search history, like backmarking. We then investigate the popular reordering heuristic of next instantiating the variable with the minimum remaining values (MRV). We prove some interesting theorems about the MRV heuristic which demonstrate that if one wants to use the MRV heuristic one may as well use it with forward checking. Finally, we investigate the empirical performance of 12 different algorithms with and without DVO. Our experiments and theoretical results demonstrate that forward checking equipped with dynamic variable ordering is a very good algorithm for solving CSPs.

1 Introduction

Despite being in the class of NP-complete problems, many practical constraint satisfaction problems (CSPs) can be solved by tree-search algorithms that are derivatives or variants of backtracking. Naive backtracking (BT) is not a particularly useful method as the class of problems that it can solve within practical resource limits is small. A number of improvements to BT have been developed in the literature. None of these improvements can, of course, escape from behaving badly on certain inputs, given that NP $\neq$ P. Nevertheless, these improvements are often able to solve wider, or perhaps different, classes of problems.

Tree-search algorithms try to construct a solution to a CSP by sequentially instantiating the variables of the problem. The order in which the algorithm instantiates the variables is known to have a potentially profound effect on its efficiency, and various heuristics have been investigated for choosing good orderings. *Dynamic variable ordering* (DVO), is an even more effective technique [Pur83]. With DVO the order in which the variables are instantiated during tree-search can vary from branch to branch in the search tree. In this paper we will study in more detail dynamic variable ordering, in the context of binary constraint satisfaction problems.

* This paper appears in *Principles and Practice of Constraint Programming* (CP-95), pages 258–275, 1995. *Lecture Notes in Artificial Intelligence* #976, Springer Verlag. The research reported on here was supported by the Canadian Government through their IRIS project and NSERC programs.

The first issue we address is how the various CSP tree-search algorithms can be modified so as to utilize dynamic variable ordering. It is not immediately obvious how to do this, as some of these algorithms, e.g., backmarking (BM), maintain complicated information about the search history. How can the integrity of this information be maintained when the variable ordering is changing from path to path? Our first contribution is to describe a systematic methodology for modifying a tree-search algorithm so that it can use DVO. This methodology is described in Section 2, where we also present a brief description of the components of a CSP.

Given an algorithm capable of using DVO, the next question is how do we dynamically select the variable ordering. In Section 3 we examine the most popular DVO heuristic: instantiate next the variable that has the fewest values compatible with the previous instantiations. We call this heuristic the minimum remaining values heuristic (MRV). It turns out that MRV has some very strong properties, and we prove some key results about MRV in this section.

Our theoretical examination of the MRV heuristic does not tell the entire story, so we turn next to experimental evaluations to gather more information. In Section 4, we show the results of some of the experiments we have run using 12 different CSP algorithms with and without DVO using the MRV heuristic. Our experiments allow us to make a number of additional observations about the effectiveness of these algorithms and the effect of DVO using the MRV heuristic. Finally, we close with a summary of the main conclusions arising from our study.

2 A methodology for adding DVO

We assume some familiarity with CSPs in this paper (see, e.g., [Mac87]), but the basics can be easily described. Here we give a description of CSPs that is designed to make our subsequent description of the mechanics of certain CSP algorithms easier.

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 solutions, or to find just one. Formally:

Definition 1. A binary constraint satisfaction problem ¶ consists of:

- A finite collection of N *variables*, $V[1], \ldots, V[N]$.
- For each variable $V[i]$, a finite domain $D[1]$ containing $\#D[i]$ *values*: $D[i] = \{v[i,1], v[i,2], \ldots, v[i,\#D[i]]\}$.
- For each pair of variables $\{V[i], V[j]\}$, a *constraint* $C[i,j]$ between $V[i]$ and $V[j]$ which is simply a subset (not necessarily proper) of $D[i] \times D[j]$. The constraints must be symmetric, i.e., $(x,y) \in C[i,j]$ iff $(y,x) \in C[j,i]$.

The *assignment* of the value $v \in D[i]$ to the variable $V[i]$ is written $V[i] \leftarrow v$. If $(v, v') \in C[i,j]$ we say that the pair of assignments $\{V[i] \leftarrow v, V[j] \leftarrow v'\}$ is *consistent*. A *partial solution* for ¶ is a set of assignments such that every pair in the set is consistent. A *solution* to ¶ is a partial solution that contains an assignment for every variable. ∎

Tree-search algorithms generate search trees during their operation. The search tree consists of nodes which we take in this paper to be an assignment of a particular value to a particular variable. Each node in the search tree has an associated depth. Tree-search algorithms explore paths in the search tree. At any point during search the current path consists of a series of nodes from the root to the current node, and a set of as yet uninstantiated variables. We use the following notational conventions:

- $Nd[i]$ denotes the node at depth i in the current path.
- $Nd[i].Var$ denotes the *index* of the variable assigned at node $Nd[i]$. Hence, $V[Nd[i].Var]$ is the variable itself.[3]
- $Nd[i].val$ denotes the value assigned to the variable associated with node $Nd[i]$. It should be a member of this variable's domain; i.e., we should have $Nd[i].val \in D[Nd[i].Var]$.
- $Incon(Nd[i], Nd[j])$ denotes that the assignments associated with the i'th and j'th nodes are inconsistent. This corresponds to the condition:

$$(Nd[i].val, Nd[j].val) \notin C[Nd[i].Var, Nd[j].Var]$$

- Finally, we consider $Nd.Var$ and $Nd.val$ to be fields that can be assigned to.

With this notation we can now describe our methodology for adding dynamic variable ordering to any particular tree-search algorithm quite easily. The methodology is very simple. All that is required is to maintain a clear distinction between the *nodes* of the search tree and the *variables*. Both the nodes of the current path $Nd[i]$ and the variables $V[j]$ are indexed sets. When a static ordering is used there is a constant mapping between the depth of a node and the index of the variable that is assigned at that node. This often leads to a blurring of the distinction between the node index (the node's depth) and the variable index. With DVO there is no longer a static mapping between these two indices, and a failure to distinguish between them leads to confusion.

Once this distinction is maintained it becomes relatively straightforward to modify a tree-search algorithm to allow it to use DVO. One need only be careful about whether or not each step of the algorithm is indexing a node or a variable. A few examples should make the mechanics of the methodology clear.

First, consider simple backtracking (BT). In BT we assign a variable at the current *node*, selecting the variable to be assigned from the current set of uninstantiated variables. We then test the consistency of this assignment against the assignments of the previous *nodes*. This occurs with both static and dynamic variable orderings, but with a static order one might be tempted to state this as "test the assignment of the current *variable* against the assignments of the previous *variables*", a notion that does not apply in the dynamic case.

Second, consider forward checking (FC). In FC we do not check the assignment of the current node against the previous nodes, rather we use it to prune the domains of the as yet uninstantiated *variables*. FC needs to maintain information about when domain values are pruned, so that it can correctly restore those values when backtracking. Our

[3] We let $Nd[i].Var$ be an index instead of a variable so that it can also index into the variable's domain $D[Nd[i].Var]$ and the variable's domain size $\#D[Nd[i].Var]$.

methodology mandates that we be careful about the difference between nodes and variables. Clearly, the domain values are associated with the variables, but the pruning of the values arises from particular nodes (specifically, from the assignment made at those nodes).

A common way of implementing FC is to associate with each variable an array of flags, one for each domain value. Let `Domain` be a two-dimensional array such that `Domain`$[i,k]$ contains the flag for the k-th value of the i-th variable. If this flag is set to 0 it indicates that this value has not yet been pruned, while if the flag is set to some number j it indicates that this value has been pruned by the *node* at depth j. Thus the `Domain` array is indexed by a variable-value pair, but it stores a node index. When we change the assignment associated with the node at level j, we can scan the `Domain` array restoring all variable values that were pruned by that node. In the context of a static ordering, the `Domain`$[i,k]$ array is commonly described as storing the *variable* that pruned the k-th value of the i-th variable.

One of the most powerful demonstrations of the simplicity of our methodology comes from its application to Gashnig's backmarking (BM) algorithm [Gas77]. BM is one of the more complex tree-search algorithms, and implementing DVO with BM seems to be difficult because of all of the bookkeeping information that BM maintains. In fact, some authors have mistakenly claimed that BM and dynamic variable ordering are incompatible [Pro93].[4]

BM is standardly described in a manner that confuses between nodes and variables. A standard description that relies implicitly on a static ordering is as follows. When an assignment $V[i] \leftarrow k$ of $V[i]$ is made, BT checks the consistency of this assignment against all of the previous assignments $\{V[1] \leftarrow s_1, \ldots, V[i-1] \leftarrow s_{i-1}\}$, where s_i are the values assigned in the current path. If any of these consistency checks fail, BM takes the additional step of remembering the first point of failure in an array `Mcl`$[i,k]$ (the maximum check level array). This information is used to save subsequent consistency checks. Say that later we backtrack from $V[i]$ up to $V[j]$, 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 $V[i] \leftarrow k$. The assignments $\{V[1] \leftarrow s_1, \ldots, V[j-1] \leftarrow s_{j-1}\}$ have not changed since we last tried $V[i] \leftarrow k$, so there is no point in repeating these checks. Furthermore, if $V[i] \leftarrow k$ 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, `Mcl` is by itself insufficient. We also need to know how far up the tree we have backtracked since last trying to instantiate $V[i]$, so that we know what checks we can save given the information in `Mcl`. Hence, BM uses an additional array `Mbl` (the 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, `Mbl`$[i]$ will store the information that we have only backtracked to $V[j]$ since last visiting $V[i]$).

To apply our methodology we must distinguish between nodes and variables. Clearly, `Mcl`$[i,k]$ is not the deepest variable, but rather the deepest node that the assignment $V[i] \leftarrow k$ checked against. And similarly, `Mbl`$[i]$ is the shallowest node whose assign-

[4] It is worth noting that Haralick and Elliot [HE80] had in fact implemented DVO with BM and reported on its empirical performance. However, they never described the manner in which BM was altered to accommodate DVO.

ment has been changed since the i-th variable was the variable associated with the current node. So `Mcl` and `Mbl` are both indexed by variable indices, but store node indices. This distinction allows us implement a correct version of BM that can use dynamic variable ordering. Figure 1 gives the resulting algorithm in more detail. To our knowledge, a DVO version of BM has not previously appeared in the literature.

```
procedure BM(i)
%Tries to instantiate node at level i
%Then recurses
    Cur ← NextVar()
    Nd[i].Var ← Cur
    for k = 1 to #D[Cur]
        Nd[i].val ← v[Cur, k]
        if Mcl[Cur, k] ≥ Mbl[Cur] then
            ok ← true
            for j = Mbl[i] to i − 1 and while ok
                Mcl[Cur, k] ← j
                if Incon(Nd[i], Nd[j],) then
                    ok ← false
            if ok then
                V[Cur] ← v[Cur, k]
                if i = N then
                    print V[1], . . . , V[N]
                else
                    BM(i + 1)
    Mbl[Cur] ← i−1
    Restore(i)
```

```
procedure Restore(i)
%Updates Mbl
    for j = 1 to N
        if Mbl[j] = i then
            Mbl[j] ← i−1
```

Fig. 1. Backmarking with dynamic variable ordering

In the code the procedure *NextVar*() returns the index of the next variable to be assigned. It computes which of the uninstantiated variables should be assigned next by some heuristic measure. Backmarking is invoked, after initialization, with the call BM(1) and it will subsequently print all solutions. The only differences between this code and the standard static order BM algorithm are (1) a dynamic choice of the variable to instantiate next, (2) the indirection of consistency checks through the node structures, and (3) the restore function must scan all variables, not just $V[i]$ to $V[N]$ (these may not be the uninstantiated variables at this point in the search tree).

A large number of tree-search algorithms have been implemented by Manchak and van Beek as a library of C routines [MvB94]. These include the traditional algorithms of backtracking (BT) [BE75], backjumping (BJ) [Gas78], backmarking (BM) [Gas77], and forward checking (FC) [HE80]. They have also implemented all of Prosser's hybrid algorithms [Pro93], conflict-directed backjumping (CBJ), backmarking with backjumping (BMJ), backmarking with conflict-directed backjumping (BM-CBJ), forward checking with backjumping (FC-BJ), and forward checking with conflict-directed backjumping (FC-CBJ); Dechter's graph-based backjumping (GBJ) [Dec90]; forward checking that uses full arc consistency to prune the future domains (FCarc); and forward

checking that uses full path consistency to prune the future domains (FCpath).

We have applied our methodology to all of these algorithms, implementing a DVO version of each. In Section 4 we will report on the performance of these algorithms with and without DVO.

3 The MRV Heuristic

In the previous section we demonstrated how a tree-search algorithm can be modified so as to use DVO. It has long been known that DVO can yield a considerable performance speedup [HE80]. However, the notion of using DVO is quite distinct from the method by which the ordering is chosen. In previous work attention has focused almost exclusively on the heuristic of choosing at every point in the search, the next variable to be that which has the minimal number of remaining values, i.e., the minimal number of values compatible with all of the previous assignments [HE80, Pur83, DM94]. We call this the minimum remaining values heuristic (MRV). Previous authors have called this heuristic dynamic variable ordering [FD94], and dynamic search rearrangement [Pur83]. But such names tend to confuse the specific heuristic MRV with the general technique DVO. As we will see in this section, MRV has some very particular properties that might not be shared by other ordering heuristics.

An immediate question is how do we implement MRV in a domain independent manner? First, we should note that for algorithms that do all of their constraint checks backwards against previous assignments, computing the MRV heuristic will require additional constraint checks. If all we have access to is the set of constraint relations, then there is an obvious but inefficient strategy. The tree-search algorithm calls the MRV procedure when it has descended to the next level and needs to choose a variable to assign at this level.[5] When it is invoked, the MRV procedure can check each value in the domain of each uninstantiated variable to see if that value is consistent with the current set of assignments. By cycling through all the values in the variable domains it can count the number of remaining compatible values each variable has, and return a variable with the minimum such number.

However, as pointed out by Dechter and Meiri [DM94], with this implementation computing the MRV heuristic will consume many redundant constraint checks. For example, consider the following scenario. We make the assignment $V[1] \leftarrow 1$ at $Nd[1]$. Then we call MRV and it discovers that $V[3] \leftarrow 1$ is incompatible with this assignment, but it also finds that $V[2]$ is the variable with fewest remaining values. The tree-search algorithm then makes the assignment $V[2] \leftarrow 1$ at $Nd[2]$. Upon calling MRV again, our naive strategy will again check $V[3] \leftarrow 1$ against $V[1] \leftarrow 1$, even though it had already determined at a previous level that this pair was incompatible. This constraint check is redundant.

A superior implementation strategy for MRV is to use the same procedure as FC. Basically, we allow the MRV procedure to maintain a set of pruned domains for all the uninstantiated variables. Whenever the tree-search algorithm instantiates a variable, it can call MRV to allow it to prune the remaining variable domains using the new

[5] For example, in the BM code of Figure 1, the call to *NextVar*() is a call to a procedure that computes the MRV heuristic.

instantiation. And whenever the tree-search algorithm uninstantiates a variable it can call MRV to allow it to restore the values pruned by that instantiation. Now the MRV procedure can simply count the number of remaining elements in each of the pruned domains to determine the next variable. In the example above, when we make the assignment $V[1] \leftarrow 1$, MRV will prune the value 1 from the domain of $V[3]$. Hence, when we make the next assignment $V[2] \leftarrow 1$, MRV does not have to do any checks against $V[3] \leftarrow 1$: that value has been pruned from $V[3]$'s domain. Note that the tree-search algorithm is still using MRV as a heuristic. In particular, the structures containing the pruned domains remain local to the MRV procedure. All the MRV procedure does is to return the next variable, it *communicates no other information* to the tree-search algorithm. Let us call the MRV heuristic computed using the forward checking procedure MRV_{FC}.

In FC algorithms, on the other hand, the pruned domains are maintained as part of the algorithm itself. For these algorithms, the MRV procedure does not need to do any extra constraint checks. It can simply count the number of values in the pruned domains, and rely on the FC algorithm to maintain these structures.

Since MRV_{FC} is computed using the same procedure used by forward checking it should not be too surprising that there is a close relationship between tree-search algorithms that use MRV_{FC} and FC itself. This relation is characterized by the following theorems. Let *checks*(A) be the number of constraint checks performed by algorithm A. This is actually a function from problem instances to number of constraint checks. When we write *checks*$(A) <$ *checks*(B) we mean that the "$<$" relation holds for *all* problem instances, similarly for the other binary relations.

Theorem 2. *MRV makes standard backjumping redundant.*[6] *Specifically, we have*

$$\textit{checks}(\text{BT+MRV}) = \textit{checks}(\text{BJ+MRV}), \textit{ and}$$
$$\textit{checks}(\text{BM+MRV}) = \textit{checks}(\text{BMJ+MRV}).$$

Proof. Note that this theorem holds for the more general MRV, not only for MRV_{FC}. Backjumping operates when search hits a dead end. Say that the variable associated with the dead end node is $V[j]$. Instead of stepping back to the previous node, backjumping jumps back to the deepest previous node, say node i, that had a conflict with some value of $V[j]$; i.e., none of the nodes deeper than i had any conflicts with $V[j]$. This means that if the tree-search algorithm is using MRV, at the time that node i's assignment was made, $V[j]$ must have had zero remaining compatible domain values. Hence, it must have been chosen by MRV to be the next variable,[7] and the dead end must in fact have occured at level $i + 1$. That is, the backjump must have been simply a chronological backstep, and the search process without backjumping would have been identical. ∎

[6] This result is very closely related to (and, in fact, can be viewed as being a corollary of) Kondrak's result that FC visits fewer nodes than BJ [Kon94]. To see the similarly note that in our proof FC would have detected a domain wipeout at node i and would never had descended below that node.

[7] If there was more than one variable with zero remaining domain values, MRV would still have chosen $V[j]$. Otherwise $V[j]$ would not have been the variable associated with the dead end node.

This theorem does not hold for more sophisticated forms of backjumping such as CBJ, or GBJ. CBJ and GBJ do not perform standard backjumping. Rather they use a backjumping technique that passes information further up the tree allowing multiple backjumps. Prosser's FC-BJ algorithm [Pro93] is presented as a combination of FC and standard BJ, and one might suspect that the above theorem implies that *checks*(FC) = *checks*(FC-BJ). However, the backjumping in FC-BJ is not standard backjumping. Rather, it can be considered to be a "one-step" version of CBJ. In particular, it uses information gathered from a future variable (the variable whose domain has wiped out), plus information from the current node to decide where to jump back to. Standard backjumping only uses information from the current node in determining where to jump back to. Hence, the theorem does not apply to FC-BJ either.

Theorem 3. *The number of consistency checks performed by each of the algorithms* BT+MRV$_{\mathrm{FC}}$, BJ+MRV$_{\mathrm{FC}}$, BM+MRV$_{\mathrm{FC}}$, *and* BMJ+MRV$_{\mathrm{FC}}$, *lie in the range*

$$[\textit{checks}(\mathrm{FC{+}MRV}), 2KN\textit{checks}(\mathrm{FC{+}MRV})],$$

where N is the number of variables and K is the maximum of the variable domain sizes, and assuming that the MRV heuristics break ties in the same way,

Proof. To conserve space we provide only parts of the proof. Also in our discussion we drop the +MRV$_{\mathrm{FC}}$ suffix on the algorithm names.

The key to understanding this theorem is to realize that the MRV heuristic acts much like the domain wipeout (DWO) detection that FC performs. Recall that FC checks forward from the current node and backtracks immediately if it detects that some future variable has no remaining consistent values. Such a variable is said to have had a domain wipeout. Now consider the search tree explored by FC. All leaf nodes are either solutions or nodes where DWO was detected. At these DWO nodes any algorithm equipped with MRV reordering would at the next level try to assign a value to a variable with zero remaining compatible values. Hence, MRV will allow the algorithm to fail at the very next level anyway.

BT and BM visit all of the nodes visited by FC [Kon94], and in the same order as FC. (The ordering follows from our requirement that ties are broken identically). The above argument can be made sufficiently precise to show that besides these nodes, BT and BM (with MRV) visit at most $K-1$ additional siblings of each of these nodes (the assignments that were pruned by FC), and at the leaf nodes of FC's search tree they can descend at most one level deeper, visiting at most K additional children nodes. Hence, we can identify three different types of nodes visited by BT and BM: (1) nodes also visited by FC, (2) children nodes of a FC leaf node, and (3) siblings of FC nodes not visited by FC.

With this in hand all we need to do is to examine the number of checks that BT and BM perform at each type of node. Consider BT. For a node also visited by FC, it must check that node's assignment against all previous assignments, consuming at most N checks. These checks must succeed (else FC would have pruned this value and would never have visited this node). Hence, BT must descend to the next level and it will invoke the MRV$_{\mathrm{FC}}$ procedure. This procedure consumes the same number of checks as FC's forward checking phase. FC, on the other hand, consumes no checks

to determine the node's consistency with the previous assignments (all of these checks were performed earlier), and then it performs forward checking, consuming the same number of checks as BT's invocation of MRV_{FC}. Hence, for these nodes BT performs at most N more checks. For nodes that are children nodes of a FC leaf node, BT performs at most N more checks. These checks must fail as the variable at this level (selected by MRV) has no values compatible with the previous assignments. Every leaf node in FC's search tree can have at most K children, so BT can consume at most NK additional checks in searching the children of FC leaf nodes. Note that because these children nodes fail against previous assignments, BT does not need to compute the MRV heuristic at these nodes. Finally, for nodes that are siblings of FC nodes not visited by FC, BT will again perform up to N additional checks, and again will not need to compute the MRV heuristic since some of these checks must fail. Each node in the FC search tree can have at most $K - 1$ unvisited siblings. Putting this all together, we see that the worst case are the FC leaf nodes. There BT can perform N more checks at the node itself, NK more checks visiting its children, and $N(K - 1)$ more checks visiting its unvisited siblings. This gives the $2NK$ upper bound of the theorem. At no node does BT perform fewer checks than FC (due to its MRV computation). This gives the lower bound.

The result for BM follows the same argument. BM will save a number of checks over BT, so for BM the upper bound becomes looser. Nevertheless, the lower bound is clearly the same. The result for BMJ and BJ follow from Theorem 2. ∎

Intuitively what is occuring here, is that in computing the MRV heuristic these algorithms are forced to consume at least as many checks as FC. Plain BT may in certain cases perform fewer checks than FC, but once we add MRV_{FC} this is no longer possible. Furthermore, MRV_{FC} has the property of promoting future failures to the next variable. This means that with MRV, algorithms like BT cannot consume an exponential amount of extra checks prior to reaching the level at which a domain wipeout occurs. Plain BT, on the other hand, often falls prey to this behavior. MRV makes the BT, BM, and BJ search spaces very similar to that of FC+MRV.

Two obvious questions arise. First, since computing the MRV heuristic causes BT and its variants to consume as many checks as FC, is it worthwhile? And second, why have we placed such strong restrictions on the information that the MRV computation can communicate to the tree-search algorithm?

The first question can only be answered experimentally. DVO using MRV is a heuristic technique, and there is no guarantee its costs are worthwhile. Nevertheless, the experimental evidence is unequivocal, DVO using MRV generates net speedups of can be orders of magnitude in size. Significant performance gains occurred in all of the experiments we performed, and such gains confirmed the results found in previous work [HE80, Pur83, DM94].

As for the second question, indeed other information could be returned from the MRV computation. For example, the MRV procedure could tell the algorithm about pruned domain values, or about domain wipeout. However, if one follows this strategy and starts to share more and more information between the MRV procedure and the tree-search algorithm, one is lead naturally to a combined algorithm that is identical to FC! In fact, our theorem shows that if one wants to do DVO with MRV, one may as

well adopt the FC algorithm from the onset. Unless, of course, one has a much more efficient method of implementing MRV than MRV_{FC}.[8]

Theorem 3 does not cover the algorithms that perform more sophisticated backjumping. One result that can be proved for these algorithms is the following (the proof is omitted due to space considerations):

Theorem 4. *The number of consistency checks performed by the algorithms* CBJ + MRV_{FC}, BM-CBJ+MRV_{FC} *lie in the range*

$$[\textit{checks}(\text{FC-CBJ+MRV}), 2KN\textit{checks}(\text{FC-CBJ+MRV})]$$

assuming that the MRV heuristics break ties in the same way, and that in the case of DWO the MRV_{FC} *heuristics return as the next variable the first variable at which* FC-CBJ *detected DWO.*

4 Experiments

Our theoretical results give some valuable insight into the relative performance of various CSP algorithms when equipped with DVO using the MRV heuristic. However, they do not answer all of the questions. So we turn now to the results of a series of experiments we ran using the 12 different algorithms mentioned in Section 2 (BT, BM, BJ, FC, CBJ, BMJ, BM-CBJ, FC-BJ, FC-CBJ, FCarc, FCpath, and GBJ). We experimented with these algorithms using both static and dynamic variable orderings, for a total of 24 different algorithms tested. The DVO equipped algorithms are all indicated with a "var" suffix: BTvar, BMvar, etc. The MRV heuristic was used throughout, with the non-FC algorithms using MRV_{FC}. In our tests we included the number of constraint checks consumed by the MRV computation.

Table 1 shows some of our experimental results. The columns "Z-1st"(Zebra first solution) and "Z-all" (Zebra all solutions) are tests run on Prosser's version of the Zebra problem [Pro93] (which has 11 solutions). Prosser used 450 different static variable orderings, of varying bandwidths, and computed the average number of consistency checks required to find the first solution. We used the same orderings and repeated Prosser's original tests, extending them to test more algorithms and to test finding all solutions. We also used these tests as a "sanity" check on our implementation: we checked that we obtained exactly the same results as Prosser on the tests he performed, and we checked the correctness of every solution found during all of the runs of each algorithm.[9]

[8] It seems difficult to make significant improvements over MRV_{FC} for a general, domain independent, MRV computation. Furthermore, if any such improvements can be made they probably could be applied to improve FC as well. However, more efficient MRV computation might be possible in specific domains,. One domain that springs to mind is n-Queens.

[9] Of course, the algorithms using DVO will not show much variance when run using different initial orderings (the difference are due to our method of tie-breaking which depends on the initial order). Nevertheless, taking the average over all tests allows for a more reasonable comparison.

Algorithm	Z-1st	Z-all	Q-1st	Q-all	R1	R2	R3	R4
FC-CBJvar	0.5023	2.921	812.5	30206	2.268	15.06	10704	1152
FC-BJvar	0.5029	2.925	816.8	30216	2.318	15.08	10952	1159
FCvar	0.5037	2.928	817.0	30225	2.318	15.08	10954	1159
BM-CBJvar	0.8633	5.328	1298	(12)	3.127	21.50	13240	1460
BMvar	0.8673	5.357	1304	(10)	3.209	21.54	13452	1467
BMJvar	0.8673	5.357	1304	(10)	3.209	21.54	13452	1467
CBJvar	1.078	7.823	13060	(15)	3.297	26.84	17667	1911
GBJvar	1.078	7.854	13500	(16)	3.413	26.98	18262	1947
BJvar	1.095	7.933	13500	(17)	3.415	27.00	18290	1947
BTvar	1.095	7.933	13500	(17)	3.415	27.00	18290	1947
FCarcvar	1.624	5.786	13871	(19)	5.954	59.38		4082
FCarc	5.581	36.16	(18)	(20)	8.517	84.97		12005
FC-CBJ	10.36	69.98	(15)	39671	12.32	70.01		
FC-BJ	16.84	101.8	(16)	39742	19.54	71.97		
BM-CBJ	25.47	160.1	(14)	35456	17.64	76.02		
FC	35.58	182.0	(17)	40021	26.48	77.23		
CBJ	63.21	397.3	(19)	(13)	31.59	284.2		
BMJ	125.5	557.2	(13)	35507	47.60	89.30		
BM	396.9	1608	(12)	34944	64.38	114.9		
BJ	503.3	2226	(20)	(14)	143.3	410.8		
FCpathvar	644.7	1472	(23)	(23)	630.3	17482.3		
GBJ	713.6	2889	(21)	(21)	382.6	876.1		
FCpath	844.5	3183	(24)	(24)	634.5	17482.7		
BT	3859	16196	(22)	(22)	415.1	942.4		

Table 1. Number of consistency checks (thousands) for various experiments

The columns "Q-1st" and "Q-all" are results from the n-Queens problem, for finding the first and all solutions respectively. The "Q-1st" column shows the total number of checks required to find the first solution for n-Queens summed over all n-Queen problems of size 2–50.[10] Not all of the algorithms were able to solve up to 50-Queens (within a limit of 40,000,000 checks). For these algorithms we show their relative ranking in brackets. For example, BM was the best of the "failed" algorithms for finding the first solution (its rank was (12) and 11 algorithms were able to solve all problems).[11] For finding all solutions we ranged over n-Queen problems of size 2–13.

The columns labeled "R1" to "R4" show averages over 30 random problem instances. We drew our problems from the range of 50% solvable classes given by Frost

[10] We took the sum over all the problems to smooth the variance that the static order algorithms display. The static order algorithms find problems with even n more difficult than the subsequent problem of size $n + 1$. The DVO algorithms do not display this behavior, but their variance is also quite high.

[11] The rank of the algorithms that could not solve all problems was computed by first ranking by the largest problem they were able to solve, and then ranking by the lowest number of checks performed.

and Dechter [FD94]. These problem classes are defined by four parameters: N the number of variables; K the number of values in each variable's domain; T the number of incompatible value pairs in each non-trivial constraint (i.e., the tightness of each constraint); and C the number of non-trivial constraints. Our random problem generator uses the same distribution as that described by Frost and Dechter. We ran other random tests, but the experiments reported here have the following parameters: for R1, $N = 25$, $K = 3$, $T = 1$, and $C = 199$; for R2, $N = 15$, $K = 9$, $T = 27$, and $C = 79$; for R3, $N = 50$, $K = 6$, $T = 4$, and $C = 71$; and for R4, $N = 35$, $K = 6$, $T = 4$, and $C = 500$. Blank entries indicate that the algorithm was unable to solve the problems within a limit of 40,000,000 checks.

Discussion. The most important result of our tests is that the three algorithms FC-CBJvar, FC-BJvar, and FCvar prove themselves to one-two-three in all of the tests we ran (including other random tests not reported here). FC-CBJvar seems to be the fastest algorithm among those tested, and we know of only one other algorithm (an "on-demand" version of FC recently developed by Bacchus and Grove [BG95]) that beats it. However, the performance gain obtained by adding conflict directed backjumping hardly seems worthwhile (in the tests we performed). A 5% gain over FCvar was the largest we found in any of our tests (the table shows at most a 2% gain). Given the simplicity of FCvar (it is FC with a simple domain counting implementation of MRV) it would seem that this is the algorithm of choice.

In fact, a plausible argument can be given as to why adding CBJ to an algorithm that already uses MRV is unlikely to yield much improvement. With MRV, variables that have conflicts with past assignments (i.e., that have values eliminated from their domain by these assignments) are likely to be instantiated sooner. Thus, MRV will tend to cluster conflicted variables together. Hence, CBJ is unlikely to generate large backjumps, and its savings are likely to be minimal.[12] In essence what is happening with plain backjumping (the table results for BTvar, BJvar and BMvar, BMJvar are as predicted by Theorem 2) is also happening with CBJ, but to a lesser extent.

Our results also show that DVO using MRV is generally a great win. The only case where static order algorithms beat their DVO counterparts is in the case of finding all solutions to n-Queens. There, BM was the fourth best algorithm. Nevertheless, it was still beaten by FCvar by about 15%. In the other tests the DVO algorithms display much greater performance gains over their static order counterparts.

These performance gains show that the forward checking algorithm is a very powerful CSP solution technique. Frost and Dechter suggest, in a recent paper [FD94], that "FC should be recognized as a more valuable variable ordering heuristic than as a powerful algorithm". This claim misses the important features of FC. Our results show that plain FC is a very good algorithm, but not always the best static order algorithm. In n-Queens, for example, it is beat by BM. However, when equipped with dynamic variable ordering it is clearly superior (empirically it beats all of the others except for its

[12] Prosser (personal communication) has pointed out that CBJ might still yield significant benefits when the variables have very different domain sizes (as the MRV heuristic may then be fooled by this initial difference in domain sizes). In all of the tests we presented here the variables have identical domain sizes. We hope to do some additional tests in the future to test this conjecture.

own variants, and our theoretical results also show its superiority to some of the other algorithms). So in one sense the above claim is correct: FC with dynamic variable ordering is more valuable than plain FC. But the main feature of FC is that it is supports a check-free implementation of MRV dynamic ordering.[13]

There is one final point that is worth making here. It has often been claimed that static order FC performs the "optimal" amount of forward checking [Nad89]. Our results dispute this claim. Static order FCarc (which uses full arc consistency to prune the future domains) often beats static order FC, in the Zebra and in many of the random tests. However, with the extra edge provided by DVO, FC once again seems to be the "right amount" of forward checking. FCpath, on the other hand, is definitely doing too much forward checking in all of these tests.

In conclusion, in this paper we have (1) provided a methodology for implementing dynamic variable ordering that can be applied to a range of CSP tree-search algorithms; (2) have implemented a range of DVO algorithms and tested them empirically; and (3) have provided some useful theoretical insights into the popular MRV heuristic. The main practical lesson of our study is that forward checking with variable reordering is both empirically and theoretically a very good algorithm for CSPs. Its main limitation seems to be its application to very large but "easy" problems (i.e., problems were there are an exponential number of solutions). For large problems the forward checking phase of FC can be quite expensive. Guided "random" guesses in the solution space, e.g., the GSAT technique [SLM92], seems to be a superior method for such problems. However, FC with variable reordering (or some variant of FCvar) may well be the best algorithm for small "hard" problems (i.e., problems with fewer variables and values that have only a few solutions scattered in an exponentially sized search space).

References

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

[BG95] Fahiem Bacchus and Adam Grove. On the Forward Checking algorithm. Submitted to this conference, 1995.

[Dec90] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[DM94] R. Dechter and Itay Meiri. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence*, 68:211–241, 1994.

[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.

[13] Frost and Dechter report in [FD94] on some experiments showing inferior performance by FCvar, but they suspect an error in their code (personal communication).

[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.

[MvB94] D. Manchak and P. van Beek. A 'C' library of constraint satisfaction techniques, 1994. Available by anonymous ftp from: ftp.cs.ualberta.ca:pub/ai/csp.

[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.

[Pur83] P. W. Jr. Purdom. Jr. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, 21:117–133, 1983.

[SLM92] B. Selman, H. J. Levesque, and D. G. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the AAAI National Conference*, pages 440–446, 1992.

This article was processed using the LaTeX macro package with LLNCS style