Forward Checking with Backmarking

Technical Report AISL-48-93
June 1993

Patrick Prosser
Department of Computer Science
University of Strathclyde
Glasgow G1 1XH
Scotland
pat@cs.strath.ac.uk

Abstract

The forward checking routine (FC) of Haralick and Elliott attempts to encourage early failures within the search tree of constraint satisfaction problems, leading to a reduction in nodes visited, which tends to result in reduced search effort. In contrast, Gaschnig's backmarking routine (BM) attempts to avoid performing redundant consistency checks. These two algorithms are combined to give us FC-BM, an algorithm that attempts to minimise the number of nodes visited, while avoiding redundant consistency checks. This algorithm is further enhanced such that it incorporates conflict-directed backjumping (CBJ) to give us FC-BM-CBJ. A series of experiments are then carried out on *really hard* problems in an attempt to position these new algorithms with respect to the known algorithms.

1 Introduction

In the binary constraint satisfaction problem we are given a set of variables, where each variable has a domain of values, and a set of constraints, where a constraint acts between a pair of variables. The problem is then to find an assignment of values to variables, from their respective domains, such that the constraints are satisfied [Dechter 1992, Kumar 1992, Mackworth 1992, Meseguer 1989]. There are many instances of this problem that are of practical value, for example constraint satisfaction scheduling problems [Muscettola 1993, Smith and Cheng 1993], and problems of design [Nadel and Lin 1991]. Typically these problems may be addressed by backtracking search, such as naive backtracking (BT) [Bitner and Reingold 1975 and Golomb and Baumert 1965], backmarking (BM) [Gaschnig 1977], backjumping (BJ) [Gaschnig 1979], forward checking (FC) [Haralick and Elliott 1980], graph-based backjumping (GBJ) [Dechter 1990], conflict-directed backjumping [Prosser 1993c], and the hybrid algorithms reported in [Prosser 1993a/c]. All of these algorithms are *complete*, in that if the csp is satisfiable they will eventually find a solution, and if the csp is over-constrained they will terminate.

Most empirical studies, so far, have considered FC to be the best algorithm of the set {BT, BM, BJ, GBJ, FC} with BM a close competitor, and some have gone so far as to suggest that a significant improvement could be brought about by combining FC with BM [Nadel 1989, and Freuder and Wallace 1992]. In this paper we present such an algorithm, namely FC-BM, and enhance it such that it also incorporates conflict-directed backjumping (FC-BM-CBJ). A series of experiments are then carried out to position these new algorithms with respect to the existing algorithms.

The algorithms are presented in unstructured English (although pseudo code is given in the appendix) but the variable names and descriptions are compatible with those in [Prosser 1993a/b/c]. We assume that we have the following global variables. We have an array of n variables, such that $v[i]$ is the ith variable. Each variable has a discrete domain of values $domain[i]$, such that $domain[i]$ has cardinality m. Each variable also has a *working* domain, namely $current-domain[i]$. When a variable is instantiated with a value $v[i] \leftarrow k$, that value k is selected from $current - domain[i]$. If that instantiation is discovered to be infeasible then k is removed from $current - domain[i]$, and when backtracking takes place from i to h (where $h < i$) $current - domain[i]$ will be reset to $domain[i]$. We have a constraint matrix C, such that $C[i, j]$ is a constraint relation between $v[i]$ and $v[j]$. If there is no constraint between $v[i]$ and $v[j]$ then $C[i, j]$ will be *nil*.

2 BackMarking (BM)

Backmarking [Gaschnig 1977] attempts to avoid the execution of redundant consistency checks. There are 2 scenarios where consistency checks can be shown to be redundant. The first is when we know that checking will fail, and the second is when we know that checking will succeed. In backmarking (BM), or in fact any backward checking algorithm, we instantiate the current variable $v[i]$ with a value k and check (backwards) against the instantiated variables (the past variables). That is, we check $v[i] \leftarrow k$ against $v[1]$. If this test succeeds we then check $v[i] \leftarrow k$ against $v[2]$, and so on to $v[i-1]$, so long as no consistency check fails. Assume that consistency checking between $v[h]$ and $v[i]$ failed (where $h < i$). We can now deduce two pieces of search knowledge. The next time that $v[i]$ becomes the current variable we may again try the instantiation $v[i] \leftarrow k$. If $v[h]$ has not been re-instantiated with a new value then we can be sure that the consistency check between $v[h]$ and $v[i] \leftarrow k$ will again fail. Therefore we do not have to attempt this instantiation. Nadel refers to this as a type (a) saving [Nadel 1989].

Conversely assume that since we last visited $v[i]$ we have backtracked to $v[g]$ (where $g < h$), and that we have reinstantiated $v[g]$ and $v[g+1]$, right up to $v[i-1]$. We can now be certain that $v[i]$ will again pass consistency check with $v[f]$, for all $f < g$, because these variables have not changed values. Therefore we need only perform consistency checks between $v[h]$ and the instantiation $v[i] \leftarrow k$, for all h, where $g \leq h < i$. Nadel calls this a type (b) saving.

These tests can be implemented by using two arrays, namely *mcl* and *mbl*. *mcl* is an n by m array. When a consistency check is performed between $v[h]$ and $v[i] \leftarrow k$ the array element $mcl[i,k]$ is set to h. That is, mcl is the 'maximum checking level' performed for a given instantiation (and initially $mcl[i,k]$ is set to zero for all i and all k). *mbl* is an n element one dimensional array, and corresponds to the 'minimum backtracking level'. $mbl[i]$ records the shallowest variable in the search tree that has been re-instantiated since we last visited $v[i]$. Initially $mbl[i]$ is set to zero, for all i. When we backtrack from $v[i]$ to $v[i-1]$ we set $v[i]$ to $i-1$, and set $mbl[j]$ to the minimum of $mbl[j]$ and $i-1$, for all j where $i < j \leq n$. Therefore, when we again attempt to instantiate $v[i] \leftarrow k$ we perform the following tests:

(a) if $mcl[i,k] < mbl[i]$ we know that the consistency check has failed in the past against some variable $v[h]$, where $h < mbl[i]$, and since this variable has not changed value the test will fail again.

(b) if $mcl[i,k] \geq mbl[i]$ we know that the instantiation has passed consistency checks with all variables $v[g]$, where $g < mbl[i]$, and that these variables have not changed values. Therefore these tests will again succeed, and we only need to test $v[h]$ against $v[i] \leftarrow k$, for all h, where $mbl[i] \leq h < i$.

3 Forward Checking (FC)

In forward checking [Haralick and Elliott 1980] we instantiate the current variable $v[i] \leftarrow k$ and then check forwards against the uninstantiated (future) variables. In doing this we remove from the current domain's of variables values which are incompatible with $v[i] \leftarrow k$. That is, we instantiate $v[i] \leftarrow k$ and check against all remaining instantiations for $v[i+1]$ (assuming of course that there is a constraint $C[i,i+1]$ between $v[i]$ and $v[i+1]$). Any incompatibilities are then removed from $current-domain[i+1]$. If there are still values remaining for $v[i+1]$ we then check forwards from $v[i]$ to $v[i+2]$ (again assuming the existence of constraint $C[i,i+2]$), and so on until we check from $v[i]$ to $v[n]$. If any forward checking from $v[i]$ to $v[j]$ (where $i < j$) removes all values from $current-domain[j]$ (ie. annihilates the current domain) then we undo the effects of forward checking and attempt a new instantiation for $v[i]$, and if there are no values remaining to be tried we then re-instantiate $v[i-1]$.

4 Forward Checking with BackMarking (FC-BM)

We can incorporate the type (a) savings of BM into FC, and we do this by making the effects of forward checking explicit. Assume that we instantiate $v[i] \leftarrow k$ and check forwards against $v[j]$. Assume that this results in the removal of incompatible values from $current-domain[j]$. We can record the fact that $v[i]$ removes values, and we do this in the array element $past-fc[j]$. That is, $past-fc[j]$ is a set of variables indices, such that if $v[i]$ checks against $v[j]$ then $i \in past-fc[j]$. We initialise $past-fc[i]$ to be $\{0\}$ for all i. That is, we assume that the pseudo variable $v[0]$ checks forwards against all other variables, and that it removes no values from the current domain's of future variables (there are a number of reasons why this is done, and these are explained later on).

Assume that when we attempt the instantiation of $v[i] \leftarrow k$ we check forwards against $v[j]$ and this results in the annihilation of $current-domain[j]$. We then uninstantiate $v[i]$ and undo the effects of forward checking from $v[i]$. We can now be sure that if we do not re-instantiate any of the variables forward checking against $v[j]$ then the instantiation $v[i] \leftarrow k$ will again annihilate $current-domain[j]$. Conversely, if h is the largest variable index in $past-fc[j]$, and any variable $v[g]$ is re-instantiated (where $g \leq h$) we might be able to re-consider the instantiation of $v[i] \leftarrow k$.

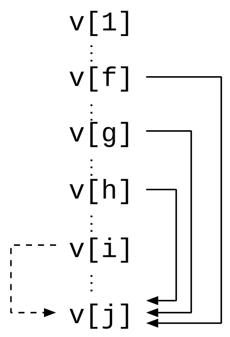

Figure 1: Forward Checking Scenario 1

Figure 1 shows a forward checking scenario that exhibits the above properties. Variable $v[f]$ has been instantiated with a value and forward checking takes place against $v[j]$, removing incompatible values from $current-domain[j]$. The instantiation of $v[g]$ also checks forwards against $v[j]$ (again removing values from $current-domain[j]$) as does the instantiation of $v[h]$ (where $f < g < h < i < j$). We then attempt an instantiation of the current variable $v[i] \leftarrow k$ and this checks forwards against $v[j]$ (the dotted line). This instantiation is incompatible with all remaining values in $current-domain[j]$ (ie. it annihilates $v[j]$). The effects of forward checking from $v[i]$ are then undone, and the array element $past-fc[j]$ will then be $\{0, f, g, h\}$. We can now be sure that the instantiation $v[i] \leftarrow k$ will be infeasible until we backtrack to (or above) $v[h]$.

We can encode this by again using the arrays mcl and mbl. In this case let mcl be initialised such that $mcl[i,k] \leftarrow i$, for all i and all k. If forward checking from the instantiation $v[i] \leftarrow k$ annihilates $current-domain[j]$ we then undo the effects of forward checking from $v[i]$ and set $mcl[i,k]$ to the maximum value in $past-fc[j]$. That is, $mcl[i,k]$ is the deepest variable that checks forwards against $v[j]$. Conversely, if $v[i] \leftarrow k$ is consistent with the current search state (or we have no evidence to believe that the instantiation is inconsistent) then we set $mcl[i,k]$ to i. $mbl[i]$ is as-before, and records the shallowest variable that has been re-instantiated since we last visited v[i] (and $mbl[i]$ is initially 0, for all i). Therefore, we have the type (a) saving for forward checking with backmarking (FC-BM):

(a) If $mcl[i,k] < mbl[i]$ then $v[i] \leftarrow k$ will annihilate the current domain of some future variable. That is, we have not re-instantiated at least one of the variables $v[h]$ that check forward against $v[j]$, such that $v[i] \leftarrow k$ no longer annihilates $current-domain[j]$. Therefore, do not consider this instantiation.

Again, we update $mbl[i]$ as in BM, but with one detailed change. Assume that we are backtracking from $v[i]$, and we are about to set $mbl[i] \leftarrow i-1$. We must now be careful, for the following reason. Assume that $mbl[i] = f$ and that we are about to set $mbl[i]$ to h, where $0 < f < h$ ($mbl[i]$ is about to increase in value from some non-zero value). This corresponds to a situation where we have previously encountered a search state where we were unable to instantiate $v[i]$ and eventually backtracking took place up to variable $v[f]$ (causing $mbl[i]$ to be set to f). The search process then advanced back down through the search tree to $v[i]$, and no compatible instantiation was found for $v[i]$. It may be the case that we have some $k \in domain[i]$ such that

(1) k has been removed from $current-domain[i]$ due to forward checking from $v[g]$ where $g < h$, and

(2) $mcl[i,k] \geq f$ implying that the instantiation $v[i] \leftarrow k$ should now be considered as a possibility, due to rule (a) above.

In the above situation, if we blindly set $mbl[i]$ to h then the next time that we visit $v[i]$ we would not consider the instantiation $v[i] \leftarrow k$ as it would appear that we had not backtracked to $v[f]$. In essence, this complication arises because when we visit $v[i]$ we might not examine all values in $domain[i]$, because some of these values may have been disallowed due to forward checking. Consequently, the information in $mcl[i,k]$ may be out of date. Therefore, when we backtrack from $v[i]$ we must analyse $mcl[i,k]$, for all k, before we set $mbl[i]$ to $i-1$. That is,

If $0 < mbl[i] < i-1$ then for all $k \in domain[i]$, if $mcl[i,k] \geq mbl[i]$ then set $mbl[i,k]$ to i

This will allow us to reconsider the instantiation $v[i] \leftarrow k$ the next time that we visit $v[i]$. If we did not do this we would have an incomplete search algorithm.

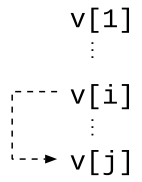

Figure 2: Forward Checking Scenario 2

It is worth noting that FC-BM automatically achieves directed 2-consistency [Dechter and Pearl 1988] during the search process. This is demonstrated in figure 2, the second forward checking scenario. Assume that we attempt the instantiation $v[i] \leftarrow k$ and this annihilates $current-domain[j]$. Further assume that no other variable forward checks against $v[j]$. Therefore, all values in $current-domain[j]$ are incompatible with $v[i] \leftarrow k$. When we undo the effects of forward checking from $v[i]$ the array element $past-fc[j]$ will be $\{0\}$, consequently $mcl[i,k]$ will be set to 0 (ie. the largest valued index in $past-fc[j]$). The instantiation $v[i] \leftarrow k$ will never be attempted again, because $mcl[i,k]$ will always be less than $mbl[i]$ during the search process. It is only when we attempt to backtrack beyond $v[1]$ to $v[0]$ that we would allow this value, and when we backtrack to $v[0]$ the search process terminates as there is no solution to the csp.

5 No type (b) saving in FC-BM?

There is no efficient type (b) saving within forward checking. If we know that forward checking from the instantiation $v[i] \leftarrow k$ to $v[j]$ does not annihilate $current-domain[j]$ we must still go ahead and do it. We must go ahead and filter out the incompatible values from $current-domain[j]$ otherwise we could then make an instantiation of $v[j]$ that was in conflict with the past variables. To be strictly correct we should then call FC-BM, FC-PBM (Forward Checking with Partial BackMarking).

6 FC with Conflict-directed BackJumping (FC-CBJ)

If we assume that we have made the effects of forward checking explicit, via $past-fc[j]$ then we can exploit this information to allow conflict-directed backjumping (CBJ) within forward checking [Prosser 1993b/c]. Assume that $v[i] \leftarrow k$ annihilates $current-domain[j]$ and that we then undo the effects of forward checking from $v[i]$ to $v[j]$. Further assume that k was the only value in $current-domain[i]$ (only to simplify this explanation). In order to resolve this conflict we have 2 options open to us. First, we may uninstantiate some variable that forward checks against $v[j]$ such that $current-domain[j]$ is relaxed, or we can uninstantiate some variable that forward checks against $v[i]$. That is we should jump back to the highest indexed variable in $past-fc[i] \cup past-fc[j]$.

In fact we can maintain a conflict set [Dechter 1990] for each variable, $conf-set[i]$ (where initially $conf-set[i]$ is set to $\{0\}$ for all i, corresponding to a conflict with the pseudo variable $v[0]$). If an instantiation of $v[i]$ annihilates $current-domain[j]$ we undo the effects of forward checking and update $conf-set[i]$ such that it becomes $conf-set[i] \cup past-fc[j]$. When there are no values remaining to be tried in $current-domain[i]$ we then jump back to $v[h]$ where h is the highest valued index in $past-fc[i] \cup conf-set[i]$, and we update $conf-set[h]$ to become $conf-set[h] \cup conf-set[i] \cup past-fc[i] - h$.

In figure 3, the third forward checking scenario, $v[f]$ checks forwards against $v[h]$, $v[g]$ checks forwards against $v[j]$, and $v[h]$ checks forwards against $v[i]$. The instantiation $v[i] \leftarrow k$ annihilates $current-domain[j]$. Consequently $conf-set[i]$ becomes {0,g}. If there are no more values to be tried in $current-domain[i]$ we then jump back to the largest index in $conf-set[i] \cup past-fc[i]$ (ie. largest element of {0,h,g}), namely $v[h]$, and $conf-set[h]$ becomes {0,g}. If there are no values remaining to be tried for $v[h]$ we then jump back to $v[g]$, and $conf-set[g]$ becomes {0,f}. We might then jump back from $v[g]$ to $v[f]$, setting $conf-set[f]$ to $\{0\}$, and ultimately jumping from $v[f]$ to the pseudo variable $v[0]$ (where the search process would terminate).

7 FC with BM and CBJ (FC-BM-CBJ)

Just as we can combine backjumping with backmarking (to give BMJ) [Prosser 1993a], and conflict-directed backjumping with backmarking (BM-CBJ) [Prosser 1993c]), we can also incorporate backmarking into FC-CBJ (with minor surgery). When an instantiation $v[i] \leftarrow k$ annihilates $current-domain[j]$ we remember

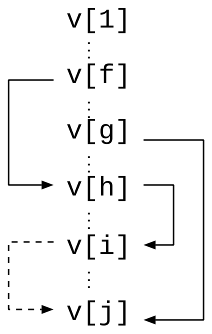

Figure 3: Forward Checking Scenario 3

that $v[j]$ was in fact the "victim" of forward checking. That is, if $v[i] \leftarrow k$ annihilates $current - domain[j]$ we undo the effects of forward checking from $v[i]$, update $mcl[i,k]$ such that it becomes the highest valued index in $past - fc[j]$, and update the array element $victim[i,k] \leftarrow j$ (where $victim$ is an n by m array, each element initialised to some 'dont care' value). We also update $conf - set[i]$ to maintain the backjumping information required by the CBJ component of FC-CBJ.

When we re-visit $v[i]$, and are about to make the instantiation $v[i] \leftarrow k$ we make a type (a) saving because $mcl[i,k] < mbl[i]$. However, we still have to update $conf - set[i]$. We do this by setting j to be $victim[i,k]$ and $conf - set[i]$ to be $conf - set[i] \cup past - fc[j]$. If there are no values remaining to be tried for $v[i]$ we then jump back to $v[h]$ where h is the largest valued index in $conf - set[i] \cup past - fc[i]$. When we jump back from $v[i]$ to $v[h]$ we update $mcl[i,k]$ in a manner similar to that described for FC-BM, set $mbl[i]$ to h, and set $mbl[j]$ to the minimum value of $mbl[j]$ or h, for all j, where $h < j \leq n$.

In fact the updating of $mcl[i,k]$, on jumping back from $v[i]$ to $v[h]$, is a bit more involved than in FC-BM. This is because we might have jumped over $v[i]$ at some earlier stage in the search process when $mbl[i] = 0$. The following scenario is possible.

(1) The array element $mbl[i] = 0$, and the instantiation $v[i] \leftarrow k_1$ fails. Consequently $mcl[i,k_1]$ is set to f (where f is the highest valued index in $past - fc[j]$).

(2) The instantiation $v[i] \leftarrow k_2$ succeeds, $mcl[i,k_2] \leftarrow i$, and the search process proceeds to $v[j]$.

(3) No consistent instantiation can be found for $v[j]$, and the search process jumps back to $v[f]$. Note that $mbl[i]$ retains the value 0.

(4) Variable $v[f]$ is assigned a new value (and this is consistent with the current search state). Consequently, the instantiation $v[i] \leftarrow k_1$ should be considered when $v[i]$ becomes current.

(5) The search process proceeds from $v[f]$ to $v[g]$, and $v[g] \leftarrow k_g$ forward checks against $v[i]$, and removes k_1 from $current - domain[i]$.

(6) $v[i]$ again becomes the current variable, no consistent value can be found, and the search process jumps back to $v[h]$ (naively setting $mbl[i]$ to h) and then jumps back from $v[h]$ to $v[g]$. Consequently k_1 is returned to $current - domain[i]$, $mcl[i,k_2] = f$, and (naively) $mbl[i] = g$ (where $f < g$).

(7) When $v[i]$ again becomes current, the instantiation $v[i] \leftarrow k_2$ will be (wrongly) disallowed (because $mcl[i,k_2]$ is out of date).

Therefore when jumping back from $v[i]$ we need to do as follows.

If $mbl[i] < h$ then for all $k \in domain[i]$, if $mcl[i,k] \geq mbl[i]$ then set $mcl[i,k]$ to i

Consequently, when $mbl[i] = 0$, and the search process jumps back from $v[i]$, all the backmarking information in $mcl[i,k]$ is lost. The backmarking information in $mcl[i,k]$ is therefore only of value on (at best) the third visit to $v[i]$ (rather than the second visit, as in BM and FC-BM).

8 FC-BM-CBJ might not be as good as FC-CBJ

We might expect, intuitively, that FC-BM-CBJ will never perform more consistency checks, nor will it visit more nodes than FC-CBJ. We shouldn't trust intuition here. It may be the case that even though FC-BM-CBJ attempts to avoid performing redundant consistency checks this may result in a loss of search knowledge, and this may degrade search performance. There are two scenarios to consider.

(1) Assume that FC-BM-CBJ is about to make a type (a) saving on the instantiation $v[i] \leftarrow k$ because it has previously been discovered that it will result in domain annihilation for $current - domain[j]$ (ie $victim[i,k] = j$). FC-BM-CBJ will not perform any forward checking from this instantiation, but will update $conf - set[i]$ to be $conf - set[i] \cup past - fc[j]$. Assume that h is the deepest variable forward checking against $v[j]$ and that FC-BM-CBJ then jumps back to $v[h]$.

(2) FC-CBJ makes the instantiation $v[i] \leftarrow k$ and checks forwards to $v[i+1]$, then to $v[i+2]$, and so on till $v[j']$, where $j' < j$. Assume that this results in the annihilation of $current - domain[j']$. That is, instantiations have changed for some $v[h']$, where $mbl[i] < h' < i$, such that $v[h']$ now checks against $v[j']$, and $v[i] \leftarrow k$ now annihilates $current - domain[j']$. We would then update $conf - set[i]$ to become $conf - set[i] \cup past - fc[j']$, and we may then jump back to $v[h']$, where $h < h'$.

Therefore FC-BM-CBJ may jump back to $v[h]$ as a result of the type (a) saving, whereas FC-CBJ jumps back to $v[h']$, where $h < h'$. FC-CBJ might then jump back from $v[h']$ to $v[g]$, where $g < h$, and therefore visit less nodes in the search tree than FC-BM-CBJ. However, it is equally likely that FC-BM-CBJ will jump back from $v[h]$ to $v[f]$, where $f < g$. In summary, adding backmarking to FC-CBJ can result in an increase in search effort due to search knowledge ignored due to the type (a) saving, or it can lead to a reduction in search effort due to the the type (a) saving combined with the preservation of search knowledge. In some respects, this phenomenon is similar to the *bridge* described in [Prosser 1993b].

9 Empirical Analysis of the Algorithms

Experiments were performed over a range of csp's, randomly generated in a manner similar to that of [Dechter 1990] and [Freuder and Wallace 1992]. An instance of the binary constraint satisfaction problem (csp) can be classified as a four-tuple $< n, m, p_1, p_2 >$ where n is the number of variables in that csp, there is a uniform domain size m, p_1 is the probability of a pair of symmetric constraints existing between any two variables, and p_2 is the probability of a conflict between a labelling of a pair of variables (where there are constraints between that pair of variables).

In [Cheeseman et al 1991] it is noted that many instances of NP-complete problems are easy to solve, and that computationally hard problems are rare. For csp's these *really* hard problems appear to exist at the critical point that separates under-constrained problems from over-constrained problems. That is, if we are given values for n, m, and p_1, there will be a value for p_2, call it Ψ, that corresponds to a *phase transition*. For a csp $< n, m, p_1, p_2 >$, if p_2 is less than Ψ the probability that the csp is satisfiable is close to 1, and the problem tends to be easy. Conversely, if p_2 is greater than Ψ the probability of satisfiability is close to zero, and again the problem is easy. However, when p_2 is close to Ψ the problem tends to be hard. This feature has been exploited in the experiments, such that the algorithms were applied only to hard problems.

The algorithms BM, FC, FC-BM, FC-CBJ, and FC-BM-CBJ were applied to 90 randomly generated csp's. That is, 10 hard csp's were generated at each parameter point $< 15, 5, p_1, p_2 >$, where p_1 and p_2 took the pairs of values $< 0.1, 0.7 >$, $< 0.2, 0.6 >$, $< 0.3, 0.5 >$, $< 0.4, 0.4 >$, $< 0.5, 0.3 >$, $< 0.6, 0.3 >$, $< 0.7, 0.25 >$, $< 0.9, 0.2 >$, and $< 1.0, 0.2 >$. The algorithms were then applied to each of the problems, and a measure was taken of the number of consistency checks performed on each problem (Table 1), the number of nodes visited in the search tree (Table 2), and the amount of CPU seconds consumed (Table 3).

Algorithm	μ	σ	min	max
BM	13,487	21,427	29	162,249
FC	5,766	7,541	39	59,906
FC-BM	5,081	6,772	39	56,254
FC-CBJ	4,070	6,288	39	53,342
FC-BM-CBJ	3,876	5,980	39	50,416

Table 1: Consistency Checks for 90 hard csp's at $n = 15$ $m = 5$

In Table 1 (and subsequent tables) column μ is the average, σ is the standard deviation, *min* is the minimum, and *max* the maximum. If we take consistency checks as a measure of search effort we see that

FC is better than BM, and that adding BM to FC gives a marginal improvement to FC. It also appears that adding conflict-directed backjumping gives a further improvement, but over these problems the improvement is not significant. The results are shown graphically in Figure 4. The graph shows the percentage of problems that were solved (y axis) within a given number of consistency checks (x axis, log scale) for the algorithms BM, FC, FC-BM, and FC-CBJ. In these trials it can be seen that FC is significantly better than BM, FC-BM is only marginally better than FC, and that FC-CBJ is the best of the 4 algorithms displayed.

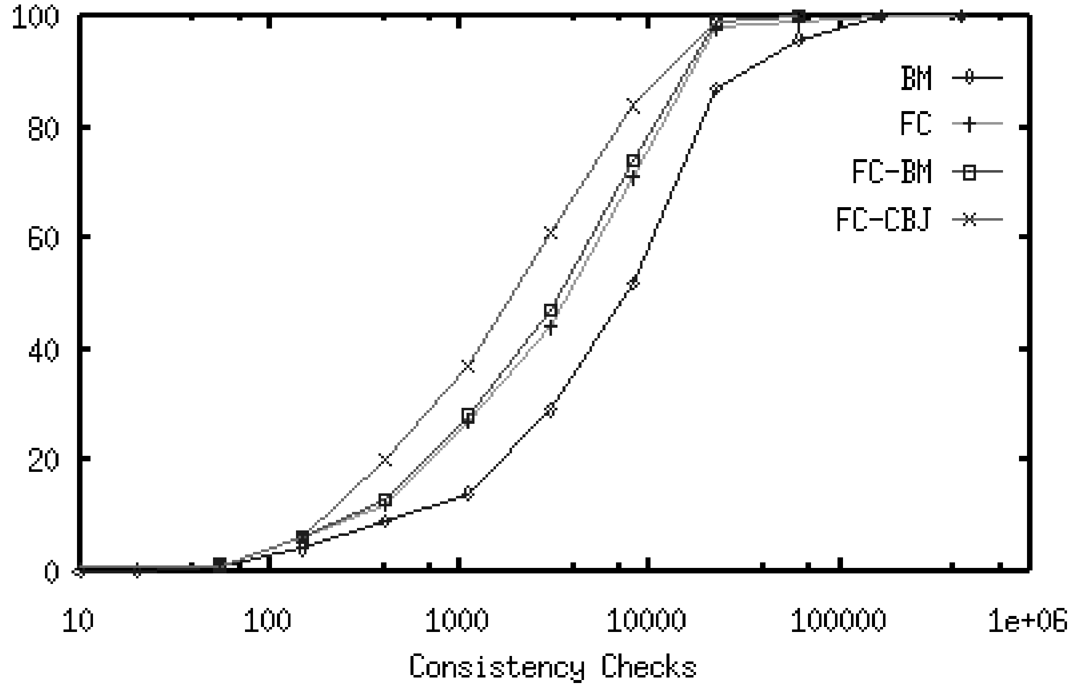

Figure 4: BM, FC, FC-BM, and FC-CBJ at n=15 m=5

Algorithm	μ	σ	min	max
BM	47,908	170,540	21	1,291,155
FC	611	1,189	2	6,724
FC-BM	611	1,189	2	6,724
FC-CBJ	269	575	2	5,379
FC-BM-CBJ	268	574	2	5,379

Table 2: Nodes Visited for 90 hard csp's at $n = 15$ $m = 5$

Table 2 shows the number of nodes visited, ie the number of times an attempt was made to instantiate a variable. BM performs very poorly in this respect (the same as chronological backtracking). Over the 90 problems there were 14 cases where BM performed less checks than FC, but only 3 cases where BM took less CPU time than FC. There were 9 cases where BM performed less checks than FC-BM, but only 2 cases where BM took less CPU time than FC-BM. FC-BM never performed more consistency checks than FC (as expected), although there were 33 cases where FC took less CPU time than FC-BM. FC-CBJ never performed more consistency checks than FC (as expected). There were 2 cases where FC-BM performed better than FC-BM-CBJ. This was not initially expected, but can be explained due to the possible loss of backmarking information during backjumping. This phenomenon has already been observed for other algorithms that combine BM with backjumping [Prosser 1993a]. That is, when FC-BM-CBJ jumps back from $v[i]$ to $v[g]$ over $v[h]$ where $g < h$, if $mbl[h] < g$ useful information may be lost in $mcl[h, k]$ (a fuller explanation is given in [Prosser 93a]).

In Table 3 we have the CPU time, measured in seconds. BM is the most expensive algorithm to run. Even though BM performs approximately 3 times as many consistency checks as any of the other algorithms, it

Algorithm	μ	σ	min	max
BM	14.8	48.7	0.01	355
FC	0.94	1.38	0.01	10.5
FC-BM	0.90	1.34	0.01	10.9
FC-CBJ	0.63	1.12	0.01	10.2
FC-BM-CBJ	0.63	1.12	0.01	10.2

Table 3: CPU Seconds for 90 hard csp's at $n = 15$ $m = 5$

takes on average 15 times as long to run. This is due to the computational effort in maintaining the array elements in *mbl* during backtracking. As can be seen, over these problems, there is little if anything to choose between FC, FC-BM, FC-CBJ, and FC-BM-CBJ. In conclusion, problems at $n = 15$ and $m = 5$ do not appear to be hard enough to discriminate between the algorithms.

The problem size was then increased to $n = 20$ and $m = 5$. Unfortunately these problems were just too hard for BM, and the experiments were terminated after a number of hours. Table 4 gives anecdotal evidence of just why this was so.

Algorithm	Problem	Checks	Nodes	CPU second
BM	$< 20, 5, 0.1, 0.3 >$	13,251,616	58,227,357	19,787
FC		74,200	4,752	17.09
FC-CBJ		277	34	0.03
BM	$< 20, 5, 0.1, 0.4 >$	1,320,172	2,571,704	1,074
FC		319,899	80,370	92
FC-CBJ		163	29	0.02

Table 4: *Really* hard problems for BM at $n = 20$ $m = 5$

As can be seen (Table 4), in the first problem $< 20, 5, 0.1, 0.3 >$ BM took in excess of 5 hours CPU time, FC took about 17 seconds, and FC-CBJ took about a fiftieth of a second. FC-CBJ was more than half a million times faster than BM. In the second problem BM took about 18 minutes, FC took a minute and a half, and again FC-CBJ took a fiftieth of a second. It appears that backmarking is practically useless when we move towards relatively large problems, and that we should use the FC based algorithms.

The problem size was then increased to $n = 25$ and $m = 5$, and 500 problems were generated at $p_1 = 0.2$ and $p_2 = 0.4$. The forward checkers FC, FC-BM, FC-CBJ, and FC-BM-CBJ were then applied to each of these problems. Of the 500 problems, 238 were satisfiable. That is, the parameters correspond to the phase transition point, where hard problems exist [Cheeseman et al 1991]. Table 5 gives the performance of the algorithms with respect to consistency checks performed and CPU time in seconds.

Measure	Algorithm	μ	σ	min	max
Checks	FC	225,383	623,201	166	7,458,694
	FC-BM	167,312	461,321	166	5,721,697
	FC-CBJ	39,690	88,963	166	1,232,463
	FC-BM-CBJ	34,281	76,320	166	1,108,056
CPU secs	FC	44	123	0.02	1,427
	FC-BM	38	108	0.02	1,246
	FC-CBJ	7.4	17	0.02	241
	FC-BM-CBJ	6.8	15	0.03	230

Table 5: 500 csp's at $< 25, 5, 0.2, 0.4 >$

Figure 5 shows the forward checking algorithms' performance graphically. We can see that adding backmarking to forward checking gives a marginal improvement (difference between FC and FC-BM, and difference between FC-CBJ and FC-BM-CBJ), typically about 10% measured in checks and cpu seconds. Adding conflict-directed backjumping gives a significant improvement (difference between FC and FC-CBJ, and FC-BM and

FC-BM-CBJ), typically an improvement of about a factor of 5. Therefore it appears that the chronological backtrackers, FC and FC-BM, should be dispensed with.

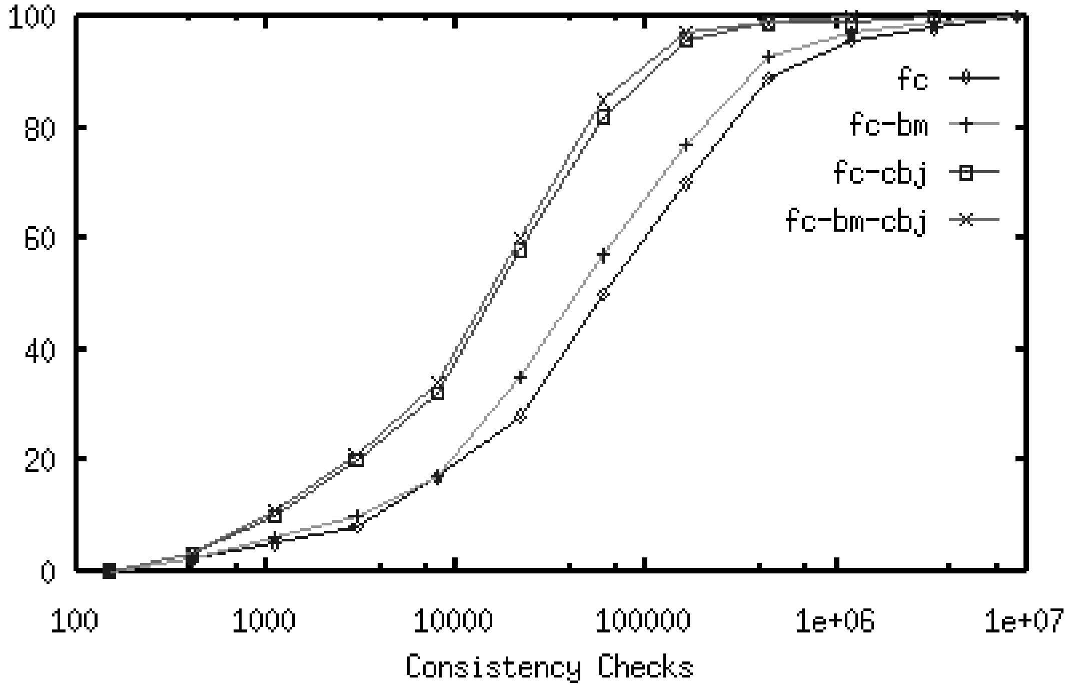

Figure 5: 500 csp's at n=25, m=5

The problem size was increased once more, and 100 problems were generated at $< 30, 7, 0.5, 0.2 >$. FC and FC-BM were dropped from the experiments. Figure 6 shows just how much harder the problems become as

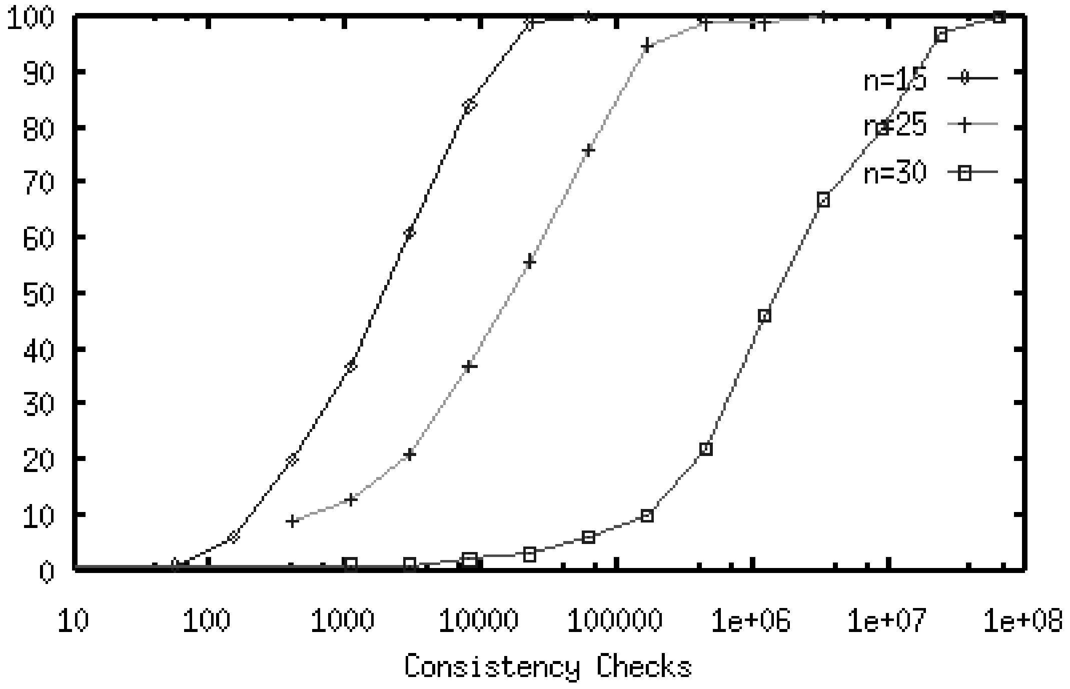

Figure 6: Increasing hardness with increasing n

we increase n. The 3 curves show (again) the percentage of problems solved (y axis) within a given number of consistency checks (x axis, log scale) for FC-CBJ, for 90 problems at $n = 15$ and $m = 5$, 100 problems at

$< 25, 5, 0.2, 0.4 >$, and 100 problems at $< 30, 7, 0.5, 0.2 >$

Measure	Algorithm	μ	σ	min	max
Checks	FC-CBJ	4,933,669	7,536,729	1,043	36,238,034
	FC-BM-CBJ	4,375,638	6,631,681	1,043	32,242,148
CPU secs	FC-CBJ	760	1,175	0.14	5,695
	FC-BM-CBJ	705	1,081	0.15	5,336

Table 6: 100 csp's at $< 30, 7, 0.5, 0.2 >$

As can be seen (Table 6 and Figure 6) these problems are *really* hard. Collectively, more than 81 hours CPU time was spent running these experiments. Of the 100 problems, 90 were satisfiable. FC-CBJ never performed less checks than FC-BM-CBJ, although there were 2 cases where FC-BM-CBJ visited more nodes than FC-CBJ (and this was anticipated). There were 2 cases where FC-BM-CBJ took more CPU time than FC-CBJ, but this difference was measured in hundredths of seconds. Figure 7 shows graphically the difference

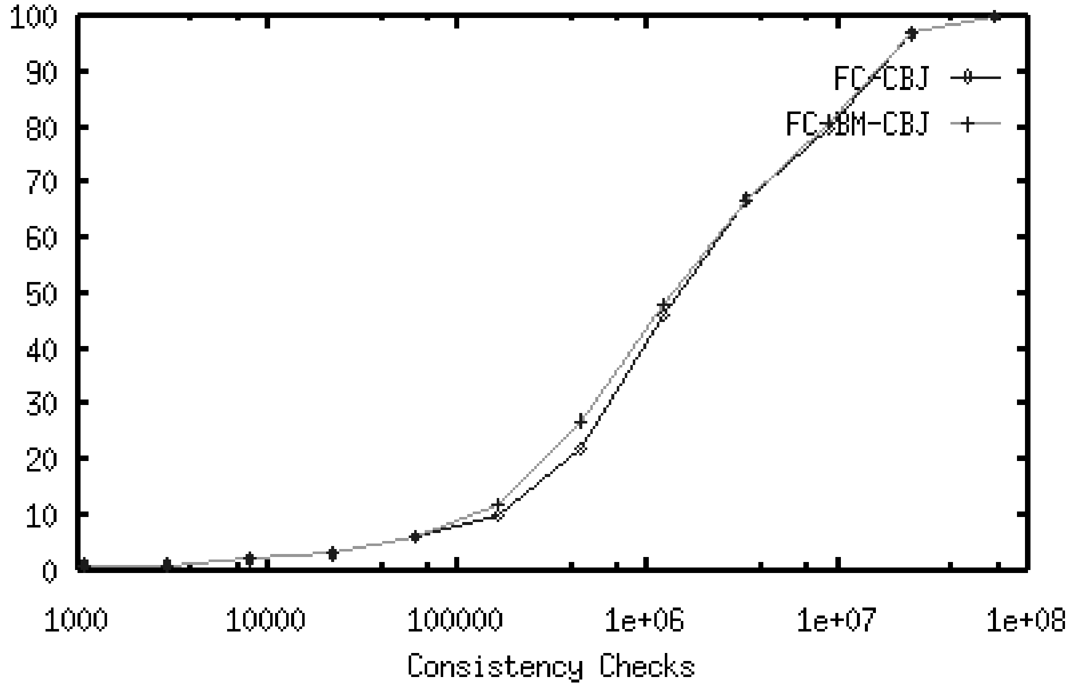

Figure 7: FC-CBJ versus FC-BM-CBJ over 100 problems at $< 30, 7, 0.5, 0.2 >$

in performance between FC-CBJ and FC-BM-CBJ. The performance improvement brought about by adding BM to FC-CBJ should be viewed in context. In the problems generated, a constraint was represented as a set of conflicting pairs of values. Therefore, to determine if a pair of values were consistent, a set membership function was performed. For the $< 25, 5, 0.2, 0.4 >$ problems a constraint $C[i, j]$ would typically contain $p_2.m^2$ pairs, ie. 10 pairs for this problem, and for $< 30, 7, 0.5, 0.2 >$ 10 pairs again. Therefore, the cost of checking a constraint is very small. If the cost of checking constraints is high then we should expect that FC-BM-CBJ becomes even more attractive.

10 Conclusion

Forward checking and backmarking have been combined to give FC-BM (although only half of BM's savings have been incorporated). This has brought about a saving in search effort (consistency checks and nodes visited) and run time. BM has also been added to FC-CBJ, to give FC-BM-CBJ. It was anticipated that in unusual circumstances FC-BM-CBJ could perform worse than FC-CBJ. Out of 790 problems, no such cases were encountered.

All the experiments were carried out using hard problems, although some of these were harder than others. The chronological backtrackers (BM, FC, FC-BM) were quickly overwhelmed by even moderately large, hard problems ($n = 20$, $m = 5$), whereas the backjumpers (FC-CBJ and FC-BM-CBJ) could handle larger, really hard problems ($n = 30$, $m = 7$). For all practical purposes chronological backtracking should be considered as obsolete. Forward checking should always be combined with conflict-directed backjumping, and this algorithm may be improved with (partial) backmarking. It remains to be seen if FC-BM-CBJ can exploit variable ordering heuristics without corrupting the backmarking information in *mcl* and *mbl*.

From a subjective point of view, it may be argued that FC-BM and FC-BM-CBJ are overly complicated, and probably unnatural (if we allow ourselves to consider FC, BM, CBJ, as natural). This is because FC and BM are unwilling bed-fellows. Backmarking, to be effective, needs to explore all values in the domain of the current variable, whereas in forward checking some of the values in the domain of the current variable may be filtered out and thus not explored. Consequently the backmarking information is compromised. Ideally we need to collect richer information in order to make better use of knowledge discovered during the search process, but in doing so we must incur the overhead of maintaining that information. Just where the break even point is, between exploitation (which involves explicating and maintaining search knowledge) and exploration is unclear, but at least we know that there is a spectrum. FC-CBJ would be at the origin, exploiting minimal information. Next up would be FC-BM-CBJ, then FC-DDB [Prosser 1990] and finally FC with deKleer's ATMS [Smith 1988].

References

[Bitner and Reingold 1975] J.R. Bitner and E. Reingold, Backtrack programming techniques, *Commun. ACM,* 18 (1975) 651-656

[Cheeseman et al 1991] P. Cheeseman, B. Kanefsky, W.M. Taylor, Where the *really* hard problems are. *Proc IJCAI-91* 331-337

[Dechter and Pearl 1988] R. Dechter and J. Pearl, Network-based heuristics for constraint-satisfaction problems, *Artif. Intell.* 34(1) (1988) 1-38

[Dechter 1990] R. Dechter, Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition, *Artif. Intell.* 41 (3) (1990) 273-312

[Dechter 1992] R. Dechter, Constraint Networks, in *Encyclopedia of Artificial Intelligence* (Wiley, New York, 2nd ed., 1992) 276-286

[Freuder and Wallace 1992] E.C. Freuder and R.J. Wallace, Partial constraint satisfaction, *Artif. Intell.* 58(1-3) (1992) 21-70

[Gaschnig 1977] J. Gaschnig, A General Backtracking Algorithm that Eliminates Most Redundant Tests, *Proceeding IJCAI-77* (1977) 457

[Gaschnig 1979] J. Gaschnig, Performance measurement and analysis of certain search algorithms, Tech. Rept. CMU-CS-79-124, Carnegie-Mellon University, Pittsburgh, PA (1979)

[Golomb and Baumert 1965] S.W. Golomb and L.D. Baumert, Backtrack Programming, *J. ACM,* 12 (1965) 516-524

[Haralick and Elliott 1980] R.M. Haralick and G.L. Elliott, Increasing Tree Search Efficiency for Constraint Satisfaction Problems, *Artif. Intell.* 14 (1980) 263-313

[Kumar 1992] V. Kumar, Algorithms for constraint satisfaction problems: a survey, *AI magazine* 13 (1) (1992) 32-44

[Mackworth 1977] A.K. Mackworth, Consistency in Networks of Relations, *Artif. Intell.* 8 (1) (1977) 99-118

[Mackworth 1992] A.K. Mackworth, Constraint Satisfaction, In: *Encyclopedia of Artificial Intelligence, Second Edition,* Volume 1, 285-293

[Meseguer 1989] P. Meseguer, Constraint satisfaction problems: an overview, *AICOM* 2 (1) (1989) 3-17

[Muscettola 1993] N. Muscettola, Scheduling by iterative partitioning of bottleneck conflicts. *Proc CAIA-93*, 49-55

[Nadel 1989] B.A. Nadel, Constraint Satisfaction Algorithms, *Computational Intelligence* 5(4): 188-224, 1989

[Nadel and Lin 1991] B.A. Nadel and J. Lin, Automobile transmission design as a constraint satisfaction problem: modelling the kinematic level. *AI EDAM* 5(3), 137-171

[Prosser 1990] P. Prosser, Distributed Asynchronous Scheduling, Phd Thesis, Department of Computer Science, University of Strathclyde, Glasgow.

[Prosser 1993a] P. Prosser, BM+BJ=BMJ, *Proceedings CAIA-93,* (1993) 257-262

[Prosser 1993b] P. Prosser, Domain filtering can degrade intelligent backtracking search. To appear in *Proc IJCAI-93*

[Prosser 1993c] P. Prosser, Hybrid algorithms for the constraint satisfaction problem, *Computational Intelligence* 9(3) 268-299

[Smith 1988] B.M. Smith, Forward Checking, the ATMS and search reduction, in Reason maintenance systems and their applications, Editors B.M. Smith and G. Kelleher, Ellis Horwood Series in Artificial Intelligence, pages 155-168

[Smith and Cheng 1993] S.F. Smith and C-C Cheng, Slack-based heuristics for constraint satisfaction scheduling. To appear in *Proc AAAI-93.*

Appendix. Pseudo-code for the Algorithms

The pseudo-code for FC-BM and FC-BM-CBJ is given below. The algorithms are described using the same conventions as those in [Prosser 1993c], but with the following exceptions. The assignment operator $\leftarrow$ has been replaced with <-, the not equals operator $\neq$ is replaced with /=, less than or equal $\leq$ is replaced with <=. To realise FC-BM the functions fc-bm-label and fc-bm-unlabel should be substituted into the function bcssp described in [Prosser 1993c], and to realise FC-BM-CBJ substitute fc-bm-cbj-label and fc-bm-cbj-unlabel into bcssp.

```
FUNCTION fc-bm-label(i,consistent): INTEGER
BEGIN
 consistent <- false;
 FOR k <- EACH ELEMENT OF current-domain[i] WHILE not consistent
 DO BEGIN
    consistent <- mbl[i] <= mcl[i,k];
    IF consistent
    THEN BEGIN
        v[i] <- k;
        FOR temp <- i+1 TO n WHILE consistent
        DO BEGIN
           j <- temp;
           consistent <- check-forwards(i,j)
           END;
        IF not consistent
        THEN BEGIN
             current-domain[i] <- remove(k,current-domain[i]);
             undo-reductions(i);
             mcl[i,k] <- max-list(past-fc[j])
             END
```

```
          ELSE mcl[i,k] <- i;
          END
     ELSE current-domain[i] <- remove(k,current-domain[i])
     END
 IF consistent THEN return(i+1) ELSE return(i)
END;
```

```
FUNCTION fc-bm-unlabel(i,consistent): INTEGER
BEGIN
 h <- i-1;
 undo-reductions(h);
 update-current-domain(i);
 IF 0 < mbl[i] and mbl[i] < h
 THEN FOR k <- EACH ELEMENT OF domain[i]
      DO IF mbl[i] <= mcl[i,k]
         THEN mcl[i,k] <- i;
 mbl[i] <- h;
 FOR j <- i+1 TO n
 DO mbl[j] <- min(mbl[j],h);
 current-domain[h] <- remove(v[h],current-domain[h]);
 consistent <- current-domain[h] /= nil
 return(h);
END;
```

```
FUNCTION fc-bm-cbj-label(i,consistent): INTEGER
BEGIN
 consistent <- false;
 FOR k <- EACH ELEMENT OF current-domain[i] WHILE not consistent
 DO BEGIN
    consistent <- mbl[i] <= mcl[i,k];
    IF consistent
    THEN BEGIN
         v[i] <- k;
         FOR temp <- i+1 TO n WHILE consistent
         DO BEGIN
            j <- temp;
            consistent <- check-forwards(i,j)
            END;
         IF not consistent
         THEN BEGIN
              current-domain[i] <- remove(k,current-domain[i]);
              undo-reductions(i);
              mcl[i,k] <- max-list(past-fc[j]);
              conf-set[i] <- union(conf-set[i],past-fc[j]);
              victim[i,k] <- j
              END
         ELSE mcl[i,k] <- i;
         END
    ELSE BEGIN
         current-domain[i] <- remove(k,current-domain[i]);
         j <- victim[i,k];
         conf-set[i] <- union(conf-set[i],past-fc[j])
         END
    END
 IF consistent THEN return(i+1) ELSE return(i)
END;
```

```
FUNCTION fc-bm-cbj-unlabel(i,consistent): INTEGER
BEGIN
 cs <- union(conf-set[i],past-fc[i]);
 h <- max-list(cs);
 conf-set[h] <- remove(h,union(conf-set[h],cs));
 IF mbl[i] < h
 THEN FOR k <- EACH ELEMENT OF domain[i]
      DO IF mbl[i] <= mcl[i,k]
         THEN mcl[i,k] <- i;
 mbl[i] <- h;
 FOR j <- i+1 TO n
 DO mbl[j] <- min(mbl[j],h);
 FOR j <- i DOWNTO h+1
 DO BEGIN
    conf-set[i] <- {0};
    undo-reductions(j);
    update-current-domain(j)
    END;
 undo-reductions(h);
 current-domain[h] <- remove(v[h],current-domain[h]);
 consistent <- current-domain[h] /= nil
 return(h);
END;
```