Minimal Forward Checking with Backmarking

Alvin Kwan and Edward Tsang
Department of Computer Science, University of Essex
Wivenhoe Park, Colchester, Essex CO4 3SQ, United Kingdom

alvin@essex.ac.uk edward@essex.ac.uk

March 1996

Abstract

The idea behind forward checking is to check forward so that a wrong decision can be detected in early stage during search and thus futile search space can be pruned. However constraint propagation in forward checking may be overdone in certain situations. In this paper we suggest a way to delay constraint evaluation in forward checking with backmarking. Conflict-directed backtracking can also be easily fitted in. Despite the new algorithms can be considered as extensions of Dent and Mercer's minimal forward checking, they employ simpler data structures which have a smaller space complexity. Experiments were conducted to compare them with minimal forward checking and other regular forward checking based algorithms. Like minimal forward checking, our new algorithms always perform no worse than forward checking and their "non-lazy" counterparts in terms of number of compatibility checks and node expanded given the same variable and value orderings. They also outperform minimal forward checking in almost all of our tested cases. Our experimental results show that the minimal width ordering heuristic can be a better choice than the fail-first heuristic for the new lazy algorithms in many occasions, particularly for problems with a sparse constraint graph. Our results also reveal that domain filtering could degrade a lazy forward checking algorithm in a static variable ordering even though it backtracks chronologically. This refutes the conjecture made by Prosser that increased consistency can guarantee a reduction in search effort for chronological backtrackers.

1 Introduction

A constraint satisfaction problem (CSP) consists of a set of variables, a set of domains for those variables and a set of constraints over those variables.[1] The aim is to find an assignment of value to each variable such that no constraints are violated. Forward checking (FC) is considered as one of the most efficient search method to solve CSPs. Enhancements have been proposed in order to make it more efficient [Prosser 93] [Prosser 93a]. One recent result of this effort is *minimal forward checking* (MFC) [Dent & Mercer 94]. MFC is interesting as it is superior to FC given the same variable and value orderings.[2] It saves the constraint propagation efforts that may be overdone in FC which in turn alleviates the problem of using FC for CSPs with large domain size. In this paper, we suggest two new algorithms which achieve similar contributions. The essence of these algorithms is to incorporate "laziness" into forward checking with backmarking (FC-BM) [Prosser 95]. One of them is further enhanced with the capability of conflict-directed backjumping (CBJ) [Prosser 93]. They can be considered as extensions of MFC despite the new algorithms employ simpler data structures and thus have a smaller space complexity than MFC.

In the next section, we present a naïve *incremental* or *lazy forward checking* algorithm and show with an example how it may work worse than FC. Then we build on top of that algorithm to obtain our *minimal forward checking with backmarking* in Section 3. With some minor changes, a more sophisticated version equipped with CBJ is given in Section 4. A comparative study was done on the regular and the lazy FC based algorithms. Descriptions on the design of experiments and results are detailed in Section 5. In analysing the experimental results, we notice that domain filtering in a lazy FC based algorithm can cause performance degradation even though it backtracks chronologically. This is contradictory to the conjecture that domain filtering should not degrade the performance of a chronological backtracker [Prosser 93a]. The details are discussed in Section 6. We conclude our work in the final section.

2 Naïve lazy forward checking

For ease of presentation, all variables in a CSP during a search are divided into three sets: past variables which have been instantiated, current variable which is being instantiated, and future variables which are not yet instantiated. Let D_i denote the domain of variable X_i and $\langle X_i, v_j \rangle$ denote the label such that variable X_i is instantiated to value v_j, the j-th value in D_i. The current domain of a variable, say X_i, at a particular search state is defined as the domain values of that variable which have not been proven infeasible in a solution at that search state.

[1] Note that the scope of this paper is confined to binary finite constraint satisfaction problems only.

[2] Unless otherwise stated, we take the number of compatibility checks performed by an algorithm as the performance indicator.

The motivation of delayed evaluation in forward checking is to avoid propagating constraints forward unnecessarily. For ease of reference, we refer this class of algorithms to as *lazy FC* or simply *lazy algorithms*. Unlike FC, lazy algorithms do not filter *all* domain values from the future variables which are incompatible with the past variables. Instead, lazy algorithms only ensure that there exists at least one compatible domain value from each of the future variables that is compatible with the past variables. One advantage of using a lazy algorithm over FC occurs when a dead-end is met. Suppose a dead-end is met by FC in labelling the variable at level i of the search tree, denoted by X_i, as it cannot find a legal label for a future variable X_j during constraint propagation due to a wrong assignment of value to X_h such that $h < i < j \leq N$ where N is the number of variables in that problem.[3] Note that all domain values from D_{i+1} to D_N incompatible with the past variables from X_h to X_{i-1} have already been filtered. This effort would be wasted as constraint propagation due to the labelling of variables from X_h to X_{i-1} are undone during backtracking. Lazy algorithms avoid this problem as they only find the first support value from each of the future variables. The cost of this is that only the first value in the current domain of each future variable is compatible with the past variables. The implication is that when a future variable is picked for instantiation, any value from its current domain other than the first must be checked against the past variables in addition to the future variables.[4] Similarly, when a lazy algorithm backtracks to a variable, its alternative value has to be checked against the past variables as well as the future variables.

A naïve implementation of a basic lazy algorithm, LFC, for finding the first solution of CSPs is given in next page.[5] The search is invoked by calling `lfc(1)`. For brevity, several procedures which are straightforward to implement are omitted. These include: (1) `save_future_var_status` which saves the status of the domains of the future variables so that they can be restored later by `restore_future_var_status` when the current variable's label is found not able to extend a partial solution; and (2) `print_solution` which outputs the first solution of the problem.

Although the implementation of LFC saves some excessive search effort that might have been wasted by FC, it may perform worse than FC in some occasions. Figure 1 depicts the constraint graph of an example CSP which causes such a problem. Traces of the execution of FC and LFC on the CSP are given in Figures 2 and 3 in a similar manner as in [Dent & Mercer 94]. The ticks (√) and crosses (×) represent successful and unsuccessful compatibility checks respectively. The "Checks" entry in each row of the tables represents the number of compatibility checks performed in that step. In Figure 3, domain values are shown with the list of current and past variables with which they have been checked.

[3] For ease of illustration, a static variable ordering is assumed throughout this paper unless otherwise stated. However, dynamic variable orderings can be achieved by indexing the variables.

[4] This implies a dynamic value ordering cannot be used in lazy algorithms unless non-trivial modifications are made.

[5] Note that all algorithms presented in this paper find the first solution of CSPs.

```
PROCEDURE check_future(i)
{
FOR m := i+1 TO N
  IF (a constraint exists between X_m and X_i)
    first_value := TRUE;
    FOR each value v_n in D_m
      IF (first_value = FALSE)
        IF (check_past(m, i) = FALSE)
          skip to the next iteration of the inner for-loop;
      first_value := FALSE;
      IF (<X_m,v_n> is compatible with X_i's label)
        skip to the next iteration of the outer for-loop;
    IF (no value in D_m supports label of X_i)
      RETURN;
}


PROCEDURE check_past(i, current_level)
{
FOR m := 1 TO current_level-1
  IF (a constraint exists between X_m and X_i)
    IF (labels of X_m and X_i are incompatible)
      RETURN FALSE;
RETURN TRUE;
}


PROCEDURE lfc(i)
{
IF (i = N)
  print_solution(i);
  EXIT;
ELSE
  save_future_var_status(i);
  first_value := TRUE;
  REPEAT
    instantiate X_i with an untried value v_j from D_i;
    IF (first_value = FALSE)
      IF (check_past(i, i) = FALSE)
        restore_future_var_status(i);
        skip to the next iteration of the repeat-loop;
    check_future(i);
    IF (no domain D_m is wiped out such that m > i)
      lfc(i+1);
    restore_future_var_status(i);
    first_value := FALSE;
  UNTIL (all values in D_i has been tried);
}
```

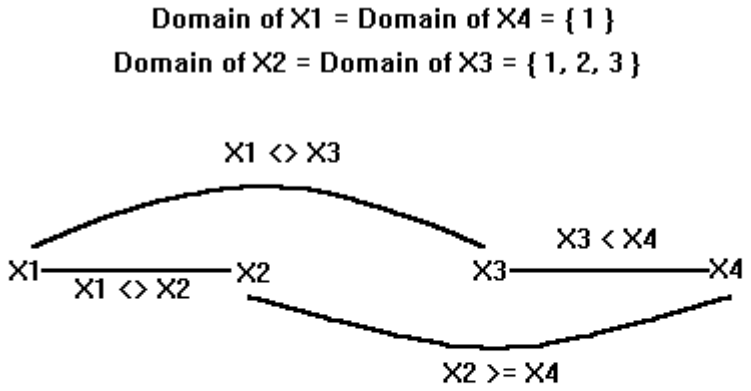

Figure 1. An example CSP

Step	D_1	D_2	D_3	D_4	Checks
0	1	1 2 3	1 2 3	1	0
1	$X_1 = 1$	1 (×) 2 (√) 3 (√)	1 (×) 2 (√) 3 (√)	1	6
2	$X_1 = 1$	$X_2 = 2$ 3	2 3	1 (√)	1
3	$X_1 = 1$	$X_2 = 2$ 3	$X_3 = 2$ 3	1 (×)	1
4	$X_1 = 1$	$X_2 = 2$ 3	$X_3 = 3$	1 (×)	1
5	$X_1 = 1$	$X_2 = 3$	2 3	1 (√)	1
6	$X_1 = 1$	$X_2 = 3$	$X_3 = 2$ 3	1 (×)	1
7	$X_1 = 1$	$X_2 = 3$	$X_3 = 3$	1 (×)	1
Total					12

Figure 2. Trace of execution of FC

For the given CSP, FC takes 12 compatibility checks to prove the insolubility of the problem whereas LFC takes 13. The reason for the inferiority of LFC is due to its failure to avoid some excessive consistency checks. As shown in Figure 3, LFC $<X_3, 3>$ checked backward against $<X_1,1>$ in steps 4 and 9 (which have been highlighted by shaded boxes) whereas FC does once only (step 1 in Figure 2). It is not difficult to observe that the excessive check can be avoided if LFC notices that the result of the previous check between $<X_3,3>$ and $<X_1,1>$ succeeded and the value of variable X_1 has not been changed since then. To achieve such a saving, MFC uses specialised data structures to record the results of consistency checks to avoid doing this type of redundant checks.[6] This ensures its superior performance over FC given the same variable and value orderings.

[6] MFC performs 12 compatibility checks to tackle the example problem depicted in Figure 1.

Step	D_1	D_2	D_3	D_4	Checks
0	1	1 2 3	1 2 3	1	0
1	$X_1 = 1$	1 (X_1 ×) 2 (X_1 √) 3	1 (X_1 ×) 2 (X_1 √) 3	1	4
2	$X_1 = 1$	$X_2 = 2$ 3	2 3	1 (X_2 √)	1
3	$X_1 = 1$	$X_2 = 2$ 3	$X_3 = 2$ 3	1 (X_2, X_3 ×)	1
4	$X_1 = 1$	$X_2 = 2$ 3	3 (X_1 √)	1 (X_2)	1
5	$X_1 = 1$	$X_2 = 2$ 3	$X_3 = 3$	1 (X_2, X_3 ×)	1
6	$X_1 = 1$	3 (X_1 √)	2 3	1	1
7	$X_1 = 1$	$X_2 = 3$	2 3	1 (X_2 √)	1
8	$X_1 = 1$	$X_2 = 3$	$X_3 = 2$ 3	1 (X_2, X_3 ×)	1
9	$X_1 = 1$	$X_2 = 3$	3 (X_1 √)	1 (X_2)	1
10	$X_1 = 1$	$X_2 = 3$	$X_3 = 3$	1 (X_2, X_3 ×)	1
Total					13

Figure 3. Trace of execution of LFC

3 Minimal forward checking with backmarking

As mentioned in [Dent & Mercer 94], backmarking [Gaschnig 77] can be used to improve MFC. Two types of saving can be made by BM [Nadel 89]: (a) if an instantiation of the current variable was found incompatible with a past variable's label, unless that past variable's label has been changed, otherwise the compatibility check between the same pair of variables' labels can be saved as it will fail again; (b) if an instantiation of the current variable was found compatible with a past variable's label, unless that past variable's label has been changed, otherwise the compatibility check between the same pair of variables' labels can be saved as it will succeed again. By integrating LFC and BM, a new algorithm which we call it *minimal forward checking with backmarking* (MFC-BM2) is produced.[7] Despite MFC-BM2 can be viewed as Prosser's FC-BM with delayed evaluation, there are two major differences. Firstly, we adopt a 2-dimensional data structure for storing the minimum backup level as suggested in [Kondrak & van Beek 95] while FC-BM uses an one dimensional array for the same data structure. Secondly, unlike FC-BM which does not have efficient type (b) saving [Prosser 95], MFC-BM2 benefits from both type (a) and type (b) saving. It is the "laziness" of the new algorithm that makes type (b) saving feasible.

[7] The '2' in MFC-BM2 emphasises the fact that a 2-dimensional *minimum backup level* is used.

Two 2-dimensional arrays are used for storing backmarking information, namely the maximum checking level (*mcl*) and the minimum backup level (*mbl*). The more involved *mbl* allows more precise backmarking information to be kept during search than Gaschnig's backmarking and usually helps to improve search efficiency. The *mcl*[*i*][*j*] entry is maintained to record the index of the shallowest variable that needs to be checked against the *j*-th value in variable X_i's domain. The *mbl*[*i*][*j*] entry stores the number of the shallowest variable whose value has changed since X_i was last instantiated with the *j*-th value. Note that all entries of *mcl* and *mbl* are initialised to zero.

Unlike Gaschnig's backmarking in which *mbl* is only updated during backtracking, MFC-BM2 updates *mbl* during constraint checking as well as backtracking. Different from FC-BM which does not update *mcl* and *mbl* of future variables during constraint checking, MFC-BM2 does update the data structures. These allow the new algorithm to achieve type (b) saving in addition to type (a) saving.

The *past_fc* data structure is used to record information about which past variables reduce a variable's domain. There are *N* entries for the structure, *i.e.*, one for each variable, and each entry is considered as a set of maximum size *M* where *M* is the largest domain size of the variables. Initially a zero is added to each entry of *past_fc*.

The algorithm of MFC-BM2 is given below. Note that both `check_past` and `check_future` are modified to take advantage of backmarking saving. The three omitted procedures used by LFC are also used by MFC-BM2. Two additional simple-to-code procedures are omitted. They are: (1) `min` which returns the smaller value of its two input parameters; and (2) `max` which returns the largest value in a set of numbers.

In the algorithm, "-" stands for the set difference operator for non-numeric operands whereas "∪" is the set union operator. Comment lines begin with "%". Note that the algorithm is described in a syntax similar to the one used to describe FC-BM in [Prosser 95] so that readers can contrast the two algorithms.

```
% check_future checks forward and tries to find the 1st
% support value from each of the future variables which
% are compatible with the current variable's label.

PROCEDURE check_future(i)
{
FOR m := i+1 TO N
  IF (a constraint exists between Xi and Xm)
    FOR (each value vn in the domain of Xm)
      old_mbl := mbl[m][n];
      mbl[m][n] := i;
      IF (mcl[m][n] >= old_mbl) % type (a) saving
        mcl[m][n] := check_past(m, i, old_mbl);
        IF (mcl[m][n] = i)  % no constraints is violated
          mcl[m][n] := i;
          IF (<Xm,vn> is incompatible with Xi)
            past_fc[m] := past_fc[m] ∪ {i};
          ELSE
            skip to the next iteration of the outer for-loop;
    IF (no value in Dm supports label of Xi)
      RETURN;
}
```

```
% check_past checks compatibility of Xi's label
% with past variables. It returns index of the past
% variable which is found incompatible with Xi's label
% or the index of the current variable otherwise.

PROCEDURE check_past(i, current_level, mbl)
{
FOR j = mbl TO current_level-1  % type (b) saving
  IF (a constraint exists between Xi and Xj)
    IF (labels of Xi and Xj are incompatible)
      past_fc[i] := past_fc[i] ∪ {j};
      RETURN j;
RETURN j;
}

PROCEDURE mfc_bm2(i)
{
IF (i = N)  % N is no. of variables in CSP
  print_solution(i);
  EXIT;
ELSE
  save_future_var_status(i);
  REPEAT
    instantiate Xi with an untried value vj;
    old_mbl := mbl[i][j];
    mbl[i][j] := i;
    IF (mcl[i][j] >= old_mbl)  % type (a) saving
      mcl[i][j] := check_past(i, i, old_mbl);
      IF (mcl[i][j] = i) % no constraint is violated
        check_future(i);
        IF (Dm is wiped out such that m > i)
          mcl[i][j] := max(past_fc[m] - {i});
        ELSE
          FOR m := i+1 TO N
             IF (a constraint exists between Xi & Xm)
              % suppose vn is the first value
              % in current domain of Xm;
              mcl[m][n] := i+1;
              mbl[m][n] := i+1;
          mfc_bm2(i+1);
    restore_future_var_status(i);
  UNTIL (all values in Di have been tried);
  IF (i = 1)
    print "no solution";
  ELSE
    FOR m := i TO N
      FOR (each value vn in unpruned domain of Xm)
        mbl[m][n] := min(mbl[m][n], i-1);
}
```

A description of the algorithm is as follows. When a value v_j is picked for the current variable, X_i, MFC-BM examines the *mcl* and *mbl* of value v_j to see whether type (a) saving can be achieved. Suppose no type (a) saving is achieved, *i.e.*, $\langle X_i, v_j \rangle$ has not been proved infeasible in a partial solution in early search, $\langle X_i, v_j \rangle$ is checked against the

past variables by `check_past`.[8] `check_past` returns a value i if no constraints are violated. If the returned value is smaller than i, that means $<X_i, v_j>$ was found incompatible with some past variable. The returned value is assigned to the *mcl* of $<X_i, v_j>$ which points to the shallowest variable that needs to be checked against if $<X_i, v_j>$ is to be used in future.

Suppose consistency has been maintained with the past variables, $<X_i, v_j>$ is then checked against the future variables by `check_future`. The *mbl* of the label of a future variable $<X_m, v_n>$ is set to the current level i. This is important for achieving backmarking saving. For example, if the search encounters a dead-end and backtracks to X_h where $h < i$, the *mbl* of $<X_m, v_n>$ is then set to a value g where $g \leq h$. If the compatibility of $<X_m, v_n>$ and the past variables has to be confirmed in subsequent search, due to type (b) saving, excessive checks against the already checked compatible past variables X_1 to X_{g-1} can be avoided.

`check_past` is then called to catch up any compatibility checks that have been omitted. If no inconsistency is detected, $<X_m, v_n>$ is checked against the current variable's label; otherwise an alternative value for X_m is picked. The process repeats until a support value from each of the future variables is found or the current domain of any future variable is exhausted.

If the current variable's label $<X_i, v_j>$ are supported by the future variables, the *mbl* and *mcl* of the support value of those future variables are updated to i+1. This allows potential type (b) saving when any of those values is picked for labelling in subsequent search. Note that the *mcl* entry of $<X_i, v_j>$ is not set to i+1 even $<X_m, v_n>$ supports $<X_i, v_j>$ since v_j is provisional and might be found incompatible with some other future variable later in the search. However, if $<X_i, v_j>$ exhausts the domain of a future variable, say D_m, that means the label has no chance of being in a solution unless the search backtracks to change the label of the deepest culprit variable preceding X_i which reduces D_m. To do this, we set the *mcl* of $<X_i, v_j>$ to that deepest culprit. Due to type (a) saving, $<X_i, v_j>$ will not be used again until the search backtracks to that deepest culprit. Note that if v_j is the only culprit that causes a domain "wipe-out", its *mcl* is set to zero because zero (which was added to all *past_fc* entries during initialisation) and i are the only elements in the *past_fc* of X_m. This effectively stops v_j in D_i from being used in future. Note that this type of domain filtering is similar to that of FC-BM [Prosser 95], but does not exist in MFC.

If all values in the current variable's domain are removed, MFC-BM backtracks to the previous level after maintaining the *mbl* entries of the current and future variables as in FC-BM.

A trace of execution of MFC-BM2 on the previous problem is given in Figure 4. MFC-BM2 has saved two additional checks in addition to eliminating the redundant check between $<X_3,3>$ and $< X_1,1>$. In step 3, $< X_3,2>$ is checked against $<X_4,1>$ which is the only possible label for variable X_4. They are found incompatible with one another. However, since D_4 has never been reduced by the past variables X_1 and X_2, "2" can be removed from D_3 permanently as it can never satisfy the constraint between X_3 and X_4. This is done by assigning the maximum checking level of the value to be pruned to zero.

[8] Note that due to type (b) saving, `check_past` performs no compatibility checks at all if $<X_i, v_j>$ is the first instantiation of X_i after the labelling of X_{i-1}.

Due to type (a) saving, "2" is effectively removed from D_3. Similarly, "3" is removed from D_3 in step 5. As D_3 is now empty, the search backtracks to X_2. X_2 is then instantiated with "3". The label is checked backward against X_1, then forward against X_4 and found compatible with them. Then X_3 is picked for labelling but its domain has been annihilated. Thus the search backtracks to X_2 again. Finding no more values in D_2, the search backtracks to X_1 and terminates since D_1 has been exhausted. MFC-BM2 takes ten compatibility checks to tackle the problem.

Step	D_1	D_2	D_3	D_4	Checks
0	1	1 2 3	1 2 3	1	0
1	$X_1 = 1$	1 (X_1 ×) 2 (X_1 √) 3	1 (X_1 ×) 2 (X_1 √) 3	1	4
2	$X_1 = 1$	$X_2 = 2$ 3	2 3	1 (X_2 √)	1
3	$X_1 = 1$	$X_2 = 2$ 3	$X_3 = 2$ 3	1 (X_2, X_3 ×)	1
4	$X_1 = 1$	$X_2 = 2$ 3	2 (pruned) 3 (X_1 √)	1 (X_2)	1
5	$X_1 = 1$	$X_2 = 2$ 3	$X_3 = 3$	1 (X_2, X_3 ×)	1
6	$X_1 = 1$	3 (X_1 √)	2 (pruned) 3 (pruned)	1	1
7	$X_1 = 1$	$X_2 = 3$	2 (pruned) 3 (pruned)	1 (X_2 √)	1
Total					10

Figure 4. Execution of Minimal Forward Checking with Backmarking

4 Minimal forward checking with backmarking and conflict-directed backjumping

The last two compatibility checks shown in Figure 4 can be avoided because D_3 has been reduced to empty after step 5 and X_1 is the only past variable which causes a domain reduction in D_3. Backtracking to X_2 cannot solve the problem. To avoid backtracking to a non-culprit, we add MFC-BM2 the capability of intelligent backjumping. We decided to incorporate CBJ into MFC-BM2. Some of the information that CBJ needs for backjumping is already available in *past_fc*. An extra data structure that it needs is the conflict sets of variables, *conf_set*, for storing the culprits leading to dead-ends during search. Like *past_fc*, *conf_set* has *N* entries -- one for each variable. Each entry has a maximum size of *M*. Initially a zero is added to each entry of *conf_set*.

The essence of the enhanced algorithm, named MFC-BM2-CBJ, lies on the maintenance of the conflict sets during the search. The process is similar to extending FC-

BM by CBJ as described in [Prosser 95] with one minor difference. Since MFC-BM2-CBJ is able to keep more precise backmarking information during search due to its 2-dimensional *mbl*, the 2-dimensional data structure *victim* used in FC-BM-CBJ [Prosser 95] can be eliminated. This makes the incorporation of CBJ into MFC-BM2 even simpler. Note that MFC-BM2-CBJ uses `check_past` and `check_future` as implemented in MFC-BM2. We believe the algorithm is self explanatory and therefore we do not provide more explanation here. The algorithm of MFC-BM2-CBJ is given below.

```
PROCEDURE mfc_bm2_cbj(i)
{
IF (i = N)  % N is no. of variables in CSP
  print_solution(i);
  EXIT;
ELSE
  save_future_var_status(i);
  conf_set[i] := {0};
  REPEAT
    instantiate Xi with an untried value vj;
    old_mbl := mbl[i][j];
    mbl[i][j] := i;
    IF (mcl[i][j] >= old_mbl)  % type (a) saving
      mcl[i][j] := check_past(i, i, old_mbl);
      IF (mcl[i][j] = i) % no constraint is violated
        check_future(i);
        IF (Dm is wiped out such that m > i)
          mcl[i][j] := max(past_fc[m] - {i});
          conf_set[i] :=
            conf_set[i] ∪ past_fc[m] - {i};
        ELSE
          FOR m := i+1 TO N
             IF (a constraint exists between Xi & Xm)
              % suppose vn is the first value
              % in current domain of Xm;
              mcl[m][n] := i+1;
              mbl[m][n] := i+1;
          back_level := mfc_bm2_cbj(i+1);
          IF (back_level =/= i)
            RETURN back_level;
    restore_future_var_status(i);
  UNTIL (all values in Di have been tried);
  IF (i = 1)
    print "no solution";
  ELSE
    conf_set[i] :=
      conf_set[i] ∪ past_fc[i] - {i};
    back_level := max(conf_set[i]);
    conf_set[back_level] := conf_set[back_level]
      ∪ conf_set[i] - {back_level};
    FOR m := back_level+1 TO N
      FOR (each value vn in unpruned domain of Xm)
        mbl[m][n] := min(mbl[m][n], back_level);
  RETURN back_level;
}
```

MFC-BM2-CBJ takes eight compatibility checks to prove insolubility of the example problem. The first five steps in the trace of the execution of MFC-BM2-CBJ is the same as those for MFC-BM2. However, when a dead-end is met in labelling X_3, the new algorithm analyses X_3's conflict set and finds X_1 as the only culprit in the conflict set. Thus it jumps back to X_1. As there is no more values in D_1, the search ends.

One point that is worth mentioning is that both MFC-BM2 and MFC-BM2-CBJ have a space complexity of O(*NM*) as both the data structures for storing backmarking information (*mbl* and *mcl*) and backjumping information (*conf_set*, *past_fc*) use O(*NM*) space. This is smaller than MFC's O(N^2M) as reported in [Dent & Mercer 94]. [9]

5 Experiments

The aim of our experiments was to provide preliminary answers to the following questions. How good are the new lazy algorithms when compared with FC, MFC and their "non-lazy" counterparts in dealing with problems of different sizes, constraint densities and constraint tightness? Which variable ordering heuristic is good for our lazy algorithms?

In our experiments, random binary CSPs were used. Each CSP is characterised by the tuple <*n*, *m*, *p*1, *p*2> where *n* is the number of variables, *m* is the uniform domain size of variables, *p*1 is the proportion of pairs of variable that a constraint exists between them and *p*2 is the proportion of inconsistent pairs of values for each pair of variables given that a constraints exists between them.[10] Only problems with connected constraint graphs were generated. Furthermore, neither null nor universal constraints were allowed, that is, $p2 \neq 0$ or 1.

For each data point, 50 problems were generated and tested. Since we are interested in the behaviour of our algorithms for CSPs of different characteristics, *p*2 was varied across a wide range of values, from 0.04 to 0.96, with a step of 0.04. The range of *p*2 was chosen because the domain sizes that used were 5, 10 and 20. *p*1 was set to 0.2, 0.5 and 0.8 to cover problems of low, medium and high constraint density. The CSPs that were tested are characterised by the following tuples. Asterisks means all values within the range as mentioned above were used.

<10, 5, *, *>, <10, 10, *, *>, <10, 20, *, *>, <20, 5, *, *>, <20, 10, *, *>,
<20, 20, *, *>, <30, 5, *, *>, <30, 10, *, *>, <40, 5, *, *>.

In all our experiments, in addition to FC, MFC, and our lazy algorithms, the "non-lazy" counterparts of MFC-BM2 and MFC-BM2-CBJ, FC-BM2 and FC-BM2-CBJ were also used. For comparison purpose, forward checking with conflict-directed backjumping

[9] Another implementation of MFC is described in [Bacchus & Grove 95] which has a space complexity of O(*NM*).

[10] We have used *p*2 as constraint looseness, which was given by [Haralick & Elliott 80] and [Nudel 83], in our previous work. However it appears that *p*2 has been used as constraint tightness in more recent research [Prosser 94] [Smith 94]. Due to the popularity of the more recent interpretation, we have decided to adopt that definition for the sake of uniformity in the field.

(FC-CBJ) was also included.[11] The major difference between FC-BM2 and FC-BM2-CBJ and Prosser's FC-BM and FC-BM-CBJ is that a 2-dimensional *mbl* is used for the former algorithms.[12] For brevity, we do not include the details of FC-BM2 and FC-BM2-CBJ in this paper. However the algorithms have similar structures as their "lazy" counterparts except that their "laziness" is taken away.

The variable ordering heuristics we used included the minimal width ordering heuristic (MWO) [Freuder 82] and the fail-first heuristic (FF) [Haralick & Elliott 80]. MWO was used instead of a random ordering because it also provides a static ordering. Without using any heuristic, the time needed for the experiments could be extremely long. Unless otherwise stated, MWO is assumed in the following results.

5.1 Varying the number of variables

In the first series of experiments, CSPs consists of different number of variables were generated and tested. The performance of the tested algorithms in terms of the average number of compatibility checks in solving the problems characterised by <10, 5, 0.5, *> and <20, 5, 0.5, *> are shown in Figures 5 and 6.

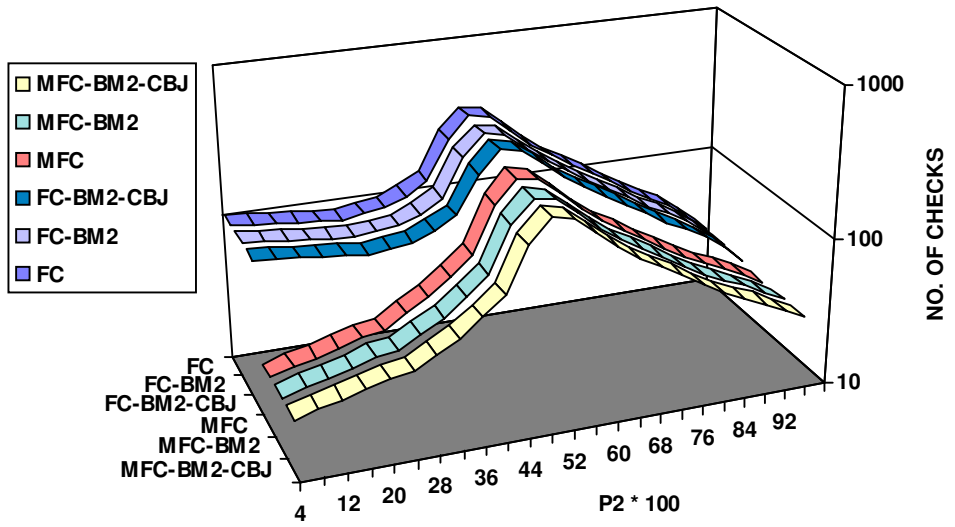

Figure 5. Average number of compatibility checks performed by FC, the lazy algorithms and their "non-lazy" counterparts with MWO for CSPs characterised by <10, 5, 0.5, *>

[11] Interestingly FC-CBJ was found to have a similar performance as FC-BM2 in most of our experiments when MWO was used. For brevity, we skipped the results on FC-CBJ except for those in Table 13.
[12] Again the *victim* data structure used in FC-BM-CBJ is avoided in FC-BM2-CBJ.

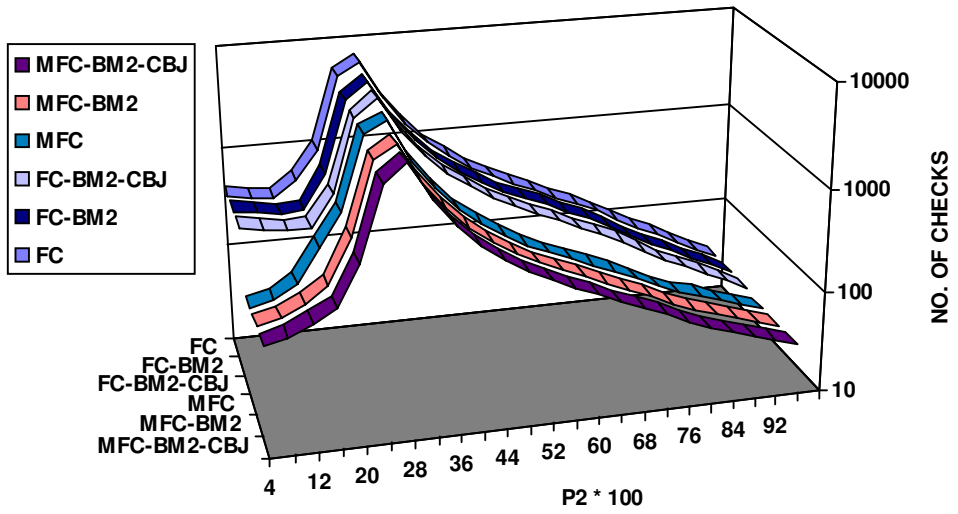

Figure 6. Average number of compatibility checks performed by FC, the lazy algorithms and their "non-lazy" counterparts with MWO for CSPs characterised by <20, 5, 0.5, *>

Figures 5 and 6 suggest that the lazy algorithms outperform their "non-lazy" counterparts which are in turn superior to FC *on average*. The improvement is rather clear. For example, MFC-BM2 performs about 73.8 percent of the compatibility checks done by FC on those problems characterised by <10, 5, 0.5, 0.52>. For the problems characterised by <20, 5, 0.5, 0.28>, MFC-BM2 performs about 58.2 percent of the compatibility checks done by FC. Note that these two data points are predicted to be near to phase transitions according to Smith's predictor [Smith 94].

Another point worth mentioning is that from Figures 5 and 6, it is clear that the regular FC based algorithms perform more compatibility checks as constraint tightness decreases from one until the phase transition point [Cheeseman *et. al.* 91] [Williams & Hogg 94], then the search effort decreases when constraints are further loosened. However, when constraints are extremely loose, the search efforts increase again. It is because when constraints are very loose, the cost of constraint propagation exceeds the benefit that it might be able to gain as few or even no values are removed from the future variables' domains. For example, for problems characterised by <20, 5, 0.5, *>, FC performed 536.1, 370.1, 369.9 and 435.2 compatibility checks on average when $p2$ was set to 0.84, 0.88, 0.92 and 0.96 respectively. However, due to delayed evaluation, this anomaly does not happen to the lazy algorithms. MFC-BM2-CBJ performed 217.7, 158.5, 121.7 and 107 compatibility checks on average over the same sets of problems. The lazy algorithms have a simple "low-high-low" profile as $p2$ is varied from one end to the other.

Figures 7 and 8 shows the performance of the five algorithms in solving the CSPs characterised by <30, 5, 0.5, *> and <40, 5, 0.5, *> respectively.

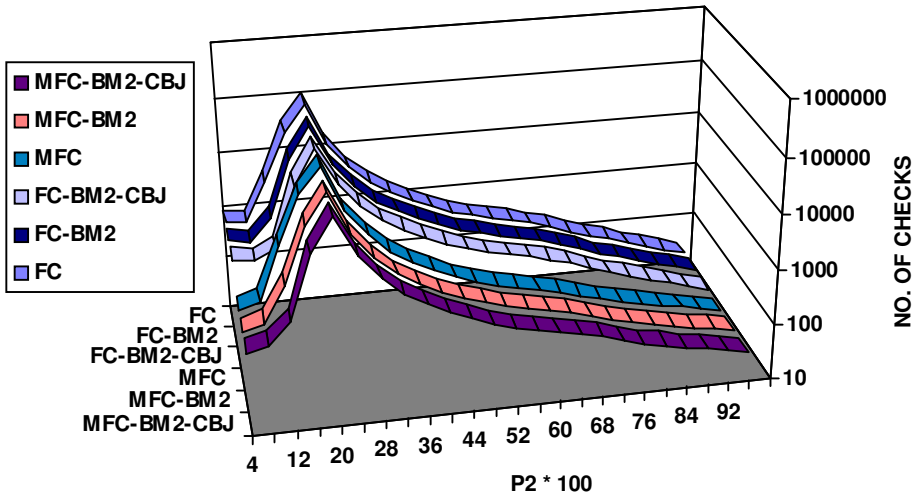

Figure 7. Average number of compatibility checks performed by FC, the lazy algorithms and their "non-lazy" counterparts with MWO for CSPs characterised by <30, 5, 0.5, *>

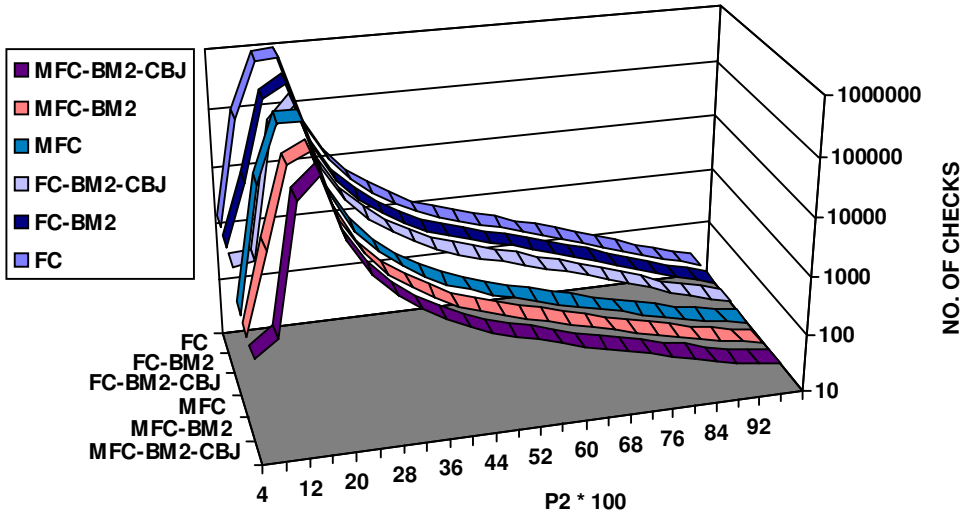

Figure 8. Average number of compatibility checks performed by FC, the lazy algorithms and their "non-lazy" counterparts with MWO for CSPs characterised by <40, 5, 0.5, *>

5.2 Varying domain size

The second series of experiments was done to investigate the effectiveness of the lazy algorithms compared to FC and MFC for solving problems of different domain sizes. In order to compare our results with some corresponding results using MFC (Figure 7 in [Dent & Mercer 94]), we computed the percentage of FC's compatibility checks performed by MFC, MFC-BM2 and MFC-BM2-CBJ near to the predicted crossover points. They are specified in the first, the second and the third rows in each entry respectively in Table 9. The parenthesised values in italics are the $p2$ values in percentage.

In order to contrast our results with those described in [Dent & Mercer 94], geometric mean was used instead of arithmetic mean.

	$p1 = 0.2$			$p1 = 0.5$			$p1 = 0.8$		
n	$m = 5$	$m = 10$	$m = 20$	$m = 5$	$m = 10$	$m = 20$	$m = 5$	$m = 10$	$m = 20$
10	86.8 69.5 68.6 (*84*)	83.3 66.7 65.4 (*92*)	81.9 66.3 64.8 (*96*)	76.1 72.2 71.4 (*52*)	71.5 69.4 68.7 (*64*)	65.3 63.4 63.1 (*76*)	75.6 73.2 72.3 (*36*)	71.8 69.7 69.1 (*48*)	68.3 66.9 66.6 (*56*)
20	71.8 64.4 62.0 (*56*)	62.3 57.7 57.0 (*72*)	57.4 53.6 52.9 (*80*)	67.6 57.9 54.1 (*28*)	57.7 53.3 52.3 (*40*)	52.1 48.4 47.8 (*48*)	66.7 56.8 54.4 (*20*)	60.5 54.2 53.2 (*28*)	57.6 52.5 51.7 (*32*)
30	65.1 52.5 44.9 (*40*)	53.8 47.4 45.6 (*56*)	--	63.0 48.3 42.3 (*20*)	54.5 45.2 42.9 (*28*)	--	64.6 44.1 36.8 (*12*)	54.9 44.8 43.2 (*20*)	--
40	61.6 44.2 35.8 (*32*)	--	--	58.8 40.7 34.4 (*16*)	--	--	63.6 32.1 20.2 (*8*)	--	--

Table 9. Percentage of FC's compatibility checks performed by MFC, MFC-BM2 and MFC-BM2-CBJ with MWO heuristic (based on geometric mean)

The results suggest that the efficiency gain of the three lazy algorithms over FC increases with the number of variables. For MFC, it appears that increasing a problem's domain size improves its edge over FC. This is consistent with the observation reported in [Dent & Mercer 94] despite no variable ordering heuristic was used in their research. MFC-BM2 and MFC-BM2-CBJ also produced results with similar trend for problems with $n \leq 20$. However the same trend cannot be observed for larger n. Concerning the efficiency of the lazy algorithms with respect to $p1$, no obvious trend is observed.

Figures 10 and 11 show the performance of the tested algorithms in solving the problems characterised by <20, 10, 0.5, *> and <20, 20, 0.5, *> respectively. Note that the lower peak of the FC and MFC curves in Figure 10 appeared at <20, 10, 0.5, 0.24> is due to a problem which is computationally hard to them, but not to the others algorithms.

By examining Figures 6, 10 and 11, we observe that the lazy algorithms performed about 100 compatibility checks on average when $p2$ was set to 0.04 for m = 5, 10, 20. However, the "non-lazy" counterparts performed about 450, 850 and 1700 compatibility checks for m = 5, 10, 20 respectively. The performance of the lazy algorithms is very alike disregard the domain sizes when constraints are extremely loose. For example, when n, $p1$ and $p2$ were set to 20, 0.5 and 0.04 respectively, both MFC-BM2 and MFC-BM2-CBJ performed about 100 compatibility checks for m = 5, 10, 20. When $p2$ was set to 0.16, both algorithms performed about 200 compatibility checks for m = 5, 10, 20. This supports the conjecture that unlike the regular FC based algorithms, large domain size is not a burden to lazy algorithms when they are used to solve loosely constrained problems.

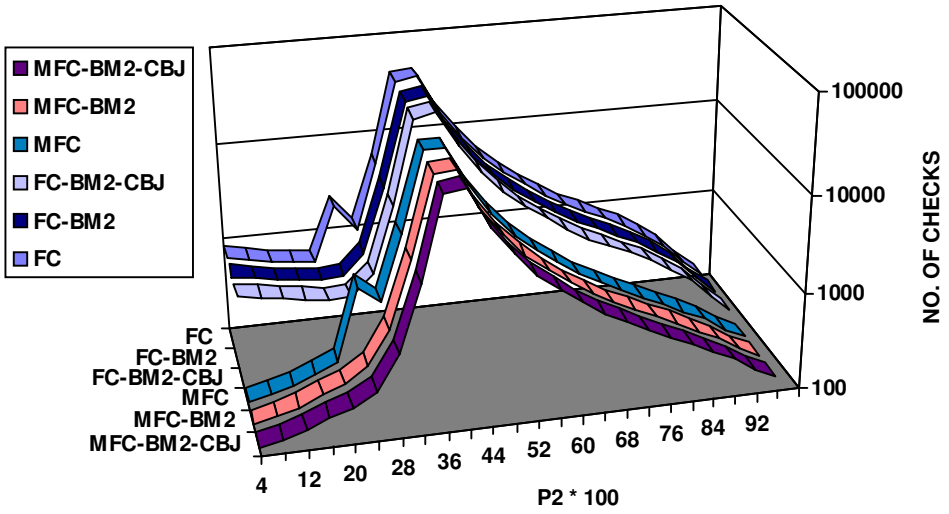

Figure 10. Average number of compatibility checks performed by FC, the lazy algorithms and their "non-lazy" counterparts with MWO for CSPs characterised by <20, 10, 0.5, *>

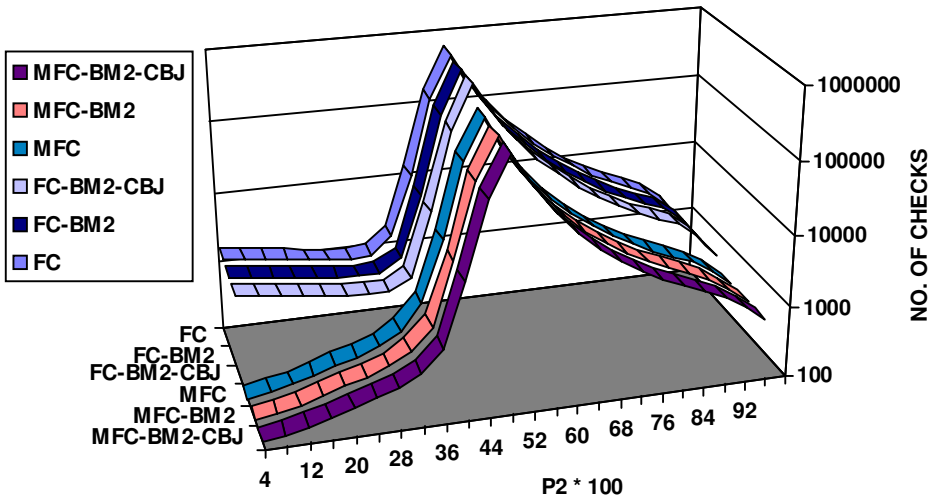

Figure 11. Average number of compatibility checks performed by FC, the lazy algorithms and their "non-lazy" counterparts with MWO for CSPs characterised by <20, 20, 0.5, *>

5.3 Efficiency gain of the lazy algorithms over FC for CSPs near to phase transitions

Table 12 displays the average number of compatibility checks performed by the three lazy algorithms for problems close to the crossover point. There are four rows in each entry. The first numbers in the first, the second and the third rows are the average number of compatibility checks performed by MFC, MFC-BM2 and MFC-BM2-CBJ respectively. The parenthesised number in the second row represents the percentage of the number of checks MFC-BM2 performed when compared to that of MFC whereas the parenthesised number in the third row represents the percentage of the number of checks MFC-BM2-

CBJ performed when compared to that of MFC-BM2. The bottom parenthesised figure in italics is the $p2$ value in percentage of the tested problems. For example, the top leftmost entry in the table says MFC, MFC-BM2 and MFC-BM2-CBJ performed on average 64, 51 and 50 checks on the problems characterised by <10, 5, 0.2, 0.84>. For this set of problems, MFC-BM2 performed about 80 percent of the checked done by MFC whereas MFC-BM2-CBJ performed about 98 percent of the checked done by MFC-BM2 on average.

	$p1 = 0.2$			$p1 = 0.5$			$p1 = 0.8$		
n	$m = 5$	$m = 10$	$m = 20$	$m = 5$	$m = 10$	$m = 20$	$m = 5$	$m = 10$	$m = 20$
10	64 51 (79.7) 50 (98.4) *(84)*	235 185 (78.7) 180 (97.4) *(92)*	853 675 (79.2) 655 (97.1) *(96)*	254 241 (94.8) 238 (98.9) *(52)*	1031 998 (96.8) 987 (98.9) *(64)*	3817 3684 (96.5) 3666 (99.5) *(76)*	660 632 (95.7) 623 (98.6) *(36)*	3170 3073 (97.0) 3043 (99.0) *(48)*	15478 15113 (97.6) 15047 (99.6) *(56)*
20	372 331 (89.2) 316 (95.3) *(56)*	1001 926 (92.4) 913 (98.6) *(72)*	7082 6520 (92.1) 6446 (98.9) *(80)*	4742 4062 (85.7) 3785 (93.2) *(28)*	26119 24131 (92.4) 23688 (98.2) *(40)*	323043 303290 (93.9) 299818 (98.9) *(48)*	13778 11730 (85.1) 11222 (95.7) *(20)*	192242 172105 (89.5) 169156 (98.3) *(28)*	5145260 4702289 (91.4) 4637407 (98.6) *(32)*
30	4358 3404 (78.1) 2857 (83.9) *(40)*	12276 10633 (86.6) 10167 (95.6) *(56)*	--	68618 51748 (75.4) 44909 (86.8) *(20)*	1505133 1236145 (82.1) 1174192 (95.0) *(28)*	--	670689 454208 (67.7) 375919 (82.8) *(12)*	7083326 5765551 (81.4) 5553883 (96.3) *(20)*	--
40	39168 26546 (67.8) 20572 (77.5) *(32)*	--	--	575749 387939 (67.4) 321429 (82.9) *(16)*	--	--	3103602 1524010 (49.1) 962974 (63.2) *(8)*	--	--

Table 12. Performance of MFC, MFC-BM2 and MFC-BM2-CBJ for problems close to the predicted crossover point

We can observe from Table 12 that MFC-BM2 and MFC-BM2-CBJ do not show significant improvements over MFC for small problems, particularly when n is relatively small (say not greater than 20). The new algorithms show their benefits when n is getting larger. Similarly, the superiority of MFC-BM2-CBJ over MFC-BM2 shows more clearly for problems with relatively large n (say $n \geq 30$), particularly for problems with a small domain size (say $m = 5$). By examining the statistics resulted from the Wilcoxon's signed rank test[13], we have a 95 percent confident level that MFC-BM2-CBJ performed a smaller median number of compatibility checks than MFC-BM2 for almost all the sampled data points within the *mushy regions*.[14] Similar inference can be drawn on the algorithms for problems around the mushy regions except for those with both small n and $p1$.

[13] We selected a non-parametric statistic for analysing the experimental results as there is evidence that normality assumption is often violated in CSP research [Kwan 95].

[14] Two exceptions were found at which the problems are characterised by <10, 5, 20, 68> and <10, 10, 20, 84>. However MFC-BM2-CBJ still performed on average less compatibility checks than MFC-BM2 over these two data points.

5.4 How often one algorithm is better than another

A comparison of algorithms with MWO based on the frequency that one algorithm beats another algorithm is given in Table 13. There are three pairs of numbers (in three rows) in each table entry. The first pair is the number of occasions and the percentage that *Algorithm1* outperformed *Algorithm2* out of the 32400 tested problems respectively. The second pair is for *Algorithm2* being the better one whereas the third pair is for ties. For example, the top leftmost entry in Table 13 says that there are 7,565 (23.35 percent) cases that MFC-BM2-CBJ outperformed MFC-BM2, 17 (0.05 percent) cases that the converse is true, and 24,818 (76.6 percent) cases that both performed the same of compatibility checks.

	Algorithm2					
Algorithm1	MFC-BM2	MFC	FC	FC-CBJ	FC-BM2	FC-BM2-CBJ
MFC-BM2-CBJ	7565 (23.35%) 17 (0.05%) 24818 (76.60%)	10864 (33.53%) 43 (0.13%) 21493 (66.34%)	32294 (99.67%) 0 106 (0.33%)	32293 (99.67%) 1 (~ 0) 106 (0.33%)	32224 (99.46%) 1 (~ 0) 175 (0.54%)	32225 (99.46%) 0 175 (0.54%)
FC-BM2-CBJ	236 (0.73%) 31988 (98.73%) 176 (0.54%)	1083 (3.34%) 31198 (96.29%) 119 (0.37%)	11159 (34.44%) 1 (~ 0) 21240 (65.56%)	10811 (33.37%) 4 (0.01%) 21585 (66.62%)	4781 (14.76%) 68 (0.21%) 27551 (85.03%)	
FC-BM2	0 32225 (99.46%) 175 (0.54%)	738 (2.28%) 31544 (97.36%) 118 (0.36%)	11136 (34.37%) 0 21264 (65.63%)	9765 (30.14%) 1193 (3.68%) 21442 (66.18%)		
FC-CBJ	139 (0.43%) 32155 (99.24%) 106 (0.33%)	576 (1.78%) 31646 (97.67%) 178 (0.55%)	6700 (20.68%) 0 25700 (79.32%)			
FC	0 32294 (99.67%) 106 (0.33%)	0 32226 (99.46%) 174 (0.54 %)				
MFC	45 (0.14%) 10641 (32.84%) 21714 (67.02%)					

Table 13. How often *Algorithm1* performs better than *Algorithm2* (both using MWO heuristic) in terms of number of compatibility checks

We conjecture that MFC-BM2 and MFC-BM2-CBJ should never perform more compatibility checks than FC. This conjecture is supported by the results in Table 13. From the first row and the first column of Table 13, we notice that the two new lazy algorithms almost always perform less compatibility checks than the regular FC-based algorithms when using MWO.

Examining the first entry of the second row and the second entry of the fourth row, it appears to suggest that the benefit of incorporating laziness to a FC based algorithm is greater than that offered by CBJ.

Both new lazy algorithms outperformed MFC in about one third of the problems whereas MFC-BM2-CBJ achieved a better performance than MFC-BM2 in about one fourth of the problems.

Table 13 also reveals that MFC-BM2 never performs worse than FC-BM2 and MFC-BM2-CBJ never performs worse than FC-BM2-CBJ among the problems we tested. There were cases at which MFC-BM2-CBJ was beaten by MFC-BM2. MFC-BM2-CBJ

was beaten by FC-BM2 in one occasion. That could be due to the harmful interaction between domain filtering and intelligent backjumping as mentioned in [Prosser 93a]. One counter-intuitive result is that there are occasions at which MFC-BM2 and MFC-BM2-CBJ were beaten by MFC. This is due to the harmful interaction between domain filtering and delayed evaluation. A detailed discussion on this is given in Section 6.

5.5 Choice of variable ordering heuristics

The last series of experiments was conducted to attempt to identify a good variable ordering heuristic for the lazy algorithms. Previous experimental results described in [Dent & Mercer 94] suggest that FF for MFC is not as good as in FC for large problems. They do not provide any hint on whether some other variable ordering heuristic would be a good choice. Furthermore, Dent and Mercer only experimented on problems near to crossover points. Would FF or other variable ordering heuristics be good enough for the lazy algorithms to outperform FC with FF for problems away from the phase transitions? We attempted to answer these questions.

To study whether MWO is more suitable for lazy algorithms than FF for problems near phase transitions, we counted the number of occasions that one algorithm-heuristic combination outperformed another at the corresponding data points. Results based on MFC-BM2-CBJ are summarised in Table 14.[15] Each entry has three numbers. The parenthesised numbers in italics are the $p2$ values in percentage. The first number in the first row of an entry represent the number of times that MWO is a better heuristic out of the 50 problems at that data point. The second number is for FF being the better one. For example, for problems characterised by <10, 5, 0.2, 0.84>, *i.e.*, the top leftmost entry, out of the 50 problems, MFC-BM2-CBJ with MWO heuristic performed less compatibility checks than with FF heuristic in 28 cases. FF is a better heuristic in 17 cases. There are five cases that the two algorithm-heuristic combinations performed the same number of compatibility checks.

	$p1 = 0.2$			$p1 = 0.5$			$p1 = 0.8$		
n	$m = 5$	$m = 10$	$m = 20$	$m = 5$	$m = 10$	$m = 20$	$m = 5$	$m = 10$	$m = 20$
10	28/17 *(84)*	29/14 *(92)*	28/19 *(96)*	29/21 *(52)*	35/15 *(64)*	45/5 *(76)*	23/27 *(36)*	31/19 *(48)*	38/12 *(56)*
20	38/12 *(56)*	47/3 *(72)*	48/2 *(80)*	39/11 *(28)*	50/0 *(40)*	50/0 *(48)*	9/41 *(20)*	24/26 *(28)*	42/8 *(32)*
30	45/5 *(40)*	50/0 *(56)*	--	23/27 *(20)*	49/1 *(28)*	--	1/49 *(12)*	9/41 *(20)*	--
40	46/4 *(32)*	--	--	28/22 *(16)*	--	--	13/37 *(8)*	--	--

Table 14. Number of cases that MWO/FF is a better variable ordering heuristic than the other for MFC-BM2-CBJ

[15] The results based on MFC-BM2 are similar to that of MFC-BM2-CBJ and thus is skipped for brevity.

It appears that MWO is a better heuristic than FF for MFC-BM2-CBJ in many cases, except for high $p1$. MWO seems to work, at least as good as FF, for small problems and its edge over FF is more obvious as m increases. When $p1$ is high, the constraint graph is so dense that MWO heuristic which only exploits on the topology of constraint graph might not be able to offer good 'guesses'. In fact it degenerates into a random ordering when $p1$ is one. This might be the reason why the relative performance of MWO, when compared to FF, drops for problems with high $p1$.

Next, we compare the performance of FC with FF and MFC-BM2-CBJ with both FF and MWO. Figures 15, 16 and 17 show the results for problems characterised by <40, 5, 0.2, *>, <40, 5, 0.5, *> and <40, 5, 0.8, *> respectively. Such a comparison is more sensible from a user point of view as FF is usually used with FC in practice, not MWO.

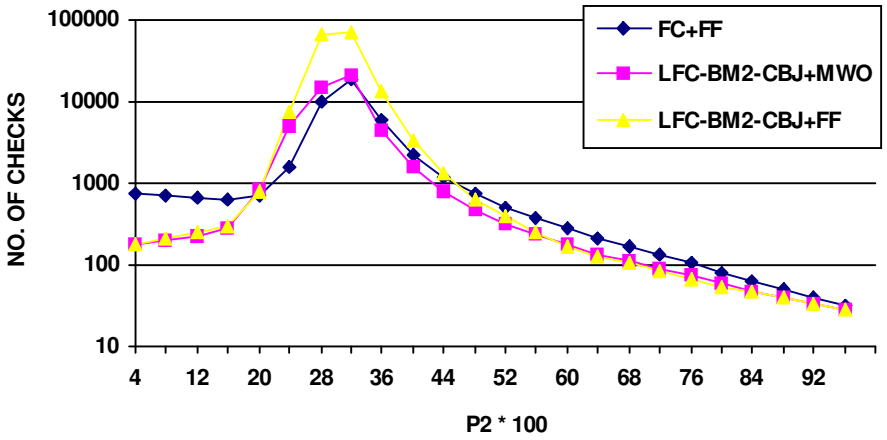

Figure 15. Average number of compatibility checks performed by FC with FF and MFC-BM2-CBJ with MWO and FF for CSPs characterised by <40, 5, 0.2, *>

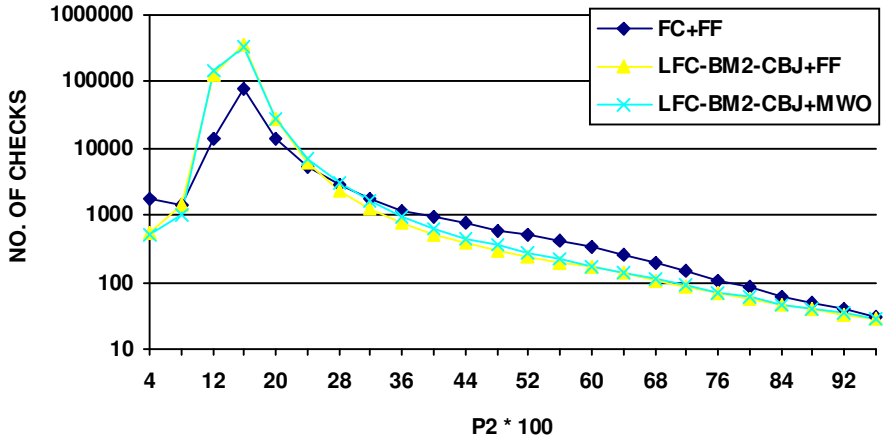

Figure 16. Average number of compatibility checks performed by FC with FF and MFC-BM2-CBJ with MWO and FF for CSPs characterised by <40, 5, 0.5, *>

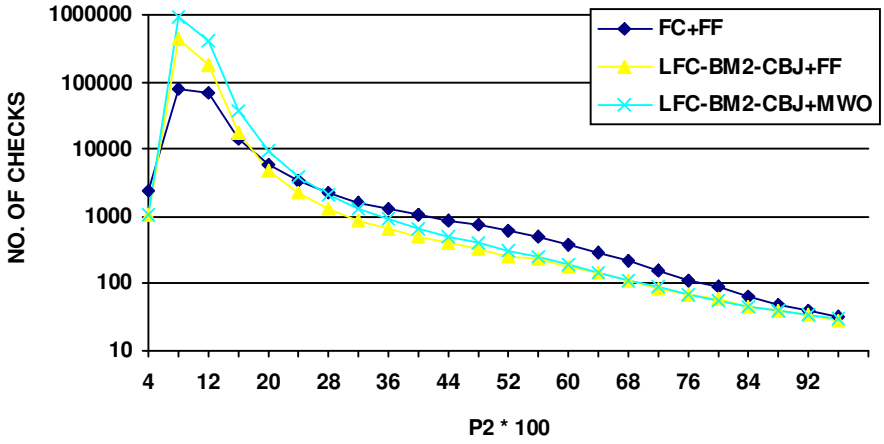

Figure 17. Average number of compatibility checks performed by FC with FF and MFC-BM2-CBJ with MWO and FF for CSPs characterised by <40, 5, 0.8, *>

Dent and Mercer suggest that the performance of MFC with FF, when compared with FC with FF, appears to worsen as the problem sizes grow. That is consistent with the results of our lazy algorithms for problems near phase transition. However, MFC-BM2-CBJ outperforms FC, with either FF or MWO heuristic, for easy problems. As mentioned before, for MFC-BM2-CBJ, MWO appears to work well when constraint density is low whereas FF seems to be a better choice when constraint density is high. Despite the figures show that FC with FF heuristic is quite often the "winner" for problems which are near to phase transition, the lazy algorithms with MWO heuristic could beat FC with FF when $p1$ is low, particularly when n-to-m ratio is small, say not greater than one. Figure 18 illustrates such a case for CSPs characterised by <20, 20, 0.2, *>.

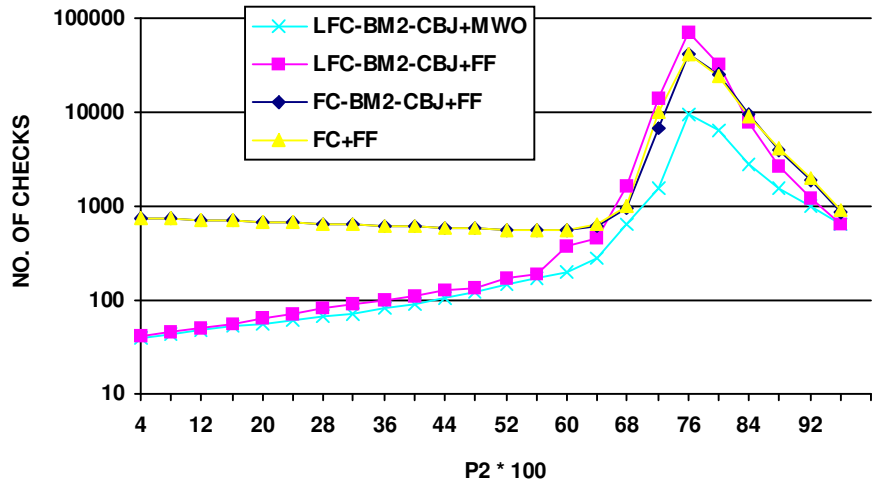

Figure 18. Average number of checks performed by FC and FC-BM2-CBJ with FF and MFC-BM2-CBJ with FF and MWO for CSPs characterised by <20, 20, 0.2, *>

MFC-BM2-CBJ with MWO outperformed FC with FF on average according to Figure 18. In order to demonstrate the difference is not due to MFC-BM2-CBJ's capability of backjumping intelligently, the results of FC-BM2-CBJ with FF are also displayed. Both FC-BM2-CBJ and FC, when combined with FF, have similar performance in solving that set of CSPs.

Although our results are not sufficient to give a clear guideline of which heuristic to use, at least they suggest that static variable ordering heuristics such as MWO should not be ruled out as a candidate heuristic for the lazy algorithms.

6 Domain filtering can degrade performance of MFC

As described in Table 13, there are a few dozens cases in our experiments that MFC outperformed MFC-BM2 and MFC-BM2-CBJ. One possible reason why MFC performs less compatibility checks than MFC-BM2-CBJ is that there exist harmful interactions between domain filtering and intelligent backjumping which is first described in [Prosser 93a]. However this does not explain why MFC-BM2 could perform worse than MFC as MFC-BM2 backtrack chronologically. The counter-intuitive phenomenon is due to the harmful interaction between domain filtering and lazy evaluation. This can be illustrated by the following example.

Suppose there is a CSP with four variables, namely X_1, X_2, X_3 and X_4, and their domains are as follows: $D_1 = D_4 = \{1\}$; $D_2 = \{1, 2\}$ and $D_3 = \{1, 2, 3\}$. Let $C_{i,j}$ denote the constraint between X_i and X_j, which is represented as a set of compatible binary labels of the constrained variables. The constraints of the problems are:

$C_{1,4} = \{ (<X_1, 1>, <X_4, 1>) \}$;
$C_{2,3} = \{ (<X_2, 1>, <X_3, 1>), (<X_2, 2>, <X_3, 1>), (<X_2, 2>, <X_3, 3>) \}$;
$C_{2,4} = \{ (<X_2, 1>, <X_4, 1>) \}$; and
$C_{3,4} = \{ (<X_3, 2>, <X_4, 1>) \}$.

Figures 19 and 20 display the traces of the execution of MFC and MFC-BM2 on the above problem respectively. Note that we assume the variables are labelled in a lexicographic ordering.

MFC and MFC-BM2 take eight and nine compatibility checks to complete the search respectively. The first five steps performed by both algorithms are the same except that MFC-BM2 permanently removes "1" from the domain of X_3 at the end of step 3 as $<X_3, 1>$ is the only culprit which causes a domain "wipe-out" of X_4. In step 6 (Figure 19), after confirming the compatibility between $<X_2, 2>$ and $<X_3, 1>$, MFC checks $<X_2, 2>$ against $<X_4, 1>$ and finds them incompatible. As the domain of X_4 has been exhausted, MFC tries to label X_2 with an alternative value but there is none left. The search then backtracks in order to find another label for X_1. As there is no more untried value D_1, MFC terminates.

Step	D_1	D_2	D_3	D_4	Checks
0	1	1 2	1 2 3	1	0
1	$X_1 = 1$	1 2	1 2 3	1 (X_1 $\surd$)	1
2	$X_1 = 1$	$X_2 = 1$ 2	1 (X_2 $\surd$) 2 3	1 (X_1, X_2 $\surd$)	2
3	$X_1 = 1$	$X_2 = 1$ 2	$X_3 = 1$ 2 3	1 (X_1, X_2, X_3 $\times$)	1
4	$X_1 = 1$	$X_2 = 1$ 2	2 (X_2 $\times$) 3	1 (X_1, X_2)	1
5	$X_1 = 1$	$X_2 = 1$ 2	3 (X_2 $\times$)	1 (X_1, X_2)	1
6	$X_1 = 1$	$X_2 = 2$	1 (X_2 $\surd$) 2 3	1 (X_1, X_2 $\times$)	2
Total					8

Figure 19. Execution of minimal forward checking

Step	D_1	D_2	D_3	D_4	Checks
0	1	1 2	1 2 3	1	0
1	$X_1 = 1$	1 2	1 2 3	1 (X_1 $\surd$)	1
2	$X_1 = 1$	$X_2 = 1$ 2	1 (X_2 $\surd$) 2 3	1 (X_1, X_2 $\surd$)	2
3	$X_1 = 1$	$X_2 = 1$ 2	$X_3 = 1$ 2 3	1 (X_1, X_2, X_3 $\times$)	1
4	$X_1 = 1$	$X_2 = 1$ 2	1 (pruned) 2 (X_2 $\times$) 3	1 (X_1, X_2)	1
5	$X_1 = 1$	$X_2 = 1$ 2	3 (X_2 $\times$)	1 (X_1, X_2)	1
6	$X_1 = 1$	$X_2 = 2$	2 (X_2 $\times$) 3	1 (X_1)	1
7	$X_1 = 1$	$X_2 = 2$	3 (X_2 $\surd$)	1 (X_1, X_2 $\times$)	2
Total					9

Figure 20. Execution of MFC-BM2 for illustrating the fact that domain filtering can cause performance degradation for lazy algorithms

MFC-BM2 proceeds its search in step 6 of Figure 20 in another way. As “1” has been filtered from the domain of X_3 at the end of step 3, MFC-BM2 checks the compatibility between $<X_2, 2>$ and $<X_3, 2>$. They are incompatible to one another. However the remaining value "3" in the current domain of X_3 is found compatible with $<X_2, 2>$ in step 7. The search proceeds by checking the compatibility between $<X_2, 2>$ and $<X_4, 1>$ and then realises that they are incompatible. As the domain of X_4 has been annihilated, the search terminates after finding no more untried values in X_2, and X_1.[16]

The example problem illustrates that domain filtering in a lazy algorithm can counter-intuitively delay detection of inconsistency which in turn cause a performance degradation.[17] Note that FC-BM2 may also filter some of its variables' domain values during search. However, as it does not make use of delayed evaluation, domain filtering does not cause any performance degradation. This ensures FC-BM2 a superiority over FC when given the same variable and value orderings, which is consistent with our experimental results as summarised in Table 13.

7 Conclusions

Our work borrows three ideas from previous research, namely the delayed evaluation in forward checking, a 2-dimensional data structure for storing minimum backup level, and the hybrid algorithm of forward checking with backmarking (and conflict-directed backjumping). Synthesising the ideas, two new hybrid algorithms have been constructed. They are useful in several aspects. Firstly, unlike FC, large domain size of variables is no longer a problem when delayed evaluation is adopted. Besides, it does not suffer from the anomaly of FC in solving loosely constrained problems. Second, they extend MFC by incorporating backmarking and conflict-directed backjumping into it. Thirdly, unlike minimal forward checking which has a space complexity of $O(N^2M)$, our algorithms have a space complexity of $O(NM)$.

Despite the empirical results show our new algorithms on average outperform the regular FC based algorithms given the same variable and value orderings, the situation could be changed entirely when dynamic variable ordering is taken into consideration. In particular, Haralick & Elliott's type of FF heuristic works extremely well with forward checking hybrids [Haralick & Elliott 80] [Bacchus & van Run 95] but delayed evaluation appears to make the heuristic less useful for lazy forward checking based algorithms for some CSPs. It is because FF in lazy FC algorithms is often less accurate than FF in regular FC in estimating the true number of remaining values in the future variables. Our experimental results show that FF appears not to be a good choice for our algorithms when a problem's constraint density is low. Interestingly, MWO seems to be a better partner with the lazy algorithms for that type of problems.

[16] "3" is also removed from D_3 at the end of step 7 in Figure 20 as it is the only culprit for the domain "wipe-out" of X_4.

[17] This could be another reason why MFC outperformed MFC-BM2-CBJ in some of our tested problems.

One way to ensure the superiority of a lazy FC based algorithm over regular FC is to find the smallest domain variable by propagating more constraints forward before FF is applied to pick the next variable [Dent & Mercer 96]. A similar idea has also been mentioned in [Bacchus & Grove 95]. However, such a modification requires extra efforts in constraint propagation and further experiments need to be conducted in order to investigate for what type of problems would the extra efforts be justified.

Our experiments on the lazy algorithms also led to a very interesting finding: domain filtering can degrade performance of chronological backtrackers. This is contrary to the following conjecture mentioned in [Prosser 93a]:

> "We should now assume that increased consistency, or the removal of redundancies, can only guarantee a reduction in search effort if that search is unintelligent (such as a chronological backtracker)."

We have illustrated that domain filtering could degrade the performance of a lazy FC chronological backtracker, due to its harmful interaction with the delayed evaluation mechanism, even under a static variable ordering. This once again demonstrates our limited understanding on the complicated interactions between different components in hybridized CSP techniques.

Acknowledgements

The authors would like to thank Michael Dent for his useful discussion. Thanks are also given to James Borrett who has lengthy discussion with us on this research. The research reported in this paper was supported by the EPSRC research grant ref. GR/J/42878 and the University of Essex research promotion fund ref. R7207.

References

[Bacchus & van Run 95]
Bacchus, F. & van Run, P. 1995. Dynamic Variable Ordering in CSPs. In *Proceedings the First International Conference on Principle and Practice of Constraint Programming*, 258-275.

[Bacchus & Grove 95]
Bacchus, F. & Grove, A. 1995. On the Forward Checking Algorithm. In *Proceedings the First International Conference on Principle and Practice of Constraint Programming*, 292-309.

[Cheeseman et. al 91]
Cheeseman, P., Kanefsky, B. & Taylor, W.M. Where the Really Hard Problems Are. In *Proceedings IJCAI-91*, 331-337, 1991.

[Dent & Mercer 94]
Dent, M. & Mercer, R.E. 1994. Minimal Forward Checking. In *Proceedings 6th IEEE International Conference on Tools with Artificial Intelligence*.

[Dent & Mercer 96]
Dent, M. & Mercer, R.E. 1996. Improvements to the Minimal Forward Checking Algorithm. Techreport UWO-CSD-458, Department of Computer Science, University of Western Ontario, Canada.

[Freuder 82]
Freuder, E. 1982. A Sufficient Condition for Backtrack-Free Search. *J. ACM* 29, 24-32.

[Gaschnig 77]
Gaschnig, J. 1977. A General Backtrack Algorithm that Eliminates Most Redundant Tests. In *Proceedings IJCAI-77*, 457.

[Haralick & Elliott 80]
Haralick, R.M. & Elliott, G.L. 1980. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence* 14, 263-314.

[Kondrak & van Beek 95]
Kondrak, G. & van Beek, P. 1995. *A Theoretical Evaluation of Selected Backtracking Algorithms*. In *Proceedings IJCAI-95*, 541-547.

[Kwan 95]
Kwan, A.C.M. 1995. The Validity of Normality Assumption in CSP Research. Technical Report CSM-261, Department of Computer Science, University of Essex, U.K. (Accepted for *PRICAI-96*)

[Nadel 89]
Nadel, B. 1989. Constraint Satisfaction Algorithms. *Computational Intelligence* (5), 188-224.

[Nudel 83]
Nudel, B. 1983. Consistent Labeling Problems and their Algorithms: Expected Complexities and Theory-based Heuristics. *Artificial Intelligence* 21, 135-278.

[Prosser 93]
Prosser, P. 1993. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence* 9(3), 268-299.

[Prosser 93a]
Prosser, P. 1993. Domain filtering can degrade intelligent backtracking search. In *Proceedings IJCAI-93*, 262-267.

[Prosser 94]
Prosser, P. 1994. Binary Constraint Satisfaction Problems: Some are Harder than Others. In *Proceedings ECAI-94*, 95-99.

[Prosser 95]
Prosser, P. Forward Checking with Backmarking. In Meyer (ed.): *Constraint Processing*, LNCS 923, Springer-Verlag, 185-204, 1995.

[Smith 94]
Smith, B.M. 1994. Phase Transition and the Mushy Region in Constraint Satisfaction Problems. In *Proceedings ECAI-94*, 100-104.

[Williams & Hogg 94]
Williams, C. & Hogg, T. 1994. Exploiting the Deep Structure of Constraint Problems. *Artificial Intelligence* 70, 73-117.

[Zweben & Eskey 89]
Zweben, M. & Eskey, M. 1989. Constraint Satisfaction with Delayed Evaluation. In *Proceedings IJCAI-89*, 875-880.