Minimal Forward Checking

M. J. Dent

Department of Computer Science
University of Western Ontario
London, ONT N6A 5B7

R. E. Mercer

Department of Computer Science
University of Western Ontario
London, ONT N6A 5B7

Abstract

Forward Checking (FC) is a highly regarded complete search algorithm used to solve Constraint Satisfaction Problems. In this paper a lazy variant of FC called Minimal Forward Checking (MFC) is introduced. MFC is a natural marriage of incremental FC and Backchecking. Given a variable selection heuristic which does not depend on domain size MFC's worst case performance on any CSP instance is the number of constraint checks performed by FC. Experiments using hard random problems are presented which show that MFC outperforms FC especially for problems with large domain sizes and/or a large number of variables.

1 Introduction

Many problems in Artificial Intelligence and Operations Research can be expressed as Constraint Satisfaction Problems (CSPs)[4, 8, 15]. A CSP is represented with a set of variables, a set of finite discrete domains for those variables, and a set of constraints over those variables. In this paper we restrict our attention to binary CSP's where all the constraints are of arity 2. The general problem is to find a satisfying assignment of values to variables under the given constraints. CSP problems are NP-complete. This paper presents the design and empirical analysis of a new CSP search algorithm called Minimal Forward Checking (MFC) which improves on the performance of a very popular CSP search algorithm called Forward Checking (FC).

Many studies have found that FC is a useful algorithm for solving CSPs[5, 3, 7, 15, 12]. FC performs a limited lookahead which is designed to help the backtracking search find and avoid failures earlier. When FC attempts to give a value to a variable it filters *all* values inconsistent with this value from the domains of variables not yet instantiated. If a "future" domain becomes empty then the current attempted instantiation is an inconsistent choice and the filtered values are returned to their respective domains. FC's efficiency is usually attributed to this ability of detecting inconsistencies earlier in the search tree with less arc consistency checking per node than other more complicated arc consistency algorithms[3, 7]. However, FC may not be efficient for problems with larger domain sizes and a large number of variables. Early failures in the search tree may make much of FC's consistency checking redundant[6].

MFC is based on the observation that FC attempts to instantiate a new variable only when there is at least *one* value in each future domain that is consistent with all the variables that have been instantiated. MFC is a lazy version of FC that finds and maintains one consistent value in every future domain, "suspending" forward checks until they are required by the search. In this way MFC avoids searching (possibly large) domains for consistent values unless it has to. This concept is similar to Bessière's idea of maintaining one supporting value in his full arc consistency algorithm AC-6[1].

The cost of incorporating laziness into FC is threefold. First, MFC needs to maintain a temporary record of successful and unsuccessful checks against each domain value. The record of successful checks is needed as MFC does not know which values are "past" consistent as FC does. The record of a constraint check is erased when the variable that caused the constraint check is uninstantiated. FC has a similar record but only in terms of unsuccessful constraint checks. The space complexity of the record for MFC is $O(n^2m)$ and for FC it is $O(nm)$ where n is the number of variables and m is the size of the largest domain. The second cost is the added complexity of code necessary to perform the partial search. If the cost of a constraint check is no more than the cost of a table lookup it may be better to use FC for smaller problems. The overhead of the algorithm would outweigh

1063-6730/94 $4.00 © 1994 IEEE

the usefulness of avoiding constraint checks. The third cost is that MFC partially disables variable selection heuristics which depend on domain size. MFC does not know the true filtered size of the future domains.

The benefit of using MFC is that in the *worst case*, MFC performs the same amount of constraint checking as FC given that both instantiate variables in the same order (i.e. with the same variable selection heuristic not depending on domain size) and that the domains are ordered. There are CSP domains where the variable selection heuristic based on domain size is inappropriate. For example, certain types of scheduling problems[10] and N-ary CSPs. Our experiments show that MFC consistently performs many fewer constraint checks than FC on hard random problems. If the cost of a constraint check is significant then MFC is a better choice.

Section 2 presents an overview and example of the FC and MFC algorithms, section 3 describes experimental results and section 4 gives conclusions and future work. A complete description of the MFC algorithm can be found in [2].

2 Minimal Forward Checking

Assume that the variable **instantiation order**, $v_1, \ldots, v_i, \ldots, v_n$, is the order in which variables are chosen to be given a value. The **current variable**, v_i, is the variable to be instantiated and d_i is the current domain. The instantiated variables $v_1, \ldots, v_{i-1}$ are called the **past variables** and the uninstantiated variables $v_{i+1}, \ldots, v_n$ are called the **future variables**. The **past-connected variables** are the past variables that are connected by a constraint to the current variable v_i, and the **future-connected variables** are the future variables that are connected by a constraint to the current variable v_i. Similar terminology is used to refer to the domains.

In this paper we divide the algorithms into a forward labeling move used to find an instantiation for the current variable and a backward unlabeling move used to undo a formerly successful instantiation. We assume that the two functions are called within the context of a backtracking search. A full description of the FC algorithm is available in a number of papers[3, 7, 12]. The forward labeling move of FC, called fc-label, takes as input the index of the variable to be instantiated and the indices of the future-connected variables. Fc-label searches through the current domain attempting to find an acceptable value for the current variable. At each attempted instantiation it removes and records all values inconsistent with

Step	D_1	D_2	D_3	D_4	Checks
0	r	g o	b g	g b r	0
1	$v_1 = \mathrm{r}$	g (√) o (√)	b (√) g (√)	g (√) b (√) r (×)	7
2	$v_1 = \mathrm{r}$	$v_2 = \mathrm{g}$	b (√) g (×)	g (×) b (√)	4
3	$v_1 = \mathrm{r}$	$v_2 = \mathrm{g}$	$v_3 = \mathrm{b}$	b (×)	1
4	$v_1 = \mathrm{r}$	$v_2 = \mathrm{o}$	b (√) g (√)	g (√) b (√)	4
5	$v_1 = \mathrm{r}$	$v_2 = \mathrm{o}$	$v_3 = \mathrm{b}$	g (√) b (×)	2
6	$v_1 = \mathrm{r}$	$v_2 = \mathrm{o}$	$v_3 = \mathrm{b}$	$v_4 = \mathrm{g}$	0
Total					18

Figure 1: Execution of Forward Checking

the attempted instantiation in the future-connected domains. If a future-connected domain is made empty the forward check is undone by replacing the values removed from the future-connected domains and the next value is considered. If the forward check is successful the current attempted instantiation is acceptable and fc-label returns true. If no value can be successfully instantiated fc-label returns false. The unlabeling move of FC is called when the search can no longer move forward. Fc-unlabel takes as input the index of the last successfully instantiated variable, say v_i, and undoes the forward check previously done for the current value of v_i and removes the value of v_i from the current domain of v_i. The unlabeling move records this removed value as being inconsistent with the value of v_{i-1}. If there are more values to choose from v_i's domain the search can move forward again, otherwise fc-unlabel is called again with the index of v_{i-1}.

Consider the following graph colouring CSP: $D_1 = \{\mathrm{r(ed)}\}$, $D_2 = \{\mathrm{g(reen), o(range)}\}$, $D_3 = \{\mathrm{b(lue), g}\}$, $D_4 = \{\mathrm{g, b, r}\}$, where the constraints restrict pairs of variables from $\{v_1, \ldots, v_4\}$ to be assigned different colours. Figure 1 outlines the search performed by FC. The checkmarks (√) show successful constraint checks and the (×) marks show unsuccessful constraint checks. In step 1, v_1 is assigned the value red and FC goes through the future-connected domains (D_2, D_3, and D_4) looking for inconsistent values. The value red in D_4 is found to be inconsistent and is removed. The search now moves forward as there are consistent values in every future-connected

```
mfc-label(i,past-vars,future-vars)

consistent ← False
FOR v[i] ← EACH ELEMENT OF current-domain[i]
WHILE not consistent DO
  IF past-consistent(i,past-vars) THEN
    consistent ← min-forward-check(i,future-vars,
          past-vars)
    IF not consistent THEN
      undo-min-forward-check(i)
      k ← {index of previous variable}
      remember-unsuccessful-check(k,i)
    ELSE
      k ← {index of previous variable}
      remember-unsuccessful-check(k,i)
  IF not consistent THEN
    current-domain[i] ← rest(current-domain[i])
    previous-checks[i] ← rest(previous-checks[i])
RETURN(consistent)
```

Figure 2: mfc-label

```
past-consistent(i,past-vars)

ok-result ← True
unchecked-past-vars ←
      {calculate past-vars not yet checked against
      current value of i from record
      (in instantiation order)}
FOR m ← EACH ELEMENT OF unchecked-past-vars
WHILE ok-result DO
  ok-result ← check(m,i)
  IF ok-result THEN
    remember-successful-check(m,i)
  ELSE
    remember-unsuccessful-check(m,i)
RETURN(ok-result)
```

Figure 3: past-consistent

domain (step 2). Variable v_2 is assigned the value green and the future-connected domains are checked. The values green in both D_3 and D_4 are inconsistent and are removed. Variable v_3 is assigned the value blue and a forward check is done. FC finds that the value blue in D_4 is inconsistent with the value chosen for v_3. As there are no further elements in D_4 and D_3 fc-unlabel is called to backtrack the search to v_2. The value blue is returned to domain D_4, and the value green is returned to domain D_3 and domain D_4. Variable v_2 is then assigned the value orange (step 4). FC checks the future-connected domains and finds no inconsistent values. In step 5, v_3 is assigned the value blue and FC removes the value blue from the domain of D_4. Finally, step 6 shows the solution. FC performed a total of 18 constraint checks.

MFC mimics the search of FC by maintaining only one value consistent with the past variables in every future domain. If the value being maintained becomes inconsistent with the current attempted instantiation a new value is found that is consistent with the past variables. The incremental nature of MFC implies that a record of both successful and unsuccessful constraint checks must be maintained. MFC records the variables involved in a constraint check, v_i and v_j $(i < j)$, the value in the domain of v_j that was checked against, and the result of the check. This record can be implemented as an array or as a set of assertions or by using list structures. The labeling move for MFC (mfc-label) is very similar to that for FC (see Figure 2). The algorithm is presented using a pseudo-code developed by Nadel and Prosser[7, 12]. Mfc-label takes the index of the current variable to instantiate and the indices of the past-connected and future-connected variables. There are two major differences from fc-label.

The first difference is that the remaining elements in the current domain of v_i other than the first are not guaranteed to be consistent with the past-connected variables and must be tested if the first value is not acceptable. Function past-consistent (see Figure 3) ensures that the current attempted instantiation is consistent with the past-connected variables that have not yet been checked with it. A call to past-consistent has the effect of waking up previously delayed forward checks. Function past-consistent is actually performing Backchecking [3] which is the counterpart to FC in that it performs and remembers checks looking backwards into the search.

The second major difference is that the forward check for MFC, called min-forward-check (see Figure 4), only finds the first consistent value in each future-connected domain. Min-forward-check ensures that the first value in each future-connected domain is consistent with past connected variables for the future domain that it is looking at. If the current first value is past consistent, a check is performed to see if it is consistent with the attempted instantiation for v_i. If it is consistent min-forward-check moves on to the next future-connected domain. If it is not consistent min-forward-check loops and tests the next value in the domain. Min-forward-check returns true if it is able to

```
min-forward-check(i,future-vars,past-vars)

ok-result ← True
FOR k ← EACH ELEMENT OF future-vars
WHILE ok-result DO
  ok-result ← False
  past-vars-k ← {calculate current past-vars for k}
  FOR v[k] ← EACH ELEMENT OF current-domain[k]
  WHILE not ok-result DO
    IF past-consistent(k,past-vars-k) THEN
      ok-result ← check(i,k)
      IF not ok-result THEN
        remember-unsuccessful-check(i,k)
      ELSE
        remember-successful-check(i,k)
    IF not ok-result THEN
      current-domain[k] ← rest(current-domain[k])
      previous-checks[k] ← rest(previous-checks[k])
RETURN(ok-result)
```

Figure 4: min-check-forward

find one past consistent value in each future-connected domain or false otherwise. Mfc-label returns true if it is able to instantiate the current variable, false otherwise. The unlabeling function is very similar to fc-unlabel. When a variable v_i is uninstantiated, the values that were unsuccessfully checked against are replaced in their respective domains and all records of checks against future-connected domains are erased.

Figure 5 outlines the search performed by MFC on our example CSP. Domain values are shown with lists of instantiated variables with which they have been checked. Some variables in the lists have superscripts ($\surd$) and ($\times$) denoting respectively successful and unsuccessful constraint checks performed in the current search step. If a domain value has not been checked, no list is shown. In step 1, v_1 is assigned the value red and a minimal forward check is performed. The first consistent value in each future-connected domain is found (in this case the first value in each of the domains). In step 2, v_2 is assigned the value green and another minimal forward check is performed. The value blue in domain D_3 is consistent with v_2 but the value green in domain D_4 is inconsistent. Min-forward-check searches through D_4 (by unsuspending previous forward checks) searching for a past consistent value (in this case blue) doing the constraint checks in the instantiation order. As there are still consistent values in each future domain, the search moves forward and v_3 is assigned the value blue. However, a minimal forward check shows that no value in domain D_4 is consistent. Value blue is inconsistent with v_3 and an unsuspension of a forward check shows that the value red is inconsistent with v_1. The search backtracks to v_3 and attempts to find another consistent value but the unsuspension of the forward checks for the value green show it to also be inconsistent (step 4). Also in this step notice that domain value blue is returned to domain D_4 as it is no longer inconsistent with v_3. In step 5, the value orange in domain D_2 is found to be past consistent with v_1. In steps 6 and 7 the search moves forward as MFC finds the first value consistent in each future-connected domain. Step 8 shows the solution to the CSP found by MFC. MFC performed 15 constraint checks compared to the 18 that FC performs on the same problem.

MFC mimics FC's search avoiding constraint checks until necessary. When the variable selection strategy depends on domain size or domains are unordered, MFC may perform more constraint checks than FC. However when the variable selection heuristic does not depend on domain size and domains are ordered the following theorem holds.

Theorem 1 *For any CSP, assuming that the variable selection order is the same and that the domains are ordered, Minimal Forward Checking's worst case performance in terms of constraint checks is the number of constraint checks performed by Forward Checking.*

3 Experiments

A series of experiments was performed with randomly generated hard binary CSPs. Each CSP is characterized by a 4-tuple $< n, m, p_1, p_2 >$ where n is the number of variables, m is the size of every domain, p_1 is the probability of a constraint existing between two variables, and p_2 is the probability that a pair of values in a constraint are inconsistent. It has been recently shown in [13, 14] that it is possible to generate random CSP's that are significantly harder than most random instances. The expected number of solutions to a particular CSP can be calculated as:

$$\mathrm{E(Soln)} = m^n(1-p_2)^{n(n-1)p_1/2}$$

Prosser and Smith both conjecture that the hardest random CSP problems occur when the expected number of solutions is 1 (especially as n gets larger). They reason that problems which have an expected number of solutions less than 1 will be over-constrained and therefore easier to prove unsatisfiable and, conversely,

Step	D_1	D_2	D_3	D_4	Checks
0	r	g o	b g	g b r	0
1	$v_1 = \text{r}$	g $\{v_1^{\checkmark}\}$ o	b $\{v_1^{\checkmark}\}$ g	g $\{v_1^{\checkmark}\}$ b r	3
2	$v_1 = \text{r}$	$v_2 = \text{g}$	b $\{v_1, v_2^{\checkmark}\}$ g	g $\{v_1, v_2^{\times}\}$ b $\{v_1^{\checkmark}, v_2^{\checkmark}\}$ r	4
3	$v_1 = \text{r}$	$v_2 = \text{g}$	$v_3 = \text{b}$	b $\{v_1, v_2, v_3^{\times}\}$ r $\{v_1^{\times}\}$	2
4	$v_1 = \text{r}$	$v_2 = \text{g}$	g $\{v_1^{\checkmark}, v_2^{\times}\}$	b $\{v_1, v_2\}$	2
5	$v_1 = \text{r}$	o $\{v_1^{\checkmark}\}$	b $\{v_1\}$ g $\{v_1\}$	g $\{v_1\}$ b $\{v_1\}$	1
6	$v_1 = \text{r}$	$v_2 = \text{o}$	b $\{v_1, v_2^{\checkmark}\}$ g $\{v_1\}$	g $\{v_1, v_2^{\checkmark}\}$ b $\{v_1\}$	2
7	$v_1 = \text{r}$	$v_2 = \text{o}$	$v_3 = \text{b}$	g $\{v_1, v_2, v_3^{\checkmark}\}$ b $\{v_1\}$	1
8	$v_1 = \text{r}$	$v_2 = \text{o}$	$v_3 = \text{b}$	$v_4 = \text{g}$	0
Total					15

Figure 5: Execution of Minimal Forward Checking

problems with an expected number of solutions greater than 1 will be under-constrained and therefore easier to satisfy. Given values for n, m, and p_1, and assuming that the expected number of solutions for a hard problem is 1, one can calculate a value for p_2 from the above equation.

In our experiments we varied n in $\{10,15,20\}$, m in $\{5,10\}$, and p_1 in $\{0.2,0.25,\ldots,1.0\}$. For each setting of the three parameters 20 random CSPs were created in a manner following [13, 14]. To create a random CSP, a graph was created by randomizing an enumeration of all possible edges and taking the first $p_1 n(n-1)/2$ as edges in the random graph. Unlike [13, 14] the graphs were unacceptable if they were not connected (disconnected graphs can be solved separately and are therefore not representative of a problem with n variables). Then, for each pair of variables that were connected by an edge a constraint was formed by randomizing an enumeration of the cross-product of the two domains and taking the first $p_2 m^2$ as unacceptable pairs.

MFC and FC were run on the random problems using both a static (given) variable selection order (MFC-NORM, FC-NORM) and a variable selection order based on smallest domain size (MFC-VAR, FC-VAR). As mentioned in the introduction, the variable selection order based on smallest domain size was not expected to perform very well with MFC as MFC does not know the true size of the future domains.

It is customary to compare CSP algorithms by the number of constraint checks that are performed in solving CSP instances. Constraint checks are an unbounded quantity in that they may only be table lookups or they may be something much more complicated. Timing results are not reliable as they may be changed by different implementations. Many papers comparing CSP algorithms have used either the mean or the median of the number of constraint checks performed over multiple CSP instances to compare algorithms. Both methods have problems with outliers. It is not unusual for an occasional ***hard*** problem to be generated. Using the mean gives too much weight to the outlier (it usually dominates all other instances) and comparisons are meaningless. Using the median more than likely ignores how the algorithms fared on those hard problems. We have chosen to use the geometric mean of the number of constraint checks performed on each instance. The geometric mean is not as susceptible to outliers yet it doesn't entirely discount them either. The geometric mean seems to be a better indicator of average performance when outliers are a problem (it is used in other fields [9] for the same purpose).

Partial results of the experiments are displayed in Figure 6 and Figure 8. Comparisons with $m = 5$ are

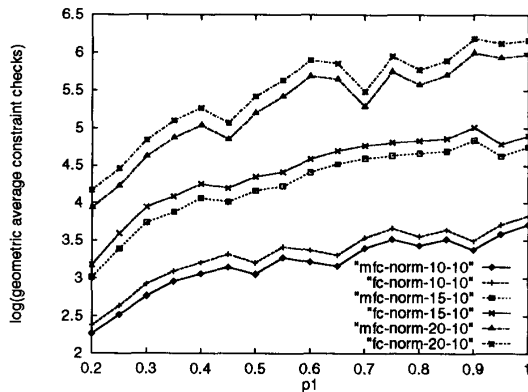

Figure 6: Comparison of the log of the geometric mean of constraint checks performed by MFC-NORM and FC-NORM varied by p_1

omitted. Along the x-axis are the values for p_1 and along the y-axis is the log (base 10) of the geometric mean of the number of constraint checks. Each point represents the log of the geometric mean of the number of constraint checks performed in solving the 20 random CSP instances. The key at the bottom right corner displays the algorithm name, n, m, and the line associated with that run.

In the first graph (Figure 6) MFC-NORM appears to perform almost uniformly better than FC-NORM. As the size of the problems increases the difference between the graphs becomes larger. As the distance between the graphs is logarithmic, the almost constant distance between the graphs is actually a multiplier for the number of constraint checks. For example, the distance between MFC-NORM and FC-NORM for $n = 10$, $m = 10$ is approximately 0.137 which is $\log_{10}(1.37)$. This means that the geometric average number of constraint checks performed by FC-NORM is 1.37 times the number of constraint checks performed by MFC-NORM. The graphs with $m = 5$ have the same characteristics as those displayed. Figure 7 shows the performance of MFC-NORM in terms of the percentage of constraint checks performed by FC-NORM. One extra data point for $n = 20$, $m = 15$, p_1 in $\{0.2, 0.25, \ldots, 0.5\}$ is added. There are two trends observable in Figure 7. The first is that as the domain size increases for every n, MFC-NORM is increasingly more efficient than FC-NORM. The second trend is that MFC-NORM becomes increasingly more efficient than FC-NORM as the number of variables increases.

In the second graph (Figure 8), MFC-VAR appears to do better or the same as FC-VAR. MFC-VAR's

	$m = 5$	$m = 10$	$m = 15$
$n = 10$	76.9	72.9	–
$n = 15$	72.5	66.2	–
$n = 20$	68.8	61.6	54.2

Figure 7: Percentage of FC-NORM's constraint checks performed by MFC-NORM

performance appears to worsen as the size (both n and m) of the problems grows larger. The choice of an incorrect variable appears to be more critical for larger problems.

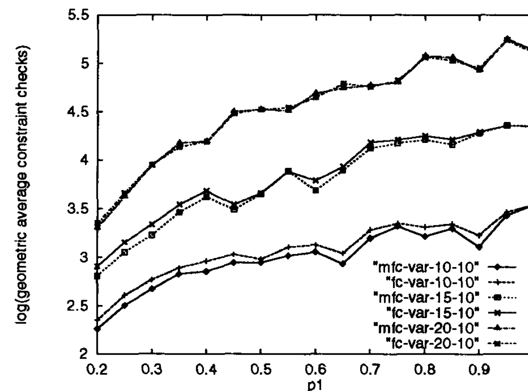

Figure 8: Comparison of the log of the geometric mean of constraint checks performed by MFC-VAR and FC-VAR varied by p_1

4 Conclusions and Future Work

Our experiments have shown that MFC is increasingly more efficient than FC for randomly generated hard problems with large domain sizes and/or a large number of variables given a variable selection heuristic not dependent on domain size. Our experiments also show that using MFC with a variable selection heuristic based on domain size is inappropriate for larger problems.

There are two reasons why MFC is better than FC in terms of constraint checks. The first is that for every minimal forward check that fails the delayed forward checks in the future-connected variables between the current variable and the variable whose domain become empty are not performed. The second reason is that sections of the search tree that have

not been backtracked over may have delayed forward checks that are avoided.

Our future work lies in improving MFC's search to avoid unnecessary constraint checks using the extra information that it has. The MFC algorithm as presented in this paper mimics the search of FC. However if we sacrifice the explicit comparison to FC we can exploit the extra information to avoid some redundant searches. For example, if a value for the current variable v_i causes some future domain d_j to become empty, instead of recording that v_i is inconsistent with v_{i-1} we could record it as inconsistent with the deepest variable that can change d_j. This would ensure that MFC never instantiates v_i to that value as long as that value would empty the future-connected domain d_j. This optimization of FC can be seen as a form of partial Backmarking and is described in detail in [11]. A second optimization missing in MFC is the addition of a intelligent backtracking component. If the search jumps back to the source of a failure instead of the previously instantiated variable the delayed forward checks for the variables between will be avoided. Finally we would like to improve the performance of MFC-VAR by learning when it is critical to completely check a domain.

Acknowledgements

The authors would like to thank Pat Prosser for his code and advice, and Eugene Freuder, Christian Bessière, Ted Elcock, Mei Wei, and the anonymous referees for their comments. This research is funded by the Institute for Robotics and Intelligent Systems (a Canadian Network of Centres of Excellence) Project B-5 and NSERC Grant 0036853.

References

[1] C. Bessière and M.-O. Cordier. Arc-Consistency and Arc-Consistency Again. In *Proceedings AAAI-93*, 1993.

[2] M. Dent and R. Mercer. Minimal Forward Checking. Technical Report UWO-CSD-374, University of Western Ontario, 1993.

[3] R. Haralick and G. Elliot. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.

[4] A. Mackworth. Constraint Satisfaction. In *2nd Edition of the Encyclopedia of Artificial Intelligence*, pages 285–293. Wiley & Sons, New York, 1992.

[5] J. McGregor. Relational Consistency Algorithms and their Application in Finding Subgraph and Graph Isomorphisms. *Information Sciences*, 19:229–250, 1979.

[6] B. Nadel. Tree Search and Arc Consistency in Constraint Satisfaction Algorithms. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 287–342. Springer-Verlag, 1988.

[7] B. Nadel. Constraint Satisfaction Algorithms. *Computational Intelligence*, 5:188–224, 1989.

[8] B. Nadel and J. Lin. Automobile Transmission Design as a Constraint Satisfaction Problem: Modeling the Kinematic Level. *Artificial Intelligence in Engineering Design Analysis and Manufacturing (AIEDAM)*, 5(3), 1991.

[9] D. Patterson and J. Hennessy. *Computer Architecture: a Quantitative Approach.* Morgan Kaufmann, 1990.

[10] P. Prosser. A Reactive Scheduling Agent. In *Proceedings IJCAI-89*, pages 1004–1009, 1989.

[11] P. Prosser. Forward Checking with Backmarking. Technical Report AISL-48-93, University of Strathclyde, 1993.

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

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

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

[15] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.