MINIMAL FORWARD CHECKING*

Michael J. Dent
Dept. of Computer Science
University of Western Ontario
michael.dent@uwo.ca

Robert E. Mercer
Dept. of Computer Science
University of Western Ontario
mercer@csd.uwo.ca

February 3, 1994

Abstract

Forward Checking is a highly regarded search method used to solve Constraint Satisfaction Problems. This method performs a limited type of lookahead attempting to find a failure earlier during a backtracking search. In this paper a new search method, Minimal Forward Checking, is introduced which under certain conditions performs the same amount of constraint checking in the worst case as Forward Checking and retains Forward Checking's ability to detect future consistent values. Minimal Forward Checking is a lazy version of Forward Checking that suspends some "forward checks" until they are needed. Minimal Forward Checking is implemented as a marriage of incremental Forward Checking with Backmarking. Experimental evidence gathered from comparisons of the two algorithms on two large samples of randomly generated problems shows that Minimal Forward Checking outperforms Forward Checking especially as the domain sizes increase. The experiments also show that the use of a dynamic variable selection heuristic benefits both of the algorithms.

1 Introduction

Many studies have found that using Forward Checking (FC) during a backtracking search is an efficient method for solving Constraint Satisfaction Problems (CSP's)[3, 2, 4, 5, 6]. FC's efficiency is usually attributed to the following: it does less arc consistency checking per node than other more complicated arc consistency algorithms, it detects inconsistencies earlier in the search tree, and it is relatively inexpensive in terms of record keeping[2, 5]. However, our intuition about FC is that for problems with large domain sizes, too much arc consistency checking may still be done. Early failures in the search tree may make much of FC's consistency checking redundant. Our intuition is not the first hint that FC may be doing to much work: Nadel states in [4, p307] that "... *forward checking works hard early to avoid wasteful work later. It is possible to overdo it though. Especially when the problem constraints are tight, more work can be expended early than is saved later ...*".

*Reference as University of Western Ontario Technical Report UWO-CSD-374

FC's "best" asset is that it can detect future failures quickly during a backtracking search. It does this by *completely* filtering all future connected domains with respect to the current attempted instantiation during search. If a future connected domain becomes empty then the current instantiation is an inconsistent choice and must be changed. An alternative way of looking at FC's asset is that it only allows the search to move forward if there is at least one consistent value in each forward domain that is connected to the instantiated variables.

In this paper we present a new algorithm, named Minimal Forward Checking (MFC), which in the *worst case* performs the same amount of arc consistency checking as FC (when the same variable instantiation order is followed and the domains are kept as sequences). MFC works on the basis of the second interpretation above — it finds one consistent value in every forward connected domain, "suspending" the other forward checks until either the search node is backtracked to or a previously consistent value in a future domain becomes inconsistent with a new attempted instantiation. The unsuspension of a forward check to find a new consistent value in a forward connected domain is implemented as a modified version of Gaschnig's BackMarking (BM)[1]. The unsuspension of a forward check at the current node is implemented as a combination of BM and MFC. Gaschnig's BM has been modified to use better data structures that allow a dynamic instantiation order.

MFC needs "lookup tables" to keep track of successful and unsuccessful constraint checks. Haralick and Elliot argued in [2, p286ff] that the (time) efficiency of backmarking is dominated by the amount of table lookups done — "*Only in the case that a computation of a relation check is significantly more expensive than the cost of a node's loop control and a table lookup will relation checks be a useful practical measure for the time complexity of backmarking.*" In our experiments with MFC we found that this is only partially true. When domain sizes are small and constraint checks inexpensive FC is more efficient (in terms of time) than MFC. However, when dealing with CSP's with large domain sizes MFC outperforms FC. We can also reasonably assume that MFC would outperform FC on CSP's with expensive constraint checks (even with smaller domain sizes).

FC can be greatly improved by using a dynamic variable selection heuristic that picks the future variable with the smallest domain size as the next to be instantiated[3]. The heuristic tries to minimize the number of possible branches leading down in the search tree (i.e. it leaves fewer backtracking points). We expected MFC to do poorly compared to FC as FC has more accurate information about the size of a search node (because it deletes all values inconsistent with the current state of the search). MFC only searches for the first consistent value in each forward connected domain, leaving the rest untouched. We expected that MFC would do poorly as it wouldn't know the true number of consistent values in the future domains. Again we found that MFC is more efficient than FC in most cases! It appears that MFC filters forward domains (in many cases) as well as FC when it matters (e.g. when the constraints are tight). In fact, in both experiments MFC with the heuristic performed the same or fewer constraint checks than FC with the heuristic for 97% of the problems and fewer constraint checks for 73% of the problems.

2 Minimal Forward Checking

In the following subsections we give terminology needed to describe FC and MFC. We outline the data structures and functions used to implement FC and MFC. Finally, we describe our implementation of FC and MFC in detail. The algorithms are described in a pseudocode based on CommonLisp and Pascal. A good description of the language used is given in [5]. Variables local to a procedure are assumed to be implicitly declared. Common list operations such as *first*, *rest*, *append*, *push*, *pop*, etc. , are assumed to be primitives of the language. The basis for our implementation of MFC and FC is a constraint satisfaction system designed by Prosser[6, 7]. No attempt was made to optimize Prosser's original implementation of FC or our implementation of MFC. Data structures were chosen mainly for their simplicity. Many of our definitions and data structures are necessarily similar to his.

2.1 Definitions

The **Binary Constraint Satisfaction Problem** (BCSP) can be represented with a set of variables $V = \{v_1, \ldots, v_n\}$, a set of domains $D = \{d_1, \ldots, d_n\}$ ($\|d_i\| = m_i$, each d_i a finite sequence of values), and a set of binary constraints on pairs of the variables $C = \{c_{12}(v_1, v_2), c_{13}(v_1, v_3), \ldots, c_{n-1,n}(v_{n-1}, v_n)\}$ ($c_{ij} = c_{ji}$). A BCSP can be represented as a constraint graph where the vertices of the graph correspond to the set of variables and the directed edges correspond to the set of binary constraints. In this paper we are only concerned with finding the first solution to a BCSP — the assignment of a value to each variable v_i, from its respective domain D_i, such that the set of constraints C is satisfied.

The variable **instantiation order** is the order in which variables are chosen to be evaluated. Assume for the remainder of this definition that the variable instantiation order is $v_1, \ldots, v_n$. Then the **current variable**, say v_k, is the variable to be instantiated. The instantiated variables $v_1, \ldots, v_{k-1}$ are called the **past variables** and the uninstantiated variables $v_{k+1}, \ldots, v_n$ are called the *future variables*. The **past-connected variables** are the subset of past variables that are connected by a constraint to the current variable v_k. The **future-connected variables** are the subset of future variables that are connected by a constraint to the current variable v_k. The **current domain** of a variable is the original domain of a variable with some values deleted that are inconsistent with the current search state.

2.2 Data Structures

In our implementation all of the following arrays are declared globally. We include the space complexity for each array. In the analysis, m is the size of the largest domain and n is the number of variables. The following three arrays are used by both algorithms.

v[i] ($1 \leq i \leq n$) contains the current assignment for v_i. The space complexity is $O(n)$.

current-domain[i] ($1 \leq i \leq n$) is a list of the current domain of v_i kept as a sequence. The space complexity is $O(nm)$.

constraint[i,j] ($1 \leq i, j \leq n$) contains the name of a function that determines if the current values of v_i and v_j satisfy the constraint between v_i and v_j. If there is no constraint between v_i and v_j, *constraint[i,j]* = *nil*. The space complexity is $O(n^2)$.

FC uses the following three arrays.

future-fc[i] ($1 \leq i \leq n$) is a list of future variables that have been checked with v_i. The space complexity is $O(nm)$.

past-fc[i] ($1 \leq i \leq n$) is a list of past variables that have been checked with v_i. The space complexity is $O(nm)$.

reductions[i] ($1 \leq i \leq n$) is a list of lists containing domain elements of v_i that have been deleted by FC. The space complexity is $O(nm)$.

MFC needs the following four arrays, the last three for record keeping during a search.

attached-to[i] ($1 \leq i \leq n$) is a list of variables that v_i is connected to by a constraint. The space complexity is $O(nm)$.

previous-checks[i] ($1 \leq i \leq n$) is a list of lists where each sublist, corresponding to an element of the current domain of v_i, is a list of variables that the corresponding domain element is consistent with at any particular stage in the search. For example, if v_i has two elements in its current domain, *previous-checks[i]* = ((5 1 3)(4 1 2)) means that the first domain element of v_i is consistent with the current instantiations of v_5, v_1, and v_3 and that the second domain element is consistent with the current instantiations of v_4, v_1, and v_2. Each sublist is kept in the reverse of the instantiation order.

The space complexity is $O(n^2m)$. There are n variables, each with a maximum m elements, and each domain element corresponds to a list of variables which can be at most n.

future-consistent[i] ($1 \leq i \leq n$) is a list of lists where each sublist is a pair (v_x k), $v_x \in V$, $k \in D_{v_x}$, with the intended meaning that the current instantiation of v_i is consistent with the value k in the domain of future (with respect to v_i) variable v_x.

The space complexity is $O(n^2m)$. Variable v_1 can only be consistent with at worst (a fully connected graph) $m(n-1)$ values, variable v_2 with $m(n-2)$ values, and so on with variable v_{n-1} begin consistent with $m(1)$ values. Therefore the size complexity is $O(m\sum_{i=1}^{n-1} i) = O(n^2m)$.

future-inconsistent[i] ($1 \leq i \leq n$) is a list of lists where each sublist is a triple (v_x k previous-checks-of(v_x, k)) with the intended meaning that the current instantiation of v_i is inconsistent with the value k in the domain of future (with respect to v_i) variable v_x and previous-checks-of(v_x k) is a list of past variables (with respect to v_x) that domain element k is consistent with. For example, the list (10 *a* (1 3 4)) would mean that

v_i is inconsistent with the value a for v_{10} and that that value for v_x is consistent with the current instantiations of $v_1, v_3,$ and v_4 (which must be before v_i in the instantiation order).

The space complexity is $O(n^2m)$. This is the most difficult datastructure to get a complexity bound on. Insertion of an element in this array implies that a domain element of some v_i is removed. The worst case total number of values that can possibly be inconsistent is $O(nm)$ and the worst case number of variables that each element could have been successfully been checked against is $O(n)$. Therefore a very loose bound on the worst case complexity is $O(n^2m)$.

2.3 Global Functions

The following functions are also globally accessible. Only the first function is used by our implementation of FC but all of the following functions could conceivably be used by an implementation of FC.

check(i,j) returns true if there is no constraint between v_i and v_j and returns the value of the call to the constraint otherwise.

find-past(i) returns a list of past-connected values (in instantiation order) for v_i.

find-future(i) returns a list of future-connected values (in instantiation order) for v_i.

find-unseen-vars(i,pastvars) takes a list *pastvars* containing the past variables and returns a list (in instantiation order) of the past-connected variables that haven't yet been checked with the first value in the current domain of v_i.

MFC also uses the following functions to update MFC's record keeping arrays.

add-positive-check(i,j) takes two variables where v_i is before v_j in the instantiation order. It adds i to the front of the list in *previous-checks[j]* and it adds the list *(j v[j])* to the front of the list in *future-consistent[i]*.

add-remember-list(i,j) takes two variables where v_i is before v_j in the instantiation order. It updates *future-inconsistent[i]* by adding the sublist *(j v[j] previous-checks-of(j,v[j]))* to the front of the list.

delete-var-reference(i,j,x) deletes a reference to i in *previous-checks[j]* for the value x in *current-domain[j]*.

putback(i,a,previous-checks-of(i,a)) puts a back into *current-domain[i]* (in order) and adds *previous-checks-of(i,a)* (in order) back into *previous-checks[i]*. Domains are kept as sequences only for the purposes of comparison.

```
1   FUNCTION bcssp-dynamic(label,unlabel,unlabeled-vars,choice-function):string;
2   BEGIN
3       labeled-vars ← nil;
4       status ← "unknown";
5       re-label ← False;
6       consistent ← True;
7       WHILE status ="unknown" DO
8           BEGIN
9               IF consistent THEN
10                  BEGIN
11                      IF re-label THEN
12                          BEGIN
13                              i ← first(unlabeled-vars);
14                              re-label ← False
15                          END
16                      ELSE
17                          i ← choice-function(unlabeled-vars);
18                      consistent ← label(i,labeled-vars,unlabeled-vars);
19                      IF consistent THEN
20                          BEGIN
21                              unlabeled-vars ← remove(i,unlabeled-vars);
22                              push(i,labeled-vars)
23                          END
24                  END
25              ELSE
26                  BEGIN
27                      consistent ← unlabel(i,labeled-vars,unlabeled-vars);
28                      push(pop(labeled-vars),unlabeled-vars);
29                      IF not consistent THEN
30                          i ← first(unlabeled-vars)
31                      ELSE
32                          re-label ← True
33                      END
34                  END;
35              IF consistent and null(unlabeled-vars) THEN
36                  status ← "solution"
37              ELSE
38                  IF null(labeled-vars) and not consistent THEN
39                      status ← "impossible"
40          END
41      return(status)
```

Figure 1: bcssp-dynamic

2.4 Description of the Algorithms

Usually, tree search algorithms are presented in a recursive style. In this paper we follow Prosser's lead in unravelling the tree search algorithm into a forward labeling and a backward unlabeling move[6]. Figure 1 shows the calling program *bcssp-dynamic* for the two tree search algorithms. Function *bcssp-dynamic* is a dynamic version of Prosser's static *bcssp* function that allows a dynamic selection strategy to be used to pick the next variable to be instantiated. The function *bcssp-dynamic* takes as parameters the name of the labeling function *label*, the name of the unlabeling function *unlabel*, a list of the variables to be instantiated *unlabeled-vars*, and the name of a function that can select the next variable to be instantiated.

Function *bcssp-dynamic* continues to call the labeling and unlabeling functions until the CSP is solved or proved to be unsatisfiable. If the current state of the search is consistent, then a labeling is attempted. Lines 13 and 17 show how a variable is chosen for instantiation. If the search has backtracked to a search node then a relabeling is attempted (line 13), otherwise a user supplied choice function is called with the unlabeled variables to pick the next variable to instantiate. The labeling function then attempts to instantiate the chosen variable. If the labeling function succeeds, the variable chosen is recorded as being instantiated and the search continues.

If the current state of the search is inconsistent then the current variable is unlabeled and

```
1   FUNCTION fc-label(i,labeled-vars,unlabeled-vars):Boolean;
2   BEGIN
3       consistent ← False;
4       FOR v[i] ← EACH ELEMENT OF current-domain[i] WHILE not consistent DO
5           BEGIN
6               consistent ← True;
7               FOR j EACH ELEMENT OF unlabeled-vars WHILE consistent DO
8                   consistent ← check-forward(i,j);
9               IF not consistent THEN
10                  BEGIN
11                      current-domain[i] ← remove(k,current-domain[i]);
12                      undo-reductions(i)
13                  END
14          END;
15      return(consistent)
16  END
```

Figure 2: fc-label

removed from the instantiated list (*labeled-vars*). If the domain of the unlabeled variable is empty the search returns to the unlabeling part to unlabel the previous variable. Otherwise, a relabeling is attempted.

If the current state of the search is consistent and there are no more variables to be instantiated then the string "solution" is returned. If there are no more variables to instantiate and the current state is inconsistent the string "impossible" is returned. The function *bcssp-dynamic* is a decision function for CSP's.

Figures 2 to 6 show the algorithm for FC. These figures are almost the same as those presented by Prosser in [6]. Prosser's version of FC has been slightly modified to allow for dynamic variable selection. Figures 7 to 11 give the algorithm for MFC. In the following paragraphs we describe in detail the two algorithms.

2.5 Forward Checking

Functions *fc-label* (Figure 2) and *fc-unlabel* (Figure 3) are the labeling and unlabeling functions respectively for FC. Function *check-forward* (Figure 4) is used by *fc-label* to perform a forward check. Function *undo-reductions* (Figure 5) is used by both *fc-label* and *fc-unlabel* to undo domain reductions done by *check-forward*. Finally, *update-current-domain* (Figure 6) is used by *fc-unlabel* to restore a domain after backtracking to it.

Function *fc-label* attempts to find an element in v_i's current domain that satisfies a forward check (using *check-forwards*) of its forward domains. If a forward check fails for a particular domain element, that is a future domain becomes empty, the domain element is removed and the search continues. If *fc-label* finds a consistent value, the value *true* is returned, otherwise *false* is returned.

Function *fc-unlabel* finds the variable instantiated previous to v_i, undoes the forward checks done for that variable, returns v_i's domain to the state it was in before the tree search went past it, and deletes the previous variable's value from its domain. Function *fc-unlabel* returns *true* if the previous variable can be relabeled (its current domain is not empty), *false* otherwise.

Function *check-forwards* goes through a forward variable's (v_j) domain and deletes any

```
1  FUNCTION fc-unlabel(i,labeled-vars,unlabeled-vars):Boolean
2  BEGIN
3     h ← first(labeled-vars);
4     undo-reductions(h);
5     update-current-domain(i);
6     current-domain[h] ← remove(v[h],current-domain[h]);
7     consistent ← current-domain[h] ≠ nil;
8     return(consistent)
9  END
```

Figure 3: fc-unlabel

```
1  FUNCTION check-forward(i,j):Boolean;
2  BEGIN
3     IF constraint[i,j] ≠ nil THEN
4        BEGIN
5           reduction ← nil;
6           FOR v[j] EACH ELEMENT OF current-domain[j] DO
7              IF not check(i,j) THEN
8                 push(k,reduction);
9           IF reduction ≠ nil THEN
10             BEGIN
11                current-domain[j] ← set-difference(current-domain[j],reduction);
12                push(reduction,reductions[j]);
13                push(j,future-fc[i]);
14                push(i,past-fc[j])
15             END
16       END
17    return(current-domain[j] ≠ nil)
18 END
```

Figure 4: check-forward

values inconsistent with the current instantiation (v_i). Deleted values (here called reductions) and v_i and v_j are noted in arrays *reductions, future-fc* and *past-fc*.

Function *undo-reductions* undoes a forward check done for variable v_i. It goes through the future domains where values were deleted (because of inconsistency with the current value of v_i) and puts them back in order. Domain elements are kept in order only for the purposes of comparison.

Function *update-current-domain* restores v_i's current domain to what it should be after variable v_i was searched over. This function is necessary to restore values deleted because of the backtracking search.

The above algorithm is obviously not very efficient. It may be more efficient to use function *find-future* to find future-connected variables instead of looking at all future variables during

```
1  FUNCTION undo-reductions(i):Nil;
2  BEGIN
3     reduction ← nil;
4     FOR j EACH ELEMENT OF future-fc[i] DO
5        BEGIN
6           reduction ← pop(reductions[j]);
7           current-domain[j] ← union(current-domain[j],reduction);
8           pop(past-fc[j])
9        END
10    future-fc[i] ← nil
11 END
```

Figure 5: undo-reductions

```
1  FUNCTION update-current-domain(i):Nil;
2  BEGIN
3     current-domain[i] ← domain[i];
4     FOR reduction EACH ELEMENT OF reductions[i] DO
5        current-domain[i] ← set-difference(current-domain[i],reduction);
6     current-domain[i] ← sort(current-domain[i])
7  END
```

Figure 6: update-current-domain

```
1  FUNCTION mfc-label(i,labeled-vars,unlabeled-vars):Boolean;
2  BEGIN
3     past-vars ← find-past(i);
4     future-vars ← find-future(i);
5     consistent ← False;
6     FOR v[i] ← EACH ELEMENT OF current-domain[i] WHILE not consistent DO
7        BEGIN
8           IF backward-consistent(i,past-vars) THEN
9              BEGIN
10                consistent ← min-check-forward(i,future-vars,labeled-vars);
11                IF not consistent THEN
12                   BEGIN
13                      undo-min-check-forward(i);
14                      add-remember-list(first(labeled-vars),i)
15                   END
16                ELSE add-remember-list(first(labeled-vars),i)
17             END;
18          IF not consistent THEN
19             BEGIN
20                current-domain[i] ← rest(current-domain[i]);
21                previous-checks[i] ← rest(previous-checks[i])
22             END
23       END
24    return(consistent)
25 END
```

Figure 7: mfc-label

forward checks. However, calls to *check* for non-existent constraints were not counted as constraint checks. Another source of inefficiency are the calls to *sort* (line 7 in *undo-reductions* and line 6 in *update-current-domain*). These calls are used to maintain the domains as sequences (for the purpose of comparison) although better methods could be used.

2.6 Minimal Forward Checking

Function *mfc-label* (Figure 7) and *mfc-unlabel* (Figure 12) are the labeling and unlabeling functions, respectively, for MFC. Function *min-check-forward* (Figure 9) performs a minimal forward check (similar to *check-forward* used in FC). Function *backward-consistent* (Figure 8) is used to check if domain elements are consistent with the variables instantiated so far. Function *undo-min-check-forward* (Figure 11) is used to undo a minimal forward check much like *undo-reductions* in FC. Finally, function *find-one-consistent* (Figure 10) finds one backward consistent value in a forward domain.

Function *mfc-label* searches through the current domain of v_i for a value that is consistent with the past-connected variables and with at least one value in each of the future-connected variables domains. Line 8 calls *backward-consistent*. Function *backward-consistent* goes through the past-connected variables (in the instantiation order) that haven't yet been checked with the current value of *v[i]*. Successful and unsuccessful checks are recorded

```
1  FUNCTION backward-consistent(i,backvars):Boolean;
2  BEGIN
3      ok-result ← True;
4      unseen-vars ← find-unseen-vars(i,backvars);
5      FOR m EACH ELEMENT OF unseen-vars WHILE ok-result DO
6          BEGIN
7              ok-result ← check(m,i);
8              IF ok-result THEN
9                  add-positive-check(m,i)
10             ELSE
11                 add-remember-list(m,i)
12         END
13     return(ok-result)
14 END
```

Figure 8: backward-consistent

```
1  FUNCTION min-check-forward(i,future-vars,labeled-vars)
2  BEGIN
3      result-ok ← True;
4      FOR k EACH ELEMENT OF future-vars WHILE result-ok DO
5          result-ok ← find-one-consistent(i,k,labeled-vars);
6      return(result-ok)
7  END
```

Figure 9: min-check-forward

using *add-positive-check* and *add-remember-list* (the loop ends immediately at the first unsuccessful check). If there are unchecked past-connected variables, *backward-consistent* is "unsuspending" a forward check. Function *backward-consistent* is performing a form of Gaschnig's BackMarking(BM)[1]. BM is backtracking with arrays that keep track of successful and unsuccessful constraint checks to avoid redundant constraint checks. We do the same recording as BM using arrays *previous-checks* for successful checks and *future-inconsistent* for unsuccessful checks.

If a backward consistent value is found, then a minimal forward check is performed. For each future-connected variable *min-check-forwards* attempts to find a consistent value with a call to *find-one-consistent*. If an empty forward domain is found the forward search is terminated. Function *find-one-consistent* tries to find one value in v_k's domain that is consistent with v_i. If a backward consistent value is found and *check*(v_i, v_k) is true, the successful check is remembered and the function returns. Otherwise, the function remembers which variable is in conflict with which domain element using *add-remember-list* and deletes the value out of v_k's domain (including previous checks for that value). If the result of *min-check-forward* is successful then the current value and previous checks list for that value are saved using *add-remember-list* (needed to restore the domain when backtracking) and the function exits with a successfully instantiated v_i. If the current value of v_i does not have a successful minimal forward check, then the minimal forward check is undone using *undo-min-check-forward*, the value and inconsistency are noted and the value is deleted from the domain of v_i. Function *undo-min-check-forward* puts back (for the purposes of comparison values are returned in order), all future domain elements that were deleted and deletes all successful checks that were made with the future domain values and v_i. If no domain element can be successfully instantiated then *mfc-label*

```
1  FUNCTION find-one-consistent(i,k,labeled-vars):Boolean;
2  BEGIN
3      result-ok ← False;
4      all-vars ← attached-to[k];
5      backvars ← find-past(labeled-vars,all-vars);
6      FOR v[k] ← EACH ELEMENT OF current-domain[k] WHILE not result-ok DO
7          BEGIN
8              IF backward-consistent(k,backvars) THEN
9                  BEGIN
10                     result-ok ← check(i,k);
11                     IF not result-ok THEN
12                         add-remember-list(i,k)
13                     ELSE
14                         add-positive-check(i,k)
15                 END;
16             IF not result-ok THEN
17                 BEGIN
18                     current-domain[k] ← rest(current-domain[k]);
19                     previous-checks[k] ← rest(previous-checks[k])
20                 END
21         END;
22     return(result-ok)
23 END
```

Figure 10: find-one-consistent

```
1  FUNCTION undo-min-check-forward(i):Nil
2  BEGIN
3      FOR m EACH ELEMENT OF future-inconsistent[i] DO
4          putback(first(m),second(m),third(m));
5      FOR m EACH ELEMENT OF future-consistent[i] DO
6          delete-var-reference(i,first(m),second(m));
7      future-inconsistent[i] ← nil;
8      future-consistent[i] ← nil
9  END
```

Figure 11: undo-min-check-forward

```
1  FUNCTION mfc-unlabel(i,labeled-vars,unlabeled-vars):Boolean;
2  BEGIN
3      h ← first(labeled-vars);
4      IF null(labeled-vars) THEN
5          consistent ← current-domain[i] ≠ nil
6      ELSE
7          BEGIN
8              undo-min-check-forward(h);
9              current-domain[h] ← rest(current-domain[h]);
10             previous-checks[h] ← rest(previous-checks[h]);
11             consistent ← current-domain[h] ≠ nil
12         END;
13     return(consistent)
14 END
```

Figure 12: mfc-unlabel

returns false.

Function *mfc-unlabel* (Figure 12) first finds the variable instantiated before v_i. It then checks to see if it is at the beginning of the search by seeing if there are any labeled variables left. If it is at the beginning of the search it returns true if there are any values left in the first domain, false otherwise. If it is not at the beginning of the search then the previously successful minimal check forward for v_h is undone (the value and previous checks already recorded by *mfc-label*), the value removed from the domain and a flag returned — true if there are any more possible values in the current domain of v_h, false otherwise.

In the above descriptions of FC and MFC no value in a domain is used unless it was backward consistent. All "backchecks" are done in the order of instantiation and successful and unsuccessful checks are remembered until the variable-value pair is backtracked over. Values are returned to domains in the original order. MFC and FC only move forward in a search with the same variables and values and MFC moves backward whenever FC would have skipped back. From the above description one can see that MFC is a lazy version of FC that performs "forward checks" only when necessary. The forward checks are unsuspended using the same instantiation order as FC and domains are always kept as sequences. Therefore the following theorem holds true.

Theorem 1 *Minimal Forward Checking's worst case performance in terms of constraint checks performed in attempting to solve a CSP is the number of constraint checks performed by Forward Checking on the same CSP. This assumes that variable selection is static and domains are kept as sequences.*

3 Example Executions of FC and MFC

Consider the following graph colouring CSP:

$D_1 = \{\text{red}\}$, $D_2 = \{\text{green}, \text{orange}\}$, $D_3 = \{\text{blue}, \text{green}\}$, $D_4 = \{\text{green}, \text{blue}, \text{red}\}$,

where the constraints restrict pairs of variables from $\{v_1, \ldots, v_4\}$ to be assigned different colours.

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

Figure 13: Execution of Forward Checking

Figure 13 outlines the search performed by FC using a static variable ordering. The checkmarks (√) show successful constraint checks and the (X) marks show unsuccessful constraint checks. Step 0 shows the initial state of the domains before the search. In step 1, v_1 is assigned the value red and forward checking goes through the forward-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 forward-connected domain (step 2). Variable v_2 is assigned the value green and the forward-connected domains are checked. The value green in both D_3 and D_4 are inconsistent and are removed. Again the search moves forward as there are consistent values in the forward-connected domains (step 3). Variable v_3 is assigned the value blue and a forward check is done. Forward checking finds that the value blue in D_4 is inconsistent with the value chosen for v_3. There are no further elements to try in D_4 nor in D_3 so the search backtracks 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 forward-connected domains and finds no inconsistent values. In step 5, v_3 is assigned the value blue and the forward check deletes the value blue from the domain of D_4. Finally, step 6 shows the solution found by FC. FC performed a total of 18 constraint checks.

Notice that for the solution, in step 4 the value green in domain D_3, has been unnecessarily

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

Figure 14: Execution of Minimum Forward Checking

checked with the current value of v_2. Also notice that in step 5 the value blue in domain D_4, has been unnecessarily checked with the current values of v_2 and v_3. In general, for CSP's with large domain sizes and many variables, there may be many unnecessary checks performed by FC.

Figure 14 outlines the search performed by MFC using the same variable ordering as above. Domain values have lists of instantiated variables that they have been checked with. Some variables in the lists have superscripts ($\surd$) and (X) 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 of the forward-connected domains is performed. The first consistent value in each forward-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 found to be consistent with the current instantiation of v_2 but the value green in domain D_4 is found to be inconsistent. MFC searches through the rest of v_4's domain (by unsuspending previous forward checks) searching for a backward consistent value (in this case blue) doing the constraint checks in the instantiation order. As there are still consistent values in each forward domain, the search moves forward and v_3 is assigned the value blue (step 3). However, a minimal forward check shows that no value in domain D_4 is consistent. Value blue is inconsistent with the current instantiation of v_3 and an unsuspension of a forward check shows that the value red is inconsistent with the current instantiation of 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, domain value blue is returned to domain D_4 as it is no longer inconsistent with the value of v_3. In step 5, the value orange in domain D_2 is backchecked with the current instantiation of v_1 and the value green is returned to domain D_3. In steps 6 and 7 the search moves forward as MFC finds the first value in each forward-connected domain consistent. Step 8 shows the solution to the CSP found by MFC.

MFC only performed 15 constraint checks compared to the 18 performed by FC. MFC avoids the redundant checks performed by FC. In the above example executions, domains are kept as sequences and backchecks are done in the instantiation order. Neither of these two actions needs to be done in a real implementation.

4 Experiments

MFC and FC have been tested on two large samples of randomly generated CSP's. The parameters used to generate these random CSPs are:

- p_1 – the probability of a constraint existing between two variables.
- p_2 – the probability that a pair of values in a constraint are inconsistent. This parameter is also called *constraint tightness*.
- m – the maximum size of any domain.
- n – the number of variables.

A uniformly distributed random number generator is used to create the random CSP's. Each CSP is created under the following restrictions. The constraint graph has to be connected (that is, there is a path from v_i to v_j $(1 \leq i, j \leq n)$) and if a constraint existed it has to be non-empty. The only constraint graphs that are useful to us are those that are connected — unconnected constraint graphs have component CSP's that can be solved separately and are not representative of the hardness of n variable CSP's. Constraint graphs with empty constraints are the same as constraint graphs without those edges.

In total, four algorithm's have been tested.

- MFC-NORMAL – Minimal Forward Checking with a fixed instantiation order.
- FC-NORMAL – Forward Checking with a fixed instantiation order.
- MFC-VARIABLE – Minimal Forward Checking with a dynamic instantiation order based on picking a variable with the smallest current domain size.
- FC-VARIABLE – Forward Checking with a dynamic instantiation order based on picking a variable with the smallest current domain size.

In the first experiment p_1 and p_2 are allowed to vary in $\{0.1, 0.3, 0.5, 0.7, 0.9, 1.0\}$, m to vary in $\{5, 10, 15, 30\}$, and n to vary in $\{5, 10\}$. These parameters are chosen so that we can see how the different algorithms performed on a wide range of random CSP's without exceeding our computational resources. For each possible combination of the parameters, 15 random CSP's are generated. The total number of random CSP instances generated for the first experiment was 4,320.

In the second experiment p_1 and p_2 are allowed to vary in $\{0.1, 0.2, \ldots, 1.0\}$, m to vary in $\{5, 10, 15, 30\}$, and n to vary in $\{5, 10\}$. The second experiment has been performed to confirm the results of the first experiment. A more complete set of values could be chosen for p_1 and p_2 since more computational resources were available for this experiment. As with the first experiment, 15 random CSP's are generated for each combination of the parameters. The total number of random CSP instances generated for the second experiment is 12,000.

All tests have been done on a SUN-SPARC 10.

4.1 Analysis of Experiments

The following statistics are calculated in both experiments for each algorithm: the average number of constraint checks (C_μ), the average time (T_μ), the average number of nodes (N_μ), and the average number of checks per node (W_μ), calculated over all the problems and by domain size. The standard deviation for each statistic is also calculated (denoted by the subscript σ). In addition, the relative and cumulative performance of MFC over FC by constraint checks, the average number of checks by p_1, and the average number of checks by p_2 are calculated.

Four sets of graphs have been generated. The first two sets of graphs (Figures 17 to 24 and Figures 25 to 32) show the results for the first experiment, and the last two sets the results for the second experiment (Figures 33 to 40 and Figures 41 to 48). The first set in each experiment shows the results of MFC-NORMAL versus FC-NORMAL, and the second set the results of MFC-VARIABLE versus FC-VARIABLE. In each set there are 8 graphs comparing MFC to FC. The first graph shows the average number of checks, the second the average time (in seconds), the third the average number of nodes, and the fourth the average number of checks per node, as the probable domain size increases. The fifth and sixth graphs show the relative and cumulative performance of the two algorithms. The relative performance being a tally of the constraint checks of MFC over the constraint checks of FC and the cumulative performance

being a summation of the relative performance (expressed as a percentage). The seventh and eighth graphs show how the average constraint checks vary over p_1 and p_2.

A table giving the actual numbers has also been generated for each experiment. Figure 15 gives the results for the first experiment. Figure 16 gives the results for the second experiment.

4.1.1 First Experiment — 4,320 Problems

In the first set of graphs we can see that MFC-NORMAL performed much better than FC-NORMAL in terms of constraint checks and time as the domain sizes increase. For problems with domain sizes much less than 15 elements, FC-NORMAL performed better in terms of time. However, once the domain sizes were allowed to range up to 15 or more, MFC-NORMAL is better. As expected, FC-NORMAL does do more average checks per node than MFC-NORMAL as the domain sizes increase. MFC-NORMAL is performing better by doing fewer checks per node. The Performance Tally graph shows that the random CSP generator generates a uniform number of problems of different hardness. The cumulative graph shows that almost 75% of the problems are solved by MFC-NORMAL with fewer constraint checks, and around 25% are solved with the same number of constraint checks. Half of the problems are solved with 63% or less of the constraint checks done by FC-NORMAL.

Parameter p_1 (the probability of a constraint), does seem to have an effect on MFC-NORMAL and FC-NORMAL. MFC-NORMAL performs fewer checks when the graph is sparse and much fewer checks around $p_1 = 0.7$. The dramatic jump in the graph at p_7 is caused by a few hard problems and one very hard problem. On that one very hard problem, FC-NORMAL performs 8,947,852 constraint checks to MIN-FC's 1,490,589 constraint checks. FC-NORMAL takes 1847 seconds and MFC-NORMAL takes 656 seconds.

Parameter p_2 (the probability of a conflict) also affects the number of constraint checks done. For easy problems (those with easily satisfied constraints), MFC-NORMAL does much better. Again the large jump at p_2 is caused by the one very hard problem.

The comparison of MFC-VARIABLE with FC-VARIABLE in the second set of graphs is a bit surprising. MFC-VARIABLE again clearly performed better than FC-VARIABLE once the domain size increased. MFC-VARIABLE is looking at more nodes but it is still performing better than FC-VARIABLE as it is doing fewer checks per node. The cumulative performance graph shows that about 70% of the problems were solved by MFC-VARIABLE with fewer constraint checks and around 27% were done with the same number of constraint checks. Half of the problems are solved with 59% or less of the constraint checks done by FC-VARIABLE. FC-VARIABLE solved only 2.5% of the problems with fewer constraint checks than MFC-VARIABLE.

As p_1 grows larger the number of constraint checks rises (almost equally) for both MFC-VARIABLE and FC-VARIABLE. The graphs for MFC-VARIABLE and FC-VARIABLE nearly parallel each other for p_2 but MFC-VARIABLE again does fewer average constraint checks (much fewer for easier constraints).

The peaks in the graphs for p_1 and p_2 seen in the comparison of MFC-NORMAL and FC-NORMAL do not appear for MFC-VARIABLE and FC-VARIABLE. The dynamic variable selection heuristic drastically reduces the number of constraint checks necessary. For the one very hard problem MFC-VARIABLE only does 6 constraint checks and FC-VARIABLE does 66 constraint checks.

4.1.2 Second Experiment — 12,000 Problems

In the first set of graphs for the second experiment we can see that MFC-NORMAL again performs better (though not by as much in the first experiment) than FC-NORMAL in terms of constraint checks and time as the domain sizes increase. Again, for problems with domain sizes much less than 15 elements, FC-NORMAL performed better in terms of time. However, once the domain sizes were allowed to range up to 30, MFC-NORMAL is slightly better. FC-NORMAL does do more average checks per node than MFC-NORMAL as the domain sizes increase. The cumulative performance graph shows that almost 83% of the problems were solved by MFC-NORMAL with fewer constraint checks, and around 17% were solved with the same number of constraint checks. Half of the problems are solved with 59% or less of the constraint checks done by FC-NORMAL. Again the tally and cumulative graph show a uniform number of problems of equal hardness.

The value of p_1 does not seem to have any effect on MFC-NORMAL or FC-NORMAL. Both graphs are nearly parallel with MFC-NORMAL doing fewer constraint checks for all values of p_1. The second experiment failed to generate a "hard" problem for FC-NORMAL as it did in the first experiment.

Parameter p_2 also does not seem to have any effect on the two algorithms. Again both graphs are nearly parallel with MFC-NORMAL doing fewer constraint checks for all values of p_2.

The comparison of MFC-VARIABLE with FC-VARIABLE in the second set of graphs for the second experiment show the same surprising result as for the first experiment. MFC-VARIABLE again clearly performed better than FC-VARIABLE once the domain size increased. MFC-VARIABLE is looking at more nodes but it is still performing better than FC-VARIABLE as it is doing fewer checks per node. The cumulative performance graph shows that about 76% of the problems were solved by MFC-VARIABLE with fewer or the same number of constraint checks and around 24% were done with the same number of constraint checks. Half of the problems are solved with 55% or less of the constraint checks done by FC-VARIABLE. FC-VARIABLE solved only 2.6% of the problems with fewer constraint checks than MFC-VARIABLE.

As p_1 grows larger the number of constraint checks rises for both MFC-VARIABLE and FC-VARIABLE. The graph does have a bump at $p_1 = 0.3$ which is probably caused by a bad variable selection order. For p_2, MFC does much better than FC-VARIABLE when constraints are loose. When $p_2 = 0.9$ FC-VARIABLE actually does better than MFC-VARIABLE (for the

first time in this analysis). Again this bump is probably cause by a bad variable selection order.

4.1.3 Summary

MFC seems consistently to do better than FC (with or without the heuristic) in terms of constraint checks. MFC performs much better as domain sizes become larger. In terms of time MFC only performs better as the domain sizes increase. If constraint checks are fairly expensive or domains are large, MFC seems to be a better choice than FC. However, timing results should be taken with a grain of salt as neither algorithm has been optimized in any way. MFC consistently does fewer checks per node than FC (as expected). In Nadel's arc-consistency spectrum MFC falls to the left of FC in terms of arc-consistency taking up a position between backtracking and FC. Parameters p_1 and p_2 don't really seem to have a definite effect on the way MFC and FC work.

5 Conclusions

Our experimental evidence seems to show that Minimal Forward Checking is a better algorithm than FC. For CSP's with small domain sizes FC is more efficient in terms of time. However, for problems with large domain sizes or expensive constraints, Minimal Forward Checking is better. A few caveats about the work reported here. The random problems chosen for this experimental analysis may have been too easy and too small. We conjecture that the performance of MFC will improve for larger and harder problems. Also, random problems may not be totally satisfactory as a benchmark. How well do these results carry over to a real application space?

6 Acknowledgements

The authors would like to thank Pat Prosser for his code and for his helpful advice, and Richard Wallace for fruitful communications. 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.

Figure 15: Results Table for the Four Algorithms by Domain Size (4,320 Problems)

M		MFC-NORMAL	FC-NORMAL	MFC-VARIABLE	FC-VARIABLE
5	C_μ	29	42	9	15
	C_σ	105.72	142.0	15.54	21.5
	T_μ	0.01	0.01	0.0	0.0
	T_σ	0.04	0.02	0.02	0.02
	N_μ	5	5	2	2
	N_σ	18.96	17.98	3.36	3.28
	W_μ	7.63	10.54	4.68	6.94
	W_σ	6.8	10.42	3.32	5.5
10	C_μ	135	241	25	48
	C_σ	887.76	1665.86	39.63	57.16
	T_μ	0.05	0.04	0.01	0.01
	T_σ	0.36	0.27	0.01	0.02
	N_μ	28	27	4	3
	N_σ	325.15	323.94	4.62	4.44
	W_μ	16.01	23.09	8.47	14.59
	W_σ	19.87	25.78	9.28	13.36
15	C_μ	489	1144	48	96
	C_σ	8041.62	24458.04	88.6	146.87
	T_μ	0.21	0.18	0.01	0.02
	T_σ	3.26	3.0	0.03	0.04
	N_μ	173	171	5	5
	N_σ	3716.36	3710.16	7.16	8.16
	W_μ	25.96	38.02	12.9	22.74
	W_σ	39.32	50.02	17.5	20.98
30	C_μ	2013	9425	151	283
	C_σ	45509.5	272321.47	447.49	552.1
	T_μ	1.2	2.54	0.11	0.14
	T_σ	20.27	56.41	0.46	0.42
	N_μ	305	301	9	7
	N_σ	8476.28	8412.85	21.74	15.06
	W_μ	59.53	84.37	25.55	45.73
	W_σ	132.99	168.04	47.38	55.55
All	C_μ	666	2713	58	111
	C_σ	23117.28	136719.44	235.65	305.14
	T_μ	0.37	0.69	0.03	0.04
	T_σ	10.27	28.25	0.23	0.22
	N_μ	128	126	5	4
	N_σ	4630.42	4600.13	12.06	9.31
	W_μ	27.28	39.0	12.9	22.5
	W_σ	72.83	93.02	26.89	33.82

Figure 16: Results Table for the Four Algorithms by Domain Size (12,000 Problems)

M		MFC-NORMAL	FC-NORMAL	MFC-VARIABLE	FC-VARIABLE
5	C_μ	27	40	10	16
	C_σ	81.09	108.96	15.47	21.21
	T_μ	0.01	0.01	0.0	0.0
	T_σ	0.05	0.03	0.02	0.02
	N_μ	6	5	2	2
	N_σ	29.8	29.5	3.51	3.06
	W_μ	7.45	10.41	4.7	7.13
	W_σ	6.73	9.91	3.49	5.65
10	C_μ	623	757	27	52
	C_σ	23708.72	24099.14	45.19	64.71
	T_μ	0.34	0.18	0.01	0.01
	T_σ	14.38	6.89	0.03	0.04
	N_μ	485	479	4	4
	N_σ	24059.84	23782.46	5.49	4.46
	W_μ	14.06	22.43	7.6	14.16
	W_σ	19.1	29.88	7.35	12.34
15	C_μ	236	416	53	101
	C_σ	2912.38	5982.41	113.63	134.68
	T_μ	0.11	0.1	0.02	0.02
	T_σ	1.57	1.05	0.07	0.03
	N_μ	52	51	6	5
	N_σ	1272.16	1239.71	9.53	6.41
	W_μ	19.74	32.2	10.75	20.71
	W_σ	33.74	47.96	13.65	18.77
30	C_μ	719	1273	172	282
	C_σ	7130.04	9823.15	1073.44	531.66
	T_μ	0.66	0.91	0.14	0.15
	T_σ	5.63	4.6	1.64	0.43
	N_μ	76	73	10	8
	N_σ	1789.0	1703.86	45.77	13.62
	W_μ	36.53	59.46	18.84	39.13
	W_σ	91.76	110.65	37.96	45.2
All	C_μ	401	621	66	113
	C_σ	12465.88	13357.69	543.82	294.52
	T_μ	0.28	0.3	0.04	0.04
	T_σ	7.77	4.19	0.82	0.22
	N_μ	155	152	6	5
	N_σ	12079.91	11937.85	23.66	8.14
	W_μ	19.45	31.13	10.47	20.28
	W_σ	51.06	64.88	21.24	28.04

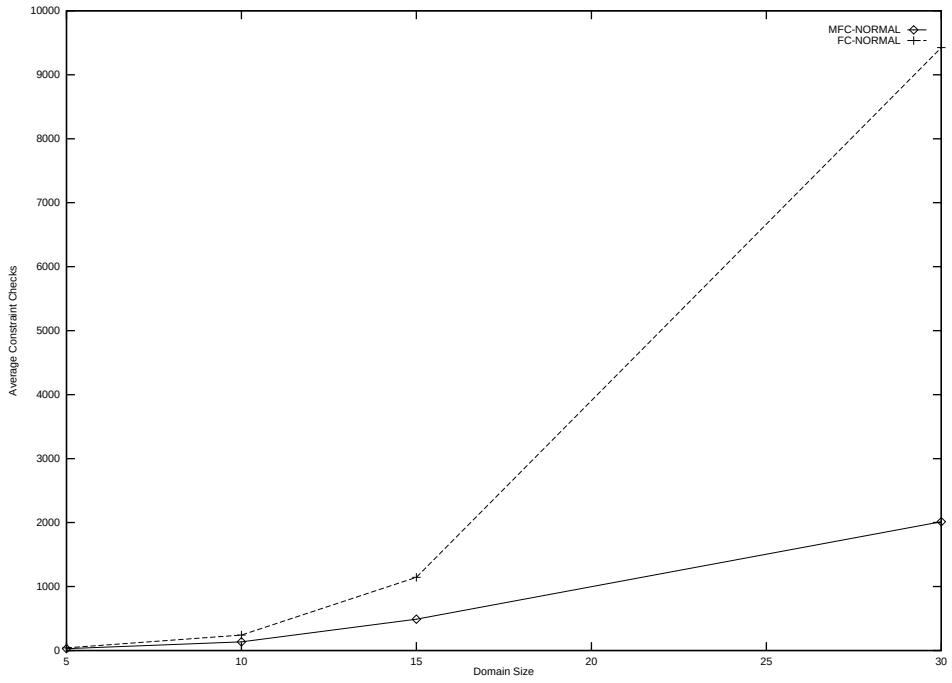

Figure 17: Comparison of MFC-NORMAL vs. FC-NORMAL by the average number of constraint checks performed by each algorithm broken down by domain size (4,320 problems).

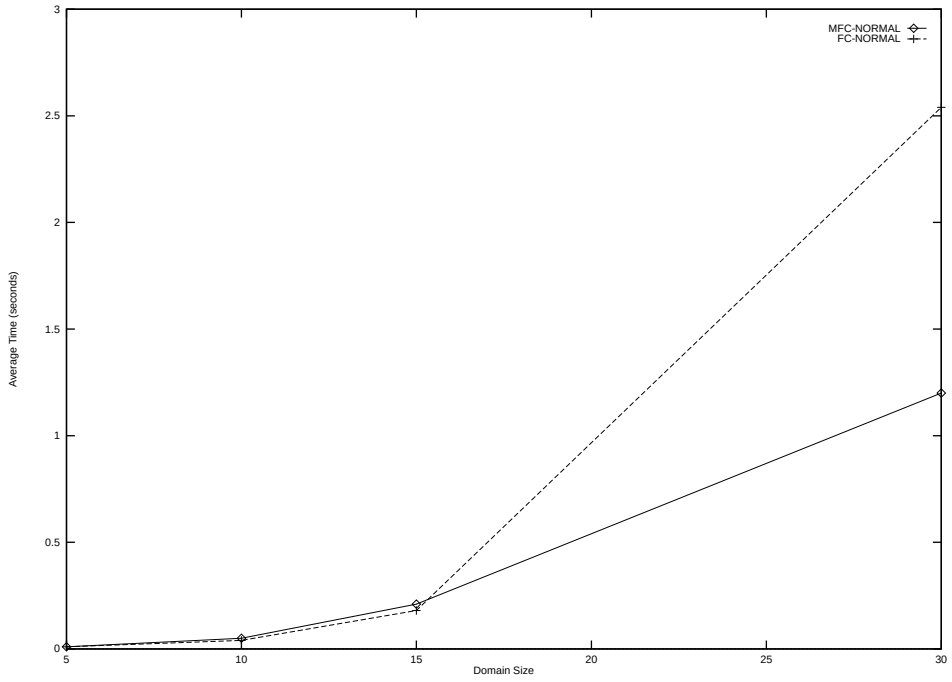

Figure 18: Comparison of MFC-NORMAL vs. FC-NORMAL by the average time used (in seconds) by each algorithm broken down by domain size (4,320 problems).

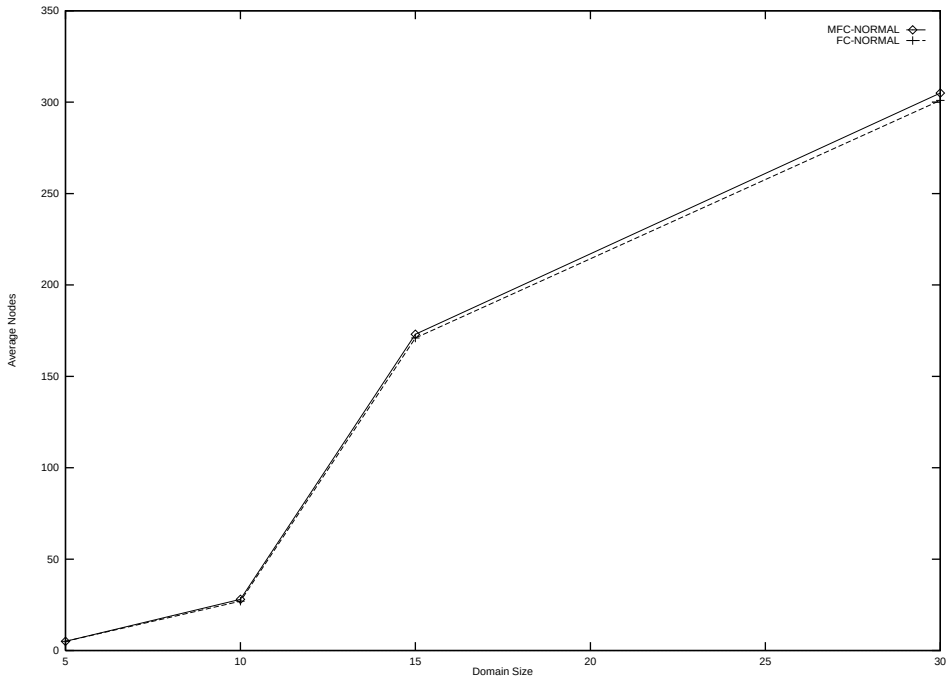

Figure 19: Comparison of MFC-NORMAL vs. FC-NORMAL by the average number of nodes visited by each algorithm broken down by domain size (4,320 problems).

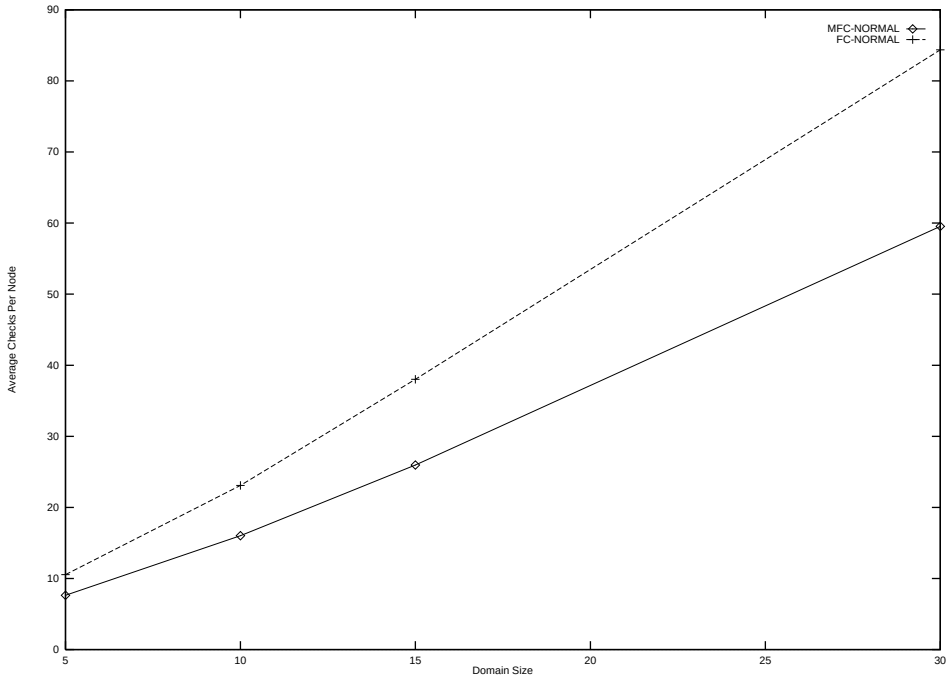

Figure 20: Comparison of MFC-NORMAL vs. FC-NORMAL by the average number of constraint checks performed per node by each algorithm broken down by domain size (4,320 problems).

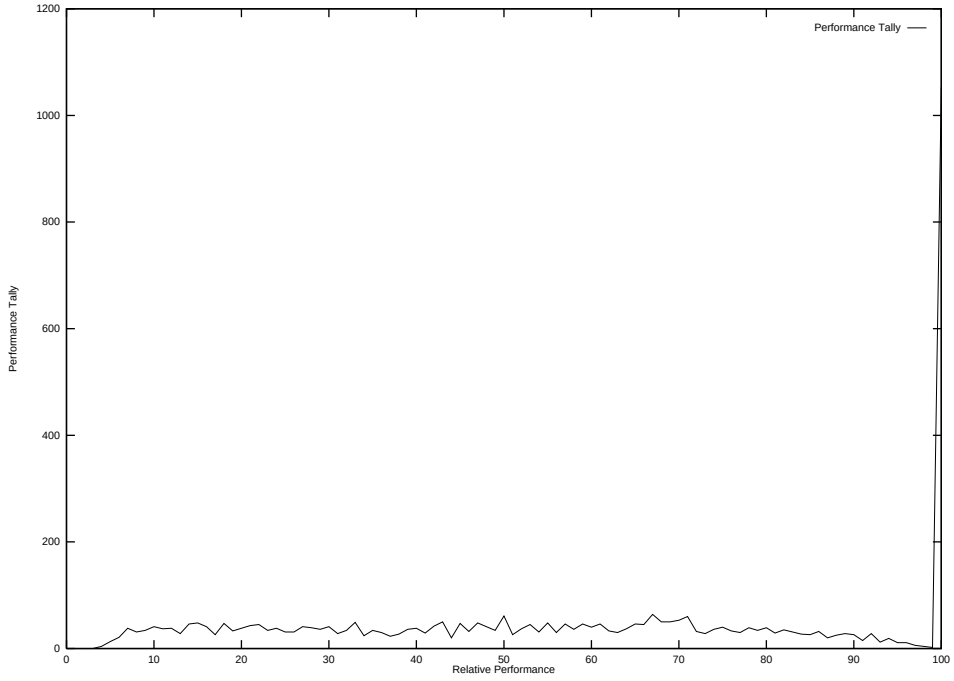

Figure 21: Tally of the Relative Performance by constraint checks of MFC-NORMAL over FC-NORMAL (4,320 problems).

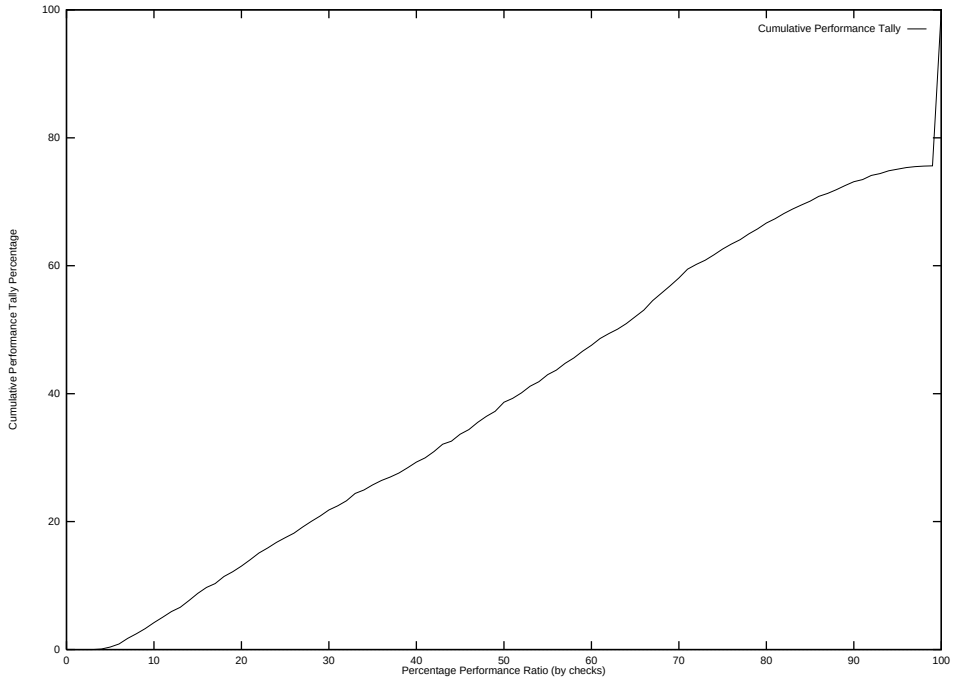

Figure 22: Cumulative Relative Performance by constraint checks of MFC-NORMAL over FC-NORMAL (4,320 problems).

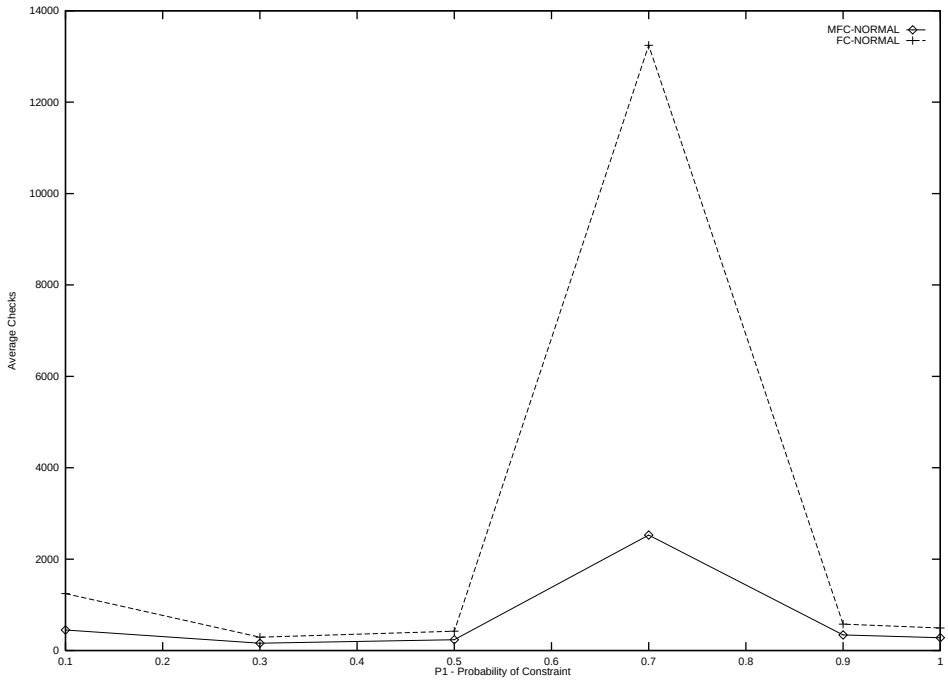

Figure 23: Comparison of MFC-NORMAL vs. FC-NORMAL by the average number of constraint checks performed by each algorithm broken down by P1 (4,320 problems).

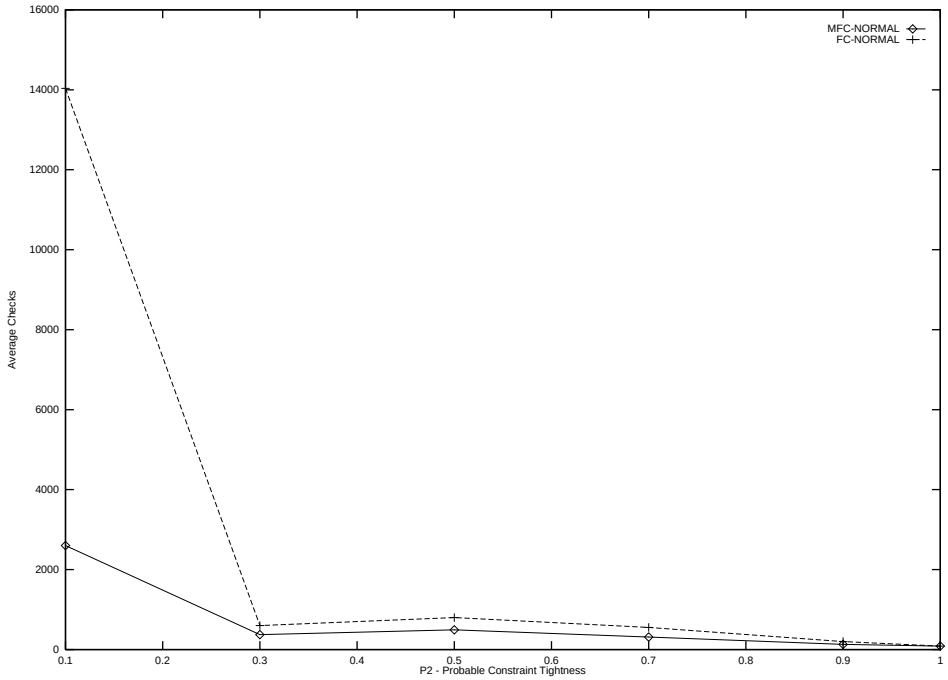

Figure 24: Comparison of MFC-NORMAL vs. FC-NORMAL by the average number of constraint checks performed by each algorithm broken down by P2 (4,320 problems).

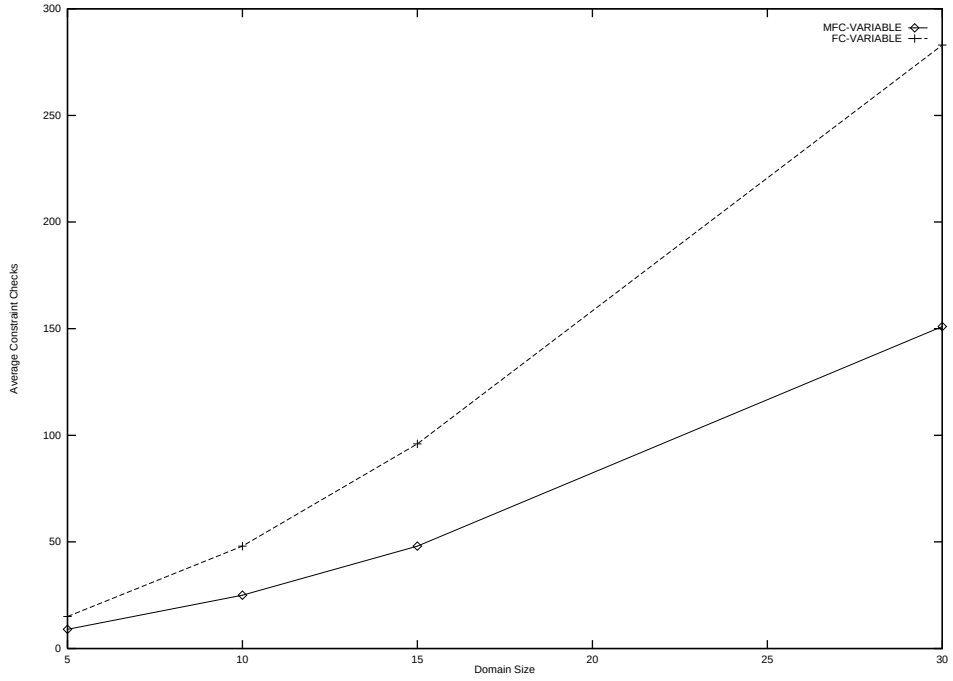

Figure 25: Comparison of MFC-VARIABLE vs. FC-VARIABLE by the average number of constraint checks performed by each algorithm broken down by domain size (4,320 problems).

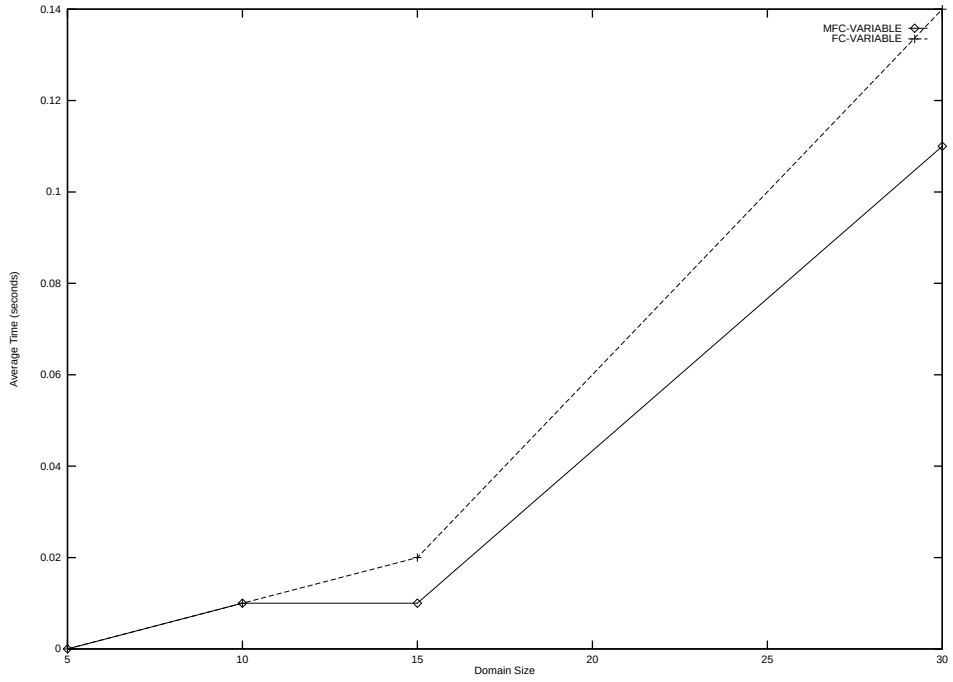

Figure 26: Comparison of MFC-VARIABLE vs. FC-VARIABLE by the average time used (in seconds) by each algorithm broken down by domain size (4,320 problems).

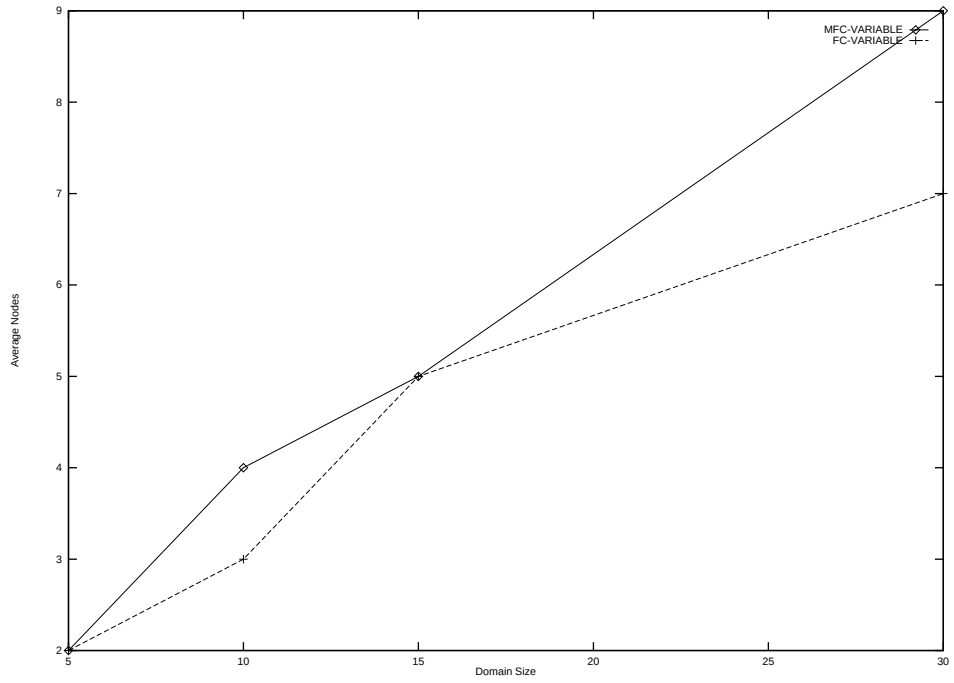

Figure 27: Comparison of MFC-VARIABLE vs. FC-VARIABLE by the average number of nodes visited by each algorithm broken down by domain size (4,320 problems).

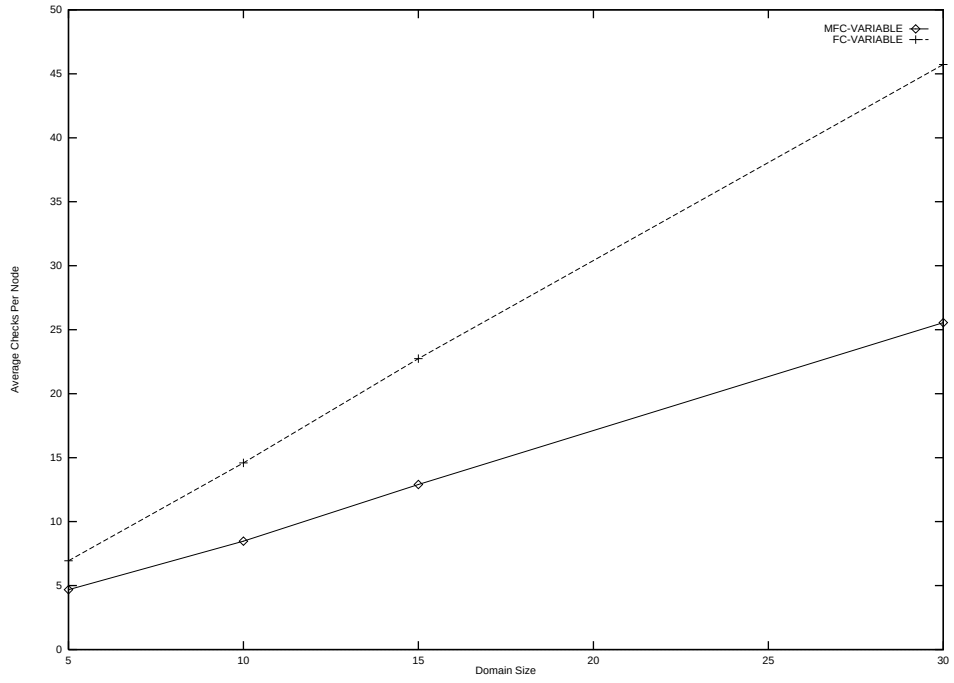

Figure 28: Comparison of MFC-VARIABLE vs. FC-VARIABLE by the average number of constraint checks performed per node by each algorithm broken down by domain size (4,320 problems).

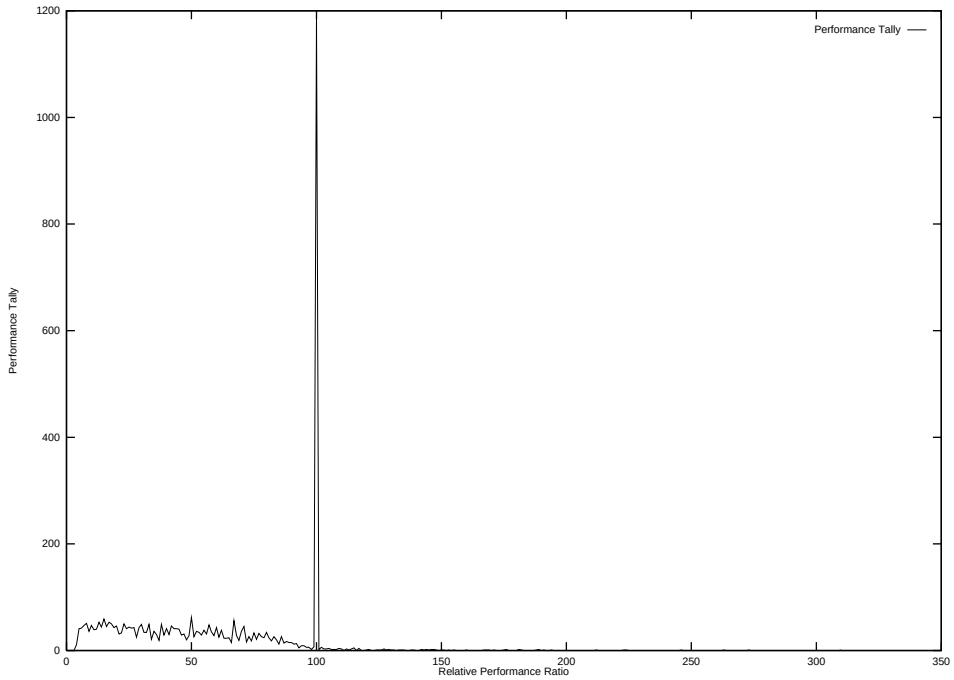

Figure 29: Tally of the Relative Performance by constraint checks of MFC-VARIABLE over FC-VARIABLE (4,320 problems).

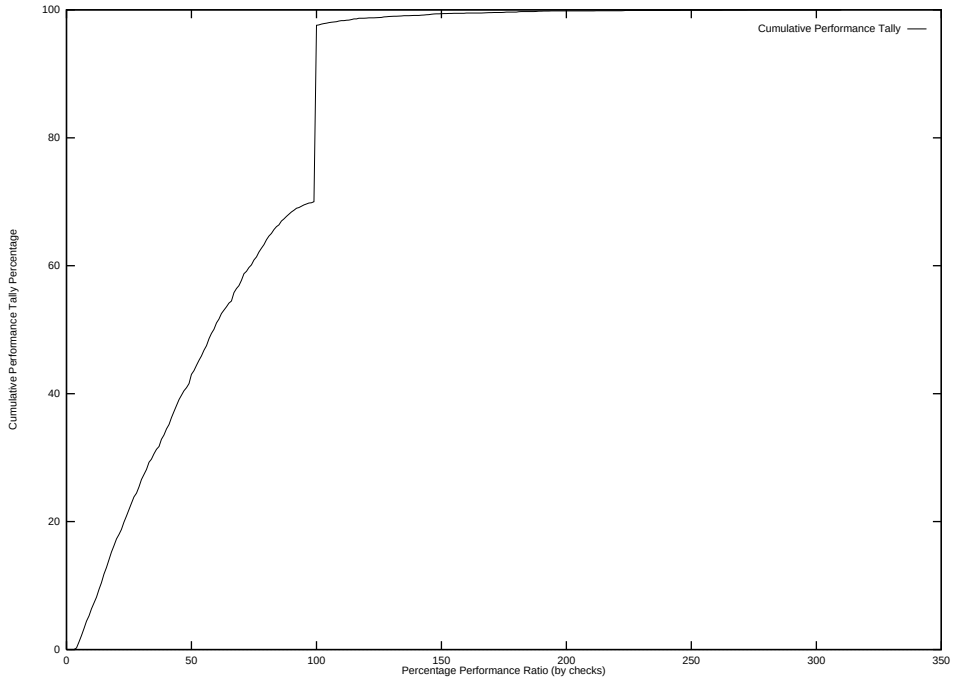

Figure 30: Cumulative Relative Performance by constraint checks of MFC-VARIABLE over FC-VARIABLE (4,320 problems).

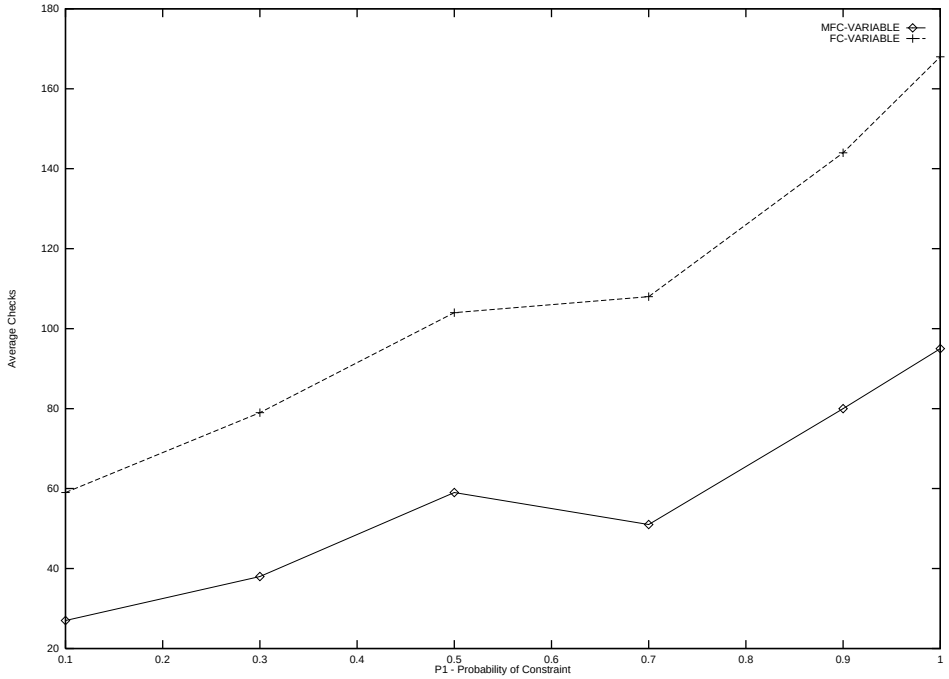

Figure 31: Comparison of MFC-VARIABLE vs. FC-VARIABLE by the average number of constraint checks performed by each algorithm broken down by P1 (4,320 problems).

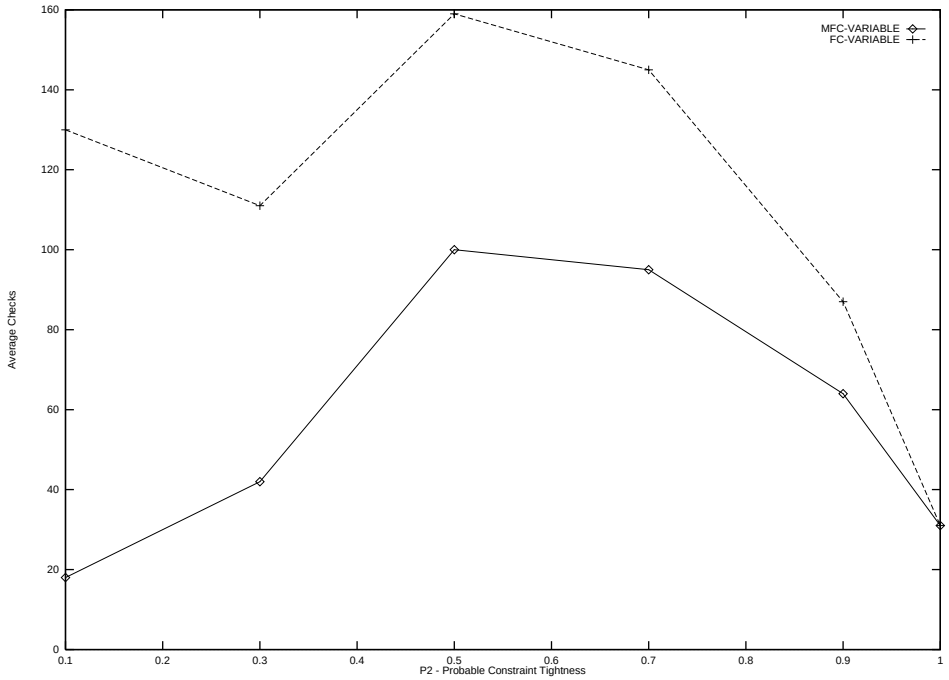

Figure 32: Comparison of MFC-VARIABLE vs. FC-VARIABLE by the average number of constraint checks performed by each algorithm broken down by P2 (4,320 problems).

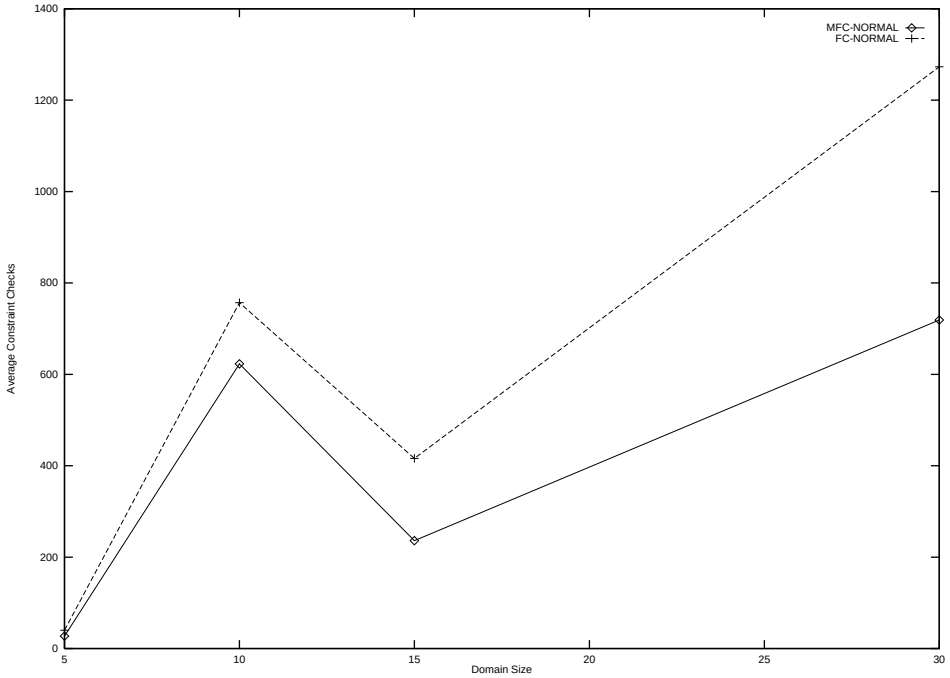

Figure 33: Comparison of MFC-NORMAL vs. FC-NORMAL by the average number of constraint checks performed by each algorithm broken down by domain size (12,000 problems).

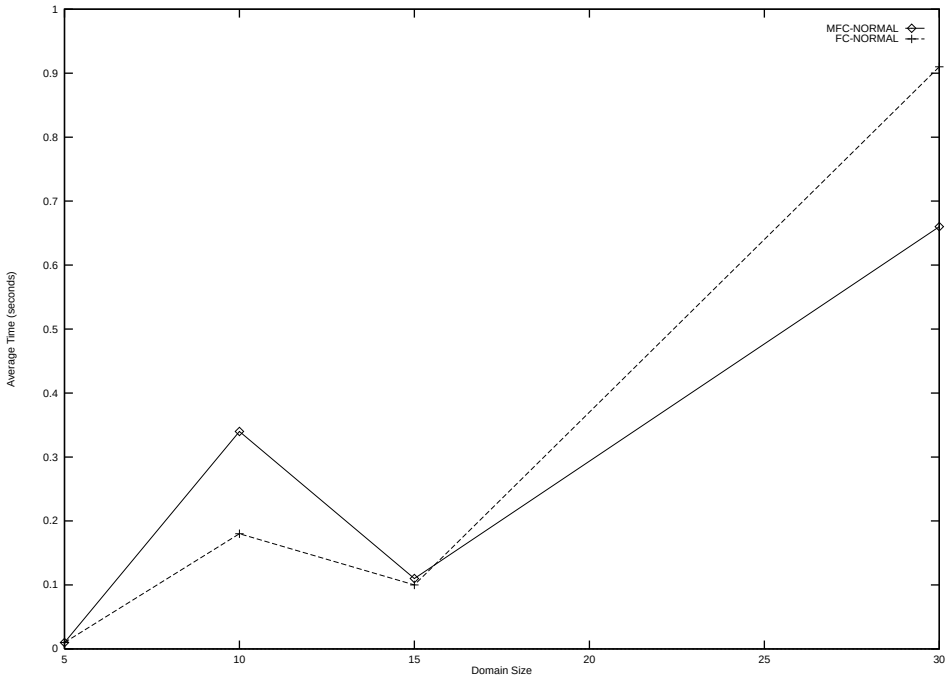

Figure 34: Comparison of MFC-NORMAL vs. FC-NORMAL by the average time used (in seconds) by each algorithm broken down by domain size (12,000 problems).

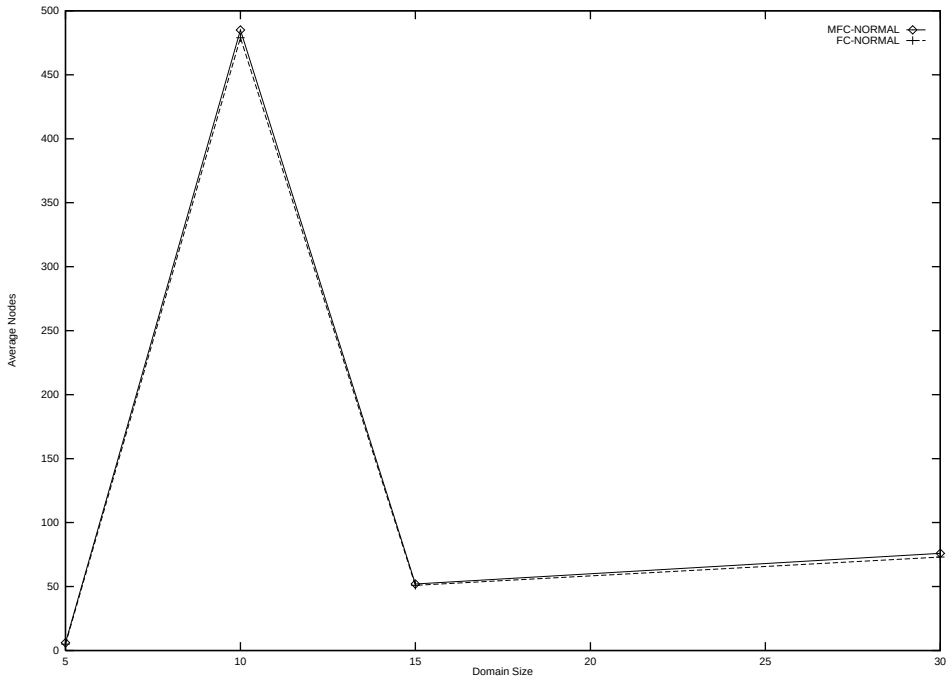

Figure 35: Comparison of MFC-NORMAL vs. FC-NORMAL by the average number of nodes visited by each algorithm broken down by domain size (12,000 problems).

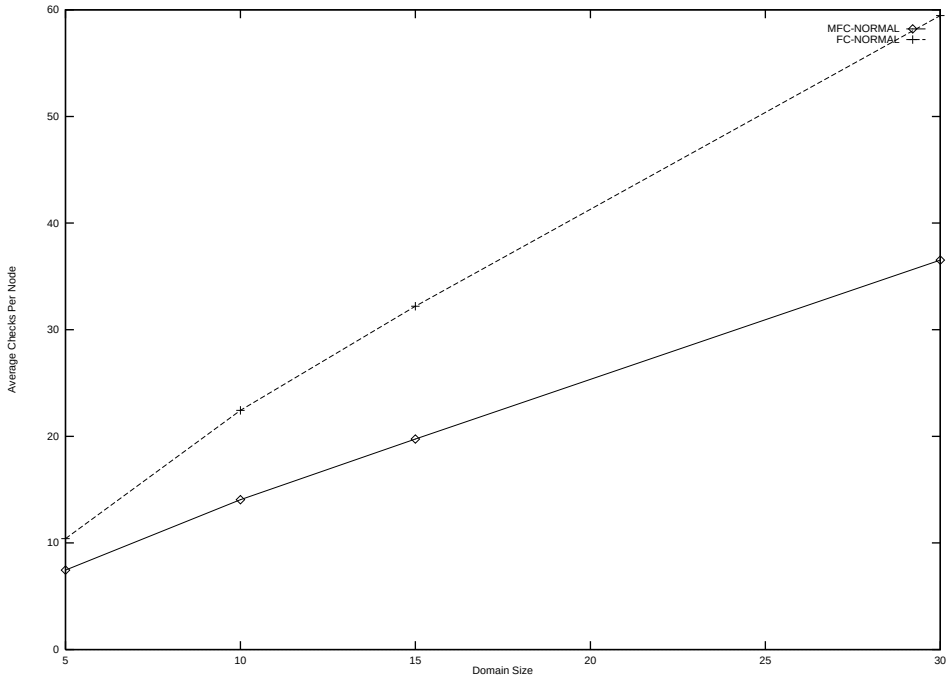

Figure 36: Comparison of MFC-NORMAL vs. FC-NORMAL by the average number of constraint checks performed per node by each algorithm broken down by domain size (12,000 problems).

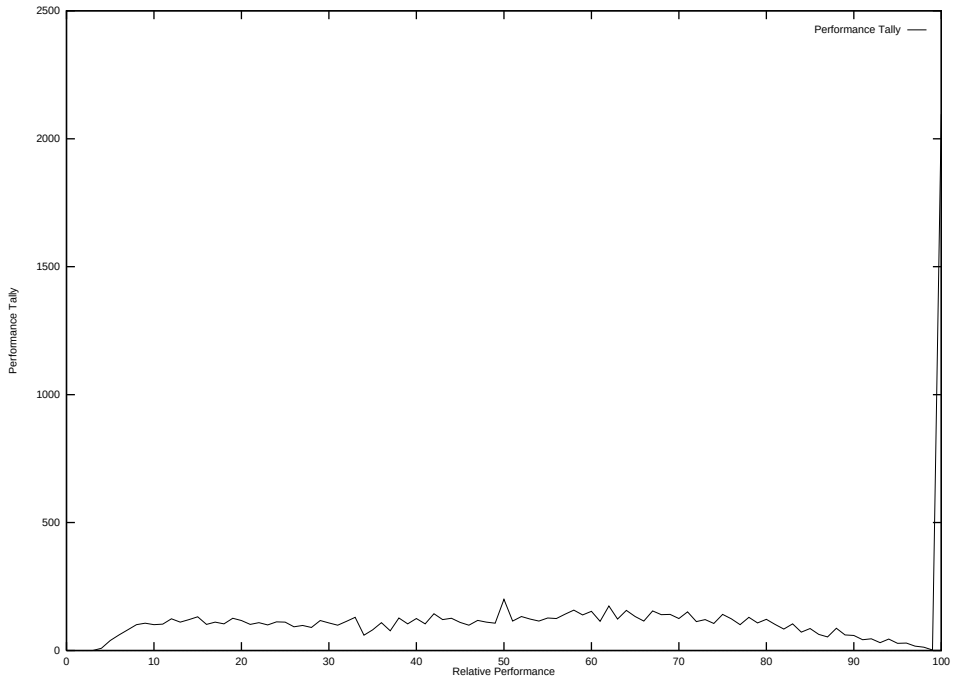

Figure 37: Tally of the Relative Performance by constraint checks of MFC-NORMAL over FC-NORMAL (12,000 problems).

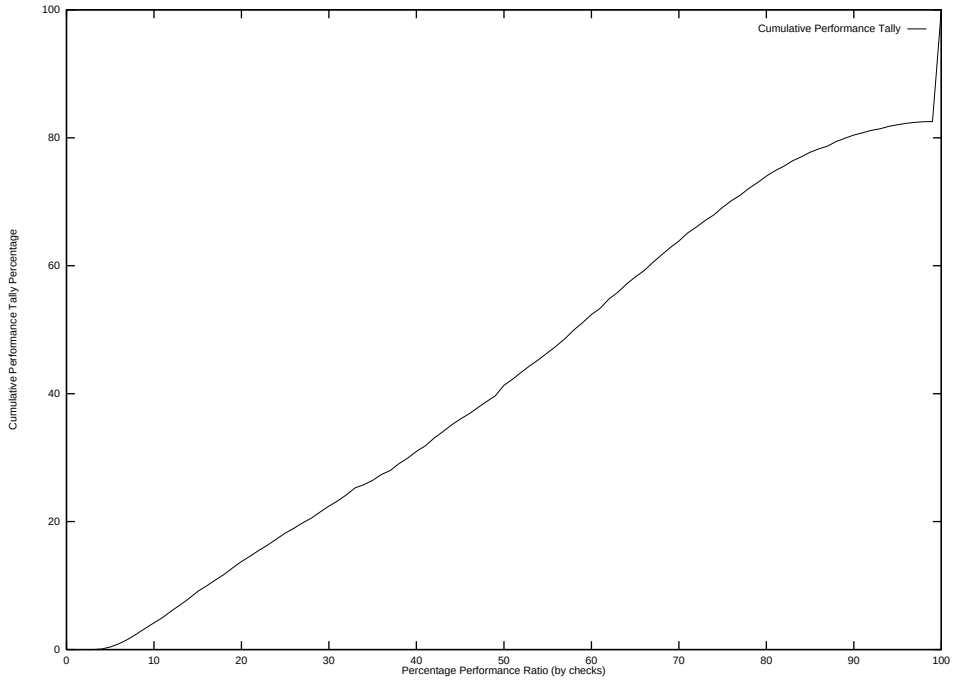

Figure 38: Cumulative Relative Performance by constraint checks of MFC-NORMAL over FC-NORMAL (12,000 problems).

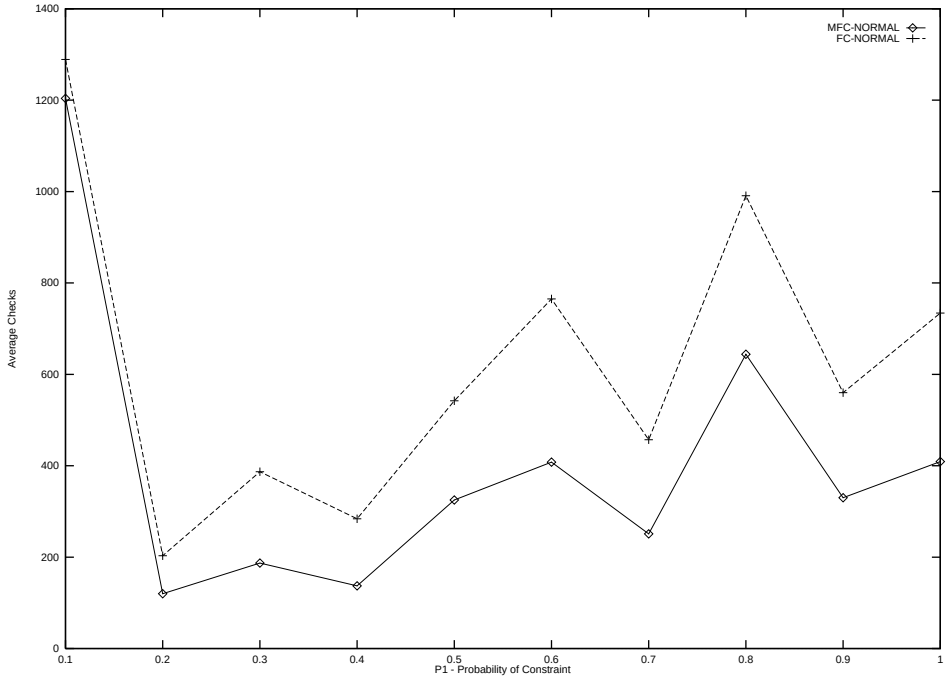

Figure 39: Comparison of MFC-NORMAL vs. FC-NORMAL by the average number of constraint checks performed by each algorithm broken down by P1 (12,000 problems).

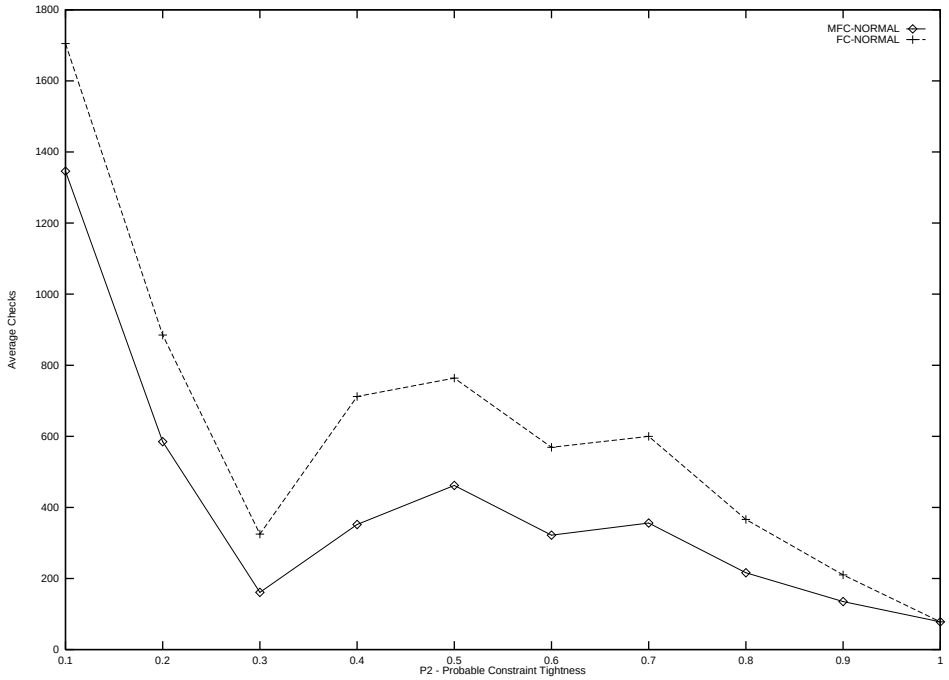

Figure 40: Comparison of MFC-NORMAL vs. FC-NORMAL by the average number of constraint checks performed by each algorithm broken down by P2 (12,000 problems).

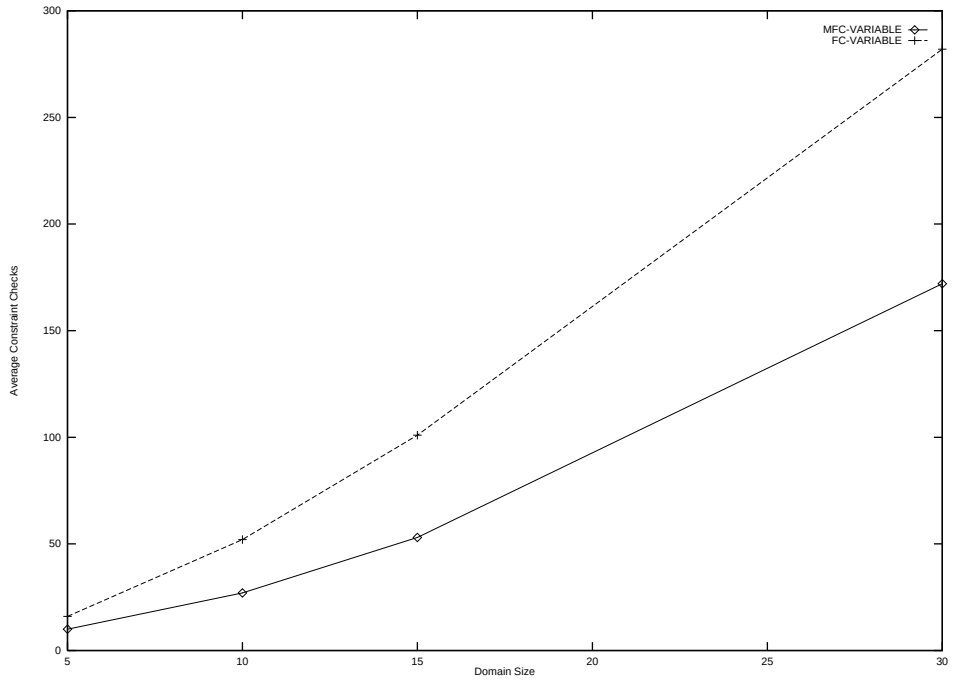

Figure 41: Comparison of MFC-VARIABLE vs. FC-VARIABLE by the average number of constraint checks performed by each algorithm broken down by domain size (12,000 problems).

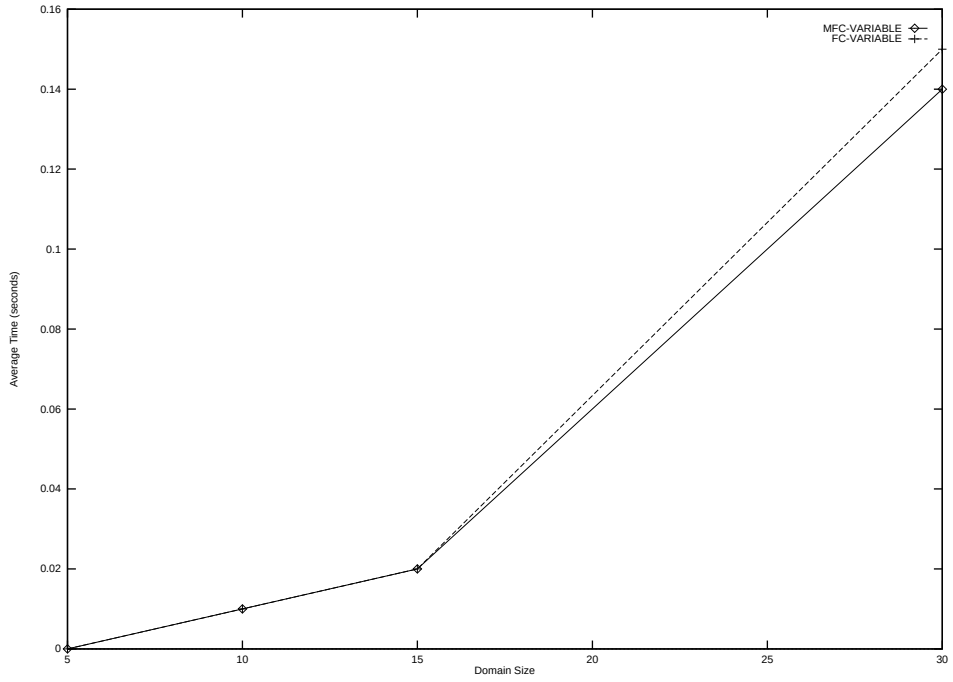

Figure 42: Comparison of MFC-VARIABLE vs. FC-VARIABLE by the average time used (in seconds) by each algorithm broken down by domain size (12,000 problems).

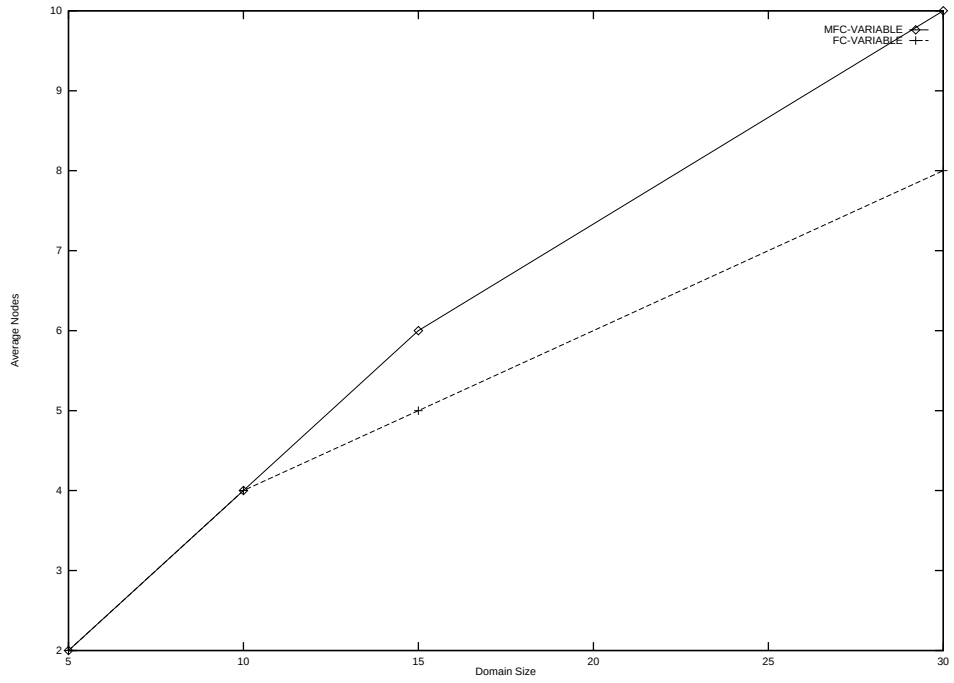

Figure 43: Comparison of MFC-VARIABLE vs. FC-VARIABLE by the average number of nodes visited by each algorithm broken down by domain size (12,000 problems).

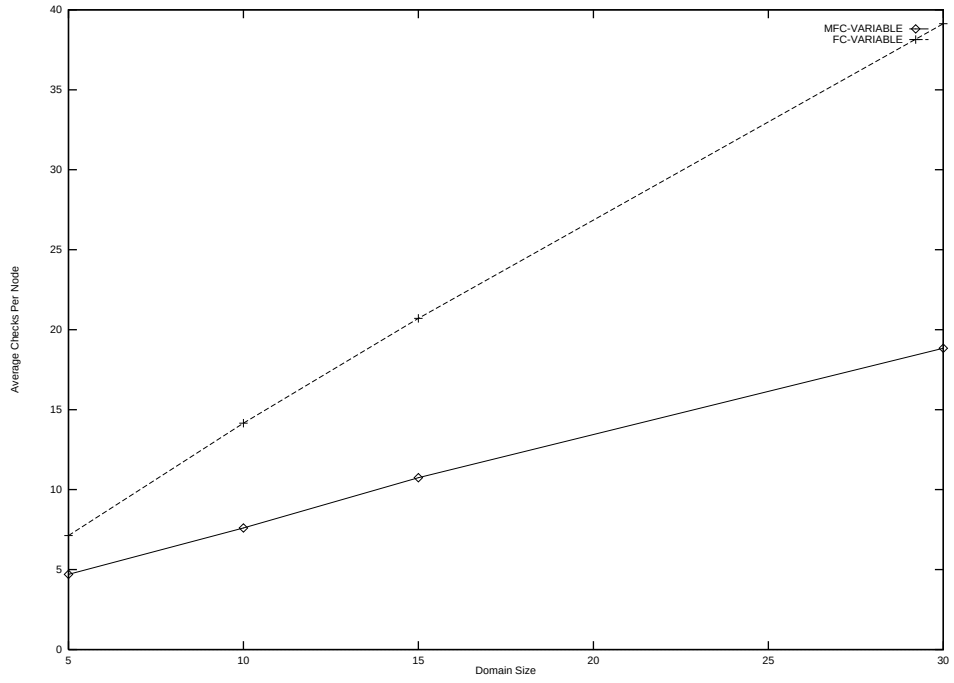

Figure 44: Comparison of MFC-VARIABLE vs. FC-VARIABLE by the average number of constraint checks performed per node by each algorithm broken down by domain size (12,000 problems).

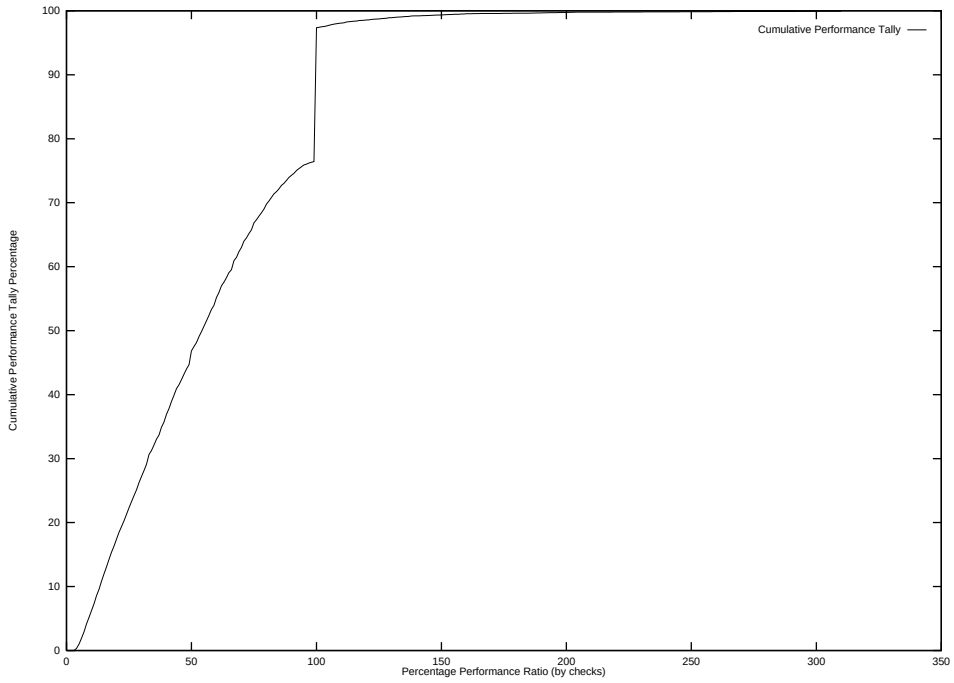

Figure 45: Tally of the Relative Performance by constraint checks of MFC-VARIABLE over FC-VARIABLE (12,000 problems).

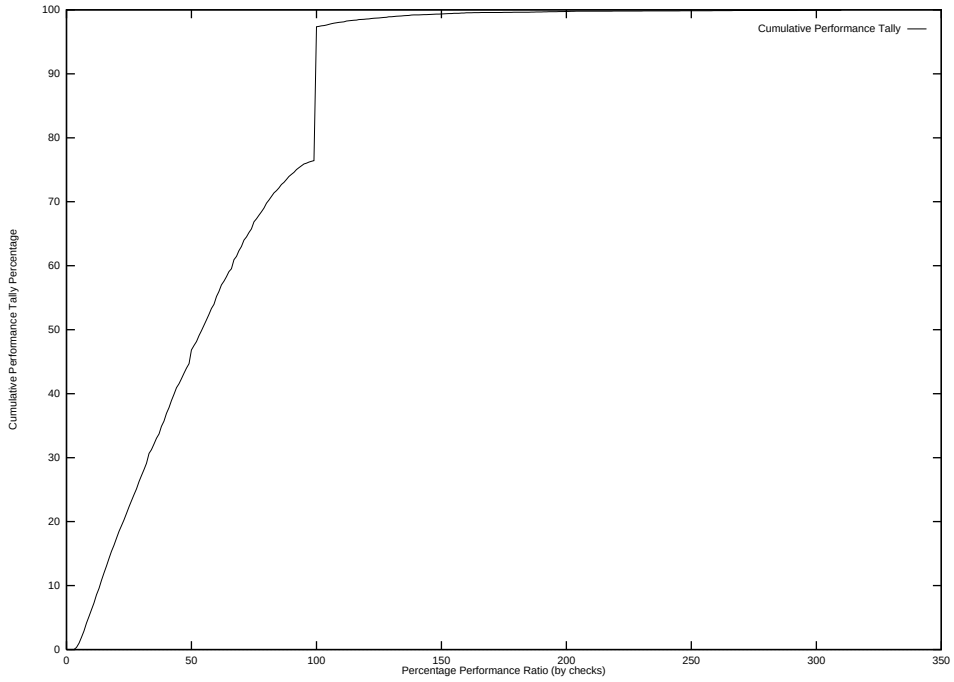

Figure 46: Cumulative Relative Performance by constraint checks of MFC-VARIABLE over FC-VARIABLE (12,000 problems).

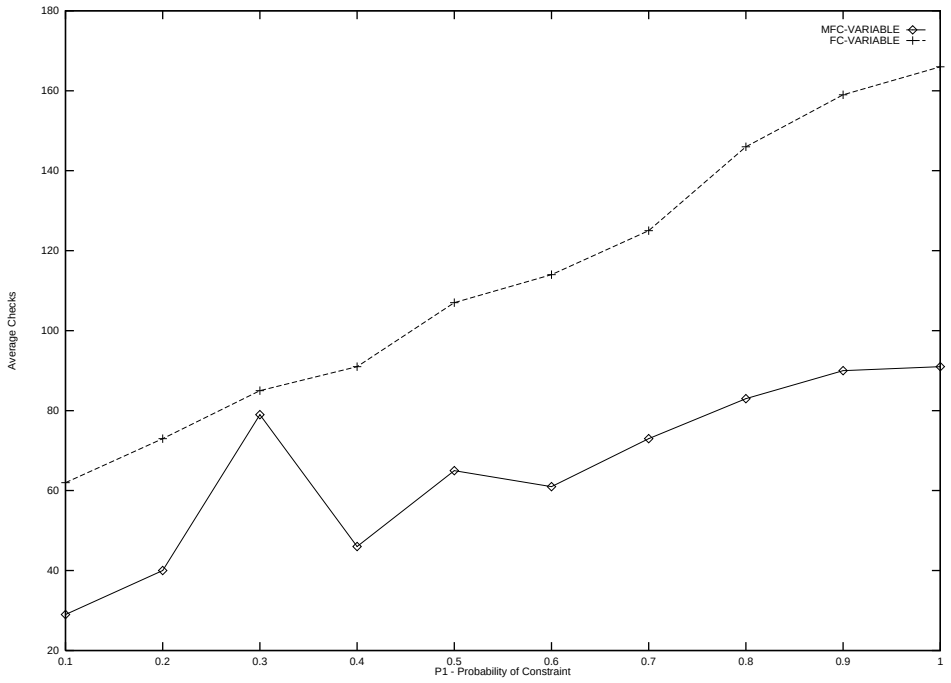

Figure 47: Comparison of MFC-VARIABLE vs. FC-VARIABLE by the average number of constraint checks performed by each algorithm broken down by P1 (12,000 problems).

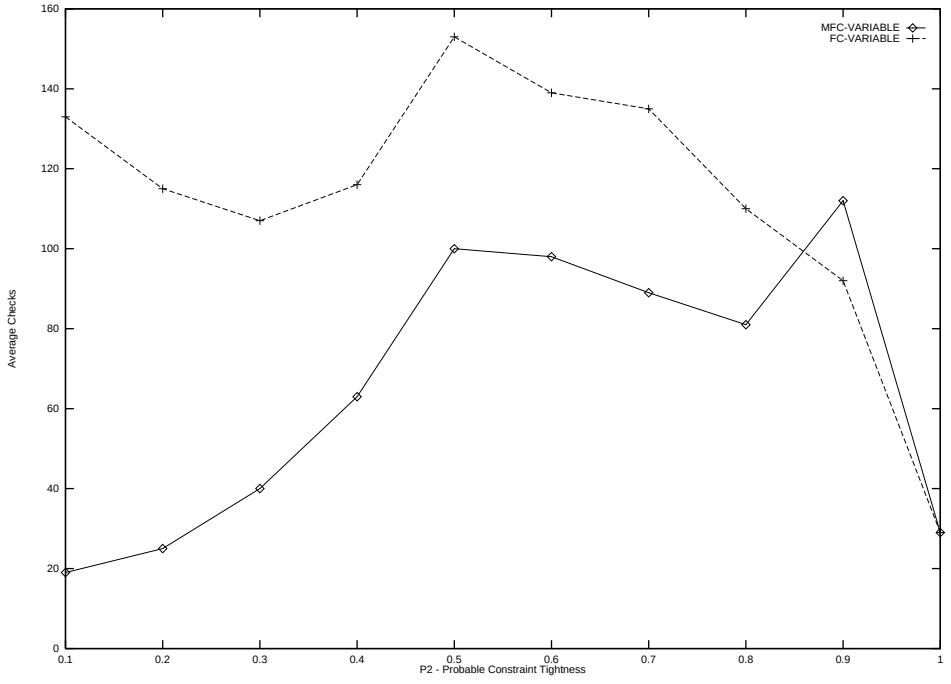

Figure 48: Comparison of MFC-VARIABLE vs. FC-VARIABLE by the average number of constraint checks performed by each algorithm broken down by P2 (12,000 problems).

References

[1] J. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms.* PhD thesis, Carnigie-Mellon University, 1979. CMU-CS-79-124.

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

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

[4] B. Nadel. Tree Search and Arc Consistency in Constraint Satisfaction Algorithms. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, chapter 9, pages 287–342. Springer-Verlag, 1988.

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

[6] P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. Technical Report AISL-46-91, University of Strathclyde, 1991.

[7] P. Prosser. Backjumping Revisited. Technical Report AISL-47-92, University of Strathclyde, 1992.