1996

Minimal Forward Checking--a Lazy Constraint Satisfaction Search Algorithm: Experimental And Theoretical Results

Michael James Dent

Minimal Forward Checking — A Lazy Constraint Satisfaction Search Algorithm: Experimental and Theoretical Results

by

Michael James Dent

Department of Computer Science

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario
July 1996

ISBN 0-612-15040-2

ABSTRACT

Many problems that occur in Artificial Intelligence and Operations Research can be naturally represented as Constraint Satisfaction Problems (CSPs). One of the most popular backtracking search algorithms used to solve CSPs is called Forward Checking (FC). FC performs a limited amount of lookahead during its search attempting to detect future inconsistencies thereby avoiding inconsistent parts of the search tree. In this thesis we describe a new backtracking search algorithm called Minimal Forward Checking (MFC) which maintains FC's ability to detect inconsistencies but which is lazy in its method of doing so. We prove that MFC is sound and complete. We also prove that MFC and FC visit the same nodes in the search tree. Most significantly, we prove that MFC's worst case performance in terms of number of constraint checks performed (the common measure of performance of these algorithms) is the number of constraint checks performed by FC. We then describe how the MFC algorithm can be seen as one algorithm in a family of lazy CSP search algorithms.

As theoretical results on the average case complexity for CSP search algorithms are extremely difficult to derive, empirical comparisons need to be performed. A commonly used testbed is randomly generated problems drawn from a standard model of binary CSPs at a specific location known to contain problems that are relatively hard to solve. We generalize the standard model of binary CSPs and show how to find problems in this model that are relatively hard to solve. We also show that these "hard problems" are of similar hardness or harder than hard problems drawn from the standard model especially as the problem size grows and the problem has a relatively sparse structure. We perform large empirical studies of many CSP search algorithms including variants of MFC and FC with non-chronological backtracking and variants of the Fail First heuristic on two testbeds of hard random problems, each drawn from

one of the two models. Our empirical comparisons on both testbeds indicate that the average case performance of algorithms based on MFC are better than all the other algorithms in the comparison in terms of the number of constraint checks performed.

Keywords: constraint satisfaction problems, search algorithms, forward checking algorithm, minimal forward checking algorithm, hard random problems

To my family

ACKNOWLEDGEMENTS

There are a number of people that deserve recognition for their contribution to this thesis. Starting on the academic side, Patrick Prosser gave me the initial kick start that I needed many years ago, providing me with his current suite of search algorithms. Bernard Nadel also provided me with encouragement and his suite of algorithms. Barbara Smith has been a constant help in the development of my model of hard problems. Her insightful comments have greatly improved my results. Christian Bessière was brave enough to read my first paper on MFC and gave me the encouragement I needed. I would also like to thank the many people named "Anonymous" who have contributed value comments on submitted drafts of my papers.

I would especially like to thank my external examiner Peter van Beek whose review of my thesis made me think it was all worth it. I would also like to thank my other examiners, Jim Mullin, Mike Bauer, and Mike Dawes for their valuable comments and for making my defense (and party afterwards) a happy occasion.

I have been "in contact" with UWO even though I've been away from London for 3 years now because of the efforts of Hector Levesque at the University of Toronto and Fahiem Bacchus at the University of Waterloo. They both provided me with an internet connection I so desperately needed! I would also like to thank Linus Torvalds for developing Linux, without which my life would have been hell.

I would like to thank the department of Computer Science for its funding, and NSERC (grant OGP0036853), IRIS (B-5) and ITRC for partial funding. I would also like to thank the graduate secretary Ursula Dutz for her kindness and support.

Finally, I would **especially** like to thank my supervisor Dr. Robert Mercer. He has been a constant source of inspiration and guidance. I have had so many enjoyable experiences, in San Diego, Estonia, France, Vancouver, Montreal, Seattle, France

again. I have made numerous contacts and associations with people in my area simply because Bob has made it possible. I can't thank him enough. I am especially indebted to him for "getting it wrong" about what was needed to improve FC. If I hadn't tried to get something simpler than FC I would never have found MFC. Bob has become family to my wife and I, we like him even if he does talk and laugh in his sleep like the little kid he is. I would also like to acknowledge the contributions of Iggy who is a "close friend" of Bob's.

On the personal side, I would like to thank my family. My Dad for his inquisitive mind and humour, my Mom for her kindness and humour (when she chooses to show it), my brother Matt for endless (fun) arguments, and my Grandma who is one of the most quick witted persons I know. All four have been a constant source of love and encouragement. My grandfather, who passed away many years ago, is now a sad memory but still a source of strength. I would like to thank my mother and father-in-law for giving me love and care and my wife Mei! Finally, I would like to thank Min Wei for his support and for his comments on my proposal.

Throughout my thesis, my closest friends Gary and Ann (and Jade when she isn't telling me "NO", and now Roxy who loves to sleep on my stomach), Melissa, Khun Yee, Ophe, and Stan (the deathmatch Man) have given me much love and support.

Finally, my thesis could never have been finished without the love and support of my wife Mei Wei Dent. She has been a source of invaluable advice and reason over the years. Whenever I get fuzzy headed she always seems to know what to do. I have learned that I am no match for her mathematical or reasoning skills! The best day of my life was the day that I met her.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

> "The author once waited all night for the output from such a (backtracking) program, only to discover that the answers would not be forthcoming for about 10^6 centuries." Donald Knuth

1.1 Introduction

Many problems in Artificial Intelligence and Operations Research can be expressed as Constraint Satisfaction Problems (CSPs)[22, 69, 74, 76, 82, 108]. A CSP is represented by a set of variables, a set of finite discrete domains for those variables, and a set of constraints over those variables. A solution to a CSP is an assignment of values to variables which satisfies all the given constraints. In this thesis we study CSPs in their binary form where all constraints have arity 2, that is each constraint is between two variables.

A simple example of a CSP is the graph colouring problem. One is given a graph and a set of colours. The objective is to colour the graph such that no adjacent node has the same colour. In terms of a CSP, the variables represent the nodes of the graph, the domains are the available colours, and the constraints ensure that no adjacent nodes (variables) are assigned the same colour. Graph colouring is a known NP-complete problem[46]. As this example suggests, CSPs are also NP-complete problems[46, 60].

CSPs (or a close relative) are a natural representation for problems in domains as diverse as vision[75, 117], software diagnosis[98], music composition[85], lexical acquisition[100], scheduling[12, 38, 78, 95, 101, 121], planning[107], automotive transmission design[83, 84], temporal reasoning[26, 110], belief maintenance[23, 20], graph problems[77], and puzzles[54, 61, 90]. Good surveys of the area can be found in [22, 69, 76, 108].

CSPs are also an integral part of the Constraint Logic Programming (CLP) languages[64]. In the CLP framework, unification is replaced by the more general concept of constraint solving over some computational domain[112] (for example, Booleans, rational numbers, infinite lists, finite domains, or real numbers). Many of the basic ideas underlying logic programming and CLP have their roots in a programming language called ABSYS[14, 37] which performed equational constraint solving over "assertions". Modern examples of CLP languages are CHIP[111], CLP($\mathcal{R}$)[65], and Prolog III[15]. These languages have been successfully applied to a number of real life problems, for example, the cutting-stock problem[36, 111], option trading[72], and digital circuit design and testing[15, 62, 99]. A general overview of CLP can be found in [14, 111, 114].

One of the simplest methods of solving a CSP is chronological BackTracking (BT) search[10, 55]. However, it is well known that backtracking search suffers from a pathological phenomenon called "thrashing"[74] in which the backtracking search explores subtrees that cannot possibly contain a solution. In the worst case, chronological backtracking search algorithms perform an exponential number of steps.

A number of backtracking search algorithms have been developed to solve CSPs (called *CSP search algorithms*) which attempt to avoid this thrashing behavior. For example, BackJumping(BJ), Conflict-Directed Backjumping (CBJ)[90], Forward Checking (FC)[55, 61, 77], and maintaining arc-consistency (MAC)[47, 82, 97]. One of the most successful of these CSP search algorithms is FC. Theoretically, the worst case behavior of FC is the same as BT. However, many empirical studies have shown that FC can on average dramatically outperform BT[61, 77, 82, 90, 113].

Although the FC algorithm has been known since the mid 1960's[55] and redis-

covered in the late 1970's[61, 77] there have been few improvements made to it. One reason for this may have been the belief in the research community that FC performed the least amount of extra work necessary to avoid some of the thrashing behavior and still get fairly efficient behavior. In this thesis we re-examine this belief and derive a new algorithm called Minimal Forward Checking (MFC) which substantially outperforms FC. The first part of our thesis describes the MFC algorithm and investigates the relationship of this new algorithm to a number of well known search algorithms. We derive theoretical relationships and show empirically on "hard" randomly generated problems that MFC performs substantially better than FC.

As theoretical results on the average case complexity for CSP search algorithms are extremely difficult to obtain, empirical studies need to be performed to compare the general performance of the algorithms[25, 82]. Previous empirical studies using specific problems, such as the n-queens problem[2, 47, 61, 115] or the zebra problem[90], are not convincing as the results are only representative for one problem. Previous empirical studies using randomly generated problems[28, 29, 47, 61, 115] are unconvincing as the "random problems" generated were usually easy to solve and therefore unable to convincingly differentiate between the different algorithms. Only recently has it been understood how to create random problems that are relatively hard to solve. The second part of our thesis investigates how these "hard random problems" are created. We generalize the current method used to create these hard random problems and derive a new class of random problems that are empirically shown to be as hard or harder to solve by the best algorithms than those previously known. We use the current method of generating hard random problems and our new class of hard random problems to empirically compare the algorithms described in this thesis.

We begin our thesis by describing the algorithms which are the foundation of the MFC algorithm. Section 1.2 gives a formal definition of a binary CSP and the terminology used to describe a chronological backtracking search. Section 1.3 gives the formal definition and example executions of the algorithms underlying the MFC algorithm. Section 1.4 describes consistency enforcing algorithms and the notion of a hybrid search algorithm. Section 1.5 discusses past theoretical and experimental

comparisons. Finally, Section 1.6 gives a summary for this introductory chapter.

The rest of the thesis is divided into two parts. In the first part we describe the MFC algorithm and give a number of theoretical results (Chapter 2); we discuss hard random problems and empirically compare MFC with previously known algorithms using these hard random problems (Chapter 3); and we discuss a number of extensions that can be made to MFC to increase its performance and give a comprehensive empirical comparison of all the algorithms derived in this thesis (Chapter 4). In the second part of our thesis we describe our new model of hard random problems and empirically compare the new model with the old model (Chapter 5); we then use the new model to empirically compare some of the algorithms discussed in this thesis (Chapter 6); finally, we give our conclusions and future work (Chapter 7).

1.2 Constraint Satisfaction Problems

In this thesis we study CSPs in their binary form where all constraints have arity 2, that is each constraint is between two variables. Most CSP search algorithms have been developed within the framework of binary CSPs and we continue this tradition. This restriction on each constraints arity is only a restriction on the formulation of a problem. Although it is true that non-binary CSPs can be translated into equivalent[1] binary CSPs[96] the translated CSP may not be an appropriate representatic. We leave exploration of CSP search algorithms tailored to non-binary CSPs for future work.

Definition 1.1 A *binary CSP* is represented with a set of variables, $V = \{v_1, \ldots, v_n\}$, a set of finite discrete domains $D = \{d_1, \ldots, d_n\}$ where each $d_i = \{v_i^1, v_i^2, \ldots, v_i^{m_i}\}$, $m = \max(m_i)$, and a set of symmetric constraints $C = \{c_{i,j} | 1 \leq i < j \leq n\}$ where each $c_{i,j}$ is a subset of $d_i \times d_j$. If $(v_i^k, v_j^l) \in c_{i,j}$ then the assignment $\{v_i \leftarrow v_i^k, v_j \leftarrow v_j^l\}$ is consistent. If $c_{i,j}$ contains all elements of $d_i \times d_j$ the constraint is called a *trivial constraint* as all possible assignments are consistent. A *solution* is an assignment

[1] Equivalent in the sense that it is possible to obtain the solution of one from the other and vice versa.

$\{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_n \leftarrow v_n^{s_n}\}$ where for all i, j, $1 \leq i < j \leq n$, $\{v_i \leftarrow v_i^{s_i}, v_j \leftarrow v_j^{s_j}\}$ is consistent. If the CSP has a solution we say that it is *soluble*. If the CSP has no solution we say that it is *insoluble*.

Definition 1.2 It is sometimes convenient to picture a binary CSP as a *constraint graph* where a variable in the CSP is represented by a node in the graph, and a non-trivial constraint between two variables is an edge between the nodes corresponding to the variables.

There are two basic questions that can be asked about a CSP. Does the CSP have a solution and how many solutions does the CSP have ? We are only interested in the first question, that of finding a solution if it exists. A systematic backtracking search depends on the notion of extending a partial solution:

Definition 1.3 A *partial solution* is a *partial assignment* $\{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_k \leftarrow v_k^{s_k}\}$ where for all i, j, $1 \leq i < j \leq k$, $\{v_i \leftarrow v_i^{s_i}, v_j \leftarrow v_j^{s_j}\}$ is consistent. A partial solution $\{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_k \leftarrow v_k^{s_k}\}$ can be *extended* to a new partial solution $\{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_k \leftarrow v_k^{s_k}, v_{k+1} \leftarrow v_{k+1}^{s_{k+1}}\}$ if for all $i \leq k$, $\{v_i \leftarrow v_i^{s_i}, v_{k+1} \leftarrow v_{k+1}^{s_{k+1}}\}$ is consistent. If $k = n$ then the partial assignment is a *solution* to the CSP.

Although the definition of a CSP does not enforce an ordering on the domains, in order to fairly compare implementations of search algorithms we must have an ordering.

Definition 1.4 An *ordered domain* is a domain which is in the original order given in a problems definition.

We assume that all domains are ordered domains.

Consider a graph colouring problem where you are given 4 nodes that are adjacent to each other and are told that the first node can only be coloured red, the second node with green or orange, the third node with blue or green, and the fourth node with green, blue or red. Does this problem have a solution in which adjacent nodes are coloured differently? To represent this problem as a CSP, we first define four

variables. $\{v_1, v_2, v_3, v_4\}$ which represent the four nodes. The domains of the four variables are

$$d_1 = \{r\}, d_2 = \{g, o\}, d_3 = \{b, g\}, d_4 = \{g, b, r\}$$

where we just use the first letter of each colour to represent the domain elements. Finally we define a set of constraints which enforce the general constraint that adjacent nodes (variables) cannot be assigned the same colour.

$$c_{1,2} = \{(r, g)(r, o)\}$$

$$c_{1,3} = \{(r, b)(r, g)\}$$

$$c_{1,4} = \{(r, g)(r, b)\}$$

$$c_{2,3} = \{(g, b)(o, b)(o, g)\}$$

$$c_{2,4} = \{(g, b)(g, r)(o, g)(o, b)(o, r)\}$$

$$c_{3,4} = \{(b, g)(b, r)(g, b)(g, r)\}$$

Constraints are symmetric so for example $c_{2,1} = \{(g, r)(o, r)\}$. A solution to this CSP is $\{v_1 \leftarrow r, v_2 \leftarrow o, v_3 \leftarrow b, v_4 \leftarrow g\}$. We will use this trivial example to illustrate the execution of the algorithms described in the following sections.

Chronological backtracking tree search algorithms perform a systematic search of the possible partial assignments until a solution is found or all possible partial assignments have been searched and no solution is found. The following definitions give terminology we use to describe parts of the search and the assumptions that we make about the search. We assume in these definitions, for reasons of clarity, that the *order of instantiation* is static, that is the order that the algorithms will select variables to assign them values is $v_1, v_2, \ldots, v_i, \ldots, v_n$.

Definition 1.5 The variable that is to be assigned a value, say v_i, is called the *current variable*. The domain of the current variable is called the *current domain*.

Definition 1.6 The variables in the partial assignment $\{v_1 \leftarrow v_1^{s_1}, \ldots, v_{i-1} \leftarrow v_{i-1}^{s_{i-1}}\}$, are called the *past variables*, where each $v_k \leftarrow v_k^{s_k}$ is called the *instantiation* of v_k, and

the variables $\{v_{i+1}, \ldots, v_n\}$ are called the *future variables*. The domains of the past and future variables are called the *past domains* and *future domains* respectively. The past variables are also called the *past instantiations*. When the current variable is instantiated it is called the *current instantiation*.

Definition 1.7 A *future-connected variable* v_j is a future variable v_j that is connected by a non-trivial constraint to the current variable. A *future-connected domain* is a domain d_j for which v_j is a future-connected variable. A *past-connected variable* is a past variable v_k that is connected by a non-trivial constraint to the current variable.

Definition 1.8 A backtracking search is a *search tree* traversal. The *nodes* of the search tree cover all possible partial assignments. The root of the search tree is the empty partial assignment ϵ. The first level of the tree consists of all sets containing a possible assignment for v_1. The second level consists of all possible sets containing a possible partial assignment for v_1 and v_2 where a node on the second level is a child of a node on the first level if it has the same assignment for v_1. This process continues until the last level n which contains the sets of all possible complete assignments. The size of this search tree is $O(m^n)$ nodes. The levels closer to the top are called the *shallower levels* and the levels closer to the bottom are called the *deeper levels*. The partial assignments that are partial solutions are called the *consistent nodes*. The search tree that is made up of consistent nodes is called a *backtrack tree*. We adopt the notational convention that nodes of the search tree are labeled by the partial assignment associated with that node.

Definition 1.9 A backtracking search algorithm *visits* a node if at some point in the search the instantiation of the current variable and the instantiations of the past variable form the set that identifies the node.

Definition 1.10 Given that v_i is the current variable, we say a value $v_j^l \in d_j$ $(j \geq i)$ is *past-consistent* if the assignment $\{v_k \leftarrow v_k^{s_k}, v_j \leftarrow v_j^l\}$ is consistent for all past variables v_k.

Definition 1.11 A *constraint check* is performed whenever an assignment pair $\{v_i \leftarrow v_i^k, v_j \leftarrow v_j^l\}$ is checked to see if it is an element of $c_{i,j}$.

Constraint checks are considered a fair measure of the performance of CSP search algorithms. There are two reasons for this. The first reason is timing results are not considered as a reliable measure as it is very hard to evaluate whether the performance of an algorithm is due to the implementation, the programming language, the operating system, the hardware, or to the algorithm itself. Irrespective of the implementation of an algorithm the number of constraint checks is invariant. The second reason is that constraint checks may involve some unknown amount of computation (although the results of a performed constraint check can be cached at which point it becomes a table lookup). The caveat of using constraint checks as a performance measure is that it gives no indication of the amount of overhead an algorithm incurs manipulating data structures during its search[2]. The cost of manipulating data structures is problem dependent.

1.3 Backtracking Algorithms

In this section we describe a number of chronological backtracking tree search algorithms for CSPs. These algorithms work by generating partial assignments, backtracking when a partial assignment is not a partial solution, or backtracking when the partial assignment is a partial solution but it is somehow discovered that the partial solution can not be extended to a solution. The difference between the algorithms is at what point in the search tree they decide to backtrack and how many constraint checks they perform trying to discover that a partial solution cannot be extended to a solution.

Usually, CSP search algorithms are presented in a recursive style. However some authors[82, 90] have found that it is clearer to present these algorithms in terms of two functions, a forward labeling function used to find an instantiation for the current

[2]If by the problem definition there is a trivial constraint between v_i and v_j then constraint checks between these variables are not counted.

```
function solve-csp(label-fcn,unlabel-fcn,selection-fcn)
    direction ← "forwards"                                          1
    status ← "unknown"                                              2
    ii ← 1                                                          3
    consistent ← True                                               4
    loop while status ="unknown"                                    5
        if consistent then                                          6
            if direction = "forwards"                               7
                select-fcn(ii)                                      8
            label-fcn(ii)                                           9
            if consistent then                                      10
                ii ← ii+1                                           11
                direction ← "forwards"                              12
        else                                                        13
            ii ← unlabel-fcn(ii)                                    14
            direction ← "backwards"                                 15
        if ii > n then status ← "soluble"                           16
        if ii = 1 and (not consistent) then status ← "insoluble"    17
    return(status)                                                  18
```

Figure 1.1: Pseudo-code for the **solve-csp** function.

variable and a backward unlabeling function used to uninstantiate (that is, undo) a formerly successful instantiation. This unraveling of the recursive call corresponding to a forward move and the return corresponding to the backward move makes explicit search knowledge that may be hidden in the procedure stack[90].

Figure 1.1 shows pseudo-code for the **solve-csp** function which is used to call an algorithm's forward and backward labeling functions repeatedly until a solution is found. Function **solve-csp** takes as input a labeling function, an unlabeling function and a selection function. The selection function is used to select a new variable to instantiate. Up to this point we have assumed that the order of instantiation is static. That is, the variables are selected for instantiation in the order $v_1, v_2, \ldots, v_n$. However

this is an unnecessary restriction as it does not matter in which order the variables are instantiated. The selection function allows one to use a *dynamic variable selection heuristic* which we will show later to be quite useful. In function **solve-csp** the variables **direction**, **status**, **consistent** and ii have the following meanings:

direction is assigned the value "**forwards**" or "**backwards**" to indicate whether the search is moving forward (that is, moving deeper in the search tree) to select a new variable or it is moving backwards (that is, moving shallower in the search tree) to give a new value to a variable whose instantiation is no longer consistent.

status is assigned the value "**unknown**" or "**soluble**" or "**insoluble**" to indicate respectively whether the CSP is still being searched, a solution has been found, or no solution exists.

consistent is a global variable that is assigned the Boolean value **True** or **False**. In the labeling function it is used to indicate whether or not the current variable has been successfully instantiated. In the unlabeling function it is used to indicate whether or not there are more values to be tried in the current variable's domain.

ii is the index of the current instantiation. In order to allow for a dynamic variable selection heuristic we must use an indirection array called $\mathbf{refv}_{ii}$ where the variable ii refers to the current instantiation (in the instantiation order) and $\mathbf{refv}_{ii}$ is the variable that has been chosen to be the ii'th instantiation. For example, if the 3rd variable to be instantiated was v_5 then $\mathbf{refv}_3 = 5$. Future variables are kept in $\mathbf{refv}_{ii+1}$ to $\mathbf{refv}_n$. Past variables are kept in $\mathbf{refv}_1$ to $\mathbf{refv}_{ii-1}$. The explicit indirection array for instantiations that we use here was suggested in [2, 115].

Function **solve-csp** loops calling the labeling and unlabeling functions until either a solution is found by the underlying algorithm or it is exhaustively proven that no solution exists. The selection function (line 8) is used only when the algorithm is performing a forward move to try instantiating a new variable. It chooses one of the future variables for instantiation and notifies **solve-csp** of its decision by setting $\mathbf{refv}_{ii}$ equal to that chosen variable. If a variable is uninstantiated and there are values

remaining in its domain, the variable **direction** will be equal to "**backwards**" and the selection function will not be called. **solve-csp** expects that the labeling function will set **consistent** equal to **True** if it finds an instantiation for the current variable and set **consistent** to **False** otherwise. **solve-csp** expects that the unlabeling function returns an index which indicates the new current variable. The value ii is returned if the domain of the variable referenced by $\mathbf{refv_{ii}}$ still has values left to choose from (and **consistent** is set to **True**) or it can be ii-1 otherwise (and **consistent** is set to **False**)[3]. If the value of ii is greater than n then the underlying algorithm is signaling that a solution has been found (line 16) as every variable has been assigned a value. If ii is equal to 1 and there are no further values left in $\mathbf{d_1}$ (that is **consistent** is **False**) then the underlying algorithm is signaling that a complete search has been performed and no solution has been found. It should be obvious that the correctness (that is the soundness and completeness of the underlying algorithm) of **solve-csp** depends on the actions of the labeling and unlabeling functions. For example, if the labeling and unlabeling functions are those for the Generate and Test algorithm (*cf.* Section 1.3.1) then **solve-csp** will explore the whole search tree if no solution exists and will find a solution if one exists.

To make the exposition of the algorithms presented in this thesis easier to understand and to make certain relationships between algorithms obvious we have chosen a simple array data structure that will be used throughout the algorithm descriptions[4].

---

[3]This is not entirely true as otherwise we wouldn't need the unlabeling function to return the next instantiation index. If the algorithm uses an intelligent form of backtracking (non-chronological backtracking), one form of which we will discuss later in the thesis (called Conflict-directed BackJumping or CBJ), the value returned may be an index of any one of the past instantiations.

[4]This data structure may not be the best one for some of the algorithms we describe. For some algorithms it uses more space O(nm) than is required. For others, its lack of sophistication *may* increase the amount of overhead (but not the number of constraint checks) when an instantiation is attempted or undone. One common practice is to loop over a variable's current domain rather than the whole domain. This requires another array which keeps track of the values that have been deleted and introduces the problem of efficiently updating the current domain. In this thesis we are more concerned with a clear and understandable introduction to our algorithms rather than implementation details and efficiency considerations. We discuss alternative data structures for the MFC algorithm in the next chapter. Alternative data structures for the other algorithms can be found in the papers

Definition 1.12 The values of the array $\mathbf{domain}_i^j$ will have the following meanings:

$\mathbf{domain}_i^j = 0$ means $\mathbf{v}_i^j$ has not been checked against any past instantiation.

$\mathbf{domain}_i^j = -\mathbf{kk}$ means the instantiation of a past variable indexed by **kk** ($\mathbf{refv}_{kk}$) is inconsistent with value $\mathbf{v}_i^j$. The negative value is used to indicate that the value $\mathbf{v}_i^j$ has been marked as deleted from the domain of $\mathbf{d}_i$. If the algorithm being described does not keep track of which past instantiation caused the deletion of $\mathbf{v}_i^j$ we use the value $-\infty$ to show that the value has been deleted.

$\mathbf{domain}_i^j = \mathbf{kk}$ means the instantiation of a past variable indexed by **kk** ($\mathbf{refv}_{kk}$) is consistent with value $\mathbf{v}_i^j$.

The initial value for each element in $\mathbf{domain}_i^j$ is 0.

The interpretation of the $\mathbf{domain}_i^j$ array we will use in our algorithms is, if the value $\mathbf{domain}_i^j$ is negative then the absolute value of $\mathbf{domain}_i^j$ is the index of the shallowest instantiation with which the value $\mathbf{v}_i^j$ is inconsistent, and if the value of $\mathbf{domain}_i^j$ is positive then the value of $\mathbf{domain}_i^j$ is the index of the deepest instantiation which is consistent with $\mathbf{v}_i^j$. The implication of a positive value for $\mathbf{domain}_i^j$ is that $\mathbf{v}_i$ is consistent with all past instantiations up to the instantiation indexed by $\mathbf{domain}_i^j$.

Definition 1.13 The *pruned domain* of a variable $\mathbf{v}_i$ is the subset of $\mathbf{d}_i$ which is not marked as deleted in the $\mathbf{domain}_i^j$ array. The *pruned domain size* is the number of elements in the pruned domain. A *completely pruned domain* is a pruned domain which has no elements (that is, its pruned domain size is 0). A *completely pruned variable* is a variable whose domain is completely pruned. The *true domain size* of a variable $\mathbf{v}_i$ is the pruned domain size of the variable when each undeleted value is past-consistent.

1.3.1 Generate and Test

The simplest tree search algorithm is called Generate and Test (GT). In GT systematically chosen assignments from $\mathbf{d}_1 \times \mathbf{d}_2 \times \ldots \times \mathbf{d}_n$ are tested to see if they are a

---

referenced.

```
function gt-label(ii)
    i ← refv_ii                                                    1
    consistent ← False                                             2
    for each v_i^l ∈ d_i while (not consistent)                    3
        if domain_i^l = 0 then                                     4
            consistent ← True                                      5
            v_i ← v_i^l                                            6
            if ii=n then                                           7
                for jj = 1 to n − 1 while consistent               8
                    for kk = jj + 1 to n while consistent          9
                        j ← refv_jj                                10
                        k ← refv_kk                                11
                        if (v_j, v_k) ∉ c_j,k then                 12
                            consistent ← False                     13
```

Figure 1.2: Pseudo-code for the GT labeling function.

```
function gt-unlabel(ii)
    i ← refv_ii                                                    1
    hh ← ii − 1                                                    2
    h ← refv_hh                                                    3
    for each v_i^l ∈ d_i                                           4
        domain_i^l = 0                                             5
    domain_h^el(v_h) ← −∞                                          6
    if ∃k domain_i^k = 0                                           7
        then consistent ← True                                     8
        else consistent ← False                                    9
    return(hh)                                                     10
```

Figure 1.3: Pseudo-code for the GT unlabeling function.

Step	d_1	d_2	d_3	d_4	Checks
0	r	g o	b g	g b r	0
1	$v_1 \leftarrow r$	$v_2 \leftarrow g$	$v_3 \leftarrow b$	$v_4 \leftarrow g$	5
2	$v_1 = r$	$v_2 = g$	$v_3 = b$	$v_4 \leftarrow b$	6
3	$v_1 = r$	$v_2 = g$	$v_3 = b$	$v_4 \leftarrow r$	3
4	$v_1 = r$	$v_2 = g$	$v_3 \leftarrow g$	$v_4 \leftarrow g$	1
5	$v_1 = r$	$v_2 = g$	$v_3 = g$	$v_4 \leftarrow b$	4
6	$v_1 = r$	$v_2 = g$	$v_3 = g$	$v_4 \leftarrow r$	3
7	$v_1 = r$	$v_2 \leftarrow o$	$v_3 \leftarrow b$	$v_4 \leftarrow g$	6
Total					31

Figure 1.4: Sample execution of GT.

solution to the CSP. The algorithm stops when it has found a solution or all possible assignments have been exhausted. In the worst case it must evaluate all possible assignments (that is the complete search tree) giving a worst case time complexity of $O(m^n)$. The labeling and unlabeling functions for the GT algorithm are displayed in Figures 1.2 and 1.3.

The labeling function **gt-label** first dereferences the instantiation index ii finding which variable to instantiate (line 1). It then loops through the domain of the current variable v_i until it finds a non-deleted value (line 3-13)[5]. If it finds a non-deleted value (line 4) it assigns v_i to that value. If $ii \neq n$ no consistency checking is performed (line 7) and the function immediately returns with **consistent** set to **True**. If there is no non-deleted value in the current domain the function returns with **consistent** set to **False**. If $ii = n$ then all the constraints are tested (lines 7 13) to see if the total assignment is a solution to the CSP. If it is a solution then **consistent** will be **True** otherwise it is **False**.

[5]The semantics of a **for/while** loop is that the loop will be executed unless the while condition is false or becomes false in the loop. The while condition is tested immediately on entry into the **for** loop.

The unlabeling function **gt-unlabel** first dereferences the instantiation index ii finding the variable (line 1) that it is to move back from in the instantiation order (as it has no more values left in its domain to be instantiated). It then finds the variable that is to be uninstantiated (**h** = ii – 1) by dereferencing **hh** (lines 2 3). To move back it resets the deleted flag in **domain** for every element in d_i (lines 4 5). The function **el**(v_h) used in line 6 returns a number indicating which value in d_h v_h has been assigned to. It uses this function as an index to **domain** in order to delete the current assignment for v_h from d_h (line 6). Finally, **gt-unlabel** returns **hh** as the index to the current variable with **consistent** equal to **True** if there are more values to try in d_h otherwise **False** (lines 7 9).

In this algorithm the **domain** array is used to simply mark off alternatives in the search tree whose subtrees have been fully explored. There is no specific information stored about the results of constraint checks.

A sample execution of GT is given in Figure 1.4 using our small graph colouring problem. We assume for all the sample executions that the variable ordering is static, that is variables are chosen for instantiation in the order $\{v_1, v_2, v_3, v_4\}$. New variable instantiations are shown with a arrow pointing left (←) while past variable instantiations are shown with an equal sign (=). GT begins at step 0 with the initial domains as given in the problem definition. To save space we have collapsed multiple calls to the labeling and unlabeling functions until the consistency checks are performed at the bottom of the search tree (otherwise there would be 14 steps which would go beyond a page). The algorithm generates the first (complete) assignment $\{v_1 \leftarrow r, v_2 \leftarrow g, v_3 \leftarrow b, v_4 \leftarrow g\}$ in Step 1. Only after generating the first complete assignment does GT check to see if the assignment is consistent. The algorithm finds that the assignment is not consistent as v_2 and v_4 cannot both be coloured green and backtracks to the next total assignment. GT eventually finds a solution using 31 constraint checks.

```
function bt-label(ii)
    i ← refv_ii                                                    1
    consistent ← False                                             2
    for each v_i^l ∈ d_i while (not consistent)                    3
        if domain_i^l = 0 then                                     4
            consistent ← True                                      5
            v_i ← v_i^l                                            6
            for kk = 1 to ii-1 while consistent                    7
                k ← refv_kk                                        8
                if (v_k, v_i) ∉ c_k,i then                         9
                    consistent ← False                             10
                    domain_i^l = −∞                                11
```

Figure 1.5: Pseudo-code for the BT labeling function.

```
function bt-unlabel(ii)
    i ← refv_ii                                                    1
    hh ← ii − 1                                                    2
    h ← refv_hh                                                    3
    for each v_i^l ∈ d_i                                           4
        domain_i^l = 0                                             5
    domain_h^el(v_h) ← −∞                                          6
    if ∃k domain_i^k = 0                                           7
        then consistent ← True                                     8
        else consistent ← False                                    9
    return(hh)                                                     10
```

Figure 1.6: Pseudo-code for the BT unlabeling function.

Step	d_1	d_2	d_3	d_4	Checks
0	r	g o	b g	g b r	0
1	$v_1 \leftarrow$ r	g o	b g	g b r	0
2	$v_1 =$ r	$v_2 \leftarrow$ g $\{v_1^{\surd}\}$	b g	g b r	1
3	$v_1 =$ r	$v_2 =$ g $\{v_1\}$	$v_3 \leftarrow$ b $\{v_1^{\surd}, v_2^{\surd}\}$	g b r	2
4	$v_1 =$ r	$v_2 =$ g $\{v_1\}$	$v_3 =$ b $\{v_1, v_2\}$	$v_4 \leftarrow$ g $\{v_1^{\surd}, v_2^{\times}\}$	2
5	$v_1 =$ r	$v_2 =$ g $\{v_1\}$	$v_3 =$ b $\{v_1, v_2\}$	$v_4 \leftarrow$ b $\{v_1^{\surd}, v_2^{\surd}, v_3^{\times}\}$	3
6	$v_1 =$ r	$v_2 =$ g $\{v_1\}$	$v_3 =$ b $\{v_1, v_2\}$	$v_4 \leftarrow$ r $\{v_1^{\times}\}$	1
7	$v_1 =$ r	$v_2 =$ g $\{v_1\}$	$v_3 \leftarrow$ g $\{v_1^{\surd}, v_2^{\times}\}$	g b r	2
8	$v_1 =$ r	$v_2 \leftarrow$ o $\{v_1^{\surd}\}$	b g	g b r	1
9	$v_1 =$ r	$v_2 =$ o $\{v_1\}$	$v_3 \leftarrow$ b $\{v_1^{\surd}, v_2^{\surd}\}$	g b r	2
10	$v_1 =$ r	$v_2 =$ o $\{v_1\}$	$v_3 =$ b $\{v_1, v_2\}$	$v_4 \leftarrow$ g $\{v_1^{\surd}, v_2^{\surd}, v_3^{\surd}\}$	3
Total					17

Figure 1.7: Sample execution of BT.

1.3.2 Backtracking

It is not hard to see that GT is very inefficient. Imagine that the first value in d_1 is inconsistent with all values in v_n for a problem with 25 variables each variable having 9 values. Then in the worst case, GT will test 9^{24} complete assignments before it changes the assignment of v_1. This is a very simple example of a pathological phenomenon called "thrashing"[74] which occurs in all CSP backtracking search algorithms. Thrashing occurs when a CSP search algorithm cannot detect that there is some inconsistency between a set of variables that have been instantiated and a variable that has no consistent value given the instantiations of those variables. This failure to detect the inconsistency can cause the needless exploration of subtrees until one of the conflicting variables is assigned a new value.

Backtracking (BT) is a tree search algorithm much like GT except that it assigns values to variables one at a time and immediately tests to see if the assignment is past-consistent. That is, BT only searches through the set of consistent nodes in the search tree. BT extends a partial solution only when a past-consistent value is found for the current variable avoiding needless thrashing caused by an inconsistency between the current instantiation and one of the past variables (as in GT). If a partial solution cannot be extended the algorithm chronologically backtracks to the deepest previous variable and tries to give it a new value. This incremental instantiation of variables on average prunes out many subtrees. However, it does not improve on the worst case performance of GT, that is BT's worse case performance is still $O(m^n)$.

BT is a well known search algorithm. One early description of BT is given in [73] which uses the algorithm to thread mazes. More recent descriptions of the BT algorithm and its susceptibility to thrashing can be found in [10, 55, 66, 74, 116].

The BT algorithm is displayed in Figures 1.5 and 1.6. The labeling function **bt-label** is very similar to the labeling function for GT. The difference between the algorithms occurs in lines 7- 11. Instead of waiting until all n assignments have been made before performing consistency checking, the BT algorithm checks each instantiation with the past variables. If a value is not consistent with one of the past variables it is marked

as being deleted (line 11) and the outer loop (line 3) moves on to the next possible value. If no value is found that is past-consistent the variable **consistent** is **False** and the function returns. If a value is found that is past-consistent, v_i is set to that value, **consistent** is **True** (lines 7-11) and the function immediately returns.

The unlabeling function for BT is exactly the same as for GT. In addition to deleting a value that leads to a failed subtree as in GT. BT marks off a value if it cannot possibly be part of a solution as it is inconsistent with one of the past variables. Effectively BT is recording the same information as GT except that it learns earlier not to go down a subtree because of an inconsistency with one of the past variables.

A sample execution of the BT algorithm is given in Figure 1.7 using our small graph colouring problem. BT begins at step 0 with the initial domains as given in the problem definition. It then assigns v_1 to red without performing any past consistency checks (as there are no past variables). In step 2, BT successfully instantiates v_2 to green after checking the value with the past variable v_1. The current action of the algorithm and its current knowledge are reflected in a set following the value which shows the past variables that were checked with each superscripted with either a $\surd$ if the constraint check was successful, a $\times$ if the constraint check was unsuccessful, or nothing if a constraint check was not performed because the algorithm already knew that the constraint check would be successful. The algorithm continues until it tries to instantiate v_4 where it finds that all the colours for that variable are inconsistent with the past variables (steps 4–6). When the algorithm backtracks (step 7) to v_3 one can see that all the domain values for v_4 are returned to d_4. No attempt is made to retain the fact that the value green in d_4 is inconsistent with the instantiation of v_2 and that the colour red is inconsistent with the instantiation of v_1. These two values will be needlessly re-checked until v_2 and v_4 are assigned new values. Eventually the algorithm finds a solution after 10 steps with only 17 constraint checks which is much better than GT.

```
function bm-label(ii)
    i ← refv_ii                                                          1
    consistent ← False                                                   2
    for each v_i^j ∈ d_i while (not consistent)                          3
        if domain_i^j ≥ 0 then                                           4
            consistent ← True                                            5
            v_i ← v_i^j                                                  6
            for kk = domain_i^j + 1 to ii-1 while consistent             7
                k ← refv_kk                                              8
                if (v_k . v_i) ∉ c_k,i then                              9
                    consistent ← False                                   10
                    domain_i^j = −kk                                     11
                else                                                     12
                    domain_i^j = kk                                      13
```

Figure 1.8: Pseudo-code for the BM labeling function.

```
function bm-unlabel(ii)                                                  1
    i ← refv_ii                                                          1
    hh ← ii − 1                                                          2
    h ← refv_hh                                                          3
    for jj = ii to n                                                     4
        j ← refv_jj                                                      5
        for each v_j^j ∈ d_j                                             6
            if abs(domain_j^j) = hh then                                 7
                domain_j^j = hh − 1                                      8
    domain_h^{el(v_h)} ← −(hh − 1)                                       9
    if ∃k domain_h^k ≥ 0                                                 10
        then consistent ← True                                           11
        else consistent ← False                                          12
    return(hh)
```

Figure 1.9: Pseudo-code for the BM unlabeling function.

Step	d_1	d_2	d_3	d_4	Checks
0	r	g o	b g	g b r	0
1	$v_1 \leftarrow$ r	g o	b g	g b r	0
2	$v_1 =$ r	$v_2 \leftarrow$ g $\{v_1^{\surd}\}$	b g	g b r	1
3	$v_1 =$ r	$v_2 =$ g $\{v_1\}$	$v_3 \leftarrow$ b $\{v_1^{\surd}, v_2^{\surd}\}$	g b r	2
4	$v_1 =$ r	$v_2 =$ g $\{v_1\}$	$v_3 =$ b $\{v_1, v_2\}$	$v_4 \leftarrow$ g $\{v_1^{\surd}, v_2^{\times}\}$	2
5	$v_1 =$ r	$v_2 =$ g $\{v_1\}$	$v_3 =$ b $\{v_1, v_2\}$	$v_4 \leftarrow$ b $\{v_1^{\surd}, v_2^{\surd}, v_3^{\times}\}$	3
6	$v_1 =$ r	$v_2 =$ g $\{v_1\}$	$v_3 =$ b $\{v_1, v_2\}$	$v_4 \leftarrow$ r $\{v_1^{\times}\}$	1
7	$v_1 =$ r	$v_2 =$ g $\{v_1\}$	$v_3 \leftarrow$ g $\{v_1^{\surd}, v_2^{\times}\}$	b $\{v_1, v_2\}$	2
8	$v_1 =$ r	$v_2 \leftarrow$ o $\{v_1^{\surd}\}$	b $\{v_1\}$ g $\{v_1\}$	g $\{v_1\}$ b $\{v_1\}$	1
9	$v_1 =$ r	$v_2 =$ o $\{v_1\}$	$v_3 \leftarrow$ b $\{v_1, v_2^{\surd}\}$	g $\{v_1\}$ b $\{v_1\}$	1
10	$v_1 =$ r	$v_2 =$ o $\{v_1\}$	$v_3 =$ b $\{v_1, v_2\}$	$v_4 \leftarrow$ g $\{v_1, v_2^{\surd}, v_3^{\surd}\}$	2
Total					15

Figure 1.10: Sample execution of BM.

1.3.3 Backmarking

The BT algorithm can easily be improved upon. Gaschnig [47] noticed that BT was needlessly performing constraint checks that it had already performed. Imagine that during the course of a BT search, the variables $\{v_1, v_2, \ldots, v_i\}$ are instantiated, at which point no extension to the partial solution is found and the search backtracks to a variable v_h, at which point the search starts moving forward again. Those consistency checks (both successful and unsuccessful) that were performed for values in the domains of the variables $\{v_h, \ldots, v_i\}$ with respect to the variables $\{v_1, \ldots, v_{h-1}\}$ need not be repeated as the instantiations of $\{v_1, \ldots, v_{h-1}\}$ have not changed. The BackMarking (BM)[47] is a form of the BT algorithm which avoids these redundant constraint checks. In Gaschnig's original description of BM the variable ordering was static. However, this is an unnecessary restriction as what should be remembered is which past instantiations have changed and not which variables. As an interesting aside, we should note that Prosser[90] stated that the BM algorithm could not be used with a dynamic variable ordering due to Gaschnig's original description. Bacchus and Van Run[2, 115] took that as a challenge and showed that BM can be used with a dynamic variable ordering providing one uses an indirection array to differentiate between variables and instantiations. We also indirectly showed this result in the development of our algorithm Minimal Forward Checking[28, 30] which we will show later incorporates BM and allows for a dynamic variable selection.

The BM algorithm is shown in Figures 1.8 and 1.9. The BM algorithm displayed here uses the **domain** array rather than Gaschnig's original data structures in order to compare the difference between the algorithms described in this thesis. Descriptions of BM using Gaschnig's data structures can be found in [47, 82, 90]. The BM labeling function **bm-label** is very similar to the BT labeling function in Figure 1.5. The differences are in lines 4, 7, and 10 13. At line 4 the test is now $\geq$ as the function needs to find a domain element in d_i that is at least not deleted. The value it finds may have been successfully checked against some of the past variables and **domain**$'_i$ may therefore be positive. In line 7 the lower index of the loop is now the index of the

last instantiation plus one that successfully checked against the value v_i^j. Finally, in lines 10 13 the result of a constraint check is now recorded.

The BM unlabeling function **bm-unlabel** is more complicated than the unlabeling function for the BT algorithm. Lines 4 8 have now been added and line 9 has been changed. In lines 4 8 the function looks through the future domains (including ii) resetting the array **domain** to reflect the uninstantiation of v_h. The **abs** function in line 7 is the absolute value function. Whether a value in a future domain was deleted by, or successfully checked with, the instantiation of v_h the flag for this value is changed to $hh - 1$. This is correct behavior as either the value of $domain_j^j$ is the deepest past instantiation that was successfully checked against v_j^j which implies that v_j^j must be consistent with instantiation $hh - 1$, or the value of $domain_j^j$ is the shallowest past instantiation that deleted v_j^j which also implies that v_j^j must be consistent with instantiation $hh - 1$. Line 9 has been changed to reflect that the current value of v_h leads to a dead end and should remain deleted until the instantiation previous to **hh** is uninstantiated.

A sample execution of the BM algorithm is given in Figure 1.10 using our small graph colouring problem. Steps 0 -6 are the same as in BT. In line 7, when BM backtracks to v_3 there are two differences to note in d_4. The first is the values green and red are now deleted because the value green is inconsistent with the current value of v_2 and the value red is inconsistent with the current value of v_1. The second is that BM remembers that blue has been successfully checked against the values of v_2 and indirectly v_1. In Step 8 the value green is undeleted in d_4 as variable v_2 has been uninstantiated and BM remembers that green is consistent with the current instantiation of v_1. In steps 9 and 10 the successful consistency checks that BM remembered are not performed and the algorithm terminates with a solution performing a total of 15 constraint checks.

Although the BM algorithm now avoids the redundant checks performed by BT, it does not improve on the worst case performance. However, the algorithm is useful in practice as it does help avoid many redundant checks[61, 82].

1.3.4 Forward Checking

Imagine that during the course of a backtracking search, the variables $\{v_1, v_2, \ldots, v_i\}$ are instantiated and that the instantiation of variable v_i is inconsistent with every value in some future domain d_j. BT will completely explore the subtree from v_{i+1} to v_j before it uninstantiates v_i. BM will also explore the same subtree, avoiding redundant checks, even though it has enough information to jump out of this tree once it reaches v_j and finds that d_j is completely pruned[6]. This needless thrashing could be avoided if the algorithm could somehow look ahead and find out that the current instantiations are inconsistent with some future variable. The Forward Checking (FC) algorithm is based on this idea.

FC is a backtracking algorithm that performs a limited amount of lookahead during its search to detect inconsistencies between the current instantiation and the future variables. When the current variable is instantiated, a *forward check* is performed that deletes *all* values inconsistent with the current instantiation from the future domains[7]. If the forward check is successful (that is, every future domain has at least one value remaining), the search continues by attempting to instantiate one of the future variables. Every value in these future domains is consistent with the past variables. If the forward check is unsuccessful, some future variable is completely pruned because no value in its domain is consistent with the current instantiation. At that point the forward check is undone by replacing the values deleted because of the current instantiation and the search goes on to another value in the current domain. If there are no more values the search moves back to the previous variable instantiated, undoes

[6] If d_j is completely pruned then the algorithm can jump back to the deepest instantiation (marked in the **domain** array) that deleted a value out of the domain. Until this instantiation is undone the search has no chance of finding a solution. A non-chronological backtracking algorithm that analyzes the cause of an inconsistency and jumps back in this manner is called BackJumping (BJ)[47]. The hybrid algorithm BM with BJ (BMJ) is described in[88, 89, 90]. A more refined version of BJ is called Conflict-directed BackJumping (CBJ)[90] which is described in more detail in Section 4.1.2. A simpler version of BJ called Graph-based BackJumping (GBJ)[21] is a non-chronological backtracking algorithm which jumps back to the deepest past-connected variable.

[7] FC only needs to delete inconsistent values from future-connected domains.

```
function fc-label(ii)
    i ← refv_ii                                                    1
    consistent ← False                                             2
    for each v_i^j ∈ d_i while (not consistent)                    3
        if domain_i^j = 0 then                                     4
            consistent ← True                                      5
            v_i ← v_i^j                                            6
            for jj = ii + 1 to n while consistent                  7
                consistent ← forward-check(ii,jj)                  8
            if not consistent then                                 9
                domain_i^j = −(ii − 1)                             10
                undo-reductions(ii)                                11
```

Figure 1.11: Pseudo-code for the FC labeling function.

the forward check associated with that instantiation and tries another value in that domain.

The FC algorithm is shown in Figures 1.11 to 1.14. To understand the labeling function **fc-label** we first explain what the functions **forward-check** and **undo-reductions** do. In **forward-check** a forward check is performed between the current instantiation indexed by ii and a future domain indexed by jj. **forward-check** loops through the elements of d_j (lines 3 6) that have not been deleted (line 4) and deletes any value that is inconsistent with the current instantiation (lines 5 6). If d_j has been completely pruned the function returns **False** otherwise it returns **True**. Function **undo-reductions** is used to undelete values that were deleted from future domains because of an inconsistency with the instantiation of the variable indexed by ii. It loops thorough each future domain (lines 2 6) undeleting values that were deleted because of the instantiation of the variable indexed by ii.

The labeling function **fc-label** is quite different in lines 7 11 from the previous algorithms. Instead of checking the current instantiation against the past variables it

```
function forward-check(ii,jj)
    i ← refv_ii                                              1
    j ← refv_jj                                              2
    for each v_j^l ∈ d_j                                     3
        if domain_j^l = 0 then                               4
            if (v_i, v_j^l) ∉ c_{i,j} then                   5
                domain_j^l = −ii                             6
    if ∃k domain_j^k = 0 then                                7
        return(True)                                         8
    else                                                     9
        return(False)                                        10
```

Figure 1.12: Pseudo-code for the forward-check function.

```
function undo-reductions(ii)
    i ← refv_ii                                              1
    for jj = ii + 1 to n                                     2
        j ← refv_jj                                          3
        for each v_j^l ∈ d_j                                 4
            if domain_j^l = −ii then                         5
                domain_j^l = 0                               6
```

Figure 1.13: Pseudo-code for the undo-reductions function.

```
function fc-unlabel(ii)
    i ← refv_ii                                                    1
    hh ← ii − 1                                                    2
    h ← refv_hh                                                    3
    undo-reductions(hh)                                            4
    domain_h^{el(v_h)} ← −(hh − 1)                                 5
    if ∃k domain_h^k = 0                                           6
        then consistent ← True                                     7
        else consistent ← False                                    8
    return(hh)                                                     9
```

Figure 1.14: Pseudo-code for the FC unlabeling function.

Step	d_1	d_2	d_3	d_4	Checks
0	r .	g o	b g	g b r	0
1	$v_1 \leftarrow$ r	g $\{v_1^{\surd}\}$ o $\{v_1^{\surd}\}$	b $\{v_1^{\surd}\}$ g $\{v_1^{\surd}\}$	g $\{v_1^{\surd}\}$ b $\{v_1^{\surd}\}$ r $\{v_1^{\times}\}$	7
2	$v_1 =$ r	$v_2 \leftarrow$ g	b $\{v_1, v_2^{\surd}\}$ g $\{v_1, v_2^{\times}\}$	g $\{v_1, v_2^{\times}\}$ b $\{v_1, v_2^{\surd}\}$	4
3	$v_1 =$ r	$v_2 =$ g	$v_3 \leftarrow$ b	b $\{v_1, v_2, v_3^{\times}\}$	1
4	$v_1 =$ r	$v_2 \leftarrow$ o	b $\{v_1, v_2^{\surd}\}$ g $\{v_1, v_2^{\surd}\}$	g $\{v_1, v_2^{\surd}\}$ b $\{v_1, v_2^{\surd}\}$	4
5	$v_1 =$ r	$v_2 =$ o	$v_3 \leftarrow$ b	g $\{v_1, v_2, v_3^{\surd}\}$ b $\{v_1, v_2, v_3^{\times}\}$	2
6	$v_1 =$ r	$v_2 =$ o	$v_3 =$ b	$v_4 \leftarrow$ g	0
Total					18

Figure 1.15: Sample execution of FC.

checks the current instantiation against every future domain using **forward-check**. If at least one consistent value is found in each future domain the search can go forwards and the function returns with **consistent** equal to **True**. If the current instantiation causes one of the future domains to be completely pruned then it can immediately be discarded (line 10) and the reductions performed undone (line 11). The function then goes on to the next non-deleted domain value and attempts another instantiation. If no instantiation is found the function returns with **consistent** equal to **False**. There is an important property enforced by **fc-label**. Only values that are consistent with the past instantiations will be in the future domains. This has two important implications (which will become more obvious in the next chapter). The first is that a forward check never needs to repeat constraint checks with past variables as it already knows the result. The second is that **fc-label** never has to check a value in the current domain against the past variables as it already knows that they are past-consistent.

The unlabeling function **fc-unlabel** is quite simple. It undoes the previous instantiation indexed by **hh** using **undo-reductions** (line 4) and deletes the assignment from d_{hh} (line 5). It returns the index **hh** as the new current variable and **consistent** equal **True** if there are any remaining values in d_{hh}, otherwise it returns with **consistent** equal to **False**.

At this point is it convenient to give two seemingly obvious lemmas about FC. We will use these two lemmas to discuss FC's relationship to BM in the next chapter.

Lemma 1.1 *Given that variables* $\{v_1, \ldots, v_i\}$ *have been instantiated by the FC algorithm, every non-deleted future domain value* v_k^l $(i + 1 \leq k \leq n)$ *is consistent with* each *past variable* v_j *and the constraint checks on* v_k^l *have been performed in the order of instantiation.*

Proof of Lemma 1.1 Every future domain value must be past-consistent. For every instantiation of v_j, $(1 \leq j \leq i)$, FC (Figure 1.11) deletes all inconsistent future values (lines 7-8) using the **forward-check** function (Figure 1.12). Therefore every non-deleted future domain value v_k^l $(i + 1 \leq k \leq n)$ is consistent with each past variable v_j. As forward checks are performed at each level of the search tree from the current

instantiation against the future domains (that is, deeper in the tree) the constraint checks against the future domain values are performed in the order of instantiation. □

Although this lemma may seem obvious, in the usual description of the FC algorithm the fact that each future value is past-consistent with *each* past instantiation is implicit.

Lemma 1.2 *Given that variables* $\{v_1, \ldots, v_i\}$ *have been instantiated by the FC algorithm, each future domain value* v_k^l $(i+1 \leq k \leq n)$ *that is deleted as a result of the instantiation of* v_i *is past-consistent with each instantiation in* $\{v_1, \ldots, v_{i-1}\}$ *and it is undeleted when the FC algorithm backtracks and uninstantiates* v_i.

Proof of Lemma 1.2 The first part of the lemma is a direct consequence of Lemma 1.1. For the second part of the lemma consider Figure 1.12 lines 4-6. If a future value v_k^l is inconsistent with the value of v_i it is marked as deleted in the domain_k^l array as a result of the instantiation of v_i. If the search goes deeper, this value is never considered again as it is marked as deleted. When v_i is backtracked to, function **fc-unlabel** (Figure 1.14, line 4) calls function **undo-reductions** (Figure 1.13) which undeletes all values deleted because of the instantiation of v_i. □

We now give a sample execution of the FC algorithm in Figure 1.15 using our small graph colouring problem. In step 1, v_1 is assigned the value red and FC goes through the future domains looking for inconsistent values. The value red in d_4 is found to be inconsistent and is deleted. The search now moves forward as there are consistent values in every future domain (step 2). Variable v_2 is assigned the value green and a forward check is performed for each future variable. The values green in both d_3 and d_4 are inconsistent and are deleted. Variable v_3 is assigned the value blue and a forward check is performed for variable v_4. 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 backtracks to v_2. The value blue is undeleted in d_4, and the value green is undeleted in d_3 and d_4. Variable v_2 is then assigned the value orange (step 4). FC checks the future domains and finds no inconsistent values. In step 5, v_3 is

assigned the value blue and FC deletes the value blue from d_4. Finally, step 6 shows the solution found with a total of 18 constraint checks.

Up to this point in the thesis we have only mentioned that CSP search algorithms may benefit from a dynamic variable ordering heuristic. FC is the first algorithm mentioned whose performance can often be greatly improved through the addition of such a heuristic called the Fail First (FF)[61] (the addition of FF to FC is called FC-FF). The FF heuristic as defined in [61, p. 266] is: "pick the variable that has the fewest remaining values in its domain as the next variable to instantiate". A stronger version of the FF heuristic which is also (accidently) defined in [61, p. 302] is: "pick the variable that has the fewest remaining *consistent* values in its domain"[8]. This stronger heuristic causes no difficulty when used with FC as all values inconsistent with the past instantiations have already been deleted before the next variable to instantiate is chosen. However, the distinction is important as not all "forward checking" algorithms know exactly how many past-consistent values there are in each future domain as we shall see in the next chapter.

Intuitively, the FF heuristic improves performance by dynamically rearranging the search tree so that lower branching factor nodes (smaller domains) are shallower in the search tree and higher branching factor nodes (larger domains) are deeper in the search tree. Since all variations of the BT algorithm prune subtrees when an inconsistent instantiation is found, if the larger domains are deeper in the search tree, then more of the search tree would be pruned by an inconsistent instantiation than if the smaller domains are deeper in the search tree. The FF heuristic is particularly appropriate for FC as FC deletes values from the future domains that are inconsistent with the current instantiation thereby changing the bushiness of the part of the search tree that it may visit. The FF heuristic dynamically minimizes the bushiness of the tree by bringing the variable with the fewest past-consistent values to the "top". Haralick and

[8]The FF heuristic is a heuristic for picking the next variable to instantiate which can be used by *any* CSP backtracking search algorithm. Haralick and Elliot were imprecise in their definition of FF as they relied on the fact that FC removes all values inconsistent with the past variables from the future domains. Only in that sense are their two definitions equivalent.

Elliot[61] use a probabilistic argument to show that the FF heuristic minimizes the "expected branch depth"[9] thereby reducing the expected number of nodes that need to be searched.

The dynamic optimization of the search tree by FC-FF also has the side effect of bringing variables that are "related" by constraints closer together in the instantiation order. There are two senses of "relatedness" here that are quite similar but distinct in effect. The first "relatedness" is governed by the tightness of the constraints, that is how few consistent assignments are allowed. If a constraint is tight between the current variable and some future variable then more than likely that future variable will be picked fairly soon in the instantiation order as it will be pruned to a small domain size by the forward check. The second "relatedness" is governed by how many constraints a variable participates in. If a future variable is highly connected to the past variables then it will be pruned more than other future variables and will therefore be picked fairly closely to the past variables that it is related to by a constraint. FC-FF will bring variables that are related in these two senses closer together in the instantiation order, more for one sense than another for different types of CSPs. This "relatedness of variables" side effect has implications for the use of non-chronological backtracking with FC-FF which we discuss in Chapter 4.

Although the FF heuristic is quite effective for some types of CSPs (for example, randomly generated problems), it is not appropriate for all types of problems. Some problems benefit from a preliminary static ordering of the variables before search (for example the minimal width ordering[25, 40]) while others benefit from dynamic variable ordering heuristics that depend on factors other than domain size[87, 101]. Therefore it is important to find improvements for both the FC algorithm and the FC-FF algorithm.

Whether or not the FF heuristic is used, the FC algorithm or a hybrid FC algorithm with non-chronological backtracking is considered the most effective general CSP backtracking search algorithm[2, 61, 77, 82, 90, 111]. That is, if the FF heuristic is not suitable for a particular problem, then a FC algorithm (or hybrid) with

[9]Read "expected branch depth" as "expected path length".

```
function bc-label(ii)
    i ← refv_ii                                                    1
    consistent ← False                                             2
    for each v_i^l ∈ d_i while (not consistent)                    3
        if domain_i^l = 0 then                                     4
            consistent ← True                                      5
            v_i ← v_i^l                                            6
            for kk = 1 to ii-1 while consistent                    7
                k ← refv_kk                                        8
                if (v_k, v_i) ∉ c_k,i then                         9
                    consistent ← False                             10
                    domain_i^l = −kk                               11
```

Figure 1.16: Pseudo-code for the BC labeling function.

or without some static pre-ordering of the variables is the most effective backtracking search algorithm for solving CSPs. If the FF heuristic is suitable for a particular problem then FC-FF or a FC-FF hybrid is the most effective algorithm for solving the CSP.

The FC algorithm is in fact quite old. It is first mentioned in [55, pg. 521] where it is called *preclusion*. However (believe it or not) the authors, Golomb and Baumert, rejected the idea as being too expensive to use on "today's digital computers". Not only did Golomb and Baumert accurately describe FC they also described the FF heuristic. Again they rejected the idea as being too expensive. The FC algorithm made another appearance in [77] before finally being given the name by which it is currently known [61].

1.3.5 BackChecking

Finally, the last CSP search algorithm that we describe in this chapter is called BackChecking (BC)[61]. BC is an algorithm like FC except that it looks backwards instead of forwards. BC is BT with the the ability to delete values from the current

```
function bc-unlabel(ii)
  i ← refv_ii                                                          1
  hh ← ii − 1                                                          2
  h ← refv_hh                                                          3
  for jj = ii to n                                                     4
    j ← refv_jj                                                        5
    for each v_j^l ∈ d_j                                               6
      if domain_j^l = −hh then                                         7
        domain_j^l = 0                                                 8
  domain_h^{el(v_h)} ← −(hh − 1)                                       9
  if ∃k domain_i^k ≥ 0                                                 10
    then consistent ← True                                             11
    else consistent ← False                                            12
  return(hh)                                                           13
```

Figure 1.17: Pseudo-code for the BC unlabeling function.

domain if they are past inconsistent and not return them until the past variable that the value is inconsistent with is uninstantiated. The difference between BC and BM is that BM also remembers past successful checks.

The BC algorithm is shown in Figures 1.16 and 1.17. The labeling function **bc-label** is the same as **bt-label** except for line 11 where the instantiation that deleted a value is recorded. The unlabeling function 1.17 is quite similar to **bm-unlabel** except in lines 7 8. In line 7, BC only checks for values that have been deleted by **hh** (unsuccessful checks) and resets them to indicate that they are no longer deleted (line 8). The difference between BC and BM is that BM both successful and unsuccessful constraint checks while BC only records unsuccessful constraint checks.

A sample execution of the BC algorithm is given in Figure 1.18 using our small graph colouring problem. The execution of the algorithm is quite similar to BT (Figure 1.7) until step 7 at which point BC remembers that the colours green and red in d_4 are inconsistent with v_2 and v_1 respectively. The rest of the execution is the

same as in BT.

1.4 Consistency Enforcing Algorithms

Thrashing occurs when a CSP search algorithm cannot detect that there is some inconsistency between a set of variables that have been instantiated and a variable that has no consistent value given the instantiations of those variables. It seems reasonable that one might want to eliminate some sources of thrashing as a preprocessing step before a CSP is searched for a solution. That is, it may be beneficial to the search if a preprocessing step is performed to remove some of the inconsistencies, reducing the given CSP into one which has a smaller search space but is equivalent to the original in the sense that the reduced CSP has the same set of solutions. For example, one might want to ensure that for every value in every domain there exists a consistent value in every other domain. If no such consistent value exists the value under consideration may be deleted. Or, for every consistent pair of values for v_i and v_j ensure that there exists a third consistent value in v_k ($1 \leq k \leq n, k \neq i, k \neq j$). If no such consistent value is found then disallow the consistent pair in $c_{i,j}$. The "level of consistency" being enforced in the first example is called *Arc Consistency* (AC) also known as 2-consistency. The level of consistency being enforced in the second example is called *PATH consistency* (PATH) also known as 3-consistency[74, 81]. In general, the level of consistency of a general CSP can be extended to k-consistency[39] where the idea is that given a consistent labeling of $k-1$ variables ensure that there exists a k'th consistent value in the domains of the rest of the variables. If one enforces strong n-consistency on a CSP (that is the CSP is k-consistent for all $k \leq n$) then the set of all solutions is formed[39]. Descriptions of the k-consistency algorithms can be found in [16, 27, 39, 41, 40]. Although this approach does offer another avenue for finding all solutions to a CSP it does not improve on the worst case time complexity of $O(m^n)$. Consistency enforcing algorithms are usually not used beyond path consistency as they are not considered worthwhile.

There are many algorithms available to perform AC. The simplest (brute force

Step	d_1	d_2	d_3	d_4	Checks
0	r	g o	b g	g b r	0
1	$v_1 \leftarrow$ r	g o	b g	g b r	0
2	$v_1 =$ r	$v_2 \leftarrow$ g $\{v_1^{\surd}\}$	b g	g b r	1
3	$v_1 =$ r	$v_2 =$ g $\{v_1\}$	$v_3 \leftarrow$ b $\{v_1^{\surd}, v_2^{\surd}\}$	g b r	2
4	$v_1 =$ r	$v_2 =$ g $\{v_1\}$	$v_3 =$ b $\{v_1, v_2\}$	$v_4 \leftarrow$ g $\{v_1^{\surd}, v_2^{\times}\}$	2
5	$v_1 =$ r	$v_2 =$ g $\{v_1\}$	$v_3 =$ b $\{v_1, v_2\}$	$v_4 \leftarrow$ b $\{v_1^{\surd}, v_2^{\surd}, v_3^{\times}\}$	3
6	$v_1 =$ r	$v_2 =$ g $\{v_1\}$	$v_3 =$ b $\{v_1, v_2\}$	$v_4 \leftarrow$ r $\{v_1^{\times}\}$	1
7	$v_1 =$ r	$v_2 =$ g $\{v_1\}$	$v_3 \leftarrow$ g $\{v_1^{\surd}, v_2^{\times}\}$	b	2
8	$v_1 =$ r	$v_2 \leftarrow$ o $\{v_1^{\surd}\}$	b g	g b	1
9	$v_1 =$ r	$v_2 =$ o $\{v_1\}$	$v_3 \leftarrow$ b $\{v_1^{\surd}, v_2^{\surd}\}$	g b	2
10	$v_1 =$ r	$v_2 =$ o $\{v_1\}$	$v_3 =$ b $\{v_1, v_2\}$	$v_4 \leftarrow$ g $\{v_1^{\surd}, v_2^{\surd}, v_3^{\surd}\}$	3
Total					17

Figure 1.18: Sample execution of BC.

method) is called AC-1 and is described in [74]. Basically, the algorithm loops through every value in every domain checking every other domain for a consistent value matching the value under consideration. If the algorithm successfully finishes one complete pass over every value in every domain without deleting one value then the CSP is arc consistent. The algorithm has been further improved upon in [74] (AC-2,AC-3),[80] (AC-4), in [7] (AC-6), in [9] (AC-6++), and in [8] (AC-7). A recently proposed version of AC, called LAC_7[48], is strongly related to the MFC algorithm (*cf.* Section 2.8). The best arc consistency algorithm has a worst case time complexity of $O(\mathbf{m^2e})$ where $\mathbf{e}$ is the number of edges (non-trivial constraints) in the graph.

Algorithms that ensure path consistency are described in [74] (PC-1,PC-2), [81] (PC-1, called Algorithm C), [80] (PC-3) which had minor mistakes which led to [59] (PC-4). The best path consistency algorithm has a worst case time complexity of $O(\mathbf{m^3n^3})$. A comprehensive overview of many of these algorithms and their time and space complexities can be found in [108].

Usually the amount of preprocessing performed is restricted to arc consistency (if at all), due to the cost of these algorithms. It is still an open question (which is not addressed in this thesis) that causes much debate on when it is useful to perform even arc consistency before performing a search. We introduce these consistency algorithms here, as CSP search algorithms can be seen as tree search with the addition of (partial) consistency processing (usually a partial amount of arc consistency processing)[82]. For example, BT, BM and BC all enforce a partial amount of arc consistency between the current variable and the past variables. FC enforces a partial amount of arc consistency between the current variable and the future variables.

1.4.1 Hybrid Algorithms

The term *hybrid algorithm* is used in two different senses in the literature. In the first sense, a hybrid algorithm is a backtracking search algorithm with the addition of a partial amount of consistency processing performed during the search[82]. In the second sense, a hybrid algorithm is a algorithm which melds the forward labeling move of one known CSP search algorithm with the backward labeling move of an-

other. Nadel has identified a number of algorithms as being hybrid algorithms of the first type including BT, BM, FC, BJ, Full and Partial Lookahead (FL and PL respectively) and Really Full Lookahead (RFL)[61]. FL is a CSP backtracking search algorithm which performs not only FC on the future domains but also makes one pass over all the values in every future domain ensuring that there is a consistent value matching it in every other future domain. PL is the same as FL except that the pass over the values in the future domains only ensures that there is a matching value in the domains that are in the future of the value being checked. Neither FL nor PL enforce full AC. RFL is a backtracking search algorithm which performs full AC on the future domains at every instantiation. A version of this algorithm was actually developed in Gaschnig's thesis[47] (called DEEB for "Domain Element Elimination with Backtracking"). This algorithm has not been used because of its perceived cost but has recently been recalled to life by Sabin and Freuder[97](who have now renamed it MAC for Maintaining Arc Consistency).

Prosser[90] has introduced a number of hybrid algorithms in the second sense that combine the attributes of BM and FC with BJ and CBJ giving BM with BJ (BMJ), BM-CBJ, FC-BJ, and FC-CBJ. As we have developed and use many hybrid algorithms (in both senses) in this thesis we need a standard notation to identify an algorithm. We will use the notation "FOR-BAC-EXT-DVO" where FOR is the forward labeling function, BAC is the backward labeling function (chronological backtracking if omitted), EXT is any extra processing performed after the forward labeling is successful and before the selection of the next variable (omitted if no extra processing is used), and DVO is the type of variable ordering heuristic used (omitted if the ordering is static).

In [82], Nadel proposed a notation for hybrid algorithms which makes explicit the relative amount of partial arc consistency performed by the algorithm. For example, he suggested that FC could be referred to as TS+AC(1/4) which means that FC performs Tree Search with approximately 1/4 arc consistency processing. Nadel's proposed notation has never been used as the old names are so well known in the literature. The important contribution that Nadel made is that he identified a relative

ordering by the approximate amount of arc consistency performed at a node for the then known algorithms. BT performed the least amount of arc consistency processing followed by BJ, BM, FC, PL, FL and RFL. Although his experimental evidence is weak (which we will describe in the next section) he ordered the algorithms from best to worst in terms of average number of constraint checks performed as FC, BM, PL, FL, RFL, BJ, and BT. He noted that FC is the best algorithm according to his experiments and that perhaps a better algorithm could be found somewhere between BT and FC or between FC and PL. This comment is somewhat responsible for our focus on the FC algorithm and the development of Minimal Forward Checking. We will discuss this point further in the next chapter.

1.5 Previous Theoretical and Empirical Comparisons

In general, the worst case time complexity of CSP search algorithms is $O(n^n)$. This worst case time complexity is not representative of the typical performance of CSP search algorithms in practice. Although it is highly desirable to have average case complexity results for these algorithms, such results are extremely difficult to derive[25, 76, 82]. An analysis of the average time complexity for CSP search algorithms needs to make overly simplifying assumptions about the distribution of problems in order to make the analysis somewhat tractable. However, it turns out that the average case time analysis is highly sensitive to these simplifying assumptions[25, 68]. Knuth[58, 66] has developed a Monte Carlo method of estimating the search tree size for a backtracking algorithm but only for a particular problem under consideration.

The difficulty in deriving analytical average time complexity results leaves two avenues of approach in comparing CSP search algorithms. The first approach is to show dominance relationships between algorithms, that is, show that one algorithm is always the same or better than another algorithm by the number of constraint checks performed or by the number of search tree nodes visited on any given problem[30, 34, 33, 67, 68]. The second approach is to empirically compare algorithms by their performance on randomly generated problem instances[1, 2, 5, 7, 8, 9, 24, 25, 28, 29,

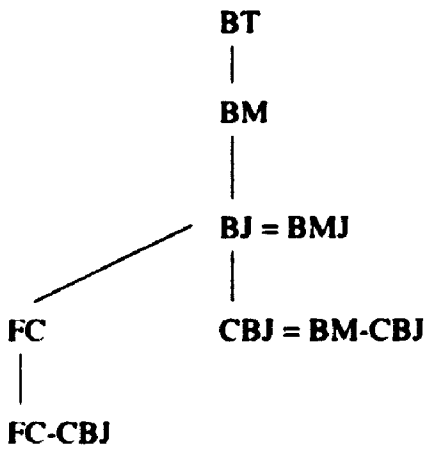

Figure 1.19: A hierarchy of CSP search algorithms with respect to the number of nodes visited.

30, 34, 33, 42, 43, 44, 47, 49, 61, 71, 93, 97, 109, 115] or specific problem instances[1, 2, 24, 25, 47, 77, 82, 90, 113, 115].

1.5.1 Previous Theoretical Comparisons

A significant paper that follows the first avenue of approach of showing dominance relations is that of Kondrak and van Beek[67, 68]. They prove a number of interesting theorems about the algorithms we have seen so far. The basis of their proofs lie on necessary and sufficient conditions for a node to be visited (called the *characterizing condition*) by BT, BJ, BM, CBJ, and FC given a static ordering of the variables. They prove the following theorem: [68]:

Theorem 1.1 *A backtracking search algorithm is correct if it is sound, that is it only finds solutions, and it is complete, that is it finds all solutions, and it terminates. BT, BM, BJ, CBJ, and FC are correct.*

They also present two hierarchies, displayed in Figures 1.19 and 1.20 for the number of nodes visited and the number of constraint checks performed by a particular algorithm.

In Figure 1.19, algorithms that visit the same nodes have an equal sign between them. If an algorithm visits a subset (possibly equal) of the nodes of another then it

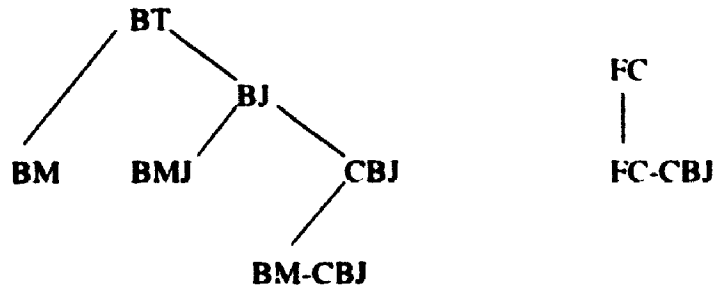

Figure 1.20: A hierarchy of CSP search algorithms with respect to the number of constraint checks performed.

is displayed lower than that algorithm and there is a connection between them. There is a minor difference between Figure 1.19 and Kondrak and van Beek's hierarchy. We have BM visiting a subset of the nodes visited by BT whereas they have BM and BT visiting the same nodes. The cause of the difference is the point at which a variable is instantiated. We check the domain'_i array before attempting to instantiate v_i to v'_i to see if it has already been deleted. They check for deletion after instantiating the variable. In Figure 1.20 algorithms which perform the same or less constraint checks than another algorithm are displayed lower than that algorithm and there is a connection between them. We use the above results later in the thesis for the algorithms that we develop.

1.5.2 Previous Empirical Comparisons

There are many papers that follow the second approach of performing empirical tests to compare the performance of different CSP search algorithms. As there are so many, we limit our discussion of previous comparisons to those that are most relevant to our thesis. Basically, there have been two types of empirical comparisons performed. The first type of comparison uses a specific problem such as the well known n-queens problem. The n-queens problem is: given a $n \times n$ chessboard, find a placement of n queens such that no queen can take another. An example solution to the 6-queens problem is shown in Figure 1.21. The first thing to notice about this problem is

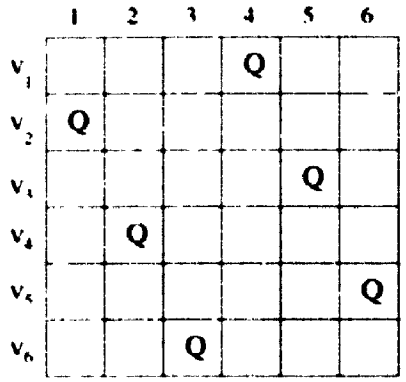

Figure 1.21: A solution to the 6-queens problem.

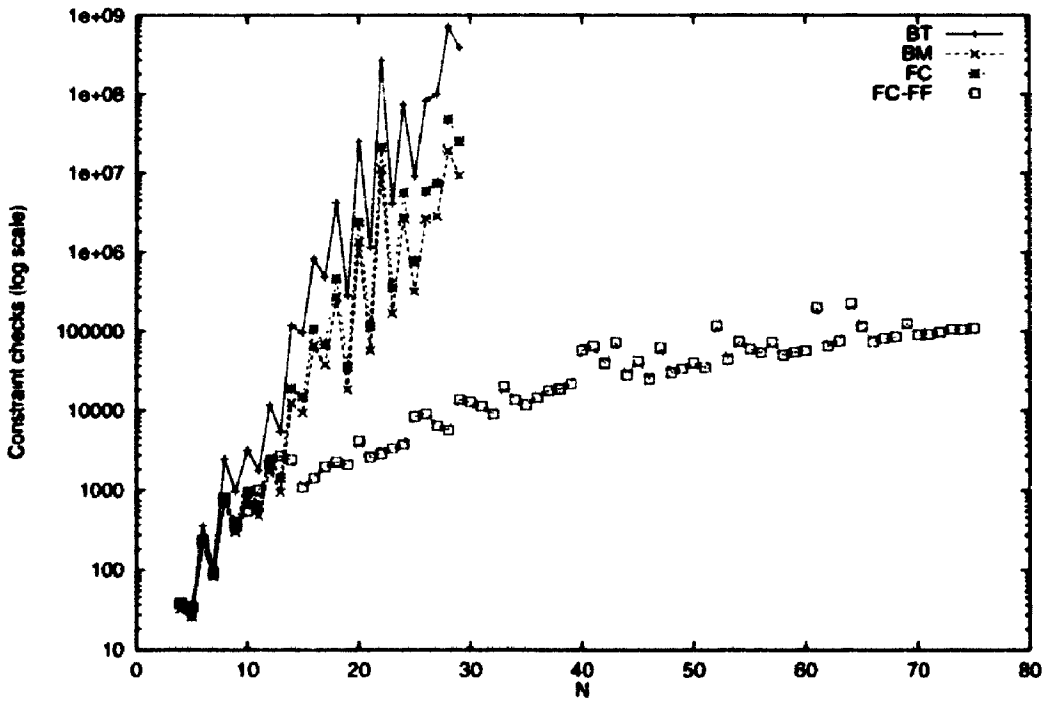

Figure 1.22: Comparison of BT, BM, FC, and FC-FF by constraint checks on the n-queens Problem.

that every queen must be in a distinct row (alternatively column). This makes the representation of the problem easy. Let a variable v_i represent the i'th queen which is in the i'th row. The domain of v_i is $\{1, \ldots, n\}$ which are the possible column positions. The constraint between every pair of variables v_i and v_j is that the queens are not in the same column and that they are not on the same diagonal. To keep up with a long standing tradition we perform a small empirical comparison of BT, BM, FC and FC-FF using the n-queens problem.

To perform the experiment we run the BT, BM and FC algorithms on the n-queens problem for $n \in \{4..29\}$ and for the FC-FF algorithm we use $n \in \{4..75\}$. For each algorithm we count the number of constraint checks performed in finding the first solution. The results of this experiment are displayed in Figure 1.22. Clearly FC-FF is far superior to the other three algorithms. Of the three algorithms that do not use FF the ordering from best to worst is roughly BM $\geq$ FC $>$ BT. These results are similar to past empirical comparisons of these algorithms[1, 2, 47, 61, 82, 113, 115] using the n-queens problem. Some of these past comparisons have not looked only at the above mentioned algorithms but also BC, BJ, FC, PL, FL and RFL with and without BJ or CBJ. The general conclusion reached by all of these past comparisons is that algorithms that look ahead such as FC, PL, FL and RFL explore fewer nodes in the search space (in the order RFL, FL, PL and then FC) than past looking algorithms such as BT, BM, BJ, and BC. Of the lookahead algorithms FC (or FC with CBJ or BJ) explores the most nodes but performs the least number of constraint checks. Of all the algorithms, a FC hybrid or BM performs the best without FF and of all the algorithms using FF, FC-FF (or FC-CBJ-FF) performs the best[10].

The results of these empirical comparisons should not be taken too seriously. The n-queens problem is not at all representative of a general CSP problem. The domain size is uniform, the constraint graph is complete (completely connected by non-trivial constraints), and all the constraints are the same. Even worse the constraints between variables get easier to satisfy as n grows as there are fewer pairs ruled out between

[10]This is not exactly true. The comparisons in [1, 2] also include the MFC algorithm but we defer commenting on their results until the next chapter.

variables[82, 108]. This fact allows local search algorithms (that is non-systematic hill climbing search algorithms) to find a solution to the three million queen problem in less than a minute[57, 105, 106][11]. Actually, finding one solution to the n-queens problem is not really a problem as there are deterministic algorithms for constructing one solution[6]. However, if one wants all solutions then the problem is still quite challenging as a benchmark problem.

Other comparisons using specific problems can be found in [90] which uses a logic puzzle called the Zebra problem (problem definition given in [21]), in [82] which uses the confused n-queens problem, and in [77] which uses the subgraph isomorphism problem. FC or a FC hybrid algorithm is found to be the best algorithm of those tested given a static ordering (they do not investigate FC-FF).

The second type of empirical comparison uses randomly generated problems instead of specific problems in order to classify the performance of CSP algorithms on a range of different problems. CSP's are modeled using 4 parameters $\langle n, m, p_1, p_2 \rangle$ where n is the number of variables, m is the domain size for each variable, p_1 is the probability that a non-trivial constraint exists between two variables (called the *constraint density*), and p_2 is the probability, conditional on the existence of a non-trivial constraint, that a pair of values between the two variables is inconsistent (called the *constraint tightness*). These comparisons may use a slightly different parameterization for these 4 parameters but there is no overall difference from the above parameterization [92]. These comparisons can be broken into two categories, those before it was discovered how to create "hard" randomly generated problems[28, 29, 47, 61, 115], and those afterward[1, 2, 30, 34, 33, 43, 49, 71, 93, 97, 109]. As hard problems are explained in Chapter 3 we defer an explanation of what a "hard" problem is and the descriptions of empirical comparisons using hard problems (*cf.* Section 3.2.3).

Many comparisons in the first category use randomly generated problems that were relatively easy to solve, are limited in the number of instances generated, and use

[11] Smith [103] points out that for problems such as n-queens, with completely connected constraint graphs, the constraints must become easier to satisfy as the number of variables increases in order for the problem to remain soluble.

relatively small problems. One of the earliest comparisons using randomly generated problems is in Gaschnig[47]. Gaschnig creates random problems that are very similar to the n-queens problem for $n \in \{4, 5, \ldots, 14\}$ with the number of instances generated ranging from 50 for $n = 4$ to 250 for $n = 14$. A total of 1,100 instances were generated. Gaschnig compared the performance of BT, BM, BJ and DEEB. He found that BM performed the best. Haralick and Elliot[61] also performed comparisons with randomly generated problems that are similar to the n-queens problem for $n \in \{4, 5, \ldots, 10\}$ with 5 random instances being generated for each value of n. Of the algorithms they tested, including BT, BM, BC, FC, FC-FF, PL and FL they found that FC is the best of the algorithms not using FF and FC-FF is the best overall. We also compared our MFC algorithm and FC algorithm using randomly generated problems in the first category[29, 28]. We performed two experiments. In the first experiment we vary p_1 and p_2 in $\{0.1, 0.3, 0.5, 0.7, 0.9, 1.0\}$, m in $\{5, 10, 15, 30\}$, and n in $\{5, 10\}$, generating 15 instances for each setting of the 4 parameters giving a total of 4,320 random instances. In the second experiment p_1 and p_2 are allowed to vary in $\{0.1, 0.2, \ldots, 1.0\}$, m in $\{5, 10, 15, 30\}$, and n in $\{5, 10\}$, generating 15 instances for each setting of the 4 parameters giving a total of 12,000 random instances. A more complete set of values for p_1 and p_2 were used in the second experiment as more computational resources were available for this experiment. The empirical results from these preliminary papers implied that MFC was a much better algorithm than FC. However, we found that many of the problems generated were very easy to solve except for a few exceptional cases which were extremely hard (by many orders of magnitude) to solve. We believe our experience with random problems up to this point was very similar to many other authors.

In general, unless some heuristic was used (for examples see [25, 115]) to generate harder problems, most randomly generated binary CSPs created with the above 4 parameters were not very satisfactory to compare search algorithms as most problem instances were easy to prove soluble or insoluble.

1.6 Summary

In this chapter we have given a formal definition of a binary CSP and described in detail a number of algorithms which are directly related to our work on MFC and lazy constraint satisfaction algorithms. We have taken the approach of making the algorithms as simple as possible to understand at the cost of making the algorithms slightly less efficient (in terms of overhead, not constraint checks) than they could be. Many of the algorithms described in this chapter have been described in other papers in detail which tends to obscure the basic ingredients of the algorithms. We have also described the past theoretical and empirical results that directly affect our thesis. In the rest of the thesis we build on this foundation. We describe a promising new search algorithm called Minimal Forward Checking which uses the idea that a CSP search algorithm shouldn't perform any extra arc consistency processing that it doesn't need to in order to move deeper in a search. We show that this idea is actually quite robust, applying not only to FC but also to RFL leading to the concept of a lazy CSP search algorithm. We also describe and investigate hard randomly generated problems which are used as the basis of empirical comparisons in this thesis. We show that the current model of randomly generating hard problems can actually be generalized so that it applies to individual problems instead of a class of problems. We show empirically that this generalization produces random problems that are on average of similar hardness or harder than random problems generated with the old method. Finally, we perform a large and extensive comparison using both the hard random problems described by others and by ourself.

Chapter 2

Minimal Forward Checking

> "We are to admit no more causes of natural things than such as are both true and sufficient to explain their appearances. To this purpose the philosophers say that Nature does nothing in vain, and more is in vain when less will serve; for Nature is pleased with simplicity, and affects not the pomp of superfluous causes." Isaac Newton

2.1 Introduction

In the last chapter we described in detail a number of constraint satisfaction backtracking search algorithms, namely Generate and Test (GT), Backtracking (BT), Backmarking (BM), Forward Checking (FC), and BackChecking (BC). Of those algorithms, FC is thought to be one of the best algorithms for solving CSPs. FC performs a limited amount of lookahead during its search to detect inconsistencies between the current instantiation and the future variables thereby avoiding unnecessary search. When FC instantiates a variable it filters *all* values inconsistent with the instantiation from the future domains. If a future domain is completely pruned then the current instantiation is an inconsistent choice and the filtered values are returned (undeleted) to their respective domains. FC's efficiency is attributed to its ability of detecting inconsistencies higher in the search tree with less arc consistency checking per node than other more complicated arc consistency algorithms such as Partial Lookahead (PL), Full Lookahead (FL) and Really Full Lookahead (RFL, also known as MAC for Maintaining Arc Consistency)[61, 82].

In this chapter we describe a new constraint satisfaction backtracking algorithm,

called Minimal Forward Checking (MFC) which is a "lazy" version of the FC algorithm. Originally, the discovery of the MFC algorithm[28, 29, 30] was motivated by the observation that FC allows the instantiation of a variable only when there is at least *one* value in every future domain that is past-consistent. FC not only finds one consistent value in every future domain but continues to prune out all values in the future domains inconsistent with the current instantiation. This extra pruning is unnecessary when a forward check fails, that is some future domain is completely pruned, a phenomenon which obviously happens quite often. Many useless constraint checks are performed while searching for a solution especially for problems with larger domain sizes and for problems with a large number of variables. Useless constraint checks are also performed for problems that have loose constraints, that is the constraints are easy to satisfy. The MFC algorithm finds and maintains one consistent value in every future domain, "suspending" forward checks until they are required by the search. This lazy approach of performing FC avoids many constraint checks which need not be performed because a future domain is completely pruned. Effectively, MFC mimics the search of FC avoiding constraint checks until they are needed.

We begin this chapter, in Section 2.2, with another look at the FC algorithm. The relationship between FC and BM is made explicit and we discuss how the MFC algorithm can be derived from this relationship. The MFC algorithm is then described in Section 2.3 followed by an example execution of the algorithm in Section 2.4. Section 2.5 gives a number of theoretical results. We then revisit the n-queens problem giving a small empirical comparison of FC, MFC, FC-FF and MFC-FF. Section 2.7 discusses work related to the MFC algorithm. Finally, we describe in Section 2.8 what makes a CSP search algorithm lazy and discuss whether other lazy CSP search algorithms exist.

2.2 The Relationship of FC to BM

The usual description of FC is that it is a forward looking search algorithm that deletes all values inconsistent with the current instantiation before moving deeper into the

```
function fc-label(ii)
  i ← refv_ii                                                    1
  consistent ← False                                             2
  for each v_i^! ∈ d_i while (not consistent)                    3
    if domain_i^! ≥ 0 then                                       !4
      consistent ← True                                          5
      v_i ← v_i^!                                                6
      for jj = ii + 1 to n while consistent                      7
        consistent ← forward-check(ii,jj)                        8
      if not consistent then                                     9
        domain_i^! = −(ii − 1)                                   10
        undo-reductions(ii)                                      11
```

Figure 2.1: Pseudo-code for the FC labeling function with explicit BM.

search tree. This description hides two basic properties of the FC algorithm for which we gave two lemmas in Chapter 1, namely Lemma 1.1 and Lemma 1.2. Lemma 1.1 states that all future domain values are past-consistent and the order of constraint checks performed on a future domain value is in the order of instantiation. Lemma 1.2 states that values remain deleted from domains until the algorithm backtracks and uninstantiates the past variable that caused the deletion of the value, and that this past variable is the shallowest variable whose instantiation is in conflict with the value. FC maintains information about constraint checks for each domain value in the same way as BM. During the search, constraint checks that have been performed (both successful and unsuccessful) between future values and past instantiations are not repeated. Figures 2.1 to 2.4 give a version of the FC algorithm that explicitly remembers its past successful and unsuccessful checks using the $\mathsf{domain}_i^!$ array in the same way that BM uses it.

Differences between this new version of FC and the old version (Figure 1.11 to Figure 1.14) are marked with an exclamation mark beside the line number. One

```
function forward-check(ii,jj)
    i ← refv_ii                                                    1
    j ← refv_jj                                                    2
    for each v_j^l ∈ d_j                                           3
        if domain_j^l ≥ 0 then                                     !4
            if (v_i, v_j^l) ∉ c_i,j then                           5
                domain_j^l = −ii                                   6
            else                                                   !7
                domain_j^l = ii                                    !8
    if ∃k domain_j^k > 0 then                                      9
        return(True)                                               10
    else                                                           11
        return(False)                                              12
```

Figure 2.2: Pseudo-code for the forward-check function with explicit BM.

```
function undo-reductions(ii)
    i ← refv_ii                                                    1
    for ij = ii + 1 to n                                           2
        j ← refv_jj                                                3
        for each v_j^l ∈ d_j                                       4
            if abs(domain_j^l) = ii then                           !5
                domain_j^l = ii − 1                                !6
```

Figure 2.3: Pseudo-code for the undo-reductions function with explicit BM.

```
function fc-unlabel(ii)
    i ← refv_ii                                                    1
    hh ← ii − 1                                                    2
    h ← refv_hh                                                    3
    undo-reductions(hh)                                            4
    domain_h^el(v_h) ← −(hh − 1)                                   5
    if ∃k domain_h^k ≥ 0                                           !6
        then consistent ← True                                     7
        else consistent ← False                                    8
    return(hh)                                                     9
```

Figure 2.4: Pseudo-code for the FC unlabeling function with explicit BM.

minor difference to note is that the tests to see if a domain value is not deleted (that is, $\mathsf{domain}_i^j = 0$) have been changed to a test to see if the value is past-consistent (that is, $\mathsf{domain}_i^j \geq 0$). This difference does change the underlying algorithm. Other minor changes have been made in **forward-check** (lines 7–8 are added) and in **undo-reductions** (lines 5–6). In **forward-check** a successful past check is now recorded and in **undo-reductions** both past successful and past unsuccessful checks are now forgotten when a forward check is undone. Effectively these modifications do not change the FC algorithm although they make explicit the information that FC (implicitly) keeps about each domain value. That is, it keeps information in the same manner as BM. FC not only has the information of BM but also information of the future domains. At the time we developed the MFC algorithm we noted MFC's relationship to BM in terms of FC using the same data structure as BM but did not explore the issue further[28, 29]. Kondrak and van Beek[67, 68] have shown that FC visits a subset (not necessarily proper) of the nodes that BM visits (see Section 1.5.1). Bacchus and Grove[2] have shown that BM with a forward looking "oracle" that detects completely pruned future domains visits the same nodes as FC (*cf.* Section 2.7).

2.3 The Minimal Forward Checking Algorithm

The goal of MFC is to be a lazy version of FC which mimics FC's search but avoids unnecessary constraint checks, performing them only if they are needed. The FC algorithm only moves forward (deeper in the search tree) when there is at least one past-consistent value in each future domain. It ensures that this is true by pruning out all values inconsistent with the current instantiation, failing immediately if any future domain becomes completely pruned. The MFC algorithm minimizes the number of constraint checks performed by a forward check finding only the first value in each future (ordered) domain that is past-consistent. In order to maintain its relationship with FC it must remember all constraint checks with past variables in exactly the same manner as FC and it must perform constraint checks only when FC would need a value to be past-consistent. In order to remember all necessary constraint checks in exactly the same way as FC it must use the **domain**$_i^j$ array as used in Section 1.3.3 for BM. The MFC algorithm must also have a function which ensures that a value is past-consistent (where constraint checks are done in the order variables are instantiated) and a call to this function must be placed in the algorithm in places where the FC algorithm needs a value to be past-consistent in order to continue the search and maintain the properties of the FC algorithm. We present in Figures 2.5-2.9 the MFC algorithm which uses these two ingredients to perform a lazy forward checking search.

The MFC algorithm is very similar to the version of FC, shown in Figures 2.1-2.4, which uses the **domain**$_i^j$ array as used by BM. Differences between the two sets of figures for functions they have in common are marked with an exclamation mark on the MFC algorithm (beside the line numbers). One major difference between the two algorithms is the addition of a new function **past-consistent**. This function is the one mentioned above which ensures that a value is past-consistent. We begin our description of MFC by first explaining what the functions **past-consistent** and **min-forward-check** do.

The **past-consistent** function takes as input a value v_j^i in d_j and the index of the current instantiation ii. If the domain value v_j^i, has not been deleted (line 2), it checks

```
function mfc-label(ii)
    i ← refv_ii                                                    1
    consistent ← False                                             2
    for each v_i^j ∈ d_i while (not consistent)                    3
        if past-consistent(v_i^j,ii) then                          4
            consistent ← True                                      5
            v_i ← v_i^j                                            6
            for jj = ii + 1 to n while consistent                  7
                consistent ← min-forward-check(ii,jj)              8
            if not consistent then                                 9
                domain_i^j = −(ii − 1)                             10
                min-undo-reductions(ii)                            11
```

Figure 2.5: Pseudo-code for the MFC labeling function.

```
function past-consistent(v_j^j,ii)
    ok ← False                                                     1
    if domain_j^j ≥ 0 then                                         2
        ok ← True                                                  3
        for kk = domain_j^j + 1 to ii-1 while ok                   4
            k ← refv_kk                                            5
            if (v_k, v_j^j) ∈ c_k,j then                           7
                domain_j^j = kk                                    7
            else                                                   8
                domain_j^j = −kk                                   9
                ok ← False                                         10
    return(ok)
```

Figure 2.6: Pseudo-code for the **past-consistent** function.

```
function min-forward-check(ii,jj)
    i ← refv_ii                                              1
    j ← refv_ij                                              2
    found ← False                                            !3
    for each v'_j ∈ d_j while not found                      !4
        if past-consistent(v'_j,ii) then                     !5
            if (v_i, v'_j) ∉ c_i,j then                      6
                domain'_j = −ii                              7
            else                                             8
                domain'_j = ii                               9
                found ← True                                 !10
    return(Found)                                            !11
```

Figure 2.7: Pseudo-code for the min-forward-check function.

```
function min-undo-reductions(ii)
    i ← refv_ii                                              1
    for jj = ii + 1 to n                                     2
        j ← refv_ij                                          3
        for each v'_j ∈ d_j                                  4
            if abs(domain'_j) = ii then                      5
                domain'_j = ii − 1                           6
```

Figure 2.8: Pseudo-code for the min-undo-reductions function.

```
function mfc-unlabel(ii)
    i ← refv_ii                                                    1
    hh ← ii − 1                                                    2
    h ← refv_hh                                                    3
    min-undo-reductions(hh)                                        4
    domain_h^el(v_h) ← −(hh − 1)                                   5
    if ∃k domain_h^k ≥ 0                                           6
        then consistent ← True                                     7
        else consistent ← False                                    8
    return(hh)                                                     9
```

Figure 2.9: Pseudo-code for the MFC unlabeling function.

the domain value against every past instantiation that it has not already been checked against (line 4) recording the result of the checks in domain_j^i. This "past checking" is performed in the order variables were instantiated. If the result of any past constraint check is false no further checking is performed and the function returns **False**. If every past check succeeds, the function returns **True**.

The **min-forward-check** function in Figure 2.7 is quite different from the **forward-check** function in Figure 2.2. The purpose of the **forward-check** function is to delete all values in a future domain that are inconsistent with the current instantiation, returning **True** if a value remains, and **False** otherwise. The purpose of the **min-forward-check** function is to find the first past-consistent value in a future domain, return **True** if it finds one, and **False** otherwise. The **min-forward-check** function is passed the index to the current instantiated variable ii and the index to some future variable jj. It loops through the future domain indexed by jj until it exhausts the domain or it finds a past-consistent value (line 4-10). As it is looping through the future domain values, it first checks each value to make sure that FC would be examining this value. It does this by calling the **past-consistent** function in line 5. If the value is past-consistent a forward check is performed (lines 5-10) otherwise the value is skipped. Notice that the function **past-consistent** records the deepest instantiation which successfully checked against that

value or the shallowest value which unsuccessfully checked against that value. That is, **past-consistent** records the same information about successful and unsuccessful constraint checks for the value as FC does. If the forward check is successful, **found** is set to **True**, the loop exits and the function returns **True**. If all values in the future domain have been examined and no value is past-consistent the function returns **False**.

We now look at the labeling function **mfc-label**. This function is quite similar to the **fc-label** function in Figure 2.1. The only major difference between the two (except for calling **min-forward-check** instead of **forward-check**) is that **mfc-label** no longer assumes that every value in the domain of the current instantiation is past-consistent (line 4). To see why this is necessary consider how MFC is avoiding constraint checks. When the MFC algorithm is moving forward to instantiate a variable that isn't being revisited because of a backtrack, the first non-deleted value in every future domain is past-consistent. The call to **past-consistent** in this case is just a test to see whether a value has been deleted[1]. However, if the call to **mfc-label** is to relabel a variable which has been backtracked to then there is no guarantee that any value in the domain is past-consistent (the first past-consistent value was used in the forward move). The call to **past-consistent** must be performed at this point. Altogether there are only two points where **past-consistent** must be called, in **mfc-label** and in **min-forward-check**. The function **min-undo-reductions** in Figure 2.8 which undoes the effect of a minimal forward check is exactly the same as the function **undo-reductions** for FC shown in Figure 2.3. The function **mfc-unlabel** in Figure 2.9 is exactly the same as the function **fc-unlabel** for FC shown in Figure 2.4.

2.4 A Sample Execution of the MFC Algorithm

In order for the reader to compare the execution of MFC to the previously described algorithms we continue to use the same example graph colouring problem as before: Consider a graph colouring problem where you are given 4 nodes that are adjacent to each other and are told that the first node can only be coloured red, the second node

[1]This call can easily be avoided if one is willing to maintain an array which points to the first non-deleted value in every future domain.

with green or orange, the third node with blue or green, and the fourth node with green, blue or red. Does this problem have a solution in which all adjacent nodes are coloured differently? To represent this as a CSP, we first define four variables, $\{v_1, v_2, v_3, v_4\}$ which represent the four nodes. The domains of the four variables are

$$d_1 = \{r\}, d_2 = \{g, o\}, d_3 = \{b, g\}, d_4 = \{g, b, r\}$$

where we just use the first letter of each colour to represent the domain elements. Finally we define a set of constraints which enforce the general constraint that adjacent nodes (variables) cannot be assigned the same colour.

$$c_{1,2} = \{(r, g)(r, o)\}$$

$$c_{1,3} = \{(r, b)(r, g)\}$$

$$c_{1,4} = \{(r, g)(r, b)\}$$

$$c_{2,3} = \{(g, b)(o, b)(o, g)\}$$

$$c_{2,4} = \{(g, b)(g, r)(o, g)(o, b)(o, r)\}$$

$$c_{3,4} = \{(b, g)(b, r)(g, b)(g, r)\}$$

Constraints are symmetric so for example $c_{2,1} = \{(g, r)(o, r)\}$.

Figure 2.10 outlines the search performed by MFC on this graph colouring problem. As before, domain values are shown with lists of instantiated variables with which they have been checked. Some variables in the lists have superscripts ($\surd$) and ($-$) denoting respectively successful and unsuccessful constraint checks performed in the current search step. If the variable in the list has no superscript then it represents a past successful check. 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 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

Step	d_1	d_2	d_3	d_4	Checks
0	r	g o	b g	g b r	0
1	$v_1 \leftarrow$ r	g $\{v_1^{\surd}\}$ o	b $\{v_1^{\surd}\}$ g	g $\{v_1^{\surd}\}$ b r	3
2	$v_1 =$ r	$v_2 \leftarrow$ g	b $\{v_1, v_2^{\surd}\}$ g	g $\{v_1, v_2^{\times}\}$ b $\{v_1^{\surd}, v_2^{\surd}\}$ r	4
3	$v_1 =$ r	$v_2 =$ g	$v_3 \leftarrow$ b	b $\{v_1, v_2, v_3^{\surd}\}$ r $\{v_1^{\surd}\}$	2
4	$v_1 =$ r	$v_2 =$ g	g $\{v_1^{\surd}, v_2^{\surd}\}$	b $\{v_1, v_2\}$	2
5	$v_1 =$ r	o $\{v_1^{\surd}\}$	b $\{v_1\}$ g $\{v_1\}$	g $\{v_1\}$ b $\{v_1\}$	1
6	$v_1 =$ r	$v_2 \leftarrow$ o	b $\{v_1, v_2^{\surd}\}$ g $\{v_1\}$	g $\{v_1, v_2^{\surd}\}$ b $\{v_1\}$	2
7	$v_1 =$ r	$v_2 =$ o	$v_3 \leftarrow$ b	g $\{v_1, v_2, v_3^{\surd}\}$ b $\{v_1\}$	1
8	$v_1 =$ r	$v_2 =$ o	$v_3 =$ b	$v_4 \leftarrow$ g	0
Total					15

Figure 2.10: Sample execution of MFC.

performing previously avoided 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. The value blue is inconsistent with v_3 and the value red is inconsistent with v_1. The search backtracks to v_3 and attempts to find another consistent value but the performance of previously avoided forward checks for the value green shows it to be inconsistent with v_2 (step 4). Also in this step notice that domain value blue is undeleted in domain d_4 as it is no longer inconsistent with v_3 (and domain value red remains deleted as it is inconsistent with the instantiation of v_1). 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 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.

2.5 Theoretical Results

Many CSP search algorithms have been presented in the past without a formal proof of correctness. Recently, Kondrak and van Beek[67, 68] have given a methodology for proving CSP search algorithms sound and complete (*cf.* Section 1.5.1). A CSP search algorithm is sound if all solutions claimed by the algorithm really are solutions (that is, the set of assignments satisfies all the constraints). A CSP search algorithm is complete if it is able to find all possible solutions. Kondrak and van Beek's methodology rests on the characterization of the necessary and sufficient conditions (called the characterizing conditions) for a search tree node to be visited by a particular search algorithm. From the proofs that the search algorithm under question satisfies those necessary and sufficient conditions one can show the correctness (soundness and completeness) of the algorithm. One can also use the characterizing conditions to give partial orders of the algorithms according to the number of nodes visited, and give partial orders of the algorithms according to the number of constraint checks

performed. The following proofs assume that the order of instantiation is fixed.

Before proving the soundness and completeness of MFC we first clarify when MFC visits a node. In the following proofs we first need a lemma that shows that MFC does not extend a branch in the search tree for a value in the current domain that is not past-consistent (that is, MFC does not visit a node for a value that is not past-consistent).

Lemma 2.1 *MFC will not visit a node* $\{v_1 \leftarrow v_1^{s1}, v_2 \leftarrow v_2^{s2}, \ldots, v_k \leftarrow v_k^{sk}\}$ *if the value* v_k^{sk} *is inconsistent with the assignment of any past variable* $\{v_1, \ldots, v_{k-1}\}$.

Proof of Lemma 2.1 In Figure 2.5, before attempting an instantiation of a non-deleted value (lines 3–4), mfc-label calls the function past-consistent (Figure 2.6). past-consistent checks the value v_k^{sk} against each past variable that it has not already been checked against in the order of instantiation. If the value v_k^{sk} is inconsistent with any past variable it is marked as deleted because of that instantiation and the node is not visited (lines 4–6, mfc-label) as the value returned by past-consistent is False (lines 8–10). □

As in the case for FC[67, 68], MFC has only one characterizing condition for a search tree node to be visited, namely,

A node is consistent and its parent is consistent with all variables.

This condition states that the set of assignments associated with the node must be a partial solution and that the set of assignments for the nodes' parent must be consistent with at least one value in every future (with respect to the parent) domain. We now prove the sufficient and necessary conditions for MFC. These proofs are based on similar proofs given for FC in [67] with appropriate modifications made for MFC.

Theorem 2.1 *MFC visits a node if it is consistent and its parent is consistent with all variables.*

Proof of Theorem 2.1 Assume that the node $\{v_1 \leftarrow v_1^{s1}, v_2 \leftarrow v_2^{s2}, \ldots, v_{i-1} \leftarrow v_{i-1}^{s_{i-1}}\}$ is consistent with all variables and that its child $p = \{v_1 \leftarrow v_1^{s1}, v_2 \leftarrow v_2^{s2}, \ldots, v_i \leftarrow v_i^{s_i}\}$

is consistent but not visited by MFC. We derive a contradiction. First, find the deepest node in the search tree $p' = \{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_j \leftarrow v_j^{s_j}\}$, $j < i - 1$ which is visited by MFC but its child $\{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_j \leftarrow v_j^{s_j}, v_{j+1} \leftarrow v_{j+1}^{s_{j+1}}\}$ is not visited by MFC. (At least $\{v_1 \leftarrow v_1^{s_1}\}$ is visited by MFC). Node p' is consistent with all variables as $\{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_j \leftarrow v_j^{s_j}\}$ is along the path of $\{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_{i-1} \leftarrow v_{i-1}^{s_{i-1}}\}$. Therefore when MFC visits p' it does not delete all values from any future domain. A branch is therefore extended for variable v_j for each past-consistent value in its filtered domain. As $\{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_i \leftarrow v_i^{s_i}\}$ is consistent its subtuple $\{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_j \leftarrow v_j^{s_j}, v_{j+1} \leftarrow v_{j+1}^{s_{j+1}}\}$ is also consistent and therefore the partially filtered domain of v_{j+1} must still contain $v_{j+1}^{s_{j+1}}$. Therefore $\{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_j \leftarrow v_j^{s_j}, v_{j+1} \leftarrow v_{j+1}^{s_{j+1}}\}$ is visited which is a contradiction. □

Theorem 2.2 *MFC visits a node only if it is consistent and its parent is consistent with all variables.*

Proof of Theorem 2.2 We first prove the first conjunct. Assume that MFC visits a node $p = \{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_i \leftarrow v_i^{s_i}\}$ and that p is inconsistent. Then there exists a pair of values in p which are inconsistent[2]. Find the node $p' = \{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_k \leftarrow v_k^{s_k}\}$ such that $v_k \leftarrow v_k^{s_k}$ is the shallowest assignment inconsistent with an assignment $v_l \leftarrow v_l^{s_l}$ in $\{v_{k+1} \leftarrow v_{k+1}^{s_{k+1}}, \ldots, v_l \leftarrow v_l^{s_l}, \ldots, v_i \leftarrow v_i^{s_i}\}$, $k < i - 1$. We have three mutually exclusive situations. (i) When MFC visits node $\{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_k \leftarrow v_k^{s_k}\}$, the value $v_l^{s_l}$ is deleted from the current domain of v_l and is not undeleted until the instantiation of v_k is changed. (ii) When MFC visits one of the nodes $\{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_j \leftarrow v_j^{s_j}\}$, $k \leq j \leq l - 1$ the value $v_l^{s_l}$ is deleted from the current domain of v_l and is not undeleted until the instantiation of v_k is changed. Or (iii) when MFC tries to label v_l, the value $v_l^{s_l}$ is deleted from the current domain of v_l and is not undeleted until the instantiation of v_k is changed. Therefore $\{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_l \leftarrow v_l^{s_l}\}$ and its descendant p are not visited by MFC.

To prove the second conjunct, assume that MFC visits a node $p = \{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_i \leftarrow v_i^{s_i}\}$ although its parent is inconsistent with some future variable. Find

[2]Kondrak[67] mistakenly assumes that $v_i \leftarrow v_i^{s_i}$ is inconsistent with some past variable.

the deepest node $\{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_j \leftarrow v_j^{s_j}\}$ $j \leq i-2$ which is consistent with all variables. If no such node exists then the instantiation of $v_1 \leftarrow v_1^{s_1}$ must have deleted all values out of a future domain and MFC never visits p. Otherwise, the child $p' = \{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_{j+1} \leftarrow v_{j+1}^{s_{j+1}}\}$ is visited by MFC. When MFC is at p' it must delete out one of the future domains and this branch is abandoned. Therefore no descendant of p' is visited by MFC which is a contradiction. □

The following two theorems are proven with similar proofs in [67].

Theorem 2.3 *FC visits a node if it is consistent and its parent is consistent with all variables. (Kondrak and van Beek[67])*

Theorem 2.4 *FC visits a node only if it is consistent and its parent is consistent with all variables. (Kondrak and van Beek[67])*

We can now show the correctness of the MFC algorithm.

Theorem 2.5 *The MFC algorithm is sound.*

Proof of Theorem 2.5 A solution is claimed by MFC when it visits the node $\{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_n \leftarrow v_n^{s_n}\}$. Theorem 2.2 guarantees that the node is consistent. □

Theorem 2.6 *The MFC algorithm is complete.*

Proof of Theorem 2.6 Assume that there is a node $p = \{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_n \leftarrow v_n^{s_n}\}$ which is consistent. Then its parent $\{v_1 \leftarrow v_1^{s_1}, v_2 \leftarrow v_2^{s_2}, \ldots, v_{n-1} \leftarrow v_{n-1}^{s_{n-1}}\}$ is also consistent and the parent is consistent with v_n. Therefore the parent is consistent with all variables. Theorem 2.1 guarantees that MFC will visit the node. As p is a node with all n variables assigned it will be claimed as a solution by MFC. □

We can now state a few theorems on the relationship between FC and MFC.

Theorem 2.7 *The FC algorithm and the MFC algorithm visit the same nodes in the same order.*

Proof of Theorem 2.7 That they visit the same nodes is a direct consequence of Theorems 2.1 2.4. They visit nodes in the same order as both algorithms are performing backtracking tree search. □

Theorem 2.8 *The worst case performance of MFC in terms of the number of constraint checks performed is the number of constraint checks performed by FC.*

Proof of Theorem 2.8 From Theorem 2.7 we know that both algorithms visit the same nodes in the same order. We also know from the description of the algorithm (Section 2.3) that MFC remembers past consistency checks in order to avoid redundant constraint checks. Therefore MFC performs no more constraint checks than FC. In order to show the worst case results we need to show that the MFC algorithm either performs the same number of constraint checks as FC or fewer constraint checks. We do so by two simple examples. First, assume we are given a CSP with two variables where each domain has only one value. Whether or not the two variables are consistent when assigned those values, the FC and MFC algorithm perform the same number of constraint checks. Therefore MFC and FC can perform the same number of constraint checks. We now show that MFC can perform fewer constraint checks than FC. Assume that we are again given a CSP with two variables v_1 and v_2. The first domain is $d_1 = \{a\}$ and the second domain is $d_2 = \{b, c\}$. Assume the constraint $c_{1,2}$ allows $\{v_1 \leftarrow a, v_2 \leftarrow b\}$. MFC will perform only one constraint check while FC will perform 2. Therefore MFC can perform better than FC. □

These proofs rely on a fixed instantiation ordering. We conjecture that the soundness and completeness results hold for both MFC and FC under a dynamic variable ordering. We also conjecture that Theorem 2.8 holds for dynamic variable orderings that depend on the (sub)structure of the constraint graph. A proof of these results are left as future work. What is easily provable is that the MFC algorithm can perform worse in terms of the number of constraint checks than the FC algorithm if they are using the FF heuristic.

Theorem 2.9 *MFC-FF can perform worse than FC-FF in terms of the number of constraint checks performed.*

Proof of Theorem 2.9 We prove this theorem by a simple example. Assume that we are given the following CSP. There are four variables $\{v_1, v_2, v_3, v_4\}$ with domains

$$d_1 = \{a\}, d_2 = \{b, c\}, d_3 = \{d\}, d_4 = \{e\}$$

and the constraints are

$$c_{1,2} = \{(a, b)\}$$

$$c_{1,3} = \{(a, d)\}$$

$$c_{1,4} = \{(a, e)\}$$

$$c_{2,3} = \{(b, d)(c, d)\}$$

$$c_{2,4} = \{(c, e)\}$$

$$c_{3,4} = \{(d, e)\} \ .$$

A search by FC-FF is outlined in Figure 2.11 and a search by MFC-FF is outlined in Figure 2.12. In Figure 2.11, after instantiating v_1, FC-FF picks d_2 as the next variable (in lexical order) to instantiate as it has the smallest domain size. After instantiating v_2, FC-FF finds that there is no consistent value left in d_4 and the search terminates with no solution found. A total of 6 constraint checks performed. In Figure 2.12, after instantiating v_1, MFC-FF picks d_3 as the next variable to instantiate as it has the smallest *known* domain size. After instantiating v_3 MFC-FF finds that the first undeleted values in d_2 and d_4 are past-consistent and it then picks v_4 to instantiate. Only then does it find that there is no consistent value in d_2 and the search terminates. MFC-FF performs 7 constraint checks.. □

Finally, we note where MFC belongs in Nadel's relative ordering of CSP algorithms by approximate amount of arc consistency performed (*cf.* Section 1.4.1). MFC can perform in the worst case the same amount of arc consistency performed by FC. MFC performs more arc consistency processing than BM as it prunes the future domains. Therefore the the proper place in Nadel's relative ordering for MFC is between BM and FC.

Step	d_1	d_2	d_3	d_4	Checks
0	a	b c	d	e	0
1	$v_1 \leftarrow a$	b $\{v_1^{\checkmark}\}$ c $\{v_1^{\times}\}$	d $\{v_1^{\checkmark}\}$	e $\{v_1^{\checkmark}\}$	1
2	$v_1 = a$	$v_2 \leftarrow b$	d $\{v_1, v_2^{\checkmark}\}$	e $\{v_1, v_2^{\checkmark}\}$	2
Total					6

Figure 2.11: The execution of FC-FF on a problem for the proof of Theorem 2.9.

Step	d_1	d_2	d_3	d_4	Checks
0	a	b c	d	e	0
1	$v_1 \leftarrow a$	b $\{v_1^{\checkmark}\}$ c	d $\{v_1^{\checkmark}\}$	e $\{v_1^{\checkmark}\}$	3
			d_2	d_4	
2	$v_1 = a$	$v_3 \leftarrow d$	b $\{v_1, v_3^{\checkmark}\}$ c	e $\{v_1, v_3^{\checkmark}\}$	2
				d_2	
3	$v_1 = a$	$v_3 = d$	$v_4 \leftarrow e$	b $\{v_1, v_3, v_4^{\checkmark}\}$ c $\{v_1^{\checkmark}\}$	2
Total					7

Figure 2.12: The execution of MFC-FF on a problem for the proof of Theorem 2.9

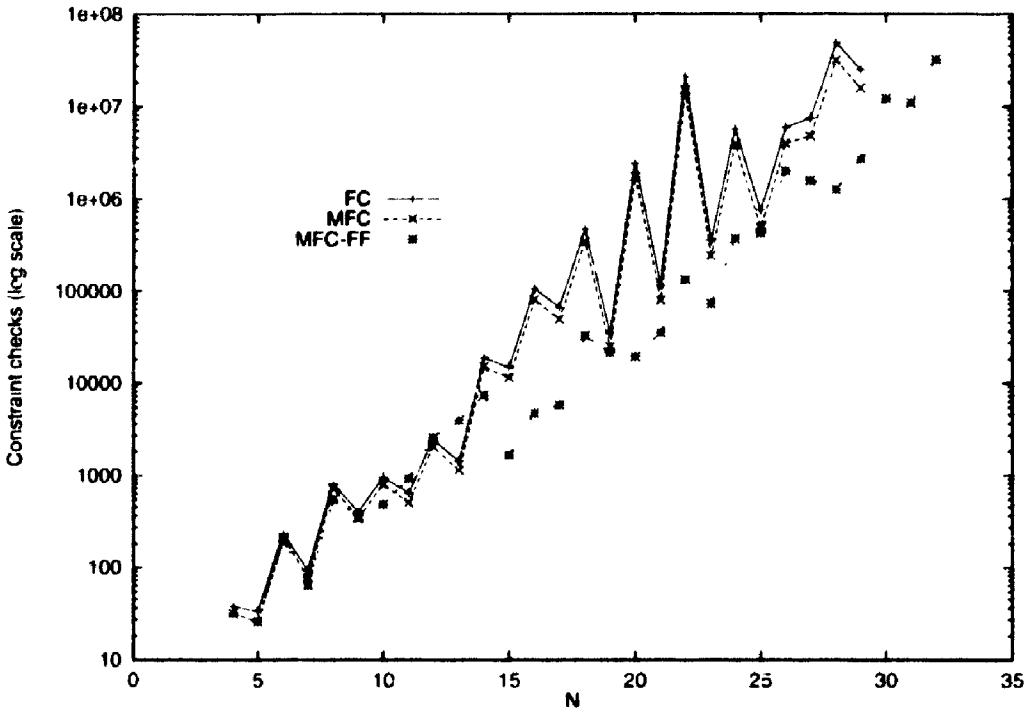

Figure 2.13: Comparison of FC, MFC, and MFC-FF by constraint checks on the n-queens problem.

2.6 The n-queens Problem Revisited

We now continue our previous empirical comparison (*cf.* Section 1.5.2) using the n-queens problem. We run the FC and MFC algorithms on the n-queens problem for $n \in \{4..29\}$ and for the MFC-FF algorithm we use $n \in \{4..32\}$. As before, for each algorithm we count the number of constraint checks performed in finding the first solution. A comparison of FC, MFC and MFC-FF is displayed in Figure 2.13. As expected by theory MFC is better than FC at every point. On average MFC performs only 75% of the constraint checks that FC performs. However, we could only run MFC-FF for $n \in \{4..32\}$ as it began to perform badly. Figure 2.14 contrasts the performance of MFC-FF with that of FC-FF. Clearly FC-FF is much better than MFC-FF. The FF heuristic obviously interacts badly with MFC's lack of knowledge of the true domain size of the future domains (at least for the n-queens problem). The n-queens problem is the one well known problem we found that shows just how poorly MFC-FF can perform. In Chapter 4 we discuss improvements to MFC-FF.

2.7 Related Work

The MFC algorithm has been rediscovered since our original papers[28, 29, 30]. Bacchus and Grove[1] rediscovered MFC by extending BM with a forward looking "oracle" that detects completely pruned future domains. They prove that BM with such an oracle explores exactly the same nodes as FC and MFC. The algorithm they describe is equivalent to MFC. However, they do use a more simplified data structure than we described in [30]. Our data structure uses $O(nm^2)$ space while Bacchus and Grove's use the $|domain|$ data structure which uses $O(nm)$ space. Although this reduction in space seems to be an improvement it is actually a tradeoff of time for space. Our data structure has information indicating exactly which future values to remove past constraint checks from (while uninstantiating a variable) while Bacchus and Grove's data structure does not have that information and the algorithm has to search all the future domains in order to remove past constraint checks. In this thesis we use Bacchus and Grove's data structure as it greatly simplifies the presentation of the MFC

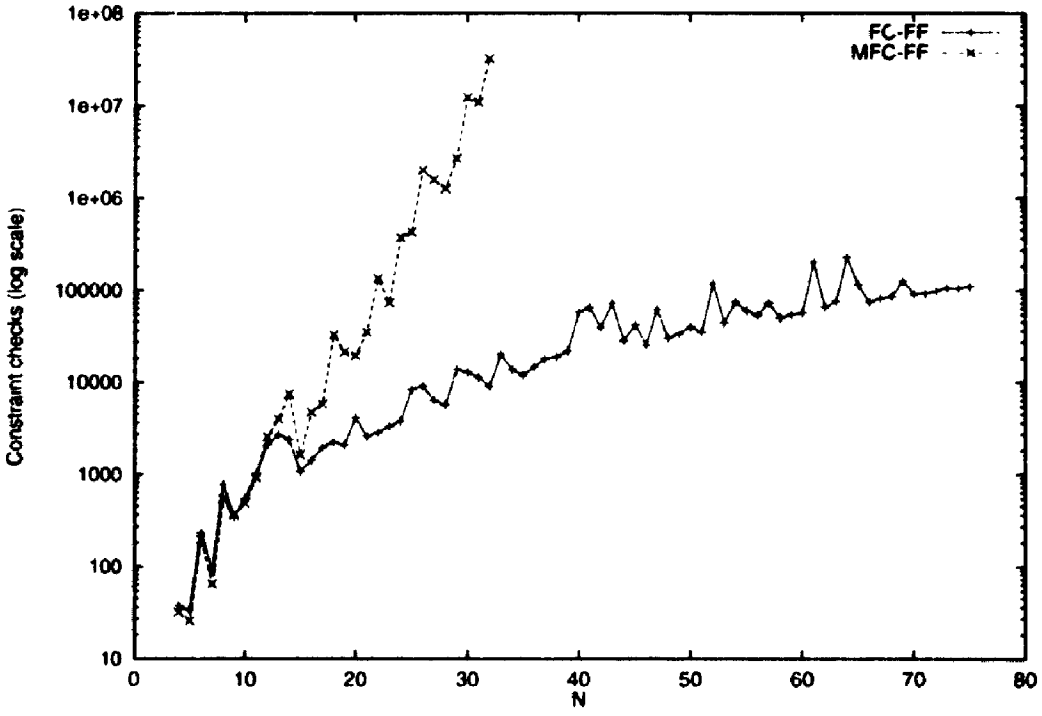

Figure 2.14: Comparison of FC-FF and MFC-FF by constraint checks on the n-queens problem.

algorithm. Many minor implementation details can be ignored.

Kwan and Tsang[71] have created a version of MFC with the addition of a small improvement that we had left for future work in [30]. The small improvement is the addition of Prosser's "BM" to FC[93] which in turn can be added to MFC. An example of Prosser's BM savings for FC or MFC is the following. If a value, v_i^j, for the current variable v_i causes some future domain d_j to become empty, instead of recording that v_i^j is inconsistent with instantiation $ii - 1$, record it as inconsistent with the deepest instantiation that can change d_j (that is, the deepest variable whose uninstantiation could undelete a value in d_j). This would ensure that FC or MFC never instantiates v_i to v_i^j as long as v_i^j would empty the future-connected domain d_j. In this thesis, we do not explore the addition of this improvement to MFC.

Finally, Zweben[121] loosely describes a "lazy" version of FC in the context of a scheduling system. Zweben describes the necessity of finding only one past-consistent value in each future (in their case possibly infinite) domain. However, they do not give a concrete description of their algorithm and the data they present for the n-queens problem leads us to believe that they did not implement the MFC algorithm. The MFC algorithm's performance is much better on the n-queens instances they report constraint checks for. From their results we infer that the algorithm they describe does not have the backmarking component to avoid repeated checks.

2.8 Are There Other Lazy CSP Search Algorithms?

> "Look, I came here for an argument."
>
> "No you didn't!" The argument sketch, Monty Python.

In this section we discuss whether it is possible that there are other CSP search algorithms that can be made lazy in the same way that MFC is lazy. The MFC algorithm is *minimal* in the sense that it performs the fewest constraint checks necessary in order to satisfy the requirement of FC that there be at least one past-consistent value in every future domain. The MFC algorithm is a *lazy* CSP search algorithm as it is minimal and it mimics FC's search preserving the results of all constraint checks

so that no redundant checks are performed. Are there other lazy CSP search algorithms where one can minimally preserve the arc consistency property that the original algorithm establishes while preserving the knowledge of the work that has been done in case more arc consistency processing is needed at a later stage in the search? Immediate candidates for possible lazy algorithms are PL, FL and RFL. These three algorithms all establish some degree of arc consistency in the future domains before allowing the search to move ahead. As the PL and FL algorithms do not really establish any nice theoretical property in the future domains (and any argument based on RFL will also apply to them) we focus only on whether or not a lazy RFL algorithm exists.

We begin by giving the following definition of a sub-CSP.

Definition 2.1 For a given binary CSP as in Definition 1.1, $V = \{v_1, \ldots, v_n\}$, a set of finite discrete domains $D = \{d_1, \ldots, d_n\}$ and a set of symmetric constraints $C = \{c_{i,j} | 1 \leq i < j \leq n\}$, a *sub-CSP* is the above CSP, with the following changes. The set of variables V remains the same, the domains are now $D' = \{d'_1, \ldots, d'_n\}$ where each domain d'_i is allowed to be any subset of d_i, and the set of symmetric constraints $C' = \{c'_{i,j} | 1 \leq i < j \leq n\}$ are now limited to those values in D'.

During a backtracking search, the set of future domains can be seen as a CSP (which we will call the future CSP). The FC algorithm at every instantiation step ensures that every value in the future CSP is consistent with the current instantiation. The MFC algorithm however finds a partitioning of the future CSP into two mutually exclusive sub-CSPs at every instantiation step. In one sub-CSP, every value is past-consistent, and in the other the values are in some intermediate state of past consistency. To the FC algorithm only the first sub-CSP matters to establish the property of finding one past-consistent value ahead in every future domain. The necessary ingredients to creating the MFC algorithm is finding out which sub-CSP must be found at every instantiation step and creating a data structure that can remember the partial information the search will have of constraint checks being performed on the future domains.

The RFL algorithm establishes full arc consistency in the future domains at every

step. The full arc consistency algorithm is quite expensive to use every time a value is instantiated (*cf.* Section 1.4) although some preliminary results show that RFL (MAC) can be very effective on large problems with sparse graphs[97]. It appears that it may be possible to create a lazy version of RFL. The necessary sub-CSP that must be found is one in which every value is fully arc consistent. This sub-CSP will not necessarily have to consist of the entire future CSP. For example, if the CSP only consists of two variables where the first variable has one domain value and the second variable has two and the value in the first domain is consistent with the first value in the second domain, then a lazy RFL would only have to establish full arc consistency between the first values in each domain. They mutually support each other. The other value in the second domain can be ignored until the search actually needs it. We envisage a lazy RFL would begin by attempting to find supporting values for every first value in each future domain. That is the initial sub-CSP would consist of the first value of every future domain. The algorithm to create the appropriate sub-CSP would loop over the values in the sub-CSP ensuring that each value has a support that is also in the sub-CSP. If no such support is found in a particular domain then a new value must be brought in from the the rest of the future CSP which is a support. If no supporting value is found then the algorithm can safely delete the value at that level in the search tree. The algorithm would finish when every value in the sub-CSP is supported by a value that is in the sub-CSP.

After investigating the MFC algorithm we left the design, implementation and empirical evaluation of a lazy RFL as future work. Recently, Gaspin et al[48] have developed a lazy RFL which they call LAC_7 (which has made the point of this section moot). The authors follow our idea of a lazy CSP search algorithm to develop their algorithm which is very similar to the way we envisage a lazy RFL. Currently they use LAC_7 as a preprocessing algorithm to tell if any domain would be completely pruned by full arc consistency preprocessing. However, from the description they give of their algorithm it is obvious that it can immediately be put into the context of a backtracking search to be a lazy RFL.

2.9 Summary

In this chapter we have motivated the development of the MFC algorithm and given a formal description of the algorithm. We have also shown that the MFC algorithm is sound and complete, and that its worst case performance in terms of the number of constraint checks performed is the number of constraint checks performed by the FC algorithm. We have also shown that MFC-FF can be expected to occasionally perform worse than FC-FF. We have also discussed the existence of other lazy CSP search algorithms. In the next chapter we discuss what "hard" randomly generated problems are and present an empirical comparison of MFC, MFC-FF, FC, FC-FF, BM, and BT on these random problems.

Chapter 3

An Empirical Comparison on Hard Randomly Generated Problems

"The fundamental principal of science, the definition almost, is this: the sole test of the validity of any idea is experiment" - Richard P. Feynman

3.1 Introduction

As theoretical results on the average case complexity for CSP search algorithms are extremely difficult to derive, empirical studies need to be performed in order to compare the general performance of CSP search algorithms[25, 82] (*cf.* section 1.5). Previous empirical studies using specific problems, such as the n-queens problem or the zebra problem, are not convincing as the results are only representative for one problem. In order to alleviate this problem researchers have focused on comparisons of search algorithms using randomly generated problems. Unfortunately, past comparisons using randomly generated problems are unconvincing as the problems generated are usually easy to solve (that is, easy to determine their solubility or insolubility) and are therefore unable to differentiate between the different algorithms in a convincing manner. Only recently, through the discovery that NP-complete problems exhibit a phase transition phenomenon in which problems in the phase transition peak are on average hardest to solve, has it been understood how to create randomly generated problems that are hard to solve. These hard randomly generated problems are now used to compare CSP search algorithms for two reasons. The first is that because they are much harder on average to solve they will bring out significant performance differences between search algorithms. That is, poorly designed search algorithms will perform worse for these problems. The second reason is that these hard randomly

generated problems are more structured than other randomly generated problems in the sense that they have long chains of partial solutions within them and structure is a characteristic of real problems. We begin this chapter with a description of the phase transition exhibited by NP-complete problems and more specifically the phase transition exhibited by CSPs. This is followed by a large empirical comparison, using hard randomly generated CSPs, of many of the algorithms introduced so far in this thesis.

3.2 Hard Constraint Satisfaction Problems

It is common in statistical mechanics to model the properties of complex systems by a few "order" parameters which summarize the properties of the system[13, 45]. Recent empirical and theoretical work [13, 17, 18, 19, 31, 35, 50, 52, 51, 63, 79, 91, 94, 102, 104, 103, 118, 119, 120] has studied the properties of NP-complete problems using this approach. The major result of these studies is that NP-complete problems appear to have a phase transition which occurs as one of the order parameters is varied causing problems to go from being soluble to being insoluble. The highest average cost (the phase transition peak) for solving these problems occurs where 50% of the problems are soluble [17, 18, 79][1]. For finite discrete problems, the phase transition occurs over a range of values of the order parameter. In the limit, as the size of the problem grows large, the phase transition becomes an instantaneous transition from soluble to insoluble problems. We follow the popular convention that problems in the transition peak are called "*hard problems*"[2].

One possible use of the phase transition is to exploit the knowledge of its existence to avoid exponential behavior by algorithms attempting to solve a problem. If one could accurately predict for a particular problem that it lay near the location of

[1] This is also known as the *crossover* point.

[2] The term "hard problems" is used as it is conjectured that *on average* many of the problems in the phase transition peak are relatively hard to solve by any CSP search algorithm. There are problems that are in the phase transition peak which are relatively easy to solve but they are in the minority.

the phase transition peak, it may then be reasonable to modify or reformulate the problem so that it would lie away from the phase transition peak. Such predictors of the location of the phase transition peak have been found. The most famous example is the ratio of clauses to variables for randomly generated 3-SAT problems being approximately 4.2[79]. The use of the phase transition phenomenon that is of interest here is the use of a predictor for the location of phase transitions in binary CSPs to randomly generate problems that are near the phase transition peak. We call these randomly generated binary CSPs that are near the phase transition peak "*hard random problems*". These hard random problems can be used as a testbed to differentiate the capabilities of CSP search algorithms.

Recent studies by Smith[102], Smith & Dyer[103], and Prosser[91, 94], have looked at the phase transition phenomenon exhibited by binary CSPs. They model binary CSPs using the 4-tuple $\langle n, m, p_1, p_2 \rangle$ where n is the number of variables, m is the domain size for all variables, p_1 is the probability that a non-trivial constraint exists between two variables (called the *constraint density*), and p_2 is the probability, conditional on the existence of a non-trivial constraint, that a pair of values between two variables is inconsistent (called the *constraint tightness*). To observe whether CSPs exhibit a phase transition they perform a series of experiments where they fix three of the parameters, namely n, m, and p_1, and vary the fourth parameter p_2 in small increments. The parameter p_2 partially determines how constrained a problem is and therefore seems to be a natural order parameter to vary[3]. For each setting of p_2, a large number of randomly generated CSPs were created and either the FC-FF or the FC-CBJ-FF algorithms were run on each problem instance counting the number of constraint checks performed as the measure of the complexity of the problem. We display a graph in Figure 3.1 which shows the result of a similar experiment. To create the graph in Figure 3.1 we first randomly generate 50 binary CSPs for each setting of $n = 10$, $m = 6$, and $p_1 = 0.5$ varying p_2 in increments of 0.01 from 0.01 and 0.99 (*cf.*

[3]One could also try to observe phase transitions by varying p_1. An example of such a phase transition occurs for graph colouring problems which exhibit a phase transition at a particular degree of graph connectivity[13, 63].

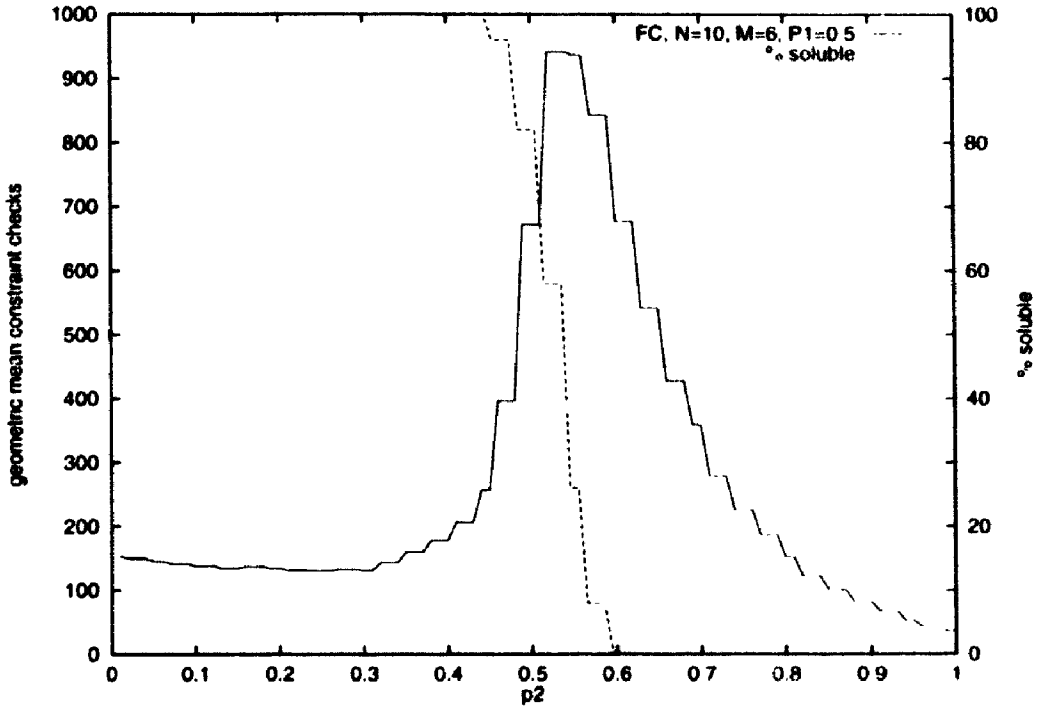

Figure 3.1: The geometric mean of the number of constraint checks performed solving each problem by the FC algorithm in the set of 50 problems randomly generated from $\langle 10, 6, 0.5 \rangle$ as p_2 is varied from 0.01 to 0.99 in increments of 0.01. The percentage of problems that are soluble in each set of 50 problems is also displayed.

Section 3.3 for a detailed explanation of how random problems are created). On each problem generated we run the FC algorithm counting the number of constraint checks performed. Along the x-axis of Figure 3.1 are the values of p_2 and along the y-axis on the left are the geometric mean number of constraint checks performed for each set of 50 problems (*cf.* the discussion in Section 3.2.1 for an explanation of why the geometric mean is used). The y-axis on the right represents the percentage of soluble problems in each set of 50 problems.

As expected, the graph in Figure 3.1 shows a rapid transition in the average cost of solving a binary CSP. The shape of the phase transition appears to begin at approximately $p_2 = 0.45$ and finishes at approximately 0.7 peaking between 0.52 and 0.54. The percentage solubility graph drops rapidly from all problems soluble to all problems insoluble starting at $p_2 = 0.45$ and finishing at 0.6. The 50% point of solubility occurs nearest to the points 0.52 to 0.54 (actually the closest we get to a 50% point is 58% for the 3 data points and then it suddenly drops to 26%). The points where the hardest problems on average are observed coincide with the approximate location of the 50% point of solubility. Smith calls the range of p_2 values over which the solubility graph drops from at least 99% soluble to less than 1% soluble the *mushy region* to emphasize that the phase transition occurs over a range of values of p_2 for these finite problems. The mushy region overlaps to a great extent the set of values of p_2 for which the phase transition is observed.

It would seem that a reasonable way of creating a predictor of the phase transition peak is to use the fact that the crossover point co-occurs with the peak. However, it is not easy to formulate such a predictor. In [102, 103], Smith observes that the phase transition peak for binary CSPs also appears to co-occur very near the point where the number of solutions to a problem is one. Intuitively, this seems reasonable as problems which have no solution are over-constrained and conversely, problems with more than one solution are under-constrained. Problems with just one solution are the intuitive "break even point"[4]. For random problems parameterized with the

[4]Of course intuition doesn't always match reality. There are problems, for example graph colouring, which have many solutions when they have one solution. Smith's intuition is valid only for

above 4-tuple. Haralick and Elliot[61], give an expected number of solutions formula for binary CSPs parameterized using the 4-tuple given above:

$$\mathsf{E(Soln)} = \mathsf{m}^{\mathsf{n}}(1 - \mathsf{p}_2)^{\frac{\mathsf{n}(\mathsf{n}-1)}{2}\mathsf{p}_1} \tag{3.1}$$

where the expected number of solutions is the number of possible instantiations in a CSP multiplied by the probability of satisfying all the constraints. Smith conjectures that Equation 3.2

$$\hat{\mathsf{p}}_{2crit} = 1 - \mathsf{m}^{\frac{-2}{(\mathsf{n}-1)\mathsf{p}_1}} \tag{3.2}$$

which is derived from Equation 3.1 by setting $\mathsf{E(Soln)} = 1$ can be used to predict the location of a phase transition peak for binary CSPs[5]. Smith uses the notation $\hat{\mathsf{p}}_{2crit}$ to emphasize that this is a predictor of the "critical point", that is, the crossover point.

Looking again at Figure 3.1, the value of $\hat{\mathsf{p}}_{2crit}$ for the values of n, m, and p_1 used to generate graph is 0.55 which is a reasonably good predictor of the 50% point of solubility (a little towards the insoluble side) and is within the mushy region. Smith[102], Smith & Dyer[103], and Prosser[91, 94] show empirically that $\hat{\mathsf{p}}_{2crit}$ is a reasonably good predictor of the location of a phase transition peak for *randomly generated binary CSPs*, especially as n grows. The predictor gives an over-estimation of the location of the phase transition peak, that is towards the right (insoluble) side. The one exception for Smith's predictor is for sparse graphs, that is, those CSPs with small values of p_1. Smith & Dyer[103] show that each individual constraint graph (given domain sizes for each variable) has its own location of the phase transition peak. The location of the phase transition peak is highly variable for constraint graphs which are sparse. Smith & Dyer argue that the *local graph topology* needs to be incorporated into any predictor of the location of the phase transition peak for these problems. The local graph topology is defined to be the *degree distribution* of the constraint graph, that is, the degree of each individual node (variable) in the constraint graph.

---

problems which don't have this "feature".

[5] As mentioned, graph colouring problems are a type of problem for which Smith's predictor is inappropriate as typically when a graph colouring problem is soluble, it has many solutions which are simple permutations of the colours.

The model that Smith[102], Smith & Dyer[103], and Prosser[91, 94] use to parameterize binary CSPs assumes that every domain has the same size and that every constraint has the same tightness. After [102, 91] were published, we decided to investigate a generalization of this model which allows for some variation in each constraint's tightness. Any generalization of Smith's model is useful as the new model will then be closer to a "real" CSP. The results of our investigation are in Chapter 5. It turns out that a predictor of a phase transition for the new model of binary CSPs includes local graph topology and better predicts the location of the phase transition peak. We believe that our new model provides the foundation for a new model that allows varying domain sizes but we leave that as future work.

Smith, Smith & Dyer, and Prosser do not show that the predictor $\hat{p}_{2crit}$ predicts the location of phase transition peaks for "real" problems. It is conjectured that if a real problem can be modeled using the given parameterization then Smith's predictor will predict the location of the phase transition peak. We also do not investigate the appropriateness of Smith's predictor or the predictor of our new model. Such a task is left for future work.

3.2.1 Statistical Measurements of the Average

There is some question as to which statistical measurement is most appropriate to calculate an "average" number of constraint checks performed to solve a sample of hard random problems. Problems in the phase transition peak do not appear to be drawn from a normal distribution[70][6]. Figure 3.2 shows the same graph as in Figure 3.1 together with similar plots using the average, the median, and the minimum (min). The maximum values, being about an order of magnitude higher, are shown in Figure 3.3. The phase transition phenomenon appears to happen for all the statistical measurements used. The average gives a taller phase transition peak than the median and the geometric mean. The geometric mean appears to be similar to the median in many cases, occasionally being a little higher (to the right of the phase transitions peak). In past studies using hard random problems the median or average has been

[6]This may be a symptom of not being able to take an adequately large sample size.

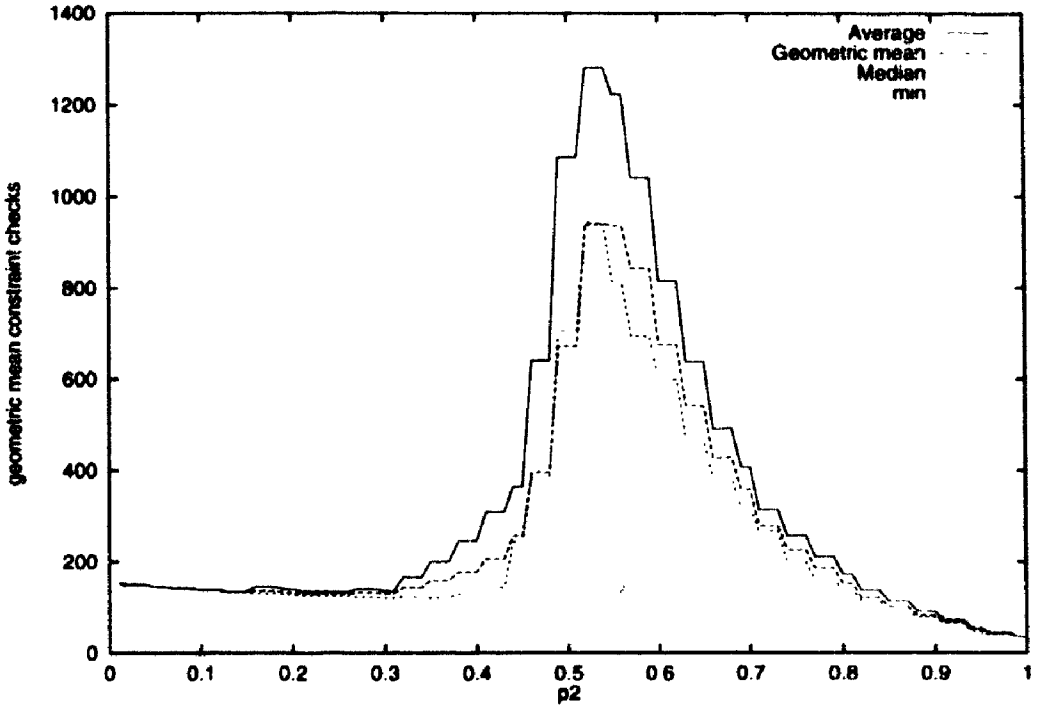

Figure 3.2: The average, the geometric mean, the median, and the minimum number of constraint checks performed solving each problem by the FC algorithm in the set of 50 problems randomly generated from $\langle 10, 6, 0.5 \rangle$ as p_2 is varied from 0.01 to 0.99 in increments of 0.01.

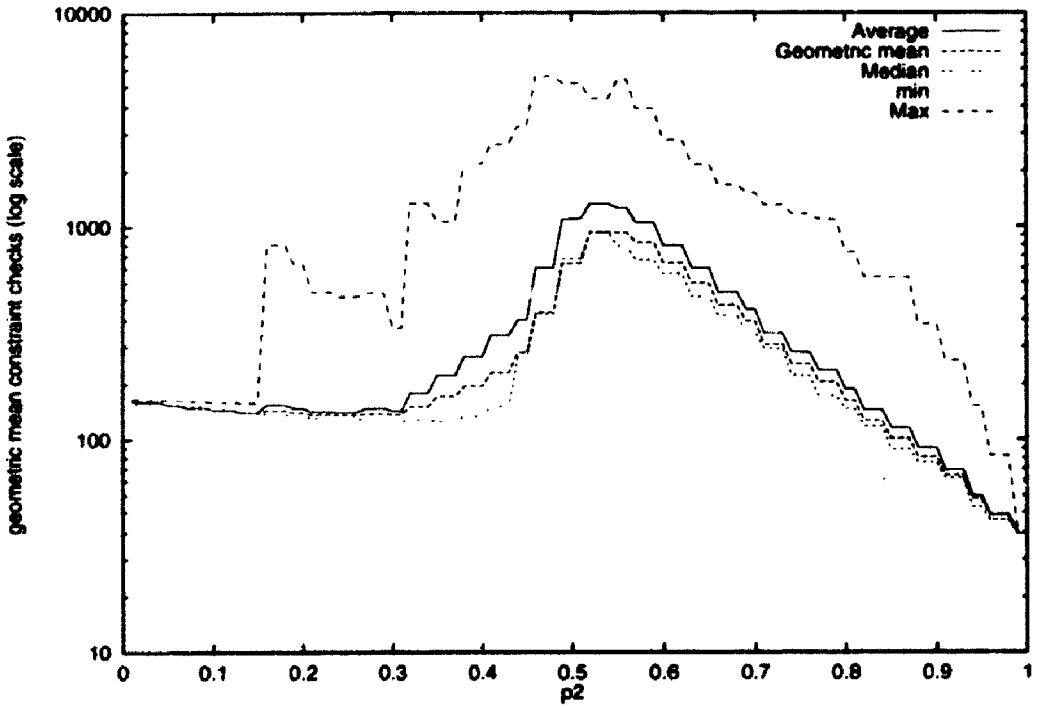

Figure 3.3: The average, the geometric mean, the median, the minimum and the maximum number of constraint checks performed solving each problem by the FC algorithm in the set of 50 problems randomly generated from $\langle 10, 6, 0.5 \rangle$ as p_2 is varied from 0.01 to 0.99 in increments of 0.01.

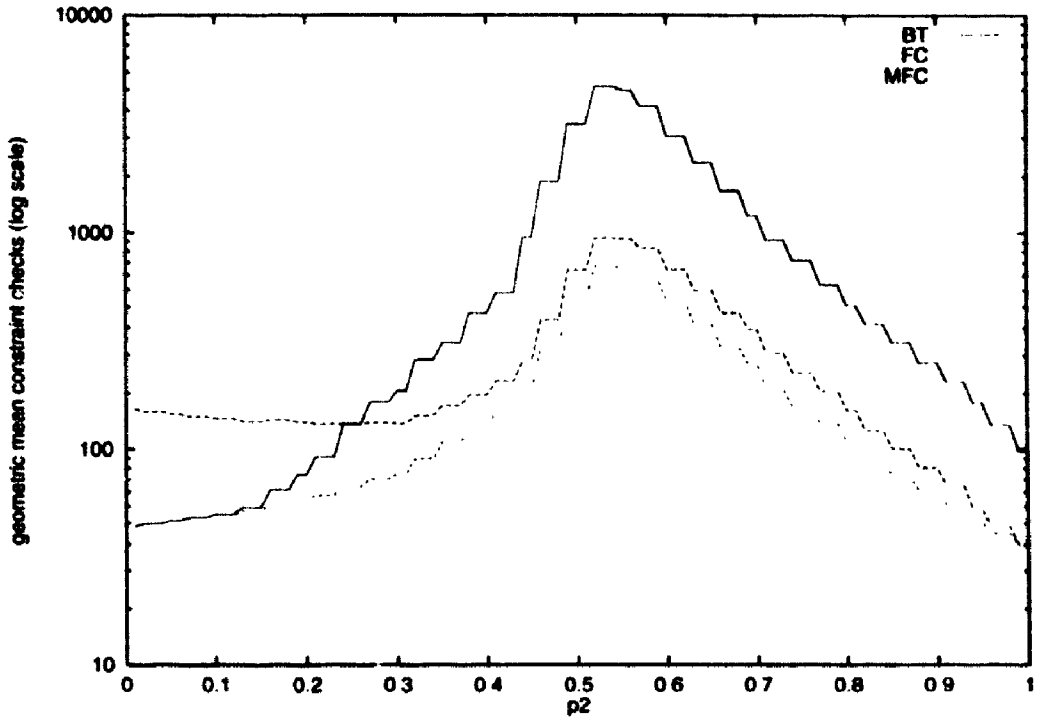

Figure 3.4: The geometric mean number of constraint checks performed solving each problem by the BT, FC, and MFC algorithms in the set of 50 problems randomly generated from $\langle 10, 6, 0.5 \rangle$ as p_2 is varied from 0.01 to 0.99 in increments of 0.01.

used[1, 2, 30, 34, 33, 43, 49, 71, 91, 93, 94, 97, 102, 103, 109]. However, we have found through experience that the geometric mean[7] appears to give the fairest measurement of how hard a set of problems really is "on average". The average gives too much weight to outliers and the median completely ignores them. The geometric mean appears to be a fair compromise between the two and is what we use predominantly throughout this thesis. (We do give the other measurements for comparison in tables). Recent work by Kwan[70] assumes that the distribution of problems in the transition peak is not normal and begins an investigation of the use of non-parametric statistical techniques.

[7]The *geometric mean* $\tilde{x}_g$ of a set of positive numbers $\{x_1, x_2, \ldots, x_n\}$ is $(x_1 x_2 \ldots x_n)^{\frac{1}{n}}$.

3.2.2 Using Different Algorithms to Map the Phase Transition

Finally, we look at how the shape of the phase transition changes when using the BT, FC, and MFC algorithms. Figure 3.4 displays a comparison of BT, FC, and MFC over the random problems generated for the previous figures. Although it is conjectured that the phase transition phenomenon is independent of the algorithm used[103], there is a clear difference in the performance of each algorithm. The MFC algorithm is clearly superior to FC, and the two forward looking algorithms, MFC and FC, perform about 5 times better than BT on the hardest problems. Interestingly, BT as well as MFC performs better than FC on the easiest problems to solve, that is, on the problems with the loosest constraints. For these problems the non-lazy forward checking of FC is wasted as the probability of finding future inconsistencies is low. MFC looks forward only enough to know that there are consistent values ahead and therefore its performance is quite similar to that of BT on these easy problems.

In this thesis we investigate the performance of the search algorithms on hard random problems only. We do not have the computational resources necessary to investigate the performance of MFC ac ss all constraint tightnesses on a wide enough range of problems. Tsang *et al*[109] have begun a mapping project in which they are attempting to find which algorithm/heuristic combination performs best for different classes of random CSPs. Their eventual goal is to have a map which can be used to select an appropriate algorithm and heuristic given a real problem's structure. This idea can be extended further to using different algorithms after every instantiation as the future domains are effectively a new CSP to solve.

Although we have used hard random problems as ou: testbed to compare algorithms in this thesis, we make no claim that these results can be carried over to all types of real problems. Real problems that arise in practice may not fit the parameterization used. That is all domain sizes equal, randomly generated edges, and randomly generated constraints. However, our comparisons do show that there are clear differences between the algorithms in their ability to solve relatively hard CSP problems over a wide range of problems.

3.2.3 Previous Comparisons Using Hard Random CSPs

Recently, a number of empirical comparisons[1, 2, 30, 34, 33, 43, 49, 71, 93, 97, 109] have been performed on CSP search algorithms using hard random problems. We briefly describe only those that are directly related to this thesis.

Bacchus and Van Run[2, 115] compare 12 algorithms, including a number of hybrid algorithms using BJ or CBJ, with and without FF (24 combinations in total) on a sample of 500 hard problems (a total of 5 settings for n, m and p_1 with 100 samples drawn from each setting). Their results show that FC-CBJ-FF is the best algorithm. However, when they published [2], they had also tested MFC-FF with a heuristic[8] similar to the one we describe[9] in Chapter 4 and found it to be the best algorithm of the 24 tested. Although no results were given in [2] they state that MFC-FF with their heuristic is better than all 24 algorithms.

Sabin and Freuder[97] have compared MAC (RFL) and FC-FF using a small sample of hard random problems consisting of only 20 problems at 4 settings of n, m and p_1. Their results suggest that MAC-FF may be better than FC-FF for some classes of problems, notably CSPs with sparse graphs and a large number of variables ($n = 50$ in their case).

Finally, we have reported some preliminary results of this thesis in [30, 34, 33].

3.3 An Empirical Comparison

To begin our comparison we first produce a large testbed of 50 hard random problems for each possible setting of $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$, $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$ and p_2 calculated according to Equation 3.2 giving a total of 10,200 problems. These particular settings are chosen to give as wide a range of problems as possible. We are interested in the performance of the search algorithms over all graph densities and as n and m increase. Larger values of n, m and the sample size (50) are not feasible given our limited computational resources.

[8] The INC-FF heuristic discussed in Chapter 4.

[9] The EXP-FF heuristic discussed in Chapter 4.

One way to create random CSPs treats p_1 and p_2 as probabilities (similar to Palmer's Model A for constructing random graphs[86]). However, we treat p_1 and p_2 as percentages for two reasons[10]. The first is that we want our results to be able to predict how a search algorithm will perform on a specific problem instance which has an observable number of edges and number of inconsistent pairs in its constraints. The second reason is that phase transition peaks are more well defined (that is, taller) when p_1 and p_2 are treated as percentages. Using Palmer's Model B[86] we construct a constraint graph by randomizing an enumeration of all possible edges and taking the first $p_1 n(n-1)/2$ as edges. Unconnected graphs are rejected as the disconnected subgraphs can be solved separately and are therefore not representative of a problem with n variables.

Given the underlying constraint graph, we generate a random CSP as follows. For each edge in the random graph a non-trivial constraint is produced by randomizing the crossproduct of the domains and taking the first $\hat{p}_{2crit} m^2$ as unacceptable pairs.

To perform a comparison we run each algorithm on each problem instance counting the number of constraint checks performed. (*cf.* Section 1.2 for a discussion why constraint checks are used.) All algorithms begin their search with the first variable in the ordering given by the CSP definition. That is, we do not use any preliminary re-ordering of the variables. Domains are ordered to prevent any possible bias caused by different domain orderings. And, the order of the future variables are memorized and reset at every instantiation/uninstantiation in order to prevent bias from the algorithm's method of choosing the smallest domain ahead[11].

3.3.1 A Comparison of BT, BM, FC, and MFC

Initially, we attempted to perform a comparison of BT, BM, FC, MFC, FC-FF and MFC-FF on the whole testbed. However, the runs of BT and BM were stopped part

[10]Smith[102], Smith & Dyer[103] and Prosser[91, 94] also treat p_1 and p_2 as percentages.

[11]This methodology is significant only for the algorithms that include CBJ. As we use the same random problems to test algorithms that include CBJ in Chapter 4 we mention this point here and will explain further in the next chapter.

BETTER THAN	BT	BM	FC	MFC
BT		0.0 (0.4)	1.2 (0.1)	0.1 (0.4)
BM	99.6		29.1 (0 4)	5.2 (0.8)
FC	98.8	70.5		0.0 (0.0)
MFC	99.5	93.9	100.0	

Table 3.1: Percentage of times one algorithm performs better than another by number of constraint checks, for problems with $n \in \{10, 15\}$, $m \in \{3, 6, 9\}$, $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. Percentage of times that the algorithms perform the same number of constraint checks are in brackets. (5, 100 problems).

way through the random problems with $n = 20$ and $m = 6$ as it became infeasible to continue. This subsection gives an initial comparison of BT, BM, FC, and MFC on the random problems in $n \in \{10, 15\}, m \in \{3, 6, 9\}, p_1 \in \{0.2, 0.25, \ldots, 1.0\}$ with the primary focus being on BT and BM. We compare FC, MFC, FC-FF and MFC-FF on the whole testbed in Section 3.3.2. Due to the cost of running the BT and BM algorithm they are not used in any further comparisons.

We begin the comparison by displaying in Table 3.1 the percentage of times one algorithm was better than another in terms of constraint checks over the 5, 100 random problems generated. The numbers in brackets indicate the percentage of problems for which both algorithms performed the same number of constraint checks. The results in Table 3.1 indicate that the ordering of algorithms from best to worst is MFC > FC > BM > BT with BT performing much worse than the other algorithms. The BM algorithm performs better or the same as BT on all problems as expected by theory, however it performs better than FC in only 29.1% of the cases and better than MFC

Alg	Ave	Std Dev	Med	Geo Mean	Max	Min	% BT
BT	252919	1270013	10622	13375	42609438	24	100.0
BM	26302	83684	2364	3051	1844898	15	22.8
FC	12958	30077	1760	2124	422693	15	15.9
MFC	8833	20105	1317	1574	286738	12	11.8

Table 3.2: Average, standard deviation, median, geometric mean, maximum and minimum number of constraint checks performed by the BT, BM, FC, and MFC algorithms for problems with $n \in \{10, 15\}$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. (5,100 problems).

in only 5.2% of the cases. FC performs much better than BT and BM, and MFC performs better than FC on 100% of the problems (as expected given Theorem 2.8).

We next examine the data in a number of ways in order to find how much better one algorithm is than another and to see if the performance of the algorithms is consistent over different graph densities. We begin by displaying in Table 3.2 an overall picture of the "average" number of constraint checks performed. Table 3.2 gives the average (Ave) with standard deviation (Std Dev), the median (Med) and the geometric mean (Geo Mean). The maximum (Max) and minimum (Min) number of constraint checks for each problem are also displayed. In the last column is the percentage of the number of BT's geometric mean number of constraint checks each algorithm performed. Again the same ordering is observed between the algorithms across all averages as well as by the standard deviation, the maximum and the minimum. The FC algorithm performs much better than BM which performs much better than BT. The MFC algorithm is the best performer with only 11.8% of the number of constraint checks of BT and only 74.1% of the number of constraint checks of FC.

We next look at how the algorithms perform over the values of p_1. Figure 3.5 shows a comparison of BT, BM, FC, and MFC by geometric mean number of constraint checks varied by p_1 for problems with $n = 15$, $m = 3$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$.

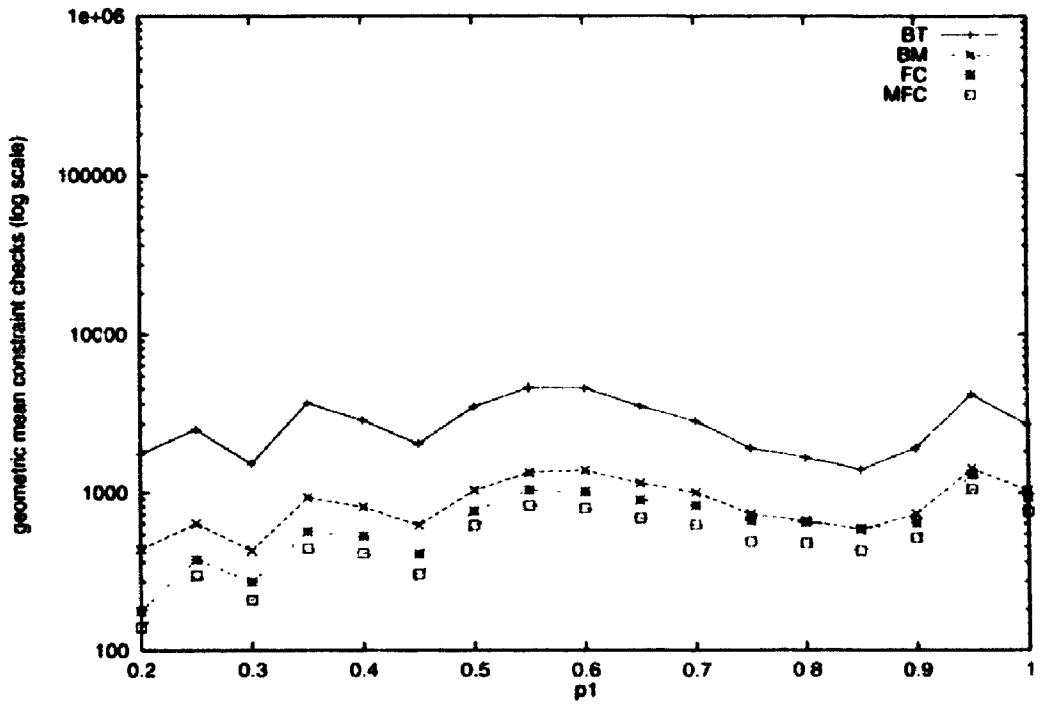

Figure 3.5: Comparison of BT, BM, FC, and MFC by geometric mean number of constraint checks varied by p_1 for problems with $n = 15$, $m = 3$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$.

Figure 3.6 shows a similar comparison except that $m = 9$. Figures 3.5 and 3.6 show the same ordering for the algorithms across all the values of p_1. There is a significant difference in performance between BM and BT which grows as m increases. Interestingly, BM appears to perform nearly as well as FC for small values of m with denser constraint graphs ($p_1 \geq 0.75$). However, looking at Figure 3.6 we see that the relative performance of BM degrades as m grows larger. We can also see that MFC performs better than all the other algorithms and it appears that it performs uniformly better than FC for all values of p_1 with the relative performance difference growing as m increases.

Finally, we look at the raw data itself to compare the algorithms. We display scatter plots for BT versus BM in Figure 3.7 and for BM versus FC in Figure 3.8. We do not compare MFC with FC here, leaving that comparison for Section 3.3.2. Figure 3.7

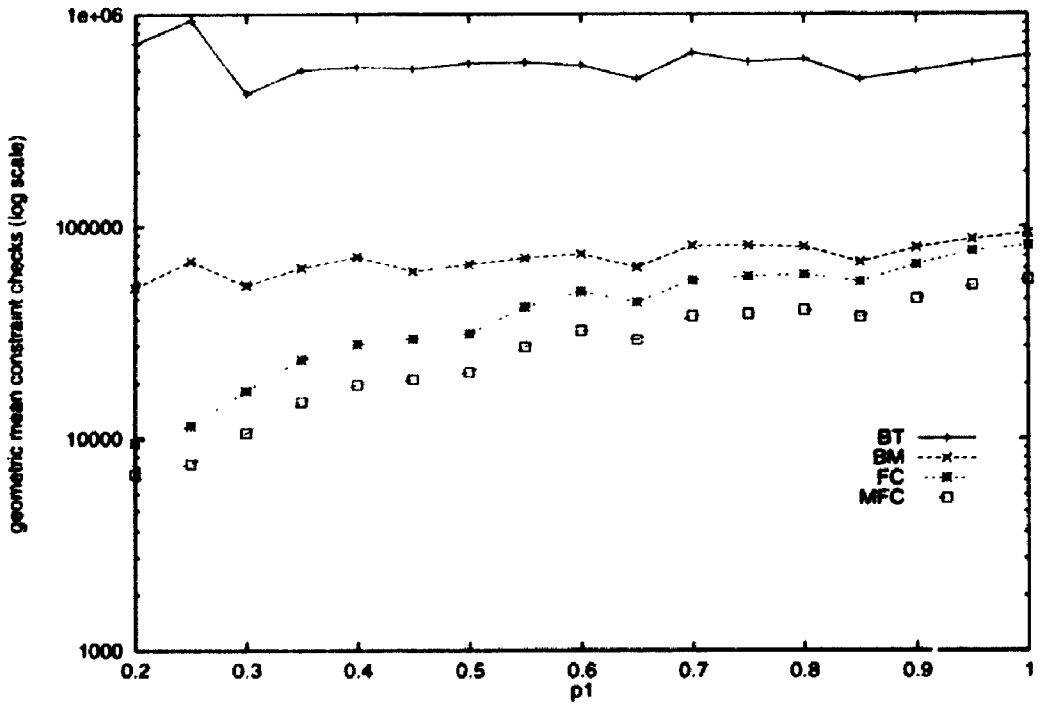

Figure 3.6: Comparison of BT, BM, FC, and MFC by geometric mean number of constraint checks varied by p_1 for problems with $n = 15$, $m = 9$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$.

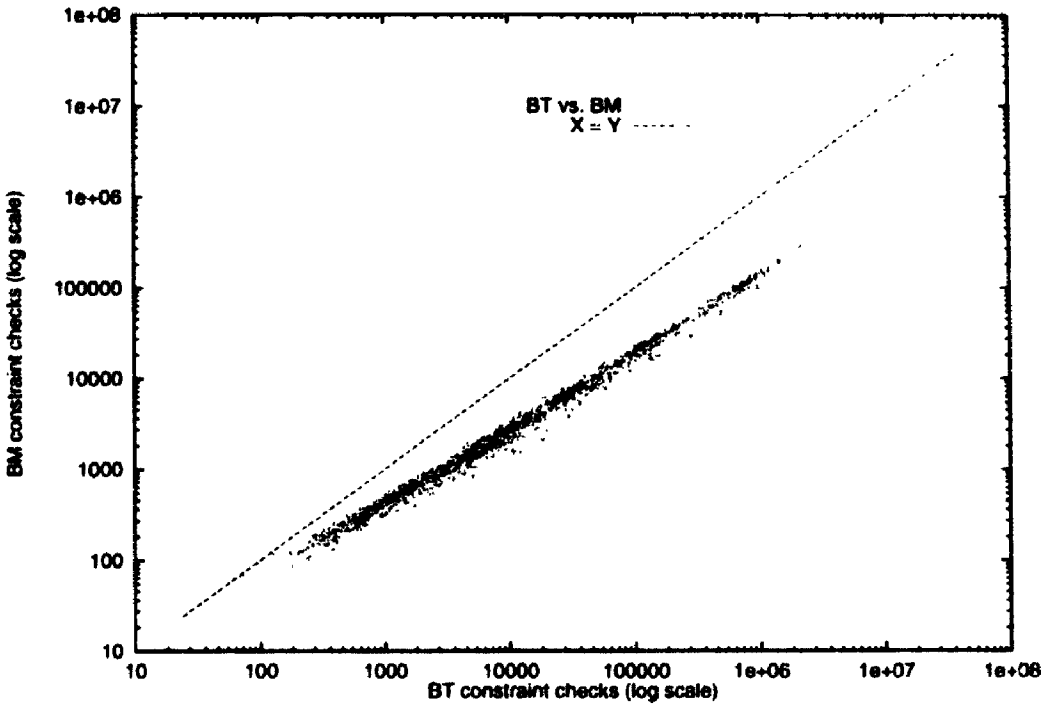

Figure 3.7: The number of constraint checks performed by BT versus the number of constraint checks performed by BM on the same problem for problems with $n \in \{10, 15\}$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. (5,100 problems)

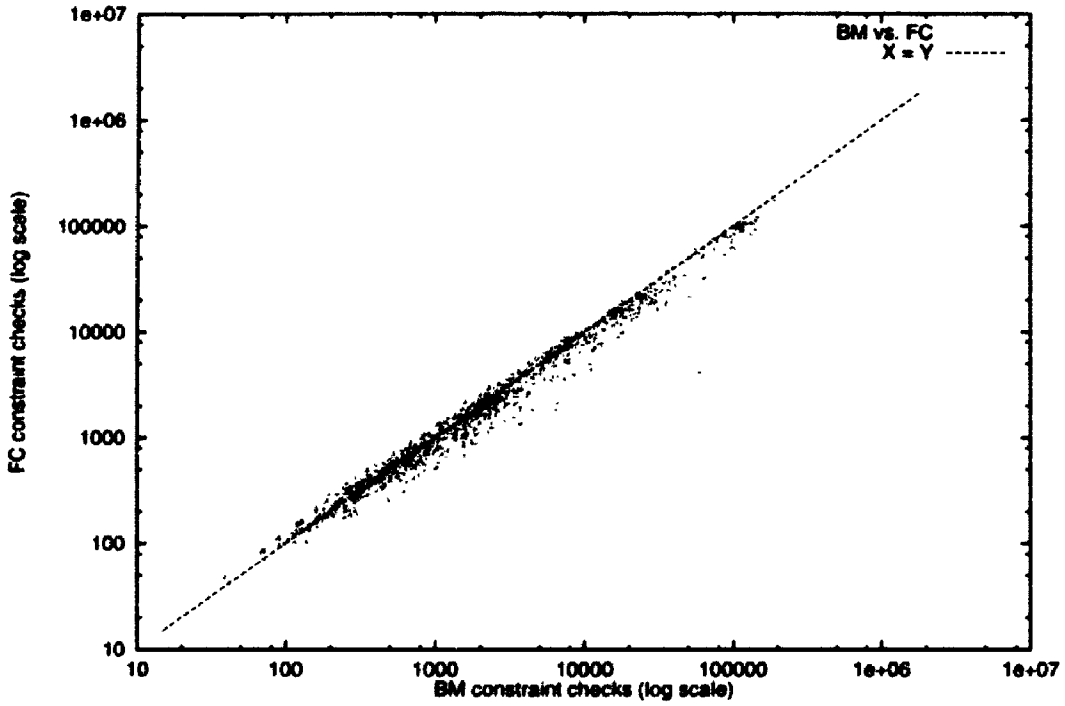

Figure 3.8: The number of constraint checks performed by BM versus the number of constraint checks performed by FC on the same problem for problems with $n \in \{10, 15\}$, $m \in \{3.6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. (5, 100 problems)

shows that BM is clearly superior to BT. BM performs fewer constraint checks than BT everywhere (as expected) and BM becomes increasingly better as the difficulty of the problem increases (that is, moving to the top right of the graph). Figure 3.8 shows that FC is better than BM for most of the problems. From Table 3.1 we know that BM performs better than FC for 29.1% of the problems. However, Figure 3.8 shows that most of those problems are solved by both BM and FC with fewer than 10,000 constraint checks and that the performance of BM on those problems is close to that of FC. What is interesting to note is that there appears to be many problems for which FC performs much better than BM (lower right corner of Figure 3.8). On these problems the forward looking property of FC is critical to avoiding poor performance. We can also conclude from Figure 3.8 that we were not able to run BM on the whole testbed because of the existence of these problems which are really hard for BM.

In this subsection we have compared BT, BM, FC, and MFC along many different dimensions. By far the worst algorithm in the comparison is BT. We find that BM is clearly superior to BT and in some relatively easy cases can be better than FC. However, BM can perform very badly on many problems as it does not have the ability to look ahead and avoid future inconsistencies. FC is clearly a superior algorithm performing better or nearly the same as BM on many problems. Finally, we have found that MFC performs uniformly better than FC (performing approximately 74% of the constraint checks performed by FC) across all values of p_1 and it appears that it performs much better than FC as m increases. The ordering of algorithms from best to worst, according to the empirical results in this subsection, is MFC > FC > BM > BT. We save a detailed comparison of FC and MFC until the next subsection where the experiments are performed over the whole testbed.

For the record, a total of 20,400 problems are used in the comparisons reported. The total CPU time used for these comparisons on a Sun Sparc 10 and a Sun Sparc 5 is 381,137 CPU-seconds. A total of 1,535,166,189 constraint checks were performed. We have implemented all programs in this thesis in CMU-Lisp.

BETTER THAN	FC	MFC	FC-FF	MFC-FF
FC		0.0 (0.0)	2.5 (0.5)	1.2 (0.0)
MFC	100.0		7.5 (0.1)	3.9 (0.5)
FC-FF	97.0	92.4		42.8 (0.2)
MFC-FF	98.8	95.6	57.0	

Table 3.3: Percentage of times one algorithm performs better than another by number of constraint checks, for problems with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$, $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. Percentage of times that the algorithms perform the same number of constraint checks are in brackets. (10, 200 problems).

3.3.2 A Comparison of FC, MFC, FC-FF, and MFC-FF

In the last section we compared BT, BM, FC and MFC on problems in $n \in \{10, 15\}$, $m \in \{3, 6, 9\}$, and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. In this section we compare FC, MFC, FC-FF, and MFC-FF on a wider range of problems, namely those with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$, and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. The BT and BM algorithms are no longer included in our comparison due to their inefficiency.

We begin our comparison by displaying in Table 3.3 the percentage of times one algorithm is better than another in terms of the number of constraint checks over the 10, 200 random problems. The numbers in brackets indicate the percentage of problems for which both algorithms performed the same number of constraint checks. The results in Table 3.3 indicate that the ordering of algorithms from best to worst is MFC-FF > FC-FF > MFC > FC. Of the algorithms not using FF, MFC performs better than FC for all of the problems in the testbed (as expected by theory). Of the

Alg	Ave	Std Dev	Med	Geo Mean	Max	Min	% FC
FC	710168	2637242	8626	13539	58051198	15	100.0
MFC	433021	1603276	6170	9483	35677917	12	70.0
FC-FF	33450	109181	1872	2538	1254426	15	18.7
MFC-FF	35608	117645	1768	2439	1334088	12	18.0

Table 3.4: Average, standard deviation, median, geometric mean, maximum and minimum number of constraint checks performed by the FC, MFC, FC-FF, and MFC-FF algorithms for problems with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. (10,200 problems).

algorithms using FF, MFC-FF is slightly better than FC-FF but the result is not as conclusive.

We next examine the data in a number of ways to determine how much better one algorithm is than another and to see if the performance of the algorithms is consistent over different graph densities. We begin by displaying in Table 3.4 an overall picture of the "average" number of constraint checks performed. Table 3.4 shows that MFC performs roughly 70% of the number of constraint checks that FC performs. By the geometric mean and median number of constraint checks the same ordering as above is again observed. However, by the average and its standard deviation, and by the maximum number of constraint checks, FC-FF is better than MFC-FF. In Chapter 2, Theorem 2.9 indicates that the MFC-FF algorithm may occasionally perform worse than FC-FF as it does not know the true domain size of the future domains. In [30] a smaller comparison of MFC-FF and FC-FF using a different set of hard random problems found that MFC-FF could occasionally do much worse than FC-FF. Similar findings are reported in [1, 71].

We next look at how the algorithms perform over the values of p_1 tested. Figures 3.9 to 3.12 show comparisons of FC and MFC by geometric mean number of constraint checks varied by p_1 for problems with $n = 10$, $n = 15$, $n = 20$, and $n = 25$

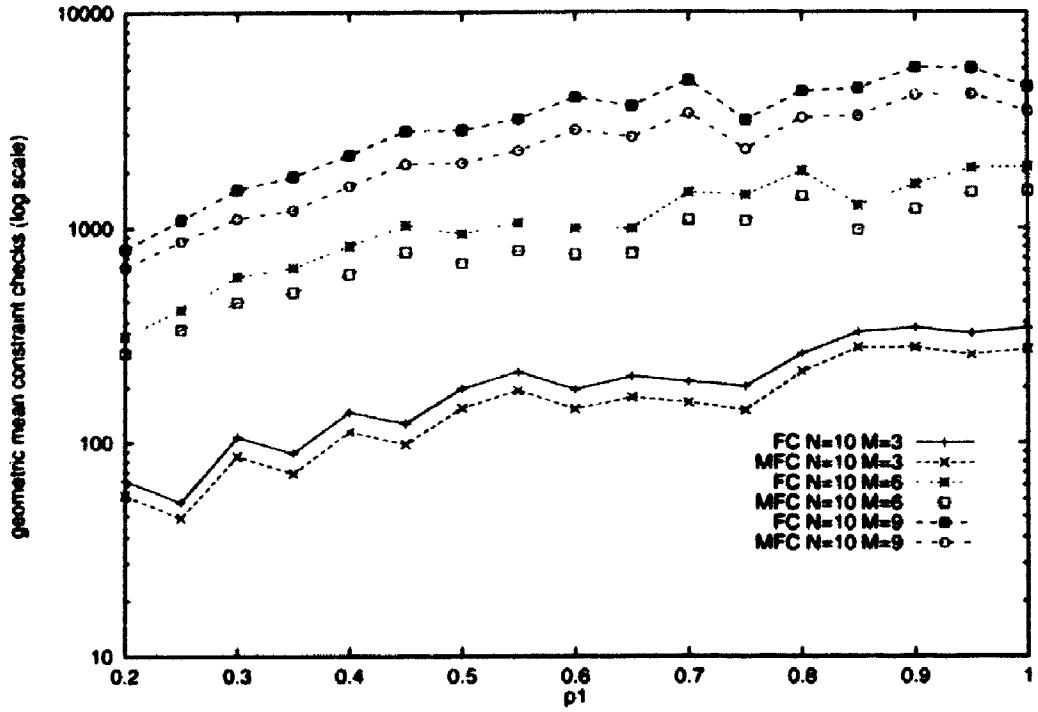

Figure 3.9: Comparison of FC and MFC by geometric mean number of constraint checks varied by p_1 for problems with $n = 10$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$.

with $m \in \{3, 6, 9\}$. It is evident that MFC performs uniformly better than FC over all values of p_1 and it appears that the separation between the graphs is growing larger as n increases and as m increases.

Next, we compare FC-FF and MFC-FF over the values of p_1 tested. Figures 3.13 to 3.16 show comparisons of FC-FF and MFC-FF by geometric mean number of constraint checks varied by p_1 for problems with $n = 10$, $n = 15$, $n = 20$, $n = 25$ with $m \in \{3, 6, 9\}$. For the smallest value of n, MFC-FF appears to be better than FC-FF. However, for $n \geq 15$ the performance of FC-FF and MFC-FF are very close with FC-FF becoming slightly better than MFC-FF for $n = 25$, $m = 9$. The closeness of the graphs may indicate that MFC-FF is fairly consistent in making poor instantiation choices across all values of p_1 especially for larger values of n and m.

We next look at the raw data itself to compare the algorithms. We display scatter

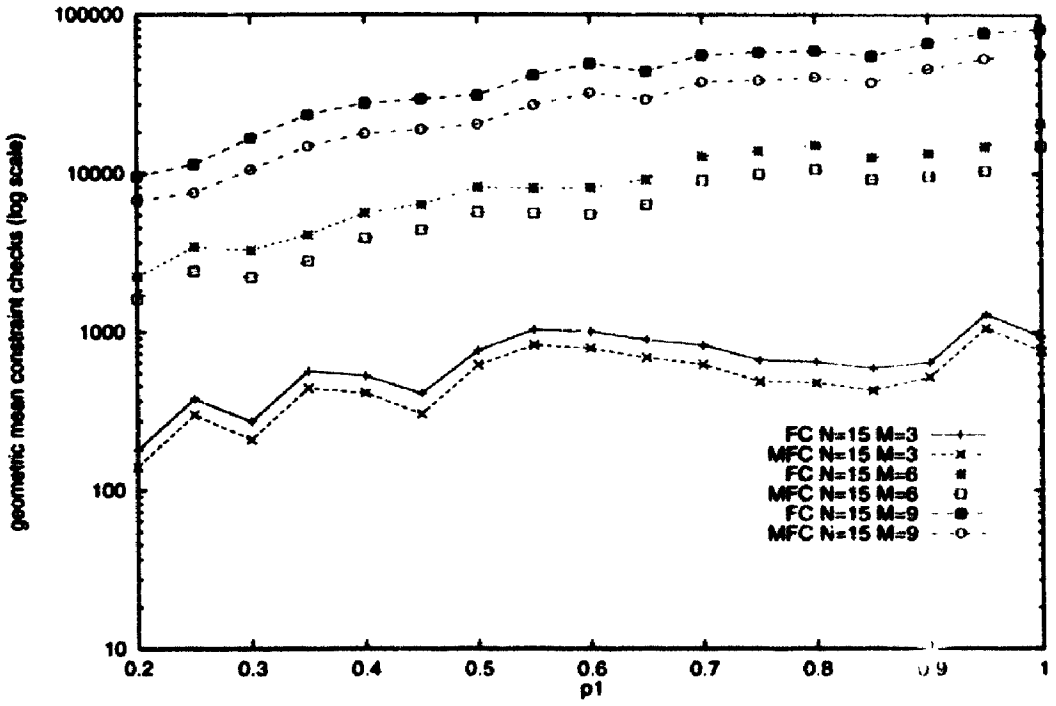

Figure 3.10: Comparison of FC and MFC by geometric mean number of constraint checks varied by p_1 for problems with $n = 15$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$.

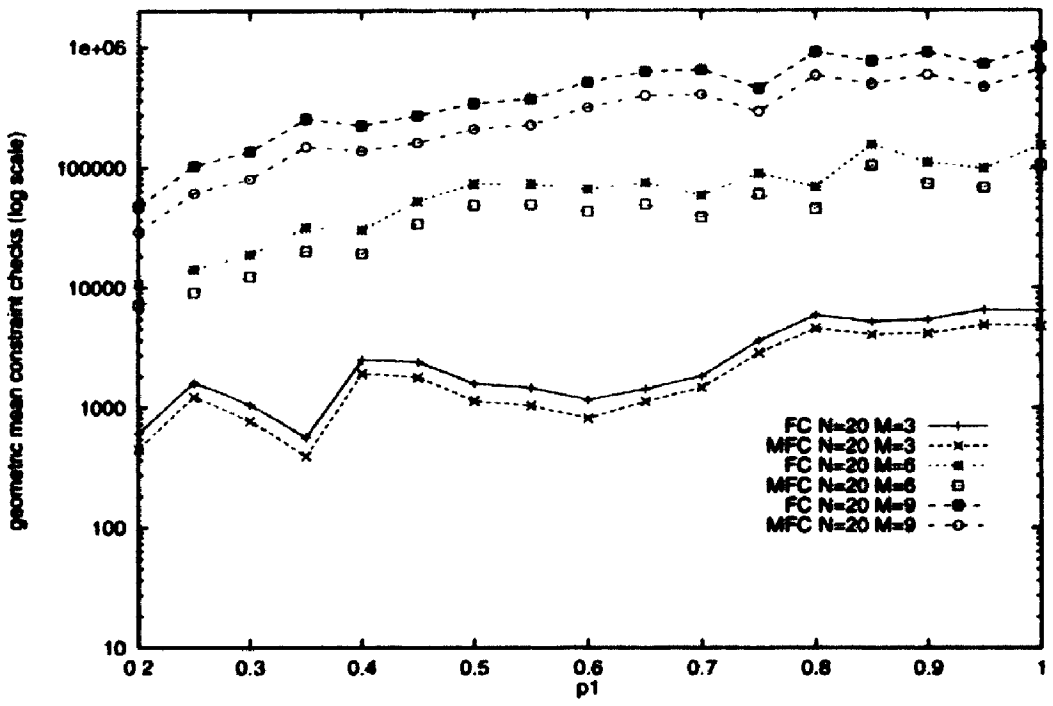

Figure 3.11: Comparison of FC and MFC by geometric mean number of constraint checks varied by p_1 for problems with $n = 20$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$.

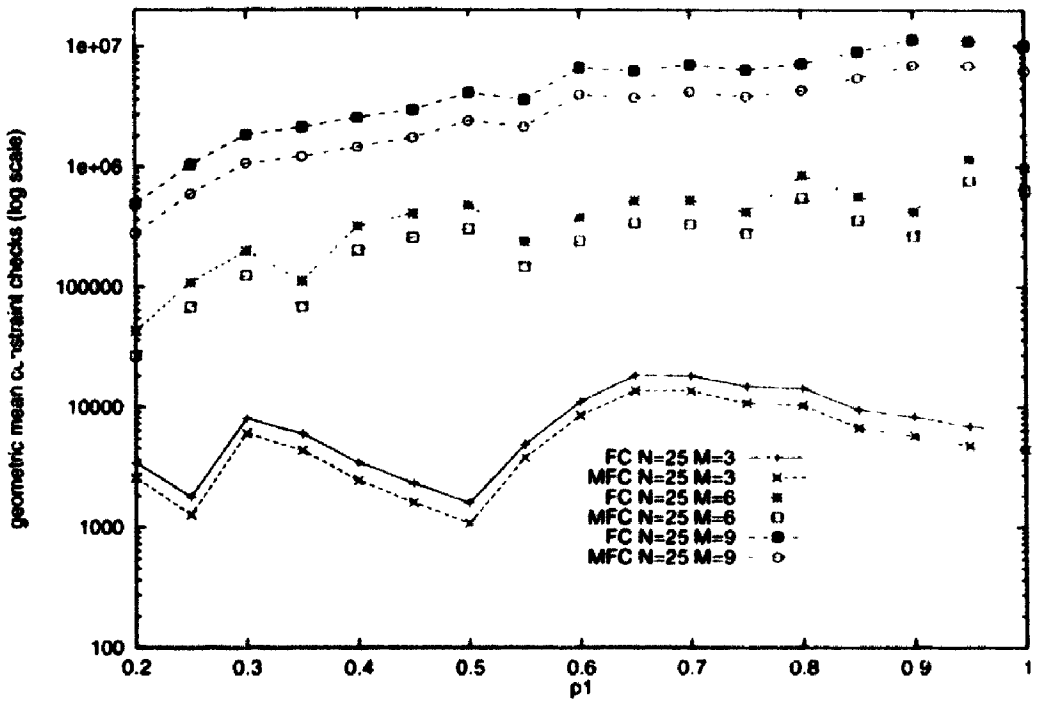

Figure 3.12: Comparison of FC and MFC by geometric mean number of constraint checks varied by p_1 for problems with $n = 25$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$.

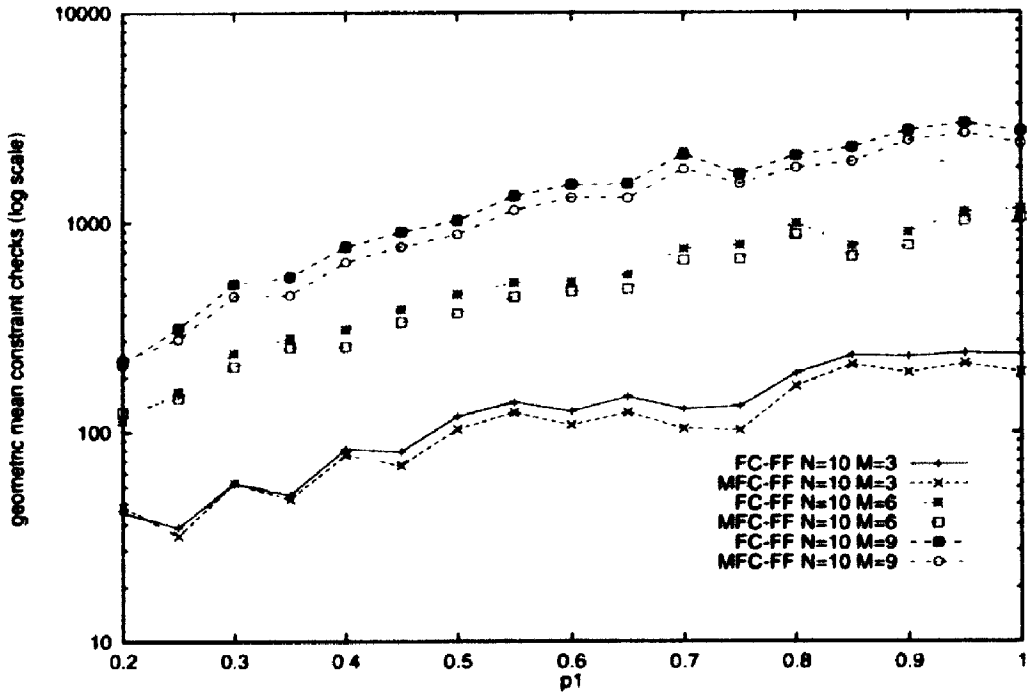

Figure 3.13: Comparison of FC-FF and MFC-FF by geometric mean number of constraint checks varied by p_1 for problems with $n = 10$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$.

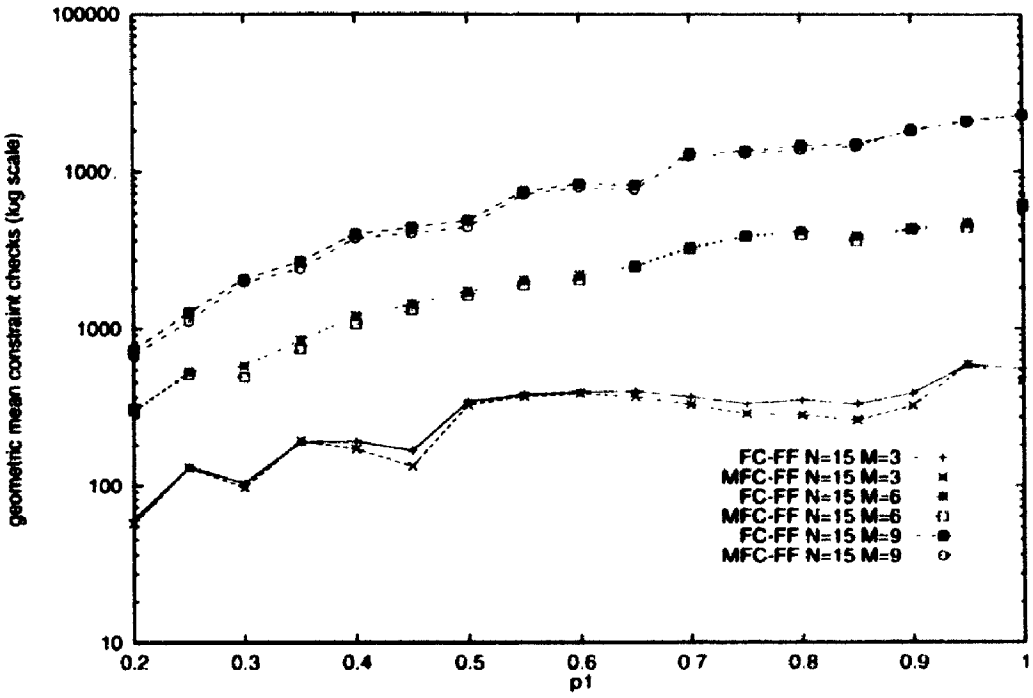

Figure 3.14: Comparison of FC-FF and MFC-FF by geometric mean number of constraint checks varied by p_1 for problems with $n = 15$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$.

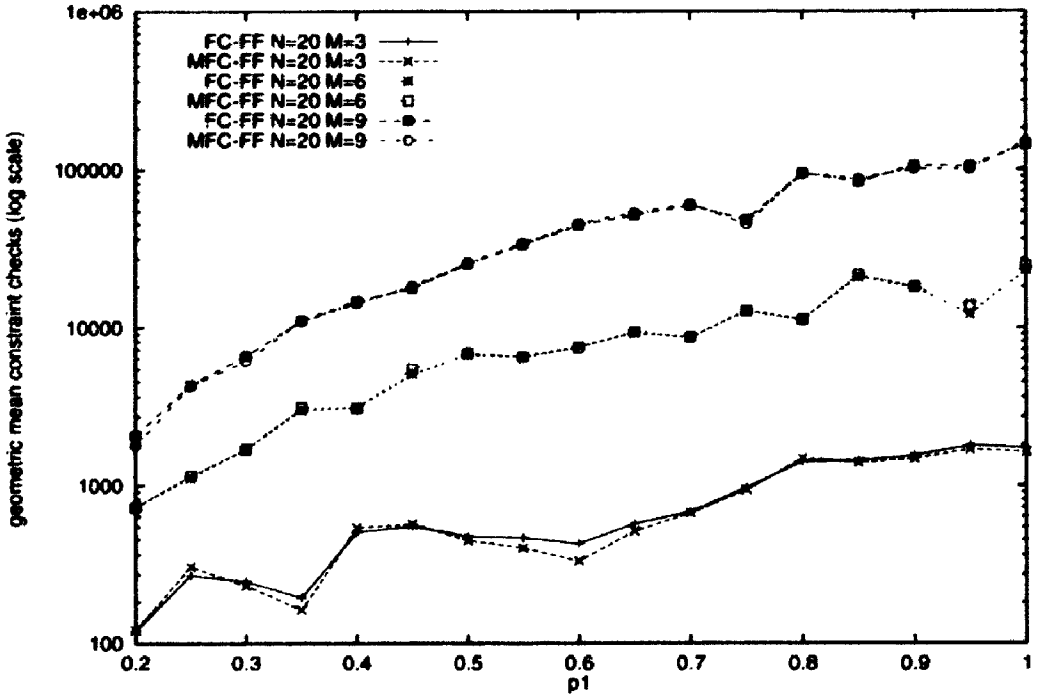

Figure 3.15: Comparison of FC-FF and MFC-FF by geometric mean number of constraint checks varied by p_1 for problems with $n = 20$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$.

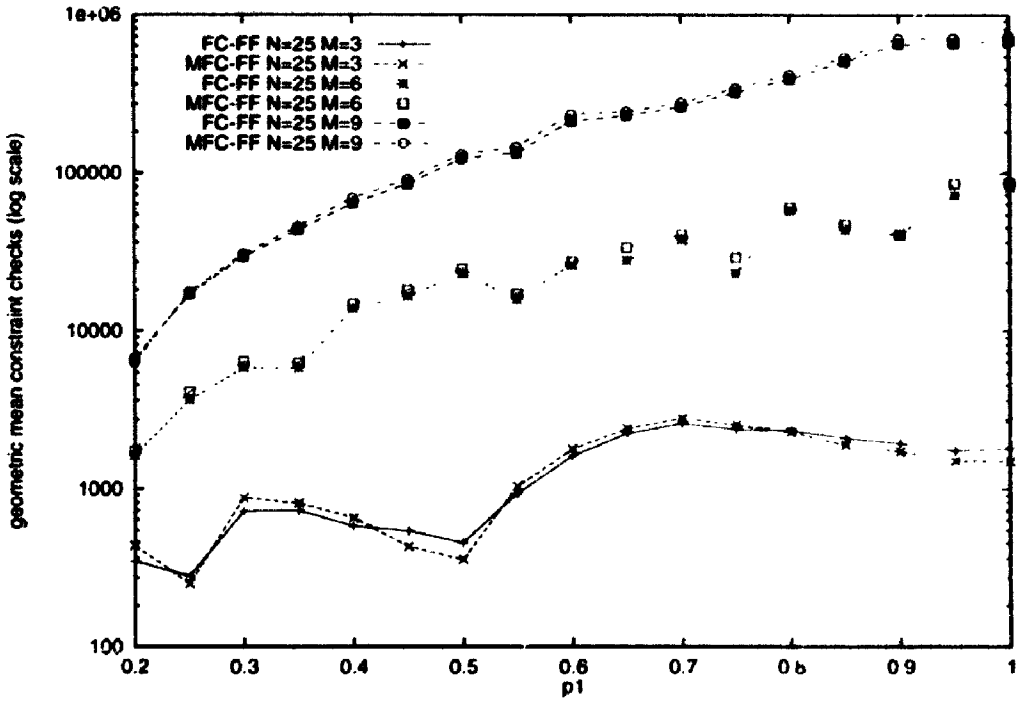

Figure 3.16: Comparison of FC-FF and MFC-FF by geometric mean number of constraint checks varied by p_1 for problems with $n = 25$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$.

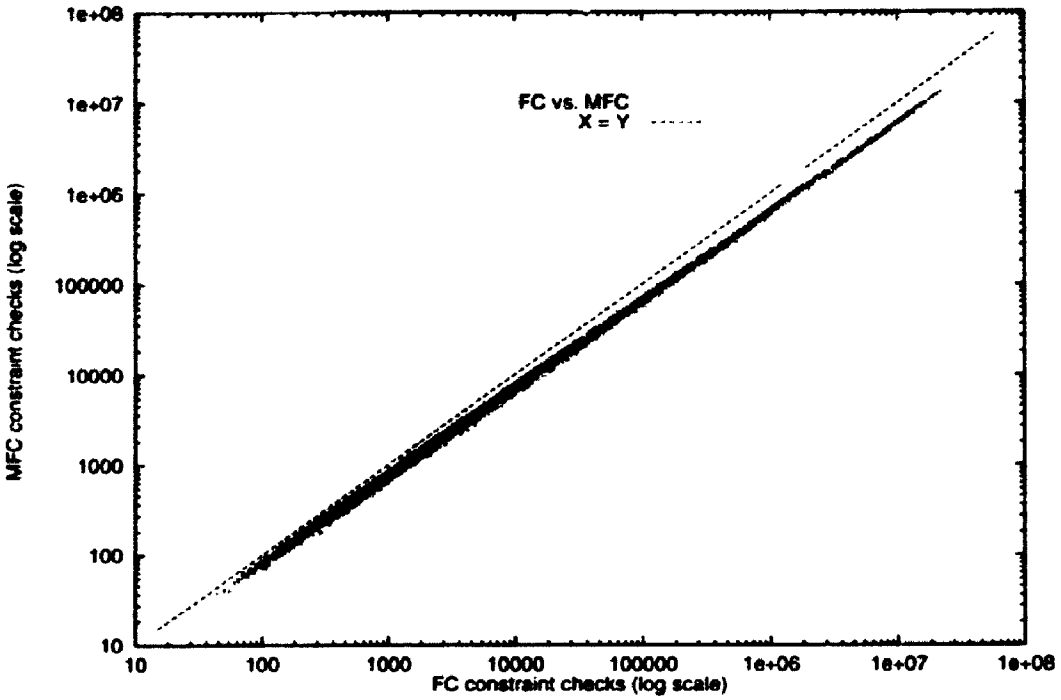

Figure 3.17: The number of constraint checks performed by FC versus the number of constraint checks performed by MFC on the same problem for problems with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. (10,200 problems)

plots for FC versus MFC in Figure 3.17, for FC-FF versus MFC-FF in Figure 3.18, for FC versus FC-FF in Figure 3.19, and for MFC versus MFC-FF in Figure 3.20.

Figure 3.17 shows that MFC's performance relative to FC becomes increasingly better as the difficulty of the problem increases. In Figure 3.18 it appears, roughly, that FC-FF and MFC-FF perform worse than the other on about half of the problems. However, on the hardest problems at the top right of the graph, MFC-FF performs worse than FC-FF. Figures 3.19 and 3.20 are quite similar. The FF versions of the algorithms are much better especially for the harder problems. However, the relative difference between FC and FC-FF appears to be greater than the relative difference between MFC and MFC-FF.

Finally, we investigate the trend in relative performance for MFC, FC-FF and

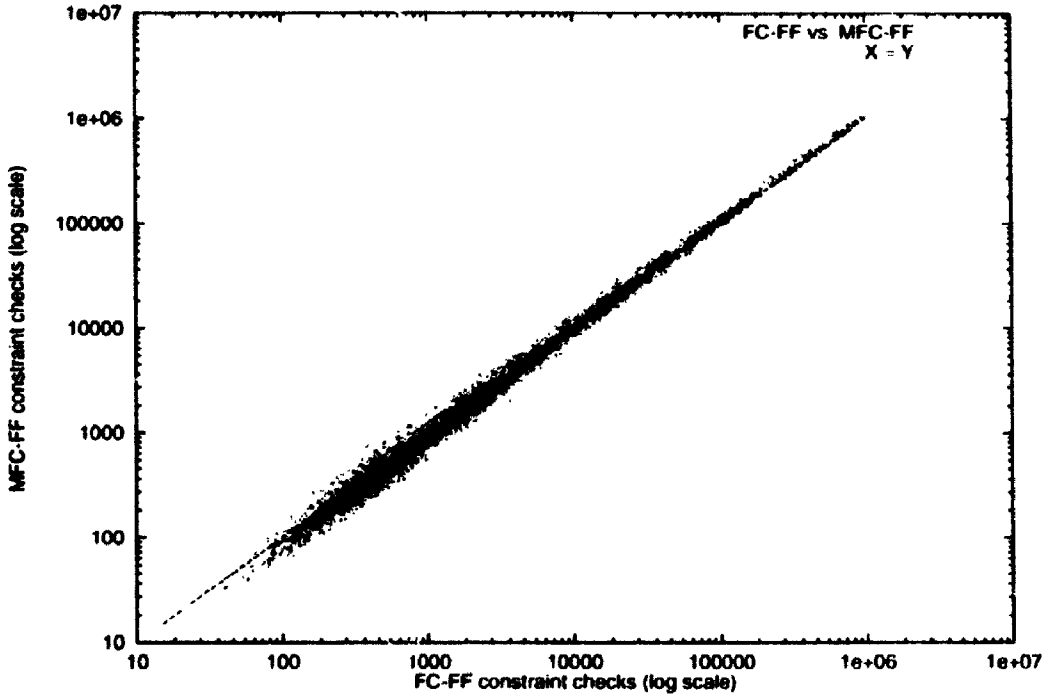

Figure 3.18: The number of constraint checks performed by FC-FF versus the number of constraint checks performed by MFC-FF on the same problem for problems with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. (10,200 problems).

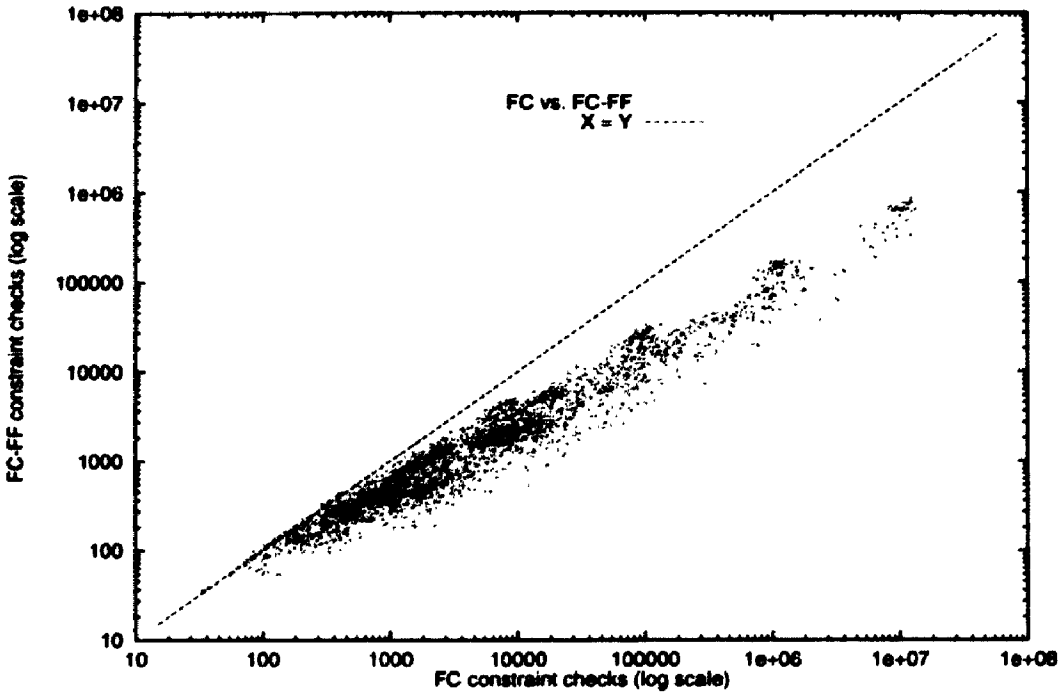

Figure 3.19: The number of constraint checks performed by FC versus the number of constraint checks performed by FC-FF on the same problem for problems with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. (10,200 problems).

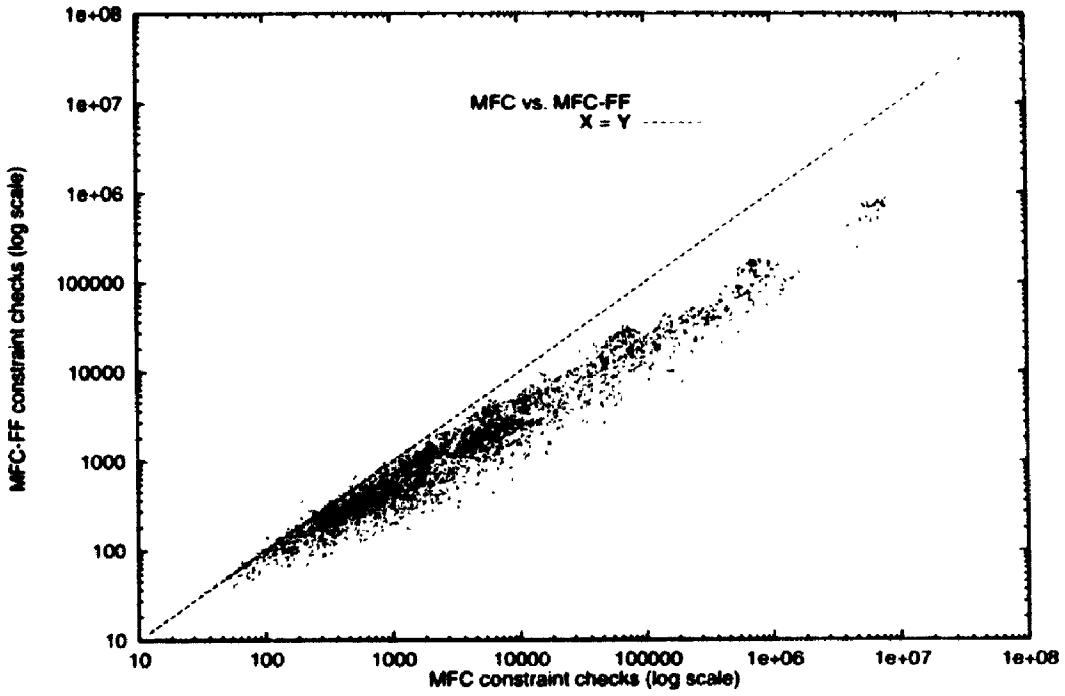

Figure 3.20: The number of constraint checks performed by MFC versus the number of constraint checks performed by MFC-FF on the same problem for problems with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. (10,200 problems).

n	10			15			20			25		
m	3	6	9	3	6	9	3	6	9	3	6	9
MFC	82.1	76.6	73.0	78.9	70.6	66.9	76.4	67.0	62.9	73.9	64.4	60.3
FC FF	63.7	45.9	38.2	40.3	23.6	18.4	22.5	11.5	8.6	13.1	5.5	4.0
MFC FF	57.1	41.6	34.1	37.6	22.9	17.9	22.5	11.8	8.8	13.6	6.0	4.3

Table 3.5: Percentage of FC's constraint checks performed by MFC, FC-FF and MFC-FF broken down by n and m.

MFC-FF as n and m increase. Table 3.5 displays the percentage of FC's constraint checks performed by the algorithms broken down by n and m. Table 3.5 verifies our previous observation. MFC performs increasingly fewer constraint checks relative to FC as n increases and as m increases. FC-FF and MFC-FF both become increasingly better than FC as n increases and as m increases. However, although MFC-FF performs better on the problems with small n and m, its relative performance edge lessens as n and m increase.

In this subsection we have compared FC, MFC, FC-FF, MFC-FF on a wide range of problems and along many different dimensions. Of the two algorithms not using FF, FC is the worst algorithm. MFC is clearly much better than FC over all problems tested. It is uniformly better than FC over all constraint tightnesses and its performance is better as n and m increase. At its worst MFC performs 82.1% of the average number of constraint checks performed by FC and at its best it performs only 60.3%. Of the two algorithms using FF the result is not as clear. FC-FF performs better than MFC-FF on 42.8% of the problems while MFC-FF performs better than FC-FF on 57.0% of the problems. It appears that although MFC-FF is better on more problems than FC-FF, its relative performance worsens as n and m increase. Overall, the best algorithm for these hard random problems is either MFC-FF or FC-FF followed by MFC and then FC.

For the record, a total of 40,800 random problems were generated for this section. The total CPU time used for the runs of the algorithms in these comparisons is

2,998,689 CPU-seconds. A total of 12,364,971,972 constraint checks were performed.

3.4 Summary

In this chapter we have introduced the phase transition phenomenon in NP-complete problems focusing on CSPs and Smith's model of hard binary CSPs. We have used Smith's model to generate a large testbed of hard random problems on which to compare many of the algorithms discussed so far in this thesis. We have shown that the BT and BM algorithms are inefficient and that FC is clearly a superior algorithm. We have also shown that the MFC algorithm is clearly superior to the FC algorithm. For problems for which the FF heuristic is inappropriate MFC is clearly the best algorithm described so far in this thesis. Finally, we have shown that although the MFC-FF algorithm's performance can be better than FC-FF it suffers from its inability to pick the future variable which has the fewest values consistent with the current instantiation. In the next chapter we address this problem.

Chapter 4

Improvements to Minimal Forward Checking

A great deal of empirical evidence[2, 34, 33, 49, 61, 94, 115] including that in Chapter 3 indicates that FC's performance on some problems can be greatly improved with the addition of the FF heuristic which picks the future variable with the smallest pruned domain size as the next variable to be instantiated. As the performance of FC is greatly improved by this heuristic for some problems it is important that MFC show a similar improvement with FF. Unfortunately, MFC's laziness can defeat the performance improvements provide by the FF heuristic. Intuitively, the FF heuristic improves performance by dynamically rearranging the search tree so that lower branching factor nodes (smaller domains) are shallower in the search tree and higher branching factor nodes (larger domains) are deeper in the search tree. Since all variations of the BT algorithm prune subtrees when an inconsistent instantiation is found, if the larger domains are deeper in the search tree, then more of the search tree is pruned by an inconsistent instantiation than if the smaller domains are deeper in the search tree. The FF heuristic is particularly appropriate for FC as FC deletes values from the future domains that are inconsistent with the current instantiation thereby changing the bushiness of the part of the search tree that it may visit. The FF heuristic dynamically minimizes the bushiness of the tree by bringing the variable with the fewest past-consistent values to the "top". MFC does not know the true domain size of the future variables and therefor has a different view of the "part of the search tree that it may visit". FC sees only the part of the search tree that is past-consistent while MFC sees that part plus the part that has not yet been pruned because of MFC's laziness. The FF heuristic with MFC then minimizes bushiness over this larger part

of the search tree, that is, the FF heuristic can be effective but not as effective as FF with FC. The FF heuristic may be misled by MFC into bringing a variable which has a larger pruned domain size than the variable with the smallest true domain size (which it doesn't know but FC does know) to the "top". This can cause MFC-FF to search a larger part of the search tree than FC-FF searches and cause MFC-FF to have relatively poorer performance. The poorer relative performance of MFC compared to FC starts to appear as n increases (the search tree gets deeper) and is most striking for the hardest problem in our testbed.

In this chapter we explore two extensions to the MFC algorithm that help to overcome this conflict between FF and MFC's lazy forward checking. The first extension is a new heuristic, called EXtra Pruning with Fail First (EXP-FF), which, by forcing the evaluation of some constraint checks avoided by the laziness of MFC, identifies a future-connected variable which has the smallest pruned domain size. The addition of this new heuristic to MFC (MFC-EXP-FF) helps MFC make a better instantiation choice. That is, MFC-EXP-FF will take a greater advantage of the FF heuristic's ability to minimize the expected branch depth than MFC-FF will. The second extension is the addition of non-chronological backtracking, namely Conflict-directed BackJumping (CBJ)[90] to MFC (which is an interesting extension regardless of whether FF is used). The MFC-CBJ algorithm performs much better than MFC as each backjump saves substantially more constraint checks than a similar backjump by the FC-CBJ algorithm. The addition of CBJ to MFC-FF (MFC-CBJ-FF) is also beneficial (but not as beneficial as the addition of CBJ to MFC).

In Section 4.1 we describe two heuristics, the EXP-FF heuristic mentioned earlier and a second heuristic, which we denote by INC-FF (for INCremental MFC with Fail First) developed independently by Bacchus and Grove[1]. Both heuristics have been designed to help MFC make a better instantiation choice by identifying a variable that has in some sense (explained below) the smallest current domain size. We then describe the addition of CBJ to the MFC algorithm and outline a number of theoretical and conjectured relationships between different hybrid combinations of MFC and FC with CBJ and FF, EXP-FF, and INC-FF. Finally, we give a comprehensive empirical

comparison in Section 4.2 of MFC and FC with CBJ, and FF, EXP-FF and INC-FF. The results of this chapter have been published in [33, 34].

4.1 Improvements to the MFC algorithm

In Section 4.1.1 we describe the EXP-FF heuristic and Bacchus and Grove's INC-FF heuristic[1]. In Section .1.2 we describe the addition of CBJ to MFC. Finally in Section 4.1.3 we outline a number of theoretical relationships between different hybrid combinations of MFC and FC with CBJ and FF, EXP-FF, and INC-FF.

4.1.1 The EXP-FF and INC-FF Heuristics

The FF heuristic as defined in [61, p. 266] is: pick the variable that has the fewest remaining values in its domain as the next variable to instantiate. A stronger version of the FF heuristic which is also defined in [61, p. 302] is: pick the variable that has the fewest remaining *consistent* values in its domain. This stronger heuristic causes no difficulty when used with FC as all values inconsistent with the past instantiati have already been removed. However, the MFC algorithm must use the weaker form of the FF heuristic as it does not know the true domain size of the future domains. The EXP-FF and INC-FF heuristics are designed to help a search algorithm pick a variable that has in some sense the fewest past-consistent values in it.

Figure 4.1 and Figure 4.2 give pseudo-code for the EXP-FF and INC-FF heuristics, respectively. The pseudo-code for both heuristics assumes that $\{v_1, \ldots, v_{i-1}\}$ are the past variables, $\{v_{i+1}, \ldots, v_n\}$ are the future variables and v_i is the current variable which has just been successfully instantiated by the search algorithm and that the search algorithm now wants to select a future variable for instantiation.

The EXP-FF heuristic given in Figure 4.1 uses the notation $||d_j||$ to mean the pruned domain size of d_j. The EXP-FF heuristic first sets each element in the array **size-d_j** to be the pruned domain size of each future variable v_j $(i + 1 \leq j \leq n)$ (lines 1-2). It then sets a "pruning factor" k to be the smallest pruned domain size of the future variables (line 3). The variable that has the domain with this smallest size is the variable that would be picked by the FF heuristic if it was used at this point.

```
for j = i + 1 to n                                                          1
    size-d_j ← ||d_j||                                                      2
k ← min(size-d_j) (i + 1 ≤ j ≤ n)                                           3
for j = i + 1 to n                                                          4
    if c_{i,j} is non-trivial then                                          5
        Find k values in d_j consistent with each element of {v_1, ..., v_i} if possible   6
        if the number of consistent values found < k then                   7
            k ← the number of consistent values found                       8
        size-d_j ← k                                                        9
return(j for the smallest size-d_j) (i + 1 ≤ j ≤ n)                         10
```

Figure 4.1: Pseudo-code for the EXP-FF heuristic.

```
k ← 0                                                                       1
while (True)                                                                2
    k ← k + 1                                                               3
    for j = i + 1 to n                                                      4
        Find k values in d_j consistent with each element of {v_1, ..., v_i} if possible   5
        if the number of consistent values found < k then return(j)         6
```

Figure 4.2: Pseudo-code for the INC-FF heuristic.

The heuristic then loops through each *future-connected* variable v_j (lines 4-9) trying to find k past-consistent (past-consistent including the successfully instantiated current variable) values in d_j. If there are less than k past-consistent values in some v_j (that is, this is the smallest domain the heuristic currently knows of) the pruning factor is reset to the new smaller size. Whether or not the variable has the smallest known size, the number of past-consistent values that are found in that variable's domain is recorded in the array size-d_j. After the loop through the future domains, the heuristic returns the variable that has the "smallest domain"[1] that it knows of in line 10.

The EXP-FF heuristic attempts to identify a future variable (with special emphasis on the future-connected variables) that would have a reasonably small past-consistent pruned domain size. It uses the knowledge that the search algorithm has of the pruned size of the future domains to set its initial pruning factor and it attempts to find a future-connected variable that has a smaller past-consistent pruned domain size. The EXP-FF heuristic does does not always find the future variable with the fewest past-consistent values (as it only examines the future-connected domains). However, it does try to pick the variable, with the smallest domain that is closely associated to the variable just instantiated.

The INC-FF heuristic given in Figure 4.2 can be seen as an incremental version of MFC. It first attempts to find one past-consistent (including the instantiated current variable) element in each future domain (lines 2-7). This is MFC without the recording. If one of the future domains has no past-consistent values in it, the corresponding variable is returned (which is correct behavior as the search algorithm will then try to instantiate that variable, find no consistent values in it and immediately backtrack). If there is one past-consistent value in each future domain then the heuristic looks for two and so on. The **while** loop beginning on line 2 is guaranteed to terminate as the domains are finite. The heuristic immediately returns with the variable for which

[1]The EXP-FF heuristic can be seen as a black box that just returns the variable that has the fewest past-consistent values that it knows of but doesn't actually do any pruning. Of course, using the heuristic this way would be illogical and a bit of a cheat (since the constraint checks would not count). The algorithms using EXP-FF and INC-FF *do* tally the extra constraint checks performed and *do* record the results of those extra constraint checks in the domain[] array.

it cannot find the number of past-consistent values that it is looking for. The INC-FF heuristic identifies the first future variable (in the order that the future variables are in) that has the smallest pruned domain size[1]. That is, the INC-FF heuristic identifies the future variable that FF will pick with FC (given that FF picks the first variable which has the domain with the smallest pruned domain size rather than picking randomly when many variables have domains with the smallest pruned domain size).

Bacchus and Grove's INC-FF heuristic allows MFC-FF to follow the same dynamic instantiation order as FC-FF thereby receiving all the benefits of the FF heuristic and assures less cost in terms of number of constraint checks performed than FC-FF. The principle underlying our design of the EXP-FF heuristic is to maintain as much of FF's benefits as possible but to expend as few constraint checks as possible in order to identify an appropriate variable to instantiate. The EXP-FF heuristic attempts to maintain FF's property of bringing together highly related variables by looking only at the domains of variables that have been pruned by the instantiation of the current variable. As mentioned earlier, Bacchus and Grove[1] independently developed the INC-FF heuristic at the same time as we developed the EXP-FF heuristic[2].

4.1.2 Conflict-Directed Backjumping

A weakness in chronological backtracking algorithms is that they backtrack to the previously instantiated variable even if it is not the "cause of failure". Non-chronological backtracking algorithms, also called intelligent backtracking algorithms, identify past variables whose instantiations may be the reason that a search cannot go deeper (at the time a backtrack is about to occur). The non-chronological backtracking algorithm moves back to the deepest such variable (reason) uninstantiating the intermediate variables and the variable which is the deepest possible reason[3]. Prosser[90] has introduced a form of non-chronological backtracking, called Conflict-Directed Back-

[2]Personal communication.

[3]A form of non-chronological backtracking which does not uninstantiate the intermediate variables is called dynamic backtracking[3, 4, 53].

Jumping (CBJ), which allows backtracking search to jump back to a variable that may be the cause of failure. CBJ is a stronger version of the BJ algorithm mentioned in Section 1.3 as it allows the search to jump back (over more than one intermediate variable) multiple times when the variables jumped back to have no past-consistent values left in their domains. In this case, the BJ algorithm can only jump back once, say to v_h, at which point it finds no past-consistent value in d_h, and then backtracks to the variable instantiated previous to v_h. CBJ maintains a conflict set (**conf-set**$_i$) for every variable v_i which records the index of the past instantiations that deleted a value from the variables' domain. Each conflict set is initially set to $\{0\}$. Every time a constraint check fails between a value in d_i and the instantiation of some past variable v_h the index of the instantiation is added to the conflict set of v_i. When there are no more values to be tried for the current variable v_i, CBJ backtracks to the deepest instantiation in the search tree v_h whose index is in the conflict set of v_i. That is it jumps back to the deepest variable in the search which caused a conflict. This is the deepest variable whose uninstantiation can "undelete" a value in d_i. As the search backtracks, the conflict set of v_h is set to the union of the conflict set of v_i and the conflict set of v_h (removing the reference to v_h) so that no information about conflicts is lost. Soundness and completeness results for CBJ can be found in [67].

To add CBJ to FC, another set (**past-fc**$_i$) is included which records the indices of the past instantiations which have deleted a value from d_i by a forward check. This set can be calculated from the **domain**$_i'$ array. The conflict set **conf-set**$_i$ in FC-CBJ is updated in only three places. The first place is when a forward check fails, that is, the instantiation of the current variable v_i completely prunes a future domain d_j. Then, the conflict set **conf-set**$_i$ is updated to the union of **conf-set**$_i$ and **past-fc**$_j$. In Figure 4.3, the labeling function for **mfc-label** (from Figure 2.5) is updated with one extra line (line **13**) which updates the conflict set for this case. At line **13**, FC and MFC have exactly the same information about which past variables deleted values from d_j as they both completely prune the future domain. The second and third place that **conf-set**$_i$ is updated is in backtracking. To backtrack, the deepest instantiation, say v_h, which is in the union of **conf-set**$_i$ and **past-fc**$_i$ is chosen. The conflict **conf-set**$_h$ is updated to

```
function mfc-cbj-label(ii)
  i ← refv_ii                                                          1
  consistent ← False                                                   2
  for each v_i^l ∈ d_i while (not consistent)                          3
    if past-consistent(v_i^l,ii) then                                  !4
      consistent ← True                                                5
      v_i ← v_i^l                                                      6
      for jj = ii + 1 to n while consistent                            7
        consistent ← min-forward-check(ii,jj)                          8
      if not consistent then                                           9
        domain_i^l = −(ii − 1)                                         10
        min-undo-reductions(ii)                                        11
        j ← refv_jj^-1                                                 12
        conf-set_i ← union(conf-set_i,past-fc_j)                       13
```

Figure 4.3: Pseudo-code for the MFC-CBJ labeling function.

the union of $\textbf{conf-set}_h$, $\textbf{conf-set}_i$, and $\textbf{past-fc}_i$ minus the index of the instantiation of v_h. A backtrack is then performed by undoing all constraint checks performed for the instantiations indexed by $hh + 1$ to ii updating the conflict set for each intermediate variable to $\{0\}$. Figure 4.4, updates the **mfc-unlabel** function (from Figure 2.9) with extra lines (lines $2 - 8$) to update the conflict set and to jump back. A complete description of the CBJ and FC-CBJ algorithms can be found in [67, 90]. Soundness and completeness results can be found in [67].

In Section 4.2 we empirically compare on our testbed eight hybrid algorithms that combine FC and MFC with CBJ and FF, or EXP-FF, or INC-FF, namely, FC-CBJ, MFC-CBJ, FC-CBJ-FF, MFC-CBJ-FF, MFC-EXP-FF, MFC-CBJ-EXP-FF, MFC-INC-FF, and MFC-CBJ-INC-FF. We also include the comparisons reported in Section 3.3.2 for FC, MFC, FC-FF and MFC-FF.

```
function mfc-cbj-unlabel(ii)
    i ← refv_ii                                                          1
    hh ← max(conf-set_i,past-fc_i)                                       2
    h ← refv_ihh                                                         3
    conf-set_h ← union(conf-set_h,conf-set_i,past-fc_i) - hh             4
    for jj = ii downto hh + 1                                            5
        min-undo-reductions(jj)                                          6
        j ← refv_jj                                                      7
        conf-set_j ← {0}                                                 8
    domain_h^el(v_h) ← −(hh − 1)                                         9
    if ∃k domain_h^k ≥ 0                                                 10
        then consistent ← True                                           11
        else consistent ← False                                          12
    return(hh)                                                           13
```

Figure 4.4: Pseudo-code for the MFC-CBJ unlabeling function.

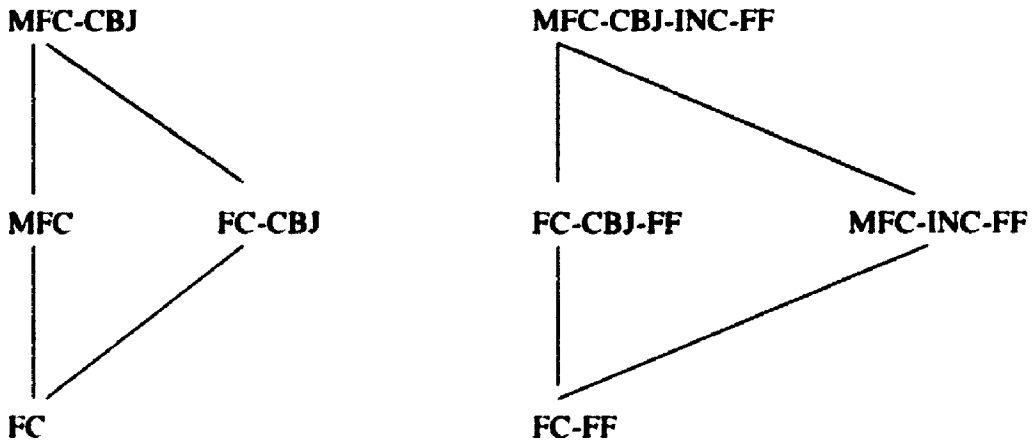

Figure 4.5: The theoretical (left) and conjectured (right) constraint check relationship between some of the algorithms.

4.1.3 Theoretical and Conjectured Relationships

The theoretical and conjectured constraint check relationships between the algorithms we investigate in this chapter are displayed in Figure 4.5. The arcs connecting the algorithms represent the relationship that the algorithm above performs the same or fewer constraint checks than the one below. Some of the relationships have been demonstrated in previous papers. The theoretical relationship between FC-CBJ and FC in [68], and between FC and MFC in [30, 1] and in this thesis, and the conjectured relationship between FC-FF and MFC-INC-FF in [1].

We first look at the algorithms that do not use FF. As in Chapter 2, the following theorems rely on a fixed instantiation order. We informally prove the following theorems. We do not use Kondrak and Van Beek's method of proving the following theorems leaving such detailed proofs as future work.

Theorem 4.1 *The FC-CBJ algorithm and the MFC-CBJ algorithm visit the same nodes in the same order.*

Proof of Theorem 4.1 The FC-CBJ algorithm uses the information regarding the deepest past variables that deleted a value out of a future domain that was completely pruned as the basis of jumping back to a past variable. From Theorem 2.7 we know that the FC and MFC algorithms visit the same nodes in the same order which means that MFC completely prunes a future domain whenever FC would completely prune a future domain. Therefore both algorithms when using CBJ would have the same information for backjumping needed by CBJ and therefore FC-CBJ and MFC-CBJ visit the same nodes in the same order. □

Theorem 4.2 *The worst case performance of MFC-CBJ in terms of the number of constraint checks performed is the number of constraint checks performed by FC-CBJ.*

Proof of Theorem 4.2 A direct consequence of Theorems 2.8 and 4.1. □

Theorem 4.3 *The MFC-CBJ algorithm visits a (not necessarily proper) subset of the nodes visited by MFC.*

Proof of Theorem 4.3 The only change in the order of MFC's search caused by CBJ is the algorithm can now jump back to previously visited nodes at which point it acts as MFC until the next backjump. Therefore MFC-CBJ visits a (not necessarily proper) subset of the nodes visited by MFC. □

Theorem 4.4 *The worst case performance of MFC-CBJ in terms of the number of constraint checks performed is the number of constraint checks performed by MFC.*

Proof of Theorem 4.4 Direct consequence of Theorem 4.3.

The conjectured relationships between the algorithms with FF are more interesting. It is important to define exactly how the FF heuristic is implemented in order to arrive at the relationships shown in Figure 4.5. In this thesis we assume that the instantiation indirection array $\mathbf{refv}_{ii}$ is used. The indices of the array refer to the instantiation level while the value of the array is the instantiated variable. The FF heuristic by its nature causes some reordering of the instantiation array i.e. it places the variable with the smallest domain in the position of the next instantiation. In our implementation we memorize the order of the future variables immediately after finding the smallest domain ahead. If a search backtracks to the same variable, the future variables are put back into that order. This ensures that the effect of finding the smallest domain for other variables deeper in the search tree will not cause a different ordering of the future variables the next time the variable is instantiated. For example, the relationship between FC-CBJ-FF and FC-FF would not be true if we didn't memorize the order of the future variables. Assume that instantiation $\mathbf{i}$ is inconsistent and FC-CBJ-FF can backjump (more than one instantiation) to instantiation $\mathbf{h}$ while FC-FF can only move back to $\mathbf{i} - 1$. By the time FC-FF's search moves back to $\mathbf{h}$, its view of the order of the future variables can change from that of FC-CBJ-FF's as it performed more search. However, if the order is returned to what it was before the search moved forward from $\mathbf{h}$ then both algorithms have the same ordering of the future variables. Therefore FC-CBJ-FF performs the same or fewer constraint checks than FC-FF as it searches only a subset of the nodes that FC-FF does. The relationship between

MFC-CBJ-INC-FF and FC-CBJ-FF and MFC-INC-FF can be shown by arguing that they have the same future variables at each level and that they pick the same variables because of the definition of the INC-FF heuristic.

It is interesting to note that MFC-CBJ-FF is not always better than MFC-FF. Although at each level they have the same future variables to choose from, the nature of MFC can cause FF to pick a different variable. Using the example from above, say MFC-CBJ-FF has backjumped to instantiation h while MFC-FF has moved back to instantiation i − 1. While MFC-FF is performing extra search until it moves back to h, it may be performing extra constraint checks with past variables (that is, it is performing avoided constraint checks). When MFC-FF moves back to h it will then have a different idea of what is the smallest current domain leading to a different search order.

4.2 An Empirical Comparison that Includes the New Hybrid Algorithms

In this section we empirically compare the 12 algorithms, FC, MFC, FC-CBJ, MFC-CBJ, FC-CBJ-FF, MFC-CBJ-FF, MFC-EXP-FF, MFC-CBJ-EXP-FF, MFC-INC-FF, and MFC-CBJ-INC-FF. As before, we run each algorithm on each problem in our testbed counting the number of constraint checks performed in solving each problem.

We begin by showing the overall results of the experiments in Tables 4.1 and 4.2. The first table shows the percentage of times one algorithm performs better than another by constraint checks for the entire testbed. For example, MFC is better than MFC-FF for 3.9% of the problems (the same for 0.5%). Of the algorithms not using fail first, MFC-CBJ is the best, as expected. Of the algorithms using fail first, MFC-CBJ-EXP-FF is the best. Of the algorithms not using CBJ and fail first, MFC is the best, as expected, and for algorithms not using CBJ and using fail first MFC-EXP-FF is best. In every case, the minimal forward checking version of an algorithm is better than the forward checking version. Of the algorithms using the EXP-FF and INC-FF heuristic, MFC-EXP-FF is better than MFC-INC-FF 65% of the time and MFC-CBJ-EXP-FF is better than MFC-CBJ-INC-FF 65.2% of the time. With and

BETTER THAN	FC	MFC	FC CBJ	MFC CBJ	FC FF	MFC FF	FC CBJ FF	MFC CBJ FF	MFC EXP FF	MFC CBJ EXP FF	MFC INC FF	MFC CBJ INC FF
FC	-	0.0 (0.0)	0.0 (7.8)	0.0 (0.0)	2.5 (0.5)	1.2 (0.0)	2.4 (0.5)	0.9 (0.0)	1.7 (0.1)	1.7 (0.1)	1.8 (0.1)	1.8 (0.1)
MFC	100.0	-	77.5 (0.2)	0.0 (6.1)	7.5 (0.1)	3.9 (0.5)	7.4 (0.1)	3.4 (0.5)	3.9 (0.3)	3.8 (0.3)	4.0 (0.3)	3.9 (0.3)
FC CBJ	92.2	22.4		0.0 (0.0)	3.0 (0.4)	1.7 (0.0)	2.9 (0.4)	1.2 (0.0)	2.0 (0.1)	1.9 (0.1)	2.1 (0.1)	2.0 (0.1)
MFC CbJ	100.0	93.9	100.0	--	10.4 (0.1)	6.2 (0.5)	10.1 (0.1)	4.9 (0.5)	5.3 (0.3)	5.1 (0.3)	5.3 (0.4)	5.1 (0.4)
FC FF	97.0	92.4	96.6	89.5	-	42.8 (0.2)	0.0 (71.4)	34.7 (0.2)	4.3 (0.6)	3.6 (0.6)	0.0 (2.9)	0.0 (2.9)
MFC FF	98.8	95.6	98.2	93.3	57.0	-	55.8 (0.3)	1.7 (19.0)	17.6 (1.2)	16.8 (1.2)	20.0 (0.8)	19.3 (0.8)
FC-CBJ FF	97.1	92.5	96.7	89.8	28.6	43.9	—	35.7 (0.2)	5.1 (0.6)	3.9 (0.6)	1.1 (2.9)	0.0 (2.9)
MFC-CBJ FF	99.1	96.2	98.8	94.5	65.1	79.3	64.1	--	21.9 (1.3)	20.8 (1.3)	24.4 (0.9)	23.4 (0.9)
MFC-EXP FF	98.2	95.7	97.9	94.3	95.1	81.2	94.3	76.8	—	0.1 (64.4)	64.9 (3.2)	62.5 (3.2)
MFC-CBJ EXP-FF	98.2	95.8	98.0	94.6	95.8	82.0	95.4	77.9	35.5	—	67.8 (3.2)	65.2 (3.3)
MFC-INC FF	98.1	95.7	97.8	94.3	97.1	79.1	96.0	74.7	31.9	29.0	—	0.0 (71.4)
MFC-CBJ INC-FF	98.1	95.8	97.9	94.5	97.1	79.9	97.1	75.7	34.3	31.5	28.6	—

Table 4.1: Percentage of times one algorithm performs better than another by number of constraint checks, for problems with $n \in \{10, 15, 20, 25\}$, $m \in \{3.6, 9\}$, $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. Percentage of times algorithms perform the same number of constraint checks are in brackets. (10,200 problems).

without CBJ the EXP-FF version of MFC is better than the INC-FF version.

The second table gives the average, the standard deviation for the average, the median, the geometric mean, the maximum, the minimum, and the percentage of constraint checks by geometric mean performed relative to FC by the 12 algorithms on the entire testbed. These measures indicate that MFC-CBJ-EXP-FF is the best algorithm, followed closely by MFC-EXP-FF, MFC-CBJ-INC-FF, and then MFC-INC-FF. These algorithms are then followed by either MFC-CBJ-FF, MFC-FF, FC-CBJ-FF, and FC-FF, or FC-CBJ-FF, FC-FF, MFC-CBJ-FF, and MFC-FF depending on whether the median and geometric mean are used or the average and the maximum number of constraint checks, respectively. Finally, these algorithms are followed by

Alg	Ave	Std Dev	Med	Geo Mean	Max	Min	% FC
FC	710168	2637242	8626	13539	58051198	15	100.0
MFC	433021	1603277	6170	9483	35677917	12	70.0
FC-CBJ	564618	2131850	6592	10552	47140736	15	77.9
MFC-CBJ	342290	1290070	4594	7283	28510271	12	53.8
FC-FF	33450	109181	1872	2538	1254426	15	18.7
MFC-FF	35608	117645	1768	2439	1334088	12	18.0
FC-CBJ-FF	33358	109038	1846	2515	1251805	15	18.6
MFC-CBJ-FF	34575	114808	1668	2324	1295503	12	17.2
MFC-EXP-FF	22600	72605	1391	1914	847274	12	14.1
MFC-CBJ-EXP-FF	22535	72499	1377	1896	846036	12	14.0
MFC-INC-FF	23474	75872	1439	1958	884719	12	14.5
MFC-CBJ-INC-FF	23414	75784	1423	1941	883040	12	14.3

Table 4.2: Average, standard deviation, median, geometric mean, maximum and minimum number of constraint checks for the FC, MFC, FC-CBJ, MFC-CBJ, FC-FF, MFC-FF, FC-CBJ-FF, MFC-CBJ-FF, MFC-EXP-FF, MFC-CBJ-EXP-FF, MFC-INC-FF, and MFC-CBJ-INC-FF algorithms for problems with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$, and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. (10,200 problems).

the algorithms without the fail first heuristic in the order MFC-CBJ, MFC, FC-CBJ, and FC.

It is evident that the algorithms based on MFC are the best. MFC performs 30% fewer constraint checks than FC. Without the fail first heuristic, MFC-CBJ reduces the number of constraint checks performed by half. MFC-CBJ performs only 69.0% of the constraint checks performed by FC-CBJ. With the fail first heuristic, MFC-EXP-FF is able to reduce the number of constraint checks to only 14.1% of FC's constraint checks, 75.4% of FC-FF's, and 76.1% of FC-CBJ-FF's. Of the two new fail first heuristics for MFC, EXP-FF and INC-FF, MFC-EXP-FF is only slightly better than MFC-INC-FF performing 97.8% of the constraint checks and MFC-CBJ-EXP-FF is only slightly better than MFC-CBJ-INC-FF performing 97.7% of the constraint checks.

We next look at a few selected scatter plots to compare visually the performance of algorithms over the whole testbed. A scatter plot for MFC versus MFC-CBJ is displayed in Figure 4.6. MFC-CBJ appears to offer large savings in the number of constraint checks performed for quite a few of the problems. As expected MFC-CBJ performs better than MFC for all problems. Figures 4.7 and 4.8 show that MFC-EXP-FF is clearly superior to FC and FC-FF except for a few relatively easy problems. The MFC-EXP-FF algorithms performance is much better for the hardest problems (that is, to the right of the figures). Comparing Figure 3.18, which shows a scatter plot for FC-FF versus MFC-FF, to Figure 4.8, which shows the scatter plot for FC-FF versus MFC-EXP-FF, it is evident that the problem which MFC has with the FF heuristic has been overcome for almost all of the hard problems (there is one minor exception). Figures 4.9 and 4.10 show that the performances of MFC-EXP-FF versus MFC-INC-FF, and MFC-CBJ-EXP-FF versus MFC-CBJ-INC-FF are about the same, in both figures the two algorithms perform (equally) worse than the other on about half of the problems.

CBJ is a valuable addition to the forward checking algorithms that do not use FF. FC-CBJ performs fewer constraint checks than FC for 92.2% of the problems and MFC-CBJ performs fewer constraint checks than MFC for 93.9% of the problems

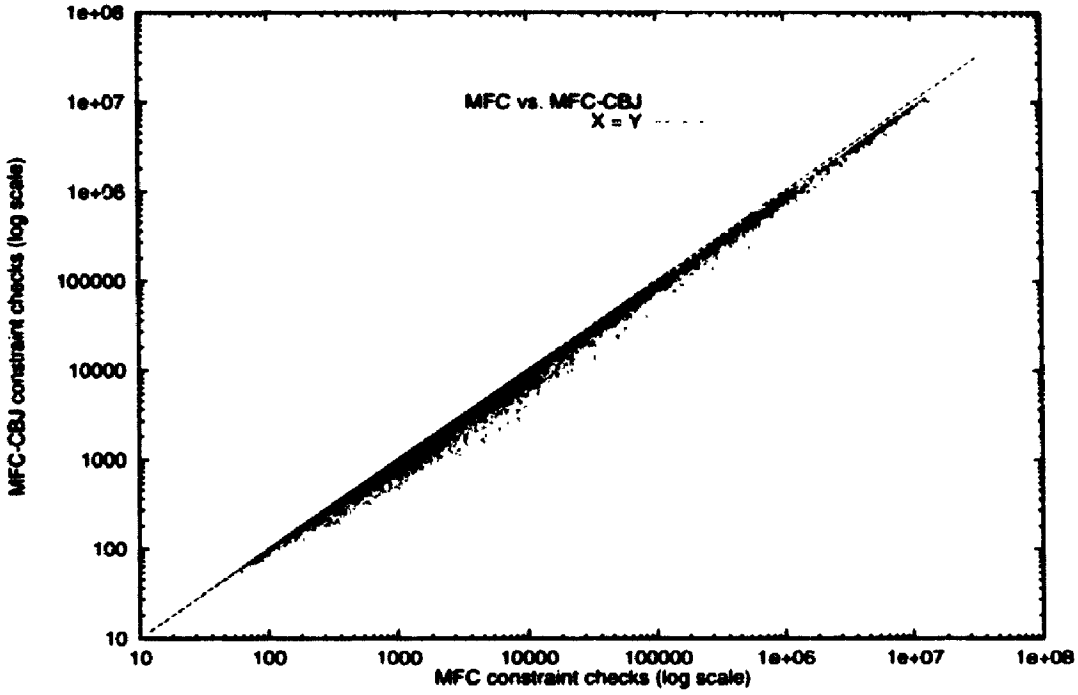

Figure 4.6: The number of constraint checks performed by MFC versus the number of constraint checks performed by MFC-CBJ on the same problem for problems with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. (10,200 problems).

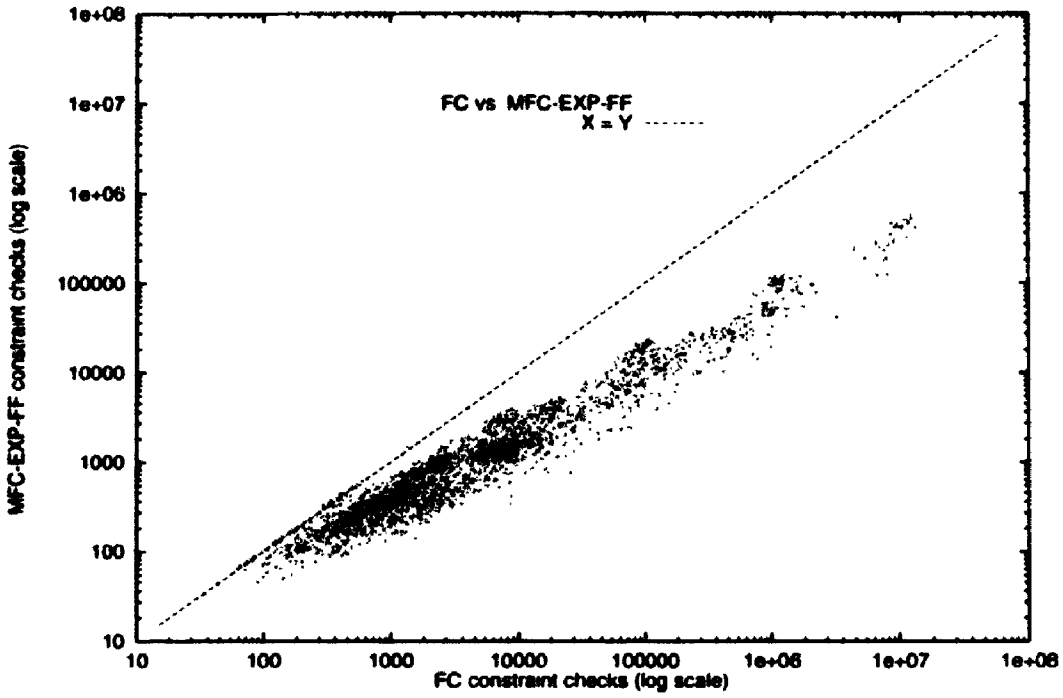

Figure 4.7: The number of constraint checks performed by FC versus the number of constraint checks performed by MFC-EXP-FF on the same problem for problems with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. (10,200 problems).

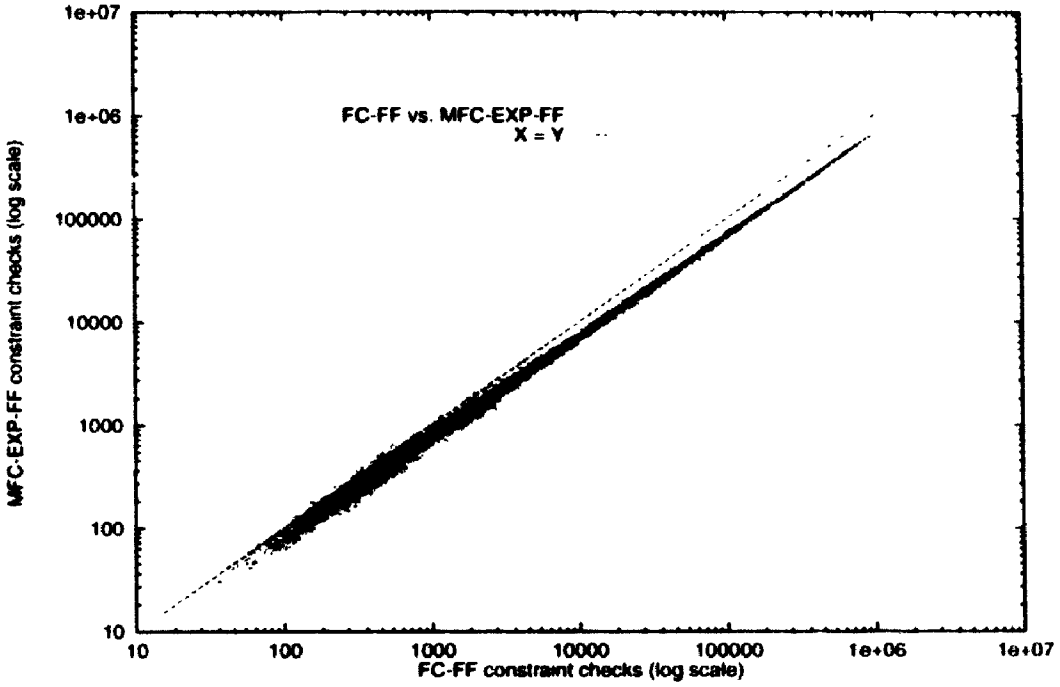

Figure 4.8: The number of constraint checks performed by FC-FF versus the number of constraint checks performed by MFC-EXP-FF on the same problem for problems with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. (10,200 problems).

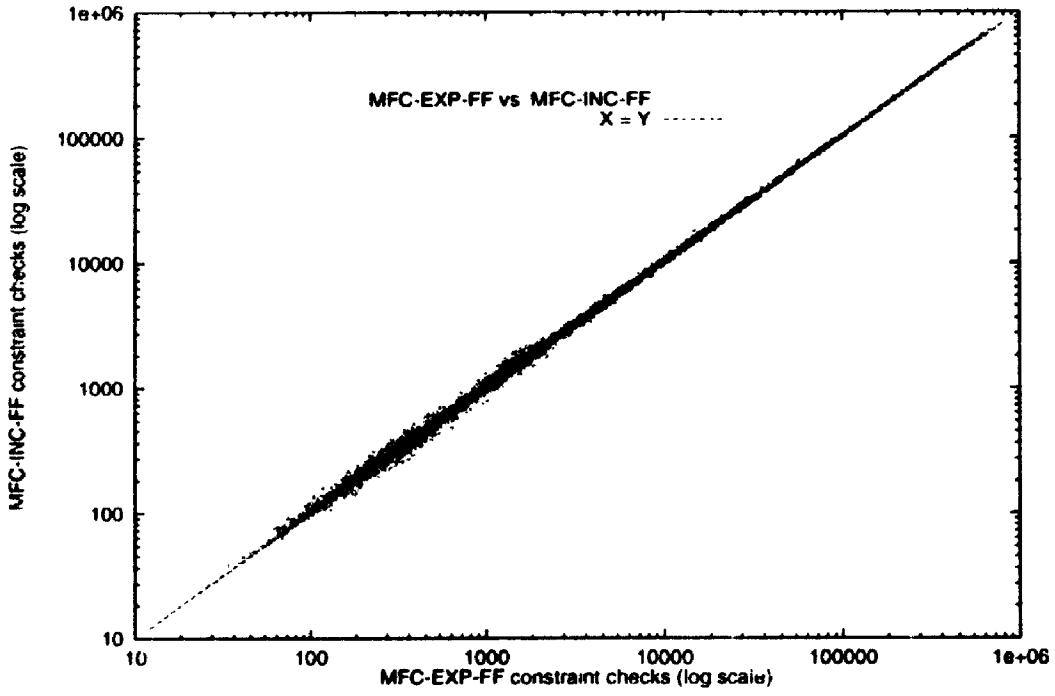

Figure 4.9: The number of constraint checks performed by MFC-EXP-FF versus the number of constraint checks performed by MFC-INC-FF on the same problem for problems with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. (10,200 problems).

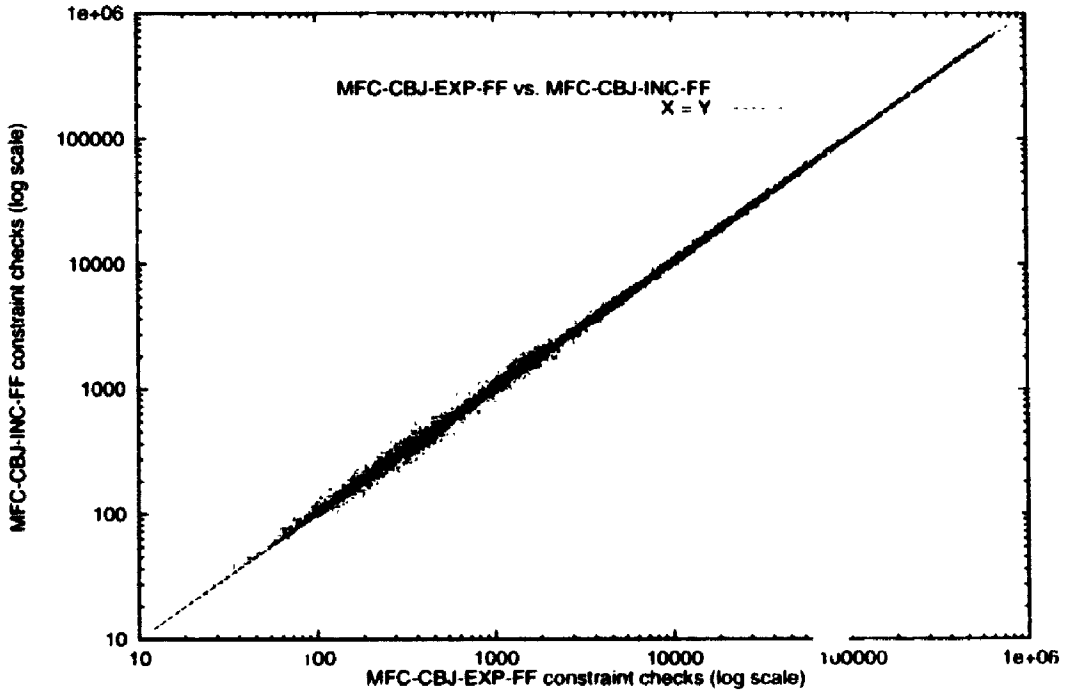

Figure 4.10: The number of constraint checks performed by MFC-CBJ-EXP-FF versus the number of constraint checks performed by MFC-CBJ-INC-FF on the same problem for problems with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. (10,200 problems).

resulting in a 22.1% reduction in the number of constraint checks performed in the first case and 16.2% in the second. However CBJ is not as valuable an addition to the forward checking algorithms that use FF. FC-CBJ-FF performs fewer constraint checks than FC-FF for 28.6% of the problems with 71.4% being the same, giving a reduction of 0.9% in the number of constraint checks. MFC-CBJ-FF performs fewer constraint checks than MFC-FF for 79.3% of the problems with 19% being the same, giving a reduction of 4.7% in the number of constraint checks. MFC-CBJ-EXP-FF performs fewer constraint checks than MFC-EXP-FF for 35.5% of the problems with 64.4% being the same, giving a reduction of 0.9% in the number of constraint checks. And finally, MFC-CBJ-INC-FF is better than MFC-INC-FF for 28.6% of the problems, with 71.4% being the same, giving a reduction of 0.9% in the number of constraint checks. The only algorithm that seems to gain from having CBJ added (although the gain is marginal) is MFC-FF. We expected that the addition of CBJ to MFC-FF would help MFC-FF to back out of poor instantiation choices quickly. However, adding CBJ does not seem to be as useful for any of the other algorithms that use FF.

Past comparisons[2, 104] have also found that adding CBJ is for the most part ineffective when an algorithm is using FF. The only exception to this is CSPs with very sparse graphs and on what are called by Smith "exceptionally hard problems"[56, 104]. Our experiments confirm that the addition of CBJ is useful only for sparse graphs. We examine our testbed (as above) at the two extremes, with sparse graphs having $p_1 = 0.2$ (Tables 4.3 and 4.4) and for dense graphs with $p_1 = 1.0$ (Tables 4.5 and 4.6). For the sparse graphs, the "percentage better than" for the algorithms with CBJ remains basically the same as in Table 4.1 but the reduction in constraint checks is about 4.2% for FC-CBJ-FF, MFC-CBJ-EXP-FF, and MFC-CBJ-INC-FF, and about 11.9% for MFC-CBJ-FF. For the dense graphs, the "percentage better than" for the algorithms with CBJ is lower than in Table 4.1 and there are almost no savings in constraint checks for any of the algorithms with fail first. The savings for FC-CBJ-FF, MFC-CBJ-EXP-FF, and MFC-CBJ-INC-FF is only 0.2% and about 2.3% for MFC-CBJ-FF.

BETTER THAN	FC	MFC	FC CBJ	MFC CBJ	FC FF	MFC FF	FC CBJ FF	MFC CBJ FF	MFC EXP FF	MFC CBJ EXP FF	MFC INC FF	MFC CBJ INC FF
FC		0 0 (0 2)	0 0 (12 8)	0 0 (0 2)	2 8 (2 0)	2 0 (0 2)	2 3 (2 0)	1 0 (0 2)	1 8 (0 3)	1 3 (0 2)	2 2 (0 2)	1 8 (0 2)
MFC	99.8		36 8 (0 7)	0 0 (11 3)	8 5 (0 3)	4 7 (1 5)	8 2 (0 3)	2 8 (1 7)	4 8 (1 7)	3 8 (1 7)	5 0 (1 2)	4 3 (1 2)
FC CBJ	87 2	62 5		0 0 (0.2)	4 5 (1 8)	4 7 (0 3)	3 7 (2 0)	2 7 (0 0)	2 8 (0 2)	1 7 (0 2)	3 0 (0 2)	2 5 (0 2)
MFC CBJ	99 8	88 7	99 8		13 5 (0 2)	10 2 (1 7)	12 3 (0 3)	6 0 (2 0)	7 5 (1 8)	6 3 (1 8)	6 8 (1 5)	6 0 (1 5)
FC FF	95 2	91 2	93 7	86 3		45 3 (0 8)	0 0 (70 5)	33 5 (1 0)	13 2 (1 3)	8 8 (1 7)	0 0 (3 7)	0 0 (3 7)
MFC FF	97 8	93 8	95 0	88 2	53 8		49 7 (1 0)	0 7 (24 7)	27 8 (5 2)	23 8 (5 5)	29 2 (4 3)	24 7 (4 2)
FC-CBJ FF	95 7	91 5	94 3	87 3	29 5	49 3		36 7 (1 2)	18 5 (1 2)	11 0 (1 5)	5 7 (3 8)	0 0 (3 7)
MFC-CBJ FF	98 8	95 5	97 3	92 0	65 5	74 7	62 2		39 7 (6 5)	33 2 (6 3)	37 0 (4 5)	33 2 (4 5)
MFC-EXP FF	97 8	93.5	97 0	90 7	85 5	67 0	80 3	53 8		0 0 (57 8)	43 7 (13 3)	37 7 (12 5)
MFC-CBJ EXP-FF	98 5	94 5	98 2	91 8	89 5	70 7	87 5	60 5	42 2		51 8 (12 5)	43 5 (13 2)
MFC-INC FF	97 7	93.8	96 8	91 7	96.3	66 5	90 5	58 5	43 0	35 7		0 0 (70 5)
MFC-CBJ INC FF	98 0	94 5	97 3	92 5	96 3	71 2	96 3	62 3	49 8	43 3	29 5	

Table 4.3: Percentage of times one algorithm is better than another by number of constraint checks, for problems with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$ and $p_1 = 0.2$. Percentage of times algorithms perform same number of constraint checks are in brackets. (600 problems).

Alg	Ave	Std Dev	Med	Geo Mean	Max	Min	% FC
FC	208806	1691698	2255	3254	29386648	18	100.0
MFC	126078	1064043	1672	2341	20976509	12	71.9
FC-CBJ	70741	706482	1143	1664	15665537	18	51.1
MFC-CBJ	44896	497224	830	1178	11399517	12	36.2
FC-FF	1313	2954	305	379	34431	18	11.6
MFC-FF	1338	2906	352	387	27584	12	11.9
FC-CBJ-FF	1199	2640	300	363	29003	18	11.2
MFC-CBJ-FF	1117	2368	315	341	25279	12	10.5
MFC-EXP-FF	949	1894	256	308	16300	12	9.5
MFC-CBJ-EXP-FF	886	1762	244	295	15545	12	9.1
MFC-INC-FF	945	2036	244	302	22978	12	9.3
MFC-CBJ-INC-FF	866	1828	239	289	19444	12	8.9

Table 4.4: Average, standard deviation. median, geometric mean, maximum and minimum number of constraint checks for the FC, MFC, FC-CBJ. MFC-CBJ, FC-FF, MFC-FF, FC-CBJ-FF, MFC-CBJ-FF, MFC-EXP-FF, MFC-CBJ-EXP-FF, MFC-INC-FF, MFC-CBJ-INC-FF algorithms for problems with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$ and $p_1 = 0.2$. (600 problems).

BETTER THAN	FC	MFC	FC CBJ	MFC CBJ	FC FF	MFC FF	FC CBJ FF	MFC CBJ FF	MFC EXP FF	MFC CBJ EXP FF	MFC INC FF	MFC CBJ INC FF
FC		0 0 (0 0)	0.0 (5.7)	0 0 (0 0)	1 7 (0 2)	0 7 (0 0)	1 7 (0 2)	0 7 (0 0)	1 8 (0 0)	1 8 (0 0)	1 5 (0 0)	1 5 (0 0)
MFC	100 0	-	93.3 (0 0)	0.0 (3 7)	5 3 (0 0)	2 5 (0 0)	5 3 (0 0)	2 3 (0 0)	3 2 (0 0)	3 2 (0 0)	3 2 (0 0)	3 2 (0 0)
FC CBJ	94 3	6 7		0 0 (0 0)	2 0 (0.0)	1 2 (0 0)	2 0 (0 0)	1 2 (0 0)	2 0 (0 0)	2 0 (0 0)	1 7 (0 0)	1 7 (0 0)
MFC CBJ	100.0	96.3	100.0	-	8 2 (0 2)	3 5 (0 0)	8 0 (0 2)	3 0 (0 2)	4 0 (0 0)	4 0 (0 0)	3 7 (0 2)	3 7 (0 2)
FC FF	98 2	94 7	98.0	91 7		43 8 (0 0)	0 0 (74 3)	36 0 (0.3)	2 8 (0 8)	2 8 (0 8)	0 0 (3 8)	0 0 (3 8)
MFC FF	99 3	97 5	98.8	96 5	56 2		56 0 (0 0)	2.5 (17 7)	11 8 (0 3)	11 7 (0 2)	16 2 (0 0)	15 8 (0 0)
FC-CBJ FF	98 2	94 7	98.0	91 8	25 7	44 0		36 5 (0.3)	3 0 (0 8)	2 8 (0 8)	0 2 (3 8)	0 0 (3 8)
MFC-CBJ FF	99 3	97.7	98 8	96 8	63 7	79 8	63 2		13 3 (0 7)	13 2 (0 7)	18 3 (0 0)	18 0 (0 2)
MFC-EXP FF	98 2	96 8	98 0	96.0	96.3	87 8	96 2	86 0		0 0 (73 0)	81 5 (1 3)	81 0 (1 3)
MFC-CBJ EXP-FF	98 2	96 8	98 0	96 0	96 3	88 2	96 3	86 2	27 0		82 0 (1 3)	81 3 (1 3)
MFC-INC FF	98 5	96.8	98.3	96 2	96 2	83 8	96 0	81 7	17 2	16 7		0 0 (74 3)
MFC-CBJ INC-FF	98.5	96 8	98 3	96 2	96 2	84 2	96 2	81 8	17 7	17 3	25 7	

Table 4.5: Percentage of times one algorithm is better than another by number of constraint checks, for problems with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$ and $p_1 = 1.0$. Percentage of times algorithms perform the same number of constraint checks are in brackets. (600 problems).

Alg	Ave	Std Dev	Med	Geo Mean	Max	Min	% FC
FC	1058191	2851042	9795	28570	14043395	134	100.0
MFC	658462	1755111	7578	20393	8781418	103	71.4
FC-CBJ	931048	2508626	8858	25492	12272659	134	89.2
MFC-CBJ	573896	1530216	6686	17931	7501832	103	62.8
FC-FF	82758	189760	4697	7544	843905	133	26.4
MFC-FF	87737	202927	4478	7154	885469	93	25.0
FC-CBJ-FF	82681	189605	4697	7530	842263	133	26.4
MFC-CBJ-FF	85988	198949	4477	6989	867546	93	24.5
MFC-EXP-FF	55740	125615	3706	5599	552540	94	19.6
MFC-CBJ-EXP-FF	55684	125498	3706	5589	552011	94	19.6
MFC-INC-FF	58230	131482	3763	5855	587167	108	20.5
MFC-CBJ-INC-FF	58181	131386	3763	5845	586665	108	20.5

Table 4.6: Average, standard deviation, median, geometric mean, maximum and minimum number of constraint checks for the FC, MFC, FC-CBJ, MFC-CBJ, FC-FF, MFC-FF, FC-CBJ-FF, MFC-CBJ-FF, MFC-EXP-FF, MFC-CBJ-EXP-FF, MFC-INC-FF, MFC-CBJ-INC-FF algorithms for problems with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$ and $p_1 = 1.0$. (600 problems).

Bacchus and Van Run[2] and Smith[104] reason that the FF heuristic tends to cluster highly related variables[5] together thereby reducing the probability of large backjumps. Our experiments show that although there is a large reduction in the number of backjumps, there is also a significant reduction in the average savings for each backjump. On average, FC-CBJ performs 0.11 backjumps per attempted labeling[6] searched while FC-CBJ-FF performs 0.04 backjumps per attempted labeling searched. This reduction in backjumps per attempted labeling does not fully account for the poorer relative performance of FC-CBJ-FF to FC-FF, as compared to FC-CBJ to FC. There is also a large reduction in the average number of constraint checks saved per backjump. FC-CBJ on average saves 76 constraint checks per backjump while FC-CBJ-FF saves only 3 constraint checks per backjump. It is easy to see why this is so. The FF heuristic picks a future variable with the smallest pruned domain size as the next variable to instantiate. This implies that even if an algorithm with FF could jump back over these "smallest pruned domains" there would be very few constraint checks avoided as the smallest pruned domain sizes are probably very close to one. There would be no part of the search tree avoided or a relatively small part of it.

We next look at how much the number of constraint checks is reduced by MFC, MFC-CBJ, FC-FF, and MFC-EXP-FF versus FC. Figure 4.11 shows the number of constraint checks performed versus the percentage of problems solved at or before that number of constraint checks. There is a clear difference between the algorithms that use FF and those that do not. The FF algorithms have a much more rapid climb with a growing separation between the two sets of algorithms implying that many more problems are being solved with fewer constraint checks. For the algorithms that do not use FF, MFC-CBJ is the best, followed by MFC and then FC. For the algorithms that use FF, MFC-EXP-FF is best followed by FC-FF.

Finally, we look at the percentage of FC's constraint checks performed by the 12 algorithms broken down by n and m (Table 4.7). Clearly, across n and m, MFC-CBJ is the best algorithm without FF and MFC-CBJ-EXP-FF is the best algorithm with

[5] *cf.* the discussion on FF with FC bringing together highly related variables in Section 1.3.4.

[6] We count the number of calls to the labeling function.

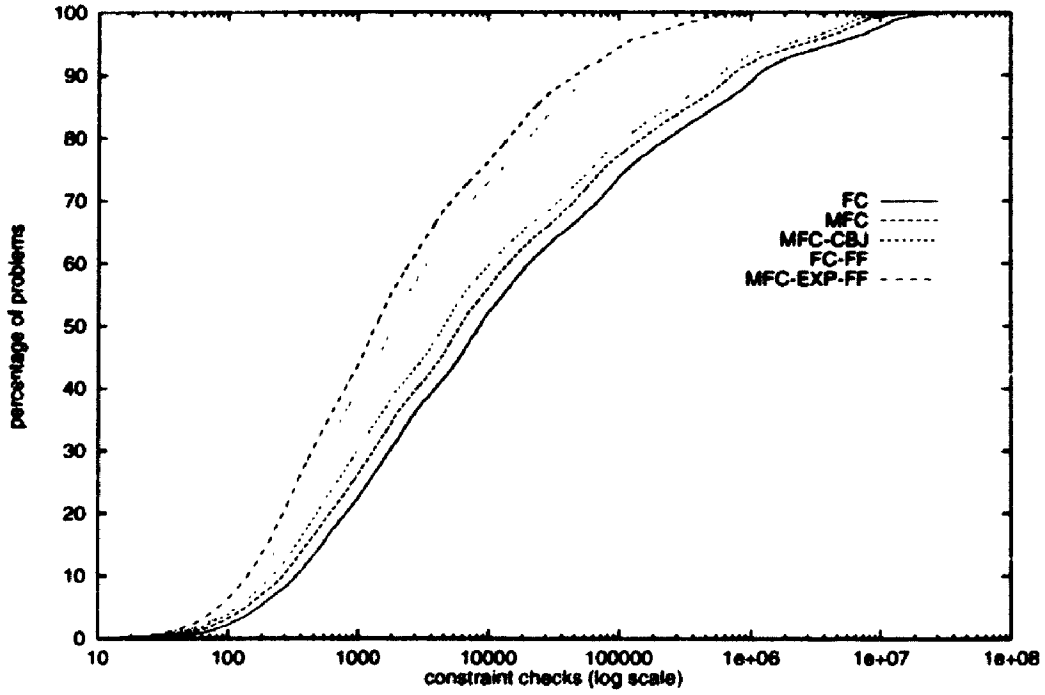

Figure 4.11: The number of constraint checks performed versus the number of problems solved at or before that number of constraint checks for problems with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. (10,200 problems)

n	10			15			20			25		
m	3	6	9	3	6	9	3	6	9	3	6	9
MFC	82 1	76.6	73 0	78 9	70 6	66 9	76 4	67 0	62 9	73 9	64 4	60 3
FC CBJ	91 4	90 8	91 1	77.9	85 4	87 4	65 7	80.9	83 7	55 0	73 0	79 6
MFC CBJ	73 4	68.5	65.9	59 7	59 4	58 0	48 8	53 5	52 4	39 3	46 4	47 8
FC FF	63 7	45.9	38.2	40 3	23.6	18 4	22 5	11 5	8 6	13 1	5 5	4 0
MFC FF	57 1	41 6	34.1	37 6	22.9	17.9	22 5	11 8	8 8	13 6	6 0	4 3
FC-CBJ FF	63.6	45.6	38.1	40 0	23 5	18.3	22 3	11 4	8 5	13 0	5 5	4 0
MFC-CBJ FF	55 3	40.7	33 5	35.6	22 2	17 6	20 8	11 4	8 5	12 3	5 7	4 2
MFC-EXP FF	53 6	36 6	29.7	33 2	17 6	13 4	18 0	8 2	5 9	9 9	3 8	2 6
MFC-CBJ EXP-FF	53.4	36.3	29.6	32 9	17 6	13 3	17 8	8 2	5 9	9 7	3 8	2 6
MFC-INC FF	55 2	36.9	29.8	34 3	18 1	13 6	18 8	8 5	6 1	10 5	4 0	2 8
MFC-CBJ INC-FF	55 2	36 7	29.7	34 0	18.0	13 5	18 7	8 5	6 1	10 3	4 0	2 7

Table 4.7: Percentage of FC's constraint checks performed by MFC, FC-CBJ, MFC-CBJ, FC-FF, MFC-FF, FC-CBJ-FF, MFC-CBJ-FF, MFC-EXP-FF, MFC-CBJ-EXP-FF, MFC-INC-FF, and MFC-CBJ-INC-FF broken down by n and m.

FF. Without CBJ, MFC is the best algorithm without FF, and MFC-EXP-FF is the best with FF. The improvement of almost all algorithms become better as n and m increase. The exceptions are FC-CBJ and MFC-CBJ which tend to perform worse as m increases for larger n but do perform better as n increases with fixed m. We believe that this phenomenon is being caused by a decrease in the ability to backjump. We find that our data confirms this hypothesis when we calculate the trend of average number of backjumps per attempted labeling. There is a decrease in the average number of backjumps per attempted labeling as m grows larger and there is an increase in the average number of backjumps per attempted labeling as n increases.

We expected that the addition of CBJ to MFC would save more constraint checks relative to the addition of CBJ to FC. Table 4.7 indicates some extra savings although it a slim relative increase in savings.

The experiments in this comparison are fairly extensive. The total number of

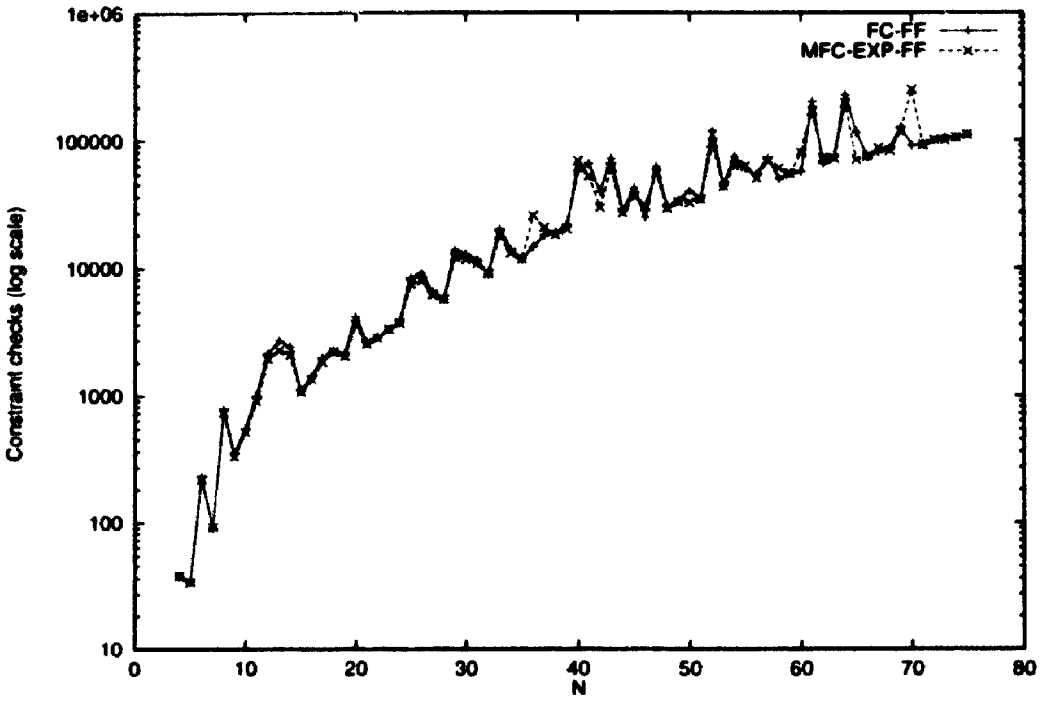

Figure 4.12: Comparison of FC-FF and MFC-EXP-FF by number of constraint checks performed on the n-queens problem.

problems solved by the algorithms without FF is 40,800. The total number of constraint checks performed was 20,911,132,413 with 5,031,340 seconds of CPU time. The total number of problems solved by the algorithms with FF is 81,600. The total number of constraint checks is 2,335,928,131 with 631,597 of CPU time. No valid comparisons can be drawn from these CPU times as the experiments were performed over 3 different machines (a Sun Sparc 5, 10, and 20).

4.3 A Final Look at the n-queens Problem

As mentioned in Section 2.6, the n-queens problem is the one well-known class of problems that we found on which MFC-FF can perform very poorly. Figure 4.12 provides a comparison of FC-FF and MFC-EXP-FF on the n-queens problem. This comparison shows that for the most part, MFC-EXP-FF overcomes the problem MFC-FF has with the FF heuristic on this particular class of problems. For all values of n

	No FF	FF
No CBJ	MFC	MFC-EXP-FF
CBJ	MFC-CBJ	MFC-CBJ-EXP-FF

Figure 4.13: The best algorithms as indicated by our experiments.

the two algorithms perform almost the same number of constraint checks except for a few points in which MFC-EXP-FF performs worse than FC-FF.

4.4 Conclusions

Figure 4.13 presents a synopsis of our results. Averaged over all problems in the testbed, MFC-CBJ-EXP-FF is the best of the 12 algorithms tested closely followed by MFC-EXP-FF. These two algorithms reduce the average number of constraint checks performed to just 14.1% of the constraint checks performed by FC and 75.4% of the constraint checks performed by FC-FF over the testbed. Bacchus and Grove's INC-FF heuristic (with and without CBJ) is a close third and fourth in terms of the average number of constraint checks performed. However, MFC-EXP-FF and MFC-CBJ-EXP-FF perform better than MFC-INC-FF and MFC-CBJ-INC-FF on approximately two thirds of the testbed problems. Given that the overhead involved in both heuristics is about the same we conclude that the EXP-FF heuristic is on average better than the INC-FF heuristic.

Of the algorithms not using FF, MFC-CBJ is the best. MFC-CBJ reduces the average number of constraint checks performed to just 53.8% of the constraint checks performed by FC. Without CBJ and FF, MFC is best, performing 30% fewer constraint checks than FC. Our results also imply that CBJ is useful for both the FC and MFC algorithms when not using the FF heuristic reducing the average number of constraint checks by 22.1% and 16.2%, respectively. Finally, adding CBJ to the algorithms with FF is apparently not as useful except in a few cases. Overall, we observe only a small change in the average number of constraint checks performed when adding CBJ to the algorithms using FF. The largest change occurs for MFC-FF

giving a 4.6% decrease in the average number of constraint checks. For sparse problems there is a larger change in the average number of constraint checks of 11.9%. Although our empirical comparison shows that CBJ is not very useful when the FF heuristic is used, we have been told informally[7] that CBJ can be critical in making a real world problem feasible to solve. We give a different comparison, of the algorithms compared in this chapter, in Chapter 6 using a different testbed.

[7] Private communication

Chapter 5

A New Model of Hard Binary Constraint Satisfaction Problems

"Re-make/Re-model" Bryan Ferry, Roxy Music

As discussed in Section 3.2, recent studies by Smith[102], Smith & Dyer[103], and Prosser[91, 94], have looked at the phase transition phenomenon exhibited by binary CSPs. They model binary CSPs using the 4-tuple $\langle n, m, p_1, p_2 \rangle$ where n is the number of variables, m is the domain size for all variables, p_1 is the probability that a non-trivial constraint exists between two variables, and p_2 is the "global"[1] probability, conditional on the existence of a non-trivial constraint, that a pair of values between the two variables is inconsistent. Smith[102] and Prosser[91] observe that the point where on average the hardest problems occur (the phase transition peak) co-occurs with the 50% point of solubility. Smith[102] also observes that this peak appears to co-occur very near to the point where the number of solutions to a problem is one. Using an expected number of solutions formula, developed by Haralick and Elliot[61] for CSPs using the above parameterization, Smith creates a predictor of the phase transition peak for binary CSPs by setting the formula to one. Smith[102], Smith & Dyer[103], and Prosser[91, 94] show empirically that the predictor, called $\hat{p}_{2_{crit}}$, is a reasonably good predictor of the location of a phase transition peak for *randomly generated binary CSPs*, especially as n grows. The predictor gives an over-estimation of the location of the phase transition peak, that is towards the right (insoluble) side. The one exception for Smith's predictor is for sparse graphs, that is, those CSPs with small values of p_1. Smith & Dyer[103] show that each individual constraint graph (given domain sizes for each variable) has its own location of the phase transition

[1] Called *global* here to emphasize that each constraint is created with the same constraint tightness.

peak. The location of the phase transition peak is highly variable for constraint graphs which are sparse. Smith & Dyer argue that the *local graph topology* needs to be incorporated into any predictor of the location of the phase transition peak for these problems. The local graph topology is defined to be the *degree distribution* of the constraint graph, that is, the degree of each individual node (variable) in the constraint graph.

The model that Smith[102], Smith & Dyer[103], and Prosser[91, 94] use for binary CSPs assumes that every domain has the same size and that every constraint has the same tightness. After [102, 91] were published, we decided to investigate a generalization of this model which allows for some variation in each constraint's tightness. Any generalization of Smith's model is useful as the new model will then be closer to a "real" CSP. In this chapter, the binary CSP parameterization given above is generalized to allow for a set of local constraint tightness values. We give a more refined version of Smith's predictor of the phase transition peak which is a set of constraint tightness values that incorporates the local graph topology around each constraint. Experiments show that there is a similar phase transition in which constraint tightness does not have to be a global value and that the refined predictor better predicts the location of the peak in this phase transition. We show that this phase transition peak which has CSPs with varying constraint tightnesses contains harder problems on average than the phase transition peak with global tightness values given a good algorithm for solving CSPs. We also show that random problems generated with the refined predictor are of similar or increased hardness to search by a good CSP algorithm than problems generated with the old predictor. We also show that the refined predictor better predicts the location of the phase transition peak for CSPs with sparse constraint graphs. Our results indicate that harder phase transition peaks can be found for NP-complete problems if the parameterizations used to model the problem are appropriately generalized to include the structure of an individual problem. The results of this chapter have been published in [31, 32, 35].

In Section 5.1 we review Smith's predictor of the phase transition peak. We then discuss our generalization of the binary CSP parameterization in Section 5.2 and give a

predictor of the phase transition peak for this generalization. In Section 5.3 we discuss how experiments were performed and in Section 5.4 we compare the two models.

5.1 The Global Constraint Tightness Predictor

In this section we give a short review of Smith's predictor of the location of the phase transition peak for binary CSPs. The intuition behind Smith's predictor is that problems which do not have a solution are over-constrained and that problems which have more than one solution are under-constrained. Problems that have one solution appear to be in the "middle ground" between solubility and insolubility. Smith conjectures that a phase transition peak for binary CSPs occurs near the region where the expected number of solutions for a problem is one. An expected number of solutions formula, Equation 5.1, derived by Haralick and Elliot [61], can be used to calculate the expected number of solutions of a CSP modeled with the above 4-tuple.

$$E(Soln) = m^n(1 - p_2)^{\frac{n(n-1)}{2}p_1} \tag{5.1}$$

The expected number of solutions is the number of possible tuples in a CSP multiplied by the probability of satisfying all the constraints.

The parameter p_2 partially determines how constrained a problem is and therefore seems to be a natural order parameter to vary. As mentioned in Section 3.2 there are phase transitions in binary CSPs for which it is appropriate to vary p_1 instead. Smith conjectures that a predictor of the phase transition, denoted $\hat{p}_{2crit}$ can be derived from (5.1) using E(Soln) = 1.

$$\hat{p}_{2crit} = 1 - m^{\frac{-2}{(n-1)p_1}} \tag{5.2}$$

Smith[102], Smith & Dyer[103], and Prosser[91, 94] show empirically that $\hat{p}_{2crit}$ is a good predictor of the phase transition peak in random binary CSPs with the given parameterization especially as n grows larger. The one exception for Smith's predictor is for sparse graphs, that is, those CSPs with small values of p_1. Henceforth, we call the model of binary CSPs used here and Smith's predictor of the phase transition peak for binary CSPs the global constraint tightness model (global model) of hard binary CSPs.

5.2 The Local Constraint Tightness Predictor

A model of binary CSPs using local constraint tightness is parameterized by a 4-tuple $\langle n, m, p_1, P \rangle$ where n is the number of variables, m is the global domain size, p_1 is the probability that a constraint exists between two variables, and P is a set $\{p_{2_{i,j}} | 1 \leq i < j \leq n\}$ of local constraint tightness probabilities conditional on the existence of the corresponding constraint. That is, $p_{2_{i,j}}$ is the probability that any pair of values between variables v_i and v_j is inconsistent given that a constraint exists between v_i and v_j. An expected number of solutions formula for a binary CSP using the new parameterization is:

$$E(\text{Soln}) = m^n \prod_{1 \leq i < j \leq n} (1 - p_{2_{i,j}}) \ . \tag{5.3}$$

Again, the expected number of solutions is the number of possible tuples in a CSP multiplied by the probability of satisfying all the constraints. Here, each constraint is considered independently. One would like a predictor similar to Smith's for this model of binary CSPs. In the global constraint tightness model, the predictor gives the looseness of each constraint that exists as:

$$1 - \hat{p}_{2_{GT}} = m^{-2/(n-1)p_1} = \frac{1}{m^{\frac{1}{(n-1)p_1}} m^{\frac{1}{(n-1)p_1}}} \ . \tag{5.4}$$

An important observation is that the expected degree of each variable (node in the constraint graph) is $(n-1)p_1$. Smith's predictor assumes that the looseness of each constraint is a proportion of the (fixed) domain sizes of the two variables according to their expected degree. We suggest that one can refine Smith's predictor of the phase transition peak *for each problem* by replacing the expected degree with the actual degree of each variable. That is, the predictor of a phase transition peak for a binary CSP under the new model is the set

$$p_{2_{i,j}} = \begin{cases} 0 & \textit{if there is no constraint between } i \textit{ and } j \\ 1 - \frac{1}{m^{\frac{1}{a_i}} m^{\frac{1}{a_j}}} & \textit{otherwise} \end{cases} \tag{5.5}$$

where a_i and a_j are the degrees of variables v_i and v_j. That this setting preserves the expected number of solutions being one is not immediately obvious. To show

that it does. we give the following general theorem which allows not only for varying constraint tightnesses but also for different sizes of domain values (where m_i is the domain size of variable v_i).

Theorem 5.1 *If* a_i *is the degree of* v_i *and* a_j *is the degree of* v_j *then if*

$$p_{2_{i,j}} = \begin{cases} 0 & \text{if there is no constraint between } i \text{ and } j \\ 1 - \frac{1}{m_i^{\frac{1}{a_i}} m_j^{\frac{1}{a_j}}} & \text{otherwise} \end{cases}$$

then

$$E(Soln) = \prod_{1 \le i \le n} m_i \prod_{1 \le i < j \le n} (1 - p_{2_{i,j}}) = 1 \ . \tag{5.6}$$

Proof of Theorem 5.1 We first note that

$$1 - p_{2_{i,j}} = \frac{1}{m_i^{\frac{1}{a_i}} m_j^{\frac{1}{a_j}}} \tag{5.7}$$

for the constraints that exist given that we are proportioning the domains m_i and m_j for variables v_i and v_j. We then note that each $m_i^{\frac{1}{a_i}}$ occurs a_i times in

$$\prod_{1 \le i < j \le n} \frac{1}{m_i^{\frac{1}{a_i}} m_j^{\frac{1}{a_j}}} \tag{5.8}$$

as v_i participates in a_i constraints ($1 \le i \le n$). This implies the product in (5.8) is

$$\frac{1}{\prod_{1 \le i \le n} m_i} \ . \tag{5.9}$$

Therefore $E(Soln) = 1$. □

We conjecture that the settings of $p_{2_{i,j}}$ given by (5.5) are a predictor of a phase transition peak for the new model as $\hat{p}_{2_{crit}}$ is a predictor for the global model (although now we have a set of predictors instead of just one). We also conjecture that if the global model's predictor accurately predicts the location of a phase transition for a real problem, then the local model's predictor would also predict the location of a phase transition. Henceforth, we call this model and predictor of the phase transition peak for binary CSPs the local constraint tightness model (local model) of hard binary CSPs. Although Theorem 5.1 is proven for a more general case where domain sizes

are allowed to be different, we focus on binary CSPs that have domain sizes all the same. At this time whether or not a phase transition exist for a problem with varying m_i's and whether or not this model is adequate to predict phase transition peaks are left as future work.

The global model and the local model (of hard random problems)[2] are closely related. When all variables (nodes) have the same degree, for instance a complete graph, the two models are the same. Many of the results presented in this chapter have this special case removed from the data because the two models do not differ. The global model and the local model also have counterparts for n-ary CSPs which we intend to explore in the future.

5.3 Experiments

In the experiments that are described below we use the FC-CBJ-FF algorithm[3]. We use FC-CBJ-FF, instead of FC-FF as in [102, 103], as its performance seems to be better than FC-FF for problems with sparse graphs. Otherwise FC-CBJ-FF's performance is nearly the same as FC-FF. If there is a real difference between the two models we expect that FC-CBJ-FF will show this difference. Backtracking algorithms not using some type of lookahead and the FF heuristic may not show a real difference between the models as they are easily fooled into poor performance. As before, a generated problem's difficulty is measured by the number of constraint checks performed by FC-CBJ-FF in attempting to solve the problem.

Our experiments use a testbed of 12,750 connected randomly generated graphs consisting of 50 connected graphs for each combination of n, m, and p_1 with $n \in \{10, 15, 20, 25, 30\}$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. This testbed of randomly generated graphs is an extension of the testbed described in Section 3.3 by

[2]Henceforth the phrase "of hard random problems" is omitted from now on if the context allows it. References to the global or local model refer to random problems generated under the model using its respective predictor.

[3]These experiments were performed before the experiments comparing the different algorithms in Chapters 3 and 4. We used the best known algorithm that we knew of at that time.

adding problems with $n = 30$.

Three comparisons are made: a comparison between the phase transitions under the two models of binary CSPs, the respective comparisons between the phase transition and how well the model's respective predictor predicted the phase transition peak, and a comparison of problems randomly generated under the two models of hard problems. It is essential that the difference in comparisons be caused only by the values chosen for the constraint tightnesses. With that in mind, throughout the various comparisons, we record for each set of 50 randomly generated graphs a random start-seed that is used as a starting point in creating all CSPs from that set. We only use the random number generator for randomizing the enumeration of the cross-product of two domains connected by a constraint. To create a constraint, we take the first $p_2 m^2$ as unacceptable pairs for the global model, the first $p_{2_{i,j}} m^2$ for the local model, and the respective amounts needed to find the phase transition under the two models of binary CSPs (described below). All CSPs created in this manner are very similar in that they only differ in the number of unacceptable pairs chosen from the *same* randomized enumerations.

The experiments to find the phase transition for problems under the two binary CSP models are limited to the graphs generated for $n = 20$, $m = 9$, and $p_1 \in \{0.2, 0.25, \ldots, 0.9\}$. This subset should be large enough to be representative of the rest of the graphs. We are limited in what we can do as it is computationally expensive to find the phase transitions. To find the phase transition for binary CSPs with a global constraint tightness we vary p_2 from 0.01 to 0.99 and create a CSP instance from each graph in the set of 50. A finer resolution for p_2 is not possible as the largest domain size is 9 meaning that a difference of 0.01 for p_2 changes the number of of inconsistent tuples by at most one. Finding the phase transition for binary CSPs with local constraint tightness values is more difficult. There are many settings for the initial values of all the $p_{2_{i,j}}$'s that may lead to a phase transition, one of which is all zeros which would just lead to the phase transition for binary CSPs with a global tightness. However, we would like to show that there is a phase transition similar to the phase transition for binary CSPs with the global constraint tightness values. For

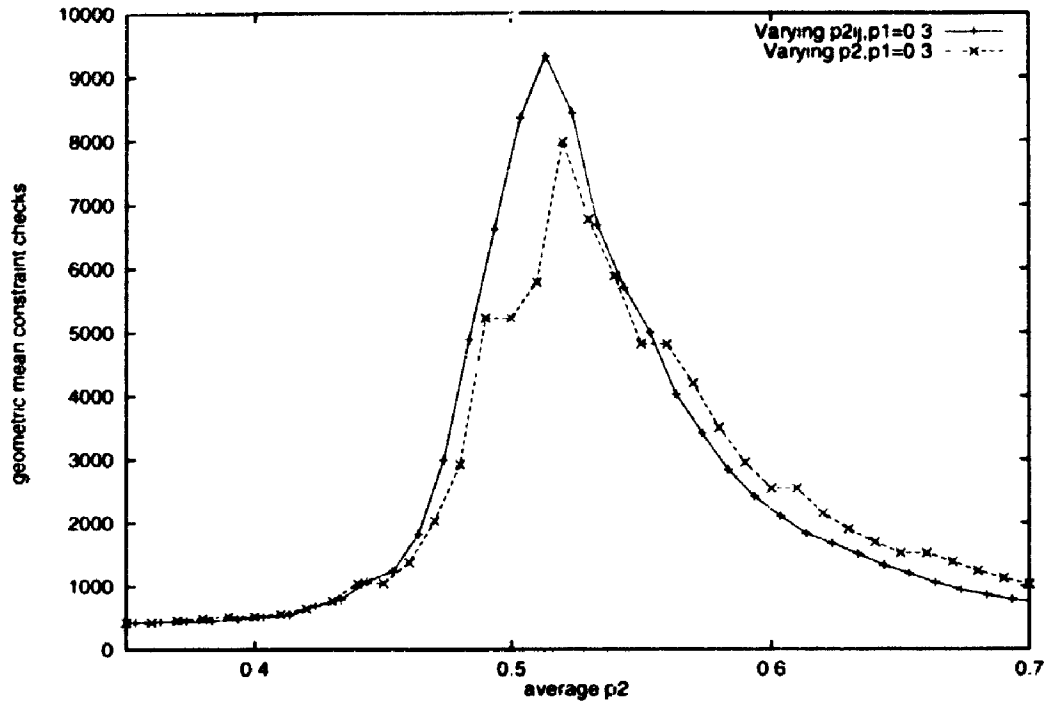

Figure 5.1: Comparison of the transition phases under the two models for $n = 20$, $m = 9$, and $p_1 = 0.3$.

each set of 50 graphs the smallest $p_{2_{i,j}}$ is found. We then subtract this smallest value from every $p_{2_{i,j}}$ created and add an increment which varies from 0.01 to 0.99. If there is a phase transition peak at the point that the predictor suggests, this should show it.

For the experiments comparing the two models of hard binary CSPs, CSPs using the p_2 ($p_{2_{i,j}}$) values as given by the two predictors are used to create randomly generated problems.

5.4 Comparison of the Global and Local Model

We begin our comparison by showing in Fig. 5.1 the phase transition for the two binary CSP models for $n = 20$, $m = 9$, and $p_1 = 0.3$. The graph for the global parameterization is labeled "Varying p2" and the graph for the local parameterization is labeled "Varying p2ij". As it is necessary to compare the two graphs along

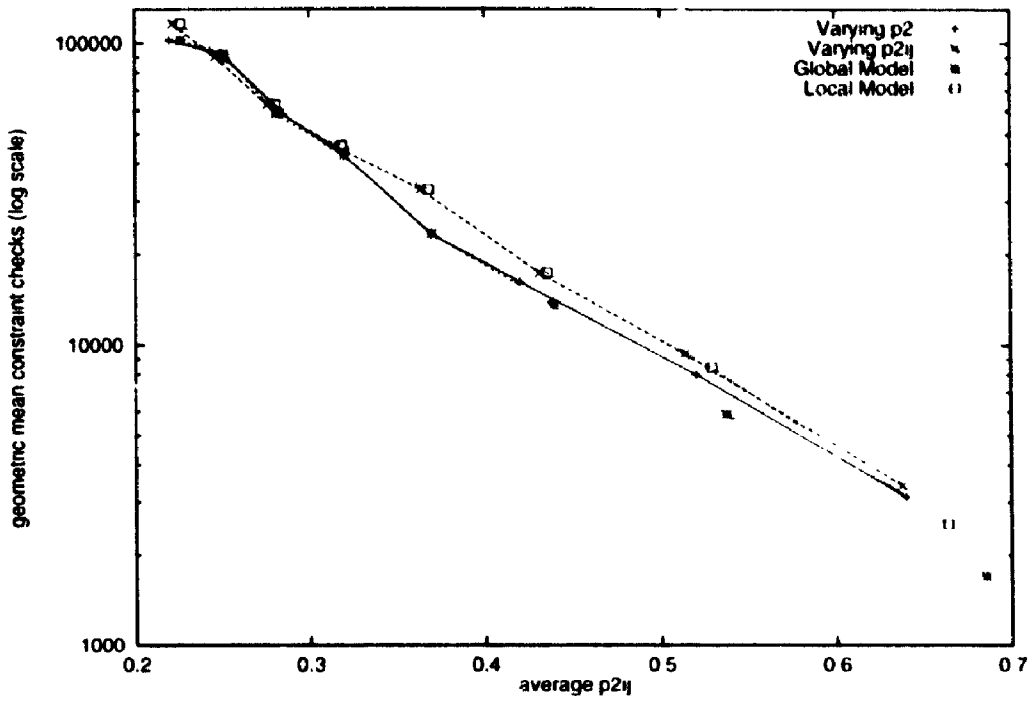

Figure 5.2: Comparison of the transition peaks and the global and local models for $n = 20$, $m = 9$, and $p_1 \in \{0.2, 0.3, \ldots, 0.9\}$.

the same axis we use the average of the $p_{2_{i,j}}$ values as a "global" average p_2 for the local parameterization. The graph shows that both parameterizations lead to a phase transition, the local parameterization having a higher peak. We cannot show all the phase transition graphs but a summary of their peaks is presented in Fig. 5.2. Four lines are displayed in Fig. 5.2. The first line labeled "Varying p2" shows the geometric mean of the transition peaks of the global parameterization for $p_1 \in \{0.2, 0.3, \ldots, 0.9\}$, the second line labeled "Varying p2ij" shows the transition peaks for the local parameterization, the third line labeled "Global Model" shows the peaks predicted by the global model, and the fourth line labeled "Local Model" shows the peaks predicted by the local model. The x-axis is the average $p_{2_{i,j}}$ value for the local model and for the phase transition peaks of the local parameterization, the value of $\hat{p}_{2_{crit}}$ for the global model, and the location of the phase transition peaks of the global parameterization.

The points for $p_1 = 0.2$ begin at the far right and move left for increasing p_1 as sparse problems are much easier to solve than more connected problems. The peaks for the local parameterization are much higher than those of the global parameterization, that is, random problems generated with local constraint tightness values are much harder than those generated with a global constraint tightness value. All the lines join together around $p_1 = 0.7$ and above which is to be expected as the two models become more similar as the graphs become more connected (there is less variation in local graph topology). The local model better predicts the peak of the phase transition for the local parameterization, effectively becoming the same after $p_1 = 0.3$. The global model does not accurately predict the peak of the phase transition for the global parameterization until $p_1 = 0.5$. For the sparser graphs with $p_1 = 0.2$ and 0.3 the local model predicts the phase transition better than the global model although there is still a gap for $p_1 = 0.2$.

Next, the global and local models are compared to see if there is a real difference in the hardness of problems randomly generated using the two models. Fig 5.3 shows a comparison of the global model and the local model for $n \in \{20, 25, 30\}$ and $m = 9$ broken down by p_1. For increasingly larger problems, the local model produces much harder problems than the global model. The separation is especially large for lower values of p_1. Fig. 5.4 shows a scatter plot of the same problems. Although there are a few outliers favouring the global model, the majority of the problems (75%) are harder when generated with the local model. Fig. 5.5 shows a comparison of the global model and the local model for $n = 30$ and $m \in \{3, 6, 9\}$ broken down by p_1. It appears that the local model produces much harder problems than the global model for increasing m especially for problems with lower values of p_1. These observations continue to hold if one uses the average or the median.

Next we display in Figure 5.6 an overall scatter plot for all the problems (problems with $p_1 = 1$ are omitted). The local model produces a majority of the hard problems. Excluding those problems with $p_1 = 1$, the global model produces a problem harder than the local model 31% of the time, the local model produces a problem harder than the global model 58% of the time, and 11% of the time the problems have the same

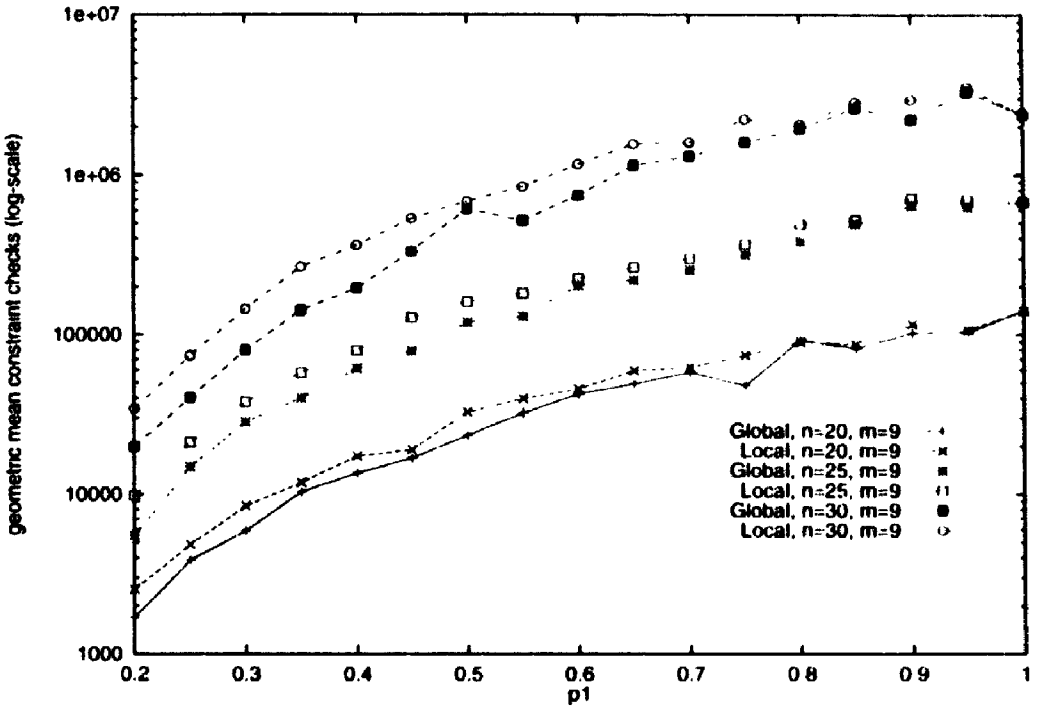

Figure 5.3: Comparison of the global model and the local model by the geometric mean number of constraint checks performed by FC-CBJ-FF broken down by p_1, $n \in \{20, 25, 30\}$, $m = 9$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$.

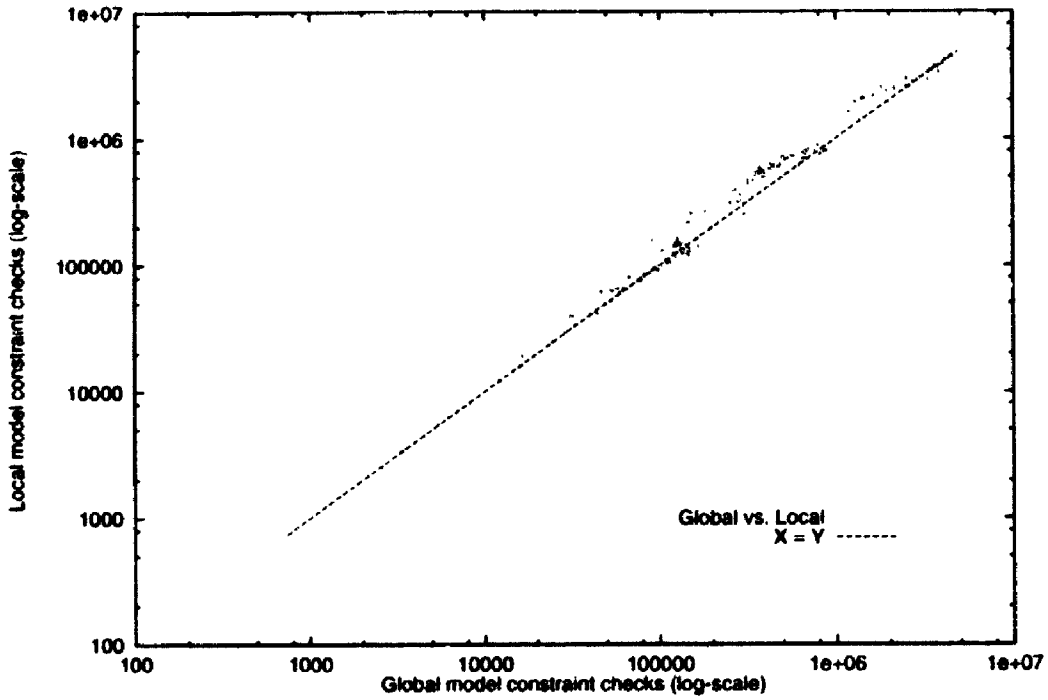

Figure 5.4: Comparison of the global model and the local model by number of constraint checks performed on each problem by FC-CBJ-FF. $n \in \{20, 25, 30\}$, $m = 9$. Problems with $p_1 = 1.0$ are omitted. (2,400 problems).

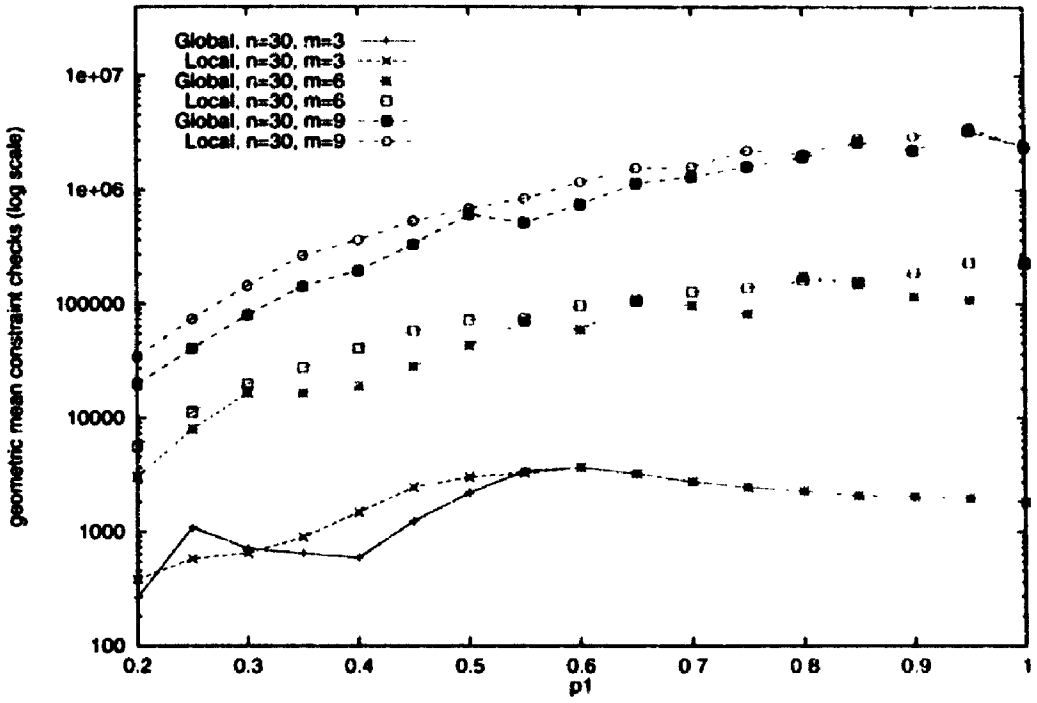

Figure 5.5: Comparison of the global model and the local model by the geometric mean number of constraint checks performed by FC-CBJ-FF broken down by p_1, $n = 30$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$.

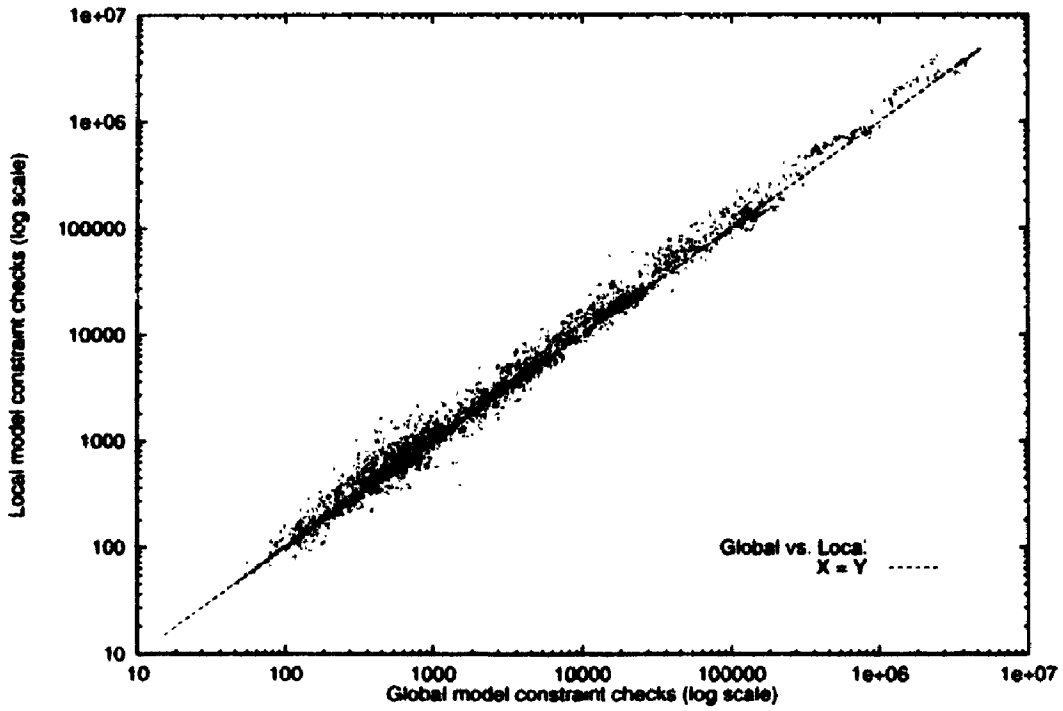

Figure 5.6: Comparison of the global model and the local model by number of constraint checks performed on each problem by FC-CBJ-FF. $n \in \{10, 15, 20, 25, 30\}$, $m \in \{3, 6, 9\}$. Problems with $p_1 = 1$ are omitted (12,000 problems).

n	the global model harder	the local model harder	Same
10	43%	57%	0.4%
15	36%	64%	0.0%
20	34%	66%	0.0%
25	22%	78%	0.0%
30	18%	82%	0.0%

Table 5.1: Percentage of times one model produces harder problems than the other for problems with $n \in \{10, 15, 20, 25, 30\}$ and $m = 9$. Problems with $p_1 = 1$ are omitted.

hardness.

Table 5.1 shows the percentage of times one model is harder than the other for problems with $m = 9$. As n increases, the local model becomes much better than the global model at producing hard problems.

Next we investigate the trends observed between the models for increasing n and m and for lower values of p_1. Table 5.2 shows by geometric mean how much harder the problems generated by the local model are than those generated by the global model. The local model is producing on average much harder problems as n increases, and for problems with $p_1 \leq 0.5$ the local model produces much harder problems especially for large m. The local model's use of local graph topology becomes increasingly important as n and m grow larger. The separation of the two models as n grows larger is contrary to Smith & Dyer's[103] assertion that for larger values of n the effect of different constraint graph topologies will be less important.

Table 5.2 shows the largest percent increase in geometric mean constraint checks for the problems with large m and $p_1 \leq 0.5$. In Figure 5.7 we display a scatter plot comparing the random problems generated under the two models for all values of n in the testbed, $m = 9$, and $p_1 \leq 0.5$. The local model is clearly superior for these problems which have large domain sizes and sparser constraint graphs.

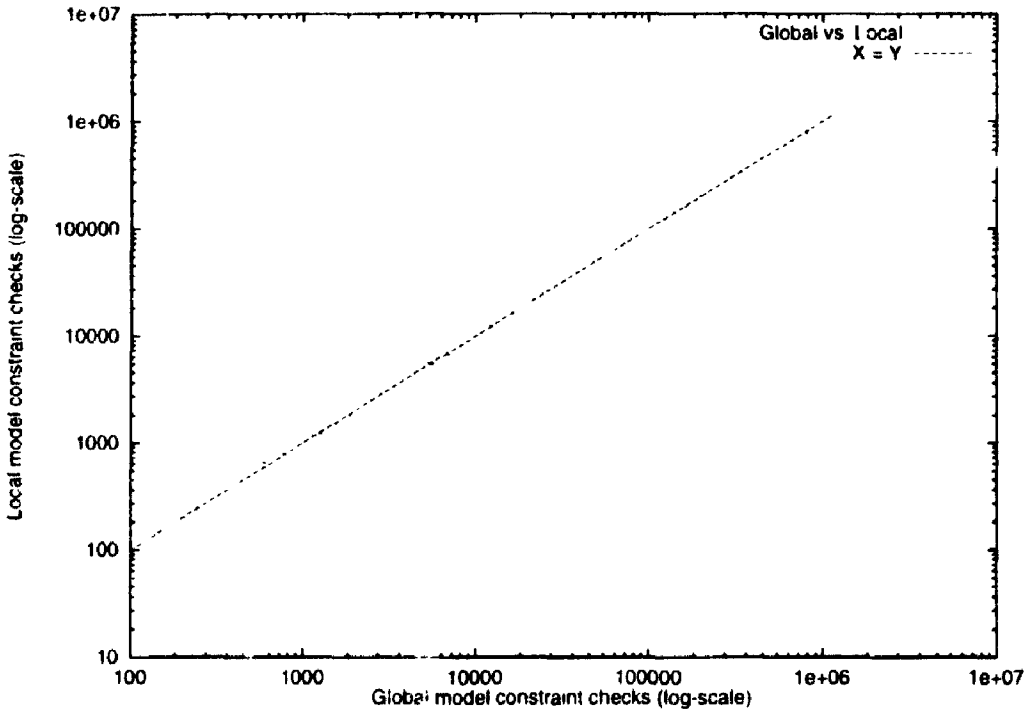

Figure 5.7: Comparison of the global model and the local model by number of constraint checks performed on each problem by FC-CBJ-FF. $n \in \{10, 15, 20, 25, 30\}$. $m = 9$. $p_1 \leq 0.5$. (1,500 problems).

	$p_2 \leq 0.95$				$p_1 \leq 0.5$			
	m = 3	m = 6	m = 9	all m	m = 3	m = 6	m = 9	all m
n = 10	4.6%	−0.4%	−2.1%	0.7%	5.2%	−2.3%	−1.5%	0.4%
n = 15	7.3%	5.2%	5.8%	6.1%	3.4%	4.0%	7.8%	5.1%
n = 20	14.6%	20.9%	20.5%	18.6%	7.0%	18.4%	29.8%	18.1%
n = 25	10.6%	25.9%	28.9%	21.6%	13.9%	16.8%	45.4%	24.6%
n = 30	12.4%	47.6%	44.0%	33.7%	31.1%	68.7%	66.4%	54.4%
all n	9.8%	18.7%	18.3%	15.5%	11.7%	18.8%	27.2%	19.1%

Table 5.2: Percent increase in geometric mean number of constraint checks performed by FC-CBJ-FF for the local model over the global model broken down by $m \in \{3, 6, 9\}$ for problems with $p_1 \leq 0.95$ and for problems with $p_1 \leq 0.5$.

The phase transition peak is associated with the 50% point in solubility. However, the global model usually gives a prediction that is a little to the right (that is, towards the insoluble side) of the actual phase transition peak. We find that both models produce similar proportions of soluble to insoluble problems (22% soluble, 78% insoluble), agreeing with Smith and Prosser's results which show the global model's prediction is to the right of the transition point.

The global and local model predict that a phase transition peak occurs near to the point where the expected number of solutions is one. To see if randomly generated problems created using the local and global models have close to one solution we calculate the average number of solutions by running FC-FF to find all solutions on the problems with $n \in \{10, 15, 20\}$. We do not consider the larger problems as finding all solutions would be too costly in terms of time. A table of the average number of solutions broken down by n and m is displayed in Table 5.3. The titles G-soln and L-soln stand for the global model and the local model with soluble problems only, and the title All-3 stands for the average number of solutions for all of the problems disregarding those with $m = 3$. The title std stands for the standard deviation. In

most instances, randomly generated problems using both models have more than the expected number of solutions although problems generated with the local model have fewer average number of solutions and a smaller standard deviation than problems generated with the global model. Both models produce problems with too many solutions for $m = 3$ although the local model is much closer to the expected number of solutions. For soluble problems with larger values of m, the local model is closer to the expected number of solutions with a smaller standard deviation. Over all the problems examined in Table 5.3, the global model produces problems with an average number of solutions of 15.89 (65.31 for soluble problems) versus 3.33 (13.18 for soluble problems) for the local model. However, if we remove the problems with $m = 3$ which both models seem to have difficulty with, the global model produces problems with an average number of solutions of 1.49 (6.95 for soluble problems) versus 1.31 (5.25 for soluble problems) for the local model. The local model overall produces problems that are slightly closer to the expected number of solutions for larger values of m.

We next count how often a particular number of solutions occurs for the problems examined in Table 5.3. We do not include insoluble problems the global model has 5789 insoluble problems and the local model has 5715 insoluble problems. Figures 5.8 and 5.9 show a comparison of the frequency of a particular number of solutions. The x-axis represents the number of solutions, and the y-axis, the number of problems that have that number of solutions. The local model has more problems with 1 to 10 solutions. The global model has more problems with 100 to 1000 solutions. The global model has some problems with more than 1000 solutions whereas the local model has only one. The global model has 69.7% of of its soluble problems with 10 or fewer solutions and the local model has 84.7% of its soluble problems with 10 or fewer solutions. We also include a point on each graph showing the average number of solutions versus the average number of problems. The local model produces on average more problems with fewer number of solutions than the global model and it produces far fewer "outliers".

Finally, we calculate how varied the values of $p_{2_{i,j}}$ are. We are most interested in how different are the values of the $p_{2_{i,j}}$ in the local model from the value of $\hat{p}_{2crit}$ for

		Global		Local		G-soln		L-soln	
n	m	ave	std	ave	std	ave	std	ave	std
10	3	1.88	5.60	1.33	3.65	6.50	8.87	4.44	5.55
	6	1.42	4.60	1.42	3.82	5.07	7.55	4.30	5.65
	9	1.41	4.24	1.60	4.70	4.82	6.72	4.64	7.08
	All	1.57	4.85	1.45	4.08	5.47	7.79	4.46	6.16
15	3	17.46	56.67	7.26	28.77	58.88	91.74	29.24	51.97
	6	1.09	6.09	1.11	3.86	6.46	13.60	4.92	6.89
	9	1.14	5.83	1.13	4.24	5.47	11.82	4.50	7.52
	All	6.56	33.95	3.17	17.18	29.21	66.89	13.09	33.02
20	3	114.70	517.71	13.59	83.38	362.45	871.18	57.19	163.89
	6	2.86	20.11	1.64	15.91	17.61	47.37	11.24	40.47
	9	1.03	4.43	0.93	3.58	5.89	9.19	4.81	6.92
	All	39.53	303.71	5.39	49.37	181.62	631.31	28.03	109.87
All	3	44.68	304.70	7.39	51.19	148.55	541.76	28.26	97.16
	6	1.79	12.44	1.39	9.70	8.77	26.39	5.94	19.38
	9	1.19	4.88	1.22	4.21	5.30	9.19	4.64	7.17
	All	15.89	177.24	3.33	30.31	65.31	354.91	13.18	59.20
	All-3	1.49	9 45	1.31	7.48	6.95	19.46	5.25	14.30

Table 5.3: Average number of solutions and the standard deviation for various problem sets. Column Global is the global model, column Local is the local model. Columns G-soln and L-soln describe the average number of solutions for soluble problems only.

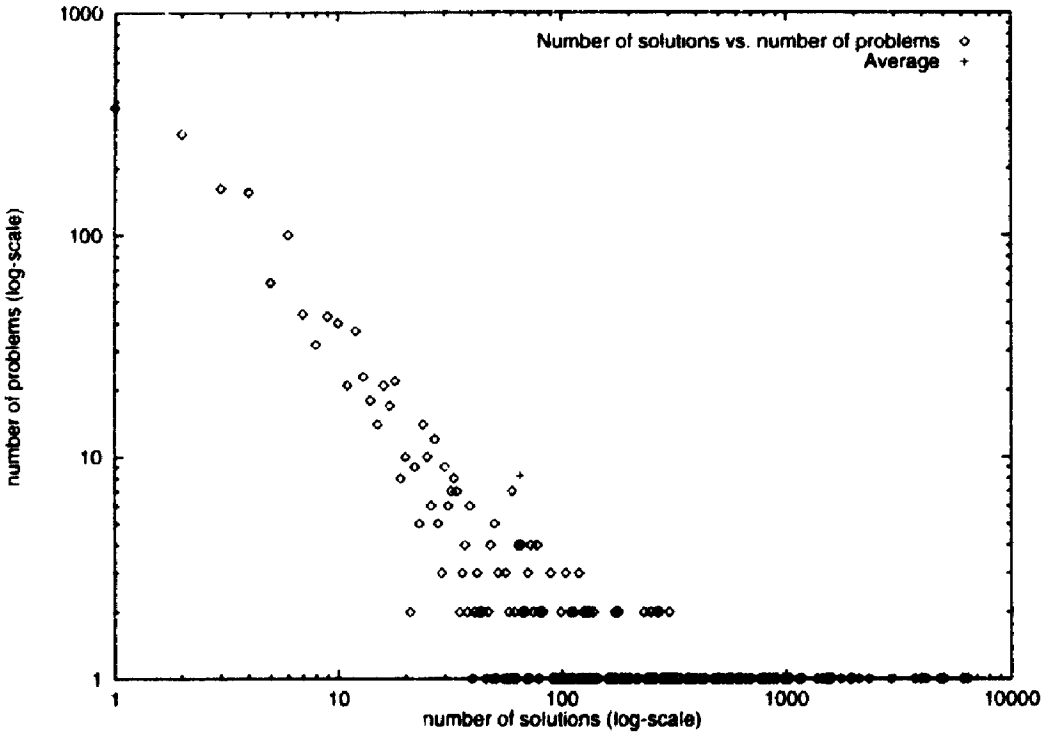

Figure 5.8: Comparison of number of solutions to number of problems with that number of solutions for the global model. problems with $n \in \{10, 15, 20\}$ and $m \in \{3, 6, 9\}$.

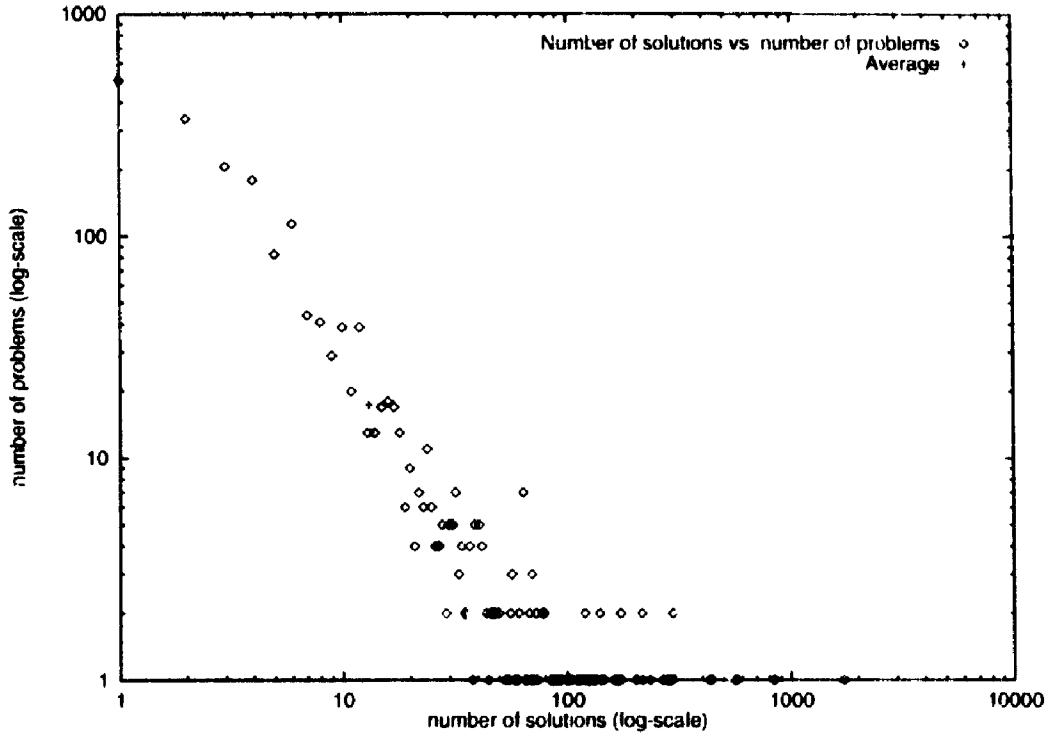

Figure 5.9: Comparison of number of solutions to number of problems with that number of solutions for the local model, problems with $n \in \{10, 15, 20\}$ and $m \in \{3, 6, 9\}$.

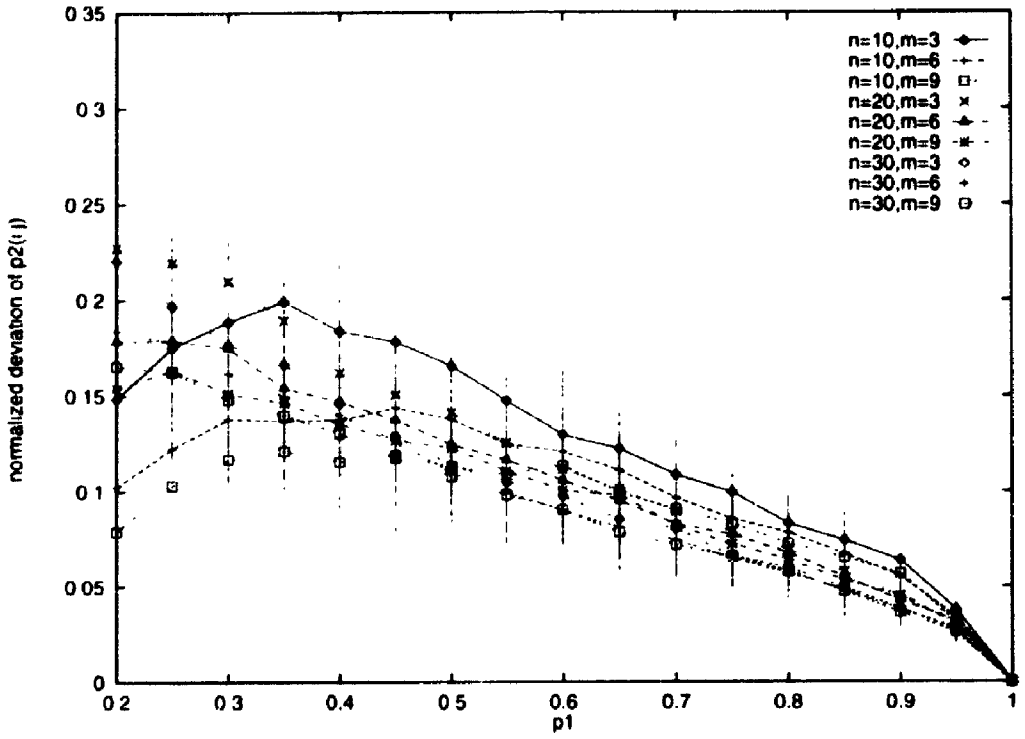

Figure 5.10: The average normalized deviation of $p_{2_{i,j}}$ broken down by p_1 and a scatter plot of all the normalized deviations for problems with $n \in \{10, 20, 30\}$ and $m \in \{3, 6, 9\}$.

the global model. For each problem, the deviation of $p_{2_{i,j}}$ from $\hat{p}_{2crit}$ is calculated. The deviation is then normalized by dividing by $\hat{p}_{2crit}$ to give the percentage of deviation over the mean. These averages are then averaged for each value of p_1. Figure 5.10 shows the average normalized deviation of $p_{2_{i,j}}$ for the problems in $n \in \{10, 20, 30\}$ and $m \in \{3, 6, 9\}$ broken down by p_1 and a scatter plot of all the normalized deviations. As an example of the variation between the values of $p_{2_{i,j}}$ and $\hat{p}_{2crit}$, consider the problems at $n = 20$, $m = 9$, $p_1 = 0.2$ which appear to be in the middle of the normalized deviations shown in the graph. The normalized deviation is 0.1539, and the value of $\hat{p}_{2crit} = 0.6854$. Therefore the deviation is $0.1539 * 0.6854 = 0.1055$. Assuming that the values are normally distributed, values of $p_{2_{i,j}}$ will fall in the range $0.6854 \pm 2 * 0.1055 = 0.6854 \pm 0.211$ (0.4744 . . . 0.8964) 95% of the time.

Figure 5.10 shows that the highest variation in $p_{2_{i,j}}$ occurs for small values of p_1,

gradually approaching no variation at $p_1 = 1$. Between $p_1 = 0.2$ and $p_1 = 0.6$ the average normalized deviation seems to be 10% or more from the value of $\hat{p}_{2crit}$. The normalized deviation is at best around 23% from the value of $\hat{p}_{2crit}$. At no point is the average normalized deviation equal to 0 except at $p_1 = 1$. It is interesting to note that even at $p_1 = 0.95$, there is still a 3% average normalized deviation which implies that even for nearly complete graphs the local graph topology is still important.

These experiments have been performed on a Sun Sparc-10 using code written in CMU-Lisp. FC-CBJ-FF uses a total of 291.088 CPU seconds to perform a total of 1,533,685,080 constraint checks for the global model and a total of 363.231 CPU seconds to perform a total of 1,803,542,793 constraint checks for the local model. A substantial amount of unreported time is used for the comparison of phase transitions under the two models of binary CSPs.

5.5 Summary

In this chapter we give a generalization of the standard model for binary CSPs which allows for local constraint tightness values. Our experiments show that there is a phase transition in this new generalization. A predictor of the phase transition peak is derived for this new generalization that includes the local graph topology of an individual problem. We have empirically shown that the new predictor accurately predicts the phase transition peak for this new generalization except for very sparse problems. Although the new predictor does not accurately predict the phase transition peak for very sparse problems with $p_1 < 0.3$, its prediction appears to be much better than Smith's predictor. Finally, we have compared random problems generated with the two predictors. The problems generated using the new local model are of similar or increased hardness than problems generated using the global model. It would seem reasonable that harder phase transition peaks may be found for other NP-complete problems by generalizing the parameterizations used to model the problems in appropriate ways to include the structure of an individual problem.

There are two main benefits of the work presented in this chapter. The first is that

we have derived a new predictor of a phase transition peak for a new more general model of binary CSPs that allows some variation in each constraint tightness. The second is that the new predictor can be used to create a testbed of problems that are of similar or increased hardness and have more variation in constraint tightness than those created using the global predictor. These problems may be more like real problems as they have a more complex structure.

We believe that our new model provides the foundation for a new model that allows varying domain sizes but we leave that as future work.

Smith, Smith & Dyer, and Prosser do not show that the predictor $\hat{p}_{2crit}$ accurately predicts the location of phase transition peaks for "real" problems. It is conjectured that if a real problem can be modeled using the given parameterization then Smith's predictor will accurately predict whether that problem will be near a phase transition peak. We also do not investigate the accuracy of Smith's predictor or the predictor for our new model for real problems. Such a task is left for future work.

Chapter 6

An Empirical Comparison Using the Local Model of Hard Random Problems

In Chapter 4 a large empirical comparison of a number of algorithms is presented using the global model of hard random problems. It is shown that the best algorithms are all based on MFC, namely MFC-CBJ-EXP-FF, MFC-EXP-FF, MFC-CBJ, and MFC. Each algorithm is best under a different set of circumstances, namely, whether or not the problem lends itself to the use of the FF heuristic and whether or not it is considered worthwhile to add non-chronological backtracking (CBJ). In Chapter 5 a new model of binary CSPs is proposed which generalizes the current model by allowing constraint tightness to vary. We develop a predictor of the location of phase transition peaks in the new model and show empirically that the new predictor accurately predicts the location of phase transition peaks for randomly generated problems. Problems randomly generated under the new model of binary CSPs at the predicted location of the phase transition peak (called the local model of hard random problems) are harder to solve "on average" then problems randomly generated under the old model (called the global model of hard random problems[1]). The local model of hard random problems produces harder problems for larger values of n and m especially for sparser graphs with $p_1 \leq 0.5$.

In this chapter we further test the algorithms in order to assess the robustness of our claims in Chapter 4. We revisit the empirical comparison performed in Chapter 4 using the local model instead of the global model. In addition to comparing the algorithms over the complete testbed, we focus on problems with sparser graphs ($p_1 < 0.5$) and

[1] We shorten "local/global model of hard random problems" to the "local/global model", respectively, when the context implies that we are discussing problems randomly generated under the specified model using the respective predictor of phase transition peaks.

larger values of n and m, the region of problems which can best exhibit any new differences between algorithms brought to light by the local model. These empirical comparisons of algorithms over both models also allows us to compare the two models of hard random problems over many algorithms. The comparison is presented in Section 6.1 below, followed by a summary of the comparison in Section 6.2.

6.1 An Empirical Comparison Using the Local Model

In this section we empirically compare the 12 algorithms, FC, MFC, FC-CBJ, MFC-CBJ, FC-CBJ-FF, MFC-CBJ-FF, MFC-EXP-FF, MFC-CBJ-EXP-FF, MFC-INC-FF, and MFC-CBJ-INC-FF using the local model of hard random problems. To perform this comparison we reuse the constraint graphs randomly generated for the testbed in Section 3.3. We create 50 randomly generated CSPs using the local model's predictor for each combination of n, m, and p_1 with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. Each specific problem in the "local model testbed" differs from its counterpart in the "global model testbed" only by the predictor used and not by any other external factors such as different randomizations[2]. Therefore, any differences in the performance of algorithms is caused only by the difference in the models of hard random problems and not by any other effect.

As in Section 4.2, we run each algorithm on each problem in our testbed counting the number of constraint checks performed in solving each random problem.

We begin by showing the overall results of the experiments in Tables 6.1 and 6.3. Tables 6.2 and 6.4 which show the results of the experiments using the global model are repeated from Chapter 4 (Tables 4.1 and 4.2) for comparison. Tables 6.1 and 6.2 show the percentage of times one algorithm performs better than another by the number of constraint checks for the local and global testbeds, respectively. Tables 6.3 and 6.4 give the average, the standard deviation for the average, the median, the geometric mean, the maximum, the minimum, and the percentage of the number of constraint checks by geometric mean performed relative to FC by the 12 algorithms for the local

[2]See the discussion in Section 5.3 for an explanation of how the local model testbed is generated.

and global testbeds, respectively.

The results displayed in Table 6.1 and 6.2 are very similar. The general ordering of the best 4 algorithms using FF remains the same as in the experiments using the global model, that is, from best to worst MFC-CBJ-EXP-FF > MFC-EXP-FF > MFC-CBJ-INC-FF > MFC-INC-FF. However, the poor performance of MFC-FF becomes more pronounced in Table 6.1. In the global model, MFC-FF performs better than FC-CBJ-FF and FC-FF. In the local model the ordering is changed and MFC-FF is the worst of the algorithms using FF. There is a similar decline in performance for MFC-CBJ-FF as compared to FC-FF and FC-CBJ-FF, however MFC-CBJ-FF remains a better algorithm. The ordering of the algorithms not using FF is the same in the experiments using the local model as the experiments using the global model. That is, the algorithms performance from best to worst is MFC-CBJ > MFC > FC-CBJ > FC.

A comparison of Tables 6.3 and 6.4 gives a similar picture. The ordering of the best 4 algorithms using FF remains the same when using the local model. The ordering of the algorithms not using FF also remains the same. Again in the local model MFC-FF and MFC-CBJ-FF perform worse than in the global model.

Tables 6.3 and 6.4 also show something interesting about the two models. The local model perf rms increasingly better than the global model "on average" as a better algorithm is used. Table 6.5 shows the percent increase in the geometric mean number of constraint checks of the local model over the global model for the increasingly better algorithms (according to our experiments), MFC, MFC-CBJ, MFC-EXP-FF, and MFC-EXP-CBJ-FF. As a better algorithm is used, the number of constrai.. checks performed decreases, but the relative difference in the hardness of the problems, as seen by the algorithm, increases in favour of the local model. That is, problems generated with the local model are more difficult to search than problems generated with the global model for the best algorithms. Tables 6.3 and 6.4 also show that the global model produces outliers more often than the local model. That is, for every algorithm, the global model gives the largest maximum. For the algorithms not using FF the local model has the least standard deviation. For the algorithms using FF, the

BETTER THAN	FC	MFC	FC CBJ	MFC CBJ	FC FF	MFC FF	FC CBJ FF	MFC CBJ FF	MFC EXP FF	MFC CBJ EXP FF	MFC INC FF	MFC CBJ INC FF
FC		0 0 (0 1)	0 0 (7 2)	0 0 (0 0)	2 3 (0 3)	1 3 (0 0)	2 3 (0 4)	1 0 (0 0)	1 6 (0 1)	1 6 (0 1)	1 7 (0 1)	1 7 (0 1)
MFC	99 9		79 3 (0 2)	0 0 (5 4)	5 9 (0 2)	3 8 (0 3)	5 8 (0 2)	3 4 (0 3)	3 5 (0 2)	3 5 (0 2)	3 7 (0 2)	3 6 (0 2)
FC CBJ	92 8	20 6		0 0 (0 0)	2 9 (0 3)	2 0 (0 1)	2 7 (0 3)	1 4 (0 1)	2 0 (0 1)	1 9 (0 1)	2 1 (0 1)	2 0 (0 1)
MFC CBJ	100 0	94 6	100 0		8 2 (0 2)	5 8 (0 4)	8.1 (0 2)	4 9 (0 4)	4 9 (0 2)	4 7 (0 3)	5 0 (0 3)	4 9 (0 3)
FC FF	97 3	93 9	96 8	91 6		52 8 (0 4)	0 0 (70 0)	44 5 (0 3)	5 0 (0 6)	4 6 (0 6)	0 0 (2 5)	0 0 (2 5)
MFC FF	98 7	95 8	98 0	93 8	46 8		45 8 (0 3)	2 1 (15.5)	15.5 (1 1)	14.7 (1 1)	17 0 (0 7)	16 4 (0 7)
FC CBJ FF	97 4	94 0	96 9	91 8	30 0	53 9		45 6 (0 3)	5 9 (0.6)	4 9 (0.6)	1 0 (2 5)	0 0 (2 5)
MFC CBJ FF	98 9	96 3	98 5	94 7	55 2	82 3	54 1		19 0 (1 1)	18.2 (1 1)	20 8 (0.8)	20 0 (0 8)
MFC EXP FF	98 2	96 3	97 9	94 9	94 3	83 4	93 5	79 9		0 3 (61 5)	61 8 (2 6)	59 5 (2 6)
MFC CBJ EXP FF	98 3	96 3	98 0	95 0	94 8	84 2	94 5	80 7	38 2		64.6 (2 5)	62 2 (2 5)
MFC INC FF	98 2	96 1	97 8	94 8	97 5	82.4	96.5	78 5	35 5	32 9		0 0 (70 0)
MFC CBJ INC FF	98 2	96 2	97 9	94 9	97 5	83 0	97 5	79 2	37 9	35 2	30 0	

Table 6.1: Percentage of times one algorithm performs better than another by number of constraint checks, for problems generated with the local model, $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$, $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. Percentage of times algorithms perform the same number of constraint checks are in brackets. (10,200 problems).

local model has the highest standard deviation. This difference in standard deviation appears to be caused by FF.

We now turn our attention to the region where the two models of hard random problems differ the most, that is, for larger n, larger m, and sparse p_1. We focus our comparison on the problems with $n = 25$, $m = 9$, and $p_1 \leq 0.5$ as the results in Section 5.4 indicate that this is the region in which the two models differ the most in hardness of random problems generated.

We display in Tables 6.6 and 6.7 the percentage of times one algorithm performs better than another by the number of constraint checks performed for the local and global model testbeds, respectively. Tables 6.8 and 6.9 give the average, the standard

BETTER THAN	FC	MFC	FC CBJ	MFC CBJ	FC FF	MFC FF	FC CBJ FF	MFC CBJ FF	MFC FXP FF	MFC CBJ FXP FF	MFC INC FF	MFC CBJ INC FF
FC		0 0 (0 0)	0 0 (7 8)	0 0 (0.0)	2 5 (0 5)	1 2 (0 0)	2 4 (0 5)	0 9 (0 0)	1 7 (0 1)	1 7 (0 1)	1 8 (0 1)	1 8 (0 1)
MFC	100 0		77 5 (0 2)	0 0 (6 1)	7 5 (0 1)	3 9 (0 5)	7 4 (0 1)	3 4 (0 5)	3 9 (0 3)	3 8 (0 3)	4 0 (0 3)	3 9 (0 3)
FC CBJ	92 2	22 4		0 0 (0 0)	3 0 (0 4)	1 7 (0 0)	2 9 (0 4)	1 2 (0 0)	2 0 (0 1)	1 9 (0 1)	2 1 (0 1)	2 0 (0 1)
MFC CBJ	100 0	93 9	100 0		10 4 (0 1)	6 2 (0 5)	10 1 (0 1)	4 9 (0 5)	5 3 (0 3)	5 1 (0 3)	5 3 (0 4)	5 1 (0 4)
FC FF	97 0	92 4	96 6	89 5		42 8 (0 2)	0 0 (71 4)	34 7 (0 2)	4 3 (0 6)	3 6 (0 6)	0 0 (2 9)	0 0 (2 9)
MFC FF	98 8	95 6	98 2	93 3	57 0		55 8 (0 3)	1 7 (19 0)	17 6 (1 2)	16 8 (1 2)	20 0 (0 8)	19 5 (0 8)
FC-CBJ FF	97 1	92 5	96 7	89 8	28 6	43 9		35 7 (0 2)	5 1 (0 6)	3 9 (0 6)	1 1 (2 9)	0 0 (2 9)
MFC-CBJ FF	99 1	96 2	98 8	94 5	65 1	79 3	64 1		21 9 (1 3)	20 8 (1 3)	24 4 (0 9)	21 4 (0 9)
MFC-EXP FF	98 2	95 7	97 9	94 3	95 1	81 2	94 3	76 8		0 1 (64 4)	64 9 (3 2)	62 5 (3 2)
MFC-CBJ EXP-FF	98 2	95 8	98 0	94.6	95 8	82 0	95 4	77 9	35 5		67 8 (3 2)	65 2 (3 3)
MFC-INC FF	98 1	95 7	97 8	94 3	97 1	79 1	96 0	74 7	31 9	29 0		0 0 (71 4)
MFC-CBJ INC FF	98 1	95 8	97 9	94 5	97 1	79 9	97 1	75 7	34 3	31 5	28 6	

Table 6.2: Percentage of times one algorithm performs better than another by constraint checks, for problems generated with the global model with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$, $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. Percentage of times algorithms perform the same number of constraint checks are in brackets. (10,200 problems).

Alg	Ave	Std Dev	Med	Geo Mean	Max	Min	% FC
FC	675490	2339719	8460	13981	35296934	15	100.0
MFC	417021	1437487	6154	9943	22441085	12	71.1
FC-CBJ	559374	1973626	6766	11320	30667461	15	81.0
MFC-CBJ	342786	1204005	4820	7934	19385318	12	56.7
FC-FF	37475	119827	1980	2774	1131018	15	19.8
MFC-FF	40923	130621	1976	2803	1170366	12	20.0
FC-CBJ-FF	37388	119668	1961	2753	1129569	15	19.7
MFC-CBJ-FF	39705	127345	1840	2673	1138132	12	19.1
MFC-EXP-FF	25739	80506	1520	2151	765981	12	15.4
MFC-CBJ-EXP-FF	25671	80382	1503	2133	764843	12	15.3
MFC-INC-FF	26632	83905	1552	2187	792313	12	15.6
MFC-CBJ-INC-FF	26573	83804	1544	2171	791377	12	15.5

Table 6.3: Average, standard deviation, median, geometric mean, maximum and minimum number of constraint checks for the FC, MFC, FC-CBJ, MFC-CBJ, FC-FF, MFC-FF, FC-CBJ-FF, MFC-CBJ-FF, MFC-EXP-FF, MFC-CBJ-EXP-FF, MFC-INC-FF, MFC-CBJ-INC-FF algorithms for problems generated with the local model with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. (10,200 problems).

Alg	Ave	Std Dev	Med	Geo Mean	Max	Min	% FC
FC	710168	2637242	8626	13539	58051198	15	100.0
MFC	433021	1603277	6170	9483	35677917	12	70.0
FC-CBJ	564618	2131850	6592	10552	47140736	15	77.9
MFC-CBJ	342290	1290070	4594	7283	28510271	12	53.8
FC-FF	33450	109181	1872	2538	1254426	15	18.7
MFC-FF	35608	117645	1768	2439	1334088	12	18.0
FC-CBJ-FF	33358	109038	1846	2515	1251805	15	18.6
MFC-CBJ-FF	34575	114808	1668	2324	1295503	12	17.2
MFC-EXP-FF	22600	72605	1391	1914	847274	12	14.1
MFC-CBJ-EXP-FF	22535	72499	1377	1896	846036	12	14.0
MFC-INC-FF	23474	75872	1439	1958	884719	12	14.5
MFC-CBJ-INC-FF	23414	75784	1423	1941	883040	12	14.3

Table 6.4: Average, standard deviation, median, geometric mean, maximum and minimum number of constraint checks for the FC, MFC, FC-CBJ, MFC-CBJ, FC-FF, MFC-FF, FC-CBJ-FF, MFC-CBJ-FF, MFC-EXP-FF, MFC-CBJ-EXP-FF, MFC-INC-FF, MFC-CBJ-INC-FF algorithms for problems generated with the global model with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$ and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. (10,200 problems).

Algorithm	Percent increase in the geometric mean
MFC	4.9
MFC-CBJ	8.9
MFC-EXP-FF	12.4
MFC-CBJ-EXP-FF	12.5

Table 6.5: Percent increase of the geometric mean number of constraint checks performed using the local model over the geometric mean number of constraint checks performed using the global model for the specified algorithm, for the problems with $n \in \{10, 15, 20, 25\}$, $m \in \{3, 6, 9\}$, and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. (10,200 problems).

deviation for the average, the median, the geometric mean, the maximum, the minimum, and the percentage of constraint checks by geometric mean performed relative to FC by the 12 algorithms for the local and global testbeds respectively.

The ordering of algorithms for these large problems with sparse graphs from best to worst is the same as for Table 6.1. However, Table 6.9 gives a slightly different ordering with MFC-CBJ-INC-FF being better than MFC-EXP-FF by geometric mean and median for the global model. The MFC-CBJ-EXP-FF algorithm is still the best algorithm.

Table 6.6 shows that MFC-CBJ-INC-FF is the best algorithm for these large problems with sparse graphs generated by the local model, being better than MFC-CBJ-EXP-FF and much better than MFC-EXP-FF. Scatter graphs of MFC-CBJ-EXP-FF versus MFC-CBJ-INC-FF for the global and local models (displayed in Figures 6.1 and 6.2 respectively) show almost no real difference in performance between the algorithms under both models. There are a few outliers in the global model where MFC-EXP-CBJ-FF performs better. Table 6.6 also indicates that MFC-INC-FF is better than MFC-EXP-FF. Scatter graphs of MFC-EXP-FF versus MFC-INC-FF for the global and local model (displayed in Figures 6.3 and 6.4 respectively) again show

BETTER THAN	FC	MFC	FC CBJ	MFC CBJ	FC FF	MFC FF	FC CBJ FF	MFC CBJ FF	MFC EXP FF	MFC CBJ EXP FF	MFC INC FF	MFC CBJ INC FF
FC		0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0.0 (0 0)	0 0 (0 0)
MFC	100 0		82 3 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0.0)	0 0 (0.0)	0 0 (0.0)	0 0 (0 0)	0 0 (0 0)
FC CBJ	100 0	17 7		0 0 (0 0)	0.0 (0 0)	0.0 (0.0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)
MFC CBJ	100 0	100 0	100 0	-	0 0 (0 0)	0.0 (0.0)	0.0 (0 0)	0 0 (0 0)	0 0 (0.0)	0 0 (0.0)	0 0 (0 0)	0 0 (0 0)
FC FF	100 0	100 0	100 0	100 0		89 7 (0 0)	0 0 (3 4)	79 7 (0 0)	0.6 (0.0)	0 6 (0 0)	0 0 (0 0)	0.0 (0 0)
MFC FF	100 0	100 0	100 0	100.0	10 3	-	9 1 (0 0)	0 9 (0.0)	0 9 (0 0)	0 6 (0.0)	0.9 (0 0)	0 6 (0 0)
FC-CBJ FF	100 0	100 0	100 0	100 0	96 6	90 9		82 3 (0 0)	1.1 (0 0)	0 6 (0 0)	0 3 (0.0)	0 0 (0 0)
MFC-CBJ FF	100 0	100 0	100 0	100 0	20 3	99 1	17 7	-	1 4 (0 0)	0 6 (0 0)	1 7 (0 0)	1 4 (0 0)
MFC-EXP FF	100 0	100 0	100 0	100 0	99.4	99 1	98 9	98.6	-	0 3 (0.9)	46 6 (0 0)	38 0 (0 3)
MFC-CBJ EXP FF	100 0	100 0	100 0	100 0	99 4	99 4	99 4	99 4	98 9	—	56.3 (0 0)	47 1 (0 3)
MFC-INC FF	100 0	100 0	100 0	100.0	100.0	99 1	99 7	98.3	53 4	43 7		0 0 (3 4)
MFC-CBJ INC-FF	100 0	100 0	100 0	100 0	100.0	99 4	100 0	98 6	61.7	52.6	96 6	-

Table 6.6: Percentage of times one algorithm performs better than another by constraint checks, for problems generated with the local model, $n = 25$, $m = 9$, $p_1 \in \{0.2, 0.25, \ldots, 0.5\}$. Percentage of times algorithms perform the same number of constraint checks are in brackets. (350 problems).

BETTER THAN	FC	MFC	FC CBJ	MFC CBJ	FC FF	MFC FF	FC CBJ FF	MFC CBJ FF	MFC EXP FF	MFC CBJ EXP FF	MFC INC FF	MFC CBJ INC FF
FC	-	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0.0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)
MFC	100 0	-	74 0 (0 0)	0 0 (0 0)	0 0 (0.0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)
FC CBJ	100 0	26.0		0 0 (0 0)	0 3 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)
MFC CBJ	100 0	100 0	100 0		0 9 (0 0)	0 6 (0 0)	0 6 (0 0)	0 6 (0 0)	0 3 (0 0)	0 3 (0 0)	0 3 (0 0)	0 0 (0 0)
FC FF	100 0	100 0	99 7	99 1		59 7 (0 0)	0 0 (8 9)	48 0 (0 0)	0 3 (0 0)	0 0 (0 0)	0 0 (0 0)	0 0 (0 0)
MFC FF	100.0	100 0	100 0	99 4	40 3		38 3 (0 0)	0 3 (0.0)	4 9 (0 0)	4 6 (0 0)	5 7 (0 0)	4 6 (0 0)
FC-CBJ FF	100.0	100 0	100 0	99 4	91 1	61 7		50 6 (0 0)	0 9 (0 0)	0 (0 0)	0 0 (0 0)	0 0 (0 0)
MFC-CBJ FF	100.0	100 0	100 0	99.4	52 0	99 7	49 4		9 1 (0 0)	6 9 (0 0)	8 0 (0 0)	7 1 (0 0)
MFC-EXP FF	100 0	100 0	100.0	99 7	99 7	95 1	99 1	90 9		0 3 (3 7)	80 3 (0 0)	51 1 (0 0)
MFC-CBJ EXP-FF	100 0	100 0	100 0	99 7	100.0	95 4	99.4	93 1	96 0		71 7 (0 0)	61 7 (0 0)
MFC-INC FF	100.0	100 0	100.0	99 7	100 0	94 3	100 0	92 0	39 7	28 3		0 0 (8 9)
MFC-CBJ INC-FF	100.0	100.0	100 0	100 0	100 0	95 4	100 0	92 9	48 9	38 3	91 1	

Table 6.7: Percentage of times one algorithm performs better than another by constraint checks, for problems generated with the global model, $n = 25$, $m = 9$, $p_1 \in \{0.2, 0.25, \ldots, 0.5\}$. Percentage of times algorithms perform the same number of constraint checks are in brackets. (350 problems).

Alg	Ave	Std Dev	Med	Geo Mean	Max	Min	% FC
FC	2958410	2568973	2289544	1880200	14754851	12977	100.0
MFC	1774454	1530444	1378744	1127571	7914159	8263	60.0
FC-CBJ	2161974	1953434	1610420	1290927	10020111	8345	68.7
MFC-CBJ	1294767	1170570	955211	773359	5852651	5334	41.1
FC-FF	77770	64212	61062	50537	323935	2865	2.7
MFC-FF	92904	73746	72281	61941	381053	2058	3.3
FC-CBJ-FF	77152	64094	60325	49733	322034	2865	2.6
MFC-CBJ-FF	87745	70993	67379	57392	365090	1818	3.1
MFC-EXP-FF	55092	44500	42682	36596	231360	1713	1.9
MFC-CBJ-EXP-FF	54606	44384	42268	36019	229972	1713	1.9
MFC-INC-FF	55190	45235	44108	36232	229288	2145	1.9
MFC-CBJ-INC-FF	54768	45165	43122	35680	228061	2145	1.9

Table 6.8: Average, standard deviation, median, geometric mean, maximum and minimum number of constraint checks for the FC, MFC, FC-CBJ, MFC-CBJ, FC-FF, MFC-FF, FC-CBJ-FF, MFC-CBJ-FF, MFC-EXP-FF, MFC-CBJ-EXP-FF, MFC-INC-FF, MFC-CBJ-INC-FF algorithms for problems generated with the local model, $n = 25$, $m = 9$ and $p_1 \in \{0.2, 0.25, \ldots, 5.0\}$. (350 problems).

Alg	Ave	Std Dev	Med	Geo Mean	Max	Min	% FC
FC	3689112	4532671	2226298	1810012	33552581	11528	100.0
MFC	2155573	2689986	1257608	1043480	20976509	6825	57.7
FC-CBJ	2361452	2944184	1428119	1100006	26956306	7100	60.8
MFC-CBJ	1381946	1761025	818433	632111	15475314	4288	34.9
FC-FF	60835	54665	43915	37164	263536	2107	2.1
MFC-FF	65719	61299	46180	38745	306448	1487	2.1
FC-CBJ-FF	60129	54529	43843	36316	263019	2098	2.0
MFC-CBJ-FF	62000	58875	42441	35806	293636	1302	2.0
MFC-EXP-FF	40572	36788	29171	24815	180758	1148	1.4
MFC-CBJ-EXP-FF	40111	36679	28938	24270	180394	1148	1.3
MFC-INC-FF	41287	37389	29394	25168	179882	1284	1.4
MFC-CBJ-INC-FF	40846	37316	28968	24631	179596	1284	1.4

Table 6.9: Average, standard deviation, median, geometric mean, maximum and minimum number of constraint checks for the FC, MFC, FC-CBJ, MFC-CBJ, FC-FF, MFC-FF, FC-CBJ-FF, MFC-CBJ-FF, MFC-EXP-FF, MFC-CBJ-EXP-FF, MFC-INC-FF, MFC-CBJ-INC-FF algorithms for problems generated with the global model with $n = 25$, $m = 9$ and $p_1 \in \{0.2, 0.25, \ldots, 5.0\}$. (350 problems).

Algorithm	percent increase in the geometric mean
MFC	8.1
MFC-CBJ	22.3
MFC-EXP-FF	47.5
MFC-CBJ-EXP-FF	48.4

Table 6.10: Percent increase of geometric mean number of constraint checks performed using the local model over the geometric mean number of constraint checks performed using the global model for the specified algorithm, for the problems in $n = 25$, $m = 9$, and $p_1 \in \{0.2, 0.25, \ldots, 1.0\}$. (350 problems).

no real difference in performance. Table 6.6 also shows FC-CBJ-FF and FC-FF performing much better than MFC-FF and MFC-CBJ-FF respectively. Scatter graphs comparing FC-FF with MFC-FF for the global and local model are displayed in Figures 6.5 and 6.6 respectively. The local model shows that nearly all problems are searched by the FC-FF algorithm with fewer constraint checks performed.

Table 6.9 gives the usual ordering for the algorithms not using FF. For the algorithms using FF, MFC-CBJ-INC-FF is best by geometric mean followed by MFC-CBJ-EXP-FF, MFC-INC-FF, MFC-EXP-FF, FC-CBJ-FF, FC-FF, MFC-CBJ-FF, and finally MFC-FF. However, by average and median MFC-CBJ-EXP-FF is better than MFC-CBJ-INC-FF followed by MFC-EXP-FF and MFC-INC-FF. As in Tables 6.3 and 6.4, Tables 6.8 and 6.9 show the same trend that the local model performs increasingly better than the global model "on average" as a better algorithm is used. Table 6.10 shows the percent increase in the geometric mean number of constraint checks of the local model over the global model for the increasingly better algorithms MFC, MFC-CBJ, MFC-EXP-FF, and MFC-EXP-CBJ-FF. The percent increase shown in Table 6.10 is much higher than in Table 6.5 as the local model produces harder problems for these sparse graphs.

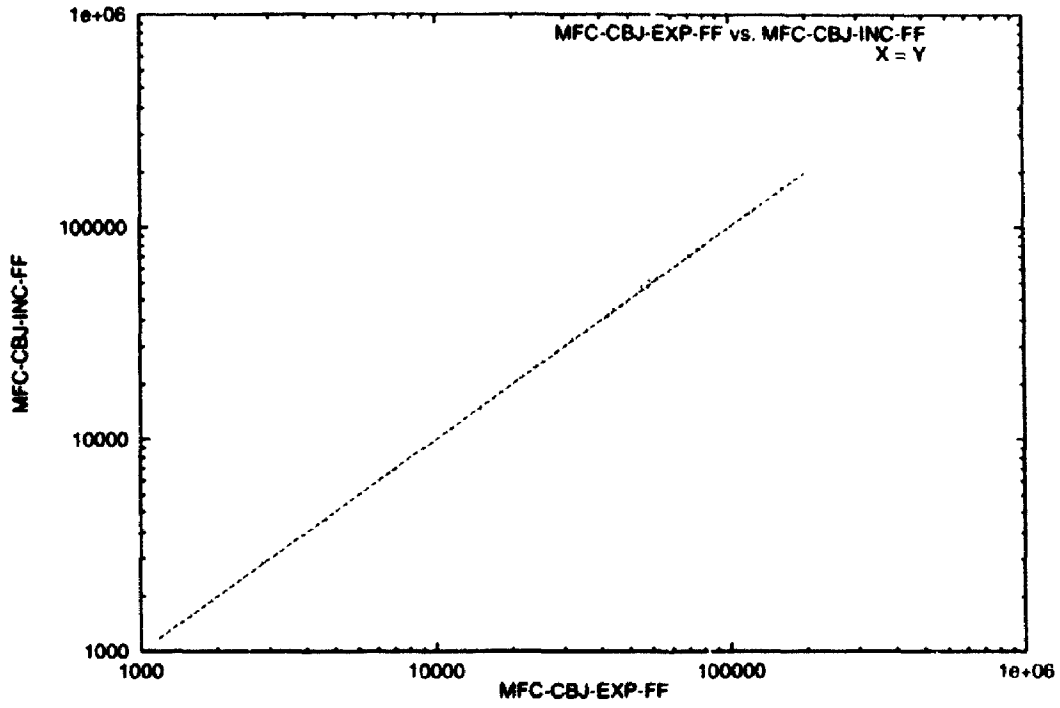

Figure 6.1: The number of constraint checks performed by MFC-CBJ-EXP-FF versus the number of constraint checks performed by MFC-CBJ-INC-FF on the same problem using the global model for problems with $n = 25$, $m = 9$ and $p_1 \in \{0.2, 0.25, \ldots, 0.5\}$. (350 problems).

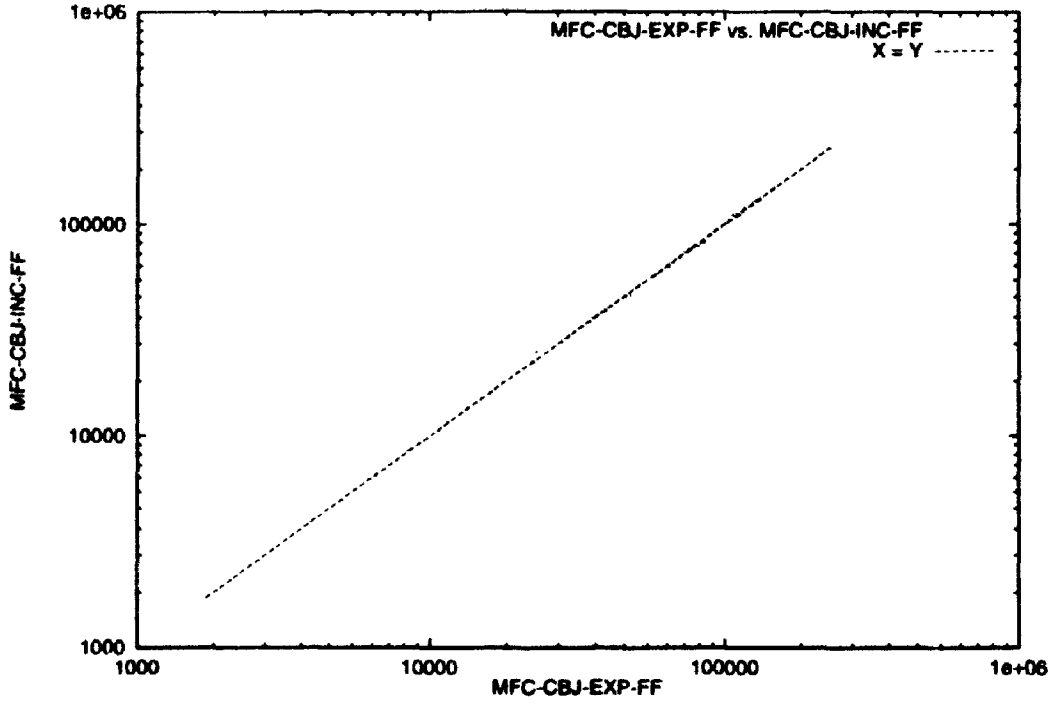

Figure 6.2: The number of constraint checks performed by MFC-CBJ-EXP-FF versus the number of constraint checks performed by MFC-CBJ-INC-FF on the same problem using the local model for problems with $n = 25$, $m = 9$ and $p_1 \in \{0.2, 0.25, \ldots, 0.5\}$. (350 problems).

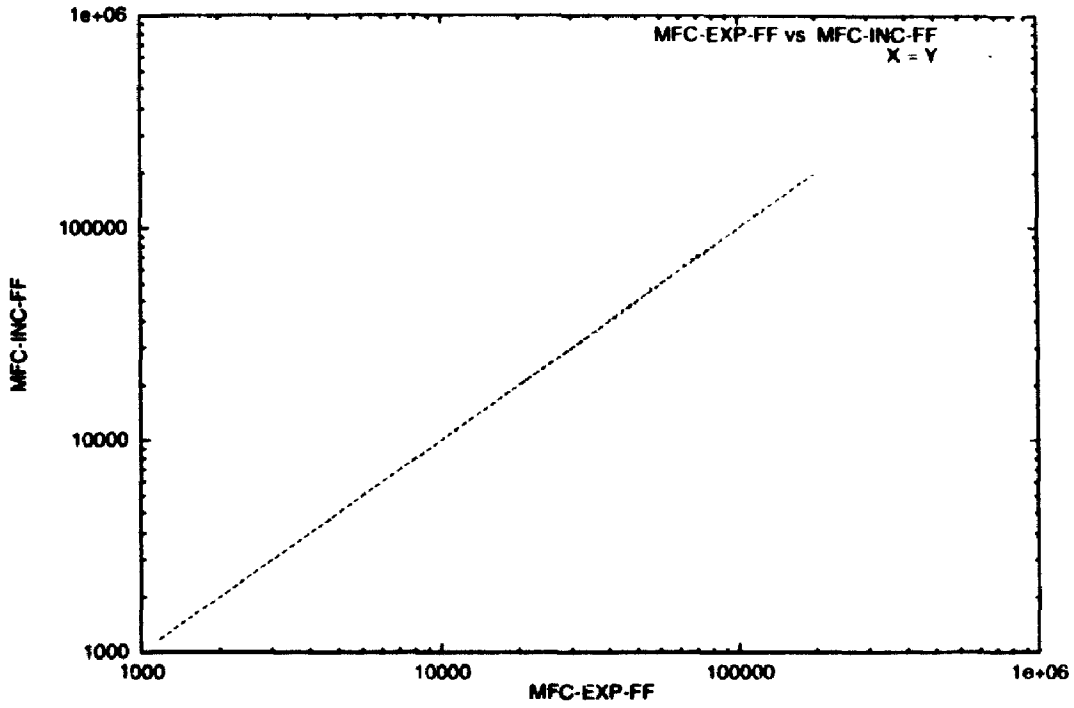

Figure 6.3: The number of constraint checks performed by MFC-EXP-FF versus the number of constraint checks performed by MFC-INC-FF on the same problem using the global model for problems with $n = 25$, $m = 9$ and $p_1 \in \{0.2, 0.25, \ldots, 0.5\}$. (350 problems).

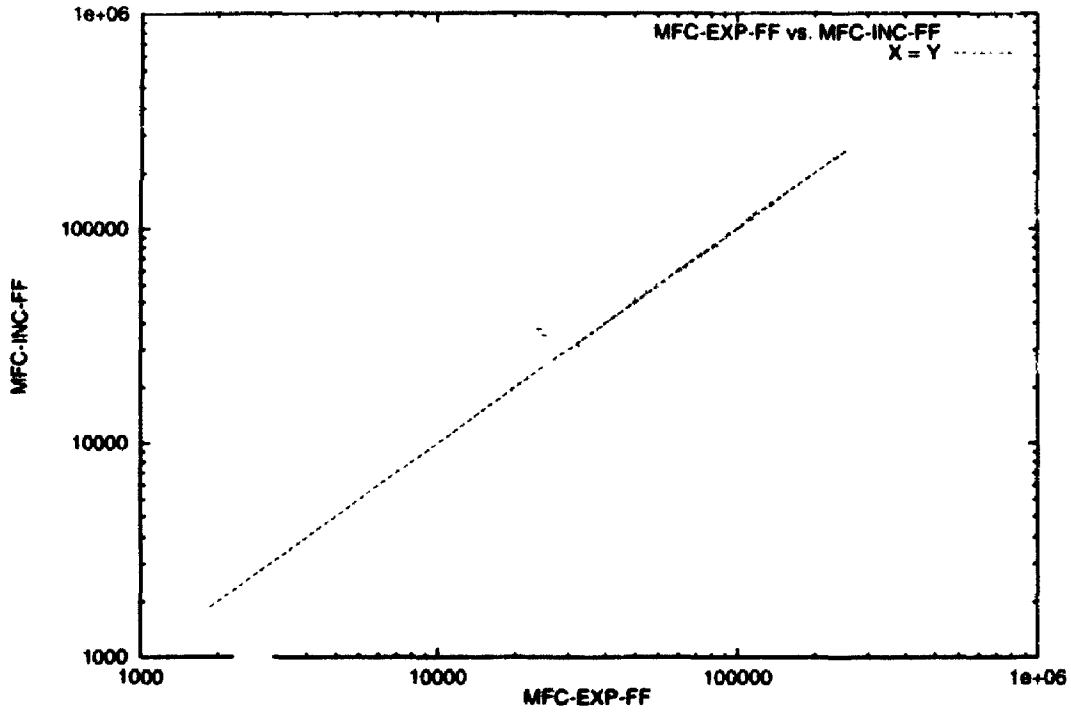

Figure 6.4: The number of constraint checks performed by MFC-EXP-FF versus the number of constraint checks performed by MFC-INC-FF on the same problem using the local model for problems with $n = 25$, $m = 9$ and $p_1 \in \{0.2, 0.25, \ldots, 0.5\}$. (350 problems).

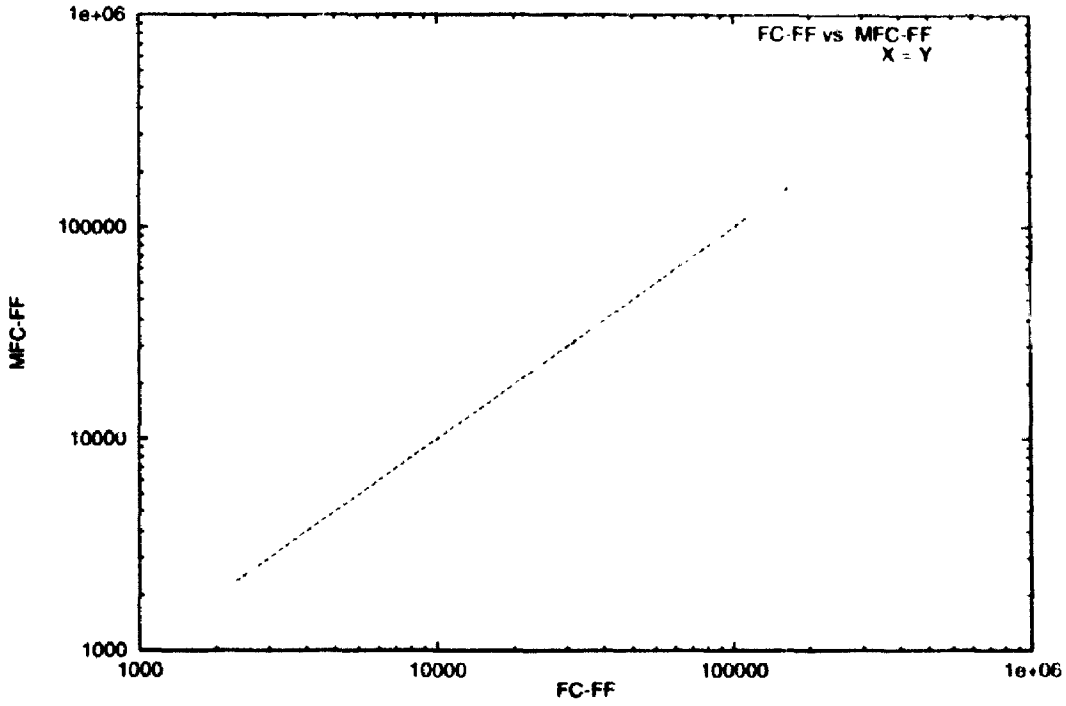

Figure 6.5: The number of constraint checks performed by FC-FF versus the number of constraint checks performed by MFC-FF on the same problem using the global model for problems with $n = 25$, $m = 9$ and $p_1 \in \{0.2, 0.25, \ldots, 0.5\}$. (350 problems).

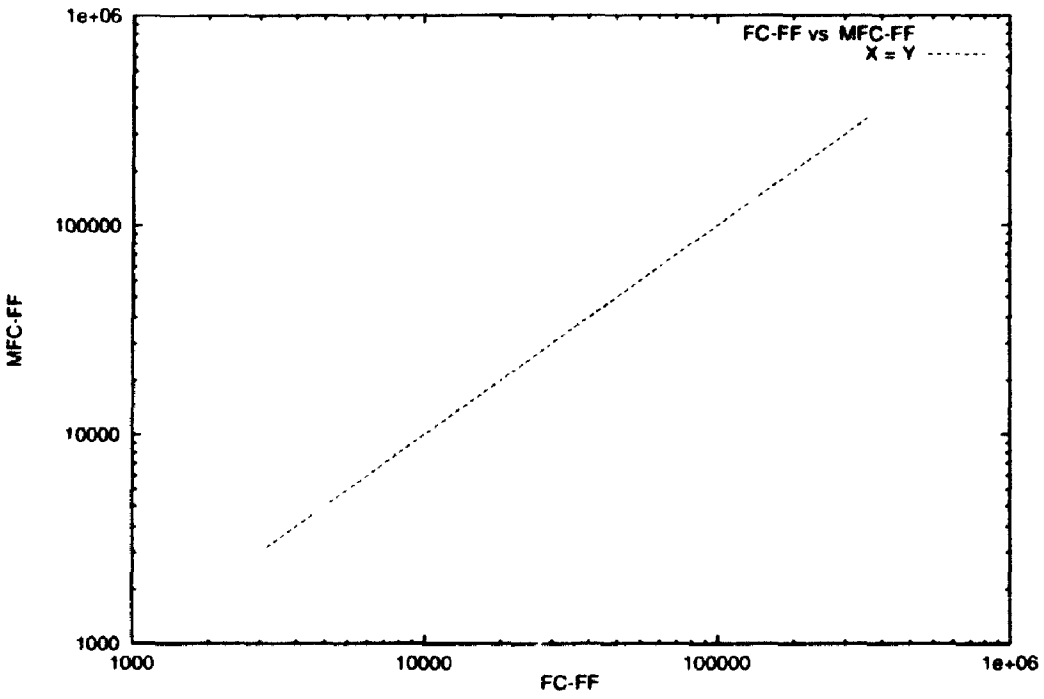

Figure 6.6: The number of constraint checks performed by FC-FF versus the number of constraint checks performed by MFC-FF on the same problem using the local model for problems with $n = 25$, $m = 9$ and $p_1 \in \{0.2, 0.25, \ldots, 0.5\}$. (350 problems).

In this comparison, the total number of problems solved by the algorithms without FF for the local model is 40,800. The total number of constraint checks performed was 20,345,785,271 with 4,502,875 seconds of CPU time. The total number of problems solved by the algorithms with FF is 81,600. The total number of constraint checks is 2,653,076,362 with 711,724 seconds of CPU time. No valid comparisons can be drawn from these CPU times as the experiments were performed on 3 different machines (a Sun Sparc 5, 10, and 20).

6.2 Summary

In this chapter we empirically compare the 12 algorithms, FC, MFC, FC-CBJ, MFC-CBJ, FC-CBJ-FF, MFC-CBJ-FF, MFC-EXP-FF, MFC-CBJ-EXP-FF, MFC-INC-FF, and MFC-CBJ-INC-FF using the local model of hard random problems. Over the whole testbed of problems generated with the local model, the ranking of the algorithms remains the same as found in Chapter 4. That is, averaged over all problems in the testbed, MFC-CBJ-EXP-FF is the best of the 12 algorithms tested closely followed by MFC-EXP-FF. Bacchus and Grove's INC-FF heuristic (with and without CBJ) is a close third and fourth in terms of the average number of constraint checks performed. Of the algorithms using FF but not CBJ, MFC-EXP-FF is the best algorithm. Of the algorithms not using FF, MFC-CBJ is the best. Without CBJ and without FF, MFC is best.

However, when we use problems with sparser graphs and larger values of n and m, namely those with $n = 25$, $m = 9$, and $p_1 \leq 0.5$, it is unclear which algorithm is the best. Problems generated via the global model suggest that MFC-CBJ-EXP-FF is the best algorithm by the average, the median, and the geometric mean, whereas problems generated using the local model indicate that MFC-CBJ-INC-FF is the best by the geometric mean and MFC-CBJ-EXP-FF is the best by the average and the median. The local model indicates that the MFC-CBJ-INC-FF algorithm is slightly better than the MFC-CBJ-EXP-FF by percentage of times one algorithm performs better than the other by number of constraint checks performed. However, scatter

plots of MFC-CBJ-EXP-FF versus MFC-CBJ-INC-FF under the local model do not show any significant difference. Both algorithms appear to be equally good on these large sparse problems.

The comparison with the large sparse problems generated with the local model also makes the poor performance of MFC-FF more obvious. The FC-FF algorithm performs fewer constraint checks than the MFC-FF algorithm on almost all the large sparse problems.

Finally, the comparison of algorithms under the two models also gives us the opportunity to compare the hardness of problems randomly generated under the two models across algorithms. The problems generated under the local model are increasingly harder to search relative to problems generated under the global model the better the algorithm used.

Chapter 7

Conclusions

7.1 Contributions

In this thesis we have developed and empirically compared several new CSP search algorithms.

In Chapter 2 we describe a promising new CSP search algorithm called Minimal Forward Checking (MFC) which is an improved version of the best known CSP search algorithm called Forward Checking (FC)[55, 61, 77]. We motivate the development of MFC by showing that FC which is a "forward looking" search algorithm has a strong relationship to BackMarking (BM)[47] which is a "backward looking" algorithm. We theoretically prove that MFC is sound and complete. We also prove that MFC and FC visit the same nodes in the search tree and that MFC's worst case performance in terms of number of constraint checks performed is the number of constraint checks performed by FC. We also show the negative result that MFC with the popular dynamic variable reordering heuristic called Fail First (FF)[61] can perform worse than FC with the same heuristic. Finally, we describe how MFC can be considered a lazy CSP search algorithm and discuss the existence of other lazy CSP search algorithms.

In Chapter 3 we make an empirical comparison of the BT, BM, FC, MFC, FC-FF, and MFC-FF algorithms on a large set of hard randomly generated CSPs. Hard randomly generated CSPs are on average much harder to solve than most randomly generated CSPs and are used for the empirical comparison as they are considered likely to bring out significant performance differences between search algorithms. The results of our empirical comparison indicate that the BT and BM algorithms are inefficient and that FC is clearly superior to both of these algorithms. The average

case performance of the MFC algorithm is clearly superior to that of the FC algorithm across all dimensions of the testbed resulting in savings ranging from 20% to 40% of the number of constraint checks performed by FC. We also find that the search algorithms using the FF heuristic clearly outperform the search algorithms not using the FF heuristic. The performance of MFC-FF is slightly better than FC-FF but it clearly suffers from its inability to use the FF heuristic to its fullest advantage. Overall the ranking of the algorithms as indicated by our experiments from worst to best is BT < BM < FC < MFC < FC-FF < MFC-FF. The MFC algorithm is the best algorithm not using the FF heuristic (the FF heuristic is not appropriate for all problems) and the MFC-FF algorithm is the best algorithm using FF although the comparison is not as conclusive as between MFC and FC.

In Chapter 4 improvements are made to the MFC algorithm. One improvement is the addition of non-chronological backtracking, called Conflict-Directed Backjumping (CBJ)[90], to the MFC algorithm which allows it to save substantially more constraint checks with every backjump than FC-CBJ. The second improvement is the creation of a new heuristic, called "Extra Pruning with Fail First" (EXP-FF), designed to help solve MFC-FF's inability to exploit the full power of the FF heuristic. We then discuss conjectured dominance relationships between some of the hybrid algorithms followed by a comprehensive empirical comparison using hard random problems of the new hybrid algorithms. Our comparisons also include hybrid algorithms that use a heuristic designed by Bacchus and Grove[1] to serve the same purpose as the EXP-FF heuristic, which we call INC-FF. Altogether the algorithms compared are FC, MFC, FC-CBJ, MFC-CBJ, FC-FF, MFC-FF, FC-CBJ-FF, MFC-CBJ-FF, MFC-CBJ-EXP-FF, MFC-INC-FF, and MFC-CBJ-INC-FF. As in Chapter 3, the performance of the algorithms using FF is much better than the performance of algorithms not using FF. Averaged over all problems, the 4 best algorithms are from worst to best MFC-INC-FF < MFC-CBJ-INC-FF < MFC-EXP-FF < MFC-CBJ-EXP-FF. Of the algorithms not using FF, the ordering of algorithms by average performance from worst to best is FC < FC-CBJ < MFC < MFC-CBJ. Whether the EXP-FF heuristic is used and/or CBJ is used, algorithms based on MFC perform much better on average than algorithms

based on FC.

In Chapter 5 we give a generalization of the standard model for binary CSPs which allows for local constraint tightness values. Our experiments show that there is a phase transition in this new model similar to the phase transition observed in the standard model. A predictor of the phase transition peak is derived for this new model that includes the local graph topology of an individual problem. We empirically show that the new predictor accurately predicts the phase transition peak for this new model except for very sparse problems with $p_1 < 0.3$. Although the new predictor does not accurately predict the phase transition peak for very sparse problems its prediction appears to be much better than Smith's predictor[102, 103]. Finally, we have compared random problems generated with the two predictors. The problems generated under the new model are of similar or increased hardness than problems generated using the old model especially for problems with larger values of n and m and sparser graphs ($p_1 \leq 0.5$). These hard random problems generated under the new model can be used to create a new testbed of problems with which CSP search algorithms can be compared.

Finally, in Chapter 6 we empirically compare the 12 algorithms, FC, MFC, FC-CBJ, MFC-CBJ, FC-CBJ-FF, MFC-CBJ-FF, MFC-EXP-FF, MFC-CBJ-EXP-FF, MFC-INC-FF, and MFC-CBJ-INC-FF using hard random problems generated with the new (local) model. Over the whole testbed, the ranking of the algorithms remains the same as found in Chapter 4. However, when we use problems with sparser graphs and larger values of n and m, namely those with $n = 25$, $m = 9$, and $p_1 \leq 0.5$ there is some doubt as to which algorithm is the best. The MFC-CBJ-INC-FF appears to be more appropriate for this region of problems although the difference in performance between MFC-CBJ-INC-FF and MFC-CBJ-EXP-FF is negligible when compared by scatter plots. The comparison with the large sparse problems generated with the local model also makes the poor performance of MFC-FF more obvious. The FC-FF algorithm performs fewer constraint checks than the MFC-FF algorithm on almost all the large sparse problems. Finally, the comparison of algorithms under the two models gives us the opportunity to compare the hardness of problems randomly generated

under the two models across algorithms. As a better algorithm is used, the number of constraint checks performed decreases, but the relative difference in the hardness of the problems, as seen by the algorithm, increases in favour of the local model. That is, problems generated with the local model are more difficult to search than problems generated with the global model for the best algorithms.

In conclusion, we develop a new CSP search algorithm, called MFC, that is a lazy version of FC. The MFC algorithm is shown theoretically to perform no worse than FC, and using two testbeds of hard random problems drawn from two models it is shown empirically that MFC's average case performance is significantly better than FC's. With the addition of CBJ, the MFC-CBJ algorithm is the best algorithm that we test that does not use the FF heuristic. Overall, the best algorithm tested is the MFC-CBJ-EXP-FF algorithm followed closely by either MFC-CBJ-INC-FF or MFC-EXP-FF. We have argued that the MFC algorithm is one instance of a lazy CSP search algorithm and that other lazy CSP search algorithms are possible.

We also develop a new model of binary CSPs. We show that this new model of binary CSPs exhibits the phase transition phenomenon, and we develop a predictor of the phase transition peak for this new model. Randomly generated problems created using this new model are of similar hardness or harder than randomly generated problems created using the standard (previous) model of binary CSPs. These new hard problems are used as a testbed for CSP search algorithms.

7.2 Future Work

There are a number of directions in which to extend this thesis.

Theoretically, a number of conjectured dominance relationships between algorithms that use the FF heuristic are left open. Also, our formal dominance relationships and soundness and completeness results for MFC depend on a fixed instantiation order. Any proof involving FF (or any dynamic variable reordering heuristic) will depend on an accurate description of the set of search trees possible under the fail first heuristic. It will also involve specifying exactly how the FF heuristic should be implemented.

It would also be interesting to develop new lazy algorithms from Partial Lookahead (PL) and Full Lookahead (FL)[61] and compare these algorithms to MFC and MAC over a wider range of problems than examined in this thesis. The empirical results in this thesis are valid only for problems that are in the phase transition peak. The empirical testbed should be enlarged to include problems across all values of p_2 in a systematic way.

The new model developed in Chapter 5 is only a step in the right direction for modeling CSPs. The old model of binary CSPs is restricted to problems with global values of p_2 and m. We have generalized this model by allowing for some variation in the constraint tightnesses and given a predictor of a phase transition peak in this model. Our predictor can be further generalized to allow for varying domain sizes but it is unknown if it would continue to predict the location of a phase transition. We would like to find a richer model of binary CSPs that hopefully will be applicable to a wider range of real CSPs. We would also like to extend our work into finding phase transitions in non-binary CSPs.

Finally, it is conjectured that if a real problem can be modeled using the standard (global) model then Smith's predictor will accurately predict whether that problem will be near a phase transition peak. We also conjecture that if a real problem can be modeled using the local model then our predictor will accurately predict whether the problem will be near a phase transition peak. The task of finding out whether this is true or not is left as future work.

The global model and the local model also have counterparts for n-ary CSPs which we intend to explore in the future.

Appendix A

Glossary of Terms and Symbols

CLP Constraint Logic Programming p. 2

binary CSP p. 4

CSP Constraint Satisfaction Problem p. 4

V Set of variables p. 4

D Set of domains p. 4

C Set of constraints p. 4

$c_{i,j}$ Constraint between v_i and v_j p. 4

v_i The i'th variable p. 4

d_i The i'th domain p. 4

n The number of variables in a CSP p. 4

m_i The number of values in domain d_i p. 4

m The size of the largest domain p. 4

soluble problem p. 4

insoluble problem p. 4

constraint graph p. 5

partial solution p. 5

[31] M. Dent and R. Mercer. Using local graph topology to model hard binary constraint satisfaction problems. In *Workshop on Studying and Solving Really Hard Problems*, pages 52–61. International Conference on the Principles and Practice of Constraint Programming, 1995.

[32] M. Dent and R. Mercer. Using local graph topology to model hard binary constraint satisfaction problems. Technical Report UWO-CSD-439, The University of Western Ontario, 1995.

[33] M. Dent and R. Mercer. An empirical investigation of the forward checking algorithm and its derivatives. In *Proceedings of the International Conference on Tools with Artificial Intelligence*. IEEE Computer Society, 1996. To appear.

[34] M. Dent and R. Mercer. Improvements to the minima' forward checking algorithm. Technical Report UWO-CSD-458, The University of Western Ontario, 1996.

[35] M. Dent and R. Mercer. A new model of hard binary constraint satisfaction problems. In G. McCalla, editor, *Proceedings CSCSI, AI'96*, LNAI 1081, pages 14–25. Springer-Verlag, 1996.

[36] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving a cutting-stock problem in constraint logic programming. In *Proceedings of the International Conference on Logic Programming*, pages 42–58, 1988.

[37] E. Elcock. ABSYS: The first logic programming language – a retrospective and a commentary. *The Journal of Logic Programming*, 9:1–17, 1990.

[38] M. Fox. *Constraint-Directed Search: A Case Study of Job Shop Scheduling*. PhD thesis, Carnegie-Mellon University, 1983.

[39] E. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, 1978. Also appears as MIT AI memo 370, July 1976.

REFERENCES

[1] F. Bacchus and A. Grove. On the forward checking algorithm. In U. Montanari and F. Rossi, editors, *Proceedings of the International Conference on the Principles and Practice of Constraint Programming*, volume 976 of *LNCS 976*, pages 292–309. Springer Verlag, 1995.

[2] F. Bacchus and P. van Run. Dynamic variable ordering in CSP's. In U. Montanari and F. Rossi, editors, *Proceedings of the International Conference on the Principles and Practice of Constraint Programming*, volume 976 of *LNCS 976*, pages 258–275. Springer Verlag, 1995.

[3] A. Baker. The hazards of fancy backtracking. In *Proceedings AAAI-94*, pages 288–293, 1994.

[4] A. Baker. *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results.* PhD thesis, The University of Oregon, 1995.

[5] B. Benson and E. Freuder. Interchangeability preprocessing can improve forward checking search. In *Proceedings ECAI-92*, pages 28–30, 1992.

[6] B. Bernhardsson. Explicit solutions to the n-queens problem for all n. *SIGART Bulletin*, 2(2):7, 1991.

[7] C. Bessière and M.-O. Cordier. Arc-consistency and arc-consistency again. In *Proceedings AAAI-93*, pages 108–113, 1993.

[8] C. Bessière, E. Freuder, and J.-C. Régin. Using inference to reduce arc consistency computation. In *Proceedings IJCAI-95*, pages 592–599, 1995.

[9] C. Bessière and J.-C. Régin. An arc-consistency algorithm optimal in the number of constraint checks. In *Proceedings of the International Conference on Tools with Artificial Intelligence*, pages 397–403, 1994.

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

[11] B. Bollobas. *Random Graphs*. Academic Press, 1985.

[12] P. Burke and P. Prosser. A distributed asynchronous system for predictive and reactive scheduling. *The International Journal for Artificial Intelligence in Engineering*, 6(3):106–124, 1991.

[13] P. Cheeseman, B. Kanefsky, and W. Taylor. Where the really hard problems are. In *Proceedings IJCAI-91*, pages 331–337, 1991.

[14] J. Cohen. Constraint logic programming languages. *Communications of the ACM*, 33(7):52–68, July 1990.

[15] A. Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, July 1990.

[16] M. Cooper. An optimal k-consistency algorithm. *Artificial Intelligence*, 41(1):89–95, 1989.

[17] J. Crawford and L. Auton. Experimental results on the crossover point in satisfiability problems. In *Proceedings AAAI-93*, pages 21–27, 1993.

[18] J Crawford and L Auton. Experimental results on the crossover point in random 3-SAT. *Artificial Intelligence*, 81:31–57, 1996.

[19] A. Davenport. A comparison of complete and incomplete algorithms in the easy and hard regions. In *Workshop on Studying and Solving Really Hard Problems*, pages 43–51. International Conference on the Principles and Practice of Constraint Programming, 1995.

[20] J. de Kleer. A comparison of ATMS and CSP techniques. In *Proceedings IJCAI-89*. pages 290 296. 1989.

[21] R. Dechter. Enhancement schemes for constraint processing: Backjumping. learning. and cutset decomposition. *Artificial Intelligence*. 41(3):273 312. 1990.

[22] R. Dechter. Constraint networks. In S. Shapiro. editor. *Encyclopedia of Artificial Intelligence*. pages 276 285. Wiley. New York. 1992.

[23] R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings AAAI-88*. pages 37 42. 1988.

[24] R. Dechter and I. Meiri. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In *Proceedings IJCAI-89*. pages 271 277. 1989.

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

[26] R. Dechter. I. Meiri. and J. Pearl. Temporal constraint networks. *Artificial Intelligence*. 49:61 96. 1991.

[27] R. Dechter and J. Pearl. Network based heuristics for constraint satisfaction problems. *Artificial Intelligence*. 34(1):1 38. 1988.

[28] M. Dent and R. Mercer. Minimal forward checking. Technical Report UWO-CSD-374. The University of Western Ontario. 1993.

[29] M. Dent and R. Mercer. Minimal forward checking: A preliminary report. In *NATO ASI on Constraint Programming*. Technical Report CS 57/93. pages 9 22. Institute of Cybernetics. Tallinn. Estonia. 1993.

[30] M. Dent and R. Mercer. Minimal forward checking. In *Proceedings of the International Conference on Tools with Artificial Intelligence*. pages 432 438. 1994. IEEE Computer Society.

[31] M. Dent and R. Mercer. Using local graph topology to model hard binary constraint satisfaction problems. In *Workshop on Studying and Solving Really Hard Problems*, pages 52 61. International Conference on the Principles and Practice of Constraint Programming, 1995.

[32] M. Dent and R. Mercer. Using local graph topology to model hard binary constraint satisfaction problems. Technical Report UWO-CSD-439, The University of Western Ontario, 1995.

[33] M. Dent and R. Mercer. An empirical investigation of the forward checking algorithm and its derivatives. In *Proceedings of the International Conference on Tools with Artificial Intelligence*. IEEE Computer Society, 1996. To appear.

[34] M. Dent and R. Mercer. Improvements to the minima forward checking algorithm. Technical Report UWO-CSD-458, The University of Western Ontario, 1996.

[35] M. Dent and R. Mercer. A new model of hard binary constraint satisfaction problems. In G. McCalla, editor. *Proceedings CSCSI, AI'96*, LNAI 1081, pages 14 25. Springer-Verlag, 1996.

[36] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving a cutting-stock problem in constraint logic programming. In *Proceedings of the International Conference on Logic Programming*, pages 42 58, 1988.

[37] E. Elcock. ABSYS: The first logic programming language a retrospective and a commentary. *The Journal of Logic Programming*, 9:1 17, 1990.

[38] M. Fox. *Constraint-Directed Search: A Case Study of Job Shop Scheduling*. PhD thesis, Carnegie-Mellon University, 1983.

[39] E. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958 966, 1978. Also appears as MIT AI memo 370, July 1976.

[40] E. Freuder. Backtrack-free and backtrack-bounded search. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, chapter 10, pages 343–369. Springer-Verlag, 1988.

[41] E. Freuder and Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *Proceedings IJCAI-85*, pages 1076–1078, 1985.

[42] D. Frost and R. Dechter. Dead-end driven learning. In *Proceedings AAAI-94*, pages 294–300, 1994.

[43] D. Frost and R. Dechter. In search of the best constraint satisfaction search. In *Proceedings AAAI-94*, pages 301–306, 1994.

[44] D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings IJCAI-95*, pages 572–578, 1995.

[45] Y. Fu. The use and abuse of statistical mechanics in computational complexity. In D. Stein, editor, *Lectures in the Sciences of Complexity*, pages 815–826. Addison-Wesley Longman, 1989.

[46] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[47] J. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms*. PhD thesis, Carnigie-Mellon University, 1979.

[48] C. Gaspin, J.-C. Régin, T. Schiex, and G. Verfaillie. Lazy arc-consistency. In *Proceedings AAAI-96*, 1996. To appear.

[49] I. Gent, E. MacIntyre, P. Prosser, B. Smith, and T. Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. To appear: CP'96.

[50] I. Gent and T. Walsh. Easy problems are sometimes hard. *Artificial Intelligence*, 70:335–345, 1994.

[51] I. Gent and T. Walsh. The number partition phase transition. In *Workshop on Studying and Solving Really Hard Problems*, pages 84 100. International Conference on the Principles and Practice of Constraint Programming, 1995.

[52] I. Gent and T. Walsh. Phase transitions from real computational problems. In *Workshop on Studying and Solving Really Hard Problems*, pages 70 83. International Conference on the Principles and Practice of Constraint Programming, 1995.

[53] M. Ginsberg. Dynamic backtracking. *The Journal of Artificial Intelligence Research*, 1:25 46, 1993.

[54] M. Ginsberg, M. Frank, M. Halpin, and M. Torrance. Search lessons learned from crossword puzzles. In *Proceedings AAAI-90*, pages 210 215, 1990.

[55] S. Golomb and L. Baumert. Backtrack programming. *Journal of the ACM* 12(4):516 524, 1965.

[56] S. Grant and B. Smith. The phase transition behaviour of maintaining arc consistency. Technical Report Report 95.25, The University of Leeds, 1995. To appear ECAI-96.

[57] J. Gu. Efficient local search for very large-scale satisfiability problems. *SIGART Bulletin*, 3(1):8 12, 1992.

[58] M Hall and Knuth D. Estimating the efficiency of backtrack programs. *Amer. Math. Monthly*, 72:21 28, 1965.

[59] C.-C. Han and C.-H. Lee. Comments on Mohr and Henderson's path consistency algorithm. *Artificial Intelligence*, 36:125 130, 1988.

[60] R. Haralick, L. Davis, A. Rosenfeld, and D. Milgram. Reduction operations for constraint satisfaction. *Information Sciences*, 14:199 219, 1978.

[61] R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263 313, 1980.

[62] N. Heintze, S. Michaylov, and P. Stuckey. CLP($\Re$) and some electrical engineering problems. In *Proceedings of the International Conference on Logic Programming*, pages 675–703, 1987.

[63] T. Hogg and C. Williams. The hardest constraint problems: A double phase transition. *Artificial Intelligence*, 69:359–377, 1994.

[64] J. Jaffar and J-L. Lassez. Constraint logic programming. In *14th ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987.

[65] J. Jaffar and S. Michaylov. Methodology and implementation of a CLP system. In *Proceedings of the International Conference on Logic Programming*, pages 196–218, 1987.

[66] D. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29:121–136, 1975.

[67] G. Kondrak. A theoretical evaluation of selected backtracking algorithms. Technical Report TR94-10, University of Alberta, 1994.

[68] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. In *Proceedings IJCAI-95*, pages 541–547, 1995.

[69] V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, pages 32–44, 1992.

[70] A. Kwan. Validity of normality assumption in csp research. Technical Report CSM-261, The University of Essex, 1995.

[71] A. Kwan and E. Tsang. Minimal forward checking with backmarking and conflict-directed backjumping. Technical Report CSM-260, The University of Essex, 1996. Submitted to IEEE International Conference on Tools with Artificial Intelligence.

[72] C. Lassez, K. McAloon, and R. Yap. Constraint logic programming in options trading. *IEEE Expert*, Aug 1987.

[73] E. Lucas. *Récréations Mathématiques*. Gauthier-Villars, Paris, 1891.

[74] A. Mackworth. Consistency in a networks of relations. *Artificial Intelligence*, 8(1):99 118, 1977.

[75] A. Mackworth. On reading sketch maps. In *Proceedings IJCAI-77*, pages 598 606, 1977.

[76] A. Mackworth. Constraint satisfaction. In S. Shapiro, editor, *2nd Edition of the Encyclopedia of Artificial Intelligence*, pages 285 293. Wiley & Sons, New York, 1992.

[77] J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19:229 250, 1979.

[78] S. Minton, M. Johnston, A. Philips, and P. Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings AAAI-90*, pages 17 24, 1990.

[79] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proceedings AAAI-92*, pages 459 465, 1992.

[80] R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28(2):225 233, 1986.

[81] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95 132, 1974.

[82] B. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188 224, 1989.

[83] 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*, 5(3):137 171, 1991.

[84] B. Nadel and X. Wu. Multiple abstraction levels in automobile transmission design: Constraint satisfaction formulations and implementations. *International Journal of Expert Systems: Research and Applications*, pages 489–559, 1992.

[85] R. Ovans and R. Davison. An interactive constraint-based expert assistant for music composition. In *Proceedings CSCSI, AI'92*, pages 76–81, 1992.

[86] E. Palmer. *Graphical Evolution*. Wiley, 1985.

[87] P. Prosser. A reactive scheduling agent. In *Proceedings IJCAI-89*, pages 1004–1009, 1989.

[88] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. Technical Report AISL-46-91, University of Strathclyde, 1991.

[89] P. Prosser. BM+BJ=BMJ. In *Procceedings of the Conference on Artificial Intelligence for Applications*, pages 257–262, 1993.

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

[91] P. Prosser. Binary constraint satisfaction problems: Some are harder than others. In *Proceedings ECAI-94*, pages 95–99, 1994.

[92] P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. Technical Report AISL-49-94. University of Strathclyde, Glasgow, Scotland, 1994.

[93] P. Prosser. Forward checking with backmarking. In M. Meyer, editor, *Constraint Processing*, LNCS 923, pages 185–204. Springer-Verlag, 1995.

[94] P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1996.

[95] P. Prosser, C. Conway, and C. Muller. A constraint maintenance system for the distributed resource allocation problem. *Intelligent Systems Engineering*, 1(1):76–83, 1992.

[96] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. In *Proceedings ECAI-90*, pages 550 556, 1990.

[97] D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings ECAI-94*, pages 125 129, 1994.

[98] D. Sabin, M. Sabin, R. Russell, and E. Freuder. A constraint-based approach to diagnosing software problems in computer networks. In *Proceedings of the International Conference on the Principles and Practice of Constraint Programming*, pages 463 480, 1995.

[99] H. Simonis. Test generation using the constraint logic programming language CHIP. In *Proceedings of the International Conference on Logic Programming*, pages 101 112, 1989.

[100] J. Siskind. Lexical acquisition as constraint satisfaction. Technical Report IRCS-93-41, University of Pennsylvania, Institute for Research in Cognitive Science, 1993.

[101] B. Smith. Using a local improvement algorithm to solve a constraint satisfaction problem in rostering. In J. Dorn and K. Froesch[1], editors, *Scheduling of Production Processes*, chapter 9, pages 94 102. Ellis Horwood, 1993.

[102] B. Smith. Phase transition and the mushy region in constraint satisfaction problems. In *Proceedings ECAI-94*, pages 100 104, 1994.

[103] B. Smith and M. Dyer. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, pages 155 181, 1996.

[104] B. Smith and S. Grant. Sparse constraint graphs and exceptionally hard problems. In *Proceedings IJCAI-95*, pages 646 651, 1995.

[105] R. Sosic and J. Gu. A polynomial time algorithm for the n-queens problem. *SIGART Bulletin*, 1(3):7 11, 1990.

[106] R. Sosic and J. Gu. 3000000 queens in less than one minute. *SIGART Bulletin*, 2(2):22–24, 1991.

[107] M. Stefik. Planning with constraints (MOLGEN: Part 1). *Artificial Intelligence*, 16:111–140, 1981.

[108] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[109] E. Tsang, J. Borrett, and A. Kwan. An attempt to map the performance of a range of algorithm and heuristic combinations. In *Proceedings of the AI and Simulated Behaviour Conference*, pages 203–216, 1995.

[110] P. van Beek and D. Manchak. The design and experimental analysis of algorithms for temporal reasoning. *The Journal of Artificial Intelligence Research*, 4:1–18, 1996.

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

[112] P. Van Hentenryck. Constraint logic programming. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 274–275. Wiley, New York, 1992.

[113] P. Van Hentenryck and M. Dincbas. Forward checking in logic programming. In *Proceedings of the International Conference on Logic Programming*, pages 229–255, 1987.

[114] P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58:113–159, 1992.

[115] P. van Run. Domain independent heuristics in hybrid algorithms for CSP's. Master's thesis, University of Waterloo, 1994.

[116] R. Walker. An enumerative technique for a class of combinatorial problems. In *Combinatorial Analysis (Proc. Symp. Applied Math, vol. X)*, pages 91–94. American Mathematical Society, 1960.

[117] D. Waltz. Understanding line drawings of scenes with shadows. In *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, New York, 1975. First published as MIT Technical Report AI 271, 1972.

[118] C. Williams and T. Hogg. Using deep structure to locate hard problems. In *Proceedings AAAI-92*, pages 472–477, 1992.

[119] C. Williams and T. Hogg. Extending deep structure. In *Proceedings AAAI-93*, pages 152–157, 1993.

[120] C. Williams and T. Hogg. The typicality of phase transitions in search. *Computational Intelligence*, 9(3):221–238, 1993.

[121] M. Zweben and M. Eskey. Constraint satisfaction with delayed evaluation. In *Proceedings IJCAI-89*, pages 875–880, 1989.