Phase Transition Behaviour in Constraint Satisfaction Problems

by

Stuart Alexander Grant

Submitted in accordance with the requirements for the degree of Doctor of Philosophy.

The University of Leeds
School of Computer Studies

October 1997

The candidate confirms that the work submitted is his own and that appropriate credit has been given where reference has been made to the work of others.

Abstract

Many problems in artificial intelligence and computer science can be formulated as constraint satisfaction problems (CSPs). A CSP consists of a set of variables among which a set of constraints are imposed, with a solution corresponding to an assignment for every variable such that no constraints are violated. Most forms of CSP are NP-complete.

Recent research has shown that the CSP exhibits a *phase transition* as a control parameter is varied. This transition lies between a region where most problems are easy and soluble, and a region where most problems are easy but insoluble. In the intervening phase transition region, the average problem difficulty is greatest. Phase transition behaviour can be exploited to create test beds of hard and easy problems for CSP algorithms. In this thesis, we study the phase transition of the binary CSP and examine various aspects of complete search algorithms for it.

The phenomenon of exceptionally hard problems (‘ehps’) is examined in detail: these are rare searches on easy problems which become exceptionally expensive for a particular complete algorithm following a poor early search move. An explanation for the occurrence of ehps is proposed, and the relative susceptibility of certain algorithms to the phenomenon is explored.

We then show that the phase transition paradigm can be applied to two tasks of polynomial cost complexity: attempting to establish arc and path consistency in a CSP. Phase transition behaviour analogous to that found when searching for a solution is demonstrated for these tasks, and the effectiveness and cost of establishing arc and path consistency is examined.

The theme of establishing consistency in CSPs is extended by studying an algorithm which maintains arc consistency during search. Its performance is compared with that of an algorithm which maintains a lower level of consistency, and it is shown that the higher level of consistency reduces average search cost and ehp behaviour on many types of CSP.

Finally, the subject of dynamically selecting the variable to instantiate at each stage in the search process is considered. We compare a number of heuristics which attempt to select the variable most likely to lead to failure, and show that the supposed principle behind these appears to be fundamentally flawed.

Acknowledgements

I would like to thank Barbara Smith, my supervisor, for her outstanding guidance, support and cooperation over the last three years. Our close collaboration has proved to be both productive and enjoyable, and I have received a standard of supervision which has been the envy of my peers. I am also grateful to the School of Computer Studies for funding my studentship, and to my PhD examiners, Edward Tsang and Tony Cohn, for their constructive comments.

Special thanks go to Patrick Prosser, who some four years ago opened my eyes not only to the field of constraint satisfaction, but also to the whole idea of scientific research. Even when things looked to have gone all wrong, Patrick gave me encouragement, and look how things ended up!

Being a (founder) member of the APES group has provided me with a tremendous forum to discuss ideas, and also to see how good research is done. Many thanks go to Ian Gent, Toby Walsh, Craig Brind, Dave Clark, Ewan MacIntyre, Isla Ross, Paul Shaw and Iain Buchanan, as well as Barbara and Patrick.

This work has been partly funded by British Telecom plc, and I would like to thank Nader Azarmi, David Lesaint, Barry Crabtree and Hyacinth Nwana for their help, support and advice (and their money).

Respect and thanks go to all my friends, particularly Steven, Fred, Mark, Russ and Nana, who are all in the same boat, and Leigh and David, who've seen me in this state more than once.

This work is dedicated to my parents, Mary and Lindsay, and to my sister, Suzy. Without their love and support it would not have been possible.

Declarations

Some parts of the work presented in this thesis have been published in the following articles:

Chapter 3

I. P. Gent, S. A. Grant, E. MacIntyre, P. Prosser, P. Shaw, B. Smith and T. Walsh. 1997. How Not To Do It. Research Report 97.27, School of Computer Studies, University of Leeds.

Chapter 5

B. M. Smith and S. A. Grant. 1995. Sparse Constraint Graphs and Exceptionally Hard Problems. In C. S. Mellish., ed., *Proceedings of the 14th International Joint Conference on Artificial Intelligence - IJCAI-95*, volume 1, 646–651. Morgan Kaufmann.

B. M. Smith and S. A. Grant. 1994. Sparse Constraint Graphs and Exceptionally Hard Problems. Research Report 94.36, School of Computer Studies, University of Leeds.

B. M. Smith and S. A. Grant. 1995. Where the Exceptionally Hard Problems Are. In *Proceedings of the CP-95 Workshop on Studying and Solving Really Hard Problems*, 172–182. Laboratoire d'Informatique de Marseille, France.

Chapter 6

S. A. Grant and B. M. Smith. 1996. The Arc and Path Consistency Phase Transitions. In E. C. Freuder., ed., *Principles and Practice of Constraint Programming - CP-96*, volume 1118 of *Lecture Notes in Computer Science*, 541–542. Springer-Verlag. Extended abstract with poster.

S. A. Grant and B. M. Smith. 1996. The Arc and Path Consistency Phase Transitions. Research Report 96.09, School of Computer Studies, University of Leeds.

Chapter 7

S. A. Grant and B. M. Smith. 1995. The Phase Transition Behaviour of Maintaining Arc Consistency. Research Report 95.25, School of Computer Studies, University of Leeds.

S. A. Grant and B. M. Smith. 1996. The Phase Transition Behaviour of Maintaining Arc Consistency. In W. Wahlster., ed., *Proceedings of the 12th European Conference on Artificial Intelligence - ECAI-96*, 175–179. John Wiley & Sons, Ltd.

Chapter 8

B. M. Smith and S. A. Grant. 1997. Trying Harder to Fail-First. Research Report 97.45, School of Computer Studies, University of Leeds.

Contents

List of Figures

List of Tables

Abbreviations Used

Basic Terms

CSP, CSPs	Constraint Satisfaction Problem(s), usually binary
SAT	Boolean Satisfiability
3-SAT	SAT where all clauses contain exactly 3 literals
TSP	Travelling Salesperson Problem
ehp, ehps	Exceptionally hard problem(s)

Basic Search Algorithms

BT	Chronological Backtracking
BJ	Backjumping
CBJ	Conflict-Directed Backjumping
BM	Backmarking
FC	Forward Checking
MAC	Maintaining Arc Consistency

Hybrid Search Algorithms

BMJ	Backmarking with Backjumping
BM-CBJ	Backmarking with Conflict-Directed Backjumping
FC-BJ	Forward Checking with Backjumping
FC-CBJ	Forward Checking with Conflict-Directed Backjumping
MAC-BJ	Maintaining Arc Consistency with Backjumping
MAC-CBJ	Maintaining Arc Consistency with Conflict-Directed Backjumping

Consistency Algorithms

AC	Arc Consistency / Arc Consistent
PC	Path Consistency / Path Consistent
AC3, AC4, AC6, AC7	Alternative arc consistency algorithms
PC2, PC4	Alternative path consistency algorithms

Variable Ordering Terms

DVO	Dynamic Variable Ordering
SVO	Static Variable Ordering
FF	Variable Ordering using the Fail-First principle
FFdeg, FF2, FF3, FF4	Variants of the Fail-First strategy
BZ, DD, kappa	Alternative dynamic variable ordering strategies

Chapter 1

The Constraint Satisfaction Problem and its Phase Transition

Consider the task of creating an examination timetable for a large university. Every one of the exams must be assigned a room, time and invigilating member of staff. These assignments are subject to a number of *constraints* which must not be violated, such as: the capacity of the rooms must be sufficiently large to house all the candidates; two exams cannot run concurrently if one or more students plan to sit both; staff must be available for invigilation duty at the right time; disabled students must be placed in rooms with sufficient facilities; and no student must sit more than two exams in any one day and six exams in any one week. Of all the permutations of $\langle$*room, time, staff*$\rangle$ assignments which can be given to the set of examinations, only those which satisfy all of the constraints constitute valid solutions. Such a timetabling problem is an example of a *constraint satisfaction problem*.

Another problem which can be formulated as a constraint satisfaction problem is a simple chess puzzle known as the n-queens problem. The objective here is to assign n chess queens to positions on an $n \times n$ chess board, such that no two queens threaten each other. The constraints specify, therefore, that no two queens must lie on the same row, column or diagonal on the board.

The entities in a constraint satisfaction problem to which values must be assigned are its *variables*. Even small numbers of variables in a problem can create a vast search space, and the solutions to the problem, if there are any, may be widely scattered over this space. These two factors combined can often make the time taken to search for a solution unfeasibly long.

A vast search space does not necessarily mean an intractable problem, however, and many constraint satisfaction problems are in fact very easy. If the examination timetable described above were to involve, say, only three exams involving a handful of students with plenty of suitable accommodation, staff and time available, then the constraints are so loose that a solution is very easy to find. Similarly, if twenty exams must be scheduled into only two days with one room available, then the problem is hopelessly over-constrained and it is easy to prove that no solutions exist. The difficult problems lie between these extremes, when it is not so clear whether a solution can be found or not.

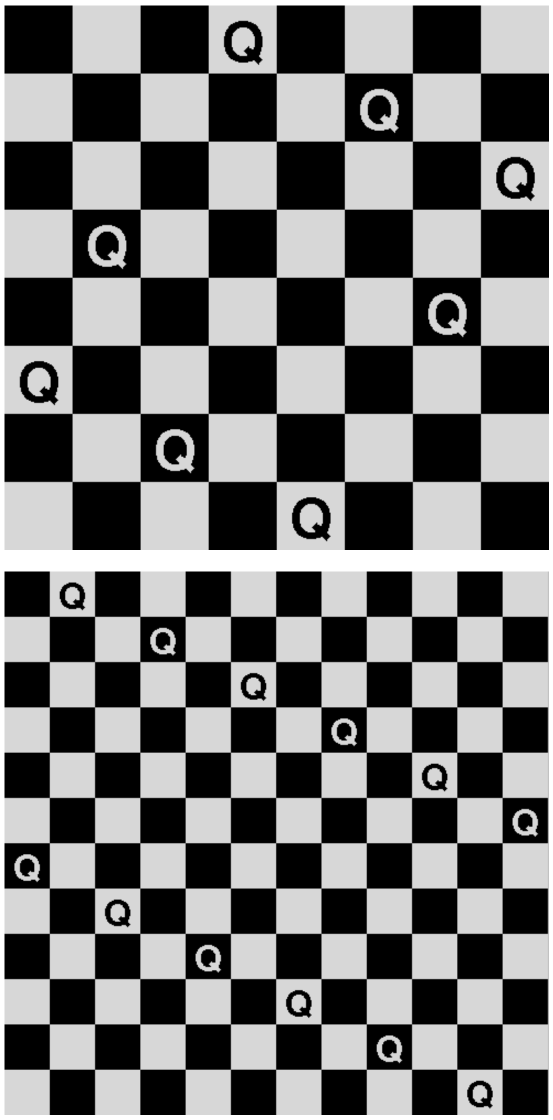


Figure 1.1: Example solutions to the 8-queens problem and the larger, easier, 12-queens problem.

Problem size is not necessarily a guide to difficulty, either. The *n*-queens problem actually becomes less tightly constrained as the number of queens (and the size of the board) increases (Tsang 1993), which essentially makes the problem easier to solve. To understand this, consider that an increase in *n* from, say, eight to nine adds one queen to the problem, but seventeen additional board squares. Figure 1.1 illustrates an example solution to the 8-queen and 12-queen problems. A queen placed on the 8 row board can rule out as many as 27 of the 63 remaining positions (43%), while a queen on the 12 row board can rule out at most 43 of the other 143 positions (30%). The very specific properties of this problem prompt Tsang to advise the use of caution if using *n*-queens to benchmark the performance of search algorithms.

The behaviour of constraint satisfaction problems varies greatly, depending on the problem size and its level of constrainedness. Recent insights into the circumstances under which problems are easy or hard, and whether they have solutions or not, have increased understanding of how they can be searched more efficiently. This in turn has promoted more rigourous testing of search techniques.

The behaviour of the constraint satisfaction problem and the study of search techniques for it form the basis of this thesis. We begin in this chapter by noting the importance of constraint satisfaction in many areas of computer science, and by formally defining the constraint satisfaction problem. The task of searching these problems is discussed, and the pattern of behaviour known as the *phase transition* is introduced. The role of phase transitions in the study of search techniques is then examined and an overview of the structure of the thesis is given. A number of important terms are introduced which are used throughout this thesis. These are introduced in bold type.

1.1 The Ubiquity of Constraint Problems

The constraint satisfaction problem is a simple but extremely powerful paradigm for representing many types of problem which arise in the fields of artificial intelligence and computer science. (Meseguer 1989), (Kumar 1992) and (Tsang 1993) discuss applications of the constraint satisfaction problem in areas including:

- Belief Maintenance (Dechter and Pearl 1988)
- Configuration (Mittal and Falkenhainer 1990)
- Databases (Dechter and Pearl 1989)
- Design (Navinchandra and Marks 1987)
- Diagnosis (Sabin *et al.* 1995)
- Machine Vision (Montanari 1974)
- Planning (Kautz and Selman 1992)
- Scheduling (Fox 1987)
- Temporal Reasoning (Allen 1983)
- Truth Maintenance (de Kleer 1986; de Kleer 1989)

The power of formulating problems in terms of entities linked by constraints lies in the ability to create a representation that closely mirrors the actual problem. Such an intuitive representation makes solutions easier to understand, and assists the process of creating search heuristics which exploit characteristics of the problems.

The ubiquity of constraint satisfaction problems throughout the world of computation makes efficient techniques for solving them highly desirable. This desire has led to the development of constraint programming tools and languages, which provide built-in constraint handling functions and the facilities to represent problems easily. Well known constraint programming languages include CHIP (Constraint Handling in Prolog) (Simonis and Dincbas 1987) and the more recent C++ based ILOG Solver (Puget 1994).

1.2 The Constraint Satisfaction Problem

Constraint satisfaction problems (CSPs) appear in many forms, linked by a number of basic properties. To provide a context for our study of the CSP, a number of important formal definitions are given below. These are followed by discussion of the issues relating to finding solutions for the problem, the conditions which make most forms of the CSP fundamentally difficult to solve, and strategies which can reduce the search space of a CSP prior to search.

1.2.1 Formal definitions

The following series of definitions relating to the constraint satisfaction problem and associated properties are based on those provided by (Tsang 1993).

A **constraint** is defined by a pair (V,R): V is a set of variables $\{v_i,\ldots,v_j\}$, each with a **domain** of possible values D; R is a relation such that $R \subseteq D_i \times \ldots \times D_j$. Informally, a constraint specifies the allowed tuples of values for the variables involved.

The **arity** of a constraint denotes the number of variables involved: binary constraints, for example, involve pairs of variables, while ternary constraints involve triples.

A general **constraint satisfaction problem** (CSP) is defined by the tuple (V,D,C): V is a set of variables; D is a function mapping each variable in V to its domain of possible values; C is a (possibly empty) set of constraints, each involving an arbitrary subset of V.

A **finite constraint satisfaction problem** imposes the restriction that the number of variables, the sizes of their domains, and the number of constraints are finite. For brevity, we use the terms *constraint satisfaction problem* and CSP to denote a finite constraint satisfaction problem throughout the rest of this thesis.

A **constraint graph**, defined by a pair (V,E), can be associated with any CSP: the vertices, V, correspond to each problem variable; the edges, E, are placed between pairs of vertices whose corresponding variables are mutually involved in one or more constraints of any arity. Properties of the constraint graph for a CSP include the density of the constraints, the degree of vertices (variables) and the bandwidth.

A **solution** to a CSP is a total assignment, S, such that for each variable $v \in V$, $S(v) \in D_v$ and for each constraint $(\{v_i,\ldots,v_j\},R)$, $\{S(v_i),\ldots,S(v_j)\} \in R$. Informally, a solution is an assignment to every variable of a value that is a member of its domain, and that does not violate any of the constraints placed on the variables.

A CSP for which there exists one or more solutions is said to be **soluble**, while one without solutions is **insoluble**.

1.2.2 Issues involved with the CSP

The solubility or insolubility of any (finite) CSP is determinable, since the set of all total assignments to the problem is finite. Thus, the CSP avails itself to a **search process** which can look for solutions.

Tasks associated with searching CSPs include: determining whether or not a solution exists (the **decision problem**); finding a solution if one exists (the **search problem**); finding all solutions (the **enumeration problem**); determining the number of solutions (the **counting problem**); and finding the optimal solution, given an optimality criterion (the **optimisation problem**).

The **arity** of a CSP is defined by the maximum arity among its constraints. The studies presented in this thesis are restricted to dealing with the **binary** constraint satisfaction problem. A binary CSP involves only unary or binary constraints. However, since unary constraints on variables can be dealt with simply by removing disallowed values from their domains, the binary CSPs we will deal with in practice contain only binary constraints.

Concentrating solely on binary CSPs does not necessarily lead to a loss of generality in our analysis. Any CSP of higher arity can theoretically be reduced to a binary CSP, and (Tsang 1993) presents two methods for this. It should be noted, however, that the representation of some high-arity constraints may require a number of binary constraints that is exponential in the arity of the original constraint.

1.2.3 Computational complexity

The CSP in its general form is NP-complete (Garey and Johnson 1979). This means that it is unlikely that an algorithm for the search problem exists that does not have a worst-case time complexity exponential in the size of the problem. A CSP with n variables, each with domains of size m, has m^n total assignments, giving rise to a worst-case time complexity that is potentially $O(m^n)$.

Even heavily restricted forms of CSP are NP-complete. To illustrate this, consider a CSP consisting of Boolean variables (domain sizes of 2) and ternary constraints (arity 3). This class of CSP subsumes the 3-satisfiability (3-SAT) problem, which was the first computational task shown to be NP-complete by (Cook 1971). Cook showed that if the arity of a Boolean Satisfiability (SAT) problem is greater than 2, the problem is NP-complete.

Similarly, the graph colouring problem is NP-complete when 3 or more colours are available (Garey and Johnson 1979). 3-colouring can be transformed into a binary CSP with domain sizes of 3. A CSP with arity 2 and only boolean variables has a worst-case time complexity polynomial in the problem size, but almost any relaxation of this will make the problem NP-complete.

1.2.4 Constraint propagation

The complexity of the CSP stems from interaction between the explicitly defined constraints, which produces many more implicit constraints. One approach to simplifying, conceptually, the search process is to **propagate** the effects of the explicit constraints around the problem prior to search. Such a preprocessing step establishes some level of **consistency** in the CSP. The level of consistency achieved by a constraint propagation process is denoted by the term ***k*-consistency**, for which (Meseguer 1989) and (Kumar 1992) provide similar definitions, summarised here as:

> A set of variables is k-consistent if for each set of $k-1$ variables with values satisfying all the constraints among them, it is possible to find a new value for a new variable such that all the constraints among the k variables are satisfied. If the set of variables is j-consistent for all all $j \leq k$, then it is *strongly* k-consistent.

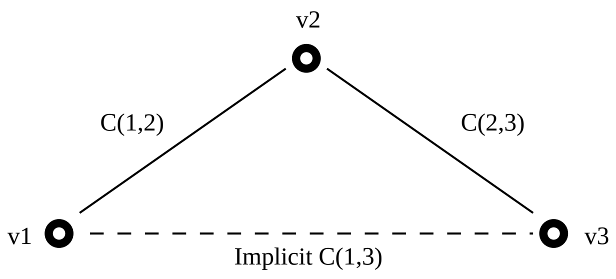


Figure 1.2: An implicit binary constraint created by two explicit ones.

Strong 2-consistency is known as **arc consistency**. Informally, all pairs of variables in an arc consistent CSP are mutually consistent, in that any value in the domain of one variable is consistent with at least one value in the domain of every other variable. Establishing arc consistency in a CSP often results in the removal of inconsistent (or *unsupported*) values from the domains of variables, thereby reducing the size of the problem's search space. Values that are unsupported do not form part of any solution to the CSP. If the process of making a CSP arc consistent causes a **domain wipe-out** for a variable, then the problem is proven to be insoluble.

Strong 3-consistency is known as **path consistency**. Here, all triples of variables in the CSP are made mutually consistent. For a binary CSP, establishing path consistency effectively makes explicit the implicit constraints that bind triples of variables. Hence, this process may add extra explicit binary constraints to the problem in addition to removing unsupported values from variable domains. To illustrate this, consider three variables, v_1, v_2 and v_3, linked by two binary constraints as shown in Figure 1.2. The path consistency process calculates the implicit binary constraint between v_1 and v_3 that arises from the combined effects of the two explicit constraints which indirectly link them. If this new constraint is non-trivial, it is added to the problem and takes part in the arc consistency phase of the process.

If a CSP is strongly k-consistent, where k is the number of variables in the problem, then solutions may be obtained without any need for search. The task of establishing such a level of consistency is, however, NP-complete itself (Tsang 1993).

1.3 Searching CSPs

Three important properties are associated with any search process: **soundness**, **completeness** and **termination**. Given a problem, an algorithm which is sound will find only valid solutions, and one that is complete will find all solutions given enough time. The termination property is usually a by-product of completeness.

Any credible algorithm for the CSP must be sound, though it need not necessarily be complete. **Complete algorithms** for the CSP usually involve some systematic search process which methodically explores the search space of the problem. **Incomplete algorithms** for the CSP sacrifice the property of completeness in return for greater mobility to jump around areas of the search space, usually using some stochastic exploration technique. Brief reviews of complete and incomplete methods for searching the CSP are presented below.

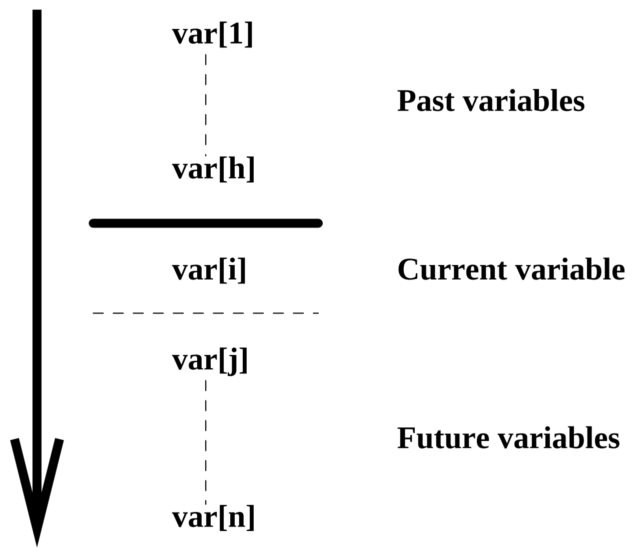


Figure 1.3: The three variable states during backtracking search.

1.3.1 Complete algorithms

Many systematic search algorithms exist to find a solution to a CSP or show that it is insoluble. These algorithms tend to be refinements of **backtracking search** (Golomb and Baumert 1965). Backtracking search aims to build and extend **partial solutions** until a complete solution is found. This is achieved by **instantiating** the problem variables with values from their domains, using a pre-defined **instantiation ordering**.

At any stage in a backtracking search process, each variable is in one of three possible states:

- **Past variables** have been instantiated with values from their domains that are consistent with the instantiations of all other past variables.
- The **present variable** is about to be instantiated with a value from its domain. This must be consistent with the instantiations of all past variables.
- **Future variables** have not yet been instantiated.

Figure 1.3 illustrates these variable states graphically. The search process consists of a number of **search moves** with the following characteristics:

- A **forward** search move successfully instantiates the present variable and moves on to the next variable in the instantiation order.
- A **dead end** move occurs when all values of the current variable are inconsistent with the current partial solution.
- A **backward** search move follows a dead end. The search steps back to a past variable and tries a new instantiation for it.
- A solution is found if the search moves forward past the final variable in the instantiation order.

- The search is exhausted if it steps back beyond the first variable. It then terminates.
- Insolubility is proved if the search terminates without finding any solutions.

Each forward move in a backtracking search requires some form of **consistency checking** to validate the instantiation being made. This can be performed either by checking backward against the instantiations of the past variables, or by checking forward against the domains of the future variables and making them consistent to some extent with the current instantiation. These styles of forward move are known as **lookback** and **lookahead** respectively.

Similarly, there are two possible styles of backward search move. **Chronological backtracking** allows the search to step back only to the most recently instantiated variable, while **backjumping** allows the search to jump back past a number of instantiated variables.

The generic backtracking algorithm (Golomb and Baumert 1965) uses lookback consistency checking and chronological backtracking. Refinements to this algorithm which introduce backjumping capability include Backjumping (Gaschnig 1979) and Conflict-Directed Backjumping (Prosser 1993). Algorithms which use lookahead techniques with chronological backtracking include Backmarking (Gaschnig 1977; Gaschnig 1979), Forward Checking (Haralick and Elliott 1980) and algorithms which maintain arc consistency during search (Gaschnig 1979; Sabin and Freuder 1994). The sophistication of lookahead and backjumping techniques can be combined to produce hybrid algorithms (Prosser 1993). Complete algorithms for the CSP are discussed in detail in Chapter 2.

1.3.2 Incomplete methods

CSPs with large numbers of variables often prove to be intractable to complete search methods, due to the overwhelming search spaces involved. Incomplete search techniques, however, can often find solutions to these problems within a reasonable amount of time by sacrificing completeness for efficiency.

Incomplete algorithms tend not to terminate naturally. This means that they may be have to be terminated forcefully before they can find a solution, and also means that they are incapable of proving insolubility in a problem.

Incomplete search algorithms tend to use stochastic techniques and are often based on physical models (eg. hill climbing and simulated annealing) or biological models (eg. genetic algorithms and neural networks). Well known hill climbing procedures include GSAT (Selman *et al.* 1992) and the min-conflicts procedure (Minton *et al.* 1992). The GENET system (Wang and Tsang 1991) is an example of a neural network for constraint satisfaction.

1.4 Phase Transition Behaviour

Many types of NP-complete search problems are known to exhibit *phase transition behaviour* as a control parameter is varied. This behaviour can be observed in populations of problems whose characteristics, apart from those defined by the control parameter, remain relatively homogenous. (Cheeseman *et al.* 1991) first identified the phase transition in graph colouring problems as the interface between a region where almost all problems have many solutions and are relatively easy to solve, and a region where almost all problems have no solution and their insolubility is relatively easy to prove. In this intervening region, the probability of problem solubility falls from close to 1 to close to 0. Additionally, they observed empirically that the average cost of searching these problems reaches a peak in this region.

Phase transition behaviour has been reported in an increasing number of computational problems, including boolean satisfiability (SAT) problems (Mitchell *et al.* 1992; Kirkpatrick and Selman 1994; Crawford and Auton 1996; Gent *et al.* 1996a), Hamiltonian paths (Cheeseman *et al.* 1991), the travelling salesperson problem (TSP) (Gent and Walsh 1995a) and number partitioning (Gent and Walsh 1996a). (Williams and Hogg 1993) suggest that this phenomenon exists for many general problems of search, and phase transitions are generally believed to be ubiquitous amongst NP-complete problems.

Section 1.2.3 notes that graph colouring is a restricted case of a binary CSP. Phase transition behaviour in the binary CSP was observed as early as (Gaschnig 1979), though only properly recognised in the light of the seminal paper by Cheeseman *et al*.

The initial identification of phase transition behaviour has prompted a flurry of subsequent work on the phenomenon. (Williams and Hogg 1994) have developed approximations to the cost of finding the first solution and to the probability that a problem is soluble, both for specific classes of constraint satisfaction problem (graph colouring, SAT) and for the general case. These approximations are based on the asymptotic behaviour of the problems as the number of variables becomes large; an instantaneous phase transition is predicted in the limit, where a step change in the probability of problem solubility coincides with the peak in the cost of finding a solution at a critical value of the control parameter.

Phase transitions in finite-sized problems tend not to be instantaneous, however, and occur over a range of values of the control parameter. In drawing an analogy between the phase transition in finite binary CSPs and the physical phase transitions modelled by applied mathematicians, (Smith 1994) terms the region over which probability of problem solubility falls from 1 to 0 the *mushy region*. Significant theoretical work on predicting the location of the phase transition in binary CSPs is presented by (Smith and Dyer 1996).

(Mitchell *et al.* 1992) empirically show that the peak in average cost to find the first solution to SAT problems occurs at the value of the control parameter where 50% of the instances sampled are soluble. This feature has been demonstrated empirically for other types of problem, for instance in binary CSPs by (Prosser 1996). Theoretical work on predicting the location of the point of 50% solubility is presented by (Crawford and Auton 1996), who term this point the *crossover point*.

An illustration of phase transition behaviour in a type of computational problem is provided by Figure 1.4. The *x*-axis of both plots shows the control parameter of the problems, which in this example are a type of binary CSP. Ensembles of sample problems are searched at each point along the control parameter range, with the *y*-axis of the top plot showing how the median cost of these searches varies as the control parameter changes[1]. The *y*-axis of the bottom plot shows the

[1]The *y* units used in this plot are not important, but can be regarded as 'time'

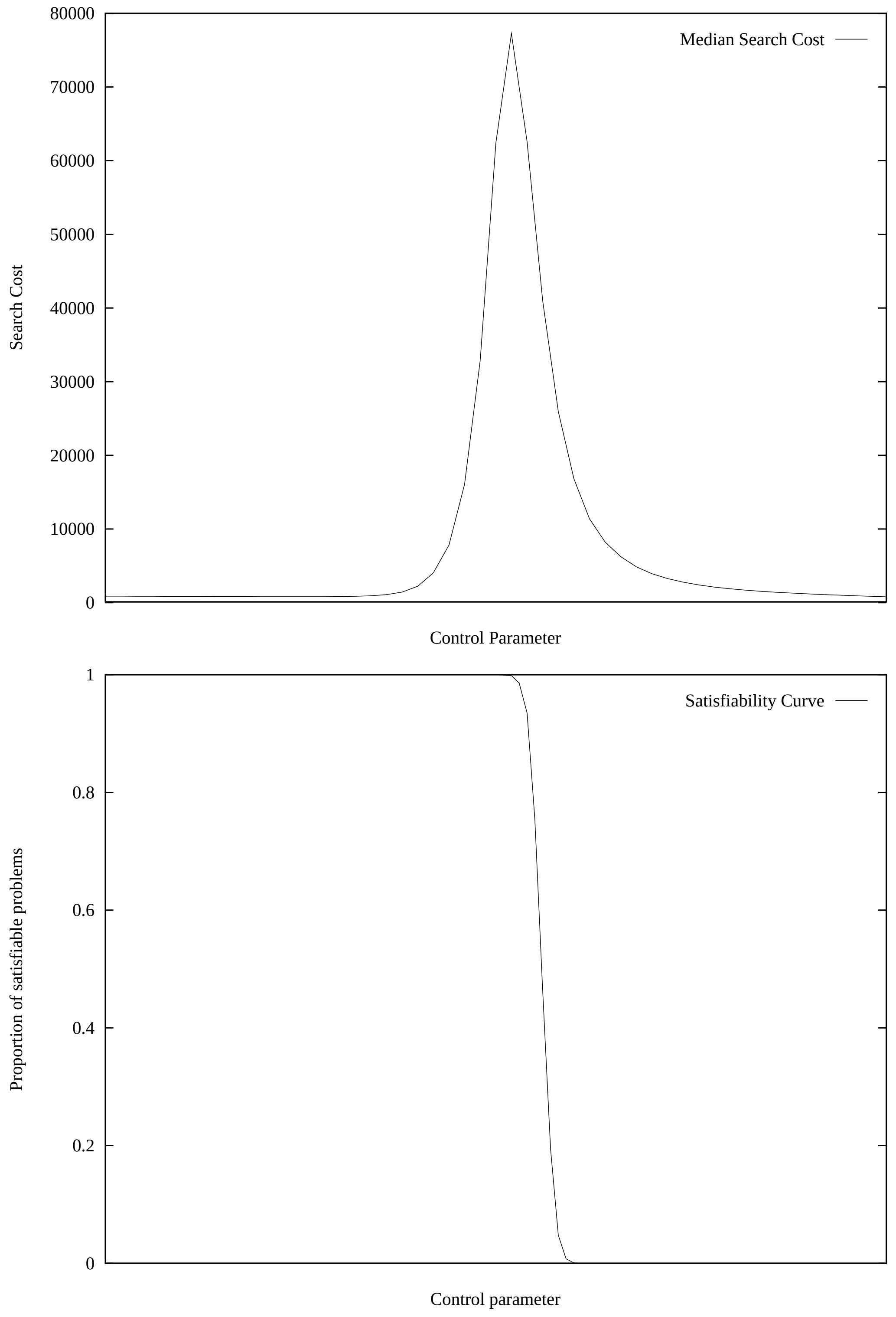


Figure 1.4: An illustration of phase transition behaviour in a computational search problem.

proportion of soluble problems observed for each ensemble.

The plots show that low values of the control parameter correspond with an *easy-soluble* region, and that high values correspond with an *easy-insoluble* region. In between, search cost rises to a sharp peak, and there is a mushy region where problem populations are a mixture of soluble and insoluble instances. Comparison of the plots shows that the observed peak in median search cost does coincide with the point at which around half of the problems sampled are soluble. It is noticeable that the median cost at the crossover point is almost two orders of magnitude greater than that in the easy regions.

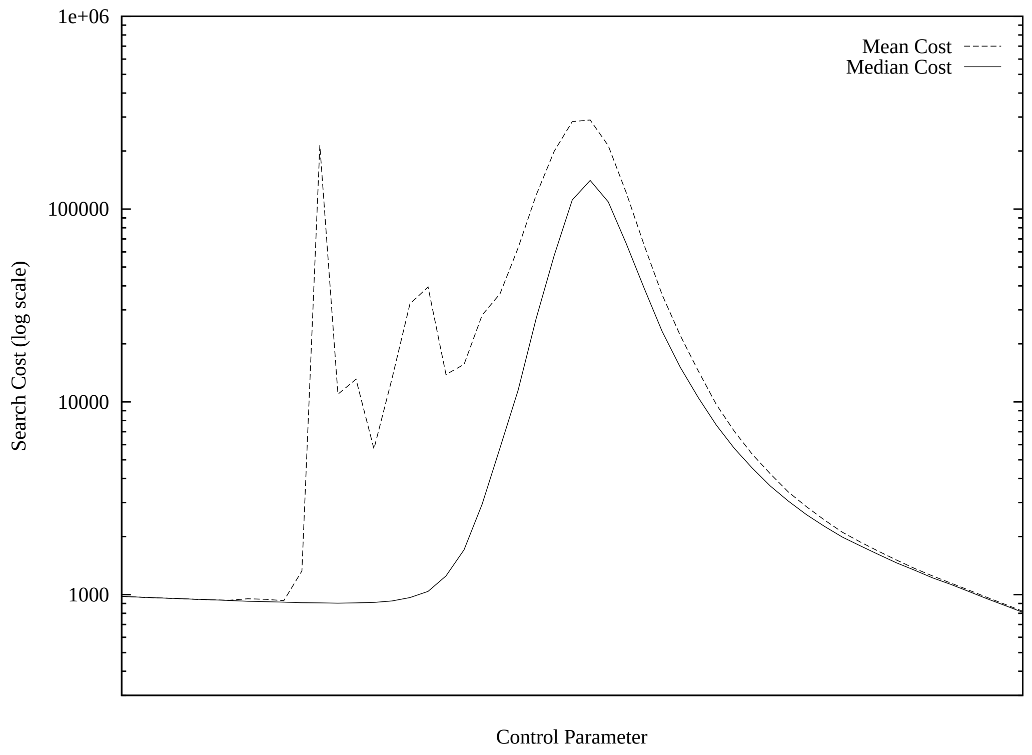
To summarise, at the heart of what we will term the *phase transition model* is the notion that, for certain types of problem, the likelihood that solutions exist can be governed by adjusting one or more parameters. In cases where solutions are very likely, we should expect the cost of finding one to be low. Similarly, where solutions are very unlikely, we can expect the cost of proving that there are none to be low. It is on the problems that are as likely to be soluble as insoluble that the cost of resolving the issue may be very high.

1.5 Exceptionally Hard Problems

Recent studies have highlighted a phenomenon that complicates the phase transition model. The existence of *exceptionally hard problems* (‘ehps’) has been reported in graph colouring (Hogg and Williams 1994), SAT (Gent and Walsh 1994a), and binary CSPs (Smith 1994; Frost and Dechter 1994). These studies show that although there is a well-defined peak in the *median* cost of finding a solution in the region of the phase transition, this is often not where the hardest individual instances occur. Given a large sample of problems, individual problems which are very hard to solve with a particular algorithm may occur in the region where most problems are relatively easy to solve. These searches may be so hard that their cost significantly affects the value of the mean cost; it is for this reason that authors reporting phase transition behaviour have often used the median rather than the mean as a measure of average difficulty.

Figure 1.5 illustrates the phenomenon of exceptionally hard problems occurring outside of the mushy region. The upper graph plots the mean and median costs of finding a solution (using a log scale) against the control parameter. The lower graph shows the median and higher cost percentile levels, up to the maximum observed. The mean and median costs are similar and stable, except in the region leading up to the phase transition, where the mean cost becomes high and very erratic. Looking at the higher percentiles, stable behaviour in cost of search is observed right up to the 99% level at all values of the control parameter. In the region leading up to the phase transition, however, it can be seen that there are some outlying searches whose cost could be described as exceptional. In particular, the most expensive search that can be seen lies well into the easy-soluble region of the control parameter. This search is more than six orders of magnitude more expensive than 90% of all others seen at this point, and is more than an order of magnitude more expensive than the hardest observed at the crossover point. These few exceptional searches are sufficiently expensive to affect the mean cost in the way that can be seen.

Ehps appear to be a feature of a particular search algorithm behaving abnormally. Individual



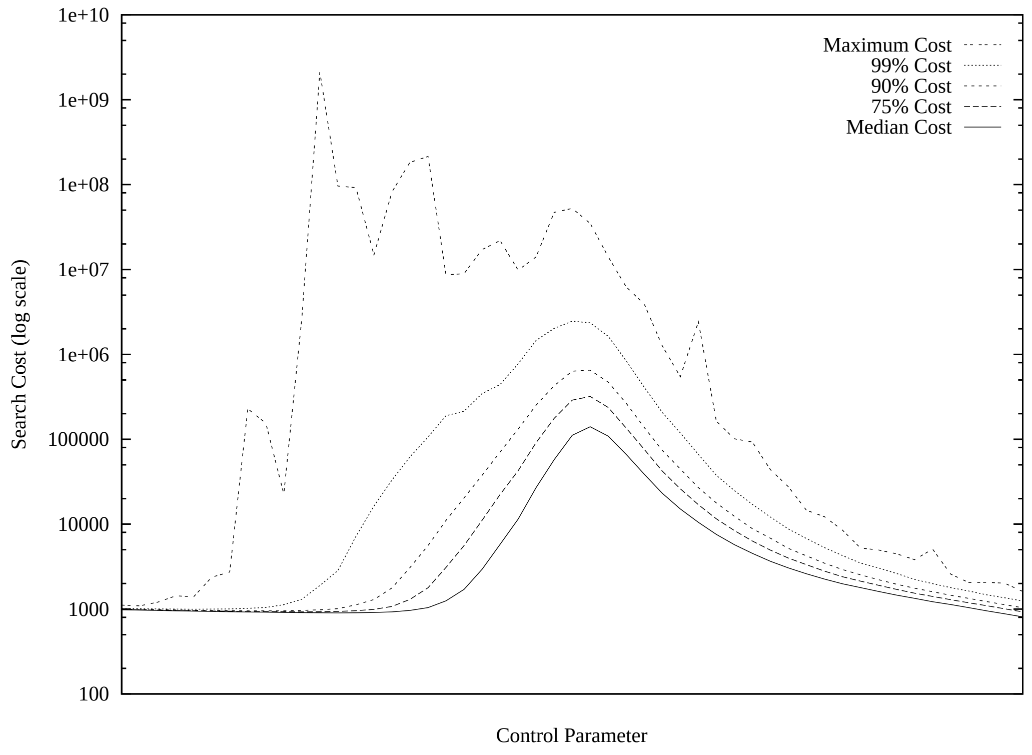


Figure 1.5: An illustration of exceptionally hard problem behaviour in a computational search problem.

problems that are exceptionally hard for one algorithm tend to be easy for any another algorithm. This creates a clear distinction between ehps and the hard problems occurring at the phase transition: phase transition problems are thought to be fundamentally difficult, and expensive to search with any algorithm; ehps are easy problems which appear to highlight deficiencies in a particular algorithm.

To date no complete search method has been shown to be completely immune from ehps, although studies of various algorithms (Smith and Grant 1995a; Smith and Grant 1995b; Davenport and Tsang 1995; Baker 1995) have shown that their incidence and magnitude varies greatly between search methods. It is clearly important, therefore, to consider relative ehp behaviour when comparing the performance of complete algorithms. Ehps appear to be a feature only of complete search methods: no such behaviour has yet been reported for credible incomplete methods (Hogg and Williams 1994; Gent and Walsh 1994b; Davenport and Tsang 1995). The issue of exceptionally hard problem behaviour in complete search methods is explored in detail in Chapter 5.

1.6 Applications of Phase Transition Behaviour

The existence of phase transition behaviour in NP-complete problems presents a great opportunity for increasing knowledge of both hard computational search problems and the algorithms used to search them. The phase transition model shows us that while a class of NP-complete problems contains many instances that are fundamentally difficult, even intractable, these problems reside in a relatively narrow region of the class' parameter space. Outside of these regions, problems are usually easy to solve or prove insoluble (though peculiar phenomena such as ehp behaviour add a level of uncertainty to this).

For well defined types of NP-complete problems, such as binary CSPs, SAT and graph colouring, the phase transition is well understood and theory already exists to make predictions about its characteristics. These principles may, in future, be extended to more complex types of problems containing 'real world' features. If this is achieved, then powerful predictions about the nature of any given search problem might be possible. As an example, consider a large scheduling problem which might require days of computational effort to solve. If the underlying characteristics of this problem could be mapped to a well known set of order parameters, then it may be possible to make predictions like:

- whether the problem has a solution or not.
- the likely number and distribution of solutions.
- how expensive it is likely to be to solve.
- the best search method to apply to the problem.
- if the problem is hard or insoluble, how constraints could be relaxed to move the problem into the 'easy' region.

The ability to make sound predictions about the characteristics of hard computational problems before any search is attempted could take these tasks from the edge of intractability into the realm of real-time processing. However, current understanding of phase transition behaviour places these goals beyond the foreseeable future.

A much more immediate application for the phase transition model, however, lies in the testing and comparison of search techniques. Knowledge of the phase transitions for many types of problem has forced a major re-think of the way that algorithm performance is assessed. There has been a rapid move away from attempting to classify algorithms on the basis of limited testing on sets of homogenous problems, such as the *n*-queens or the zebra (Dechter and Pearl 1988) problem, towards attempts to classify algorithms in terms of performance on large samples of problems of varying size, topology and position in relation to the phase transition. (Tsang *et al.* 1995), for example, use a series of empirical studies to produce a 'map' of good algorithm and heuristic combinations for various types of binary CSP. This data has been used to produce a technique of *adaptive constraint satisfaction* (Borrett *et al.* 1996), which switches to the most appropriate algorithm given the current problem characteristics.

1.7 Overview

The objective of the work presented in this thesis is to study the phase transition of the binary constraint satisfaction problem, and to use the phase transition model as a platform for conducting rigourous empirical studies of search techniques for the CSP. The techniques studied are restricted to complete search algorithms, heuristics and constraint propagation techniques, with those covered introduced in detail in Chapter 2 . The empirical studies to be performed require large populations of CSPs to which the various algorithms can be applied, and the method used for generating these problems is presented in Chapter 3.

Chapter 4 then introduces the framework and methodology used to conduct the empirical studies, discussing the generic format of the experiments and the task of collecting and presenting the data produced. A major objective pursued throughout the thesis, given its empirical nature, is that implementation details and the experimental methodology should be clear and consistent throughout.

Chapters 5 to 8 present the results of our studies into phase transition behaviour in the CSP. These studies focus initially on the problems and their behaviour, but gradually move towards the algorithms and the effect of the phase transition on their performance. Relevant literature and related work specific to these individual chapters is reviewed in detail at the beginning of each.

Chapter 5 looks in depth at the issue of exceptionally hard problems which occasionally occur for complete search methods. The circumstances under which these abnormal searches arise are investigated, and the relative susceptibility of certain algorithms to the phenomenon is explored. Chapter 6 reports the existence of phase transition behaviour associated with the task of establishing levels of consistency in CSPs, and exploits this to gain new insights into the performance of consistency algorithms.

Exceptionally hard problems and the phase transitions found in both full search and establishing consistency are considered in the study of two CSP search algorithms which maintain arc consistency during search, presented in Chapter 7. The study of these algorithms in the context of phase transitions shows that there are types of CSP for which they perform significantly better than more commonly used techniques.

Having looked in depth at the behaviour of complete algorithms for the CSP, Chapter 8 looks at the heuristics with which they are often combined. A theoretical interpretation of the principle behind many popular heuristics is tested empirically and, surprisingly, the results obtained suggest that this principle may be fundamentally flawed.

Finally, Chapter 9 analyses the work that has been presented, drawing conclusions and discussing future related work.

Chapter 2

Complete Algorithms for the CSP

The search techniques for constraint satisfaction problems that are used throughout this thesis are complete methods, based on backtracking search. The particular algorithms used are introduced and discussed below, together with a framework for their implementation that includes searching for one solution, searching for all solutions, and searching using additional heuristic techniques. A number of algorithms which establish arc and path consistency in CSPs before search is undertaken are also discussed.

2.1 Search Algorithms

The backtracking search paradigm is introduced in Section 1.3.1. Recall that this search process involves forward moves, which instantiate variables with consistent values, and backward moves, which undo instantiations when dead ends are reached. Forward search moves may be made using lookback or lookahead techniques, while backward moves may involve a chronological backtrack or a jump back over several variables.

2.1.1 Backtracking and backjumping

Chronological Backtracking (BT) (Golomb and Baumert 1965) is the generic backtracking algorithm, employing the most primitive forms of forward and backward search move. To instantiate the current variable, *i*, BT chooses a value from its domain and checks for consistency with the instantiations of all past variables. If inconsistent, this value is removed from the domain of *i* and the next available value is tried. If the domain of *i* is exhausted, a dead end has been reached and BT backtracks to the most recently instantiated variable, *h*. This instantiation is discarded and removed from the domain of *h*, and the search attempts to move forward again.

Backjumping (BJ) (Gaschnig 1979) employs the same forward move as BT, but attempts to jump back to the cause of a dead end rather than simply to the previous variable. The algorithm records the most recently instantiated past variable, *h*, which precludes a value from the domain of the current variable, *i*, upon consistency checking. If BJ cannot move forward past *i*, then it jumps back to *h*. The reasoning behind BJ is that re-instantiating any of the variables between *i* and *h* is guaranteed to be fruitless since the reason for the dead end at *i* will not have been addressed.

Conflict-Directed Backjumping (CBJ) (Prosser 1993) is a refinement of BJ which attempts to preserve knowledge about the cause of conflicts over a series of backward moves. A *conflict set* is maintained for each variable, which records every past variable precluding values from its domain. If a dead end is reached with the current variable, *i*, CBJ jumps back to the most recently instantiated variable named in its conflict set, *h*. The search knowledge of *i* is carried upwards to *h* by unifying their conflict sets, minus *h* itself. Thus if *h* itself cannot be re-instantiated, CBJ will then jump back to the deepest variable in conflict with either *i* or *h*.

2.1.2 Looking ahead

Backmarking (BM) (Gaschnig 1977) adds a primitive level of lookahead to basic chronological backtracking. It attempts to save consistency checking cost by recording, for each instantiation attempted during search, the most recently instantiated variable (if any) which causes this to fail. Each variable also maintains a record of the deepest variable backtracked to since this information was recorded. If an instantiation is re-attempted at some later point, consistency checks with past instantiations that are unchanged become unnecessary. BM's consistency checking still takes place against past variables, but future variables are notified every time a backtrack occurs. A considerable drawback of the algorithm is that a static order of instantiation must be observed, ruling out the use of dynamic variable ordering (see Section 2.4).

Forward Checking (FC) (Haralick and Elliott 1980) performs its consistency checking against the future, rather than the past, and backtracks chronologically. To instantiate the current variable, *i*, FC checks its selected value against the future (uninstantiated) variables. Inconsistent values in the domains of the future variables are removed: if this process does not result in annihilation of a future variable domain, then FC moves forward to the next variable; otherwise, the effects of checking forward are undone and a new value is tried for *i*. The backward search move of FC is the same as that for BT.

The FC algorithm can be seen as making the subproblem of future variables node consistent with respect to the current partial solution, by removing inconsistent values from domains. If further propagation of the effects of removing these inconsistent values is performed around the subproblem, some form of arc consistency can be achieved. Partial Look Ahead (Haralick and Elliott 1980) extends FC by making additionally the variables in the future of each uninstantiated variable consistent with its remaining domain, according to the current instantiation order.

This algorithm saves around half of the consistency checking cost of Full Look Ahead (Haralick and Elliott 1980), which establishes full arc consistency in the subproblem of future variables during search. Full Look Ahead was originally reported by (Gaschnig 1979)[1] and more recently by (Sabin and Freuder 1994), who term the algorithm MAC, for Maintaining Arc Consistency, which is the name that we use here. A MAC algorithm uses an arc consistency technique (discussed later in Section 2.5) to propagate the effects of instantiating the current variable around the

[1]Gaschnig originally termed this algorithm DEEB (domain element elimination with backtracking).

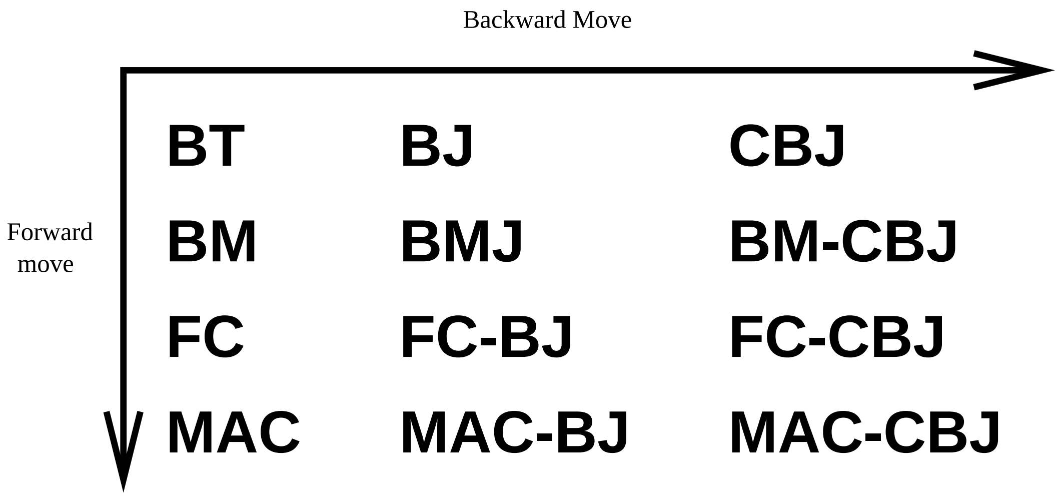


Figure 2.1: Hybrid combinations of CSP search algorithms.

future subproblem. If a future domain is annihilated, MAC tries the next value or chronologically backtracks if no more values remain.

(Haralick and Elliott 1980) investigated these four styles of forward search move for CSP algorithms, testing each one empirically on a set of problems including the *n*-queens. They concluded that FC provides the most efficient search, striking a balance between the effectiveness and the cost of its lookahead technique. We challenge this conclusion in Chapter 7, where the performance of FC is compared to that of MAC over a wider variety of test problems.

2.1.3 Hybrid combinations

(Prosser 1993) shows how the forward search move of a lookahead algorithm may be combined with the backward search move of a backjumper to produce the hybrid combinations BMJ, BM-CBJ, FC-BJ and FC-CBJ. Later, in (Prosser 1995), he also shows how MAC and CBJ may combine to form MAC-CBJ (and by assumption how MAC and BJ may form MAC-BJ).

Figure 2.1 is similar to Figure 2 in (Prosser 1993). It shows how the basic BT algorithm may be refined through more sophisticated forward moves (vertical axis) or backward moves (horizontal axis). Hybrid combinations lie at the intersections of the basic forward and backward styles. The figure has been extended to include hybrid combinations of MAC.

2.1.4 Theoretical evaluation

A theoretical evaluation of many backtracking search algorithms is presented by (Kondrak and van Beek 1997). They prove the correctness of BT, BJ, CBJ, BM, FC and all hybrid combinations of these algorithms. Hierarchies of algorithm performance are also produced, in terms of both the search tree and the consistency checking cost. These hierarchies apply to dynamic variable ordering strategies (see Section 2.4) as well as static instantiation orderings.

It is shown that the search tree for $\{$CBJ,BM-CBJ$\}$ is always a subset of the search tree for

```
function bcssp(n, status)
begin
  consistent := TRUE;
  status := UNKNOWN;
  i := 1;
  while status == UNKNOWN do
    begin
      if consistent then i := label(i,consistent)
      else i := unlabel(i,consistent);
      if i > n then status := SOLUTION
      else if i == 0 then status := IMPOSSIBLE;
    end
end
```

Figure 2.2: Driver function for backtracking search.

{BJ,BMJ}, which in turn is a subset of that for {BT,BM}. The search tree for FC-CBJ is a subset of that for FC, which is a subset of that for {BJ,BMJ}.

In terms of consistency checks, BM costs no more than BT, while BMJ costs no more than BJ, which costs no more than BT. BM-CBJ costs no more than CBJ, which costs no more than BJ, and FC-CBJ costs no more than FC.

Kondrak and van Beek show that the search tree for an algorithm employing CBJ is always a subset of that for an algorithm employing the same forward move with chronological backtracking. We can therefore infer that the search tree for MAC-CBJ is a subset of that for MAC.

2.2 Implementing Backtracking Search

The nature of the backtracking search process suggests a recursive implementation, with instantiations being pushed onto or popped off from the solution ‘stack’. A disadvantage to this approach, however, is that all search ‘knowledge’ is hidden within the procedure stack and becomes inaccessible during search. This is undesirable if hybrid combinations of algorithms are to be constructed.

(Prosser 1993) describes an alternative iterative implementation of backtracking search. This encoding uses two functions, **label** and **unlabel**, which perform forward and backward search moves respectively. These functions are called from a driver function which controls the search process.

In order to solve the binary constraint satisfaction search problem (that is, to find the first solution to a CSP), Prosser defines a driver function `bcssp` to implement a backtracking search. Figure 2.2 describes this function in a pseudo-code form. The code used here is similar to, though not exactly the same as, that used by Prosser. It is based on Pascal and C, and uses `:=` as the assignment operator and == as the equality operator.

Function `bcssp` takes in a CSP containing `n` variables, returning a status flag denoting the result of the search. The function assumes a static instantiation order, with the identifier `i` denoting the current search variable. A successful call to the function `label` returns a positive `consistent` flag and sets `i` to be the next variable in the instantiation order. An unsuccessful call to `label` leaves `i` unchanged and causes a call to `unlabel`. This function sets `i` to be the variable that the search steps (or jumps) back to. It returns a positive `consistent` flag if a new instantiation is found for this variable, otherwise it causes a further call to `unlabel` on the next iteration.

The central loop of `bcssp` calls forward and backward search moves until all `n` problem variables have been instantiated (and a solution has been found), or the search backtracks past the first problem variable (and we have exhausted the search space).

2.3 Finding All Solutions

Descriptions of CSP search algorithms are traditionally given in terms of finding the first solution to problems, at which point the algorithm terminates. In some instances, however, it may be desirable to find all solutions to a CSP. One application of finding all solutions is in verifying the correctness of an algorithm implementation by comparing the number of solutions found with that of another implementation (and this has been carried out for all of the algorithm implementations reported here). Another application is in selecting problems with a desired number of solutions for specific purposes. For example, when studying incomplete search methods it is desirable to know beforehand whether a problem has any solutions, and if so how many there are.

To be capable of finding all solutions, a refinement must be made to a backtracking algorithm's move upon finding each solution. Kondrak discusses these refinements in his Masters thesis (Kondrak 1994), and studies which include finding many or all solutions to CSPs have been reported (Kwan *et al.* 1995; Smith and Dyer 1996). Although the refinements that must be made are not particularly difficult, precise implementation details are not readily available. For this reason, we present implementation details below, showing how Prosser's iterative description of backtracking algorithms may be extended to enable all solutions to be found.

2.3.1 Driving the search for all solutions

We can amend the `bcssp` function, shown in Figure 2.2, to make it find all solutions to the binary constraint satisfaction problem. The new driver function, `bcsp`, is shown in Figure 2.3. The new function maintains a facility for finding only the first solution, making the description a little more complex than is strictly necessary.

Lines 10 and 11 of `bcssp` have been replaced with lines 11 to 24 in `bcsp`. In the new function, when a solution is found and we wish to find more, a call is made to a special instance of the algorithm's backward move (line 16), which we have termed `backtrack`. This special move is described for the BT, BJ and CBJ styles of backward move below.

2.3.2 Backtracking and Backjumping

The backward move which must be made upon finding a solution in the case of a chronologically backtracking algorithm is extremely simple. Prosser's description of the chronological backtrack-

```
function bcsp(n, status, all_solutions?)
begin
  consistent := TRUE;
  status := UNKNOWN;
  num_solutions := 0;
  i := 1;
  while status == UNKNOWN do
    begin
      if consistent then i := label(i,consistent)
      else i := unlabel(i,consistent);
      if i > n then
        begin
          if all_solutions? == TRUE then
            begin
              num_solutions := num_solutions + 1;
              i := backtrack(i,consistent);
            end
          else status := SOLUTION;
        end
      else if i == 0 then
        begin
          if num_solutions == 0 then status := IMPOSSIBLE
          else status := SOLUTION;
        end
    end
end
```

Figure 2.3: Driver function for backtracking search finding all solutions.

```
1. function bt_unlabel(i, consistent)
2. begin
3.   h := i - 1;
4.   current_domain[i] := domain[i];
5.   remove_from_set(instantiation[h], current_domain[h]);
6.   consistent := not_empty(current_domain[h]);
7.   return(h);
8. end
```

Figure 2.4: Unlabelling function for the BT algorithm.

```
1. function bt_backtrack(i, consistent)
2. begin
3.   h := i - 1;
4.   remove_from_set(instantiation[h], current_domain[h]);
5.   consistent := not_empty(current_domain[h]);
6.   return(h);
7. end
```

Figure 2.5: Special backtrack function for the BT algorithm.

ing algorithm BT uses the function `bt_unlabel`, shown in Figure 2.4. Having unsuccessfully tried to instantiate variable `i`, BT backtracks to the previous variable, `h`, in the instantiation order, restores the domain of `i` and removes the previous instantiation of `h` from its domain.

In the case of having found a solution to a CSP with n variables, `i` has the value $n+1$ and we must backtrack to variable n. Having done this, we then remove the current instantiation of variable n, but do not restore the original domain of n. Thus the search continues by attempting to instantiate the next value in the current domain of n. Removing line 4 of `bt_unlabel` produces `bt_backtrack`, shown in Figure 2.5

In the case of the Backjumping algorithm, BJ, its ability to jump back over problem variables appears to give rise to the potential for pruning out solutions, should this happen immediately after a solution is found. However, as (Kondrak 1994) notes, BJ can only jump back over variables if an attempt to instantiate a variable fails, and in the case of having found a solution this is clearly not the case. Therefore the `bt_backtrack` function is also suitable for use by algorithms employing BJ.

2.3.3 Conflict-directed backjumping

The backward move which must be made upon finding a solution in the case of an algorithm employing CBJ is a little more complex. Kondrak discusses the problem of conflict set interpretation on page 25 of his Masters thesis (Kondrak 1994):

> The problem here is that the conflict sets of CBJ are meant to indicate which instantiations are responsible for some previously discovered inconsistency. However, after a solution is found, conflict sets cannot always be interpreted in this way. It is

```
function cbj_unlabel(i, consistent)
begin
  h := max_set(conflict_set[i]);
  union_set(conflict_set[i], conflict_set[h]);
  remove_from_set(h, conflict_set[h]);
  for j := (h+1) to i do
    begin
      clear_set(conflict_set[j]);
      current_domain[j] := domain[j];
    end
  remove_from_set(instantiation[h], current_domain[h]);
  consistent := not_empty(current_domain[h]);
  return(h);
end
```

Figure 2.6: Unlabelling function for the CBJ algorithm.

> the search for other solutions, rather than an inconsistency, that forces the algorithm to backtrack.
>
> We need to differentiate between these two causes of CBJ backtracks: (1) detecting an inconsistency, and (2) searching for other solutions. In the latter case the backtrack must always be chronological, that is, to the immediately preceding variable (otherwise we risk pruning out solutions).

(Prosser 1993) describes the backward move of CBJ in terms of the function `cbj_unlabel`, shown in Figure 2.6. This function jumps back from variable `i` to variable `h`, the deepest variable named in the conflict set of `i`. In the case of having found a solution, we cannot trust the conflict set, as the instantiations are consistent, and so we chronologically backtrack from $n+1$ to variable n and undo this instantiation. We now need to ensure that the algorithm chronologically backtracks on its *first* visit to each search level after each solution is found. After a level has been re-visited following each solution, it then becomes safe to jump back to that level on subsequent visits. For example, suppose that in a search involving 20 variables (which has search levels 0 to 20), a solution has been found and the algorithm has chronologically backtracked to level 14 before moving forward again to level 18. A ‘backjumping window’ between levels 20 and 14 now exists, and the algorithm may now safely backjump from level 18 to levels 14, 15, 16 or 17. If the conflict set at level 18 indicates a backjump to a level higher than 14, the algorithm must backjump only to level 14 and chronologically backtrack from there.

Kondrak suggests an implementation of this which makes use of an array *cbf* (chronological backtrack flag) to record the shallowest level visited since the last solution. However, a simpler way of achieving the same effect is simply to ‘fill’ the conflict set of variable n with variables 1 to $(n-1)$, forcing chronological backtracking to each level for the first time after each solution. Thus we have the function `cbj_backtrack`, shown in Figure 2.7, which is the same as `bt_backtrack` but with the addition of lines 5 and 6.

```
function cbj_backtrack(i, consistent)
begin
  h := i - 1;
  remove_from_set(instantiation[h], current_domain[h]);
  for g := 1 to (h-1) do
    add_to_set(g, conflict_set[h]);
  consistent := not_empty(current_domain[h]);
  return(h);
end
```

Figure 2.7: Special backtrack function for the CBJ algorithm.

Note that an equivalent action upon finding each solution would be to add to the conflict set of each variable k the preceding variable $(k-1)$ in the instantiation order (for $2 \leq k \leq n$ and again assuming a static ordering).

2.3.4 Remarks

The `backtrack` functions that are presented here for BT, BJ and CBJ do not need to be modified in any way for use with hybrid algorithm combinations (such as FC, FC-BJ and MAC-CBJ). Also, the descriptions presented here have assumed a static variable instantiation order purely for simplicity. Dynamic variable ordering can easily be introduced, and Section 2.4 shows how this is done.

A situation in which it may be undesirable to attempt to find all solutions to a problem clearly arises when there is likely to be a very large number of solutions. An example is when a problem is highly underconstrained and so almost every assignment is consistent. A theoretical expression to calculate the expected number of solutions to a CSP, plus data on the search effort involved in finding all solutions, has been presented in (Smith and Dyer 1996).

2.4 Search Heuristics

In its basic form, a backtracking algorithm is given a static variable instantiation ordering (SVO) which is unchanged during search. Significantly more efficient search can be achieved for certain algorithms, however, by allowing dynamic variable ordering (DVO). At each forward search move, a DVO uses a heuristic method to select the next variable which should be instantiated, given the current search conditions.

(Bitner and Reingold 1975) first proposed the use of ‘dynamic search rearrangement’ as a technique to improve backtracking search. (Purdom 1983) performed a theoretical analysis of dynamic variable ordering for backtracking search, finding all solutions to sets of random conjuctive normal form predicates. This type of problem has clearly defined parameter regions where the average number of solutions is either polynomial or exponential. Purdom showed that DVO reduces from exponential to polynomial the average time cost of algorithms on the problems containing polynomial numbers of solutions. He also showed that the overhead associated with DVO does not make it worthwhile on very easy instances, and conjectures that on the hard instances with exponential numbers of solutions, DVO achieves an exponential reduction in average cost from that of static ordering, even though this reduced cost remains exponential.

More recent empirical studies of DVO include (Dechter and Meiri 1994). They showed empirically that for backtracking search on binary CSPs, dynamic ordering was superior to every one of a number of informed static orderings that were tested. Since a dynamic choice of the next variable to instantiate is based on the changing state of the future subproblem, it is obvious that DVO is useful only with algorithms that perform a lookahead style of forward move, such as FC and MAC. (Bacchus and van Run 1995) state this explicitly.

(Haralick and Elliott 1980) show that a simple and effective DVO heuristic is to instantiate the variable that has the fewest remaining values left in its domain. The reasoning behind this 'smallest domain first' strategy is that the most heavily constrained variable at each stage of the search should be tackled as a priority: if it leads to a failure and subsequent backtracking, it is better that this is discovered sooner rather than later. This notion of selecting the variable to instantiate which is most likely to lead to failure has become known as the 'fail-first' principle.

Other heuristics that may be used to guide backtracking search include value ordering heuristics, which order the domain elements of variables a way designed to reduce search cost (Dechter and Pearl 1988). The scope of this thesis, however, is restricted to dynamic variable ordering heuristics. An implementation of DVO consistent with the descriptions of backtracking search algorithms presented by (Prosser 1993) is described below.

2.4.1 Implementing dynamic variable ordering

A DVO implementation should allow for 'fair' comparison of algorithms employing the same forward search move and DVO, but a different backward move. For example, consider the case of FC and FC-CBJ both employing some form of DVO. FC-CBJ will always perform no more consistency checks than FC (Kondrak and van Beek 1997), *as long as* the algorithms always have the same choices at each forward search move. When the order in which future variables are examined is not the same, any tie-breaking condition means that the above requirement is not satisfied, and the algorithms may follow entirely different search paths and find different first solutions. Two or more algorithms employing the same style of forward search move and DVO should therefore always examine future variables in the same order when selecting the next variable to instantiate.

2.4.2 DVO with a backtracker

At search depth *i*, the next variable to be instantiated is selected and swaps positions in the ordering with the variable currently at position *i*. To present all algorithms employing the same forward move with an identically ordered set of future variables at each search depth, we must explicitly record the swap (if any) that was most recently performed at each search level, and undo this swap every time search backtracks past a particular level.

Function `bcssp_dvo` is shown in Figure 2.8, and shows how function `bcssp` (Figure 2.2) is modified to enable dynamic variable ordering. Note that the variable `i` of `bcssp` has been re-

```
 1.  function bcssp_dvo(n, status)
 2.  begin
 3.    consistent := TRUE;
 4.    status := UNKNOWN;
 5.    ii := 1;
 6.    while status == UNKNOWN do
 7.      begin
 8.        if consistent then
 9.          begin
10.            dvo_select_next_variable(ii);
11.            ii := label(ii,consistent);
12.          end
13.        else
14.          begin
15.            ii := unlabel(ii,consistent);
16.            dvo_undo_last_swap((ii+1));
17.          end
18.        if ii > n then status := SOLUTION
19.        else if ii == 0 then status := IMPOSSIBLE;
20.      end
21.  end
```

Figure 2.8: Driver function for a backtracking search with dynamic variable ordering.

named `ii` in order to emphasize that its value represents the current search depth only. Lines 10 and 16 have been added to the basic `bcssp` function to produce `bcssp_dvo`. It is assumed that an initial instantiation order has already been created.

Line 10 of `bcssp_dvo` selects the next variable to be instantiated, according to the rules, by calling a function `dvo_select_next_variable` to make the instantiation selection and record any swap that must be made. This function is described in Figure 2.9. The call in Line 3 returns the position `jj` in the instantiation order of the future variable that has been selected. If this variable is not already the next in the instantiation order (Line 4) then its position is swapped with that of the variable that is currently next (Line 6). If such a swap has to be made at the current search level, `ii`, then the original position, `jj` of the selected variable is recorded in element `ii` of a

```
 1.  function dvo_select_next_variable(ii)
 2.  begin
 3.    jj := select_according_to_dvo_criteria();
 4.    if jj != ii then
 5.      begin
 6.        swap_instantiation_places(ii, jj);
 7.        swaps[ii] := jj;
 8.      end
 9.  end
```

Figure 2.9: Dynamically selecting the next variable to instantiate.

```
1.  function dvo_undo_last_swap(ii)
2.  begin
3.    if ii == 1 then return
4.    else if swaps[ii] != 0 then
5.       begin
6.         swap_instantiation_places(ii, swaps[ii]);
7.         swaps[ii] := 0;
8.       end
9.  end
```

Figure 2.10: Undoing the effects of a dynamic variable swap.

one-dimensional array `swaps`(Line 7).

Line 16 of `bcssp_dvo` undoes any swap that has been made at a search level, in the event that the search backtracks past this depth, by calling a function `dvo_undo_last_swap`. The call to `unlabel` of Line 15 has reset the search depth `ii` and so the level `(ii+1)`must be given to the call to `dvo_undo_last_swap`. A pseudo-code description of this function is shown in Figure 2.10. Once again, if search is at depth 1 then this operation does not apply (Line 3), otherwise if a swap has been made at the given search level (Line 4) then the swap is undone (Line 5) and the entry in the array *swaps* is cleared.

2.4.3 DVO with a backjumper

In the case of dynamic variable ordering being used by an algorithm with backjumping capability, the same process is carried out. However, as the search may now backjump up past several search levels, it must undo any swaps that have been made for *each intermediate search level, in order*. To realise this, a minor refinement must be made to function `bcssp_dvo`, represented by the following pseudo-code fragment:

```
14.          begin
14b.           mark := ii;
15.            ii := unlabel(ii,consistent);
15b.           for level := mark downto (ii+1) do
16.              ff_undo_last_swap(level);
17.          end
```

Line 14b marks the search depth we jump back from, and following the call to `unlabel` in Line 15, `ii` now represents the search level we have jumped back to. Lines 15b and 16 undo the swaps made at levels `mark` down to `(ii+1)` in that order.

2.5 Consistency Algorithms

Many algorithms have been described which use constraint propagation (Section 1.2.4) to establish levels of consistency in CSPs. Although algorithms exist which can establish k-consistency for any value of k (Cooper 1989), the high cost associated with even low-level consistency means that for practical purposes only arc consistency and occasionally path consistency are ever attempted. In some circumstances, however, it might be feasible to propagate the effects of particular constraints in the problem. Consistency is often introduced into a CSP as a preprocessing step to search, or as a lookahead component of the actual search process.

A wide range of arc consistency algorithms exist, the most popular and useful of which are, arguably, AC3, AC4 and AC6. AC3 is a relatively simple method, introduced by (Mackworth 1977). It places all binary constraints on a queue and propagates the effects of each one in turn, adding all constraints affected by each round of propagation to the back of the queue. The algorithm terminates when the queue is empty. The general AC3 algorithm is shown by (Mackworth and Freuder 1985) to have a worst-case time complexity that is bounded from above by $O\left(m^3e\right)$ and from below by $\Omega\left(m^2e\right)$, where m is the size of the largest variable domain and e is the number of constraints in the CSP. The algorithm's space complexity is $O(e+nm)$, where n is the number of variables.

AC4 is presented by (Mohr and Henderson 1986), and has a worst-case time complexity of $O\left(m^2e\right)$, which is optimal. This algorithm runs in two stages. For each value of each variable, a record is made of all the values in the problem which support its inclusion in the arc consistent domain. Arc consistency is then enforced by removing unsupported domain elements, and propagating the effects of their removal by adjusting the relevant support records for other variable domain members. The additional space overhead associated with the support counters make the space complexity of AC4 $O\left(m^2e\right)$.

(Bessière and Régin 1995) address the expensive space requirements of AC4, introducing AC6. This algorithm eliminates some redundant support checking activities carried out by AC4, reducing the space complexity to $O(me)$.

Although the worst-case time complexity of AC4 is optimal, empirical study shows that its *average* performance is poor, often close to the worst case, and is in fact more expensive than AC3 on many types of CSP (Wallace 1993). Bessière, meanwhile, demonstrates that AC6 can outperform the other two algorithms on many problems. He does note, however, that AC3 is "never really bad", and that AC6 is not particularly suited to incorporation within a search process.

Algorithms for path consistency tend to be derived from those for arc consistency. The most common examples are PC2 (Mackworth 1977), derived from AC3, and PC4 (Han and Lee 1988), derived from AC4. Path consistency algorithms are expensive, both in terms of space and time: PC2 has worst case time complexity $O\left(m^5n^3\right)$ and space complexity $O\left(n^3+n^2m^2\right)$; PC4 has worst case time complexity $O\left(m^3n^3\right)$ and space complexity $O\left(m^3n^3\right)$.

The algorithm to establish k-consistency which (Cooper 1989) proposes, KS-1, is based on PC4 and has worst case time complexity and space complexity that are both $O\left(\sum_{i=1}^{k}\left({}_nC_i.m^i\right)\right)$.

Chapter 3

Random Problem Generation

Randomly generated binary CSPs form the basis of the empirical studies reported throughout this thesis. Thus, we require a random generation model which allows the creation of large *ensembles* of CSPs with similar properties, enabling conclusions to be made about the general behaviour of these problems.

The issue of random problem generation in empirical AI is complex, and recent literature points to examples of how poor random generation models can lead to flawed results and conclusions in SAT (Mitchell and Levesque 1996) and CSP (Gent *et al.* 1997a). Therefore, care must be taken in generating suitable sets of random problems. This chapter describes in detail the creation and implementation of the random generation model used, discussing the options available and justifying the choices made.

Before devising and implementing a random CSP generator, we need to be satisfied that the use of random problems to test algorithms is justified. Having done this, the set of parameters used to produce the CSPs must be defined. From these parameters, a method of generating populations of CSPs which have consistent and known properties, but without bias towards particular features, can then be devised. Potential sources of variation within populations of problems should be kept to a minimum, and those that remain must be understood and controlled. Finally, a good random generation model also requires a good random number generator.

These issues are discussed over the following sections, after which the implementation of the resulting random problem generator is described. A number of potential extensions to the random generation model are then discussed.

3.1 Justification for Random Problems

The use of randomly generated problems in the empirical study of algorithms has been criticised in some quarters. Methodological discussions such as (Mitchell and Levesque 1996) and (Johnson 1996) warn that these problems may lack the features and characteristics of the ‘real world’ problems for which an algorithm is ultimately intended. Conclusions based on random problems may be flawed unless sufficient structure is imposed upon the random generation model to give the resulting problems an element of realism.

If random problems are to be avoided, the alternative is to test algorithms on ‘real’ benchmark problems which can be found in various repositories. However, these problems may be similarly criticised for being unrepresentative (Hooker 1994). A further important caveat is that it is currently impossible to obtain populations of ‘real world’ problems in sufficient quantities to enable meaningful statistical analyses of their general behaviour. The current state of the art means that large populations of sample problems invariably entail some form of random generation.

The whole dilemma of finding representative problems to test algorithms on may not in fact be as significant as it appears. (Hooker 1994) argues that while any choice of problem may be criticised for being unrepresentative, the experiments can be designed around this. In particular:

> One can investigate *how algorithm performance depends on problem characteristics*. The issue of problem choice, therefore, becomes one of experimental design. Rather than agonise over whether a problem set is representative of practice, one picks problems that vary along one or more parameters.

We choose to follow this suggestion, developing a random generation model for creating ensembles of CSPs based on a set of variable parameters. The random generation model provides an effectively limitless supply of CSPs whose properties can be adjusted to simulate ‘real’ situations, and whose sizes may be increased to arbitrary levels. The parameters to be used are introduced in the following section.

3.2 CSP Parameters and Properties

A binary CSP, as described in Chapter 1, consists of a set of variables, each of which has a domain of possible values, and a set of constraints defining allowed combinations of values between certain pairs of variables. The binary CSPs that form the basis of our empirical studies are characterised by four parameters:

n the number of problem variables.

m the number of values in each variable’s domain.

$\mathbf{p_1}$ the probability that there is a constraint between a pair of variables (the *constraint density*).

$\mathbf{p_2}$ the conditional probability that a pair of values is inconsistent for a pair of variables, given that there is a constraint between them (the *constraint tightness*).

Using these parameters, we can generate ensembles of CSPs with similar characteristics, denoted by the 4-tuple $\langle n, m, p_1, p_2 \rangle$. Randomly generated CSPs of this type have been studied extensively, for example by (Freuder and Wallace 1992), (Dechter and Meiri 1994), (Williams and Hogg 1994), (Frost and Dechter 1995), (Tsang *et al.* 1995), (Smith and Dyer 1996) and (Prosser 1996). Some of these studies use alternative nomenclatures to $\langle n, m, p_1, p_2 \rangle$, between which (Prosser 1996) provides a series of translations.

3.2.1 Properties of $\langle n,m,p_1,p_2\rangle$ CSPs

It is known that a phase transition (Section 1.4) occurs for these CSPs as p_2 is varied while n, m and p_1 are fixed (Smith and Dyer 1996). A significant amount of theoretical study has been devoted to making predictions about the location and properties of the phase transition for these problems. (Smith and Dyer 1996) and (Williams and Hogg 1994) have developed an expression for the expected number of solutions, $E(N)$, for an $\langle n,m,p_1,p_2\rangle$ CSP. This is calculated as:

$$E(N) = m^n(1-p_2)^{p_1.n(n-1)/2}$$

which is the number of possible assignments of m values to n variables, multiplied by the probability that a randomly-chosen assignment is consistent.

(Smith and Dyer 1996) suggest that a CSP for which $E(N) = 1$ can be expected to lie on or near the crossover point at the phase transition. Making this assumption, an estimate for the critical value of constraint tightness, $\hat{p}_{2crit}$, at which the crossover point for an $\langle n,m,p_1\rangle$ class of CSP lies can be calculated as:

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

Empirical studies by (Prosser 1996) show $\hat{p}_{2crit}$ to be an accurate prediction of the location of the crossover point, except for sparsely constrained problems (with low p_1). For the sparse problems, the prediction tends to be an overestimation, suggesting that $E(N)$ for these CSPs is greater than 1 at the crossover point; reasons for this are discussed in depth by (Smith and Dyer 1996).

More general work on phase transition behaviour in combinatorial problems has lead to (Gent *et al.* 1996b) formalising a notion of problem *constrainedness*, encapsulated by the *general constrainedness parameter*, κ. This parameter generalizes the specific parameters defining constraints in several classes of NP-complete problem, such as CSP, SAT and graph colouring. The constrainedness of an $\langle n,m,p_1,p_2\rangle$ CSP containing e constraints, and for which a solution can be represented in $\mathcal{N}$ bits, is calculated as:

$$\begin{aligned}\kappa &= 1 - \frac{\log_2(E(N))}{\mathcal{N}} \\ &= \frac{-e\log_2(1-p_2)}{n\log_2(m)}\end{aligned}$$

The predicted crossover point occurs at $\kappa = 1$, which is equivalent to the prediction of its occurrence at $E(N) = 1$. Thus, in general, under-constrained problems have $\kappa < 1$ and over-constrained problems have $\kappa > 1$. Since κ is based on $E(N)$, it tends to be underestimated for sparse CSPs.

3.3 Models for Problem Generation

Having defined four parameters for the experimental CSPs, we must choose the way in which they are applied to produce ensembles of randomly generated problems. The constraint graph

(Section 1.2) of a CSP can be represented by a symmetrical $n \times n$ matrix of boolean values, where a binary constraint between variables v_i and v_j is indicated by a 'true' value at positions (i, j) and (j, i). Each binary constraint relation between two variables with domain size m can be represented with an $m \times m$ boolean relation matrix. Here, each 'true' value indicates that the corresponding pair of values are disallowed by the constraint.

There are a number of ways in which we can use values p_1 and p_2 to generate constraint graphs and constraint relation matrices respectively. (Smith and Dyer 1996), for instance, refer to two models for matrix generation: 'Model A' and 'Model B', based on work by (Palmer 1985):

Model A treats p_1 and p_2 as probabilities, selecting each of the $n(n-1)/2$ possible edges in the constraint graph independently with probability p_1. Relation matrices for each constraint are then generated, assigning 'true' values to each of the m^2 value pairs independently with probability p_2. Problems generated using Model A should contain, on average, $p_1.n(n-1)/2$ constraints, each containing on average $p_2.m^2$ disallowed value pairs.

Model B treats p_1 and p_2 as fixed proportions, which specify *precisely* how many constraints and inconsistent value pairs each problem should contain. To construct the constraint graph for a problem we randomly select $p_1.n(n-1)/2$ pairs of variables as constraints, and for each of these we randomly select $p_2.m^2$ pairs of values as being disallowed. Where necessary, values are rounded to the nearest integer to give the actual number of constraints and conflicts required.

Both models of constraint and conflict generation have been used in recent empirical studies; for example, (Tsang *et al.* 1995) use Model A generation, while (Smith and Dyer 1996) use Model B generation. A combination of models is also possible; for instance Model A constraint graph generation with Model B conflict generation. (Smith and Dyer 1996) note that while the expected number of solutions, $E(N)$, for Model A and Model B CSPs is the same, the variance in this quantity is not. They also note that Model B problem generation allows p_2 to be varied in steps no finer than $1/m^2$. The use of Model A generation simplifies theoretical analysis, due to the use of probabilities. This use of probability, however, does imply variations in the properties of CSPs in an ensemble.

For most the experiments with randomly generated CSPs presented in this thesis, the Model B method will be used for generating both constraint graphs and relation matrices, unless specifically stated otherwise. Model B problem generation allows the empirical studies to use sets of problems containing identical numbers of constraints and conflicts, whereas Model A generation produces ensembles of problems containing an unknown amount of variation in these quantities.

The loss of control over basic properties of the experimental CSPs that would occur using Model A generation may obscure attempts to study the behaviour of problems with particular properties, threatening the validity of any conclusions drawn. However, use of Model A is appropriate when testing algorithms or heuristics whose reasoning is based on probabilistic analysis of CSPs. Several techniques of this kind are examined in Chapter 8, which uses a hybrid generation model that employs Model B to generate constraint graphs and Model A to generate relation matrices.

3.4 Constraint Graph Connectivity

CSPs produced using the Model A or Model B random generation methods cannot be guaranteed to contain constraint graphs that are connected, if p_1 is sufficiently small. If constraint density, p_1, is very low then a significant proportion of the random constraint graphs may in fact consist of two or more independent sub-graphs, forming the basis of two or more independent sub-problems.

The appearance of disconnected CSPs in the experimental ensembles is undesirable, since these problems would in practice be dealt with separately, using a ‘divide and conquer’ strategy (Tsang 1993). We feel that any conclusions based on disconnected constraint graphs would be flawed, and so choose to exclude them from our study. This can be achieved either by a method of forcing the generation of connected constraint graphs, or by simply discarding instances of disconnected graphs. It should be noted that the requirement for connected constraint graphs imposes a lower limit on values of p_1, since connectivity requires at least $n-1$ constraints.

An example of a technique to guarantee the generation of connected constraint graphs is provided by (Sabin and Freuder 1994), who randomly generate a tree of constraints, after which the remaining constraints are added. However, we choose the simpler option for excluding disconnected CSPs. Each constraint graph produced is tested for connectivity, with disconnected graphs discarded and replacements generated until a connected one is found.

Excluding disconnected CSPs makes our populations of sparse constraint graphs unrepresentative of all random graphs. However, the aim is to study the properties of random CSPs not random graphs, and the method of ensuring connectivity allows us to retain control of a vital property of the CSPs whilst keeping the random generation model simple. Theoretical work on the connectivity of random graphs can be found in (Bollobas 1985).

3.5 Random Number Generation

The key features of a random generation model for our experimental CSPs have been established, designed to enable the generation of problems with consistent properties but without bias towards particular features. To successfully implement this model, access to a reliable source of random numbers is essential. The issue of reliable random number generation is not simple, and the dangers of poor use of random numbers in problem generation have been documented (Gent *et al.* 1997a). Therefore, the requirements for the random number generator are discussed below, after which an appropriate source method of generation is selected.

3.5.1 Required properties

The chosen model for random matrix generation (Section 3.3) requires long streams of random numbers. Streams of random numbers produced using a deterministic method are of course only *pseudo*-random, and as such will generate pseudo-random CSPs.

(Knuth 1981) discusses the requirements of a 'good' pseudo-random number generator in depth. Essentially, a good generator must reasonably represent a known probability function, avoiding any built in trends, biases or periodicities. It is usually expected that a value generated should not be correlated in any way with previous values in the stream.

3.5.2 The Linear Congruential Method

A simple and popular method for generating streams of pseudo-random numbers is the Linear Congruential Method (Knuth 1981), which uses a recurrence relation of the form:

$$I_k = (aI_{k-1} + c) \bmod m$$

The values a, c and m are constants, known as the multiplier, increment and modulus respectively. (Knuth 1981) notes that the choice of values for these constants is crucial in determining the quality of the generator.

We choose to use a Linear Congruential generator to produce our experimental CSPs using the constant values $a = 16807$, $c = 0$ and $m = 2,147,483,647$. This generator is proposed by (Park and Miller 1988) as being the best choice for the production of 32-bit pseudo random number streams, and is known as the *minimal standard* random number generator. The period of this generator (the length of the number stream produced before repetition) is $2^{31} - 1$, the maximum possible, and this is appropriate for the problem generation application.

The minimal standard generator is commonly used by compilers and software applications. However, to avoid any issues of platform dependency for our CSP problem generator, the random number generator will be implemented internally.

3.6 Random Problem Generator Implementation

The framework required for the implementation of a suitable pseudo-random problem generator is now complete. The experimental CSPs will be generated from the parameters $\langle n, m, p_1, p_2 \rangle$ using either the Model A or B generation method, with disconnected problems excluded. Random number streams will be provided from the minimal standard random number generator.

Each CSP at a particular $\langle n, m, p_1, p_2 \rangle$ can be generated from a single integer 'seed' value, from which the required stream of random numbers can be produced. This allows the easy identification and reproduction of any individual CSP or ensemble.

Our implementation of a random CSP generator is publicly available on the World Wide Web, as part of the software system used to conduct the empirical studies. Further details on this can be found in Appendix A.

3.7 Extensions to the Random Generation Model

Section 3.1 discusses some of the criticisms that have been levelled at the use of randomly generated problems in the empirical study of algorithms. Reservations about random generation tend to centre around the risk that features of the resultant problems can be unrepresentative of ‘real world’ situations. Methodological discussions such as (Mitchell and Levesque 1996) and (Johnson 1996) do suggest, however, that random problems might be given some extra structure to make them more representative. (Johnson 1996) also calls for problem generators which allow extra parameters to be set which allow the investigation of certain properties.

There are several extensions which can be made to CSPs generated using the $\langle n, m, p_1, p_2 \rangle$ model which might add more realism to these problems. (Gent *et al.* 1996a), for example, propose two extensions to the $\langle n, m, p_1, p_2 \rangle$ model designed to add elements of (controlled) non-uniformity to the domain sizes and the tightness of constraints:

Varied domain sizes may be achieved by selecting one of a number of cardinalities for each variable domain with certain probabilities. Gent *et al.* use this technique to experiment with problems containing domain sizes of $m = \{10, 20\}$, where the probability of each value is 0.5.

Varied constraint tightness may be achieved in a similar way, with each constraint within a problem generated with one of a number of tightness values according to probability. Gent *et al.* use this technique to generate problems where $p_2 = \{0.2, 0.8\}$ with probabilities 0.8 and 0.2 respectively.

These modifications are compatible with the current random generation model, and it may be appropriate to make use of them during the experiments. However, the introduction of an additional two degrees of freedom to the experimental CSPs would increase the complexity of the empirical studies to the extent that this is not feasible. Therefore, we restrict the main experiments to CSPs using $\langle n, m, p_1, p_2 \rangle$ according to the basic random generation model.

Chapter 4

The Empirical Study of Algorithms

In calling for an *empirical science of algorithms*, (Hooker 1994) laments the infrequent use of experimental design principles, and the failure of most published empirical studies to observe "even minimal standards of reproducibility". The methodological discussion presented by (Johnson 1996) warns of a number of common mistakes which can compromise the integrity of empirical studies. A number of these opportunities for flawed studies revolve around the issue of irreproducibility, particularly the failure to disclose key implementation and environmental details.

In an attempt to address the criticisms levelled at empirical studies of algorithms, we begin here by laying out a clear framework for the experiments that will be conducted. A consistent nomenclature for the description of the problems and algorithms to be used is presented, and the environment under which the experiments are performed is recorded. A generic format for the main experiments, based on the phase transition model, is then presented, and the issues of collecting meaningful search data and presenting it lucidly are addressed.

4.1 Experimental Nomenclature

To promote clarity and brevity when describing the empirical studies reported throughout this thesis, a consistent nomenclature for describing aspects of the experiments has been devised. This nomenclature covers the description of groups of CSPs and the combinations of algorithms and heuristics used to process them.

4.1.1 CSP nomenclature

The term CSP refers to a pseudo-random binary constraint satisfaction problem, generated according to the model presented in Chapter 3. Groups of CSPs are described using the following terms:

- An *ensemble* of CSPs is generated for a particular value of the 4-tuple $\langle n, m, p_1, p_2 \rangle$.
- Phase transitions from under- to over-constrained problems are observed as p_2 varies, while n, m and p_1 are kept fixed (Section 3.2); *classes* of CSPs with fixed n, m and p_1 and varying

p_2 are referred to by the tuple $\langle n, m, p_1 \rangle$.

- A measure of the 'local' density of constraints around each CSP variable is the *average degree*, γ. This is the average number of constraints incident upon each variable, calculated as $(2 \times \mathit{num\_constraints})/n$. Thus, a CSP class might also be referred to by the tuple $\langle n, m, \gamma = x \rangle$.

- Discussion of CSP *size* refers to the tuple $\langle n, m \rangle$.

4.1.2 Algorithm nomenclature

An overview of complete CSP search algorithms is provided in Section 2.1, and many of these are used in the empirical studies. Table 4.1 names all of the basic algorithms to be used, notes the source of the original descriptions, and records the published implementation details that are used.

Algorithm	Source	Implementation	Description
BT	(Golomb and Baumert 1965)	(Prosser 1993)	Chronological backtracking
CBJ	(Prosser 1993)	(Prosser 1993)	Conflict-directed backjumping
FC	(Haralick and Elliott 1980)	(Prosser 1993)	Forward checking
FC-CBJ	(Prosser 1993)	(Prosser 1993)	Hybrid combination
MAC	(Gaschnig 1977)	(Prosser 1995)	Maintaining arc consistency
MAC-CBJ	(Prosser 1995)	(Prosser 1995)	Hybrid combination

Table 4.1: Sources, implementations and brief descriptions for the CSP algorithms used.

Each reference to a named algorithm, such as MAC-CBJ corresponds to the implementation recorded in Table 4.1. In addition to the basic CSP algorithms, features such as dynamic variable ordering heuristics (Section 2.4) are often used. The use of a heuristic HEU with MAC-CBJ would be denoted as MAC-CBJ-HEU.

Some of the experimental studies may use other extensions to basic CSP algorithms, such as preprocessing or a fixed initial variable ordering. Naming strategies for these combinations will be introduced pragmatically in the relevant chapters.

4.2 Experimentation Environment

The software system which conducts the empirical studies is implemented using the C language. This system is comprised of three main modules: a problem generator, capable of producing ensembles of random CSPs; a suite of search algorithms, problem preprocessors and heuristics which can be combined to process these problems; and a data collection module which produces statistics about individual searches and groups of searches.

Development of this system is carried out under a UNIX environment on a Sun SparcStation platform. The system is then transferred to a UNIX environment on a Silicon Graphics Indigo platform, enabling the experiments to be distributed over a network of 75 workstations. Access to this considerable computing resource in practice provides approximately one CPU year of available processing power per week.

As has been mentioned in Section 3.6, software capable of reproducing the experiments reported throughout this thesis is publicly available on the World Wide Web. Further details on this can be found in Appendix A.

4.3 Phase Transition Experiments

The main purpose of the empirical studies is to investigate the performance of different algorithms on CSPs of varying size and constraint topology. This usually requires the application of the algorithms to several classes of CSP over a range of problem sizes.

The application of an algorithm to an $\langle n, m, p_1 \rangle$ problem class forms the basic building block of most of the empirical studies, and we use the term *phase transition experiment* to describe such an operation. A phase transition experiment is formed by varying constraint tightness, p_2, in steps of $1/m^2$ over the range $[1/m^2..1]$. At each of these points, an ensemble of CSPs is generated for the current $\langle n, m, p_1, p_2 \rangle$, and each member problem is searched by the algorithms being studied.

By covering each possible value of p_2, a phase transition experiment will apply algorithms to problems in the under-constrained 'easy-soluble' problem region, through the 'hard' phase transition region, and into the over-constrained 'easy-insoluble' problem region. In order to accurately gauge the average and extremes of algorithm behaviour, each $\langle n, m, p_1, p_2 \rangle$ ensemble must be sufficiently large. A typical ensemble consists of $1,000$ CSPs, while sample sizes of $10,000$ problems are often required if rare features of algorithm behaviour are being investigated.

To illustrate the typical format of an empirical study into algorithm behaviour, consider the example of testing the FC algorithm (Section 2.1) on problems of size $\langle 30, 10 \rangle$. To investigate the performance of the algorithm as constraint density varies, p_1 is varied in steps of 0.1 over the range $[0.1..1.0]$. A phase transition experiment is then performed for each $\langle 30, 10, p_1 \rangle$ class: p_2 is varied in steps of 0.01 over the range $[0.01..1.00]$; ensembles of $1,000$ CSPs are generated at each of these points, and the problems searched by FC.

The above example entails the generation and search of one million CSPs, which may require several CPU days of processing power. Such an experiment would form only a small part of a comparative study of many algorithms on many types of CSP, possibly involving a total investment of several CPU years.

4.4 Measuring Search

The cost of a search performed by an algorithm on a CSP may be measured and expressed in a number of ways. These 'instruments' for measuring search cost may vary in their level of dependency upon implementation and environment, and may also vary in their perceived relevance to the 'true' cost of a search. However, many if not all of these measures may make a valid contribution to the understanding of a search process, and methodological discussions such as (Gent *et al.* 1997b) encourage empirical studies to "measure with many instruments".

In order to provide as many views of the search process as possible, seven different aspects of each search are measured in the empirical studies presented here. These measures are described below.

4.4.1 Consistency checks

When checking that a value x for a variable v_i is consistent with a value y for a variable v_j, a single consistency check is counted. If the variables involved are not constrained, then no check is counted. The consistency checking cost of a search is regarded by many as a good surrogate for 'real' time.

Consistency checks are environment independent, but are highly dependent on implementation efficiency. Also, for some algorithm implementations, it may be neither possible nor sensible to count consistency checks due to the way in which the constraints are implemented: for instance the AC4 arc consistency algorithm (Mohr and Henderson 1986), which performs all of its consistency checking during an initialisation stage.

4.4.2 Nodes visited

Every trial instantiation of a variable made during search corresponds to a single node in the search tree having been visited. This definition corresponds to that given by (Kondrak and van Beek 1997).

This measure is both environment and implementation independent, given the same variable instantiation ordering, and so allows direct comparison between any implementation of a particular algorithm. However, it may not reflect the varying levels of search effort required during each trial instantiation, and so is not as analogous to real time as consistency checks.

4.4.3 CPU time

The amount of CPU time elapsed during each search is measured approximately. CPU time is regarded by many as the true 'bottom line' of search cost, and often as the *only* relevant measure. However, execution time is highly environment and implementation dependent, and is a notoriously difficult measure to take accurately.

Circumstances may arise, though, when CPU time is the only suitable measure of cost. These include trying to estimate overheads that are not reflected in the number of consistency checks or nodes visited: for example the conflict set maintenance cost associated with the CBJ algorithm, described in Section 2.1.

4.4.4 Permanent nogood values

Some CSP algorithms are capable of removing values from the domains of variables that are proven not to form part of any solution. These algorithms include preprocessors such as AC3 (Section 2.5).

For searches using these types of algorithm, the number of permanently removed 'nogoods' is recorded. This measure is both environment and implementation independent.

4.4.5 Temporary nogood values

The lookahead CSP search algorithms presented in Section 2.1, such as FC and MAC remove values from the domains of uninstantiated variables during forward search moves, which may later be reinstated upon backward search moves. These 'temporary nogoods' are counted during searches with these algorithms.

While this measure is environment independent, the number of temporary nogoods during a search can be affected by sensitive implementation features, such as the order in which the effects of instantiations are propagated. Therefore, it could be said to be partially implementation dependent.

4.4.6 Labellings

The number of forward search moves during which a variable is successfully instantiated, or labelled, are counted. This measure differs from nodes visited in that the labelling of a variable may require a number of trial instantiations before a valid one is found, corresponding to several nodes.

4.4.7 Unlabellings

The number of backward search moves, or unlabellings, made upon encountering dead ends are recorded. This measure is environment and implementation independent, given the same instantiation ordering.

The number of unlabellings during search might give a good indication of the effectiveness of an algorithm's forward move. However, this measure will not indicate the 'length' of jumps made by backjumping algorithms.

4.5 Presenting Search Data

The phase transition experiments (Section 4.3) which form the basis of the empirical studies involve the search of many thousands of CSPs. Each of these searches produces a number of search cost measures, resulting in large quantities of raw data to be analysed. In order to draw useful conclusions from all the data produced by the experiments, it is essential that it is collected and presented in a manageable and comprehensible form: that is, we must obtain a good *view* of the data.

There are essentially two types of view of a phase transition experiment that should be available. Observing general behaviour of an algorithm on a problem class requires what might be termed a 'telescopic' view of the search data, while an understanding of the mechanics of individual searches by an algorithm requires a 'microscopic' view of the search process.

A number of techniques are employed to obtain search data and present it in a way that provides the views required. These are described below.

4.5.1 Summary statistics

The 'telescopic' view of a phase transition experiment is provided by a number of summary statistics. The following statistics are calculated for each measure of search cost on each ensemble of CSPs:

- the proportion of soluble problems, p_{sat}.
- the proportion of problems shown to be inconsistent (and insoluble) by consistency methods, p_{inc}, if applicable.
- the minimum and maximum costs.
- the mean cost.
- the standard deviation of cost.
- the median cost, plus higher cost percentiles $(75\%, 90\%, 99\%, 99.9\%, 99.99\%, 99.999\%)$ as applicable.
- the separate median costs for soluble and insoluble problems in the ensemble.

4.5.2 Search profiling

A 'microscopic' view of the search process for an algorithm is provided by a search profiling facility. The total search cost incurred at each level in the search tree is measured, for each type of cost measure, and can be used to form an individual profile for a single CSP search, or a summary profile for an ensemble of CSPs.

The use of search profiling is not new, and has been presented in well known empirical studies such as (Haralick and Elliott 1980). An illustration of the insights gained by looking inside search using the profiling facility is shown by Figure 4.1. The two plots compare the work done at each search depth by two different algorithms on the same problem. Algorithm A can be seen to be doing most of its work deeper into the search tree than Algorithm B, which spends most of its time at shallow search depths. Comparisons like this might help to explain differences in scaling properties or extremes of behaviour between these algorithms.

4.5.3 Cost percentiles

Investigation of the general behaviour of an algorithm will tend to focus on the mean and median values of various search costs. Occasionally, however, the distribution and extreme values of search cost over samples of CSPs are of interest. This particularly applies to the study of exceptionally hard problem behaviour (Section 1.5).

Ehps by definition represent extreme behaviour in a population of problems, and so will not be evident in a plot of median cost to find a solution; they will affect the mean cost, but not in a way which elucidates what is happening. For some problem classes, therefore, we plot the median and higher percentiles of search cost, up to the maximum cost, for the sets of problems; this follows graphs shown by (Hogg and Williams 1994) and (Gent and Walsh 1994a).

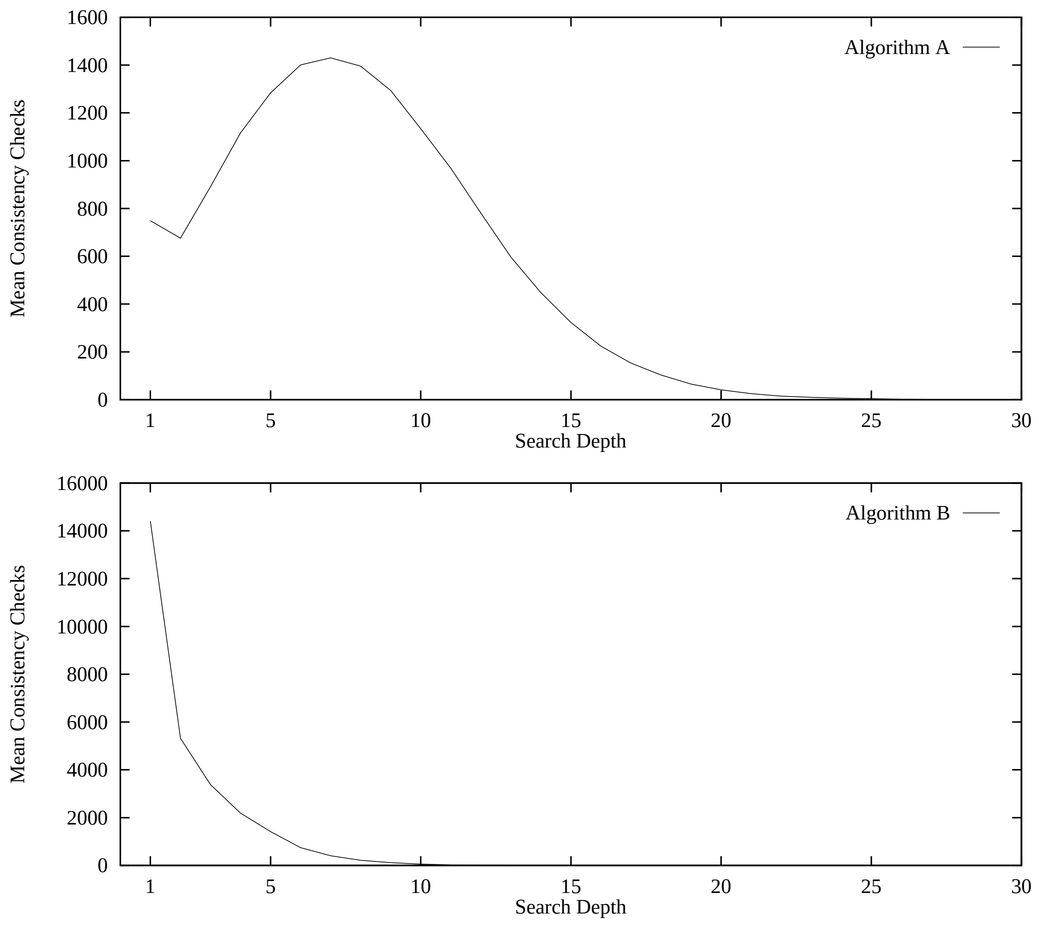


Figure 4.1: Search profiles for two algorithms on an ensemble of CSPs.

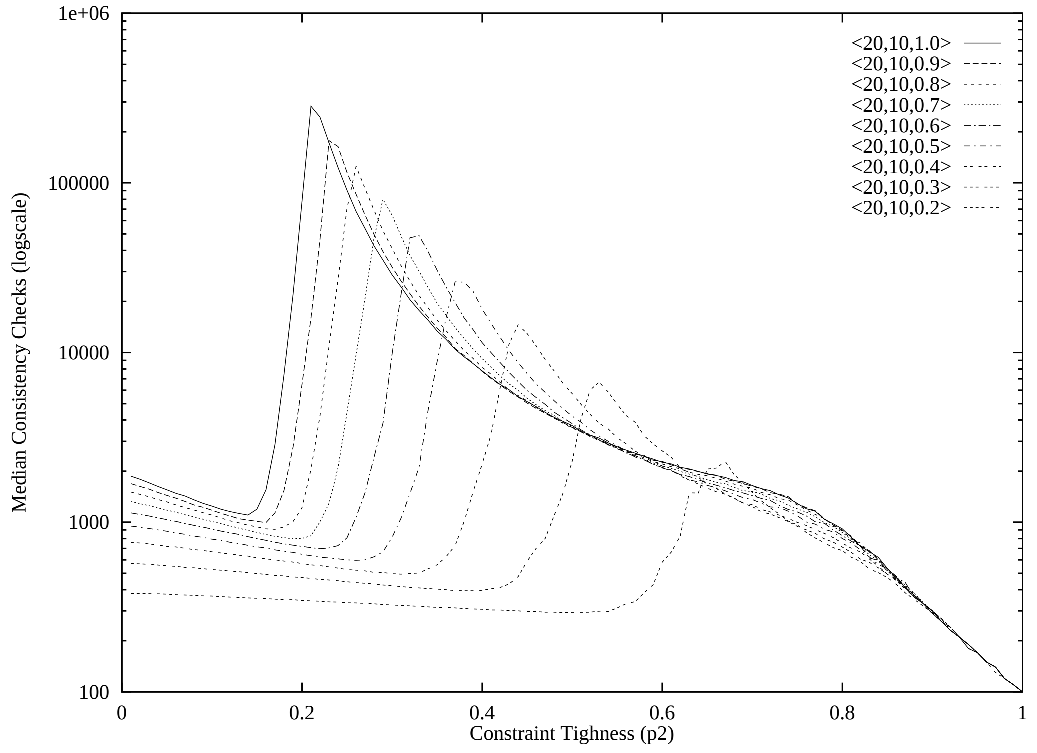
4.5.4 Cost against constrainedness, κ

Phase transition experiments (Section 4.3) on CSPs take the general form of varying the control parameter, p_2, for a number of related CSP classes. The location of the crossover point at the phase transition usually lies at different values of p_2 for each problem class studied, and the 'width' of the phase transition region often varies if the classes involved have different constraint topologies.

Plotting search cost against p_2 for a number of different CSP classes can obscure attempts to compare their phase transition behaviour, given the variations in apparent location and width. This problem can be alleviated, however, by plotting search costs for the different types of CSP against the general constrainedness parameter, κ, introduced in Section 3.2.

Plotting the empirical data for an $\langle n, m, p_1 \rangle$ problem class against κ is effectively doing so against a re-scaled p_2, and enables direct comparison of apparently diverse phase transition regions. To illustrate the improved view of phase transition behaviour offered by this technique, Figure 4.2 plots the median consistency checking cost of an algorithm applied to nine $\langle 20, 10, p_1 \rangle$ problem classes. The upper graph plots cost against p_2 while the lower graph plots cost against κ.

Re-scaling of the nine phase transitions around the value $\kappa = 1$ removes the overlapping curves that appear in the upper graph, and shows the patterns of the transitions to be more similar than



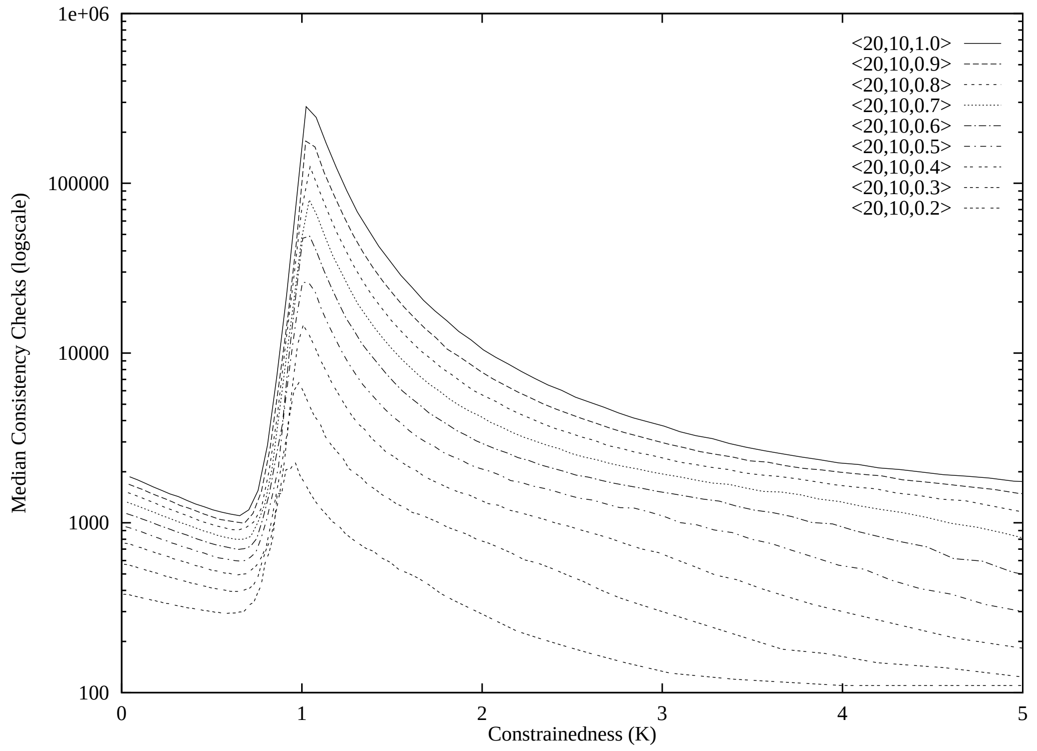


Figure 4.2: Plotting several phase transitions against p_2 (top) and κ (bottom).

the plots against p_2 would suggest. The alignment of the phase transition peaks in the lower graph also makes the scaling relationship as p_1 increases a lot clearer. A slight limitation of this technique, however, is that peaks for the more sparsely constrained problem classes do not quite line up at the value $\kappa = 1$ (this can be seen for $p_1 = \{0.2, 0.3\}$ in Figure 4.2). Recall from Section 3.2 that κ tends to underestimate the constrainedness of sparse CSPs.

Chapter 5

Exceptionally Hard Problems

The phenomenon of exceptionally hard problem (ehp) behaviour which is occasionally exhibited by complete CSP search algorithms was introduced in Section 1.5. To recap, ehps are extremely expensive searches which sometimes occur for a particular algorithm at values of the control parameter where most problems are extremely easy. The magnitude of these exceptional searches is often sufficient to greatly distort the mean cost observed for large populations of similar problems.

Exceptionally hard problems are believed to be a feature of exceptional behaviour being exhibited by a particular algorithm, rather than the discovery of a fundamentally difficult problem instance in the region where most are easy. In the phase transition, problems are hard on average due to the fact that either an exhaustive search must be undertaken if no solution exists, or an extensive search must be undertaken if the problem has few solutions. In some cases, though, a particular algorithm may find the solution to a phase transition problem very quickly due to a favourable instantiation ordering (Smith and Dyer 1996). In the easy-soluble region, most problems are very easy because they are underconstrained and have many solutions. In some cases, though, a particular algorithm may find a solution extremely difficult to find due to an exceptionally unfavourable instantiation ordering which it cannot recognise as such.

This chapter examines the incidence and magnitude of ehp behaviour for a number of complete CSP search algorithms. The study looks at ehps at both a ‘macroscopic’ level, considering their incidence over populations of problems, and at a ‘microscopic’ level, examining in detail the search process for many individual instances. A review of previous studies of the phenomenon is followed by a statement of the criteria used here to classify instances of ehp behaviour. The empirical study initially considers the basic backtracking algorithm, BT. The nature of exceptionally hard BT searches is examined, and strategies to reduce the incidence of ehp behaviour are proposed. These include the use of algorithms employing more sophisticated forms of forward and backward search move, and the use of a dynamic variable ordering strategy. Although their incidence is greatly reduced, these search strategies still produce some level of ehp behaviour for many types of CSP. We conclude by discussing the results of this study and the relevance of the ehp phenomenon in the overall study of algorithms for the CSP.

5.1 Related Work

(Hogg and Williams 1994) studied the application of depth-first search to large populations of graph colouring problems, and found that the most expensive of all searches were concentrated not at the phase transition, but in an area before it, where problems are easy in the average case. They identified the occurrence of these difficult problems with a second peak in the higher percentiles of search cost, which corresponds to the transition between polynomial and exponential scaling of the average search cost.

(Gent and Walsh 1994a) studied the performance of the Davis-Putnam satisfiability procedure on populations of 3-SAT problems, using two models of problem generation. They report that the most erratic behaviour in search cost occurred in the easy region of this class of problem, finding searches there that were orders of magnitude more expensive than the hardest phase transition problems. They attributed these exceptionally hard searches to either hard unsatisfiable instances (which can occur in relatively under-constrained problem regions for SAT), or to a bad choice of variable ordering producing a hard unsatisfiable subproblem.

In the case of binary CSPs, (Smith 1995) has reported a preliminary investigation into exceptionally hard problems occurring in the easy-soluble region. Smith suggested that factors contributing to ehps could include an unusually small number of solutions, a clustering of all solutions in one region of the search space, or an unusually large search space induced by the variable ordering strategy of the algorithm. (Frost and Dechter 1994) also report evidence of ehps in binary CSPs, noting that problems with many loose constraints can be much harder than those with fewer and tighter constraints.

Subsequent studies have empirically compared the levels of ehp behaviour between different complete search algorithms. (Davenport and Tsang 1995) showed that the use of backjumping techniques and sophisticated variable ordering techniques significantly reduces the incidence of ehp behaviour for the FC algorithm, applied to sets of graph colouring problems.

Similarly, (Baker 1995) applied various algorithms with varying levels of backjumping capability to large sets of 100 variable 3-colouring problems. Ehp incidence was reduced by more intelligent backjumping, and Baker made the claim, based purely on empirical results, that dependency-directed backtracking (Stallman and Sussman 1977) eliminates the phenomenon on these problems completely. Ehp behaviour was attributed simply to *thrashing* behaviour (Mackworth 1977), which sees algorithms with more naive forms of backward search move continue to make the same mistakes in a pathological fashion. The exact nature of this behaviour, however, was not examined closely[1].

Recently, (Gomes *et al.* 1997) have analysed the distribution of search cost over populations of soluble combinatorial problems, and show that these distributions are invariably 'heavy-tailed' in nature. That is, given sufficiently large populations, the outlying cases tend to be situated extremely far from the average case. They show that this distribution can in fact be modelled as a stable distribution, using a technique that has been applied to real-world chaotic phenomena such as weather patterns and stock market behaviour. In addition, they show that a strategy which

[1]Curiously, Baker erroneously claimed that Hogg & Williams and Gent & Walsh offered no explanation for the ehp behaviour that they observed.

re-starts search from a different point after a certain cost limit has been reached is effective in curbing the heavy-tailed distribution; that is, it reduces the likelihood of an exceptionally hard search on an easy soluble problem.

Ehps appear to be a feature only of complete search methods. Neither (Hogg and Williams 1994), in looking at heuristic repair methods on graph colouring problems, nor (Gent and Walsh 1994b) in applying the GSAT hill-climbing procedure to satisfiability problems, have found any evidence of ehp behaviour for these algorithms. Similarly, (Davenport and Tsang 1995) found no sign of ehps when applying the GENET neural network to graph colouring problems.

5.2 Criteria Used

In the study presented in this chapter (and in later studies), a problem instance is said to be an exceptionally hard problem (ehp) if:

1. it occurs in the region where almost all problems are soluble, and on average easy to solve (that is, outside the mushy region);

2. it is much more difficult, by at least an order of magnitude, than almost all other problems with the same parameter values;

3. it is more difficult than almost all the problems occurring in the mushy region.

This is intended to be a description, rather than a precise definition. As will be seen, a problem which is exceptionally hard for one algorithm may be very easy to solve for another algorithm (or even for the same algorithm with a different variable instantiation order). Hence, ehps do not appear to be exceptional problems, but rather to cause exceptional behaviour in a specific algorithm. So although we should consider individual ehps and why the algorithm gets into such extreme difficulties with these problems, it is the incidence of ehps in populations of problems, in relation to particular search algorithms, that is of most interest.

For the purposes of the empirical study presented later, we define the mushy region (Section 1.4) arbitrarily as the range of values of p_2 for which the probability that a problem has a solution lies between 0.01 and 0.99. This allows us to estimate its boundaries by finding the largest p_2 for which more than 99% of problems are soluble and the smallest p_2 for which fewer than 1% of problems are soluble.

The empirical study of ehp behaviour is presented in the following sections. The problems used are binary CSPs, generated according to the Model B random generation method presented in Chapter 3.

5.3 Basic Ehp Behaviour

The empirical study of ehp behaviour begins with the simple backtracking algorithm, BT. (Prosser 1993) notes that, "We should consider BT as describing the most primitive forward move (checking against past variables) and the most primitive backward move (chronological backtracking)".

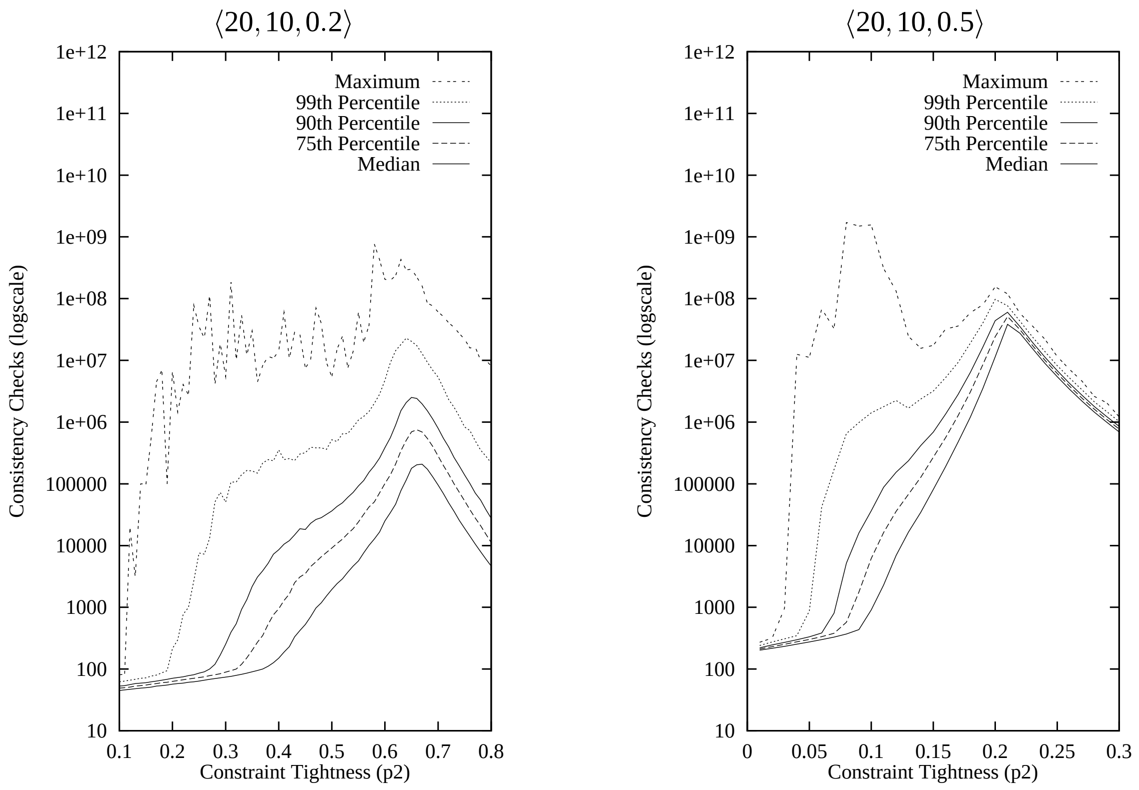


Figure 5.1: Ranges of consistency checking cost for BT on $\langle 20, 10, 0.2\rangle$ and $\langle 20, 10, 1.0\rangle$ CSPs.

Three phase transition experiments were conducted using the $\langle 20, 10, 0.2\rangle$, $\langle 20, 10, 0.5\rangle$ and $\langle 20, 10, 1.0\rangle$ CSP classes. These problem classes were chosen to provide a range of constraint tightnesses over which ehp behaviour may vary considerably. The size of each p_2 step for the three classes is 0.01, and ensembles of 5,000 problems were generated at every $\langle 20, 10, p_1, p_2\rangle$ point in order to increase the likelihood of observing extreme search behaviour. BT was applied to each CSP in the study using a static lexical (effectively random) variable instantiation ordering.

5.3.1 Naive chronological backtracking

Figure 5.1 shows the median and higher percentiles of consistency checking cost for BT over the $\langle 20, 10, 0.2\rangle$ and $\langle 20, 10, 1.0\rangle$ problem classes. Costs are plotted against constraint tightness, over a range covering the region leading up to and beyond the phase transition.

Looking first at the median cost of solving these problems, the usual phase transition behavior is seen, with a peak at the point where half the problems are soluble and half insoluble. In the case of $\langle 20, 10, 1.0\rangle$, when $p_2 < 0.2$, all the problems searched have solutions, and when $p_2 > 0.24$, none do. At small values of p_2 for both classes, most problems are very easy to solve: up to the point where the gradient of the median cost curve starts to climb steeply, at least half the problems can be solved without ever backtracking to a previous variable.

However, the higher percentiles show that for most of the easy-soluble regions, except for the smallest values of p_2, there is a small proportion of problems which is extremely costly to solve. For both classes, the most expensive searches occurring to the left of the phase transition are much more costly (by at least an order of magnitude) than 99% of the other problems occurring in the

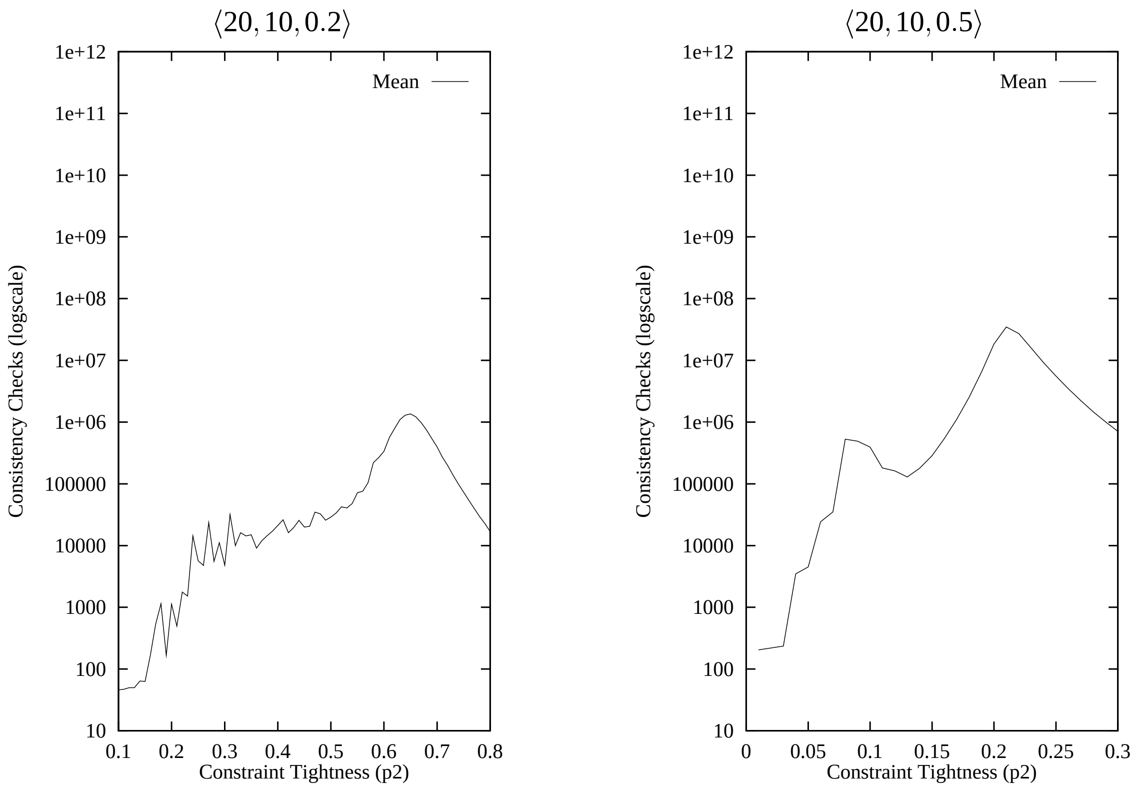


Figure 5.2: Mean consistency checking cost for BT on $\langle 20, 10, 0.2\rangle$ and $\langle 20, 10, 1.0\rangle$ CSPs.

ensemble at the same value of p_2; much more costly (again, by at least an order of magnitude) than 99% of the sample problems in the phase transition; and occur well below the values of p_2 at which the first insoluble problems are found. Thus they fit the description given earlier and are ehps. These rare problems are sufficient to increase the mean cost enormously: on the $\langle 20, 10, 1.0\rangle$ class for instance, the most expensive problems at around $p_2 = 0.1$, which take more than 10^9 consistency checks, are each sufficient to increase the mean by 200,000, at a point where the median cost is less than 1,000. This effect of ehp behaviour on the mean costs of BT over the $\langle 20, 10, 0.2\rangle$ and $\langle 20, 10, 1.0\rangle$ classes is illustrated by the plots in Figure 5.2.

It should again be emphasized that all of the ehps shown in Figure 5.1 are soluble CSPs. For all the experiments reported in this thesis, no instances of insoluble ehps in the easy-soluble region of any CSP class have been found. Furthermore, all ehps which we have studied in detail appear to have many solutions, typical for such underconstrained problems. As discussed in (Smith 1995), if an insoluble problem were to occur in the easy-soluble region of a CSP class, it would be extremely hard to prove insoluble. It seems, though, that ehps of this type are exceptional even amongst ehps.

5.3.2 Inside a BT ehp

To gain some insight into what turns a BT search into an ehp, we examined in detail the progress of the algorithm on one of the difficult $\langle 20, 10, 1.0\rangle$ problems when $p_2 = 0.07$; this problem took 31 million consistency checks to solve. BT frequently finds partial solutions with 19 of the 20 variables instantiated, and then backtracks, since it cannot find a value for the last variable which is consistent with the past assignments. This is due to the fact that the last variable (v_{20}) has no values that are consistent with the first 8 instantiations made. Eventually, the eighth instantiation is changed, from v_8=1 to $v_8 = 4$, and a solution is found almost immediately.

It is clear that the reason for the high cost of solving this problem is that the algorithm is thrashing (Mackworth 1977); repeatedly re-discovering essentially the same failure. Attributing ehp behaviour to thrashing is not new: (Baker 1995) makes a similar suggestion, although his explanation applies only to problems with disconnected constraint graphs. Mackworth pointed out that a local inconsistency can cause thrashing in an algorithm that is unable to detect it. But from results on establishing consistency in CSPs, presented later in Chapter 6, we know that in this case the ehp could not be avoided by making the problem node, arc or path consistent. However, if, after the first 8 assignments, we were to form the subproblem consisting of the future variables and those of their values which are consistent with the past assignments (as is done explicitly by the forward checking algorithm), the subproblem could be thought of as being node inconsistent, since v_{20} would then have no values remaining in its domain. BT is subject to thrashing in this situation because it cannot detect this node inconsistency without an exhaustive search of the insoluble subproblem.

This 'insoluble subproblem' explanation for ehp behaviour is very similar to the experience reported by (Gent and Walsh 1994a); they found a satisfiability problem which required more than 350 million branches to solve, using a simplified version of the Davis-Putnam SAT algorithm. The first choice made by the algorithm led to a very difficult unsatisfiable problem, which required almost all the search effort; the alternative choice led immediately to a solution. They conclude that, "These difficult problems are either hard unsatisfiable problems or are satisfiable problems which give a hard unsatisfiable subproblem following a wrong split". They also report similar experience with travelling salesman problems in (Gent and Walsh 1995b); they were concerned with the decision problem, i.e. whether there is a tour of length l or less, and found that if l is increased from the minimum possible length, the problem can sometimes become much more difficult, rather than easier, as would be expected. This is because a bad decision made early on can lead to a long and unsuccessful search for an acceptable tour of the remaining cities: the bound is insufficiently tight to cut off search until nearly all the cities have been considered, whereas a tighter bound allows backtracking to be triggered much earlier.

In order for BT to find a problem in the easy-soluble region exceptionally difficult in this way, a combination of circumstances must occur:

- the first few assignments must together conflict with every value of the last variable in the instantiation order (or perhaps, in larger problems, one of the last);
- it must be easy to find values for the intervening variables which are consistent with the first few assignments.

The first condition ensures that an insoluble subproblem is created. The second ensures that searching the subproblem takes a very long time: the algorithm does not encounter any failure causing it to backtrack to a previous assignment until the last variable is reached; and the search tree has many branches, because the first variables in the subproblem have many values which are consistent with the past assignments.

These two conditions are contradictory, given the Model B CSP generation model used to produce Figure 5.1. Hence, the combination is inherently very rare. However, circumstances can be imagined in which the conditions would be easier to meet: for instance, if the tightness of the constraints varied, in such a way that the constraints between the first few variables and the last were tighter than other constraints. As discussed in (Baker 1995), if the graph is disconnected, and the algorithm does not solve each component separately, thrashing can occur relatively easily.

Recently, (Smith and Grant 1997) have looked at exceptionally hard problem behaviour by the BT algorithm in more detail. They use the insoluble subproblem explanation as a basis for constructing a model of ehp behaviour and its expected cost. Looking initially at BT on CSPs whose constraint graph is a clique (i.e. $p_1 = 1$), the probability that the conditions leading to a hard insoluble subproblem arise has been calculated, as has an estimate for the resulting search cost. From this, Smith and Grant derive a theoretical cost distribution which suggests that the hardest searches in the easy-soluble region will form a second peak in the higher cost percentiles similar to that which (Hogg and Williams 1994) have observed empirically. It is anticipated that this model will be extended to other algorithms in future, such as forward checking, and to CSPs with non-clique constraint graphs.

5.3.3 Improving BT's variable ordering

The empirical study of BT described above deliberately uses what is effectively a random instantiation ordering, so as to test the algorithm in perhaps its most naive form. It may be the case, however, that a more informed choice of instantiation order may have an impact of the incidence of ehps. To test this, a second study of BT was conducted, using a heuristic technique described by (Gibbs *et al.* 1976) to give each search a small-bandwidth static variable ordering. The bandwidth of an ordering is the maximum distance in the ordering between any pair of variables which have a constraint between them. A small-bandwidth ordering should make ehps less likely to occur, and less costly if they do occur. This is because when the instantiation of a variable causes a domain wipeout in some future variable, the smaller the bandwidth, the smaller the distance between these two variables, and therefore the smaller the number of intervening variables over which the algorithm has to backtrack.

Dense constraint graphs are not particularly relevant to such a study, since if all variables are constrained by nearly every other, then a small-bandwidth ordering will not exist. A phase transition experiment using this technique was therefore conducted using the sparse $\langle 20, 10, 0.3\rangle$ problem class. The size of each p_2 step was again 0.01, but this time ensembles of 10,000 problems were generated at every $\langle 20, 10, 0.3, p_2\rangle$ point to further increase the chances of exceptional searches.

Figure 5.3 plots the median and higher percentiles of consistency checking cost for BT over the problem class, against constraint tightness. Compared to the $\langle 20, 10, 0.2\rangle$ plot in Figure 5.1, ehp behaviour using this version of the algorithm is as bad, if not worse, than the naive lexical ordering. It appears that a more sophisticated approach is needed to tackle the phenomenon.

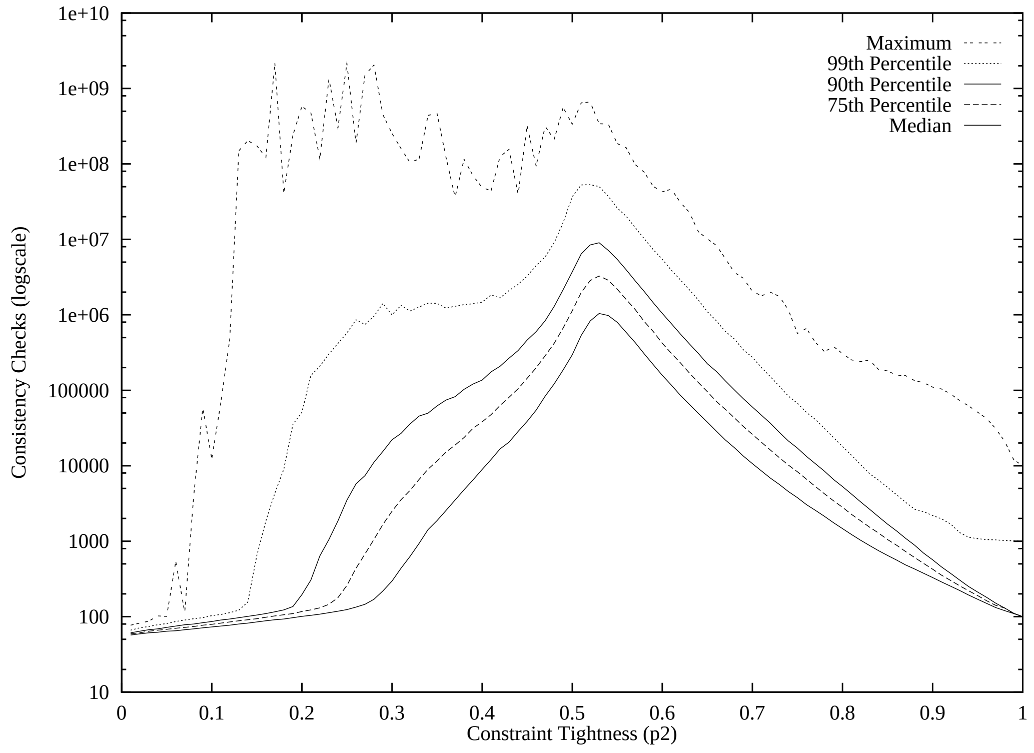


Figure 5.3: Ranges of consistency checking cost for BT (using a minimum bandwith ordering) on $\langle 20, 10, 0.3\rangle$ CSPs.

5.4 Strategies for Avoiding Ehps

BT is of course an extremely naive algorithm, and we might expect that more sophisticated algorithms will reduce the incidence of ehps, as well as improving the average cost of search. In one sense, it is trivial even for BT to avoid an ehp in an individual problem: the BT ehps that have been examined, including that described in Section 5.3.2, arise through encountering insoluble subproblems early in the search; if a different instantiation order is used, the subproblem will not be created. Another algorithm applied to an ehp may also find the problem easy simply because it considers variables in a different order. What is required, however, is an algorithm which reduces the incidence or difficulty of ehps overall.

There are, in theory, two potential ways for a search algorithm to avoid ehps which arise through encountering insoluble subproblems early in the search: one is to avoid getting into such subproblems in the first place, and the other is to detect that the subproblem is insoluble more quickly. Sections 5.5, 5.6 and 5.7 study these approaches in isolation, and then in tandem.

5.5 Looking Forward

Algorithms which perform a lookahead style of consistency checking (Section 1.3.1) to some extent make the future subproblem consistent with each instantiation made. Such a process is likely to detect the insolubility of many of the subproblems which lead to ehp searches. Below, we consider algorithms performing two levels of lookahead: forward checking and maintaining

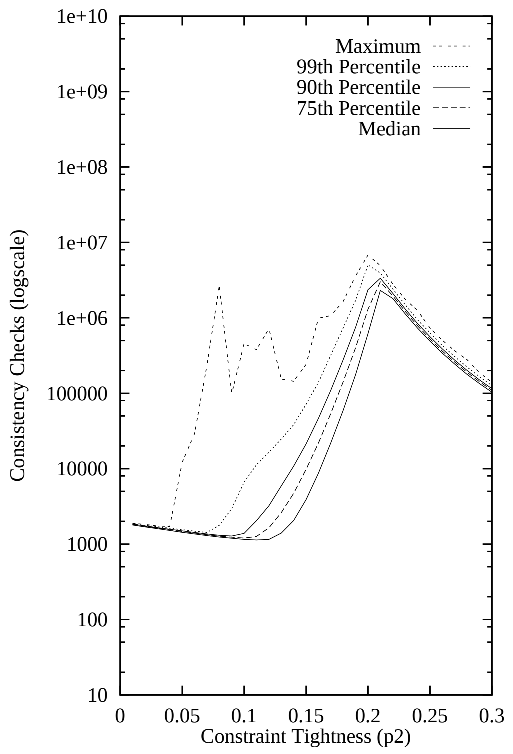


Figure 5.4: Ranges of consistency checking cost for FC on $\langle 20, 10, 1.0\rangle$ CSPs.

arc consistency. The effects of using these algorithms with a variable ordering heuristic are also examined.

5.5.1 Forward checking

BT may thrash when it encounters an insoluble subproblem early in the search. If node inconsistencies are created in the subproblem following an instantiation, the forward checking algorithm (FC) can avoid an exhaustive search. By removing those values from the domains of future variables which conflict with the past assignments, FC can detect when a future variable has no remaining values, and hence recognize immediately that the subproblem is insoluble.

FC, using a static lexical variable ordering, was applied to the $\langle 20, 10, 1.0\rangle$ CSPs that were searched by BT in the previous section. Sample sizes were increased to 10,000 problems at each p_2, however, so an additional 5,000 CSPs were effectively added to each ensemble. Figure 5.4 shows the results of applying FC to these problems, in a similar style to Figure 5.1. As expected, FC reduces the incidence and severity of ehps considerably. However, there are still some problems which are extremely difficult by comparison with the median difficulty at the same constraint tightness. Examination of individual problems of this kind, some of which is presented later in Section 5.8, shows that the ehp is still due to an insoluble subproblem formed by the first few instantiations. Once again, when the algorithm eventually backtracks to this point and tries a different instantiation, a solution can be found almost immediately. However, rather than a simple node inconsistency in the subproblem, there is an arc inconsistency involving the last variable. Although this will manifest itself eventually in a domain wipeout of the last variable (which FC can detect, and so do much better than BT even on these more complex problems) it still has to do a good deal of backtracking before proving that the overall subproblem is insoluble.

5.5.2 The effect of adding search heuristics

CSP algorithms with lookahead capability, including FC, often use a dynamic variable ordering (DVO) strategy (Section 2.4). A popular technique is to instantiate the the variable that has the fewest remaining values left in its domain, in an attempt to follow the 'fail-first' principle[2]. We use a variant of this heuristic, proposed by (Frost and Dechter 1994) which selects the *first* variable to instantiate as the one with the highest degree, i.e. the one constraining the largest number of other variables. Thereafter, the 'smallest remaining domain' strategy is used. Frost and Dechter use the generic term DVO for this specific heuristic, but here we use the name FFdeg, for fail-first with initial degree ordering.

Four phase transition experiments applying the FC-FFdeg algorithm were conducted, using the $\langle 20, 10, 0.2\rangle$, $\langle 20, 10, 0.3\rangle$, $\langle 20, 10, 0.5\rangle$ and $\langle 20, 10, 1.0\rangle$ CSP classes studied earlier. For these experiments, ensembles of 10,000 CSPs were generated and searched at every $\langle 20, 10, p_1, p_2\rangle$. Figure 5.5 plots the median and higher percentiles of consistency checking cost, against p_2, around the phase transition regions for each of these problem classes.

Remarkably, the heuristic appears to have completely eliminated thrashing in the easy-soluble regions of the densely constrained classes. It must meet arc inconsistent subproblems at roughly the same rate as BT and FC (though not the same ones, since the instantiation order is different), and it is surprising that simply considering next the variable with smallest domain is sufficient to detect the inconsistency when FC cannot. To provide further verification for the elimination of ehp behaviour on densely constrained problems, FC-FFdeg was again applied to the $\langle 20, 10, 1.0\rangle$ class. This time samples of 50,000 problems were searched at each value of p_2. A small increase in the maximum cost was observed, but without any signs of ehp behaviour.

For the more sparsely constrained problem classes, however, ehp activity can still be observed with FC-FFdeg. Clear ehps are to be found in the easy-soluble region of the $\langle 20, 10, 0.2\rangle$ class, while the instability of the maximum cost curve for $\langle 20, 10, 0.3\rangle$ suggests that ehps will be observed with larger sample sizes. The incidence of ehp behaviour with FC-FFdeg on sparse CSPs is clearly lower than that for both BT and plain FC, however. In Section 5.5.3, the incidence of ehp behaviour for this algorithm on larger sparse problems is examined. Later, Section 5.8 also looks at the behaviour of specific ehps for FC-FFdeg.

Ignoring the higher percentiles for a moment, the median cost curves in Figures 5.5 and 5.1 demonstrate how much more expensive BT is than FC-FFdeg, for most values of p_2. For the $\langle 20, 10, 1.0\rangle$ class, the peak in median cost occurs at the same value of p_2 for both algorithms, but is about two orders of magnitude higher for BT than for FC-FFdeg. When p_2 is small, however, the median cost is lower for BT than for FC: when the constraints are very loose and solutions are easy to find, the effort of checking all the values of the future variables is often not worthwhile.

[2]Chapter 8 examines this principle, and DVO heuristics based on it, in more detail.

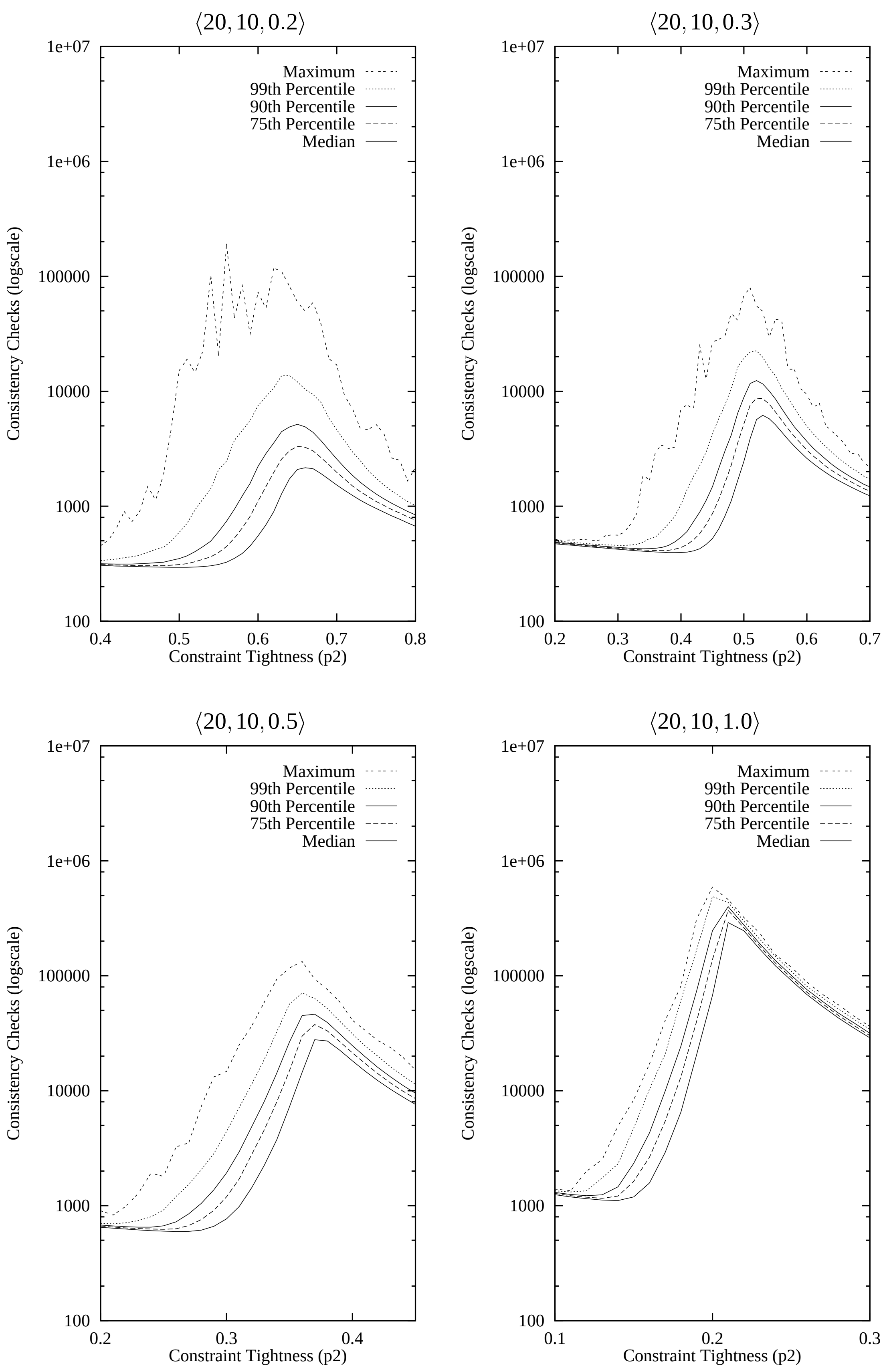


Figure 5.5: Ranges of consistency checking cost for FC-FFdeg on four $\langle 20, 10 \rangle$ CSPs.

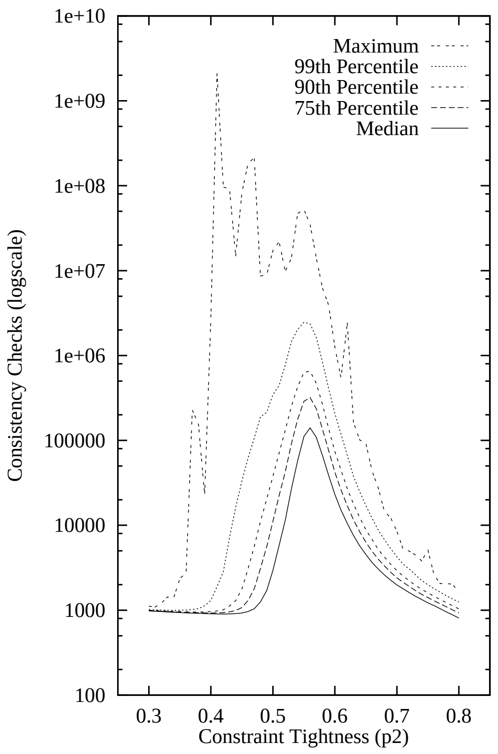


Figure 5.6: Ranges of consistency checking cost for FC-FFdeg on $\langle 50, 10, 0.1\rangle$ CSPs.

5.5.3 Large sparse problems

From the plots of phase transition behaviour that have been presented so far, it is clear that the phase transition is less sharp in sparse problem classes than in dense ones, given the same values of *n* and *m*. It is also known from many empirical studies (for instance (Prosser 1996)) that for low values of p_1, the phase transition grows sharper as *n* is increased.

It might be suspected, therefore, that the ehps still seen for FC-FFdeg in the $\langle 20, 10, 0.2\rangle$ problem class are a side-effect of this. It may be the case that in larger sparse problems, as the phase transition becomes more abrupt, ehps will disappear from these problems as well as the more densely constrained ones (for this algorithm).

Therefore, another phase transition experiment using FC-FFdeg was conducted. This time, the $\langle 50, 10, 0.1\rangle$ CSP class was examined, with p_2 varied in steps of 0.01 and ensembles of 10,000 problems generated at each point. Figure 5.6 plots the median and higher percentiles of consistency checking cost against constraint tightness. It is worth pointing out that data plotted in Figure 5.6 is based on solving 230,000 individual CSPs; since the peak median cost is approximately 1 million consistency checks, and since it takes hours to solve some of the worst individual problems, Figure 5.6 represents a considerable investment of cpu time.

As expected, the $\langle 50, 10, 0.1\rangle$ problems show a much sharper peak in the median than the $\langle 20, 10, 0.2\rangle$ problems, and the mushy region is much narrower (using the criteria given in Section 5.2 its range is 0.53 - 0.59). So as far as the median behaviour is concerned, the increase in *n* from 20 to 50 has caused the phase transition to become more abrupt, even though the density is lower. However, far from disappearing, ehps are if anything more common, and are more extreme, than in the smaller problems. This suggests that ehps are not a transient phenomenon, associated only with small problems, and that as n increases and the phase transition becomes more abrupt, ehps will continue to appear and will be increasingly extreme.

Figure 5.6 shows also a ‘spike’ in the maximum cost curve in the insoluble region, at $p_2 = 0.63$. This is extremely untypical behaviour: in the insoluble region, all the maximum cost curves that we have seen, with this one exception, decline smoothly from the peak. This problem is itself insoluble, and not an ehp by our criteria; it is so far unique in our experiments, and is examined briefly in Section 5.8 as a possible exceptionally hard problem occurring in the insoluble region.

5.5.4 Maintaining arc consistency

As discussed in Section 5.5.1, forward checking avoids many of the ehps that simple backtracking suffers from because it can detect immediately the infeasibility of the subproblem in which thrashing occurs. FC is itself subject to ehps, however, and in those cases the reasons for the infeasibility of the subproblems are clearly more complex than a simple domain wipeout.

When a future variable has no values consistent with the current partial solution, the subproblem consisting of the future variables and their remaining values can be thought of as node inconsistent. (Mackworth 1977) pointed out that a local inconsistency of this kind can cause thrashing in an algorithm that is not able to detect the inconsistency. Although node inconsistencies do not occur in our experimental CSPs, the basis of the forward checking algorithm is that node inconsistencies can occur in subproblems, even when the overall problem is node consistent. BT is subject to thrashing in problems which FC finds easy precisely because of this.

In the light of this, it is natural to suppose that some of the ehps which FC suffers from are caused by arc inconsistency in the insoluble subproblem. We therefore investigated the impact on ehp behaviour of an algorithm which re-establishes arc consistency whenever a subproblem is created. Thus, the MAC algorithm (Section 2.1) was applied to the $\langle 50, 10, 0.1 \rangle$ CSPs examined in the previous section. As for FC, MAC was combined with FFdeg dynamic variable ordering to produce MAC-FFdeg.

Figure 5.7 plots the median and higher percentiles of consistency checking cost against constraint tightness. As expected, MAC-FFdeg is less subject to ehps than FC-FFdeg. However, a very prominent ehp can be seen at $p_2 = 0.49$, which costs over 1.547 billion consistency checks and 38 million nodes to solve. This particular problem is examined in more detail in Section 5.8. We assume that in at least some MAC ehps, the insoluble subproblem is path inconsistent. Re-establishing path consistency in every subproblem would then still further reduce the risk of thrashing (which is already very small), but probably at the expense of a vastly increased average cost.

5.5.5 Summary

The approach of looking ahead into the future subproblem during search does indeed bring benefits in terms of ehp behaviour as well as average search cost. FC reduces the incidence of ehp behaviour over the $\langle 20, 10, p_1 \rangle$ problem classes, although far from completely. Those ehps that

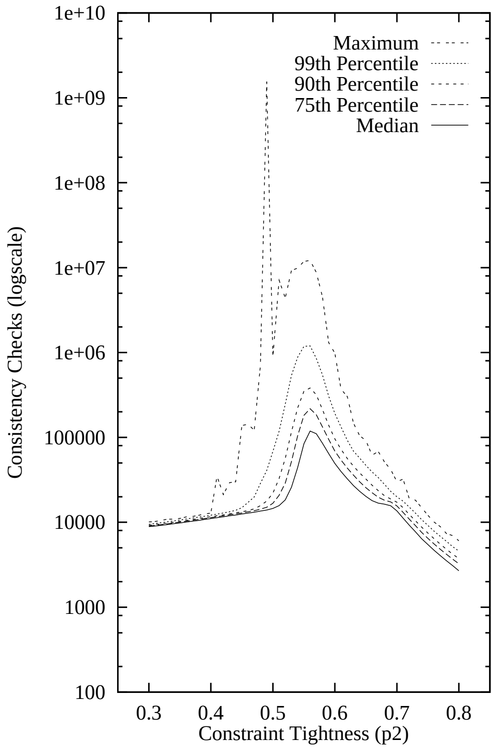


Figure 5.7: Ranges of consistency checking cost for MAC-FFdeg on $\langle 50, 10, 0.1\rangle$ CSPs.

are found for FC still appear to be caused by insoluble subproblems that the algorithm cannot identify.

The addition of 'fail-first' dynamic variable ordering, however, has a major effect on ehp behaviour in densely constrained problem classes. There is no hint of FC-FFdeg encountering ehps in the $\langle 20, 10, 1.0\rangle$ problem class, nor, we conjecture, in densely constrained problems generally. Further experimentation carried out on $\langle 20, 10, p_1\rangle$ problems shows little indication of ehps occurring for $p_1 \geq 0.5$. We have also examined larger dense problem classes ($\langle 20, 20, 1.0\rangle$ and $\langle 30, 10, 1.0\rangle$) and found similarly that the higher percentiles are all close together, as in Figure 5.5, with no sign of ehps. If these are typical, then for FC-FFdeg, ehps in dense problems must be, at the least, extremely rare compared with those in sparse problems.

Extending lookahead capability to that of MAC further reduces ehps. The MAC algorithm and its effect on both the average and extremes of search cost are examined in detail in Chapter 7, where a far wider range of CSP classes are examined.

In terms of where ehp behaviour occurs over a problem class, the experiments with BT show that it is subject to thrashing over a much wider range of values of p_2 than the lookahead algorithms are. Because the median behaviour of BT over the phase transition is so bad compared to FC, the very difficult problems in the soluble region do not always meet all the criteria for ehps given earlier. However, it is clear that problems which are at least as difficult as most of those occurring in the phase transition occur at a higher rate, as well as over a much wider range of values of p_2, for BT than for FC and MAC.

5.6 Jumping Back

In searching for an algorithm which would avoid the thrashing that BT is subject to, another possibility is to use some kind of informed backtracking, rather than chronological backtracking as in the algorithms considered so far. When an insoluble subproblem is created and the infeasibility detected as the last few variables are instantiated, such an algorithm might be able to backtrack immediately to the true cause of the failure, that is, the first few instantiations.

(Baker 1995) suggests that *all* exceptionally hard problems can be defeated by a search strategy which uses a sufficiently intelligent backtracker. He presents experiments on graph coloring problems using dependency-directed backtracking, which records all the nogoods it discovers during search and uses these to avoid repeating work. However, it accumulates an ever-increasing number of nogoods as it backtracks, which may not be practicable in difficult cases.

As discussed in Section 2.1, the conflict-directed backjumping (CBJ) algorithm maintains for each variable a *conflict set*. This is the set of past variables which it failed consistency checks with. If no consistent instantiation can be found for a variable v_i, the algorithm jumps back to the deepest variable, v_h, listed in its conflict set. In the individual CSP discussed in Section 5.3.2, which causes BT to thrash, every value for the twentieth variable conflicts with one of the first eight assignments. Applying CBJ to this problem, the conflict set for v_{20} contains (at most) the first eight variables, and when no value can be found for v_{20}, the algorithm jumps back to v_8, and tries an alternative value for it. This leads directly to the solution which BT eventually finds. Thus, CBJ can jump out of a node inconsistent subproblem as soon as the variable which has no consistent values left in its domain is reached for the first time.

Figure 5.8 shows the CBJ algorithm applied to the set of $\langle 20, 10, 1.0\rangle$ problems solved earlier using BT (Figure 5.1) and is also comparable with Figures 5.4 and 5.5. As for the experiment with BT, the algorithm uses a static lexical instantiation ordering. CBJ does much better than BT everywhere, both on average and especially in avoiding ehps. Although CBJ is more expensive than FC on average over the phase transition, its worst-case performance at low values of p_2 is better. In the problems which FC finds most difficult, the first few instantiations create an arc inconsistent subproblem, with the last variable involved in the inconsistency. FC can only detect an inconsistency later on when this causes a domain wipeout; it then has to backtrack chronologically over several variables before arriving back at the true cause of the inconsistency. CBJ, on the other hand, cannot detect the inconsistency until it reaches the domain wipeout; it then jumps back to the immediate cause, which is the point where FC would detect it. However, from there, if there are no more values for the current variable, it can often jump back to the ultimate cause of the inconsistency, i.e. the first few instantiations, and thereby beat FC.

5.7 Combining Techniques

If the two approaches of looking forward into insoluble subproblems and jumping back out of them are successful in reducing the occurrence of ehps, then we might expect that combining these techniques produce even better results. Hybrid combinations such as FC-CBJ and MAC-CBJ are discussed in Section 2.1, and here we study their effects on ehp behaviour. As with FC and

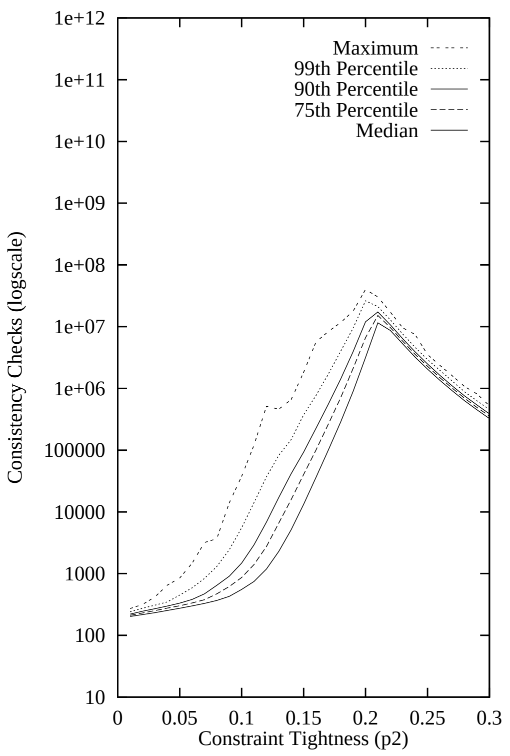


Figure 5.8: Ranges of consistency checking cost for CBJ on $\langle 20, 10, 1.0\rangle$ CSPs.

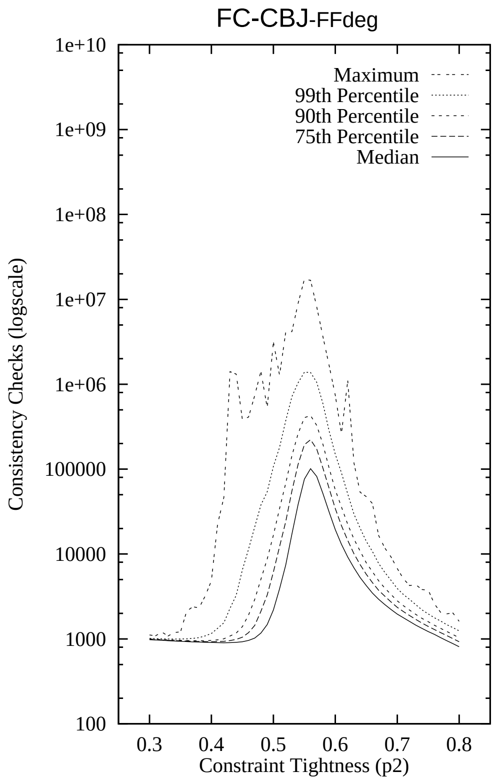
MAC, the hybrids are used with the FFdeg heuristic to produce FC-CBJ-FFdeg and MAC-CBJ-FFdeg.

The implementation of dynamic variable ordering used by both algorithms (discussed in Section 2.4) means that the sets of nodes visited by FC-CBJ-FFdeg and MAC-CBJ-FFdeg are a subset of those visited by FC-FFdeg and MAC-FFdeg respectively. Therefore the hybrid algorithms can only find a problem exceptionally hard if the basic lookahead algorithm does so. Equally, if the basic algorithm finds a problem exceptionally hard, the hybrid algorithm must meet the same insoluble subproblem which causes the basic algorithm to thrash, and can only avoid thrashing if it can jump out of it.

Figure 5.9 shows the results of running the two hybrid algorithms over the $\langle 50, 10, 0.1\rangle$ problem class, as used in Figures 5.6 and 5.7. It is noticeable that the addition of CBJ to the algorithms is effective in further reducing the occurrence of ehps, but does not reduce the average search cost by any significant amount. The effects of adding CBJ to FC and MAC are studied in greater detail in Chapter 7.

We have looked at the performance of FC-CBJ-FFdeg on a number of individual CSPs which are ehps for FC-FFdeg because of arc inconsistency in a subproblem. Sometimes the arc inconsistency is particularly simple, i.e. two future variables are left without any mutually consistent values by the first few instantiations. FC-CBJ-FFdeg can then jump back to the true cause of the difficulty as soon as it has tried to assign a value to one of these two variables. In such cases, FC-CBJ-FFdeg does better than MAC-FFdeg, because it does not have the overhead of the arc consistency algorithm.

Often, however, the proof of arc inconsistency is complex, involving many of the future variables. In such cases, FC-CBJ-FFdeg cannot jump out of the subproblem as soon as it meets the



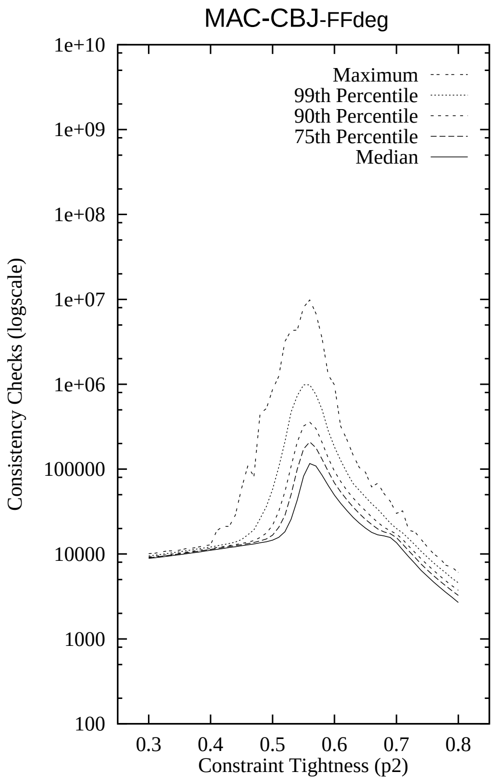


Figure 5.9: Ranges of consistency checking cost for FC-CBJ-FFdeg and MAC-CBJ-FFdeg on $\langle 50, 10, 0.1\rangle$ CSPs.

inconsistency, because of the other variables involved. However, even being able to jump back over a few variables is an advantage over chronological backtracking.

The comparison between MAC-FFdeg and MAC-CBJ-FFdeg is similar. The main benefit of MAC-CBJ-FFdeg is that it improves the worst-case performance of MAC-FFdeg in the easy-soluble region, in some cases dramatically, as with the individual problem at $p_2 = 0.49$. Just as FC-CBJ-FFdeg cannot directly detect arc inconsistency but can still do well by jumping back in arc inconsistent subproblems, MAC-CBJ-FFdeg can similarly do well even though it cannot detect whatever higher level of inconsistency is present in the subproblem.

The effect of adding conflict-directed backjumping to both FC and MAC is examined in detail for some individual ehp cases in the following section.

5.8 Inside Ehps

To understand better the causes of ehps, we have examined carefully many individual $\langle 50, 10, 0.1\rangle$ problems which FC-FFdeg and MAC-FFdeg found exceptionally hard. The focus on $\langle 50, 10, 0.1\rangle$ is not significant, but as this is the largest class of CSPs studied so far, we tend to see the most extreme individual ehp behaviour here.

An analysis of three ehps is presented here: two are found by FC-FFdeg, while the other is encountered by MAC-FFdeg. The behaviour observed is typical of that seen for the other ehps that have been analysed. The effect of increasing the level of lookahead on the two FC ehps is also examined, as is the effect of introducing CBJ to all three. Finally, graphical profiles of some interesting individual searches are examined.

5.8.1 Two forward checking ehps

Section 3.6 notes that every experimental CSP is generated from a single integer seed, so that ensembles of problems effectively contain individually numbered CSPs. The example ehps are described below are thus referred to by their seed value.

Problem 898 in the $\langle 50, 10, 0.1, 0.47\rangle$ ensemble is an ehp for FC-FFdeg. The median search cost over the ensemble is just over 1000 consistency checks, but in this case the algorithm makes over 190 million consistency checks and visits 56 million nodes before finding a solution.

Re-solving problem 898 and printing out the current partial solution at regular intervals makes the reason for its difficulty apparent. The first four instantiations made by the algorithm are $v_{31} = 1$,[3] $v_{34} = 1$, $v_{41} = 4$ and $v_4 = 2$. The subproblem created by these instantiations, consisting of the future variables and their remaining values, is insoluble. However, proving insolubility accounts for almost all of the consistency checks required to solve the overall problem. During the course of the search, partial solutions with 47 of the 50 variables instantiated are found: the 47th variable is always the same (v_{45}). Invariably, the only future variable at that point which conflicts with v_{45} is v_{38}; the instantiation of v_{45} evidently causes the domain of v_{38} to become empty, so that the algorithm has to backtrack. Hence, the insolubility of the subproblem is due to the constraint between v_{45} and v_{38}: a minimal relaxation of this constraint to allow an additional pair of values is sufficient to make the subproblem soluble, and to allow the algorithm to solve the overall problem very quickly. Without this modification, however, the algorithm eventually proves that the subproblem is insoluble and then tries an alternative instantiation for v_4, which immediately leads to a solution. The search considers only one possible instantiation of the first variable, v_{31}.

In searching the insoluble subproblem, the algorithm clearly shows thrashing behaviour, repeatedly backtracking to variables between v_4 and v_{45} and re-instantiating them. All of this work is wasted, since it is only by going back to the first four variables that any progress can be made. Since FC-FFdeg cannot recognise this, however, it is doomed to keep on thrashing until the subproblem has been exhaustively searched.

Problem 358 in the $\langle 50, 10, 0.1, 0.48\rangle$ ensemble is also an ehp for FC-FFdeg. The first four instantiations made by the algorithm are $v_5 = 1$, $v_{16} = 4$, $v_{22} = 9$ and $v_{37} = 4$. It eventually becomes clear that this set of assignments leads to an insoluble subproblem. However, proving insolubility takes more than 79 million consistency checks and 8 million nodes visited; the algorithm frequently finds partial solutions with 38 or more variables instantiated before detecting an infeasibility and backtracking. Once it has been proved that there is no solution to the subproblem, the alternative assignment of $v_{37} = 10$ is tried and leads almost immediately to a solution. The search considers only one possible instantiation of the first variable, v_5.

[3] i.e. variable 31 is assigned the value 1.

Problems 898 and 358 are typical of forward checking ehps. In every ehp that we have examined in detail, the first few assignments lead to a subproblem which has no solutions, and almost all the search effort is expended in proving this. Partial solutions involving most of the variables are found in the course of searching the subproblem, resulting in a great deal of backtracking. As with the two problems above, the solution eventually found has the first variable assigned its first value, so that there are very probably many solutions in other areas of the search space.

5.8.2 Extending lookahead

To examine the effects of increasing the level of lookahead on individual ehps, problems 898 and 358 described above were searched by MAC-FFdeg. To ensure a fair comparison, re-establishing arc consistency is suspended in the MAC algorithm until the insoluble subproblem has been created: for instance, if the first four instantiations lead to an insoluble subproblem, the MAC algorithm is constrained to behave like FC until depth 4 in the search tree, so that the same four instantiations will be made.

Re-searching problem 898 using MAC-FFdeg from depth 4, a solution is found after 6917 checks and 51 nodes. The algorithm clearly discovers that the subproblem is arc-inconsistent and performs a single backtrack before proceeding to the solution. On problem 358, MAC-FFdeg finds a solution after around 173,000 checks. In this case, the algorithm does perform some search on the inconsistent subproblem, but backtracks quickly out of it.

Occasionally, however, making the subproblem arc consistent is not sufficient to show that it is insoluble, and the cause is more complex:

Problem 4150 in the $\langle 50, 10, 0.1, 0.49 \rangle$ ensemble is an ehp for MAC-FFdeg, and is the search that stands out prominently in Figure 5.7. Nearly all of the 1.547 billion consistency checks and 38 million nodes visited are spent searching an insoluble subproblem created by the first seven assignments.

5.8.3 Additional backjumping

The effects of introducing backjumping capability to the algorithms which produce the 898, 358 and 4150 ehps was examined. Recall that the backjumping versions of the algorithms visit a subset of the nodes visited by the basic algorithm, and so are bound to encounter the same insoluble subproblems.

In the case of the forward checking ehps, problems 898 and 358, FC-CBJ-FFdeg does not find the insoluble subproblems exceptionally difficult. This is precisely because the backjumping can detect that the subproblems have no solutions much more quickly than chronological backtracking can. There is a vestige of the earlier difficulty with problem 358: FC-CBJ-FFdeg takes more than 3 million consistency checks to prove insolubility and this is one of the most difficult problems at that value of p_2. Problem 898, on the other hand, succumbs to CBJ very quickly: the subproblem is proved insoluble in only 150,000 consistency checks. Problem 4150, which is the MAC ehp, was searched by MAC-CBJ-FFdeg in just over 150,000 consistency checks.

5.8.4 Profile of an ehp

Adopting the adage that a picture tells many words, Figure 5.10 presents a profile (Section 4.5) of the consistency checking cost of four ‘interesting’ searches at each search depth. The searches examined are the three ehps discussed above (problems 898, 358 and 4150), plus the unusually hard insoluble $\langle 50, 10, 0.1, 0.63\rangle$ problem mentioned in Section 5.5.3. A linear scale is used on the y-axis of each plot to emphasise the imbalance of the cost distributions.

The profiles of the three ehps show fairly consistent patterns. Nearly all of the search effort is spread over a small subset of variables deep in the search tree. It is clear that the algorithms consistently fall just short of finding a solution before running into the same dead end. The plots in many ways drive home the enormity of the search spaces over which solutions are spread: the subproblem created by the first four instantiations, for instance, is only one of 10^4 possible; but within this, most of the searching is actually done after around twenty-five variables are instantiated; so this exceptionally long search process is actually spending most of its time in only a few out of a possible 10^{25} subproblems.

The behaviour of the unusually difficult insoluble search is interesting. It is clear that partial solutions of around twenty variables are consistently built up before a dead end is reached. From further analysis of the search trees of forward checking algorithms, presented in Chapter 7, we know that it is unusual for CSPs in this part of the insoluble region to have such a large subset of mutually consistent variables. Given the sparseness of the constraints, it may be the case that the constraint graph for this CSP is bipartite, dividing the problem into two subproblems which are connected by only a handful of constraints. The instantiation order of the algorithm might then mean it effectively searches one of the subproblems before considering the other. If the first subproblem is relatively under-constrained, then many permutations of instantiations for its variables would be possible. This is speculation, though, and more detailed analysis of this search might prove more enlightening.

The profiles of other measures of search cost (listed in Section 4.4) have also been considered, and the patterns for each are similar to those seen for consistency checking.

5.9 Conclusions

Exceptionally hard problems in the easy-soluble region occur when the first few variable instantiations lead to the creation of a subproblem which is insoluble: the insoluble subproblem causes the algorithm to thrash, repeatedly rediscovering the same inconsistency, deep in the search tree. However, once the algorithm escapes from the insoluble subproblem, a solution can be found almost immediately without further backtracking. These problems are not inherently difficult; a better algorithm will often find it trivial to prove the subproblem insoluble. This is quite different from the behaviour of tree search algorithms on the most difficult insoluble problems occurring in the phase transition region, where searches take a long time simply because every path through the search tree has to be followed and every one leads to a dead end.

All the ehps that we have seen in these experiments are themselves soluble; if an insoluble problem were to occur well below the phase transition, we expect that it would be extremely hard

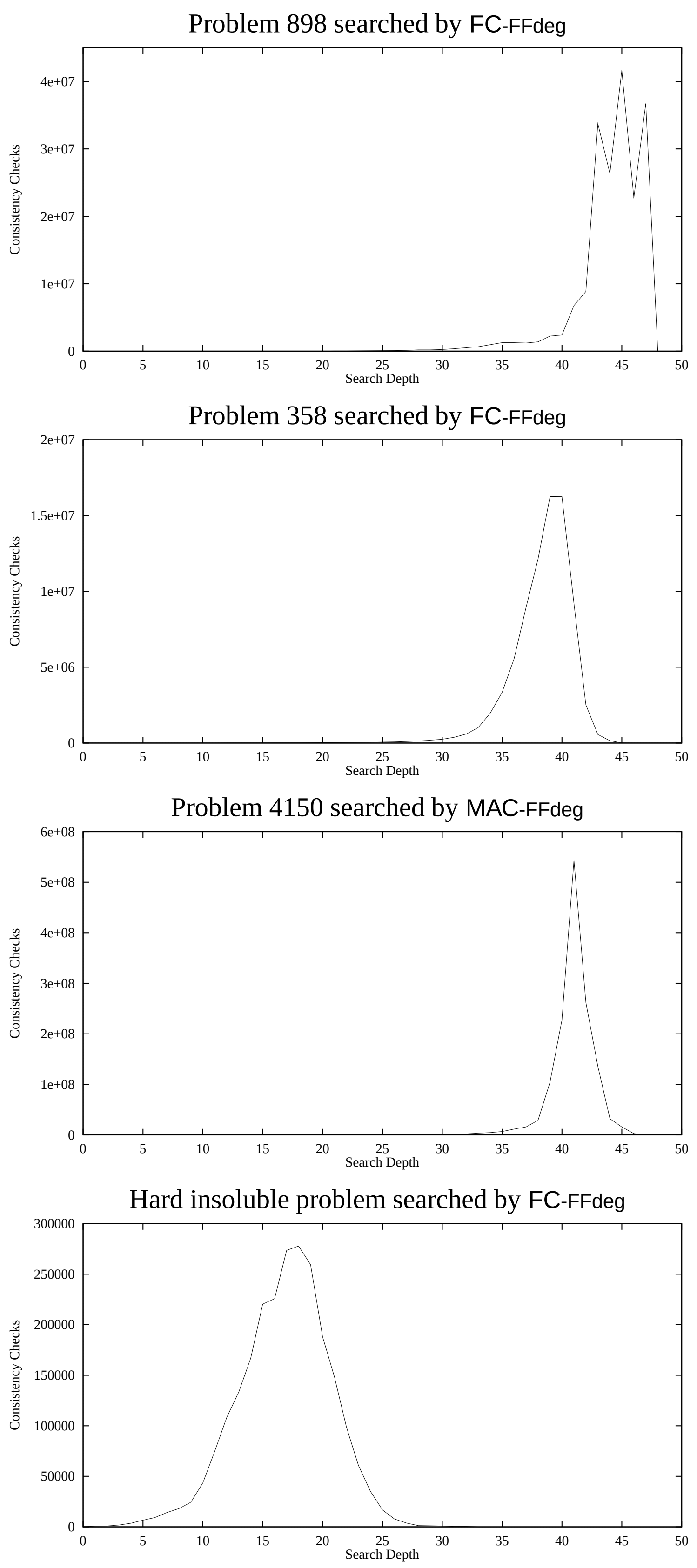


Figure 5.10: Profiles of four $\langle 50, 10, 0.1 \rangle$ exceptional searches.

to prove insoluble, just as finding all solutions is extremely hard, since a complete exploration of the search space is required. However, it seems that ehps of this type are exceptional even amongst ehps, if they occur at all.

It is remarkable that the qualitative behaviour of the median cost, over a wide range of values of p_2, is the same for the different search algorithms considered here, when the behaviour of the higher percentiles varies so markedly, sometimes tracking the median closely, and in other cases peaking much earlier and at a much higher level. We have shown that the occurrence of ehps in the easy-soluble region is highly algorithm dependent: we have found ehps using BT in populations with both high and low constraint density, whereas the other algorithms we have considered do not appear to suffer from ehps in problems with dense constraints. Furthermore, BT has ehps at values of p_2 for which almost all problems require no backtracking: when better algorithms have ehps, it is at values of p_2 much closer to the phase transition. Using the most complex algorithm we have considered (MAC-CBJ-FFdeg), we have found no ehps in the experiments reported here.

The algorithms studied can be seen as using two different strategies for avoiding thrashing in the insoluble subproblems that occur in ehps. One is to look ahead, in order to detect the inconsistency without searching the subproblem (FC and MAC); the other is to jump back when the inconsistency is met, rather than stepping back chronologically (e.g. CBJ). The strategies can be combined in the hybrid algorithms FC-CBJ and MAC-CBJ.

The first strategy is guaranteed to be able to detect the inconsistency in the subproblem, provided that it is of the right kind (node inconsistency for FC, arc inconsistency for MAC). Looking ahead can also cope to a certain extent with higher levels of inconsistency: for instance, FC is much better than BT even when the subproblem is arc rather than node inconsistent. However, there is a danger that the algorithm will then do a great deal of unnecessary backtracking, by stepping back chronologically to possibly irrelevant variables. Because they jump back to a variable involved in the conflict, CBJ-based algorithms reduce this danger. They also have some capability for handling a higher level of inconsistency. For instance, CBJ can handle node inconsistency very well; FC-CBJ can jump out of subproblems with the simpler forms of arc inconsistency; and presumably MAC-CBJ can sometimes deal with path inconsistencies, since it can sometimes solve very easily problems which MAC finds exceptionally hard. The MAC-CBJ algorithm has almost completely avoided ehps in our experiments, and shows the effectiveness of combining the two strategies.

An area which needs further investigation is the role of the fail-first heuristic in avoiding thrashing. It appears that when the constraint density is high, FC-FFdeg is virtually immune from ehps: even arc inconsistent subproblems do not cause thrashing. It is not clear how fail-first achieves this, especially as FC-FFdeg is not immune from ehps when the constraints are sparse, and the difficulties are then due to arc inconsistent subproblems.

It should be noted that our graphs exhibiting ehps show no sign of the double peak in the higher percentiles found by (Hogg and Williams 1994), except perhaps for BT over the $\langle 20, 10, 1.0 \rangle$ class (Figures 5.1 and 5.2). However, their experiments, on 3-colouring problems, used far larger samples than ours; they had between 10,000 and 1 million samples for each data point, and were thus able to see smooth behaviour in the 99.95 percentile. With much larger samples, binary

CSPs might well show a similar double peak. The recent theoretical study by (Smith and Grant 1997), described in Section 5.3.2, predicts the existence of a double peak, initially for the case of BT on CSPs with complete constraint graphs.

Finally, the whole subject of exceptionally hard problems is a source of much debate in the CSP community. Many researchers consider the issue to be either unimportant or irrelevant. An argument justifying the irrelevance of ehp behaviour is that, in practical terms, dealing with this behaviour is in fact extremely easy: a strategy of randomly re-starting search after a time bound is broken has been proposed by (Gomes *et al.* 1997); (Baker 1995) pushes the phenomenon beyond the horizon of experimental scutiny in his empirical studies by using a complex form of backjumping; and searching a CSP with a number of search 'agents', even if these agents use fairly naive algorithms, will reduce the likelihood of encountering an ehp to a negligible level ((Grant 1994) has conducted a study of 'multi-agent cooperative search' for the CSP).

However, there is a compelling case for at least understanding the causes of ehp behaviour in complete algorithms: all of the algorithms studied are essentially refinements of simple backtracking search; failure to understand how and why these simple algorithms can perform in the most unexpected ways places the development of more advanced complete methods in a precarious position. If the precise search conditions which lead to an ehp eventually become well understood, susceptible algorithms could be refined to incorporate this knowledge. They would then be able to detect dangerous situations and take very simple measures to change the nature of the search. Such an approach seems eminently sensible: increasing the sophistication of the algorithms in a 'brute force' approach will raise the average search cost on problems that are essentially very simple; whereas by accepting that such anomalies may occur for any complete search algorithm, and learning to recognise and deal with their occurrence, a simple and effective approach can be retained.

The investigation of ehps has given new insight into the behavior of CSP algorithms, including some which have been in use for a long time (BT and FC). It has also given new understanding of the phenomenon of thrashing behaviour: until its relationship with ehps was seen, thrashing was not recognized as a localized phenomenon, occurring in the easy-soluble region, which could be seen as part of the phase transition behavior of CSP algorithms.

5.10 Acknowledgements

This study of exceptionally hard problem behaviour has been conducted jointly with Barbara Smith. Some of the material presented here has appeared in (Smith and Grant 1995a) and (Smith and Grant 1995b). The recent study presented in (Smith and Grant 1997), particularly the theoretical analysis, is due mainly to Smith, and so is only reported here.

Chapter 6

Phase Transition Behaviour in Arc and Path Consistency

Phase transition behaviour has made a major impact on the empirical study of algorithms for NP-complete problems such as the CSP, leading to more rigourous experimentation and a better understanding of where algorithms perform well or badly. An interesting question that arises is whether phase transition methodology can be applied to polynomial classes of computational task. In this chapter we demonstrate that it can, showing that the polynomial tasks of establishing arc and path consistency in CSPs exhibit phase transition behaviour very much analogous to that associated with the NP-complete task of finding solutions to these problems.

Previous studies of arc consistency techniques (Bessière 1994; Borrett and Tsang 1995) suggest that the cost of consistency follows an easy-hard-easy pattern, and we show here that this pattern also exists for path consistency. These peaks in cost coincide with a transition between a region where arc or path consistency can be established in all problems, and achieving this is easy, and a region where attempting to enforce arc or path consistency fails for all problems, showing each to be insoluble, and achieving this is easy. The peak in average cost is observed to coincide with the point where around 50% of problems can be made consistent.

An empirical study applies the principles of phase transition research to examine where arc and path consistency processing is useful in removing domain values. This reveals that the effectiveness of establishing consistency rises and falls with the cost. Theoretical and empirical results are also presented which show the average cost of the AC3 arc consistency algorithm to be much lower than its worst-case time complexity suggests.

6.1 Related Work

Popular algorithms to establish arc and path consistency in CSPs are discussed in Section 2.5. Empirical studies of these techniques have revealed some interesting behaviour.

(Bessière 1994) introduced the AC6 algorithm, and conducted an empirical study to position it with respect to the AC3 and AC4 procedures. The three algorithms were applied to *n*-queens problems, the Zebra problem (Dechter 1990), and three classes of CSP: $\langle 20, 5, 0.3\rangle$, $\langle 12, 6, 0.5\rangle$ and $\langle 18, 9, p_1, p_2\rangle$. The algorithms were set to do just enough work to achieve consistency or prove insolubility. The analysis, done in terms of constraint checking cost, showed that AC3 and AC6 followed an easy-hard-easy pattern over the CSP classes. The cost of AC4 was consistently high in the under-constrained regions, due to the high basic overhead associated with setting up support counters. Although these patterns suggested some form of phase transition, Bessiére did not point this out.

(Borrett and Tsang 1995) conducted an empirical study, using AC6, of where arc consistency preprocessing is effective in terms of removing inconsistent values from the domains of variables. CSP classes of size $\langle 10, 10\rangle$ and $\langle 20, 10\rangle$ were examined, covering a range of constraint densities. Unlike Bessiére's study, AC6 was run to completion. Borrett and Tsang observed a coincidence between peaks in mean cost of the algorithm and the points where the algorithm ceases to be effective, but did not relate this to a form of phase transition behaviour. They did conclude that preprocessing of this kind is only generally effective on over-constrained problems.

6.2 Terminology

The terms *arc consistency (AC)*, *path consistency (PC)* and *k-consistency* are used as defined in (Tsang 1993) and Chapter 1.2.4. AC and PC are sometimes referred to as 2- and 3-consistency respectively. We talk of *establishing* arc consistency or path consistency in a problem as an attempt to enforce AC or PC respectively. If a problem is *arc-inconsistent* or *path-inconsistent* then it is not possible to establish AC or PC respectively; attempting to do so will show that the problem has no solution. If a domain element of a problem variable is *arc consistent* or *path consistent*, then that element will not be removed upon establishing AC or PC respectively. A domain element is *arc*- or *path-inconsistent* if it will be removed upon establishing AC or PC respectively. Each CSP is therefore in one of three states: the problem is already AC or PC, and so has no inconsistent domain elements; or the problem is *potentially* AC or PC, and establishing consistency will remove some inconsistent domain elements; or the problem is arc- or path-inconsistent, and attempting to establish consistency will prove insolubility. Finding a solution to a problem *proves* that it can be made *n*-consistent.

6.3 The Empirical Studies

The experiments reported here were performed using randomly-generated binary CSPs, generated according to the Model B method described in Chapter 3. Our empirical study of establishing arc consistency and path consistency in sets of CSPs uses the AC3 and PC2 algorithms respectively, introduced in Section 2.5, which are closely related. A useful property of these algorithms for our purposes is their relative simplicity: neither involves any complex initialisation stage that creates a high basic overhead for each run. Thus the cost of runs reflects the true complexity of establishing consistency.

We examine the effects of processing the CSPs in terms of the amount of variable domain pruning that occurs (permanent nogood values - Section 4.4), the number of consistency checking operations, the CPU time taken, and also the proportion of problems in each sample which are found to be inconsistent. As we are dealing with the practical application of AC and PC processing, we also terminate the algorithms upon a domain wipe-out (the removal of every possible value from a variable's domain), in which case we know that there are no solutions and that all other variable domains would be wiped out if the algorithm continued. This approach is similar to that of (Bessière 1994), but contrasts with that of (Borrett and Tsang 1995), who run the consistency algorithms to completion.

A number of phase transition experiments (Section 4.3) were undertaken, applying AC3 and PC2 to CSPs of size $\langle 20, 10 \rangle$. p_1 was varied in steps of 0.1 over the range $[0.2..1.0]$ in order to cover a full range of constraint densities. For each $\langle 20, 10, p_1 \rangle$ problem class, p_2 was varied in steps of 0.01 over the interval $[0.01..1.00]$. Ensembles of $1,000$ problems were generated at each $\langle 20, 10, p_1, p_2 \rangle$ point and processed with AC3 and PC2. The $\langle 20, 10, 0.1 \rangle$ problem class was omitted from the study: these sparse problems need only be made arc consistent in order to be solved, since connected constraint graphs in this class are all trees (Tsang 1993).

6.4 The 2-Consistency Phase Transition

The empirical study of AC processing by (Borrett and Tsang 1995) showed that its usefulness is restricted to problems that are over-constrained. They observed a transition between a region where constraints are very tight and, for all problems, processing eliminates the entire set of variable domains (proving the problems to be insoluble without need for search), and a region where constraints are loose and no pruning of domains occurs. In the intervening region, limited pruning of some variable domains occurs on average, which in principle should make search easier by reducing the potential search space[1]. Because the AC algorithm was run to completion, the curves of domain pruning were observed to rise to a plateau as constraint tightness was increased. However, if the algorithm is terminated as soon as a domain wipe-out occurs, the number of values pruned and the effort required to prove insolubility decrease as the constraint tightness increases.

Figure 6.1 shows the effects of AC3 processing on the sets of $\langle 20, 10, p_1, p_2 \rangle$ problems. These plots show the median curves of the consistency checking effort and the number of values pruned from variable domains, together with a curve showing the proportion of arc-inconsistent problems found at each $\langle 20, 10, p_1, p_2 \rangle$ ensemble. The three plots use the general constrainedness parameter, κ (Section 3.2), on the x-axes. The use of κ allows easy comparison of the location of the peaks in AC3 cost with those predicted for full search.

The curves of the median pruning effects for each problem class show that as constrainedness increases, the number of values removed rises from zero, increasing slowly at first but then rising sharply to a peak, and then drops down to m (the size of the variable domains). As these curves are rising, AC3 is finding an increasing number of arc-inconsistent values in median problems, although not enough to cause the complete wipe-out of any variable domain. At the peaks, domain

[1]It has, however, been demonstrated (Prosser 1994; Sabin and Freuder 1994) that algorithms employing dynamic variable ordering can occasionally perform more poorly as a result of the pruning effects of constraint propagation.

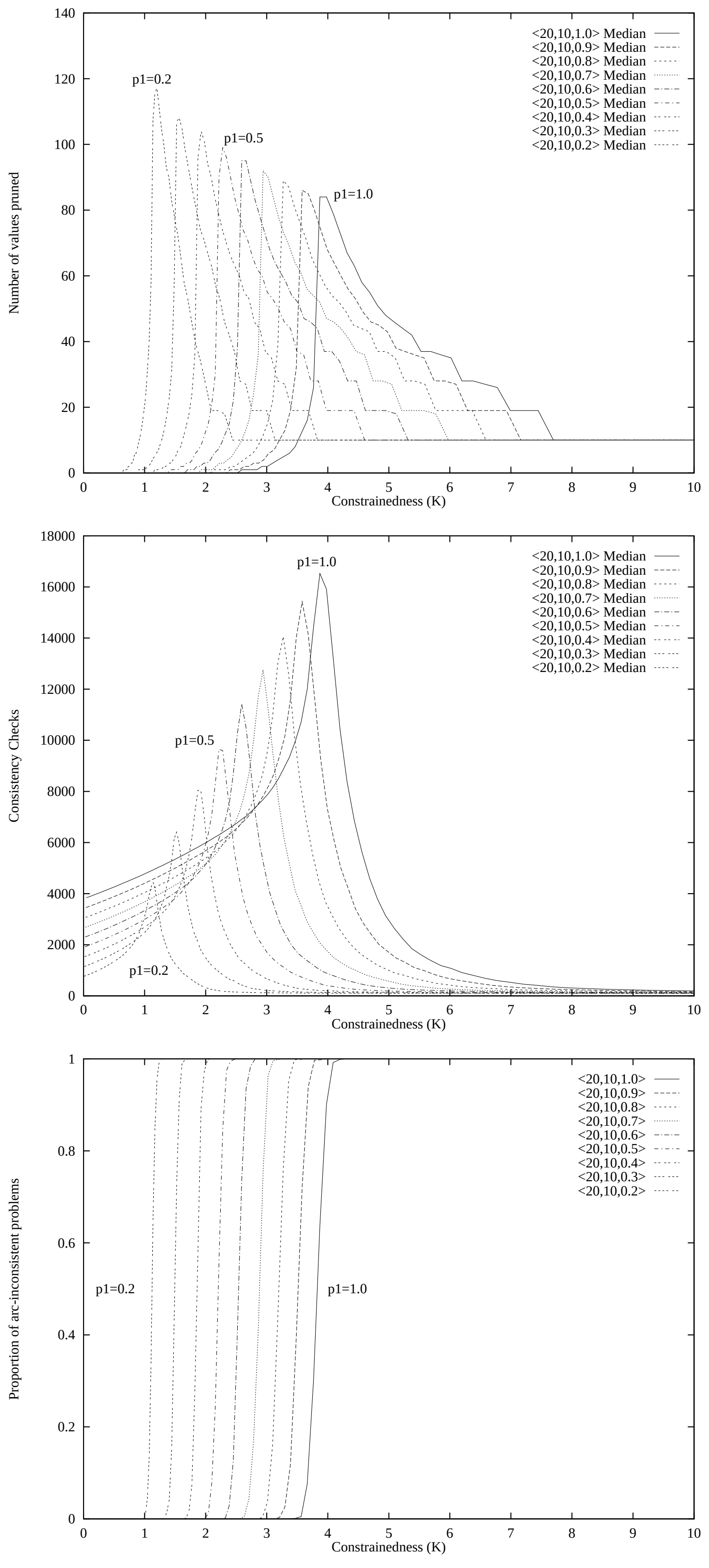


Figure 6.1: Effects of AC3 on $\langle 20, 10, p_1 \rangle$ CSPs.

Problems	κ	**peak median effort**	$\mathbf{p_{inc}}$
$\langle 20, 10, 0.2, 0.73 \rangle$	1.08	4455	0.640
$\langle 20, 10, 0.3, 0.68 \rangle$	1.41	6363	0.431
$\langle 20, 10, 0.4, 0.66 \rangle$	1.78	8246	0.582
$\langle 20, 10, 0.5, 0.64 \rangle$	2.11	9887	0.559
$\langle 20, 10, 0.6, 0.63 \rangle$	2.46	11500	0.735
$\langle 20, 10, 0.7, 0.61 \rangle$	2.72	12234	0.408
$\langle 20, 10, 0.8, 0.60 \rangle$	3.02	13741	0.440
$\langle 20, 10, 0.9, 0.60 \rangle$	3.40	15565	0.721
$\langle 20, 10, 1.0, 0.59 \rangle$	3.68	16869	0.638

Table 6.1: Properties of $\langle 20, 10, p_1, p_2 \rangle$ CSP ensembles at AC phase transition peaks.

wipe-outs occur for about 50% of problems, allowing the algorithm to terminate early. As the curves fall, the number of arc-inconsistent values for AC3 to find is still increasing, and hence domain wipe-outs are occurring more quickly, resulting in the earlier termination of the algorithm. When the curves fall to *m*, there are no arc consistent values in any variable domain, and so the algorithm immediately removes the *m* values in the domain of the first variable it examines, and terminates.

The consistency checking curves show a similar pattern, as might be expected, with a peak in cost coinciding with the peak in values pruned by the algorithm. The patterns of these peaks as constraint density varies is the reverse of that for pruning, however: although problems with high constraint density generally require fewer arc-inconsistencies to cause a domain wipe-out, as the effects of removing values tend to be greater, the propagation of these effects has a higher overhead. This results in a greater consistency checking effort on average than for less densely constrained problem classes.

From the curves showing the proportions of arc-inconsistent problems, it can be seen that AC3 is exhibiting behaviour exactly analogous to the phase transitions observed for complete search. The peaks in consistency checking and domain pruning occur between regions where all problems can be made arc consistent and AC3 quickly establishes this, and regions where all problems are arc-inconsistent and AC3 quickly proves this. In the intervening 'mushy' region, a proportion of problems are arc-inconsistent, and the peaks in checking and pruning approximately coincide with the point where this is true for 50% of problems.

In order to demonstrate this behaviour more clearly, Table 6.1 shows some properties of the $\langle 20, 10, p_1, p_2 \rangle$ ensembles for which the peaks in median consistency checking effort for AC3 occur. The values of constrainedness (κ), actual peak median effort, and the proportions of arc-inconsistent problems found (p_{inc}) are given for these sets of problems. From this data, it can be seen that the peaks in median cost do indeed occur when approximately half of the sampled problems are arc-inconsistent, and certainly in the transition region. The deviations from proportions close to 0.5 may be attributable to the coarseness of the experiments: the mushy region is very narrow and the changes in p_{inc} between consecutive values of p_2 can be very large; finer grained experiments could be expected to show the location of the 50% point more accurately.

Observation of the actual positions of the pruning curves agree with the findings reported in (Borrett and Tsang 1995), in that AC preprocessing has very little effect (in removing values) unless the problems are highly constrained. For all of the $\langle 20, 10, p_1 \rangle$ problem classes studied here, perhaps with the exception of $\langle 20, 10, 0.2 \rangle$, this means that AC3 processing only has an effect in the insoluble problem region; it is only for very sparsely constrained problems, where the phase transition between solubility and insolubility occurs at high levels of constrainedness, that AC processing has any effect on the hard problems in the mushy region. For the more densely constrained $\langle 20, 10, p_1 \rangle$ problem classes, the regions where any domain pruning occurs lie further into the insoluble problem region.

These findings suggest that establishing arc consistency as a preprocessing step before full search is only generally effective on over-constrained problems. It should be noted, however, that whether or not any values can be pruned depends upon the tightness of individual constraints, and if this is not uniform then AC preprocessing may be worthwhile at lower values of constrainedness than is implied by the plots shown. The curves in Figure 6.1 for median consistency checking effort also show that AC3 is a relatively 'cheap' algorithm in most cases. Therefore an AC3 preprocessing step will not usually add a significant overhead to overall search cost. In the next section we consider the cost of the AC3 algorithm in greater detail.

6.5 The Cost of AC3

(Mackworth and Freuder 1985) show AC3 to have a worst-case time complexity bounded from above by $\mathrm{O}(m^3 e)$ and from below by $\Omega(m^2 e)$, where e is the number of constraints. In Section 2.5, we note that (Wallace 1993) presents a series of arguments favouring the use of AC3 over AC4. Although AC4 has a better worst-case time complexity, Wallace demonstrates that the worst-case conditions for AC3 are rarely encountered.

We look at the cost of AC3 on the CSPs used in our experiments, considering the cost in the simplest cases, the cost on problems that are already arc consistent, and the cost at the AC phase transition peak where the hardest problems encountered by AC3 are found.

6.5.1 AC3 on the simplest problems

For the implementation of AC3 used here, there are two specific cases where the algorithm has a small and fixed cost.

Firstly, when $p_2 = 0$ (i.e. the constraints allow all pairs of values), AC3 requires $2me$ consistency checks: there are $2e$ arcs, and for an arc between variables v_i and v_j, all m values for v_i are supported by the first value tried for v_j.

Secondly, when the constraints are very tight (eg. where $p_2 = 1$ and the constraints forbid all pairs of values) the first arc considered, (v_i,v_j), may cause a domain wipe-out of variable v_i because no value of v_j supports any value of v_i. The algorithm will immediately terminate, having performed m^2 consistency checks.

6.5.2 AC3 on arc consistent problems

Figure 6.1 shows that there are regions of constraint tightness for each $\langle 20, 10, p_1 \rangle$ CSP class where many problems are already arc consistent. Although the effort spent here by AC3 is wasted, this cost appears to be small. We look here at how the average cost of the algorithm on AC problems grows with domain size, m.

It is possible to derive an expression for the expected consistency checking cost of AC3 for the special case of problems that are already AC. Consider the revision of an arc between two variables, (v_i, v_j), that is already consistent: each value $x_i \in domain_i$ will be supported by at least one value in $domain_j$. The probability that support for a particular x_i value will be found after c consistency checks is calculated as:

$$P(c) = \frac{p_2^{c-1}(1-p_2)}{1-p_2^m} \qquad \text{for } 1 \leq c \leq m$$

It should be noted that p_2 is treated here as a probability applied independently to each pair of values, so this is not quite accurate given that the experiments here use Model B rather than Model A problem generation. The expected number of consistency checks performed during revision of a single consistent (v_i, v_j) arc is then calculated as:

$$\begin{aligned} E(c) &= \sum_{c=1}^{m} cP(c) \\ &= \frac{1}{1-p_2} + m\left(\frac{1}{1-p_2^m} - 1\right) \end{aligned}$$

There are $2e$ arcs in a CSP with e constraints, so the total expected number of consistency checks performed by AC3 is calculated as:

$$\begin{aligned} 2me.E(c) &= 2me\left(\frac{1}{1-p_2} + m\left(\frac{1}{1-p_2^m} - 1\right)\right) \\ &\rightarrow \frac{2me}{1-p_2} \qquad \text{for large m} \end{aligned}$$

Problems that are already arc consistent tend to have low p_2, so even for small m, the $\left(\frac{1}{1-p_2^m} - 1\right)$ term becomes negligible (this has been verified for the CSPs used in the empirical studies).

Hence, the average case complexity of AC3 applied to problems that are already arc consistent is approximately linear in m. These results strengthen the case for the use of AC3 as a preprocessor to search: for the problems on which AC3 is ineffective in terms of removing values, proving arc consistency can be done in time linear in m.

6.5.3 AC3 on the hardest problems

To investigate whether the potentially cubic worst-case complexity of AC3 is encountered in practice, a further set of phase transition experiments were conducted, designed to measure the growth in the peak median cost as m is varied, with n and p_1 fixed. AC3 was applied to CSPs of size

Problems	κ	$\mathbf{p_{inc}}$	**min**	**max**	**mean**	**median**
$\langle 20, 5, 1.0, 0.35\rangle$	2.63	0.479	580	5900	3528	3404
$\langle 20, 10, 1.0, 0.59\rangle$	3.68	0.638	7371	28261	16869	16528
$\langle 20, 12, 1.0, 0.64\rangle$	3.89	0.510	12129	42264	25741	25042
$\langle 20, 15, 1.0, 0.70\rangle$	4.25	0.724	22620	73515	46434	45873
$\langle 20, 17, 1.0, 0.73\rangle$	4.39	0.688	31806	103242	65293	64957
$\langle 20, 20, 1.0, 0.77\rangle$	4.66	0.913	36359	152361	96301	96300
$\langle 20, 22, 1.0, 0.79\rangle$	4.79	0.958	55433	197838	120316	120110
$\langle 20, 25, 1.0, 0.81\rangle$	4.90	0.920	84893	274172	181515	180669
$\langle 20, 30, 1.0, 0.84\rangle$	5.12	0.975	122455	461194	281573	282203

Table 6.2: Data for the $\langle 20, m, 1.0, p_2\rangle$ CSP ensembles at AC phase transition peaks.

$\langle 20, m\rangle$, with p_1 varied in steps of 0.1 over the range $[0.2..1.0]$ as before, and with m taking the values $\{5, 10, 12, 15, 17, 20, 22, 25, 30\}$. In order to reduce the processing cost of these experiments, p_2 was varied in steps of 0.01 for all $m \geq 10$, rather than in steps of $1/m^2$. Ensembles of 1,000 CSPs were generated and processed at every $\langle 20, m, p_1, p_2\rangle$ point. For each $\langle 20, m, p_1\rangle$ problem class, the peak median consistency checking cost was recorded.

Table 6.2 shows data for the eight $\langle 20, m, 1.0, p_2\rangle$ ensembles for which the peak median consistency checking cost occurs at each value of m. Although p_{inc} rises further away from 0.5 as m increases, this can perhaps be attributed to the coarseness of the p_2 steps combined with the decreasing width of the phase transition regions. The remaining fields in the table show the minimum, maximum, mean and median of consistency checking cost for each ensemble.

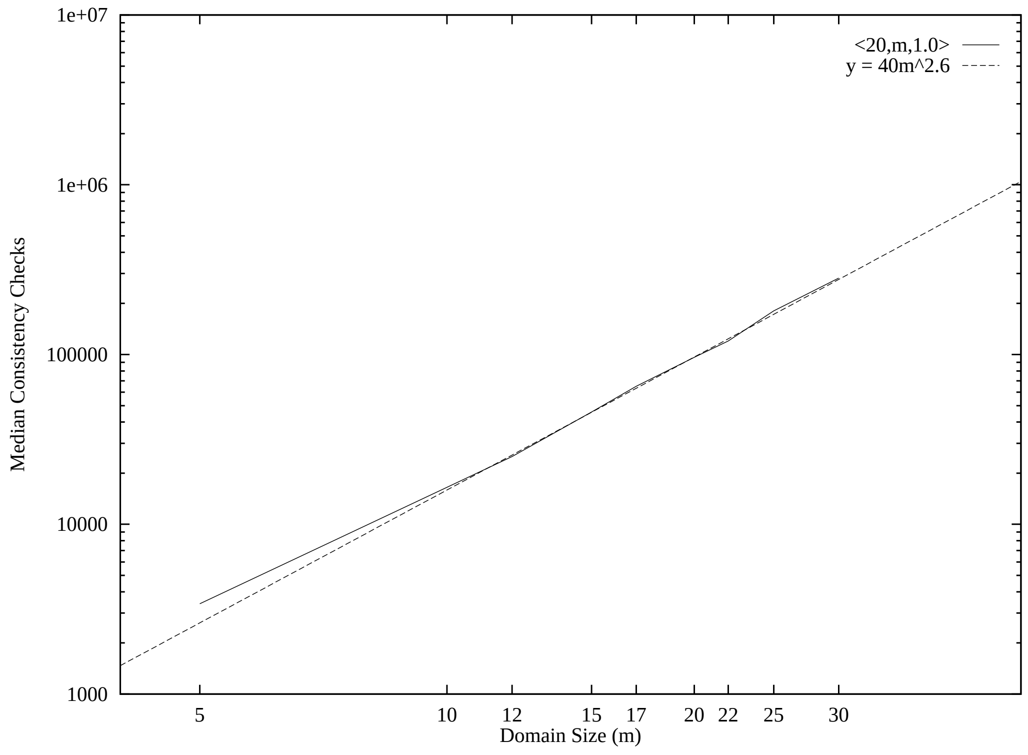
In order to determine the rate of AC3 cost growth as m increases, we attempted to fit the data in Table 6.2 to a curve of the form am^b. If the data can be accurately described using this model, we would expect to find a rate of growth somewhere between quadratic and cubic. Plotting the cost against m, using a logarithmic scale on both axes, we should expect a curve of the form am^b to appear linear (because $\log y = a + b \log m$).

Figure 6.2 presents two such plots, with the upper plot using the peak median consistency checks and the lower plot using the peak maximum values. It can be seen that the curve is close to linear over the eight points between $m = 10$ and $m = 30$. The curves $y = 40m^{2.6}$ and $y = 65m^{2.6}$ have been added to the respective plots: these close fitting curves were selected by performing linear regression using the method of least squares. This appears to confirm that the cost growth can be modelled by am^b, and that its rate is considerably less than cubic, even for the most expensive problems.

These results further support the use of AC3 as a preprocessor, and also support (Wallace 1993), who shows that the worst-case conditions for AC3 rarely arise.

6.6 The 3-Consistency Phase Transition

Figure 6.3 shows the effects of PC2 processing on the sets of $\langle 20, 10, p_1, p_2\rangle$ ensembles, plotted against constrainedness. These plots show the median curves of CPU time and values pruned from variable domains, together with a curve showing the proportion of path-inconsistent prob-



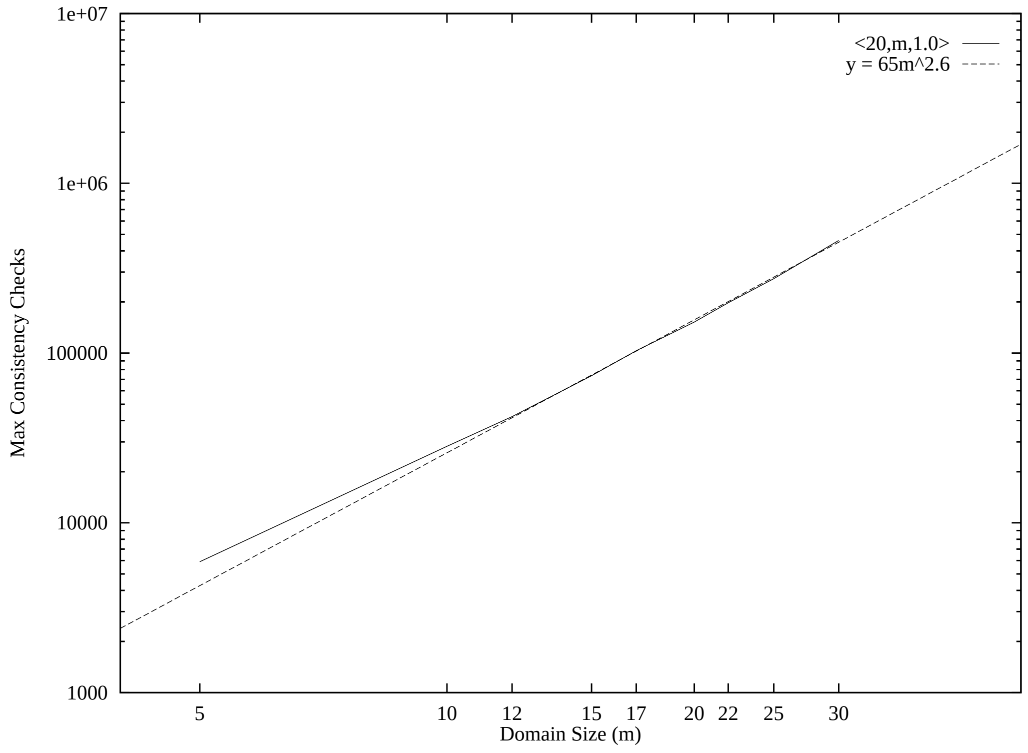


Figure 6.2: Peaks of median and maximum costs of AC3 against *m*.

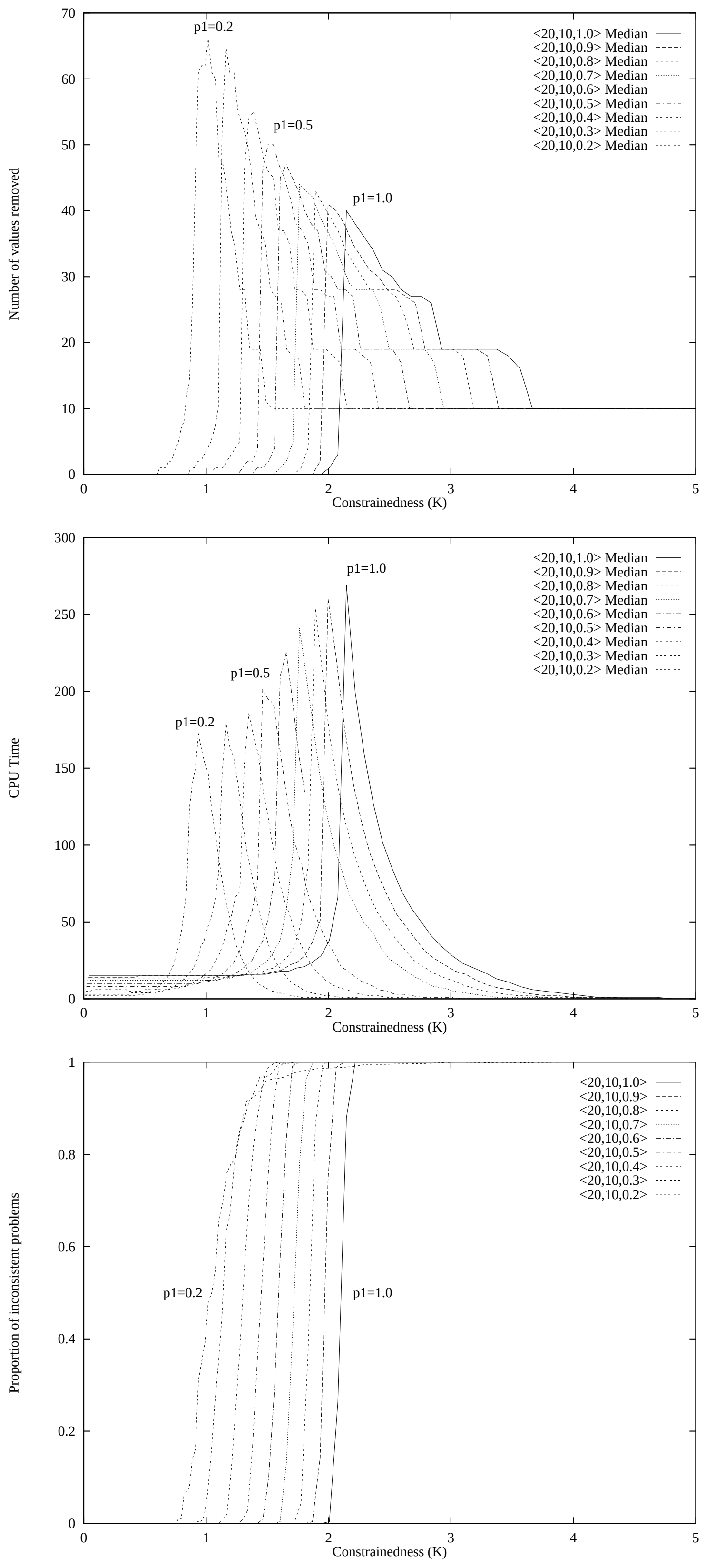


Figure 6.3: Effects of PC2 on $\langle 20, 10, p_1 \rangle$ CSPs.

Problems	κ	**med. cost**	$\mathbf{p_{inc}}$
$\langle 20, 10, 0.2, 0.69\rangle$	0.97	172s	0.412
$\langle 20, 10, 0.3, 0.59\rangle$	1.10	180s	0.628
$\langle 20, 10, 0.4, 0.53\rangle$	1.25	185s	0.548
$\langle 20, 10, 0.5, 0.49\rangle$	1.39	201s	0.548
$\langle 20, 10, 0.6, 0.47\rangle$	1.57	225s	0.576
$\langle 20, 10, 0.7, 0.44\rangle$	1.67	241s	0.779
$\langle 20, 10, 0.8, 0.42\rangle$	1.80	254s	0.832
$\langle 20, 10, 0.9, 0.40\rangle$	1.90	260s	0.748
$\langle 20, 10, 1.0, 0.39\rangle$	2.04	269s	0.838

Table 6.3: Properties of problems at PC cost peaks.

lems found at each $\langle 20, 10, p_1, p_2\rangle$. It is worth pointing out, incidentally, that path consistency preprocessing is usually performed in order to tighten the constraints in problems, and not just to remove values from domains.

The general pattern of these curves is similar to those for AC3, with the medians of both domain pruning and CPU time rising to a peak as constrainedness increases, before falling again. With PC2, these curves rise as an increasing number of path-inconsistent values are being found, although not enough to cause the complete wipe-out of any variable domain. At the peaks, domain wipe-outs occur for about 50% of problems, allowing the algorithm to terminate early.

The peaks coincide with a transition between regions where all problems can be made path consistent and PC2 quickly establishes this, and regions where all problems are path-inconsistent and PC2 quickly proves this. Table 6.3 shows some properties of the $\langle 20, 10, p_1, p_2\rangle$ problems at which the peaks in median CPU time occur, in a similar style to Table 6.1. As for AC3, the PC2 peaks coincide fairly closely with the 50% inconsistency points, although the coarseness of the p_2 steps limits accuracy in the more densely constrained problem classes.

Observation of the actual positions of the pruning curves shows that the PC phase transitions for each problem class occur at lower values of constrainedness than those for AC. This is predictable, as the set of path-inconsistent problems subsumes the set of arc-inconsistent problems (i.e. a problem which is path-inconsistent must also be arc-inconsistent). For problems to be path consistent, a lower level of constrainedness is required than that which allows the same problems to be arc consistent, and so AC can occur in more highly constrained problems than PC can. The behaviour of the $\langle 20, 10, 0.2\rangle$ transition is interesting, in that it is widely spread out: this pattern is similar to the phase transitions that occur between solubility and insolubility for many sparsely constrained problem classes (Prosser 1996). The location of the PC mushy region for $\langle 20, 10, 0.2\rangle$ problems also straddles the point at which $\kappa = 1$. Although $\kappa = 1$ is not an accurate estimate for the phase transition between solubility and insolubility for such a sparsely constrained problem class (Gent *et al.* 1996b), the PC phase transition would still appear to be close to that for *n*-consistency. This suggests that many of these problems which are path consistent are in fact *n*-consistent.

The curves in Figures 6.1 and 6.3 of the median amount of domain pruning show that the actual numbers of values removed by PC2 on average are considerably smaller than for AC3 on the same problems. This may appear slightly counter-intuitive at first, since values which are arc-inconsistent are also path-inconsistent. However, the PC phase transitions occur at lower values of constrainedness than those for AC: where large numbers of values are being removed by AC3, these problems are already path-inconsistent, and so PC2 will find domain wipe-outs and terminate more quickly, thus pruning fewer values. The propagated effects of removing path-inconsistent values are also greater than those for removing arc-inconsistent values, meaning that fewer inconsistencies are needed to induce a domain wipe-out.

As a preprocessing step, the curves of CPU time show that our implementation of PC2 is clearly not feasible. The peak average cost for some problem classes is over 200 seconds: the comparative figures for AC3 are around 0.3s, while an average forward checking search at the phase transition costs around 9s. It should be noted that the PC4 algorithm for path consistency (Section 2.5) has a significantly lower worst-case time complexity than PC2, although this is still cubic in both n and m.

6.7 Interpretation of the AC and PC Phase Transitions

In presenting the phase transition behaviour associated with establishing arc consistency and path consistency in CSPs, we have made the analogy with the phase transition behaviour observed when finding a *single* solution to the same problems. This connection may at first glance appear to be incorrect: when establishing AC we must make *all* arcs in the problem consistent; similarly when establishing PC we must make *all* paths of length 2 consistent; so should the analogy made with n-consistency not be made with respect to finding *all* solutions to the problem? However, if we consider establishing a certain level of consistency in a problem as *performing the minimal amount of work necessary to prove that the problem can possess such consistency*, then the validity of the analogy becomes clear: when attempting to establish k-consistency in a problem, where $k < n$, all paths of length k must be made consistent, and the effects of the removal of inconsistent assignments must be propagated around the other paths; but when attempting to establish n-consistency, only one path of length n exists (the variable ordering is immaterial), and the discovery of one consistent set of labels for the variables in the form of a solution is sufficient to *prove* that the problem can be made n-consistent.

In order to relate the phase transition behaviour of establishing arc consistency and path consistency with that of establishing the existence of n-consistency, the ensembles $\langle 20, 10, 0.5\rangle$ problems that were processed by AC3 and PC2 were also searched by a complete search algorithm. The algorithm selected was forward checking (FC), using the FFdeg dynamic variable ordering heuristic introduced in Section 5.5.2, although any other complete search method could have been used. Figure 6.4 shows the phase transition behaviour of establishing AC using AC3, PC using PC2, and n-consistency using FC-FFdeg. The curves of median consistency checking effort for each algorithm are shown on the respective plots, together with superimposed curves showing the proportions of inconsistent problems as constrainedness varies. These plots highlight the similarities in the phase transition behaviour that occurs for each level of consistency. Starting with the plot for FC-FFdeg, showing the phase transition between solubility and insolubility, it can

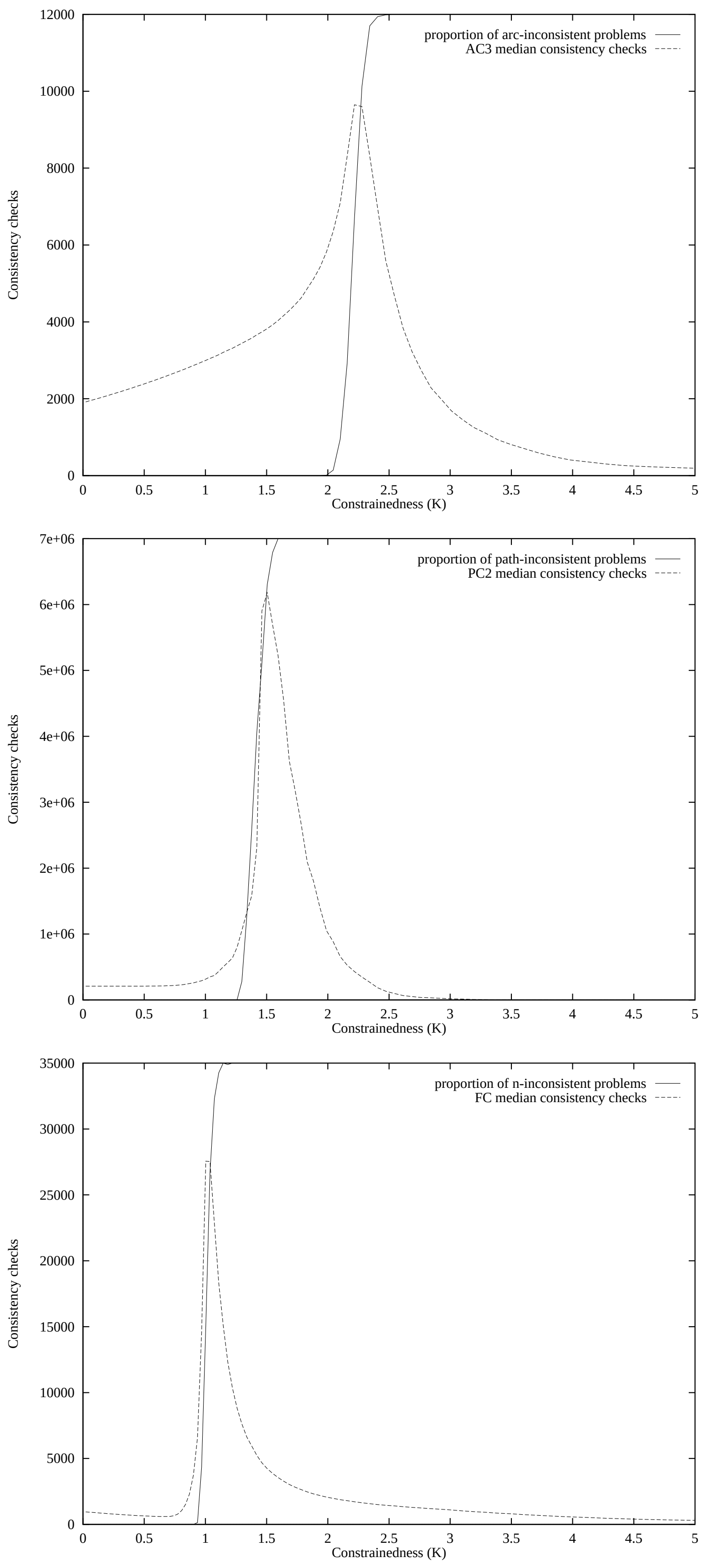


Figure 6.4: Phase transitions in establishing arc consistency, path consistency and finding a solution. The 'inconsistency' curves are superimposed and have a true range of $[0..1]$.

be seen that this phase transition is firmly centered around the value $\kappa = 1$ and the mushy region (where inconsistency levels lie between 0 and 1) is narrow, covering a range of κ of around 0.2. The PC phase transition occurs relatively close to that for *n*-consistency, although the respective mushy regions do not overlap, and for PC this region is slightly wider, covering a range of κ slightly greater than 0.25. The phase transition region for AC occurs well into the region of over-constrained problems, with a mushy region covering a range of κ of around 0.5.

From the patterns of behaviour observed in Figure 6.4, it seems reasonable to conclude that there must be a further series of phase transitions associated with establishing increasing levels of consistency, from that among subsets of 4 variables to that among subsets of $n-1$ variables. Each of these phase transitions must occur at decreasing values of constrainedness, between that for establishing PC and that for establishing *n*-consistency, forming a *hierarchy* of phase transition behaviour, and becoming increasingly narrow as the level of consistency increases. It seems likely that for many CSP problem classes, the location of the phase transitions for establishing high levels of consistency will all but converge with that for establishing *n*-consistency, as problems containing a higher degree of partial consistency are more likely to be soluble.

It might be noted that node consistency (consistency in all unary constraints on variables) is not included in this hierarchy. For the binary CSPs generated as described in Chapter 3, problems have node consistency by construction, and so a study of establishing NC is not feasible. Achieving NC is a trivial task for most conceivable types of constraint problem (with a time complexity that is $O(mn)$), and so it seems unlikely that establishing NC in problems will show any interesting phase transition behaviour.

6.8 Discussion

We have demonstrated phase transition behaviour analogous to the well-established phase transition between soluble and insoluble problem regions, occurring for two problems of polynomial complexity: establishing arc consistency and path consistency in CSPs. In the case of establishing AC using AC3, the peak in cost coincides with the transition between a region where AC already exists or can be established for all problems, and a region where all problems are arc-inconsistent. The transition appears to coincide with a change in order of the average cost of the algorithm, and it has been shown in Section 6.5 that this change in order is from a cost that is approximately linear in *m* in the case of problems that are already AC, to a cost that has been observed to be between quadratic and cubic in *m* at the AC phase transition peak. A similar pattern appears to be evident in the case of establishing PC using PC2, although the cost levels are considerably greater than for AC3. However, insufficient evidence is presented in the empirical study reported here to be able to establish the growth in average cost of PC2, and further study of the behaviour of this algorithm is required.

In addition to reporting the phase transition behaviour found in establishing AC and PC, we have also considered AC3 and PC2 in terms of their usefulness as preprocessors, to be used prior to full search. Areas where establishing AC or PC in problems has no effect in reducing CSP variable domains have been observed, as have regions where doing so is effective in this respect.

However, we have derived an expression for the expected cost of AC3 on these problems which shows it to be approximately linear in m. It has also been shown that for the hard problems at the AC phase transition peak, both average cost and maximum cost grow at a rate that is significantly slower than cubic in m. These results supports the claims by (Wallace 1993) that AC3 has an average-case cost significantly lower than its worst-case complexity analysis suggests.

That AC3 appears to be cheap to perform in nearly all cases also adds support to the use of the algorithm as a preprocessing step prior to search. The results of PC processing in Section 6.6 appear to rule out this method as a practical preprocessor, given the very high average cost of the PC2 algorithm. More efficient PC algorithms (Mohr and Henderson 1986) still have high time complexities, and so a change of algorithm is also unlikely to make PC preprocessing practical.

Understanding of the phase transition behaviour of consistency techniques is useful in understanding the phase transition behaviour of search techniques which perform consistency maintenance. An example of such a technique is a MAC algorithm, which maintains arc consistency. The phase transition behaviour of MAC is explored in Chapter 7.

It should be remembered that the empirical studies are based on random problems, generated according to the model described in Chapter 3. Problems with more structured constraint graphs, varying domain sizes and/or individual constraint tightnesses may well behave differently when attempting to establish consistency. Quite how such changes might affect the performance of the consistency algorithms remains to be studied.

The results of this study of establishing AC and PC in problems may be relevant to the notion of a ‘constraint gap’, proposed by (Gent and Walsh 1996b) as the conditions arising in sparsely constrained problem classes that give rise to the occasional occurrence of exceptionally hard problems (ehps). They show that in SAT problems, ehps tend to occur in problem regions where the propagation of ‘goods’ and ‘nogoods’ (i.e. values which can be shown to be valid in any solution, and those which can be shown to form part of no solution) is ineffective. If such a constraint gap exists for CSPs, the AC and PC data (concerning propagation of nogoods) together with data concerning the propagation of goods (such as CSP reduction operators (Rossi 1995)) may provide empirical evidence for it.

6.9 Subsequent Studies

The study of phase transition behaviour presented in this chapter was first published in (Grant and Smith 1996a) and (Grant and Smith 1996b). This work has since been followed up by (Gent *et al.* 1997b), who have devised a new constrainedness parameter, κ_{ac}. They show that the AC phase transitions for many classes of CSP occur at the same value of this parameter. They also use finite-size scaling techniques (Gent *et al.* 1995) to propose an alternative model for the growth in cost of AC3. This model suggests that the growth in average cost at the AC phase transition is cubic in m. Gent *et al.* go on to test κ_{ac} as the basis for a constraint ordering heuristic to increase the efficiency of AC3’s processing. The proposed heuristic processes first the constraint whose propagation will minimise κ_{ac} in the rest of the problem, and it is shown empirically that this produces more efficient AC3 processing than a number of other constraint ordering strategies.

Chapter 7

The Phase Transition Behaviour of Maintaining Arc Consistency

The study of establishing arc consistency presented in Chapter 6 shows that it tends to be useful, in terms of removing inconsistent values, only on CSPs in the over-constrained problem region. It is known, however, that re-establishing arc consistency in the subproblem of uninstantiated variables during search can be a worthwhile exercise on CSPs in all problem regions. In this chapter, we examine two search algorithms which do this: Maintaining Arc Consistency, MAC, and its hybrid combination with Conflict-Directed Backjumping, MAC-CBJ.

The behaviour of MAC and MAC-CBJ is studied with respect to the phase transition behaviour of binary CSPs, enabling the algorithms to be applied to many problems covering a range of sizes, topologies and expected difficulties. MAC performs a higher degree of lookahead than Forward Checking (FC), which maintains only node consistency in the future subproblem. In order to study the effects of using this increased lookahead capability, we also compare the performance of MAC with that of FC, and of MAC-CBJ with the equivalent hybrid FC-CBJ, over the same sets of problems.

It is shown that compared to FC, MAC develops a far smaller search tree, enables backtrack-free search over a wider range of problems, and greatly reduces the occurrence of exceptionally hard problems (ehps). The performance of MAC also scales much better than that of FC as the number of problem variables increases. The addition of CBJ to MAC further reduces the incidence of ehps to produce stable performance in almost all populations of problems, although its effect on the average search cost is not significant.

7.1 Related Work

Maintaining arc consistency during search is a popular technique employed by the constraint programming community, and is used by many constraint solving tools such as ILOG Solver (Puget 1994). However, its application by the constraint satisfaction community has until very recently been passed over in favour of the lesser level of lookahead provided by FC. A likely explanation for this stems from the previous lack of understanding of the great difference that exists between the behaviour of CSPs with high and low constraint density. More specifically, many of the problems that have until recently formed the test bed for assessing new algorithm performance (in particular *n*-queens and similar problems) have constraint graphs that are cliques (i.e. each variable (queen) is constrained by every other): for these problems we would expect the lookahead cost of MAC to be very high.

Much of FC's prominence in recent years can be attributed to the study of lookahead search techniques presented by (Haralick and Elliott 1980). They studied the basic backtracking algorithm, BT, plus algorithms with four levels of lookahead: backmarking[1], BM; Forward Checking, FC; Partial Look Ahead; and Full Look Ahead (which maintains arc consistency). All of these algorithms are discussed in Section 2.1. Haralick and Elliott applied these algorithms to the task of finding all solutions to the *n*-queens problem and ensembles of five random $\langle n, n, 1, 0.35\rangle$ CSPs, for the values $n = 4, 5, 6, 7, 8, 9, 10$. The results showed that FC performed the fewest constraint checks on these problems. The number of nodes visited during search was also examined, and it was shown that greater levels of lookahead resulted in smaller search trees. Looking at profiles of nodes visited at each search depth, they showed that the lookahead algorithms do most of their work at shallow levels of the search tree, carefully building up partial solutions, while non-lookahead algorithms spend most of their effort deep in the search tree, trying to complete the partial solutions that they build up quickly. Haralick and Elliott also examined the effect of dynamic variable ordering on lookahead search, showing that this can improve efficiency further. The role of dynamic variable ordering with FC and MAC is studied in detail in Chapter 8.

A further study of lookahead techniques for the CSP was reported by (Nadel 1989). A number of procedures to establish partial levels of arc consistency were proposed, and it was shown how these can be incorporated into a backtracking search framework to produce algorithms with various levels of lookahead. A number of such algorithms were tested empirically: backmarking, Forward Checking, Partial Look Ahead and Full Look Ahead, as used by (Haralick and Elliott 1980); and a new algorithm called 'Really Full Look Ahead', which makes the whole constraint network arc consistent at every stage of search (rather than just the subproblem of future variables). The set of test problems used were the *n*-queens problems, and the 'confused' *n*-queens variant, in which all queens must attack each other. Once again, the algorithms performing the least amount of checking in the future subproblems, FC and BM, proved to be the cheapest. Nadel concluded that although higher lookahead reduces the number of nodes visited, the extra work required at each node more than cancels out any savings. These results further cemented the position of FC as the standard backtracking search strategy used by the constraint satisfaction community.

The use of MAC on CSPs covering a range of problem topologies was eventually reported by (Sabin and Freuder 1994), who presented an implementation based on the AC4 arc consistency algorithm. They applied this algorithm to small samples of very sparsely-constrained CSPs with 50 variables and domain sizes of 8. Compared with Forward Checking, and with AC preprocessing followed by Forward Checking, MAC took considerably less time to search these problems.

[1]Backmarking can only loosely be said to perform lookahead: see Section 2.1.

Sabin and Freuder's study of MAC prompted the investigation of the AC3-based MAC and MAC-CBJ algorithms presented here, some results of which were presented in (Grant and Smith 1995) and (Grant and Smith 1996c). Subsequently, (Bessière and Régin 1996) presented a detailed empirical study of an AC7-based MAC, first described in (Bessière *et al.* 1995). Testing the algorithm on fifty-variable CSPs, they showed that MAC performed much better than FC-CBJ on hard and sparsely-constrained ensembles. For constraint densities below approximately $\frac{1}{3}$, MAC is the better algorithm, while above this level FC-CBJ becomes cheaper. Bessière and Régin also suggested that CBJ is an unnecessary addition to a search algorithm which combines a good lookahead technique with dynamic variable ordering.

Sabin and Freuder have recently revisited MAC. In (Sabin and Freuder 1997) they propose an improved version of MAC which takes advantage of constraint graph topology to instantiate a 'cyclic cutset' of variables. Removing this subset of variables from the CSP reduces the constraint graph of the remaining subproblem to a tree. We have already noted in Chapter 6 that such problems need only be made arc consistent in order to be solved (Tsang 1993). The new algorithm is termed MACE, for MAC Extended. AC7-based versions of MAC and MACE are compared empirically over small ensembles of $\langle 20, 20, p_1, p_2 \rangle$ and $\langle 40, 10, p_1, p_2 \rangle$ CSPs drawn from phase transition regions. MACE out-performs MAC over all but the densely constrained problem sets.

7.2 Structure of this Study

This chapter presents a study of the performance of the MAC and MAC-CBJ algorithms over a large range of CSP sizes and topologies, showing where the algorithms perform well and badly, and how their performance compares with the FC and FC-CBJ algorithms employing a lesser level of lookahead. The relative performance of the algorithms in terms of the incidence and magnitude of exceptionally hard problems (ehps) is also investigated, following on from the studies reported in Chapter 5.

The following section recaps the work on exceptionally hard problems presented in Chapter 5. Features of the AC3-based MAC and MAC-CBJ algorithms are then considered, followed by a description of the main empirical studies that were undertaken. Examination of the performance of MAC at the population level shows that the extra lookahead produces large regions of backtrack-free search, and that while ehps can still occur, their incidence is greatly reduced from that of FC. The cost of MAC is also observed to scale much better than that of FC as the number of problem variables increases. It is then shown that while the addition of CBJ to MAC gives little improvement in performance in the average case, the incidence of ehps is further reduced to the point where only one clear instance is found among several million candidate sparse CSPs. A study of the internal behaviour of the algorithms shows that the search tree of MAC is considerably smaller than that of FC, and we conclude by discussing the further issues raised.

7.3 Revisiting the Exceptionally Hard Problems

Although we are interested naturally in studying MAC and MAC-CBJ in terms of achieving improved general performance over other algorithms on certain problems, a further motivation arises from the performance of the algorithms in reducing the incidence of exceptionally hard problems, examined initially in Chapter 5.

Study of ehp behaviour for a number of algorithms showed that the unusually high search cost of these problems can be attributed to the early creation of an insoluble subproblem which the algorithm cannot detect as such without conducting an exhaustive search. Algorithms employing a lookahead style of forward move reduce the chances of being unable to detect a subproblem's insolubility by checking for some level of consistency in it. We conjectured that the ehps which remain for lookahead algorithms contain subproblems with a level of inconsistency that is beyond that which is tested for. That is, ehps for FC contain arc-inconsistent subproblems, while those for MAC contain path-inconsistent subproblems. The addition of backjumping capability appeared to assist the lookahead algorithms in detecting these higher levels of inconsistency, effecting a further reduction in ehp behaviour. Another significant factor was the use of dynamic variable ordering, which appeared to eliminate ehp behaviour completely from densely-constrained CSP classes. The most stable performance seen for all of the algorithms studied in Chapter 5 was that exhibited by MAC-CBJ-FFdeg, which combined the most advanced forms of lookahead and backjumping with dynamic variable ordering.

The empirical studies reported in this chapter extend the 'macroscopic' view of the ehp behaviour exhibited by MAC and MAC-CBJ and also FC and FC-CBJ, by covering a broader range of CSP classes than those examined in Chapter 5.

7.4 The Algorithms

It was noted in Section 7.1 that this study was prompted by that reported in (Sabin and Freuder 1994). The MAC and MAC-CBJ algorithms presented by (Prosser 1995) are based on AC3, whereas the MAC algorithm reported by Sabin and Freuder is based on AC4. Although AC4 is arguably a more efficient algorithm than AC3 (this is discussed in Chapter 6), the *effect* of establishing arc consistency is invariant of the technique used. Prosser also notes that the cost of the AC3-based algorithms may be measured in consistency checks, giving an important advantage in investigating the effects of increasing algorithm lookahead. Sabin and Freuder's AC4 based MAC performs all consistency checking during the initialisation of the AC4 support counters. During search, all the work of maintaining arc consistency is done by reference to these counters and not to the original constraints. Thus Sabin and Freuder measured the performance of their MAC implementation in terms of CPU time rather than consistency checks.

The initial stage of a algorithm which maintains arc consistency is naturally to establish a state of arc consistency in the problem. (Borrett and Tsang 1995) discuss the usefulness of AC preprocessing, while Chapter 6 shows that establishing arc consistency exhibits its own phase transition behaviour, and study the average cost of using AC3.

In all of the experiments reported here, MAC, MAC-CBJ, FC and FC-CBJ use a dynamic variable ordering heuristic based on the 'fail-first' principle: the first variable to be instantiated is that which is most constrained, and thereafter, the next variable to be instantiated is that with fewest remaining values in its domain. This is the FFdeg heuristic, used in Chapter 5 and studied further in Chapter 8. For brevity we omit the -FFdeg tag when naming the algorithms during the rest of this chapter. The implementations of FC-CBJ and MAC-CBJ, discussed in Chapter 2, are such that the set of nodes visited is a subset of those visited by FC and MAC respectively. Both base algorithms consider the uninstantiated variables in the same order as their CBJ hybrid at each forward move, and it is only on backward moves that the algorithms differ. FC-CBJ and MAC-CBJ therefore always find solutions in the same order and take no more consistency checks than FC and MAC respectively. As a side effect, the CBJ hybrids cannot find a problem exceptionally hard unless their base algorithms also do.

7.5 The Empirical Studies

The objective of the empirical studies undertaken was to establish the behaviour of the MAC and MAC-CBJ algorithms over large populations of problems of varying *size* and *topology*. The problems studied are binary CSPs, generated according to the Model B random generation method presented in Chapter 3. For these CSPs, defined by $\langle n, m, p_1, p_2 \rangle$, the effects of varying problem topology may be studied by varying p_1 whilst holding n and m constant. In order to independently study the effects of increasing problem size, the problem topology must be maintained, and this is achieved by holding the average degree (introduced in Section 4.1.1) constant whilst varying n and m. It is not sufficient to fix p_1 whilst varying problem size: γ increases linearly with the number of variables if constraint density is constant, so for instance as n is doubled with constant p_1, γ also approximately doubles. Note that the predicted critical value of constraint tightness, $\hat{p}_{2crit}$ (Section 3.2), is identical for CSP classes having the same values of γ and m. Thus, the phase transitions of these classes are expected to coincide.

7.5.1 The main experiments

To show how the behaviour of MAC and MAC-CBJ varies as problem size, n, and average degree, γ, are varied, two main groups of phase transition experiments were conducted. The first held n at 30 while varying p_1 in steps of 0.1 over the range $[0.1..1.0]$, and the second held γ at approximately 4.9 (allowing for some rounding errors since the number of constraints must be an integer) while varying n in steps of 5 over the range $[20..70]$. Variable domain size m was held at 10 in all experiments. The decision to use $\gamma \approx 4.9$ for the second set of experiments was taken in order to give problems that are fairly sparsely constrained, and for which at least one member class of problems ($\langle 50, 10, 0.1 \rangle$, studied in Chapter 5) is known to produce a number of ehp instances using FC and MAC.

The sets of $\langle 30, 10, p_1 \rangle$ and $\langle n, 10, \gamma \approx 4.9 \rangle$ CSP classes studied are listed in Tables 7.1 and 7.2 respectively, along with additional background information. The tables include the theoretical critical value of constraint tightness, $\hat{p}_{2crit}$ introduced in Section 3.2, at which average search

Problems	γ	$\hat{\mathbf{p}}_{\mathbf{2crit}}$	**$\mathbf{p_2}$ range**	**samples per $\mathbf{p_2}$**	**Algorithms**
$\langle 30, 10, 0.1, p_2 \rangle$	2.934	0.80	0.50–0.90	10,000	All
$\langle 30, 10, 0.2, p_2 \rangle$	5.80	0.55	0.20–0.70	10,000	All
$\langle 30, 10, 0.3, p_2 \rangle$	8.73	0.41	0.20–0.50	10,000	All
$\langle 30, 10, 0.4, p_2 \rangle$	11.60	0.33	0.15–0.40	10,000	All
$\langle 30, 10, 0.5, p_2 \rangle$	14.53	0.27	0.10–0.40	1,000	All
$\langle 30, 10, 0.6, p_2 \rangle$	17.40	0.23	0.10–0.30	1,000	FC/FC-CBJ
$\langle 30, 10, 0.7, p_2 \rangle$	20.33	0.20	0.10–0.30	1,000	FC/FC-CBJ
$\langle 30, 10, 0.8, p_2 \rangle$	23.20	0.18	0.05–0.30	1,000	FC/FC-CBJ
$\langle 30, 10, 0.9, p_2 \rangle$	26.13	0.16	0.05–0.30	1,000	FC/FC-CBJ
$\langle 30, 10, 1.0, p_2 \rangle$	29.00	0.15	0.01–0.30	1,000	All

Table 7.1: The set of $\langle 30, 10, p_1 \rangle$ problem classes studied.

Problems	γ	$\hat{\mathbf{p}}_{\mathbf{2crit}}$	**$\mathbf{p_2}$ range**	**samples per $\mathbf{p_2}$**	**Algorithms**
$\langle 20, 10, 0.2579, p_2 \rangle$	4.90	0.61	0.3–0.8	10,000	All
$\langle 25, 10, 0.2067, p_2 \rangle$	4.96	0.60	0.3–0.8	10,000	All
$\langle 30, 10, 0.1701, p_2 \rangle$	4.93	0.61	0.3–0.8	10,000	All
$\langle 35, 10, 0.1445, p_2 \rangle$	4.91	0.61	0.3–0.8	10,000	All
$\langle 40, 10, 0.1256, p_2 \rangle$	4.90	0.61	0.3–0.8	10,000	All
$\langle 45, 10, 0.1121, p_2 \rangle$	4.93	0.61	0.3–0.8	10,000	All
$\langle 50, 10, 0.1000, p_2 \rangle$	4.92	0.61	0.3–0.8	10,000	All
$\langle 60, 10, 0.0836, p_2 \rangle$	4.93	0.61	0.3–0.8	1,000	All
$\langle 70, 10, 0.0712, p_2 \rangle$	4.91	0.61	0.3–0.8	1,000	All

Table 7.2: The set of $\langle n, 10, \gamma \approx 4.9 \rangle$ problem classes studied.

effort is expected to be maximal.

The phase transition experiments on each $\langle n, m, p_1 \rangle$ CSP class varied p_2 in steps of 0.01. Ensembles of $10,000$ problems were generated at every $\langle n, m, p_1, p_2 \rangle$ point for sparsely constrained classes (where ehps are likely), while smaller sample sizes of $1,000$ problems were used for more densely constrained classes. CSPs generated were searched with MAC, MAC-CBJ, FC and FC-CBJ. The use of large sample sizes, particularly on the sparsely-constrained classes, was intended to increase the probability of seeing the extremes of search behaviour for each of the algorithms.

During the running of the experiments shown in Table 7.1, it became clear that the overhead of MAC becomes very high for densely constrained problems. For this reason MAC and MAC-CBJ were not run on the classes of problems where $p_1 = \{0.6, 0.7, 0.8, 0.9\}$. (Borrett and Tsang 1995) and Chapter 6 show that the range of constraint tightness in which arc-inconsistent values are found in problems shrinks as constraint density increases. Meanwhile, the cost of AC3 grows linearly with the number of constraints. It is this combination of increasing cost and decreasing effect that provides a likely explanation for the high cost of MAC on densely constrained problems.

Figure 7.1 shows the observed satisfiability curves, as p_2 is varied, for each of the $\langle 30, 10, p_1 \rangle$ problem classes and $\langle n, 10, \gamma \approx 4.9 \rangle$ problem classes studied. The left graph plots satisfiability against constrainedness, κ, in order to line up the phase transitions around one location. The right hand graph simply plots against constraint tightness, since the phase transitions for problems

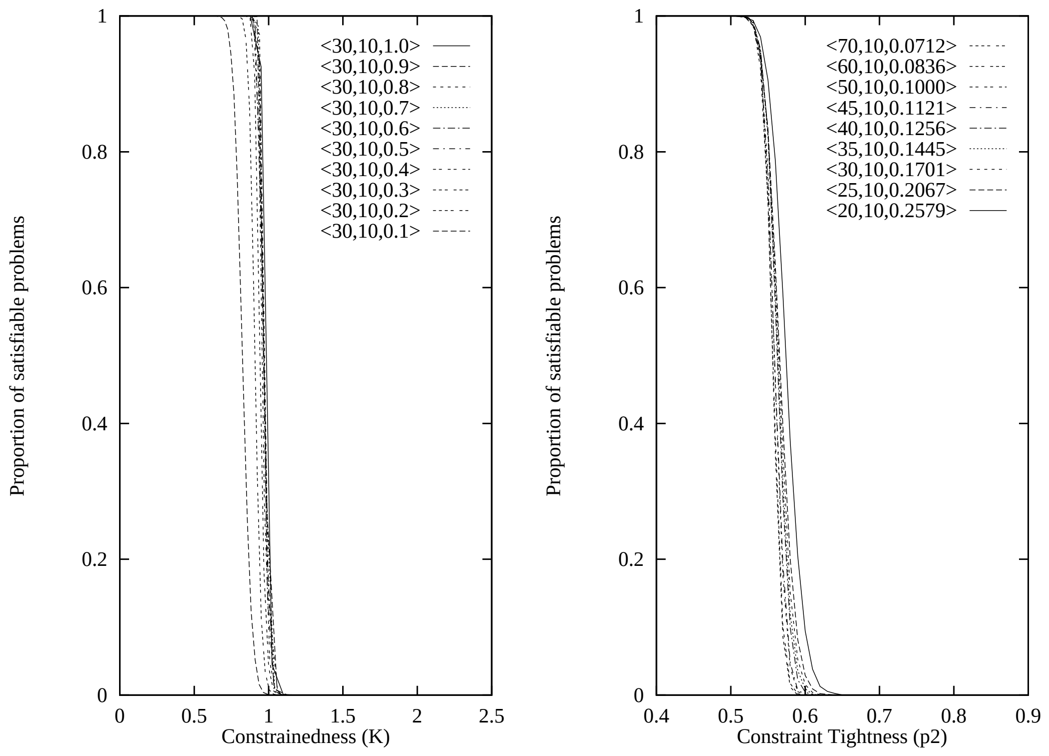


Figure 7.1: Observed satisfiability curves for the CSP classes listed in Tables 7.1 and 7.2.

with similar average degree occur at similar values of p_2. As expected (Section 3.2), the phase transitions of the smaller or more sparsely constrained classes plotted in Figure 7.1 do not line up exactly with those for the larger or more densely constrained classes.

The complete set of experiments outlined here represent a total investment of around 1,200 days (28,000 hours) of cpu time.

7.6 Macroscopic Performance of MAC

In analysing the performance of MAC at the population level, the behaviour of the algorithm is studied both in isolation, and in comparison with FC over the same populations of problems. To investigate the general behaviour of the algorithm, we plot the median search costs in terms of consistency checks. However, ehps by definition represent extreme behaviour in a population of problems, and so for some CSP classes we plot the median and higher percentiles of cost, as discussed in Section 4.5.

7.6.1 General and extreme behaviour

Figure 7.2 shows the median behaviour of MAC, in terms of consistency checks, on a selection of the $n = 30$ problem classes, while Figure 7.3 shows the median and higher cost percentiles for three of these problem classes. All graphs plot cost against p_2. We can see the clear differences in phase transition behaviour as the constraint density, p_1, of the problems (and the average degree, γ) decreases. The behaviour of the $\langle 30, 10, 1.0\rangle$ problem class is typical of problems with high

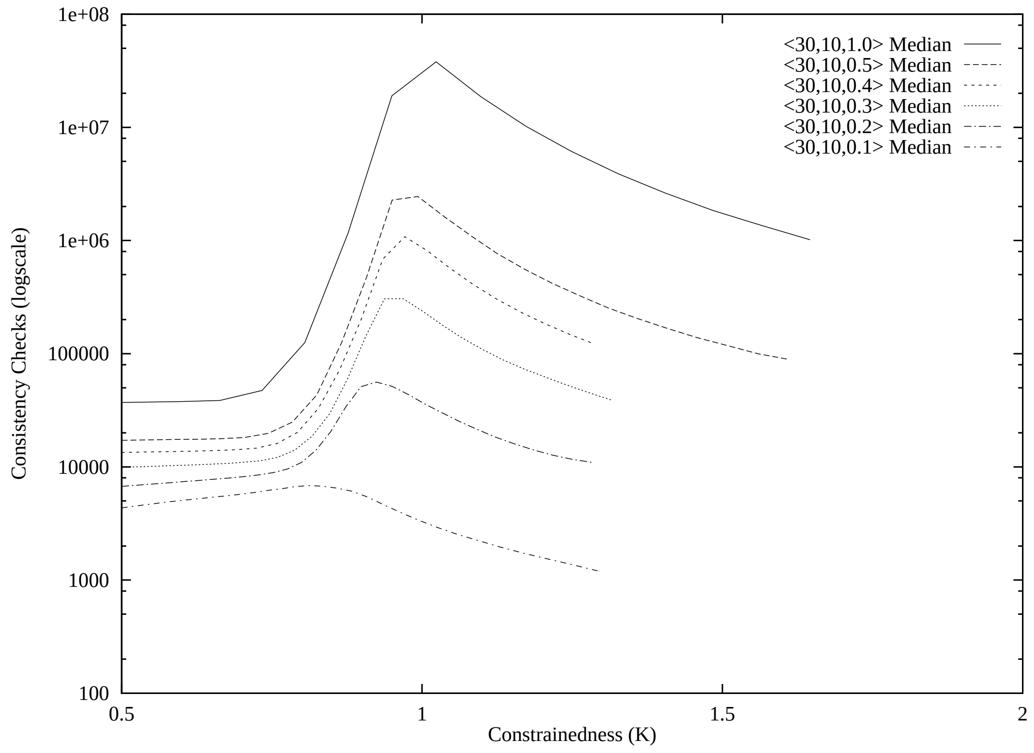


Figure 7.2: Median cost of MAC over $n = 30$ series, in terms of consistency checks.

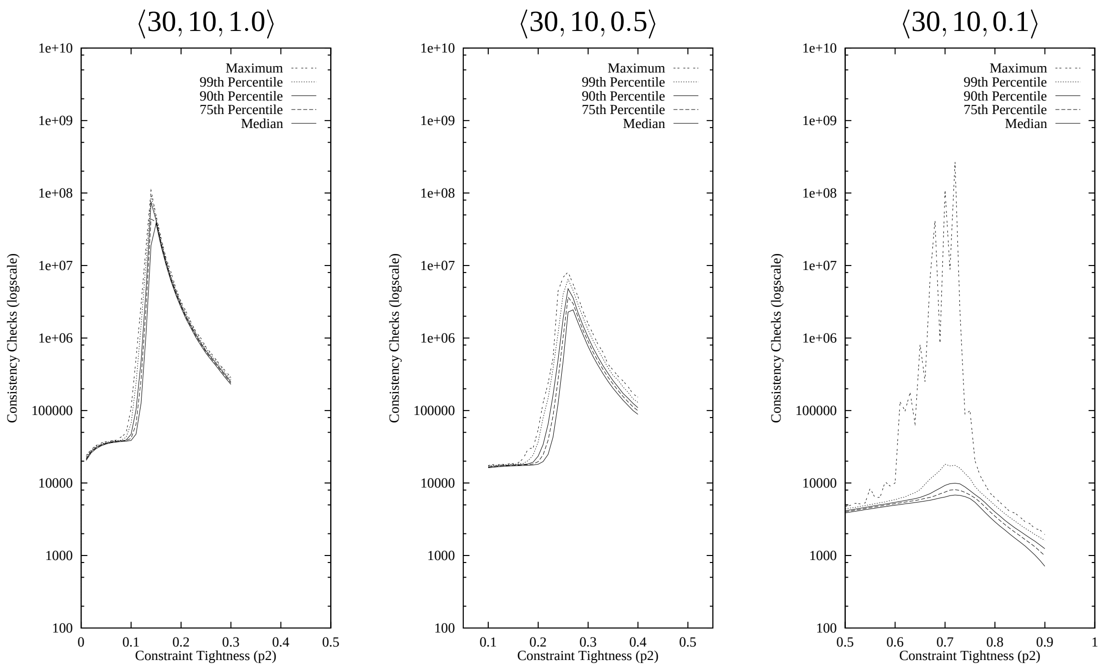


Figure 7.3: Ranges of consistency checking cost for MAC on three $n = 30$ CSP classes.

constraint density: in this case, $p_1 = 1$, i.e. the constraint graph is complete. In terms of median search cost, a sharp transition occurs as constraint tightness increases, from the region where problems have very many solutions and are easy to solve, through the crossover point, and into the insoluble region where the cost gradually decreases. Looking at the maximum cost, a similarly smooth curve is seen, with values not greatly above those of the median. In particular, in the easy-soluble region where ehps may occur, even the most difficult problem at each value of p_2 is still easy, compared with those in the mushy region.

As problems become more sparsely constrained, the peaks in median search cost become less sharply defined, although for each problem class there is still a clear phase transition peak. However, the maximum cost becomes highly erratic for the sparse problems. At $p_1 = 0.1$ we begin to observe instances of exceptionally hard problems in the easy-soluble problem regions that are much more difficult (by at least an order of magnitude) than 99% of the other problems occurring in the sample at the same constraint tightness, and much more difficult (again by at least an order of magnitude) than 99% of the sample problems in the phase transition. As in Chapter 5, no ehps were found in the easy-soluble regions that are insoluble problems: we continue to conjecture that in the case of CSPs such problems must be exceptionally rare, even among ehps (although it should be noted that this is not necessarily true for other classes of problem such as SAT (Gent and Walsh 1994a)).

Figure 7.4 shows the median behaviour of MAC, in terms of consistency checks, on a selection of the $\gamma \approx 4.9$ problem classes, while Figure 7.5 shows the median and higher percentiles for four of these problem classes[2]. As expected, the median consistency checking effort increases steadily as n increases, and the phase transition regions are in approximately the same location, as indicated by the satisfiability curves of Figure 7.1. As γ is relatively low for these problem classes we would expect to see some ehp activity, and indeed clear ehps are visible from the plots of Figure 7.5. However, the incidence of ehp behaviour is low, particularly in comparison with FC on the same problems, as is shown later in Section 7.6.3. There are indications in these plots that the incidence and magnitude of ehps increases as n is increased; this pattern is more evident in experiments with algorithms that are more susceptible to ehps, such as FC. The single most difficult problem encountered by MAC in the whole study is still that which occurs at $\langle 50, 10, 0.1, 0.49 \rangle$, which was analysed in detail in Chapter 5. Solving this problem takes MAC over 1.547 billion consistency checks – over five orders of magnitude greater than the median at that point, and over 100 times more difficult than the hardest phase transition problem.

7.6.2 Backtrack-free search

Looking at the performance of MAC in terms of search nodes visited, it can be seen that there are sets of problems for which no backtracking during search is required, on average. When a problem has a solution, a backtrack-free search will successfully instantiate the first value tried for each variable, resulting in n nodes visited. For problems with no solution, we can identify regions where no backtracking is required on average, where *either*: AC preprocessing wipes out an entire variable domain, in which case no search is required and so no nodes are visited;

[2] It should be noted that the $\langle 50, 10, \gamma \approx 4.9 \rangle$ plot also appears in Figure 5.7 of Chapter 5.

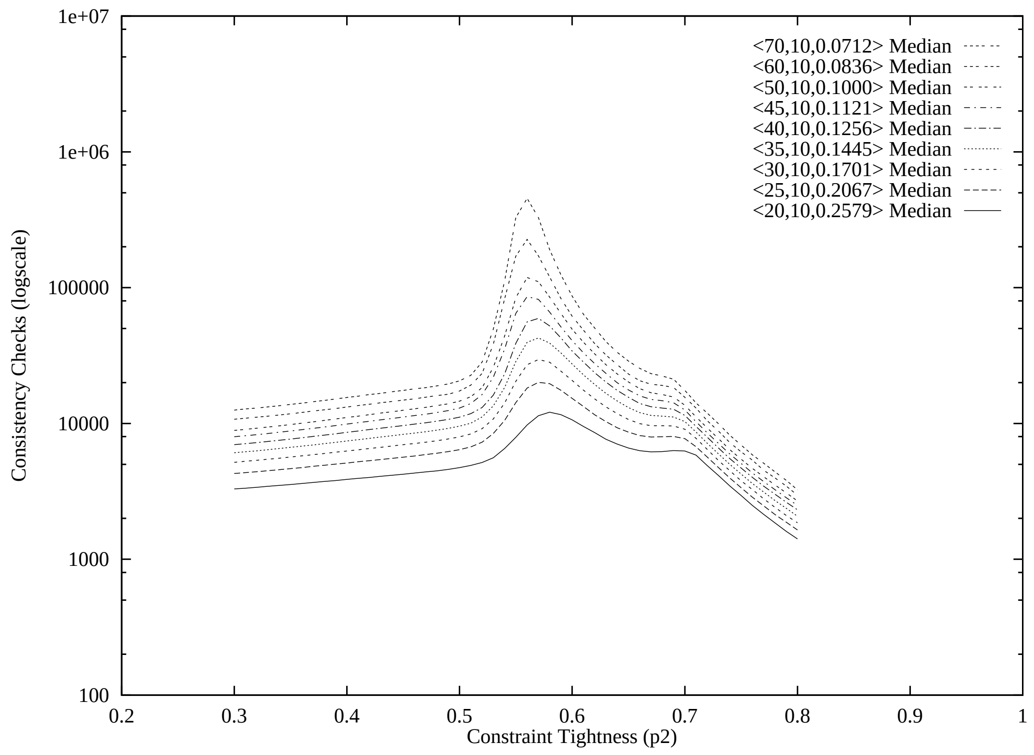


Figure 7.4: Median cost of MAC over $\gamma \approx 4.9$ series, in terms of consistency checks.

or preprocessing removes some inconsistent values and then every instantiation tried for the first variable is inconsistent, in which case k nodes are visited, where k is the size of the reduced domain of the first variable; *or* preprocessing has no effect in removing values, but the MAC lookahead shows every value of the first variable's domain to be inconsistent, in which case m nodes are visited. It should be noted that the definition of backtrack-free search on insoluble problems is not strictly correct, as the undoing of each trial instantiation made for this variable is technically a backward search move.

Figure 7.6 shows the median behaviour of MAC in terms of nodes visited, for each of the $n = 30$ and $\gamma \approx 4.9$ problem classes to which it was applied. As for Figure 7.1, the graph for the $n = 30$ classes plots cost against constrainedness, while that for the $\gamma \approx 4.9$ classes plots against constraint tightness. This places the phase transition peaks on top of each other, illustrating the differences in average behaviour more clearly. From these plots, it can be seen that for each problem class there are parts of the easy-soluble problem region for which at least half of all searches are backtrack-free. On the other side of the phase transition the median curves again fall, to m or less, and for some of the problem classes with low γ (and where a sufficiently wide range of p_2 has been covered) we can see that for much of the insoluble region, half of all searches are backtrack-free.

An interesting observation from the left hand set of graphs in Figure 7.6 is that the regions where backtracking search occurs on average are 'squeezed' as γ falls. This shows the opposite trend to that of consistency checks, where the peaks coinciding with the phase transitions spread out as γ falls. In fact, the region of backtracking search is all but squeezed out of existence for the $\langle 30, 10, 0.1\rangle$ problem class. There is a barely perceptible rise just above n in the median as

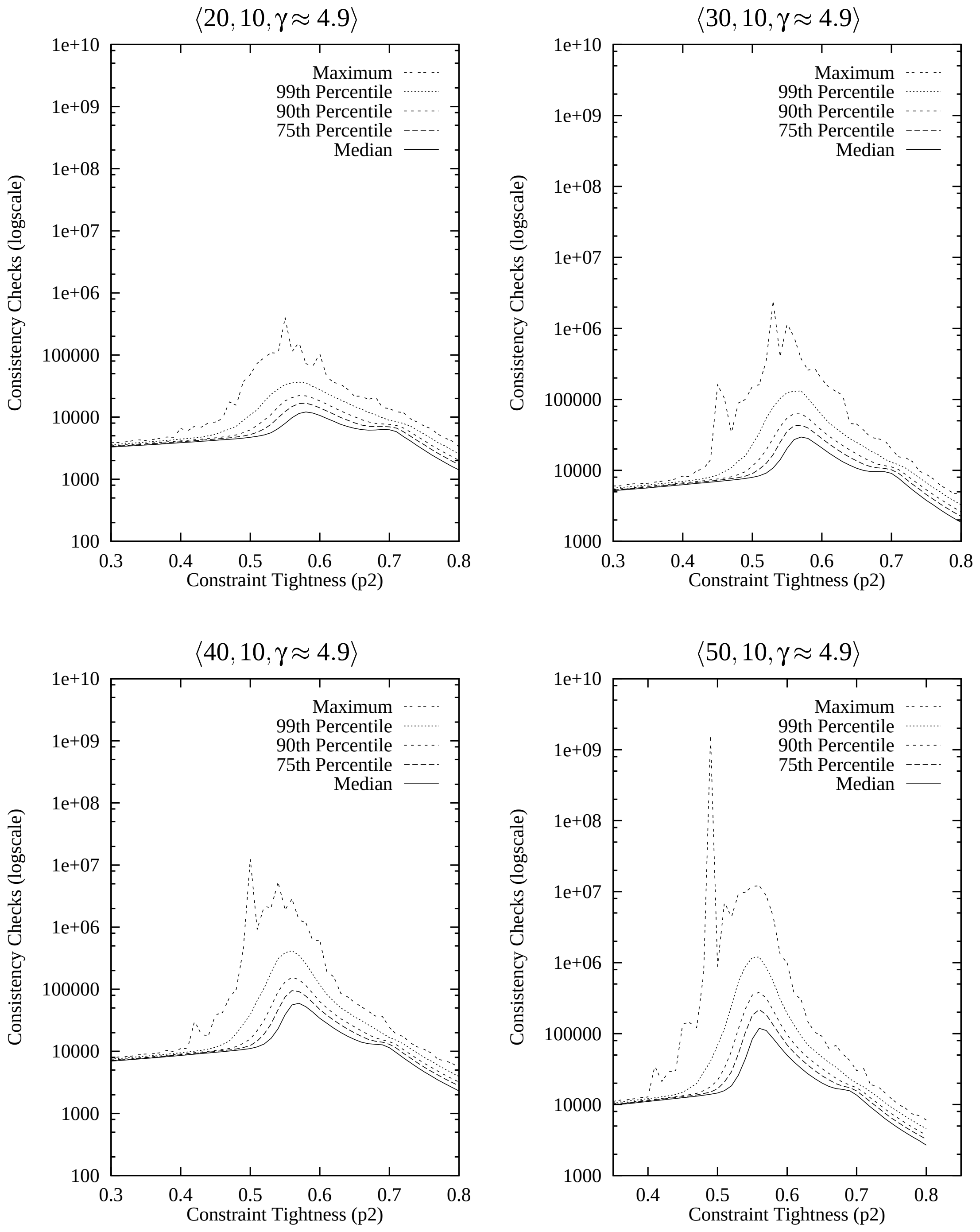
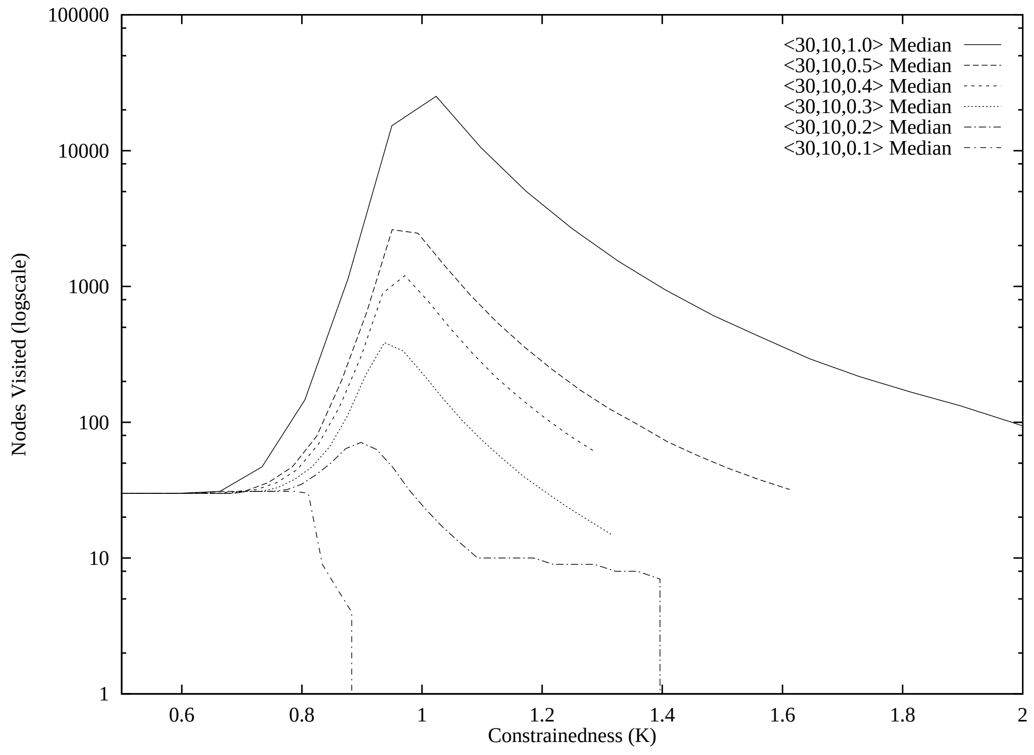


Figure 7.5: Ranges of consistency checking cost for MAC on four $\gamma \approx 4.9$ CSP classes.



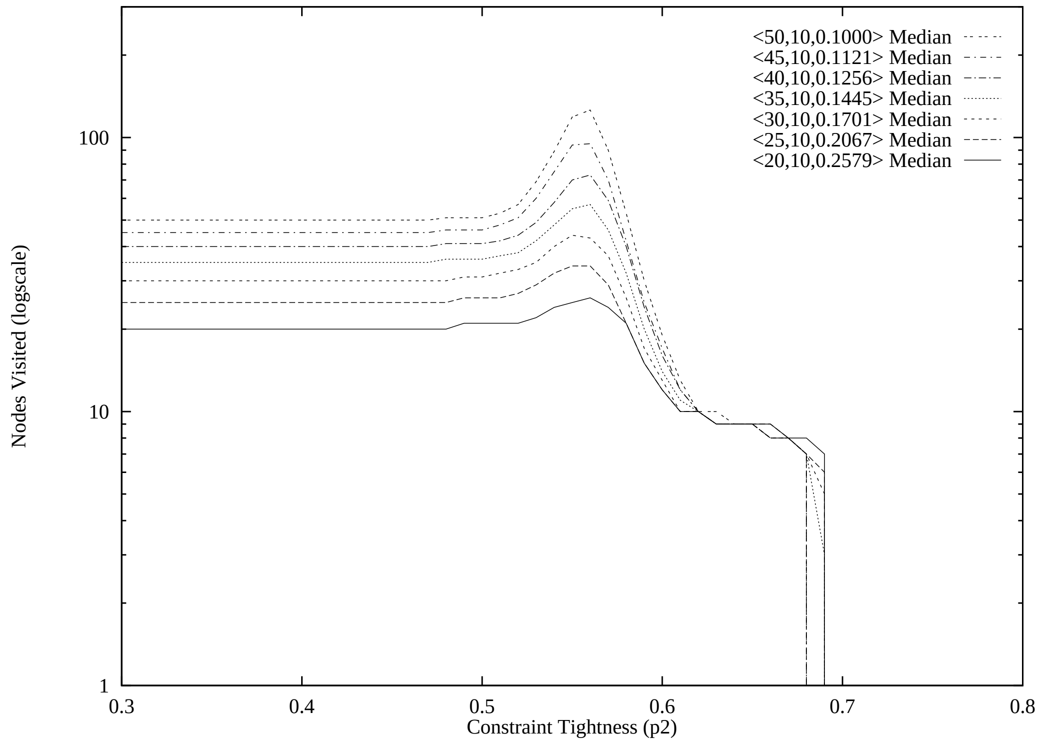


Figure 7.6: Median cost of MAC on $n = 30$ and $\gamma \approx 4.9$ series, in terms of nodes visited.

the constrainedness increases, followed by a sharp drop below n and quickly to zero. The sudden drop in the curve clearly corresponds with the crossover point, separating the regions where more than half of the problems are soluble and more than half are insoluble.

7.6.3 Comparison with FC

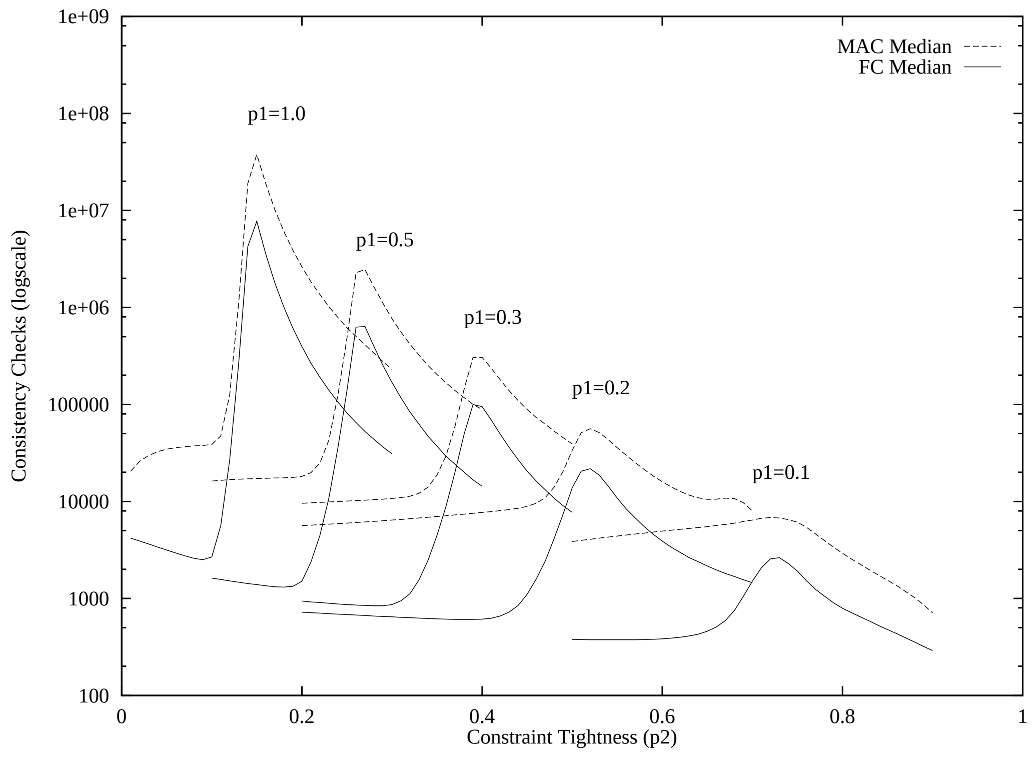
The relative performance of MAC and FC can be compared by plotting the median behaviour in terms of both consistency checks and nodes visited, and by studying the relative incidence of ehps over the same populations of problems. It should be noted that fair comparison of performance between standard MAC and FC searches on *individual* problems is difficult, as the nature of the fail-first principle means that the algorithms do not necessarily follow the same search paths — thus problems that are hard for one algorithm are not necessarily hard for the other.

Figure 7.7 plots the relative median cost of the algorithms against p_2 over each of the $n = 30$ problem classes studied, in terms of both checks and nodes visited. From these plots, it can be seen that MAC performs more poorly on average, in terms of consistency checks, for each class. The differences are particularly large over the easy-soluble regions, although this is partly to be expected as the extra lookahead of MAC (and indeed the lesser lookahead of FC) is redundant effort on most of these very easy problems.

However, a significant improvement occurs on average for MAC over FC in terms of nodes visited over each problem class. Significantly, it can be seen that MAC extends the part of the easy-soluble region in which no backtracking during search is required (for at least half of the problem samples) further towards the crossover point. This agrees with the observations of both (Haralick and Elliott 1980) and (Nadel 1989). Study of the effects of AC preprocessing in Chapter 6 show that it has little effect in removing values from the domains of variables in problems that lie in the easy-soluble region. It is therefore clear that it is the re-establishing of arc consistency that extends the region of backtrack-free search, and not the initial preprocessing. MAC also extends the part of the insoluble problem region in which no search is required for at least half of the problems further towards the crossover point. This is undoubtedly due in large part, however, to the AC preprocessing becoming effective when the constraints are tight. In short, the plots of nodes visited show that by using MAC rather than FC, the range of values over which any search is required for at least half of the problems is squeezed towards the phase transition.

Figure 7.8 compares the median consistency checking cost of MAC and FC for three of the $\gamma \approx 4.9$ problem classes. For small n, FC always outperforms this implementation of MAC. However, as problems become larger, it can be seen that the rate of growth of search effort for MAC on the hard phase transition problems is less than that for FC: at $n = 60$ MAC outperforms FC on average around the crossover point, and at $n = 70$ MAC is almost an order of magnitude better on these hard problems. To investigate this scaling relationship, the respective median consistency checking costs for MAC and FC at the *observed* p_{2crit} values (Section 3.2) were compared for each $\gamma \approx 4.92$ problem class. A plot of these values against n can be seen in Figure 7.9. With a logarithmic scale along the y-axis, the curves for MAC and FC are roughly linear, and it can clearly be seen that the curve for MAC has a lower gradient than that for FC.

Figure 7.10 shows the median and higher percentiles of consistency checking for FC on four



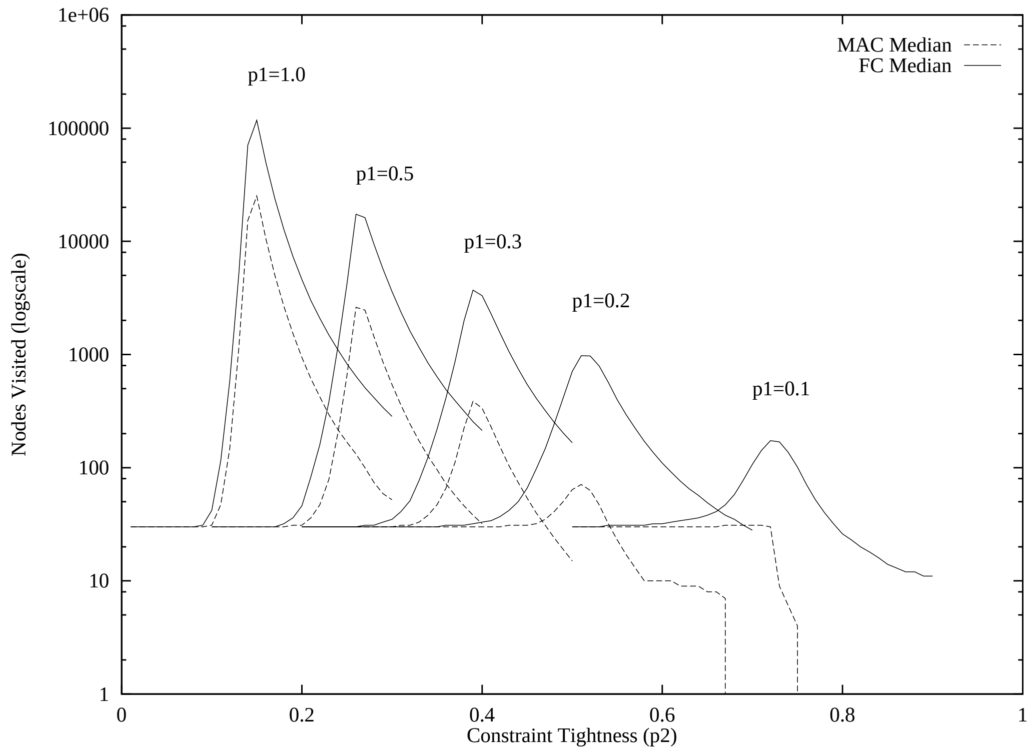


Figure 7.7: Comparison of median cost of MAC versus FC for $n = 30$ series, in terms of both consistency checks and nodes visited.

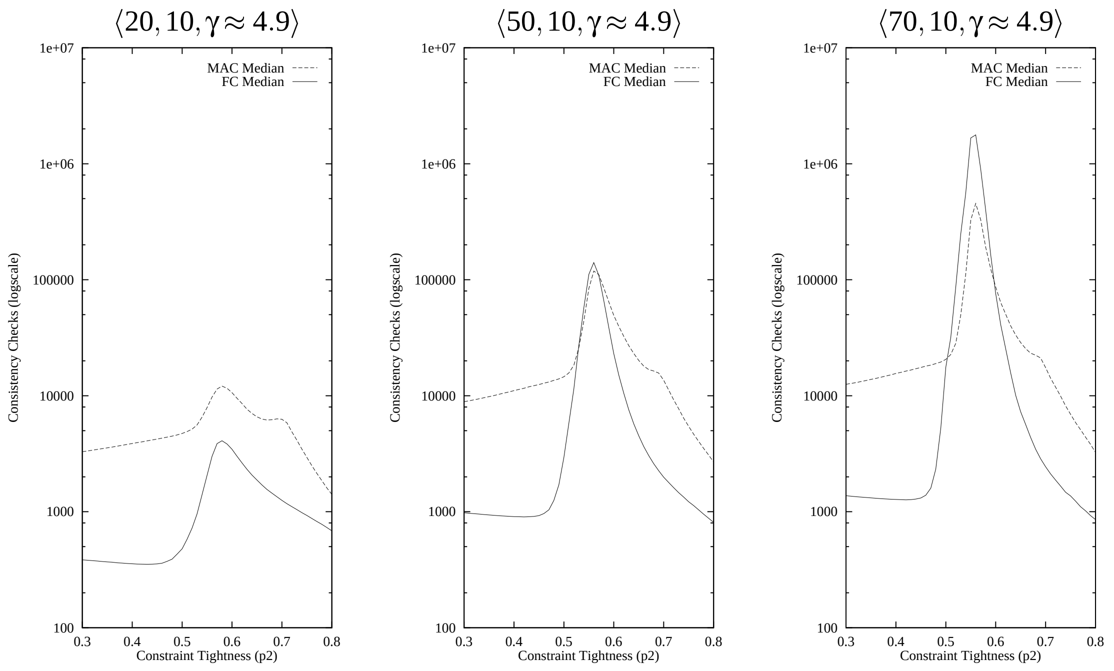


Figure 7.8: Comparison of the median cost of MAC and FC for three $\gamma \approx 4.9$ CSP classes, in terms of consistency checks.

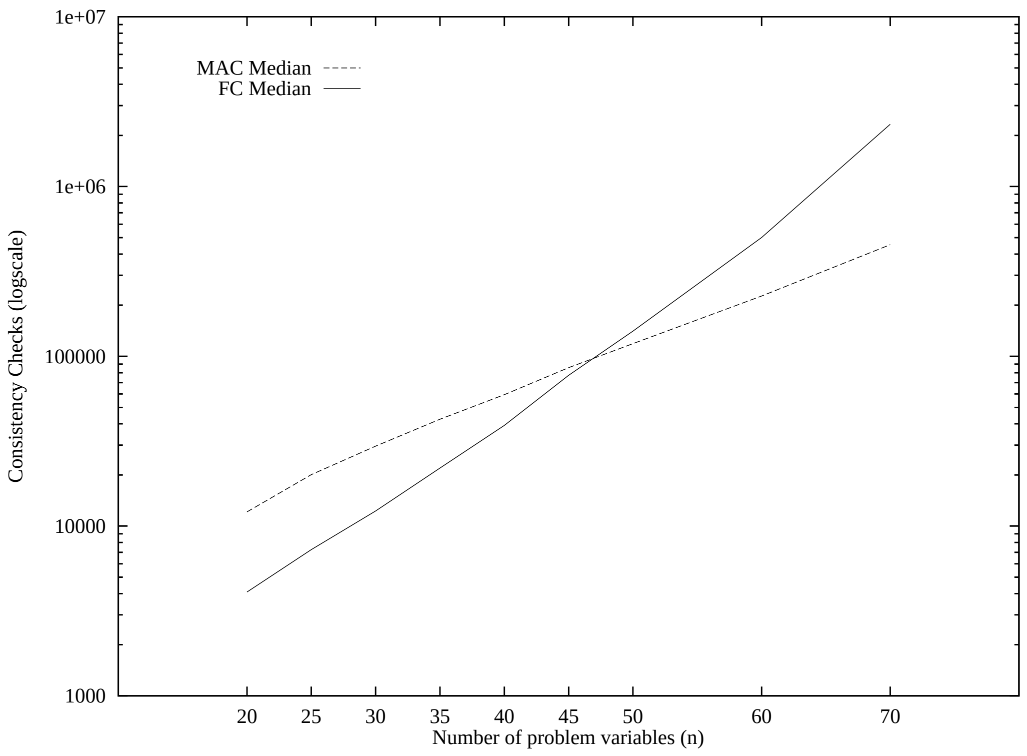


Figure 7.9: Comparison of median search cost of MAC and FC at p_{2crit} for the nine $\gamma \approx 4.92$ CSP classes.

of the $\gamma \approx 4.9$ problem classes. These plots can be directly compared with those showing the performance of MAC over the same problems, in Figure 7.5. Making this comparison, it can clearly be seen that although MAC has been shown to still be susceptible to ehps, their incidence is greatly reduced from that of FC over the same populations of problems. Whether the use of MAC also reduces the magnitude of the exceptionally hard searches that it does encounter is not entirely clear, however: although the heights of the ehp 'peaks' that can be seen in the plots for MAC are generally lower than those for FC, some MAC ehps such as that at $\langle 50, 10, 0.1, \gamma \approx 4.9 \rangle$ (Figure 7.5) are more extreme than nearly all FC ehps in the same problem class.

7.6.4 Overall performance

Since the number of consistency checks performed by a MAC algorithm will depend on the arc consistency algorithm used, it should be considered in conjunction with the number of nodes visited when assessing the maintenance of arc consistency as a general strategy. We have seen MAC perform poorly in terms of consistency checks on densely constrained problems, where Sabin and Freuder reported significant gains in terms of cpu time by their version of MAC, based on AC4 over FC. However, when comparing MAC and FC in terms of nodes visited — an implementation independent measure — MAC performs much better than FC on problems of all constraint densities. Naturally, the efficiency of the MAC implementation will be a very important issue in practical situations.

On sparsely constrained problems, MAC performs much more favourably in terms of consistency checking. It has been seen that the growth in cost at the phase transition peak as the number of problem variables increases, with problem topology maintained, is considerably slower than that for FC. It may be the case that a similar scaling relationship holds for densely constrained problems, but the high costs of an empirical study have prohibited investigation of this.

In terms of exceptionally hard problems, the value of using an increased lookahead with MAC has been demonstrated in its ability to greatly reduce the incidence of ehps compared to FC over the same populations of sparse problems.

7.7 Macroscopic Performance of MAC-CBJ

The performance of MAC-CBJ in isolation is observed in terms of its average cost, and the incidence and magnitude of exceptionally hard problems is compared with that for MAC in order to determine the effects of introducing backjumping. A comparison with FC-CBJ is then made in order to study the the effect that extra lookahead has when backjumping is also available.

7.7.1 General and extreme behaviour

Figure 7.11 shows the median behaviour of MAC-CBJ, in terms of consistency checks, plotted against constrainedness for a selection of the $n = 30$ problem classes, while Figure 7.12 shows the median and higher percentiles for three of these problem classes plotted against constraint tightness, and Figure 7.13 shows a similar analysis for a selection of the $\gamma \approx 4.9$ problem classes.

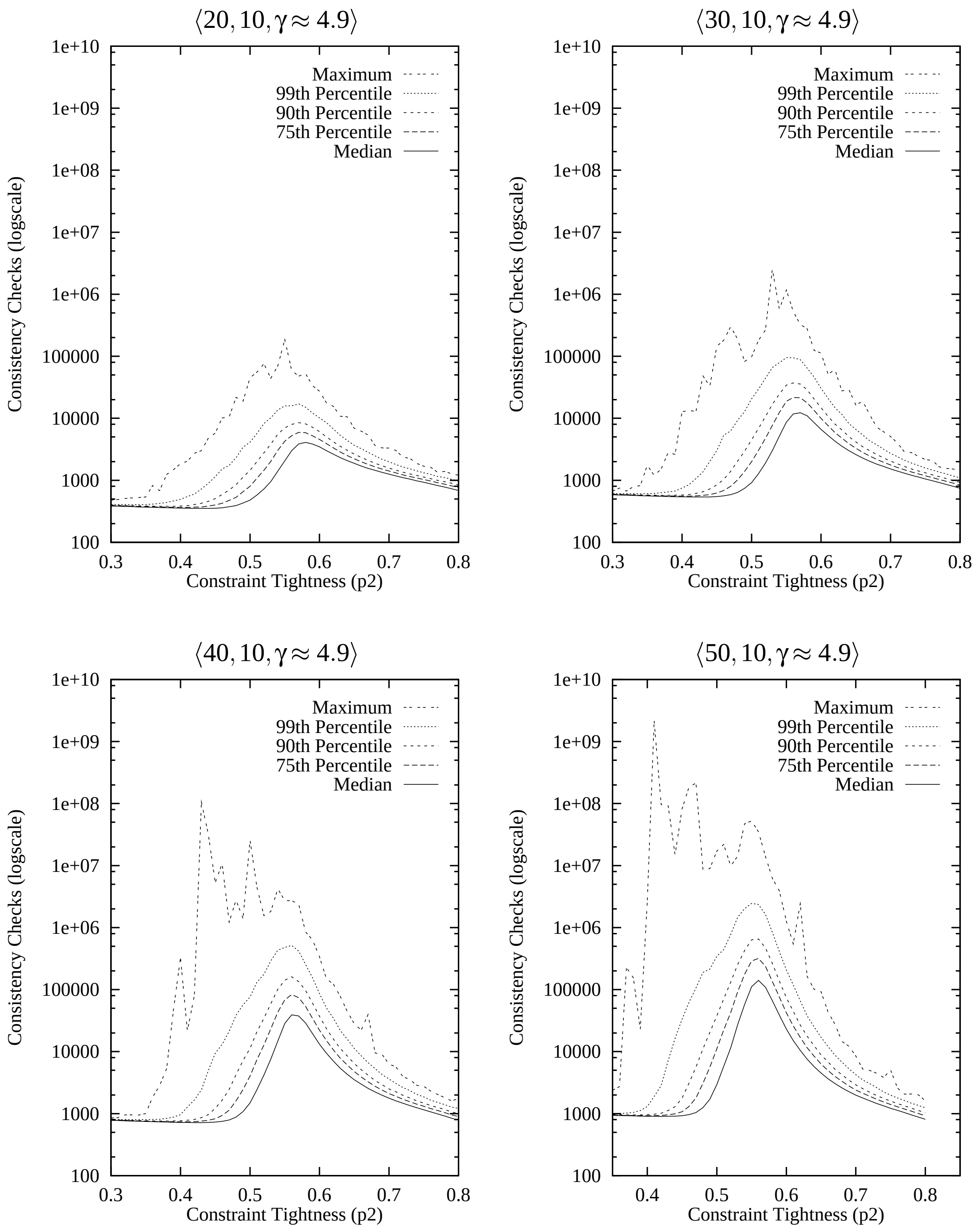


Figure 7.10: Ranges of consistency checking cost for FC on four $\gamma \approx 4.9$ CSP classes.

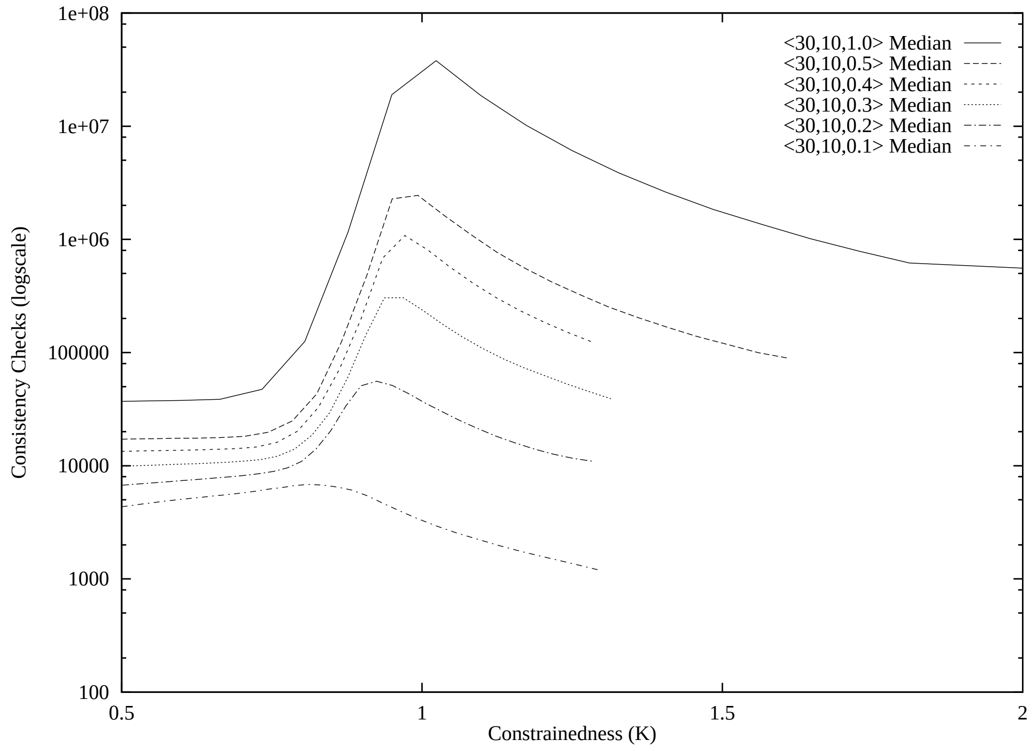


Figure 7.11: Median cost of MAC-CBJ over $n = 30$ series, in terms of consistency checks performed.

These figures can be directly compared with those showing the equivalent data for MAC, shown in Figures 7.2, 7.3 and 7.5 respectively.

By comparing the sets of plots, it is evident that the behaviour of MAC and MAC-CBJ is very similar at the median and higher percentile levels apart from the maximum, for all problem classes. This suggests that CBJ's biggest effect is on the most difficult problems, and that its performance is otherwise similar to chronological backtracking, when 'fail-first' dynamic variable ordering is used. The set of plots also agree with the observation in Chapter 5 that the addition of CBJ does significantly reduce the difficulty of the ehps that MAC finds in populations of problems. We observe from the maximum curves for the MAC-CBJ plots that backjumping greatly moderates the extreme problem behaviour for all of the sparsely constrained problem classes, including even the extremely sparse $\langle 30, 10, 0.1\rangle$ populations of problems, for which highly erratic maximum behaviour with MAC still occurs. It should be remembered that MAC and MAC-CBJ have been implemented so that they both follow the same search paths. Thus, MAC-CBJ must encounter the same subproblems which lead MAC into exceptionally hard search, but clearly deals with them much more quickly than the chronological backtracker can.

These results are similar to those reported in (Smith and Grant 1995a), where the effects of adding CBJ to FC were studied. However, the difference in performance between MAC and MAC-CBJ on the non-exceptional problems is less than that between FC and FC-CBJ. A noticeable improvement was observed with FC-CBJ at the 99% level of behaviour (and at lower levels in some cases) for sparsely constrained classes of problems. However, no noticeable differences in performance can be seen at the 99% level when CBJ is added to MAC for any of the sparsely

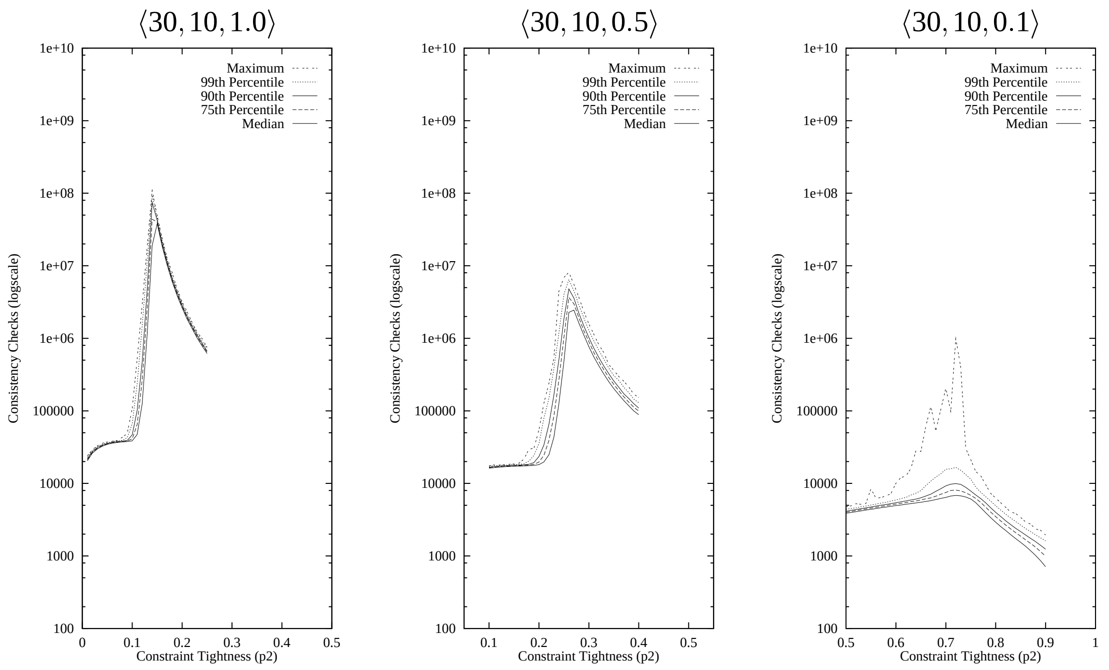


Figure 7.12: Ranges of consistency checking cost for MAC-CBJ on three $n = 30$ CSP classes.

constrained problem classes studied here. We investigate the reasons for this in Section 7.8, by studying the relative structures of the search trees of MAC and FC searches.

7.7.2 Comparison with FC-CBJ

Figure 7.14 shows the median and higher percentiles of consistency checking effort for FC-CBJ on the same $\gamma \approx 4.9$ problem classes shown for MAC-CBJ in Figure 7.13, with which it can be directly compared. Once again, it should be noted that fair comparison of performance between standard MAC-CBJ and FC-CBJ searches on individual problems is not possible as the algorithms will generally not follow the same search paths, as explained in Section 7.6. An improvement in terms of maximum and median consistency checks by MAC-CBJ over FC-CBJ similar to that of MAC over FC is observed. In particular, for the maximum curves it can be seen that while FC-CBJ clearly suffers from instances of exceptionally hard problems, the extra lookahead of MAC combined with the backjumping of CBJ results in clear ehp behaviour being almost eliminated from the populations of sparse problems. However, a small amount of ehp behaviour can still be observed for MAC-CBJ: the spike in the maximum curve for the $\langle 40, 10, 0.1256 \rangle$ problem class at $p_2 = 0.5$, which can be seen in Figure 7.13, clearly fits the ehp criteria given in Section 5.2. These results indicate that while MAC-CBJ shows the most stable performance in respect of the occurrence of exceptionally hard problems, a very few instances can still arise in sparsely constrained problem classes.

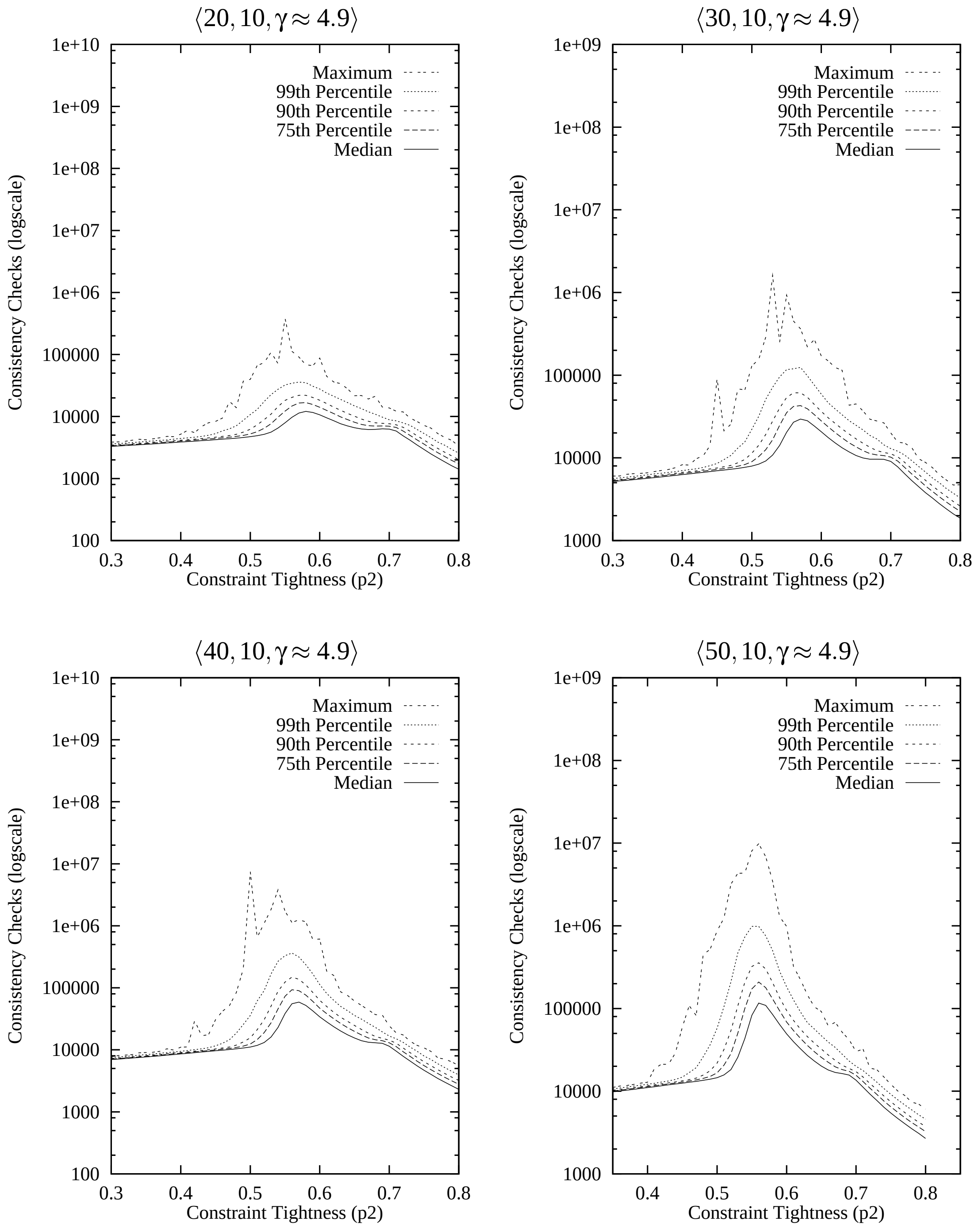


Figure 7.13: Ranges of consistency checking cost for MAC-CBJ on four $\gamma \approx 4.9$ CSP classes.

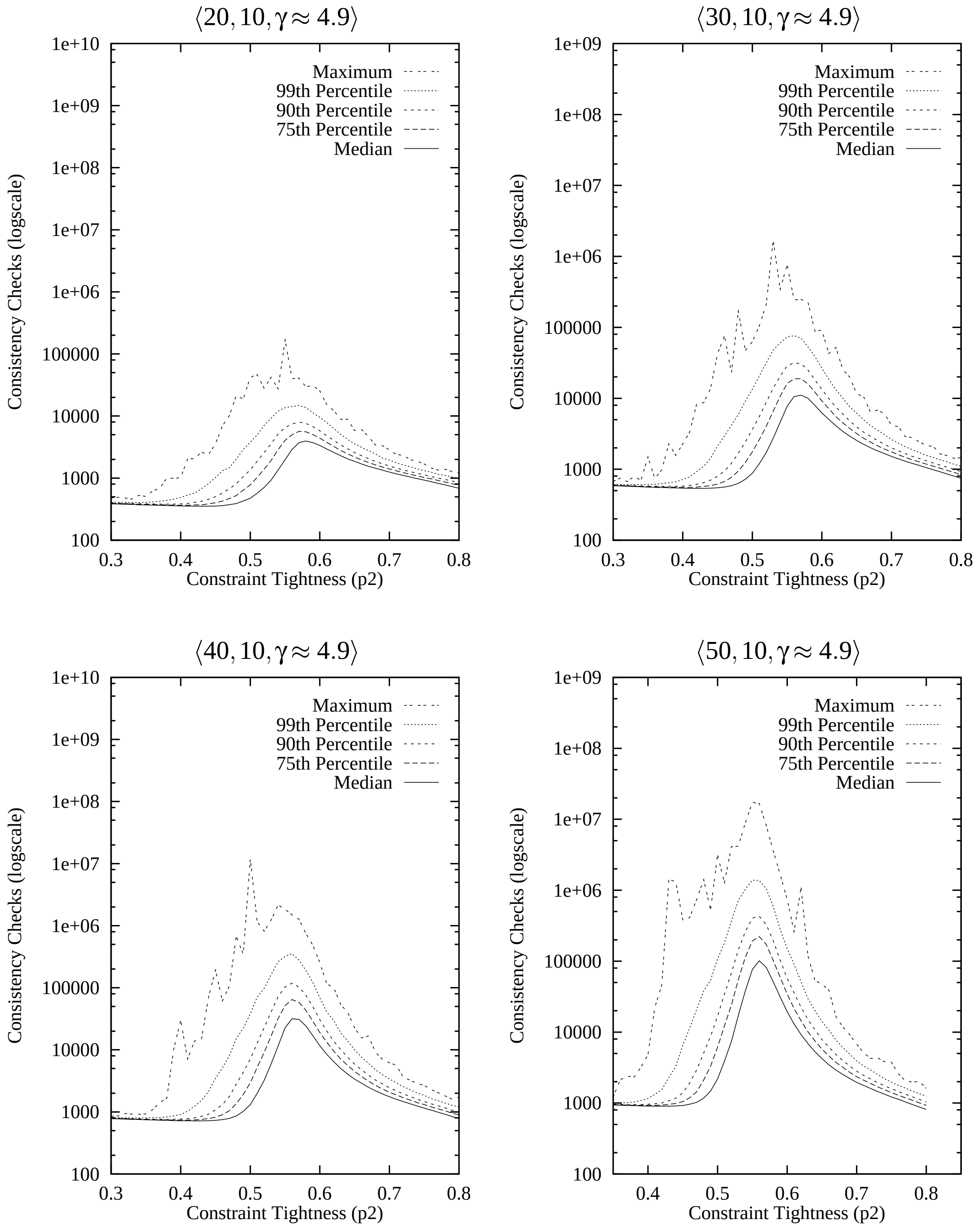


Figure 7.14: Ranges of consistency checking cost for FC-CBJ on four $\gamma \approx 4.9$ CSP classes.

7.8 The Search Trees of MAC and FC

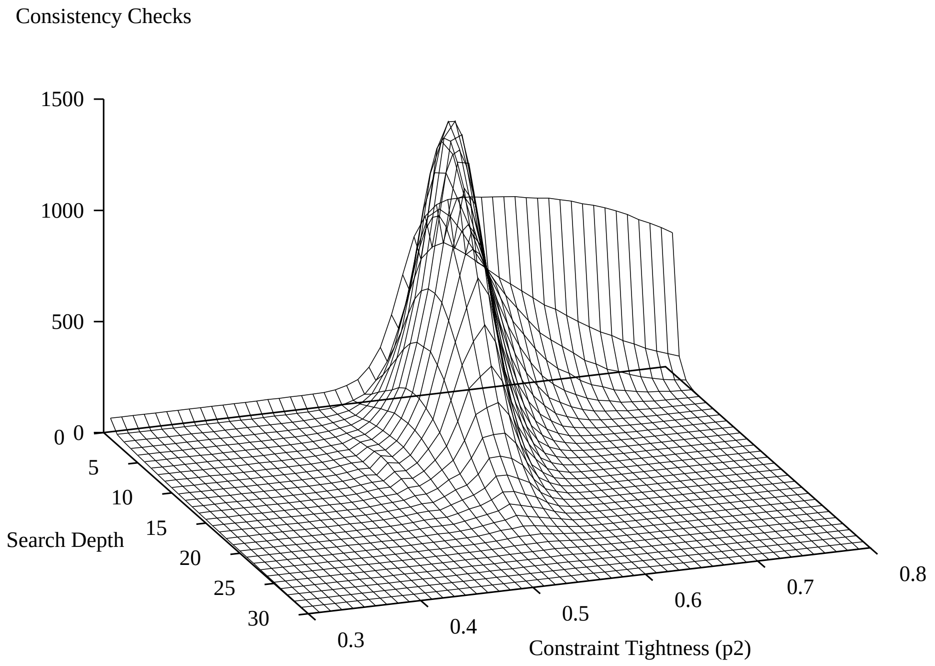
In sections 7.6 and 7.7, we compared the overall search costs of MAC-based algorithms with FC-based algorithms. However, these measurements give little indication of the comparative structure of the algorithms' search trees, which must surely be different. We have seen that for many of the problem classes examined, FC outperforms MAC in terms of consistency checks, while the opposite is true in terms of search nodes visited. This suggests that MAC tends to do more work earlier on in search than FC does.

In order to investigate the relative structures of the MAC and FC search trees, the search effort spent at each depth in the search tree was studied, focusing on the $\langle 30, 10, \gamma \approx 4.9 \rangle$ problem class. Looking inside the search process in this fashion is not new: (Haralick and Elliott 1980) performed a similar investigation into the search depths where the various lookahead algorithms they studied did most of their work. For implementation reasons, the mean numbers of consistency checks and nodes visited at each search depth for each set of problems were recorded, rather than the median. As a result, data from the FC-CBJ and MAC-CBJ searches was used instead of that from FC and MAC, so as to minimise any effects on the mean behaviour caused by exceptionally hard problems.

Figure 7.15 presents a three-dimensional plot, showing where FC-CBJ spends its consistency checking effort. The vertical and horizontal axes show the number of checks against p_2, while the axis projecting from the page shows the depth in the search tree. The range of search depth values is $[0..30]$, for consistency with later plots of MAC-CBJ, where depth 0 represents the AC preprocessing stage. For FC-CBJ there are no values at depth 0, since no preprocessing is done. Below the 3-d plot, a two-dimensional profile of the surface is shown, looking along the constraint tightness axis.

The left hand side of the surface in Figure 7.15 represents the easy-soluble problem region. Here, the highest number of checks occur at depth 1, where there is the maximum number of future variables $(0.17(n-1))$ to check against. Progressively less checking occurs at lower depths, as the number of forward checks to make decreases. As there is little or no backtracking necessary on these problems, the mean number of checks falls smoothly to zero at depth n. Moving into the clear phase transition region, we see a great deal of consistency checking deep into the search tree. From the lower profile plot, the peak in this effort lies at depth 8, and not zero. It appears to be the case than when Forward Checking on these sparse problems, the first few instantiations are fairly easy to make. Making further instantiations becomes more difficult, resulting in a lot of backtracking to the higher levels of the search tree. In the insoluble region, to the right, the majority of checking occurs at the top of the search tree. This is due to insolubility being detected early in these problems, given their over-constrainedness. The number of checks at depth 1 in this region starts off relatively high near the phase transition, where many future variables must still be examined, but falls as p_2 increases. As p_2 approaches 1 (not shown in Figure 7.15), insolubility will be determined by examining only one constraint, requiring m^2 checks.

Figure 7.16 is similar to Figure 7.15, showing the consistency checking behaviour of MAC-CBJ over the same problems. It should be noted, however, that as MAC-CBJ includes an AC3 preprocessing stage, we see consistency checking activity at depth zero. We also see the additional



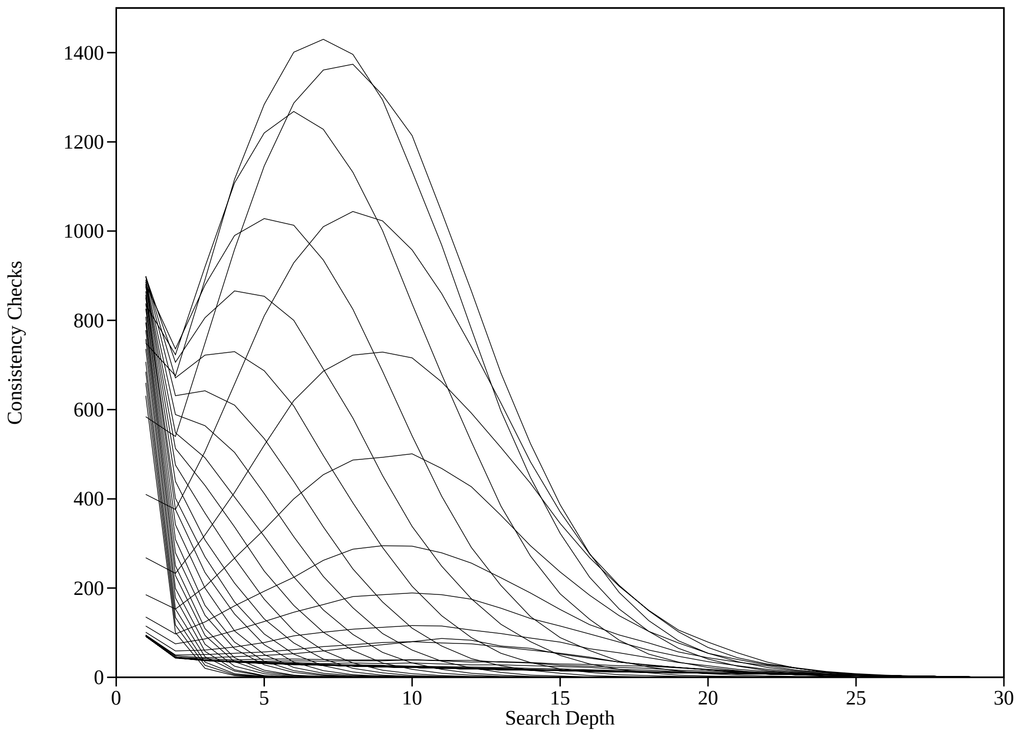
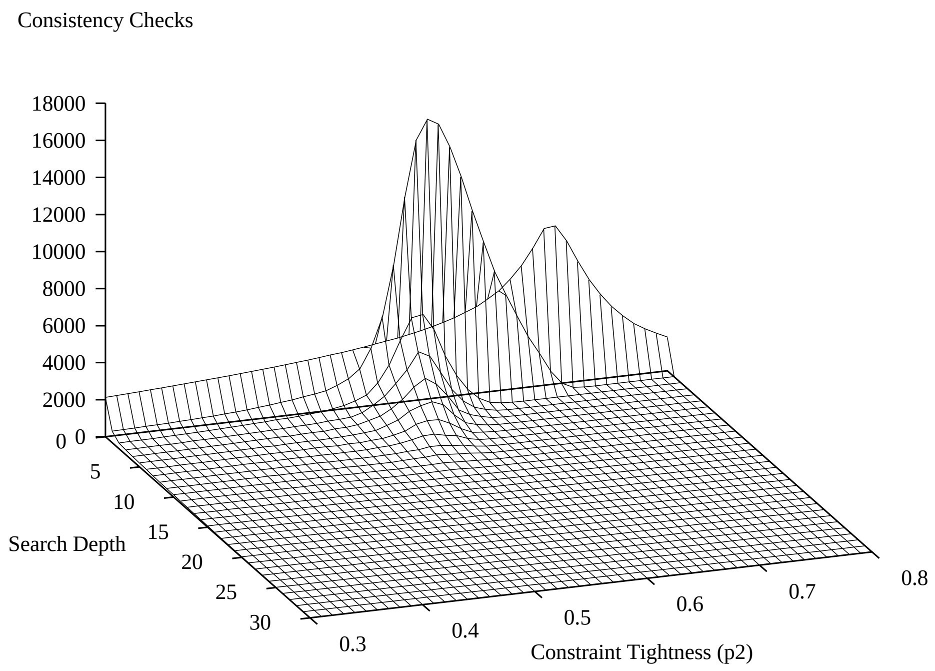


Figure 7.15: Mean consistency checks at each search depth by FC-CBJ on $\langle 30, 10, \gamma \approx 4.9 \rangle$ problems.



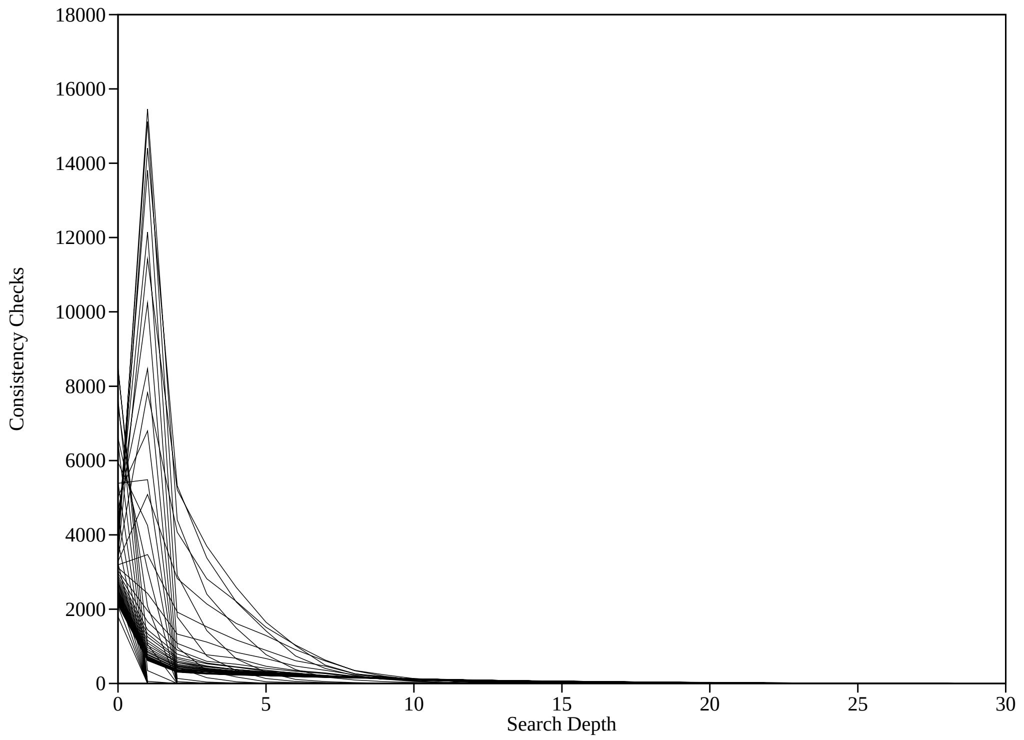


Figure 7.16: Mean consistency checks at each search depth by MAC-CBJ on $\langle 30, 10, \gamma \approx 4.9\rangle$ problems.

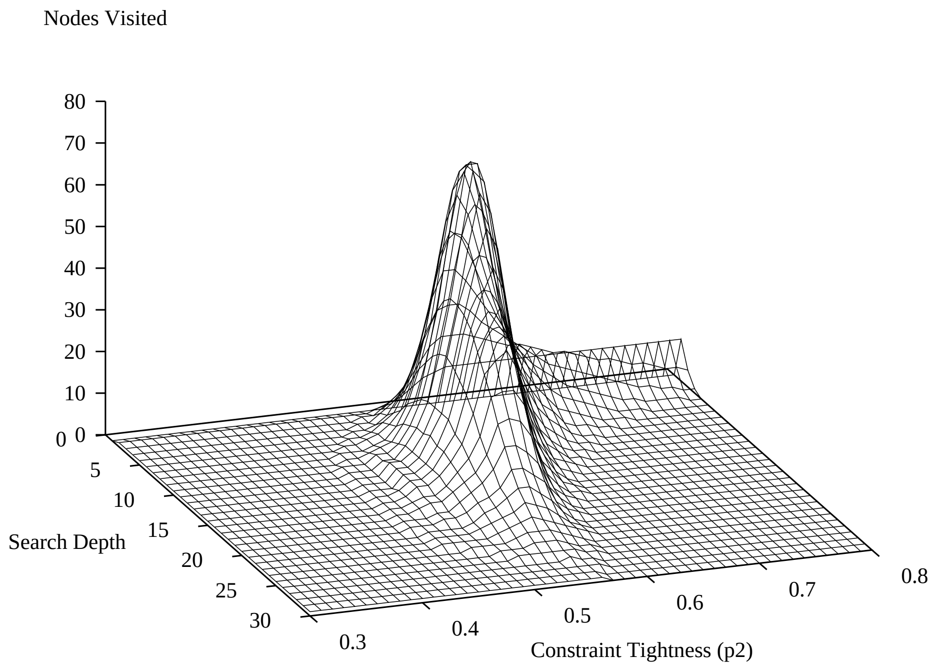
transition associated with establishing arc consistency, discussed in Chapter 6, which lies well into the insoluble problem region as expected.

It is immediately apparent from Figure 7.16 that MAC-CBJ expends the majority of its effort at shallower depths of the search tree than FC-CBJ. The profile plot shows that the peak in checking at the phase transition now occurs at depth 1, compared with depth 8 for FC-CBJ. However, the amount of effort involved is considerably larger: the vertical axis in Figure 7.16 has a range more than an order of magnitude greater than that in Figure 7.15. Thus, while MAC-CBJ could be said to do less searching, this takes more effort. In the insoluble problem region, we can see the point at which AC preprocessing becomes sufficient to prove insolubility in all problems, where no consistency checking occurs at depth 1 or beyond. Immediately prior to this, MAC-CBJ can detect insolubility at depth 1 on many problems.

Figure 7.17 examines the same set of problems for FC-CBJ, this time plotting the search effort on the vertical axis in terms of nodes visited. In the easy-soluble region, the flat area lies at height 1 on the vertical axis. This represents the backtrack-free soluble problems, where one search node is visited at each depth in instantiating a variable. Meanwhile, in the easy-insoluble region there are few nodes visited at great depths. We therefore observe a ‘step’ in nodes visited from 1 to 0 at maximum search depth, at the phase transition crossover point. In the phase transition region, we see a similar pattern to that observed in Figure 7.15, although the peak in nodes visited occurs at depth 11, rather than 8 where the peak in consistency checks occurs. This difference may be attributed to the fact that at depth 11 there is a smaller set of future variables to check against than at depth 8, so although more nodes are visited here, the amount of resultant checking is slightly less. Only slightly fewer nodes are visited at depth 8 than 11.

Figure 7.18 shows the search nodes visited by MAC-CBJ at each search depth, in a similar style to Figure 7.17. This time there are no values at depth 0, since no nodes are visited during preprocessing. It is noticeable that the range on the vertical axis is only 10, while that for FC-CBJ is 80. This coarseness results in very pronounced contours on the MAC-CBJ surface. The step in the surface between the soluble and insoluble regions at the greater depths is clear, as is the point at which preprocessing eliminates the need for any search in the insoluble region. At the phase transition, the peak in nodes visited again occurs at depth 1, and it can be seen that the mean number of nodes falls to exactly one at depth 11. This shows that on $10,000$ of the hardest problems in this class, the lookahead of MAC-CBJ means that inconsistent subproblems are detected after at most 11 instantiations.

The brief look inside the search process of FC-CBJ and MAC-CBJ presented here has demonstrated that the size of the search trees produced by MAC-based searches is considerably smaller than those produced by FC-based searches. In Section 7.7, it was observed that the addition of CBJ to MAC leads to only a tiny improvement in search cost that is difficult to notice even at the 99% level of behaviour. The likely explanation for this is that the comparative shallowness of MAC search trees provides very limited scope for effective backjumping. It appears that the large amount of effort expended by MAC in making its early search moves means that later moves are less likely to fail, so that backtracking from deep in the tree is similarly unlikely.



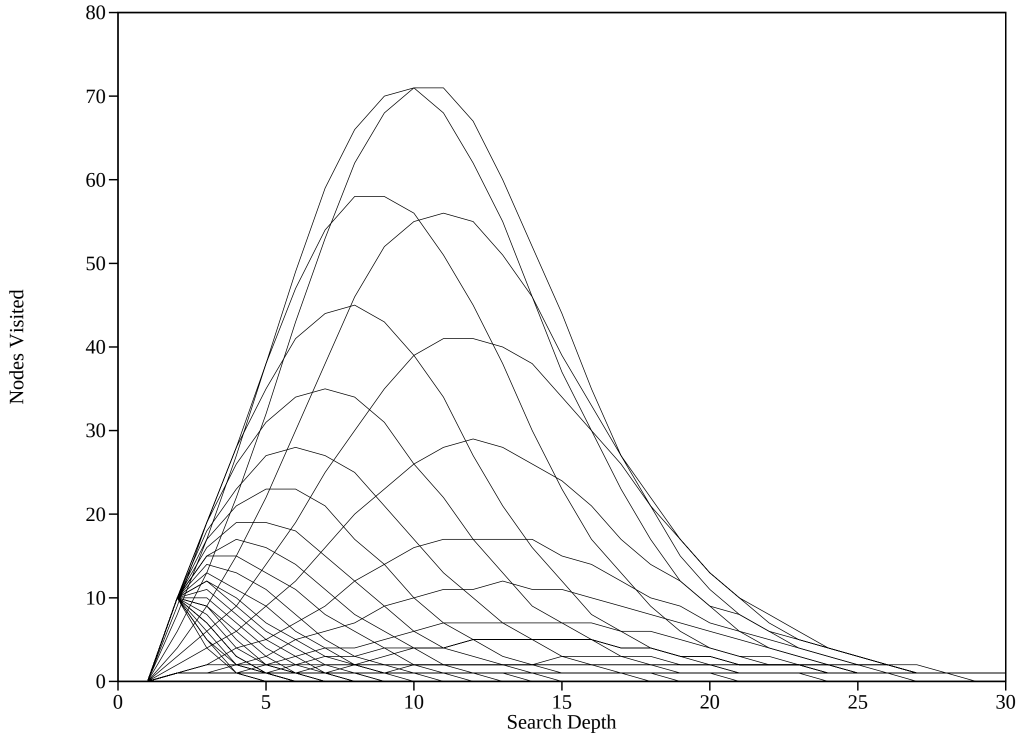
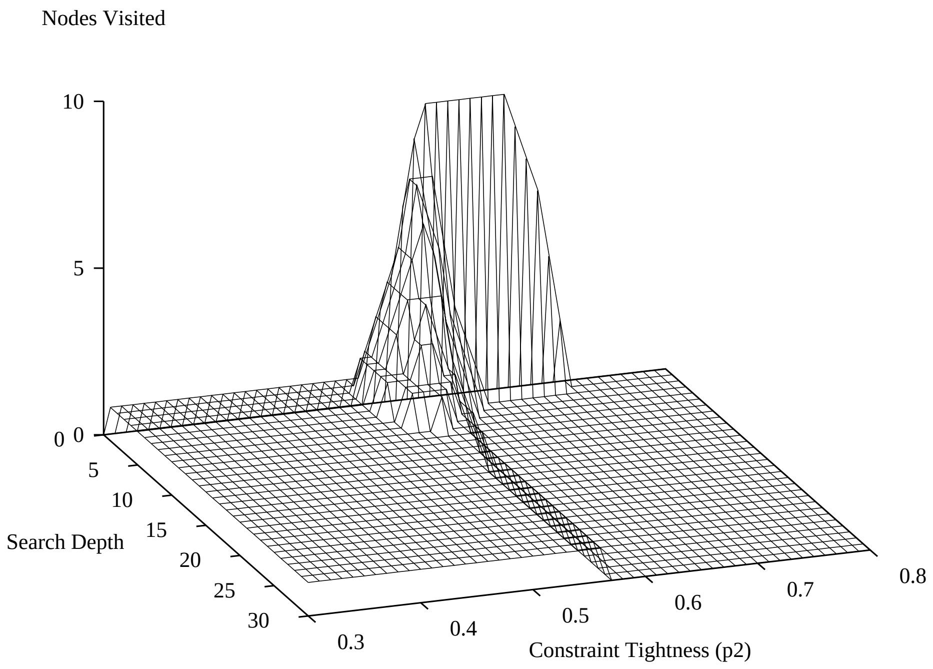


Figure 7.17: Mean nodes visited at each search depth by FC-CBJ on $\langle 30, 10, \gamma \approx 4.9 \rangle$ problems.



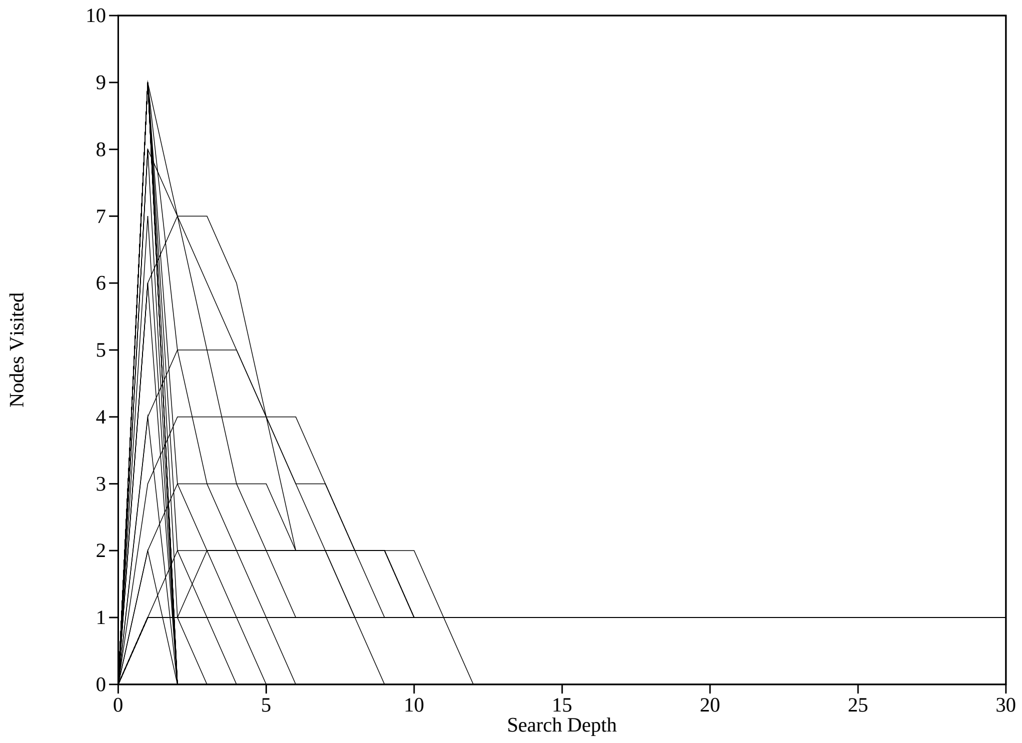


Figure 7.18: Mean nodes visited at each search depth by MAC-CBJ on $\langle 30, 10, \gamma \approx 4.9\rangle$ problems.

7.9 Discussion

We have examined two algorithms which maintain arc consistency, and attempted to position their performance in terms of the average and extremes of behaviour. This has involved rigorous empirical experimentation over a broad range of problem sizes and topologies in a manner which has until recently been extremely rare. In doing so, the findings that have been reported throw up a number of subsidiary issues, which warrant major studies in their own right and so lie beyond the scope of this investigation. These are briefly discussed a little later in this section.

In studying the general behaviour of the MAC and MAC-CBJ algorithms, and comparing them with the FC and FC-CBJ algorithms, an emphasis has been placed on the number of search nodes visited as a good measure with which to compare performance. It has been observed from studying nodes visited that the lookahead of MAC produces backtrack-free searches on average over a greater range of values of the control parameter than the lesser lookahead of FC can. This has been the case for all of the CSP problem classes studied. As might be expected, as the searches become harder the extra lookahead of MAC also allows the algorithm to visit far fewer search nodes on average than FC, again over all observed problem classes. Looking at consistency checking effort (an implementation-dependent measure), we see MAC perform poorly compared to FC on smaller problems of all constraint densities, due to the high arc consistency overheads. However, by studying the independent effects of altering problem size and topology it has been shown that as problem size increases, the FC consistency checking effort on the hard phase transition problems grows at a greater rate than that for MAC, which becomes by far the cheaper algorithm on larger problems over these regions. Study of the effects of combining MAC with CBJ show that little benefit is gained, except for the very hardest MAC searches, in a similar fashion to that of combining FC and CBJ. All of these observations appear to reinforce the increasingly prevalent view that ‘champion’ algorithms which perform extremely well on all types of problem do not exist. Algorithms should clearly be chosen to suit the problem characteristics, based on the knowledge gained from empirical studies such as those presented here and in (Tsang *et al.* 1995).

An area for future study is a more detailed investigation of exactly how algorithm performance scales as problem size increases. We have been able to show that MAC performance scales at a better rate than FC as problems become larger, but at present the rates of increase cannot be specified exactly. The application of techniques such as finite size scaling (Gent *et al.* 1995) may make this possible in future, and this would constitute a major advance in the development of an ‘empirical science of algorithms’.

Throughout the main empirical studies that have been conducted, the MAC and MAC-CBJ algorithms have employed ‘fail-first’ dynamic variable ordering, and have maintained arc consistency from search depth zero (i.e. as a preprocessing step, and at every search stage). It is clear that many alternative choices may be made for these aspects of the algorithm specifications, and these may affect their behaviour. The use of a number of alternative dynamic variable ordering heuristics with both FC and MAC is studied in Chapter 8. Varying the depth from which we choose to maintain arc consistency may have the most interesting effects on performance: for instance it has been shown in (Borrett and Tsang 1995) and Chapter 6 that AC preprocessing has no effect on problems with very loose constraints, and so it would clearly be sensible to only enforce arc consistency from search depth 1, or perhaps lower, on such problems. It should also be remembered that that all of the experiments are based on random problems generated according to the model described in Chapter 3. Problems with more structured constraint graphs, varying domain sizes and/or individual constraint tightnesses may well behave differently. Quite how such changes might affect the performance of the algorithms, or the incidence of ehps, remains to be studied.

The study reported in Section 7.8 could be said to have taken a microscope to the search process of MAC and FC, albeit still considering behaviour at the population level. From this, we have enhanced our understanding of the performance of an algorithm which has until recently been rejected by the constraint satisfaction community. The lesson appears to be that to truly understand the nature of search, we must study it using both the ‘telescopic’ means of large population studies and the ‘microscopic’ means of looking inside individual searches.

7.10 Acknowledgements

Some of the material presented here has appeared in (Grant and Smith 1995) and (Grant and Smith 1996c). We acknowledge the contribution of Patrick Prosser in suggesting the study of search trees presented in Section 7.8.

Chapter 8

Dynamic Variable Ordering Heuristics

It is well known that the efficiency of complete search algorithms for the CSP can be increased considerably by the use of dynamic variable ordering (DVO) heuristics (Section 2.4). The most popular DVO heuristics aim to instantiate the variable whose instantiation is most likely to lead to a dead end, in an attempt to observe the 'fail-first' principle, introduced by (Haralick and Elliott 1980). This principle recommends instantiation of the variable most likely to lead to failure, so that dead ends are detected as early as possible. Haralick and Elliott expressed the fail-first principle as "To succeed, try first where you are most likely to fail". They implemented it by a DVO heuristic which chooses next the variable with smallest remaining domain.

'Smallest remaining domain' is still a popular variable ordering heuristic, and is often seen as synonymous with the fail-first principle. It is not the only way of implementing the fail-first principle as a DVO, however, and many subsequent heuristics also attempt to instantiate the variable most likely to result in failure. The empirical studies reported in Chapters 5 and 7, for instance, use a popular variant of Haralick and Elliott's heuristic, which selects the first variable to be instantiated as that with the greatest degree in the constraint graph, and thereafter selects the next variable to be instantiated as that with the smallest remaining domain.

The fail-first principle has become an item of CSP folklore, to which the success of many DVO heuristics has been informally attributed. However, few convincing insights into the reasons for the success of fail-first DVO heuristics have been made, and new techniques which claim to implement the principle have tended to be introduced pragmatically.

There are dangers associated with failure to understand the true nature of algorithms and heuristics. (Hooker and Vinay 1995), for example, showed that the motivation behind a well known DVO heuristic for boolean satisfiability (SAT) problems[1] did not explain its performance. The reasons for its success were considerably more complex than the simple principle behind it, which produced an effective heuristic only by coincidence. In such a situation, any attempts at refining this heuristic would have been based on false assumptions, and successful refinements would have worked only by accident.

This chapter takes a new look at the use of the fail-first principle in designing DVO heuristics.

[1] In SAT terms, the equivalent to DVO heuristics are called *branching rules*.

A theoretical interpretation of the principle is devised, and from this a series of new heuristics is obtained which uses probabilistic methods to select the variable most likely to fail. An empirical study of these new heuristics, along with several existing fail-first DVOs, suggests that the fail-first principle is not as effective as has been assumed.

The study reported in this chapter began as an attempt to find improved dynamic variable ordering heuristics based on the fail-first principle for a class of CSPs. This attempt failed, which leads us to some unexpected conclusions about the fail-first principle itself.

8.1 Related Work

Intuitively, the rationale for the fail-first principle is that if the current path in the search tree will not lead to a solution, it is best to find this out as soon as possible: any delay means wasted effort. In terms of variable ordering, Haralick and Elliott argued that choosing next the variable to which the search algorithm is least likely to be able to assign a value successfully will minimise the expected length of each branch in the search tree. Doing so should reduce both the expected cost of search and the variance of this average.

Making the assumption that every value in a variable's domain is equally likely to succeed, Haralick and Elliott showed that choosing the variable with smallest remaining domain will minimise the expected branch length. They conducted a small empirical study, finding all solutions to the n-queens problem and ensembles of five random $\langle n, n, 1, 0.35\rangle$ CSPs, for the values $n = 4, 5, 6, 7, 8, 9, 10$. Compared to static random variable ordering, the fail-first DVO heuristic significantly improved the search efficiency of the lookahead algorithms examined, including FC and Full Looking Ahead.

The fail-first principle can be applied to static variable ordering (SVO) strategies as well as DVO heuristics. An SVO which considers variables in order of decreasing degree (Dechter and Meiri 1994) aims to cause failures early, as does an ordering which minimises the width of the constraint graph (Freuder 1982). However, studies such as (Purdom 1983; Dechter and Meiri 1994) have supported the general conclusion that dynamic variable ordering is superior to static variable ordering.

8.1.1 Alternative fail-first DVO heuristics

We give Haralick and Elliott's original fail-first DVO heuristic of selecting smallest domain first the term FF. A number of variants on the FF heuristic have been proposed, which we experimentally evaluate in the following sections. These variants are based on the intuitive idea that a variable which constrains a large number of future variables is also likely to cause a domain wipeout, so that the degree of the variables should be taken into account as well as their domain sizes.

A shortcoming of the FF heuristic is that when all variables have the same initial domain size, the choice of first variable becomes effectively random. A variant used by (Frost and Dechter 1994) selects the first variable to instantiate as the one with the highest degree, i.e. the one constraining the largest number of other variables. Thereafter, the 'smallest remaining domain' strategy is used. This is the heuristic used in Chapters 5 and 7, which we continue to term FFdeg.

A DVO heuristic originally developed for graph colouring problems by (Brélaz 1979) can also be applied to CSPs. The Brélaz heuristic, BZ, selects the variable with the smallest remaining domain and breaks ties by selecting the variable with the highest *future degree*, i.e. the one constraining the largest number of future variables.

(Bessière and Régin 1996) show that the SVO mentioned earlier which considers variables in descending order of degree gives good results in comparison with FF when the constraints are sparse, but performs very badly on complete constraint graphs, when it degenerates to lexicographic ordering. Conversely, FF does much better when the constraints are dense, since the fact that it ignores the degrees of the variables becomes less important. They introduce a heuristic, dom/deg, which combines the two by selecting the variable which minimises the ratio of current domain size:degree. Bessière and Régin report that using the 'global' degree of variables over the entire CSP is roughly as effective as calculating their degree in the future subproblem. We therefore focus on the version of this heuristic that uses global degrees[2] and use the term DD for brevity.

None of these fail-first DVO heuristics have been presented with any analytical explanation of their behaviour, although all have been shown empirically to perform well on many types of CSP. More recent work, however, has taken a more theoretical approach to the design of search heuristics.

8.1.2 Minimising subproblem constrainedness

DVO heuristics based on an entirely different principle are presented in (Gent *et al.* 1996a). These are based on minimising the *constrainedness* (Section 3.2) of the future subproblem. The heuristics use three very general theoretical measures that have been developed for CSPs and other types of problem: the expected number of solutions, $E(N)$; the expected solution density, ρ; and the general constrainedness parameter, κ, which combines these measures. By maximising the first two measures, or minimising the third, during search, it is hoped that the search process will be guided towards under-constrained subproblems with many solutions which will be easy to solve.

Although minimising the constrainedness of the subproblem is achieved by selecting the most constrained variable, this approach cannot be said to follow the fail-first principle, as it attempts to select search paths that are most likely to succeed rather than fail. However, these three new heuristics are empirically shown to perform better than existing versions of fail-first DVOs on some types of CSP, particularly those that are densely constrained or have non-uniform constraint tightness.

The success of these new heuristics leads us to consider one of them, that which minimises κ, along with the fail-first heuristics studied. We term this heuristic kappa.

[2]The variant of dom/deg which uses future degrees is considered in Section 8.7.

8.2 Studying the Fail-First Principle

Our discussion of DVO heuristics based on the fail-first principle is presented in the context of the binary CSPs used throughout this thesis. It is assumed that all variables initially have the same domain size, and that all constraints are uniformly tight: again, similar assumptions have been almost universal in experimental studies. Dynamic variable ordering is only considered in the context of the lookahead algorithms FC and MAC: Section 2.4 notes that DVO has no effect without a search algorithm that has a lookahead capability.

The following section presents the theoretical analysis of fail-first, from which three new heuristics are derived. The framework under which the empirical studies are conducted is then laid out, followed by analysis of the series of results obtained. The consequences of these results for the fail-first principle are discussed, and future trends in the design of DVO heuristics for the CSP are considered. We conclude with a summary and evaluation of this work.

8.3 A Theoretical Interpretation of Fail-First

(Hooker 1996) heavily criticises the use of *competitive testing* when evaluating the performance of a new search heuristic. He suggests that such an approach, in which the new heuristic tends to be accepted or disregarded on the basis of its performance against other heuristics, tells us “which algorithms are better, but not why”. Purely competitive testing of a heuristic (or of any algorithm) fails to provide insights into its behaviour which might aid the development of better techniques. Hooker suggests an alternative approach of *controlled experimentation*:

> Based on one’s insights into an algorithm, for instance, one might expect good performance to depend on a certain problem characteristic. How to find out? Design a controlled experiment that checks how the presence or absence of this characteristic affects performance. Even better, build an explanatory mathematical model that captures the insight, as is done routinely in other empirical sciences, and deduce from it precise consequences that can be put to the test.

We attempt here to follow this approach and construct a mathematical ‘theory’ for the fail-first principle. This theory[3] is designed to specify the circumstances under which the selection of a particular variable to instantiate might to lead to failure. The analysis is based on binary CSPs using the $\langle n, m, p_1, p_2\rangle$ model defined in Section 3.2, and assumes an algorithm with a lookahead capability.

8.3.1 Three new fail-first heuristics

In deriving the ‘smallest-remaining-domain’ heuristic (FF) as an implementation of the fail-first principle, (Haralick and Elliott 1980) assume that the probability that the assignment of a value to a variable fails (in the context of forward checking or maintaining arc consistency, results

[3]As acknowledged in Section 8.14, Barbara Smith is responsible for the probabilistic analysis of the fail-first principle presented below.

in a domain wipeout) is the same for all available values of all unassigned variables. On that assumption, the probability that the variable chosen will fail (i.e. the probability that every value will lead to a domain wipeout) is maximised by choosing the variable with smallest domain. However, it is clear that other factors, such as the number of future variables which each variable constrains, also affect this probability. The variants of FF discussed in Section 8.1 take some account of the future degree of each variable. However, if we want to follow the fail-first principle, it would be better to incorporate these other factors when calculating the probability of failure.

We assume that in the original CSP each variable has m possible values. When there is a constraint between two variables, the constraint tightness is a constant p_2 for all constraints. Suppose that after a number of successful past assignments, we have a future subproblem consisting of a set F of unassigned variables, each variable $v_i \in F$ having current domain size m_i. If there is a constraint between two of these variables, v_i and v_j, then due to the values which have been removed from their domains by the past instantiations, the current tightness of this constraint is p_{ij}, measured by the proportion of the remaining pairs of values which are not allowed.

The fail-first principle says that we should choose next the variable in F which is most likely to fail, i.e. which maximises the probability that every one of its possible values will result in a domain wipeout.

If we consider a variable $v_i \in F$ with current domain size m_i,

$$\Pr\{\text{every assignment of } v_i \text{ fails}\} = (\Pr\{v_i = x_i \text{ fails}\})^{m_i}$$

where x_i is any value in the current domain of v_i.

If there is a constraint between $v_j \in F$ and v_i and the current tightness of this constraint is p_{ij},

$$\begin{aligned}
&\Pr\{v_i = x_i \text{ is consistent with at least one value of } v_j\}\\
&\qquad = 1 - \Pr\{v_i = x_i \text{ is inconsistent with every value of } v_j\}\\
&\qquad = (1 - p_{ij}^{m_j})
\end{aligned}$$

approximately, if we take the current constraint tightness p_{ij} as applying independently to each pair of values. If there is no constraint between v_i and v_j then p_{ij} = 0.

Using the above,

$$\begin{aligned}
&\Pr\{v_i = x_i \text{ fails}\}\\
&\quad = 1 - \prod_{v_j \in F, j \neq i} \Pr\{v_i = x_i \text{ is consistent with at least one value of } v_j\}\\
&\quad = 1 - \prod_{v_j \in F, j \neq i} (1 - p_{ij}^{m_j}) \qquad (8.1)
\end{aligned}$$

Therefore, to choose the variable that is most likely to lead to failure in the future subproblem, we should choose the variable v_i which maximises

$$(1 - \prod_{v_j \in F, j \neq i} (1 - p_{ij}^{m_j}))^{m_i} \qquad (8.2)$$

Depending on how much we estimate versus how much we accurately measure in the environment of each variable v_i, this gives us a series of heuristics:

First approximation: If we assume, as Haralick and Elliott did, that the term (8.1) is the same for every value of every variable v_i, then to maximise (8.2) we should minimise the number of such terms (since they are all < 1) and hence minimise m_i. Thus, we choose the variable that has the smallest current domain, giving the FF heuristic.

Second approximation: We could estimate the current tightness of the constraints between v_i and the other future variables by their original tightness (i.e. 0 or p_2) and use the initial domain size, m, as the estimate for the current domain size of each variable. Then, to maximise (8.2), we maximise:

$$(1-(1-p_2^m)^{d_i})^{m_i}$$

where d_i is the degree of v_i in the future subproblem, i.e. the number of future variables that it constrains. This gives a heuristic that, like BZ and DD, chooses the next variable to instantiate on the basis of both its domain size and its future degree.

Third approximation: We could use the true current domain size of all future variables, but estimate the current constraint tightness by p_2. Then we want to maximise:

$$(1-\prod_{v_j \in F, v_i \text{ constrains } v_j}(1-p_2^{m_j}))^{m_i}$$

If two variables have the same current domain size, this leads us to prefer the one which minimises $\prod(1-p_2^{m_j})$. As well as maximising the number of terms in the product, this maximises $p_2^{m_j}$ and hence minimises m_j. Thus, we favour variables *adjacent to future variables with small domains*.

Fourth approximation: Finally, we can also measure the current tightness of the constraints, and calculate (8.2) accurately for each variable, selecting the one that maximises this term. Then to maximise $p_{ij}^{m_j}$ we should maximise p_{ij} (as well as minimising m_j). This chooses a variable *involved in tight constraints*, other things being equal.

It is intuitive that if we want to choose a variable such that all of its available values are likely to cause a domain wipeout in some future variable we should look for a variable which has few remaining values and which is involved in many tight constraints with future variables which themselves have few remaining values. The final heuristic uses all these factors in selecting the next variable.

We term the second, third and fourth heuristics FF2, FF3 and FF4 respectively, and in later sections we present experimental evidence on their performance relative to FF.

8.3.2 Testing the fail-first principle

Since FF, FF2, FF3 and FF4 are based on increasingly accurate estimates of the probability that a variable will fail, we should expect, if the fail-first principle is sound, to see this reflected in decreasing search effort on the part of the forward checking algorithm for CSPs of the type on which the probabilities are based. The new fail-first heuristics contain a number of attractive features which might be expected to yield good results:

- There are several obvious criteria to consider when choosing the most constrained variable: a small current domain size; a high degree in the future subproblem; adjacent variables with small current domains; and involvement in tight constraints. It is not obvious how to combine these factors, but the theoretical framework shows how it can be done.

- The second heuristic, FF2, is almost the Brélaz heuristic. However, BZ uses the degree of each variable only as a tie-breaker. In some circumstances, FF2 might choose a variable with highest degree in preference to one with smallest domain. This seems to require p_2^m to be relatively large. For example, if $p_2^m = 0.1$ (corresponding approximately to $p_2 = 0.8$, $m = 10$, say) a variable with domain size 5 and degree 3 is preferable to a variable with domain size 4 and degree 2. On highly-constrained problems the proposed heuristic might beat BZ (since p_2 is large, so the above condition may be met.)

- We know of no variable ordering heuristic which takes into account the domain size of adjacent variables. The proposed third heuristic, FF3, provides a relatively cheap way of doing this. Furthermore there is currently no cheap heuristic better than simple FF available when the constraint graph is a clique, since FFdeg and BZ are the same as FF in such cases.

The new heuristics are, of course, also increasingly expensive to apply (and we have ignored this cost in the experiments reported below) but we might hope that FF2 or FF3 provides good performance without the necessity of recalculating the tightness of every constraint after each instantiation, as required for FF4, which is particularly time-consuming.

Should the proposed heuristics not perform well on this class of problem, this would call into question the fail-first principle itself. The experiments will therefore provide more than just a ranking of the heuristics under investigation, which is the approach criticised by (Hooker 1996). Instead, they will serve to confirm or refute the predictions that have made about the new fail-first heuristics, and by extension put the whole fail-first principle to the test.

8.4 The Experimental Environment

Having devised a mathematical interpretation of the fail-first principle, we must now devise a controlled experimentation environment under which the series of suggested heuristics can be tested and compared against existing fail-first strategies.

The experiments reported in this chapter use CSPs generated according to a hybrid of the Model A and Model B methods, specified in Chapter 3. This hybrid generation method treats constraint tightness, p_1, as a fixed proportion (Model B), but treats constraint density, p_2, as a probability (Model A). Thus, the experimental CSPs have a specific number of constraints, whose individual densities may vary according to probability. This is the model used by the analysis in Section 8.3 to create the new fail-first heuristics, and so is the most appropriate with which to generate the CSPs to test them.

In all, there are seven dynamic variable ordering heuristics which are studied: FF, FFdeg, BZ, DD, FF2, FF3 and FF4. The basic CSP search algorithm with which each of the heuristics is used is initially forward checking (FC), and later maintaining arc consistency (MAC). Algorithm-heuristic combinations are described using the terms FC-FF, MAC-FF3, etc.

The cost of each search is measured in terms of consistency checks and search nodes visited by the base algorithm. These measures, discussed in detail in Section 4.4, relate only to the cost of the base algorithm and ignore the heuristic overhead. The chief aim is not to test the implementation efficiency of the heuristics, only their effect on the search process, which is why we do not incorporate their cost into the main analysis.

The results of the empirical studies are presented in the following sections. A study on the effects of the choice of first variable to instantiate is followed by a broad comparison of each of the DVO heuristics. The issue of whether the BZ and DD heuristics should base their decisions on the degrees of variables in the global problem or current subproblem is then considered. The effects of changing the base algorithm from FC to MAC and of scaling up problem sizes are also reported, followed by a brief analysis of how exceptionally hard problem (ehp) behaviour varies with each heuristic.

8.5 Effect of the Initial Instantiation

The basic FF DVO heuristic offers no tie-breaking strategy if more than one variable has the smallest current domain size. In the case of CSPs where all variables have the same domain sizes, this makes FF's initial choice of variable to instantiate effectively random. Heuristics FFdeg and BZ do offer tie-breaking strategies, and for a CSP with uniform domain sizes will select the variable with highest degree as the first to instantiate. These heuristics have been shown to improve search efficiency over FF (Frost and Dechter 1994; Gent *et al.* 1996a).

A simple initial test of the fail-first hypothesis can be made by investigating the effects on search cost of the choice of first variable to instantiate. For the random CSPs that are used in the experiments, the tightness of every constraint is initially identical, making the most constrained variable in every problem that which has the highest degree. This provides an ideal controlled situation in which the effect of starting search with the most constrained variable can be compared against starting with a random choice of variable.

We investigate the influence that the initial choice of variable has, in terms of both dynamic variable ordering strategies and static variable ordering (SVO) strategies.

8.5.1 Effect on static variable ordering

Two phase transition experiments were conducted using the $\langle 20, 10, 0.2\rangle$ and $\langle 20, 10, 0.5\rangle$ CSP classes. The use of these relatively sparsely constrained problems was intended to provide suf-

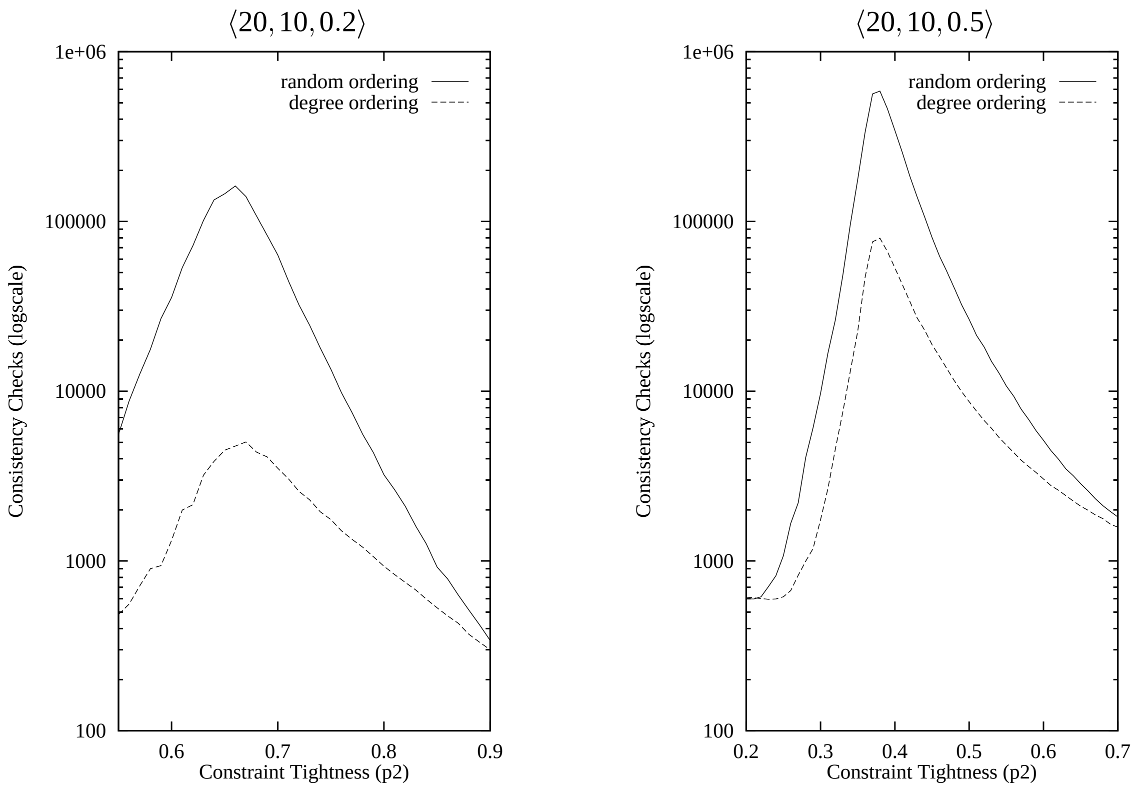


Figure 8.1: FC with static variable orderings on $\langle 20, 10, 0.2\rangle$ and $\langle 20, 10, 0.5\rangle$ CSPs.

ficient variety in the degrees of variables, in order to make the number of variables with joint highest degree reasonably low. The size of each p_2 step for both classes is 0.01, and ensembles of 1,000 problems were generated at every $\langle 20, 10, p_1, p_2\rangle$ point.

Two versions of the FC algorithm employing a static variable ordering were applied to each CSP in the study: one version places the variables in order of decreasing degree; the other uses a lexical (effectively random) ordering.

Figure 8.1 compares the median consistency checking cost of FC using degree and lexical SVO's on the $\langle 20, 10, 0.2\rangle$ and $\langle 20, 10, 0.5\rangle$ problem classes. Cost is plotted against constraint tightness, p_2, and the regions plotted include the phase transitions. For both classes, we see that using a degree SVO leads to a large saving in search cost compared to using a lexical SVO. In the case of $\langle 20, 10, 0.2\rangle$ the saving at the phase transition is more than an order of magnitude, while for $\langle 20, 10, 0.5\rangle$ the improvement is by a factor of six.

8.5.2 Effect on dynamic variable ordering

The same sets of $\langle 20, 10, 0.2\rangle$ and $\langle 20, 10, 0.5\rangle$ problems used above were searched by FC using the DVO heuristics FF and FFdeg. Figure 8.2 compares the median consistency checking cost of FC-FF and FC-FFdeg over the phase transition regions of both problem classes. In the case of these DVO heuristics, the improvement seen is due entirely to the different choices of *first* variable to instantiate. FC-FFdeg is roughly half as expensive as FC-FF at both phase transition peaks.

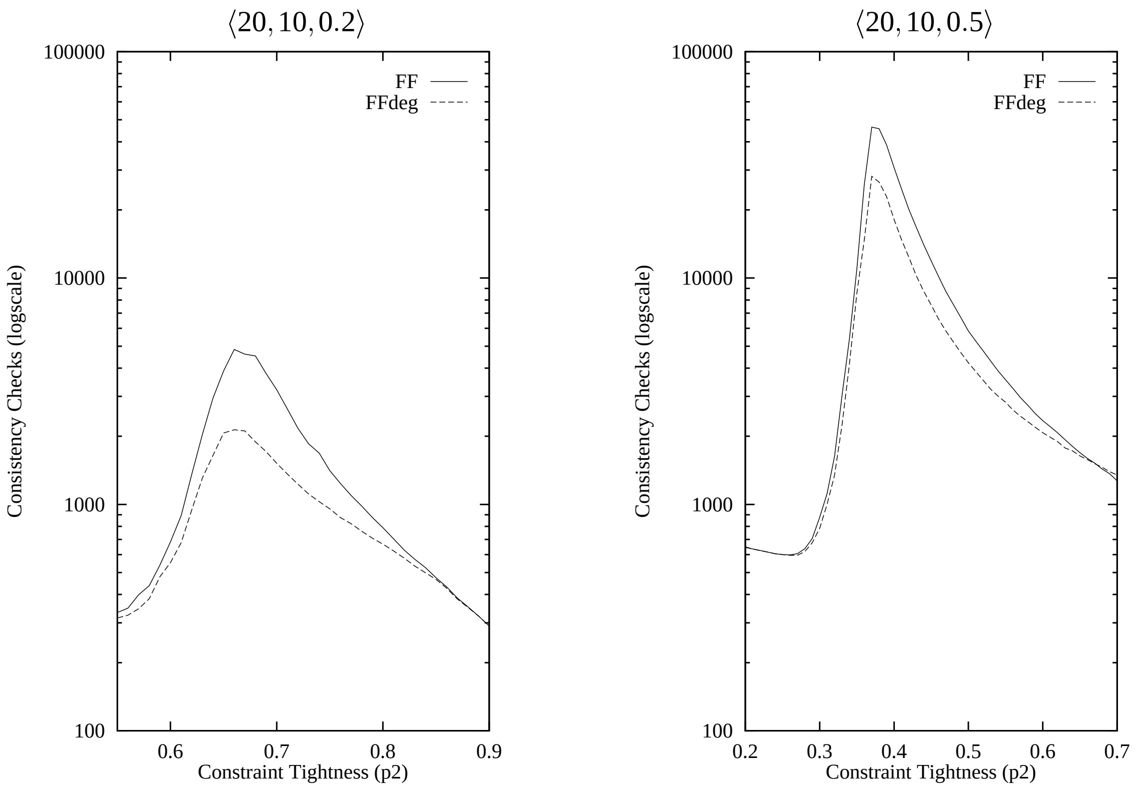


Figure 8.2: FC with dynamic variable orderings on $\langle 20, 10, 0.2\rangle$ and $\langle 20, 10, 0.5\rangle$ CSPs.

8.5.3 Summary

Closer inspection of the data produced shows that the use of static degree ordering with FC reduces the median search cost over lexical ordering by a factor of 34 at the $\langle 20, 10, 0.2\rangle$ phase transition peak, and 7.4 at the $\langle 20, 10, 0.5\rangle$ peak. Equivalent figures for FC-FFdeg over FC-FF are 1.83 and 1.6 These results clearly suggest that choosing the most constrained variable to instantiate is a good idea, at least in the case of the first variable.

8.6 A Simple Beauty Contest

In order to gain some idea of the performance of the eight DVO heuristics under examination, and create an initial ranking, three phase transition experiments on CSPs of size $\langle 20, 10\rangle$ were conducted. $\langle 20, 10, 0.2\rangle$, $\langle 20, 10, 0.5\rangle$ and $\langle 20, 10, 1.0\rangle$ problems were examined, with p_2 varied in steps of 0.01, and ensembles of 1,000 problems generated at every $\langle 20, 10, p_1, p_2\rangle$ point. Every problem was then searched by FC-FF, FC-FF2, FC-FF3, FC-FF4, FC-FFdeg, FC-BZ and FC-DD.

The three values of p_1 were selected in order to test the heuristics on problems where the constraint graph is a clique (and all variables have the same degree), where the constraint graph is of medium density, and where the constraint graph is very sparse.

The data from these experiments is presented below, with the heuristics grouped into two sets to aid the clarity of the plots. A ranking of the heuristics on the problems used is then obtained and discussed.

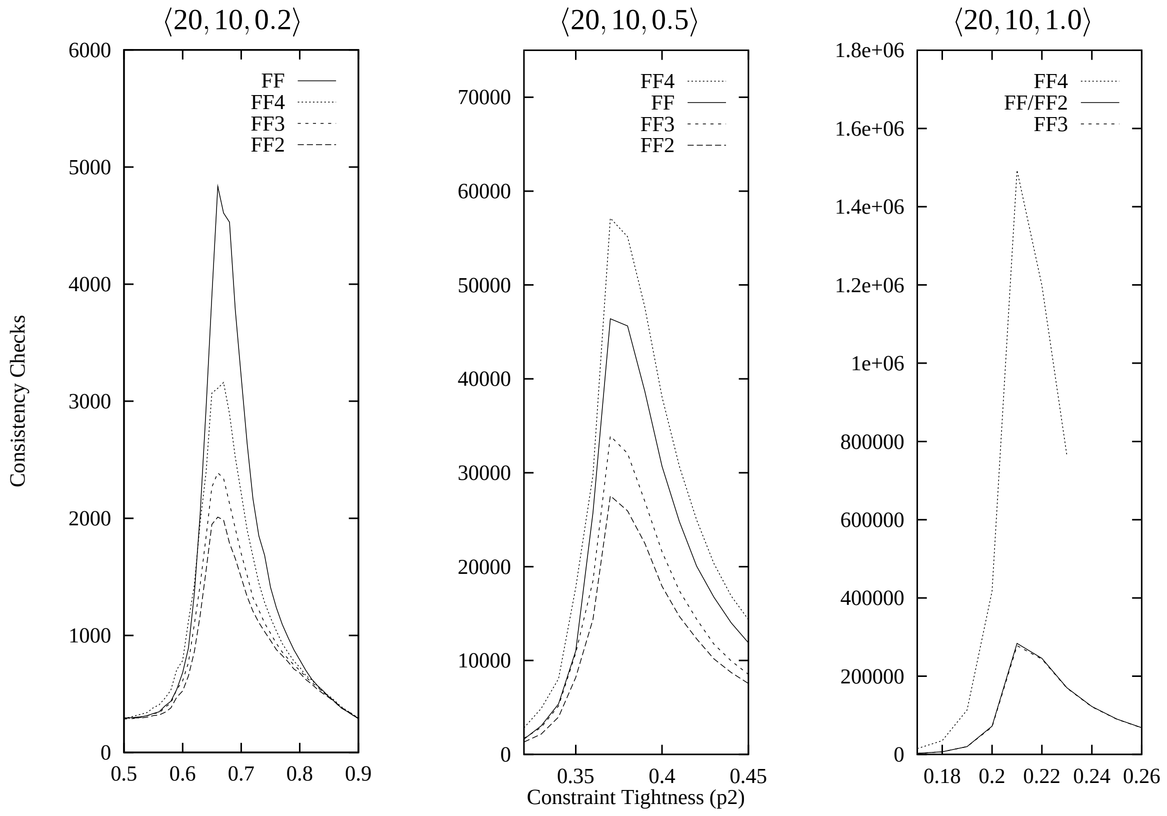


Figure 8.3: Fail-first heuristics on $\langle 20, 10\rangle$ CSPs.

8.6.1 New fail-first heuristics

Figure 8.3 plots the median consistency checking cost of FC-FF, FC-FF2, FC-FF3 and FC-FF4 on the $\langle 20, 10, 0.2\rangle$, $\langle 20, 10, 0.5\rangle$ and $\langle 20, 10, 1.0\rangle$ problem classes. Cost is plotted against constraint tightness, p_2, in each case and linear scales are used on both axes. In each plot, the key indicates the ranking of each heuristic at the phase transition.

In section 8.3, we predicted that we should see successive improvements in performance from the heuristics FF2, FF3 and FF4 compared to FF: FF4, which puts the greatest effort into accurately calculating the probability of failure for each variable should give the greatest improvement in search cost. The first feature we note from these plots, however, is the poor performance of the FF4 heuristic. It is consistently bad on all three classes, which is surprising and somewhat disappointing. In the case of the clique $\langle 20, 10, 1.0\rangle$ class, FC-FF4 is around six times more expensive at the phase transition than simple FC-FF. It is worth emphasising that this cost is the search cost, and does not include FF4's expensive probability calculations.

The FF4 heuristic attempts to 'literally' fail first by choosing the variable which has the highest probability of causing a future domain wipe-out. Its poor performance suggests that unless the probability model is wrong then the fail-first strategy is a flawed approach to DVO heuristics.

The original heuristic, FF, performs badly, though we observe in Section 8.5 that this can be attributed to the effectively random choice of starting variable on CSPs with uniform domain sizes.

FF3 does not perform as well as FF2, despite doing significantly more work. However, on

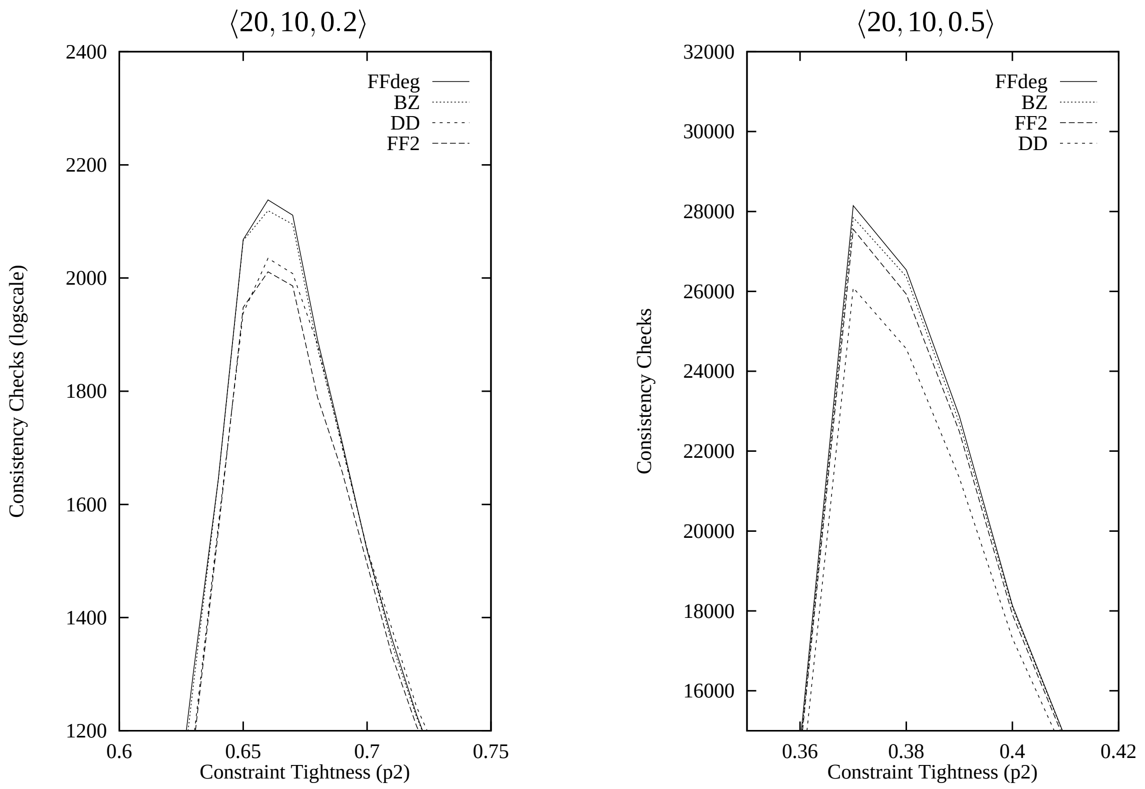


Figure 8.4: Four DVO heuristics on $\langle 20, 10\rangle$ CSPs.

the clique $\langle 20, 10, 1.0\rangle$ problems, FF3 gives a very slight improvement over FF2 and FF, which become identical when $p_1 = 1$. This appears to show us that there may be some value in taking into account the domain sizes of adjacent future variables, even if only when future degrees are identical and provide no leads.

The only one of the new heuristics which merits further consideration is FF2. As well as being the cheapest of them to implement, it is the best of the heuristics shown in Figure 8.3, except when $p_1 = 1$. In the following section we compare this heuristic with the others which take into account the degrees of variables (FFdeg, BZ and DD).

The plots in Figure 8.3 strongly suggest that putting more effort into choosing the variable which is most likely to fail does not pay off. This leads us to believe that the fail-first principle is not one which should be followed in designing variable ordering heuristics, and that the success of heuristics which are believed to implement this principle may be due to some other factor. This conclusion is discussed further in Section 8.11.

8.6.2 Heuristics based on domain size and degree

Figure 8.4 plots the median consistency checking cost of FC-FFdeg, FC-BZ, FC-DD and FC-FF2 on the $\langle 20, 10, 0.2\rangle$ and $\langle 20, 10, 0.5\rangle$ problem classes, in a similar fashion to Figure 8.3. These four heuristics use both the domain size and future degree (except DD, which uses global degree) in selecting the next variable. The $\langle 20, 10, 1.0\rangle$ class has been omitted since these heuristics are identical to FF when the constraint graph is complete.

The four heuristics give very similar performance. FF2 is marginally the best when $p_2 = 0.2$, but DD is better when $p_2 = 0.5$. It is noteworthy that the simplest of these heuristics, FFdeg, which uses degree information only in selecting the first variable, is competitive with the others. From Section 8.5 we know that this initial choice is a key factor in this heuristic's performance.

8.6.3 Failing first against minimising constrainedness

We noted in Section 8.1 that (Gent *et al.* 1996a) devise a number of new DVO heuristics which aim to minimise the constrainedness of the future subproblem during search. The effects of these new heuristics on forward checking searches are compared empirically with those of the fail-first heuristics FF and BZ. Since their studies use similar populations of $\langle 20, 10\rangle$ CSPs to those used here, we can compare both sets of results.

Figure 1 of (Gent *et al.* 1996a) plots the mean (not median) consistency checking costs of FC-FF, FC-BZ and FC-kappa, plus two other DVO heuristics which maximise the expected number of solutions, $E(N)$, and maximise the expected solution density, ρ. Heuristic kappa is beaten by BZ over the $\langle 20, 10, 0.2\rangle$ and $\langle 20, 10, 0.5\rangle$ classes, and we have shown above that both FF2 and DD in turn out-perform BZ on these CSPs.

On the clique $\langle 20, 10, 1.0\rangle$ problems, however, both $E(N)$ and kappa are significantly better than BZ. They will also be better than all of the heuristics examined here, none of which improve greatly upon BZ on this type of CSP.

8.7 Original Degree Versus Future Degree

The empirical study reported in Section 8.6 shows that detailed analysis of the subproblem of uninstantiated variables does not appear be effective in terms of delivering a good heuristic choice of variable to instantiate. The fail-first DVO heuristic that performs the most work on the subproblem, FF4, is comprehensively outperformed by much more naive heuristic methods.

Perhaps a lesson to be learned from the failure of FF4 is that it is a mistake to consider subproblems in isolation when considering search moves. It may be the case that heuristics should base some or all of their decisions on the *global* problem structure, and here we conduct a brief experiment to test this hypothesis.

The BZ and DD heuristics both perform well on the $\langle 20, 10\rangle$ CSPs studied above. While BZ considers the degree of variables in the future subproblem when breaking ties, DD uses the global degree of variables in its calculations. We have implemented alternative versions of these heuristics which use global and subproblem degrees respectively – heuristics BZg and DDs – and here compare their performance with BZ and DD. These variants have been reported before: BZg by (Frost and Dechter 1995), who use the term DVO; and DDs by (Bessière and Régin 1996), who investigate alternative versions of what they term dom/deg.

The fail-first principle suggests that BZg and DD should be worse than BZ and DDs respectively, on the kind of problems we are considering. Since the constraints are binary, the future subproblem is a self-contained CSP; the effect of the past variables on the future variables has already been accounted for in their domains, and the past variables have no further bearing on whether or

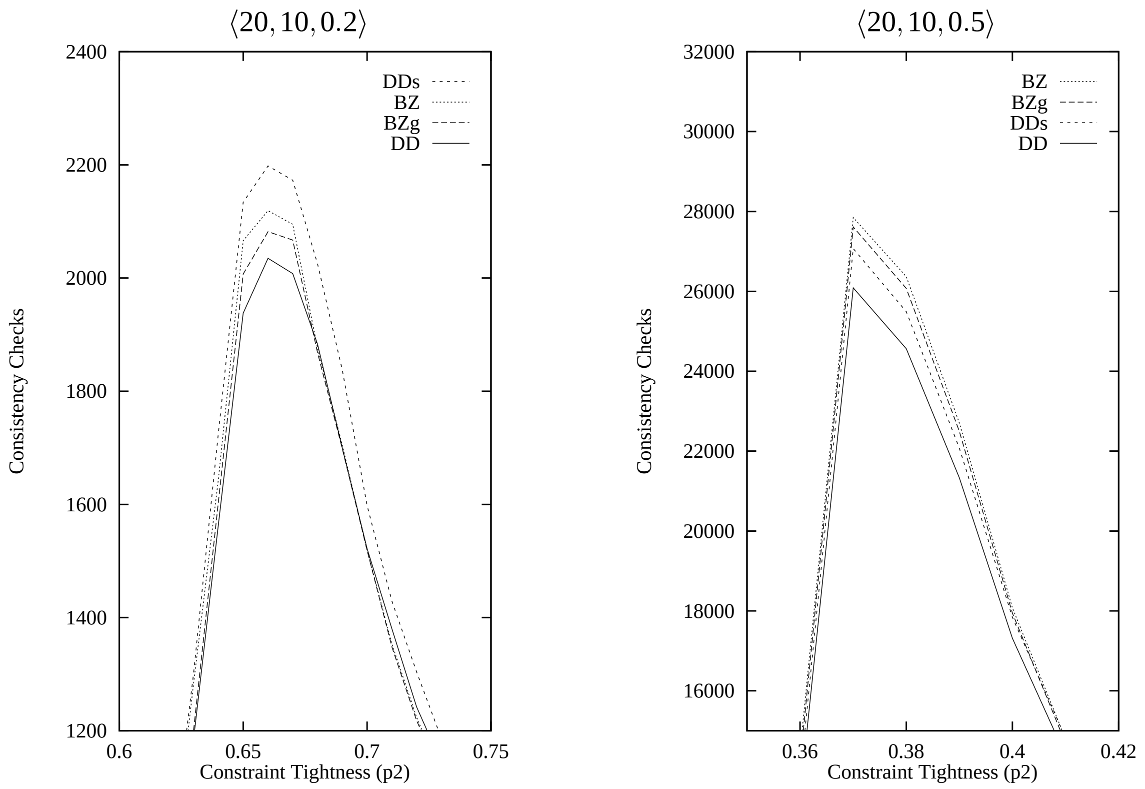


Figure 8.5: Variants of BZ and DD with FC on $\langle 20, 10\rangle$ CSPs.

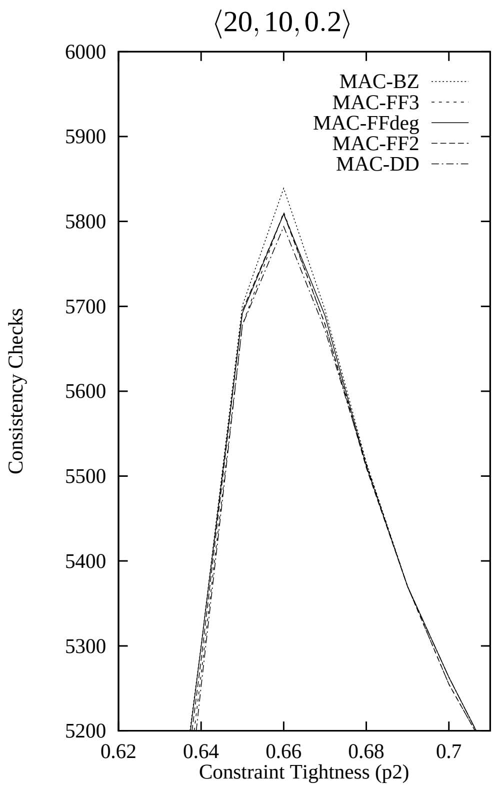
not a future variable will fail. The original degree is therefore less relevant to the probability of failure than the future degree is.

An experiment to compare the performance of the alternative versions of the BZ and DD heuristics was conducted, using the $\langle 20, 10, 0.2\rangle$ and $\langle 20, 10, 0.5\rangle$ problems used in Section 8.6. The $\langle 20, 10, 1.0\rangle$ problems were not used for the obvious reason that the degrees of variables offer no tie-breaking capability on clique CSPs.

Figure 8.5 plots the median consistency checking cost of FC-BZ, FC-BZg, FC-DD and FC-DDs on the $\langle 20, 10, 0.2\rangle$ and $\langle 20, 10, 0.5\rangle$ problem classes, in a similar fashion to the figures of Section 8.6. The behaviour that can be seen in these plots is rather surprising: in both cases the consistency checking cost of the global degree heuristic is better than that of the subproblem version, although the differences in cost are very small. On the $\langle 20, 10, 0.2\rangle$ problems, BZg becomes better than both BZ and DDs, with DD showing the lowest median cost at the phase transition. On the $\langle 20, 10, 0.2\rangle$ problems, BZg lies between BZ and DDs while DD remains the most effective heuristic.

The level of improvement experienced by using the global degrees of variables rather than the subproblem degrees is not particularly large in terms of consistency checks. However, the simplified heuristics remove a significant overhead associated with recalculating subproblem degrees, doing more useful work for less effort.

These results are consistent with the poor performance of the FF4 DVO heuristic, which looks in detail at the structure of the subproblem of a CSP at each search move. As a consequence, they raise further doubts about the wisdom of instantiating the variable that is most likely to lead to failure in the resultant subproblem.



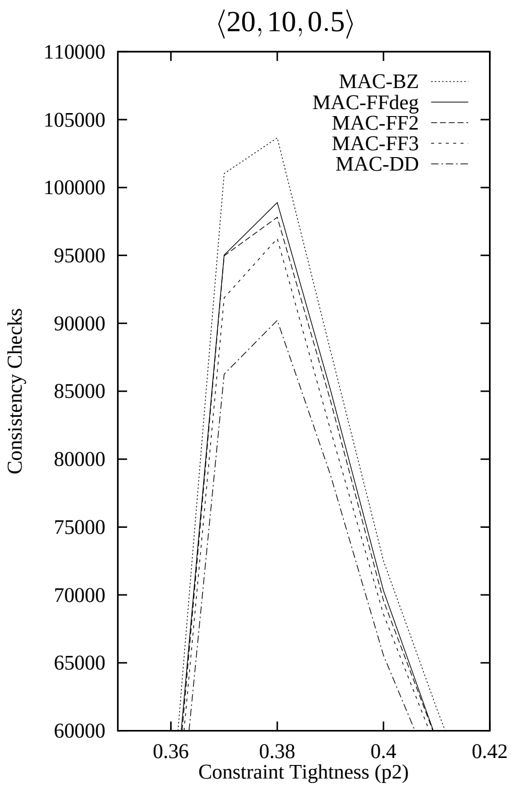


Figure 8.6: DVO heuristics with MAC on $\langle 20, 10\rangle$ CSPs.

If we consider BZ and BZg, which only refer to the degree information if more than one variable has the minimum current domain size, this must mean that sometimes it is better not to choose the variable which constrains the largest number of future variables, but instead one which is constrained by more past variables. This seems to be inexplicable in terms of the fail-first principle and appears to provide further evidence that this principle is not a good basis for dynamic variable ordering.

8.8 Changing Base Algorithm

The experiments of Section 8.6 were repeated, this time using the version of MAC studied in Chapter 7 as the base algorithm. Due to their lack of competitiveness, heuristics FF and FF4 were omitted from experimentation with MAC.

Figure 8.6 plots the median consistency checking cost of MAC-FFdeg, MAC-FF2, MAC-FF3, MAC-BZg and MAC-DD on the $\langle 20, 10, 0.2\rangle$ and $\langle 20, 10, 0.5\rangle$ problem classes. Cost is plotted against constraint tightness, p_2, in each case and linear scales are used on both axes. Once again, in each plot the key indicates the ranking of each heuristic at the phase transition. Data for $\langle 20, 10, 1.0\rangle$ CSPs is not presented due to the convergence of many of these heuristics on clique problems.

It can be seen that there is almost nothing between the different schemes over the $\langle 20, 10, 0.2\rangle$ problems. This is a little misleading, though, as it is known from Chapter 7 that MAC visits few nodes on problems of this size and constraint density.

On the $\langle 20, 10, 0.5\rangle$ problems, the costs of each scheme at the phase transition start to spread out more, enabling some ranking. The order of the heuristics is broadly consistent with that seen in Section 8.6 using FC, in that DD performs better than the new fail-first heuristics, which perform better than the BZ. Looking more closely, though, we observe that FF3 now out-performs FFdeg and FF2.

The results presented here do not add a great deal to our knowledge of the various DVO heuristics, although equally they do not significantly contradict what we have seen in Section 8.6. The main points that arise are that the extra look-ahead of the MAC algorithm may reduce the need for good heuristics, and that a scaling-up of these experiments to larger and harder CSPs is needed if a more consistent ranking of the heuristics is to be produced. We address the latter issue in the following section.

8.9 Scaling Behaviour of Heuristics

The empirical studies using the $\langle 20, 10\rangle$ CSPs provide a preliminary picture of the relative effectiveness of the DVO heuristics being studied. In order to add weight to these rankings, however, we must examine the scaling behaviour of these heuristics by testing them on larger CSPs.

Two new phase transition experiments were conducted, using the CSP classes $\langle 30, 10, 0.5\rangle$ and $\langle 50, 10, 0.1\rangle$. The former was chosen to test the heuristics on larger, medium density CSPs, while the latter is the large sparse class that has been studied extensively in Chapters 5 and 7. The size of each p_2 step for both classes is 0.01, and ensembles of 1,000 problems were generated at each $\langle n, 10, p_1, p_2\rangle$ point. The ensembles were then searched by both FC and MAC using the four 'best' heuristics from the studies on $\langle 20, 10\rangle$ problems.

Figure 8.7 plots the median consistency checking cost of FC on both classes of problem, using four DVO heuristics. Cost is plotted against constraint tightness, p_2, in each case and linear scales are used on both axes. As before, the key in each plot indicates the ranking of the heuristics at the phase transition.

This figure can be compared with Figure 8.4, as the $\langle 30, 10, 0.5\rangle$ problems are broadly similar in nature to $\langle 20, 10, 0.5\rangle$, as are the $\langle 50, 10, 0.1\rangle$ problems to $\langle 20, 10, 0.2\rangle$. In doing so, we observe that the ranking of the heuristics in both cases change little. The rankings for $\langle 30, 10, 0.5\rangle$ are the same as for $\langle 20, 10, 0.5\rangle$, with the gap between DD and the others showing some growth. The average cost of searches on the $\langle 50, 10, 0.1\rangle$ problems has spread out considerably compared to $\langle 20, 10, 0.2\rangle$, with DD the best heuristic by some margin. FC-DD searches are almost half as expensive as the next best, FC-FF2, and one-third as expensive as FC-FFdeg.

Figure 8.8 plots the median consistency checking cost of MAC on both classes of problem, using four DVO heuristics, in a similar fashion to Figure 8.7. This figure can be compared with Figure 8.6 as above. Once again, the ranking of heuristics is broadly preserved, although on the large sparse problems, FF3 out-performs FFdeg. On the $\langle 30, 10, 0.5\rangle$ problems, DD once again draws clear of the other heuristics, while on $\langle 50, 10, 0.1\rangle$ there is a broader spread with DD again the best.

The scaling-up of the empirical comparison of various DVO heuristics makes the overall picture clearer. Heuristic DD is consistently better than the others studied, on all CSPs except those

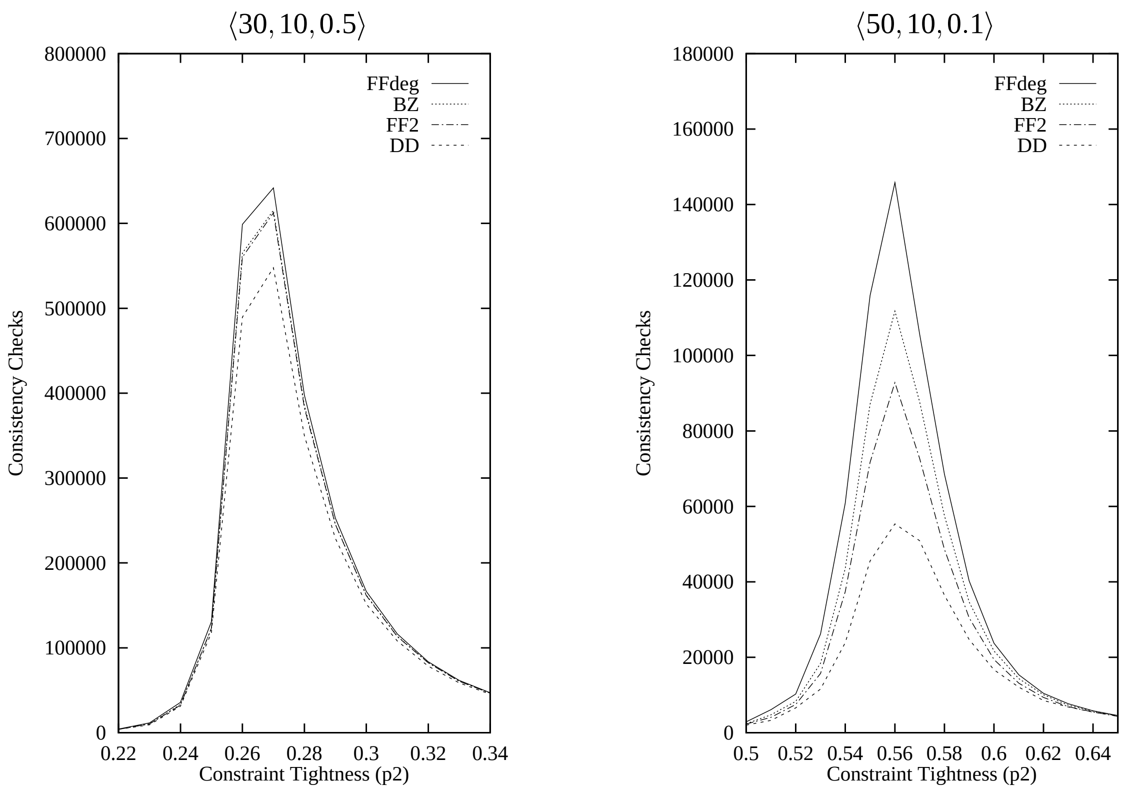


Figure 8.7: DVO heuristics with FC on $\langle 30, 10, 0.5\rangle$ and $\langle 50, 10, 0.1\rangle$ CSPs.

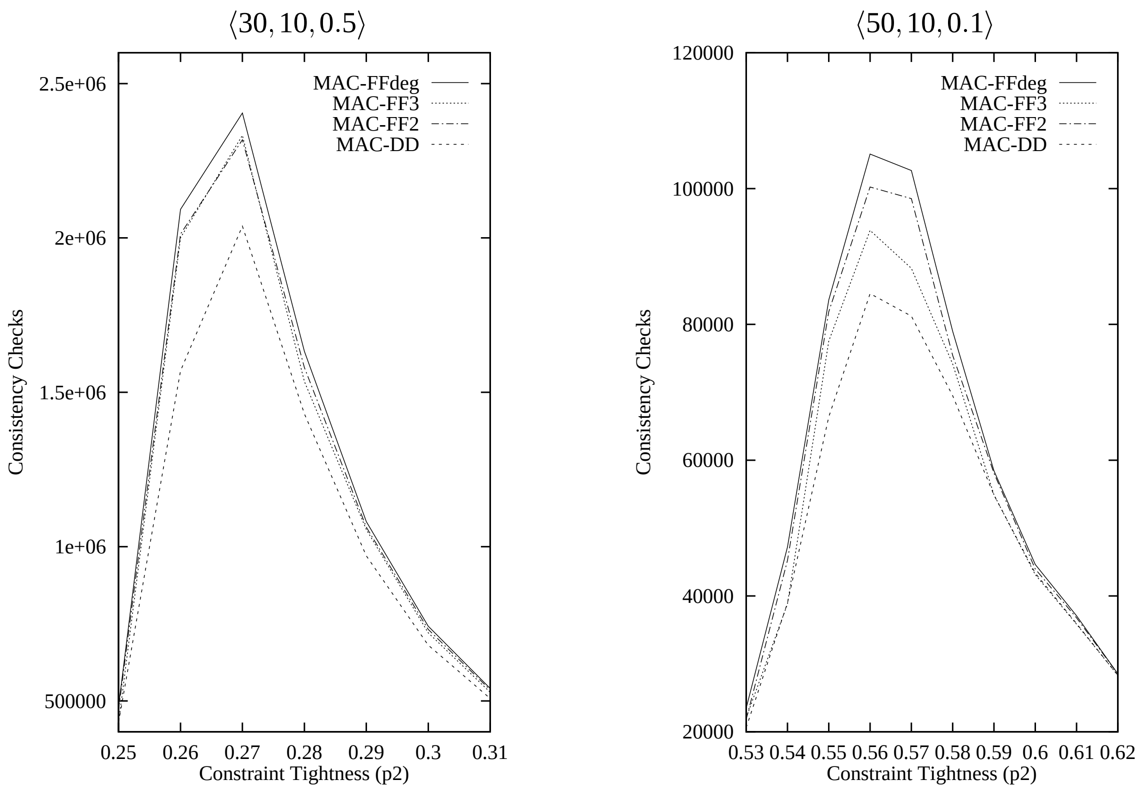


Figure 8.8: DVO heuristics with MAC on $\langle 30, 10, 0.5\rangle$ and $\langle 50, 10, 0.1\rangle$ CSPs.

that are very densely constrained. Its performance also scales well, and this has implications for the study of MAC presented in Chapter 7. Section 7.6.3 shows that the average consistency checking cost of MAC-FFdeg at the phase transition of the $\langle 50, 10, 0.1\rangle$ problem class is lower than that for FC-FFdeg. Using heuristic DD, the opposite becomes the case, and the point at which the cost of MAC falls below that of FC must now occur at some larger value of n, if at all. This exposes a shortcoming of the previous empirical study of MAC: the effect of the choice of DVO heuristic on the relative cost scaling of FC and MAC was not considered, and it was naive to use only one heuristic, FFdeg, in the experiments.

8.10 Comparative Ehp Behaviour

The relative incidence of exceptionally hard problem (ehp) behaviour is an important criterion in assessing the performance of an algorithm-heuristic combination. The studies of ehp behaviour presented in Chapters 5 and 7 investigate the use of only one DVO heuristic, FFdeg. We briefly consider whether the different flavours of DVO heuristic that have been examined in this chapter exhibit noticeably different patterns of ehp behaviour.

Figure 8.9 shows the median and higher percentiles of consistency checking cost for FC-FFdeg, FC-FF2, FC-BZ and FC-DD on the $\langle 50, 10, 0.1\rangle$ problem class examined earlier. Our existing knowledge of ehps teaches us to expect only sparsely constrained classes of CSP to contain ehp instances when forward checking is combined with dynamic variable ordering. As expected, ehp behaviour clearly occurs in the easy-soluble region for each DVO. Despite the relatively limited sample sizes of $1,000$ problems, searches which are orders of magnitude more expensive than the average are found. The most prominent ehp in Figure 8.9 is found for FC-DD, the scheme which shows the best average FC performance of those we have studied.

Overall, the patterns of ehp behaviour appear similar for all four heuristics. The affected regions of constraint tightness are identical, and we predict that with larger sample sizes, the incidence and magnitude of the most extreme cases would be broadly consistent. Although the introduction of a DVO heuristic eliminates ehp behaviour for FC on all but the most sparsely constrained classes of CSP (Chapter 5), we do not suspect that there is a good choice of ordering strategy that on its own can eliminate all ehps.

8.11 Consequences for the Fail-First Principle

This investigation into the nature of the fail-first principle was partly inspired by (Hooker and Vinay 1995), who conducted a similar investigation into branching rules for a well-known SAT algorithm, the Davis-Putnam procedure. We noted at the beginning of this chapter that Hooker and Vinay refuted the simple principle behind this heuristic (branching rules for SAT are analogous to variable ordering heuristics for CSP algorithms). They took the accepted explanation for one branching rule, derived a new branching rule which, assuming the explanation was correct, should have been superior to the original rule in terms of the number of nodes generated, and carried out an experiment to test this hypothesis. The new rule proved to be worse than the original,

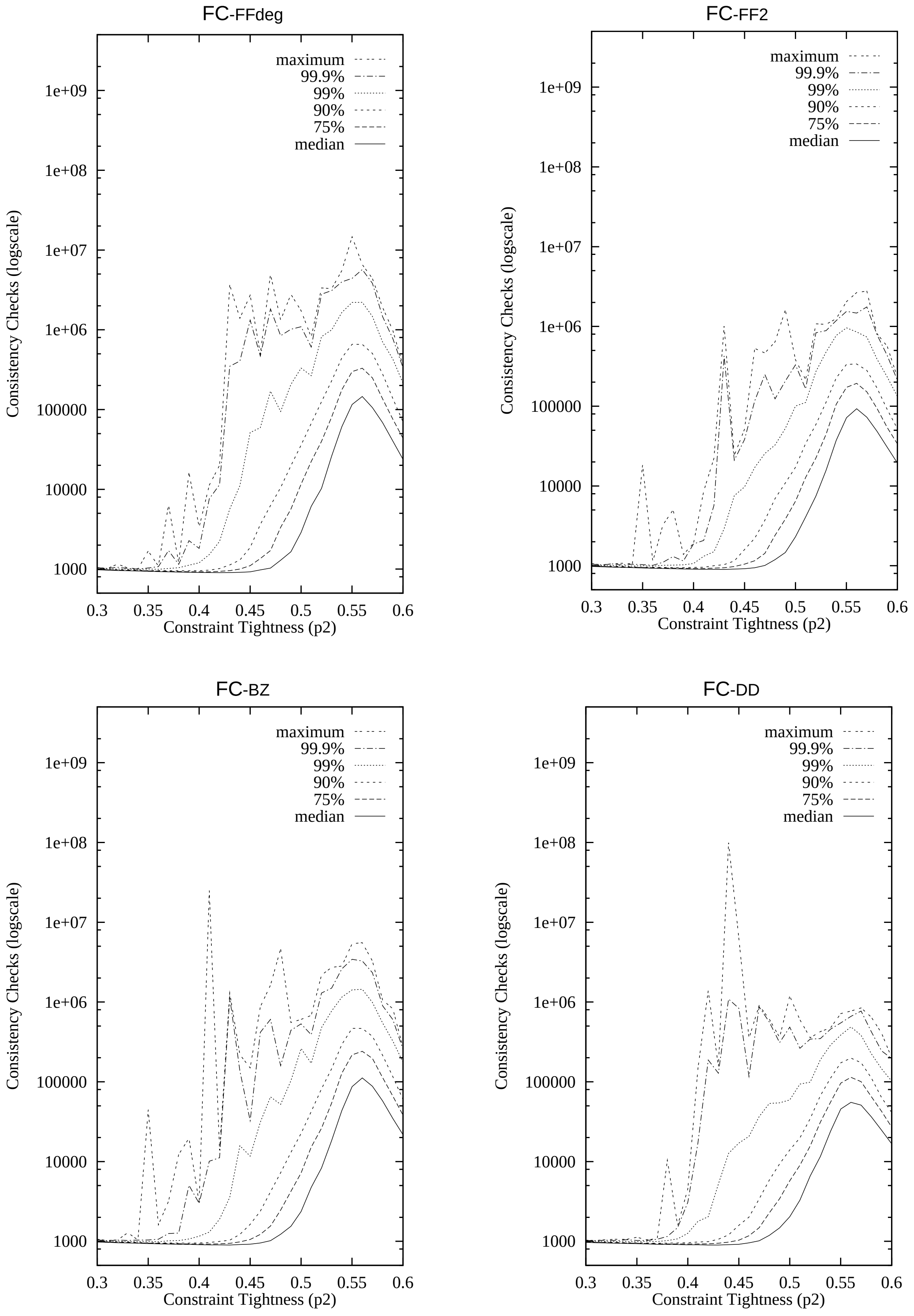


Figure 8.9: Relative ehp behaviour of four DVO heuristics on $\langle 50, 10, 0.1\rangle$ CSPs.

thus refuting the explanation as the true reason for the success of the original rule. They then went a stage further than the work presented here on fail-first does, by proposing an alternative explanation for the rule, making new predictions on the basis of this explanation and showing that these predictions were borne out by experiment. This provided evidence in favour of the new explanation, as well as giving a new branching rule which was superior to the original.

There are several remaining challenges, if our conclusion is correct that the fail-first principle is not a good basis for variable ordering. The first is to explain why it is not, especially since heuristics supposedly based on it have been successfully used since 1980. An alternative explanation for the relative performance of variable ordering heuristics is then needed. A sound principle for variable ordering, which can be used to make reliable predictions about the performance of proposed heuristics, would provide the means to develop new heuristics without resorting to trial and error, as is largely the case at present. This might also give us the basis for developing good general heuristics for problems with non-binary constraints: ‘smallest remaining domain’ is often not appropriate for such problems[4].

8.11.1 Suggested limitations of fail-first

Haralick and Elliott argued that a variable ordering heuristic should try to minimise the expected branch length in the search tree, and that this would be achieved by trying to choose a variable whose instantiation will lead to an immediate failure, assuming a search algorithm performing some form of lookahead. We suggest that the basic premise here is wrong: minimising expected branch length will not necessarily minimise the cost of searching the tree. Conceivably, a tree with many short branches could take longer to search than one with fewer longer branches.

Moreover, consider a hard insoluble problem in the phase transition, which is the kind of problem in which an improvement in the performance of variable ordering heuristics would have the greatest potential benefit. These problems are hard because it is not easy to discover that a set of assignments cannot lead to a solution. The cause of backtracking here is rarely due to the immediate failure of a variable instantiation; it more commonly follows the complete search of the subtrees resulting from each assignment. It is not clear, therefore, that there can be much benefit in maximising the probability of an event which can rarely occur.

We suggest that the reason why the new fail-first heuristics fail may be because they simply try to terminate the current branch as soon as possible and do not pay enough attention to what happens next, i.e. to backtracking. The FC and MAC algorithms backtrack chronologically, and if the previous variable has nothing to do with the failure, exploring alternative values for it will be a waste of effort. The previous variable is one of the culprits for the failure if there is a constraint between the previous and current variables and also the assignment to the previous variable reduced the domain of the current variable. To some extent the ‘smallest remaining domain’ heuristic tends to follow a chain of variables, each of which reduces the domain of the next, because the variable which now has the smallest domain tends to be one whose domain has just been reduced by the most recent assignment. However, this may not always be true,

[4]An article discussing variable ordering heuristics for n-ary CSPs can be found online in the Constraints Archive at `http://www.cirl.uoregon.edu/constraints/links/heuristics.html`.

particularly when the constraints are sparse. We suggest that heuristics which give some weight to the past degree of each variable in selecting the one to assign will do better when backtracking than heuristics which only take into account the future degree; if a variable is constrained by many past variables, the previous variable, or at least one of the recently instantiated variables, is likely to be one which has reduced the domain of the current variable. This would account for the fact that BZg and DD perform slightly better than BZ and DDs.

However, we make this suggestion tentatively; if the only reason for the failure of the fail-first principle is that it does not pay enough attention to what happens when the algorithm backtracks, we should expect that the proposed new heuristics would do well if used with an algorithm which backtracks more intelligently; in that case, the algorithm itself takes responsibility for backtracking to one of the culprits for the failure. Further empirical tests combining FF, FF2, FF3 and FF4 with FC-CBJ, however, record similar relative performances to those presented in Section 8.6.

8.11.2 An alternative strategy?

Although it is not entirely clear why the fail-first principle is not a sound basis for variable ordering, there is already an alternative explanation for the success of the ‘smallest remaining domain’ FF heuristic, in comparison with random ordering. (Nudel 1983) suggested that a good variable ordering heuristic is one which minimises the number of nodes visited in the search tree. He developed expressions for N_k, the expected number of nodes visited at level k in the tree when finding all solutions to a CSP using FC. It is suggested that the next variable selected should be the one minimising N_1, the number of nodes at the top level of the new subtree, and N_2, etc., in the case of ties. Since N_1 is equal to the size of the first variable’s domain, this implies that that at each level in the tree we should next choose the variable with smallest remaining domain. Although the expression for N_2 is more complex, for $\langle n,m,p_1,p_2\rangle$ CSPs to which no values have yet been assigned, it reduces to a simpler form which implies selection of the first variable as that with largest degree. This implies the FFdeg heuristic.

It is possible that minimising the expected number of nodes visited will prove to be a more robust principle for variable ordering than the fail-first principle. However, using this idea to generate better heuristics than FFdeg is not straightforward. Minimising N_1 and then N_1+N_2 is only an approximation to minimising the expected total number of nodes visited, but calculating $N_1+N_2+..+N_n$ can only be done for a specific ordering of all the variables. In theory we should calculate this quantity for every possible ordering, assign the first variable in the best ordering, repeat the process for the remaining $n-1$ variables and so on. Clearly this is not practicable, but we believe that Nudel’s theory could even so be the basis for deriving new heuristics for binary CSPs. If that enterprise were successful, it might also be possible to use the same principle to derive heuristics for non-binary problems as well.

8.12 Future Directions for DVO Heuristics

The heuristics which attempt to minimise the constrainedness of the future subproblems, proposed by (Gent *et al.* 1996a), offer an alternative principle for devising variable ordering heuristics.

They report the kappa heuristic producing FC searches which are on average over 10% more efficient than those using fail-first heuristics on $\langle 20, 10, 1.0\rangle$ CSPs. However, its performance on the more sparsely constrained CSPs is somewhat worse than heuristics BZ and DD, which do not support minimising constrainedness as a general principle. Moreover, the kappa heuristic, like FF4, requires the tightness of the constraints between future variables to be updated after each assignment, and the potentially high cost of doing this was not accounted for. Despite this, its good performance on the clique $\langle 20, 10, 1.0\rangle$ CSPs, where most available heuristics degenerate to FF (or, in the case of FF4, are much worse) makes it noteworthy.

Overall, the development of good new variable ordering heuristics would be considerably simplified if they could be based on an underlying principle which was:

- sound, in that the greater its degree of application, the better the resulting heuristic; and
- applicable, in that cheap heuristics which make some level of assumption can be used.

The investigation of heuristics based on fail-first leads us to believe that this principle is unsound. However, given the array of successful heuristics which have been designed to follow the fail-first principle, it could be said to be applicable.

8.13 Summary

The major conclusion from this investigation is that the fail-first principle is not a sound basis for variable ordering heuristics for CSPs. We calculated the probability that a variable will fail when using a lookahead search algorithm to solve binary CSPs. According to the fail-first principle, choosing the variable which maximises this probability will tend to minimise the size of the search tree explored by the algorithm. A series of new heuristics was derived, based on increasingly accurate estimates of this probability, and it was predicted that if the fail-first principle is sound, the more accurate the estimate the better the performance should be. The new heuristics, among others, were then studied empirically using both the FC and MAC algorithms on large, diverse classes of $\langle n, m, p_1, p_2\rangle$ CSPs. These showed that the predictions about the new heuristics were not borne out.

Further evidence against the fail-first principle is provided by the superiority of the BZg and DD heuristics, which consider the 'global' degrees of variables, over BZ and DDs which calculate subproblem degrees. Other things being equal, the future degree of a variable should be a better indication of the likelihood that it will fail than its original degree, so that according to the fail-first principle BZ and DDs should out-perform BZg and DD.

In spite of the poor results for the new fail-first heuristics, an existing heuristic, DD, has been shown to perform extremely well on many classes of CSP, except where the constraint graph is fully connected, in which case in degenerates to FF. The scaling of this heuristic's effectiveness on FC and MAC searches is also superior than the other fail-first heuristics examined, including FFdeg, which has been used in the empirical studies of Chapters 5 and 7. The scaling behaviour of heuristics rather than algorithms was not considered before, particularly in Chapter 7, whose results would be affected somewhat by the use of the DD heuristic.

We tentatively suggest that the reason why trying harder to fail first does not pay off may be because taking into account other factors such as the current constraint tightness and the current domains of adjacent variables may make the ‘improved’ heuristics less likely to choose a variable whose domain was reduced by the previous variable. This may cause unnecessary work to be done on backtracking. Three alternative principles for designing variable ordering strategies have been discussed, although these too suffer from limitations in terms of their practical application.

The current position of designing variable ordering heuristics appears somewhat precarious. If ‘failing-first’ is not in fact a sound principle, designing heuristics based on this principle will only work by accident, if they also happen to be an implementation of a better principle. Until a reliable theory exists, supported by empirical results, which can be used to predict the performance of different heuristics on different types of problem, progress in the development of improved heuristics will continue to be made only by trial and error.

8.14 Acknowledgements

This study of dynamic variable ordering heuristics has been conducted jointly with Barbara Smith, who is responsible for the probabilistic analysis of the fail-first principle, presented in Section 8.3, and the discussion on Nudel’s theory, presented in Section 8.11.

Chapter 9

Conclusions

The major results and contributions of the studies presented in this thesis are discussed below. A note on the limitations of these studies is then followed by a summary of the future work that has been suggested in Chapters 5 to 8.

9.1 Contributions

A basic complete search algorithm for a CSP combines some form of forward and backward search move. In addition, a preprocessing phase may be added to enforce some level of consistency in the problem before search, and dynamic variable ordering may be used to improve the efficiency of the search. The choice of a particular search algorithm for a type of problem is made easier by knowledge of its performance over a wide range of problem types, and its relative susceptibility to unexpected behaviour. Every one of these aspects relating to complete search algorithms has been examined in detail through rigourous empirical study.

9.1.1 Experimental methodology

The empirical studies presented throughout have been designed with the call by (Hooker 1994) for an ‘empirical science of algorithms’ in mind. The implementation of the search algorithms was discussed in detail in Chapter 2, and methods of random problem generation were explained in Chapter 3. A generic framework for the experiments was laid out in Chapter 4, and a consistent nomenclature for the description of problems and algorithms has been adopted throughout. The software used to conduct the empirical studies has also been made available via the World Wide Web, as detailed in Appendix A.

9.1.2 Exceptionally hard problems

Chapter 5 examined the nature and relative incidence of exceptionally hard problem (ehp) behaviour for a number of complete CSP search algorithms. The study looked at ehps at both a ‘macroscopic’ level, considering their incidence over populations of problems, and at a ‘microscopic’ level, examining in detail the search process for many individual instances.

An explanation for the occurrence of these exceptional searches was proposed: that the first few variable instantiations lead to the creation of a subproblem which is insoluble and causes the algorithm to thrash. Although it had previously been suggested that these problems are not inherently difficult, the exact cause of ehp behaviour in CSPs had not been examined in depth before.

The effect on ehp behaviour of more advanced forms of forward and backward search move were then studied empirically. Maintaining consistency in the set of future variables was shown to deal with ehps resulting from subproblems with lower levels of inconsistency. It was also shown that intelligent backjumping can be effective in quickly searching subproblems with higher levels of inconsistency. A significant result was that the use of ‘fail-first’ dynamic variable ordering appears to eliminate ehp behaviour from more densely-constrained classes of CSP.

9.1.3 Phase transitions in polynomial problems

Chapter 6 demonstrated that phase transition methodology can be applied to the polynomial-complexity computational tasks of establishing arc and path consistency in CSPs. It was shown that these tasks exhibit phase transition behaviour very much analogous to that associated with the NP-complete task of finding solutions to these problems. This analogy had not been made before.

Arc and path consistency were also considered as preprocessing steps before search. As had been shown elsewhere, they are generally ineffective in the hard problem regions. However, we derived an expression for the expected cost of the AC3 arc consistency procedure on problems in these regions which showed it to be inexpensive. It was also shown that around the AC phase transition peak, both the average cost and maximum cost of AC3 grow at a rate that is significantly slower than the worst-case analysis suggests, supporting previous similar claims.

9.1.4 Positioning of the MAC algorithm

Chapter 7 positioned the behaviour of two algorithms which maintain arc consistency during search with respect to two which perform a lower level of lookahead. The empirical study involved the search of over five million CSPs of varying size, topology and expected difficulty by the MAC, MAC-CBJ, FC and FC-CBJ algorithms.

As had been observed in previous studies, the extra lookahead of MAC results in smaller search trees and more frequent instances of backtrack-free search. The consistency checking cost of MAC was shown to be poor on smaller classes of CSP, again in agreement with previous studies. However, by studying the independent effects of altering problem size and topology it was shown that as problem size increases, the consistency checking cost of MAC scales at a much more favourable rate than that for FC, and so MAC becomes significantly cheaper on large sparse CSPs. Study of the effects of combining MAC with CBJ showed that little benefit was gained, except for problems that MAC found exceptionally hard. MAC-CBJ showed the most stable performance in terms of ehp behaviour of all the algorithm that have been studied here, although a few examples were still observed.

9.1.5 Scrutiny of the fail-first principle

Chapter 8 studied the ‘fail-first’ principle as a basis for the design of dynamic variable ordering heuristics, and concluded that it is not sound for this purpose. A probabilistic model of the principle (provided by Barbara Smith) was used to derive a series of new heuristics which applied the principle more accurately and so should have led to more efficient search. Empirical study of these heuristics, and several others, on large, diverse classes of CSPs showed that the predictions about the new heuristics were not borne out. Further evidence against the fail-first principle was also provided by the apparent superiority of simplified versions of two existing heuristics which should have deteriorated search efficiency.

9.2 Limitations of these Studies

The scope of this thesis has been restricted to the study of complete search methods applied to random binary CSPs. While the focus on complete methods is an acceptable restriction, the use of relatively unstructured ensembles of test problems hinders the applicability of the results observed to ‘real world’ constraint satisfaction problems. We have already suggested that problems with more structured constraint graphs, varying domain sizes and/or individual constraint tightnesses are likely to affect the performance of techniques to establish consistency (Chapter 6), and it is also expected that they would have an impact on the effectiveness of variable ordering strategies and backjumping techniques.

A first step towards more structured random problems would be the use of an extended generation model for binary CSPs, such as that considered in Section 3.7. This model, used by (Gent *et al.* 1996a), allows for variation of individual constraint tightnesses and domain sizes. As explained in Section 3.7, these extensions were not used here due to the increase in the complexity of the empirical studies that would have arisen from introducing two extra degrees of freedom.

Ultimately, the need to expose search techniques to ‘real world’ situations will require the consideration of non-binary CSPs. At present, however, a method of generating non-binary CSPs with interesting properties in sufficient quantities is not readily available. A move to non-binary constraints would also require re-implementation of the search algorithms used.

9.3 Future Work

Within the scope of complete search algorithms and binary CSPs, the studies reported in Chapters 5 to 8 suggest a series of future studies. These are summarised below.

The role of dynamic variable ordering in eliminating ehp behaviour on densely-constrained CSPs (Chapter 5) needs to be examined in more detail. Why this technique is effective for dense problems but not for sparse problems is still unclear. Investigation into the existence of a ‘double’ peak in the higher percentiles of search cost for CSPs, similar to that observed by (Hogg and

Williams 1994) in graph colouring, has been started by (Smith and Grant 1997). This analysis currently considers only the BT algorithm on clique CSPs, but it is expected that this work will be extended to other algorithms and wider varieties of problem.

The work on phase transition behaviour in arc and path consistency (Chapter 6) has already been followed up by (Gent *et al.* 1997b), who introduce the new constrainedness parameter, κ_{ac}. The development of a similar parameter to predict the location of the path consistency phase transition remains an open task. Section 6.8 also suggests that the AC and PC phase transitions could be useful in the search for a CSP 'constraint gap', which might explain the existence of ehps.

A shortcoming of the study of MAC (Chapter 7) is that more efficient arc consistency techniques exist, which could produce more efficient MAC searches. Subsequent studies using different implementations of MAC (for instance (Bessière and Régin 1996)) appear to confirm this. The behaviour of the AC3-based MAC is still of some interest, however, and further investigation of the scaling relationship between it and FC on sparse CSPs is desirable. Chapter 8, for instance, suggests that the choice of DVO heuristic used by the algorithms can affect this relationship.

The examination of heuristics for dynamic variable ordering (Chapter 8) leaves open a challenge to develop a principle which properly explains the usefulness of techniques assumed to follow the 'fail-first' principle. If a new principle is devised which is both sound and applicable, then this should lead to the development of improved search heuristics.

Appendix A

An Online CSP Experimentation Laboratory

Software capable of reproducing the experiments reported throughout this thesis is available on the World Wide Web at `http://www.scs.leeds.ac.uk/csps/`. The contents of this page include:

- An online copy of this thesis and other publications which have used the software.
- A pseudo-random problem generator module, implemented as a C++ class.
- A set of binary executable programs incorporating problem generation, search and statistical collection within a variety of experimental environments. The source language is C++, and executables are provided for use under a number of popular operating systems.
- Instructions for performing phase transition experiments using this software.
- Links to interesting CSP research pages.
- Links to other CSP software repositories.

References

J. F. Allen. 1983. Maintaining knowledge about temporal intervals. *Communications of the ACM* 26(11):832–843.

F. Bacchus and P. van Run. 1995. Dynamic Variable Ordering in CSPs. In U. Montinari and F. Rossi., eds., *Principles and Practice of Constraint Programming - CP-95*, volume 976 of *Lecture Notes in Computer Science*, 258–275. Springer-Verlag.

A. B. Baker. 1995. *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results*. Ph.D. Dissertation, Department of Computer and Information Science, University of Oregon.

C. Bessière and J.-C. Régin. 1995. Using Bidirectionality to Speed Up Arc-Consistency Processing. In M. Meyer., ed., *Constraint Processing*, volume 923 of *Lecture Notes in Computer Science*. Springer-Verlag. 157–169. Proceedings of the ECAI-94 Workshop on Constraint Processing.

C. Bessière and J.-C. Régin. 1996. MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems. In *Principles and Practice of Constraint Programming - CP-96*, volume 1118 of *Lecture Notes in Computer Science*, 61–75. Springer-Verlag.

C. Bessière, E. C. Freuder and J.-C. Régin. 1995. Using Inference to Reduce Arc Consistency Computation. In C. S. Mellish., ed., *Proceedings IJCAI-95*, volume 1, 592–598. Morgan Kaufmann.

C. Bessière. 1994. Arc consistency and arc consistency again. *Artificial Intelligence* 65:179–190.

J. R. Bitner and E. M. Reingold. 1975. Backtrack Programming Techniques. *Communications of the ACM* 18:651–656.

B. Bollobas. 1985. *Random Graphs*. Academic Press.

J. E. Borrett and E. P. K. Tsang. 1995. Observations on the usefulness of arc consistency preprocessing. Technical Report CSM-236, Department of Computer Science, University of Essex, UK.

J. E. Borrett, E. P. K. Tsang and N. R. Walsh. 1996. Adaptive Constraint Satisfaction: The Quickest First Principle. In W. Wahlster., ed., *Proceedings ECAI-96*, 160–164. John Wiley & Sons, Ltd.

D. Brélaz. 1979. New methods to color the vertices of a graph. *Communications of the ACM* 22(4):251–256.

P. Cheeseman, B. Kanefsky and W. Taylor. 1991. Where the Really Hard Problems are. In *Proceedings IJCAI-91*, volume 1, 331–337. Morgan Kaufmann.

S. A. Cook. 1971. The Complexity of Theorem-Proving Procedures. In *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing*, 151–158.

M. C. Cooper. 1989. An Optimal k-Consistency Algorithm. *Artificial Intelligence* 41:89–95.

J. M. Crawford and L. D. Auton. 1996. Experimental Results on the Crossover Point in Satisfiability Problems. *Artificial Intelligence* 81:31–57. Special Issue on Frontiers in Problem Solving: Phase Transitions and Complexity.

A. Davenport and E. P. K. Tsang. 1995. An empirical investigation into the exceptionally hard problems. Technical Report CSM-239, Department of Computer Science, University of Essex, U.K.

J. de Kleer. 1986. An Assumption-Based TMS. *Artificial Intelligence* 28:127–162.

J. de Kleer. 1989. A comparison of ATMS and CSP techniques. In *Proceedings IJCAI-89*, volume 1, 290–296. Morgan Kaufmann.

R. Dechter and I. Meiri. 1994. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence* 68(2):211–242.

R. Dechter and J. Pearl. 1988. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence* 34:1–38.

R. Dechter and J. Pearl. 1989. Tree clustering for constraint networks. *Artificial Intelligence* 38:353–366.

R. Dechter. 1990. Enhancement Schemes for Constraint Processing: Backjumping, Learning and Cutset Decomposition. *Artificial Intelligence* 41:273–312.

M. S. Fox. 1987. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling*. Morgan-Kaufmann.

E. Freuder and R. Wallace. 1992. Partial constraint satisfaction. *Artificial Intelligence* 58:21–70.

E. Freuder. 1982. A Sufficient Condition for Backtrack-Free Search. *Journal of the ACM* 29(1):24–32.

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

D. Frost and R. Dechter. 1995. Look-ahead Value Ordering for Constraint Satisfaction Problems. In C. S. Mellish., ed., *Proceedings IJCAI-95*, volume 1, 572–578. Morgan Kaufmann.

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

J. Gaschnig. 1977. A General Backtrack Algorithm that Eliminates Most Redundant Tests. In *Proceedings IJCAI-77*, volume 1, 457. Morgan Kaufmann.

J. Gaschnig. 1979. Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Carnegie-Mellon University, Pittsburgh USA.

I. P. Gent and T. Walsh. 1994a. Easy Problems are Sometimes Hard. *Artificial Intelligence* 70:335–345.

I. P. Gent and T. Walsh. 1994b. The Hardest Random SAT Problems. In B. Nebel and L. Dreschler-Fischer., eds., *Proceedings KI-94: Advances in Artificial Intelligence. 18th German Annual Conference on Artificial Intelligence*, 355–366. Springer-Verlag.

I. P. Gent and T. Walsh. 1995a. Computational Phase Transitions from Real Problems. In *Proceedings of the 8th International Symposium on AI (ISAI-95)*, 356–364.

I. P. Gent and T. Walsh. 1995b. The TSP Phase Transition. Technical Report 178-95, Department of Computer Science, University of Strathclyde, UK. Presented at the 1st International Workshop on AI and OR, Timberline, Oregon, June 1995.

I. P. Gent and T. Walsh. 1996a. Phase Transitions and Annealed Theories: Number Partitioning as a Case Study. In W. Wahlster., ed., *Proceedings ECAI-96*, 170–174. John Wiley & Sons, Ltd.

I. P. Gent and T. Walsh. 1996b. The Satisfiability Constraint Gap. *Artificial Intelligence* 81:59–80. Special issue on Frontiers in Problem Solving: Phase Transitions and Complexity.

I. P. Gent, E. MacIntyre, P. Prosser and T. Walsh. 1995. Scaling Effects in the CSP Phase Transition. In U. Montinari and F. Rossi., eds., *Principles and Practice of Constraint Programming - CP-95*, volume 976 of *Lecture Notes in Computer Science*, 70–87. Springer-Verlag.

I. P. Gent, E. MacIntyre, P. Prosser, B. M. Smith and T. Walsh. 1996a. An Empirical Study of Dynamic Variable Ordering Heuristics for the Constraint Satisfaction Problem. In E. C. Freuder., ed., *Principles and Practice of Constraint Programming - CP-96*, volume 1118 of *Lecture Notes in Computer Science*, 179–193. Springer-Verlag.

I. P. Gent, E. MacIntyre, P. Prosser and T. Walsh. 1996b. The Constrainedness of Search. In *Proceedings AAAI-96*, volume 1, 246–252. AAAI Press.

I. P. Gent, S. A. Grant, E. MacIntyre, P. Prosser, P. Shaw, B. Smith and T. Walsh. 1997a. How Not To Do It. Research Report 97.27, School of Computer Studies, University of Leeds.

I. P. Gent, E. MacIntyre, P. Prosser, P. Shaw and T. Walsh. 1997b. The Constrainedness of Arc Consistency. In *Principles and Practice of Constraint Programming - CP-97 (to appear)*, Lecture Notes in Computer Science, 328–341. Springer-Verlag.

N. E. Gibbs, W. G. Poole and P. K. Stockmeyer. 1976. An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix. *SIAM Journal of Numerical Analysis* 13:236–250.

S. W. Golomb and L. D. Baumert. 1965. Backtrack Programming. *Journal of the ACM* 12:516–524.

C. P. Gomes, B. Selman and N. Crato. 1997. Heavy-Tailed Distributions in Combinatorial Search. In *Principles and Practice of Constraint Programming - CP-97 (to appear)*, Lecture Notes in Computer Science, 121–135. Springer-Verlag.

S. A. Grant and B. M. Smith. 1995. The Phase Transition Behaviour of Maintaining Arc Consistency. Research Report 95.25, School of Computer Studies, University of Leeds.

S. A. Grant and B. M. Smith. 1996a. The Arc and Path Consistency Phase Transitions. In E. C. Freuder., ed., *Principles and Practice of Constraint Programming - CP-96*, Lecture Notes in Computer Science, 541–542. Springer-Verlag. Extended abstract with poster presentation.

S. A. Grant and B. M. Smith. 1996b. The Arc and Path Consistency Phase Transitions. Research Report 96.09, School of Computer Studies, University of Leeds.

S. A. Grant and B. M. Smith. 1996c. The Phase Transition Behaviour of Maintaining Arc Consistency. In W. Wahlster., ed., *Proceedings ECAI-96*, 175–179. John Wiley & Sons, Ltd.

S. A. Grant. 1994. Cooperative Search. Final Year Report for the degree of B.Sc. in Computer Science, Department of Computer Science, University of Strathclyde.

C.-C. Han and C.-H. Lee. 1988. Comments on mohr and henderson's path consistency algorithm. *Artificial Intelligence* 36:125–130.

R. Haralick and G. Elliott. 1980. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence* 14:263–313.

T. Hogg and C. P. Williams. 1994. The Hardest Constraint Problems: A Double Phase Transition. *Artificial Intelligence* 69:359–377.

J. N. Hooker and V. Vinay. 1995. Branching Rules for Satisfiability. *Journal of Automated Reasoning* 15:359–383.

J. N. Hooker. 1994. Needed: An Empirical Science of Algorithms. *Operations Research* 42(2):201–212.

J. N. Hooker. 1996. Testing Heuristics: We Have It All Wrong. *Journal of Heuristics* 1:33–42.

D. S. Johnson. 1996. A Theoretician's Guide to the Experimental Analysis of Algorithms. Invited talk at AAAI-96. Partial draft available at on World Wide Web at `http://www.research.att.com/∼dsj/papers/exper.ps`.

H. Kautz and B. Selman. 1992. Planning as Satisfiability. In *Proceedings ECAI-92*, 359–363. John Wiley & Sons, Ltd.

S. Kirkpatrick and B. Selman. 1994. Critical Behaviour in the Satisfiability of Random Boolean Expressions. *Science* 264:1297–1301.

D. E. Knuth. 1981. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison-Wesley.

G. Kondrak and P. van Beek. 1997. A Theoretical Evaluation of Selected Backtracking Algorithms. *Artificial Intelligence* 89:365–387.

G. Kondrak. 1994. A Theoretical Evaluation of Selected Backtracking Algorithms. Master's thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada.

V. Kumar. 1992. Algorithms for Constraint Satisfaction Problems: A Survey. *AI Magazine* 13(1):32–44.

A. C. M. Kwan, E. P. K. Tsang and J. Borrett. 1995. Phase Transitions in Finding Multiple Solutions in Constraint Satisfaction Problems. In *Proceedings of the CP-95 Workshop on Studying and Solving Really Hard Problems*, 119–129. Laboratoire d'Informatique de Marseille, France.

A. K. Mackworth and E. C. Freuder. 1985. The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence* 25:65–74.

A. Mackworth. 1977. Consistency in networks of relations. *Artificial Intelligence* 8:99–118.

P. Meseguer. 1989. Constraint satisfaction problems : an overview. *AI Communications* 2:3–17.

S. Minton, M. D. Johnston, A. B. Philips and P. Laird. 1992. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence* 58:161–205.

D. G. Mitchell and H. J. Levesque. 1996. Some Pitfalls for Experimenters with Random SAT. *Artificial Intelligence* 81:111–125. Special Issue on Frontiers in Problem Solving: Phase Transitions and Complexity.

D. Mitchell, B. Selman and H. Levesque. 1992. Hard and Easy Distributions of SAT Problems. In *Proceedings AAAI-92*, 459–465.

S. Mittal and B. Falkenhainer. 1990. Dynamic Constraint Satisfaction Problems. In *Proceedings AAAI-90*, 25–32.

R. Mohr and T. Henderson. 1986. Arc and path consistency revisited. *Artificial Intelligence* 28:225–233.

U. Montanari. 1974. Networks of Constraints: Fundamental Properties and Applications to Image Processing. *Information Science* 7:95–132.

B. Nadel. 1989. Constraint satisfaction algorithms. *Computational Intelligence* 5:188–224.

D. Navinchandra and D. H. Marks. 1987. Design Exploration Through constraint Relaxation. In *Expert Systems in Computer-Aided Design*. Elsevier Science Publishers.

B. Nudel. 1983. Consistent-Labeling Problems and their Algorithms: Expected-Complexities and Theory-Based Heuristics. *Artificial Intelligence* 21:135–178.

E. M. Palmer. 1985. *Graphical Evolution*. New York: Wiley.

S. K. Park and K. W. Miller. 1988. Random Number Generators: Good Ones are Hard to Find. *Communications of the ACM* 31(10):1192–1201.

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

P. Prosser. 1994. Binary constraint satisfaction problems: some are harder than others. In A. G. Cohn., ed., *Proceedings of ECAI-94*, 95–99. John Wiley & Sons, Ltd.

P. Prosser. 1995. MAC-CBJ: maintaining arc consistency with conflict-directed backjumping. Technical Report 95-177, Department of Computer Science, University of Strathclyde, UK.

P. Prosser. 1996. An Empirical Study of Phase Transitions in Binary Constraint Satisfaction Problems. *Artificial Intelligence* 81:81–109. Special Issue on Frontiers in Problem Solving: Phase Transitions and Complexity.

J.-F. Puget. 1994. A C++ Implementation of CLP. In *Proceedings of SPICIS-94 (Singapore International Conference on Intelligent Systems)*.

P. W. Purdom. 1983. Search Rearrangement Backtracking and Polynomial Average Time. *Artificial Intelligence* 21(1):99–116.

F. Rossi. 1995. Redundant Hidden Variables in Finite Domain Constraint Problems. In M. Meyer., ed., *Constraint Processing*. Springer-Verlag. chapter 12, 205–233. Presented at the ECAI-94 Workshop on Constraint Processing.

D. Sabin and E. Freuder. 1994. Contradicting Conventional Wisdom in Constraint Satisfaction. In A. G. Cohn., ed., *Proceedings ECAI-94*, 125–129. John Wiley & Sons, Ltd.

D. Sabin and E. C. Freuder. 1997. Understanding and Improving the MAC Algorithm. In G. Smolka., ed., *Principles and Practice of Constraint Programming - CP-97 (to appear)*, Lecture Notes in Computer Science, 167–181. Springer-Verlag.

D. Sabin, M. Sabin, R. D. Russel and E. C. Freuder. 1995. A Constraint-Based Approach to Diagnosing Software Problems in Computer Networks. In U. Montanari and F. Rossi., eds., *Principles and Practice of Constraint Programming - CP-95*, volume 976 of *Lecture Notes in Computer Science*, 463–480. Springer-Verlag.

B. Selman, H. Levesque and D. Mitchell. 1992. A new method for solving hard satisfiability problems. In *Proceedings AAAI-92*, 440–446.

H. Simonis and M. Dincbas. 1987. Using an Extended Prolog for Digital Circuit Design. In *IEEE International Workshop on AI Applications to CAD Systems for Electronics*, 165–188.

B. M. Smith and M. E. Dyer. 1996. Locating the Phase Transition in Constraint Satisfaction Problems. *Artificial Intelligence* 81:155–181. Special issue on Frontiers in Problem Solving: Phase Transitions and Complexity.

B. M. Smith and S. A. Grant. 1995a. Sparse Constraint Graphs and Exceptionally Hard Problems. In C. S. Mellish., ed., *Proceedings IJCAI-95*, volume 1, 646–651. Morgan Kaufmann.

B. M. Smith and S. A. Grant. 1995b. Where the Exceptionally Hard Problems Are. In *Proceedings of the CP-95 Workshop on Studying and Solving Really Hard Problems*, 172–182. Laboratoire d'Informatique de Marseille, France.

B. M. Smith and S. A. Grant. 1997. Modelling Exceptionally Hard Constraint Satisfaction Problems. In G. Smolka., ed., *Principles and Practice of Constraint Programming - CP-97 (to appear)*, Lecture Notes in Computer Science, 182–195. Springer-Verlag.

B. M. Smith. 1994. Phase Transition and the Mushy Region in Constraint Satisfaction Problems. In A. G. Cohn., ed., *Proceedings ECAI-94*, 100–104. John Wiley & Sons, Ltd.

B. M. Smith. 1995. In Search of Exceptionally Difficult Constraint Satisfaction Problems. In M. Meyer., ed., *Constraint Processing*. Springer-Verlag. chapter 8, 139–155. Presented at the ECAI-94 Workshop on Constraint Processing.

R. M. Stallman and G. J. Sussman. 1977. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. *Artificial Intelligence* 9:135–196.

E. P. K. Tsang, J. Borrett and A. C. M. Kwan. 1995. An attempt to map the performance of a range of algorithm and heuristic combinations. In J. Hallam., ed., *Proceedings AISB-95*, 203–216. IOS Press, Amsterdam.

E. P. K. Tsang. 1993. *Foundations of Constraint Satisfaction*. Academic Press.

R. J. Wallace. 1993. Why AC-3 is Almost Always Better than AC-4 for Establishing Arc Consistency in CSPs. In *Proceedings IJCAI-93*, volume 1, 239–245. Morgan Kaufmann.

C. J. Wang and E. P. K. Tsang. 1991. Solving constraint satisfaction problems using neural-networks. In *Proceedings IEE Second International Conference on Artificial Neural Networks*, 295–299.

C. P. Williams and T. Hogg. 1993. The Typicality of Phase Transitions in Search. *Computational Intelligence* 9(3):211–238.

C. P. Williams and T. Hogg. 1994. Exploiting the Deep Structure of Constraint Problems. *Artificial Intelligence* 70:73–117.