University of Leeds
**SCHOOL OF COMPUTER STUDIES**
**RESEARCH REPORT SERIES**
Report 97.27

# How Not To Do It

by

**Ian P Gent[1], Stuart A. Grant, Ewen MacIntyre[1], Patrick Prosser[1], Paul Shaw[1], Barbara M Smith & Toby Walsh[1]**

May 1997

[1]Department of Computer Science, University of Strathclyde, Glasgow G1 1XH, Scotland.

**Abstract**

We give some dos and don'ts for those analysing algorithms experimentally. We illustrate these with many examples from our own research on the study of algorithms for NP-complete problems such as satisfiability and constraint satisfaction. Where we have *not* followed these maxims, we have suffered as a result.

## 1 Introduction

The empirical study of algorithms is a relatively immature field with many technical and scientific problems. We support the calls of McGeoch (1986,1996), Hooker (1994), and others for a more scientific approach to the empirical study of algorithms. Our contribution in this paper is colloquial. We admit to a large number of mistakes in conducting our research. While painful, we hope that this will encourage others to avoid these mistakes, and thereby to develop practices which represent good science.

Much of our research has been on the experimental analysis of algorithms and phase transitions in NP-complete problems, most commonly in satisfiability or constraint satisfaction problems. Hayes (1997) gives a non-technical introduction to satisfiability and phase transition research, and Kumar (1992) surveys constraint satisfaction problems.[1] The advice we give should be particularly appropriate for researchers in similar areas. However, we will not assume knowledge of the research we cite, as we believe that our lessons should be more generally valuable.

As well as McGeoch and Hooker, a number of authors give advice to those seeking to perform computational experiments as part of their research into algorithms. Cohen (1995) gives invaluable advice to workers in empirical artificial intelligence, particularly in the area of experimental design and statistical testing, although the empirical study of NP-complete problems is not discussed. Johnson (1996) gives general advice to experimenters, while Mitchell and Levesque (1996) give advice to those interested in the particular domain of random satisfiability. The particular novelty of this paper is that we own up to many mistakes that we ourselves have made. As well as avoiding any need to be fair to the authors being criticised, it also means we are aware of many mistakes that cannot be seen simply from reading our papers.

[1]More technical surveys of constraint satisfaction problems are given in articles by Dechter (1992) and Mackworth (1992) as well as the book by Tsang (1993). Volume 81 of *Artificial Intelligence* is a recent source of research papers on phase transition research [Hogg *et al.* 1996].

We will maintain a convention throughout: we hold all authors of a joint paper equally responsible for any mistake that one of them may have made. In what follows, the context will usually make clear that we are referring to a particular paper. Where we write 'I' or 'we', this should be taken to mean the author or all the authors of the relevant paper.

## 2 Getting Started

Suppose you have selected an algorithm that you wish to investigate experimentally. Typically you will end up writing your own implementation of this algorithm, either because it is new, or because you are interested in features of the algorithm you cannot investigate using publicly available code. Of the many lessons we have learnt about getting started, here are some of the most important.

DON'T TRUST YOURSELF
Bugs always find their way into the most harmful parts of your code. Having implemented the algorithm GSAT, a local search procedure for satisfiability [Selman *et al.* 1992], and written a version of a research paper [Gent and Walsh 1992], we discovered a bug in one of the most frequently called subroutines which biased the picking of a random element from a set. We noticed this bug when we observed very different performance running the same code on two different continents (from this we learnt, DO USE DIFFERENT HARDWARE). All our experiments were flawed and had to be rerun. Curiously one version of the bugged code gave better performance than the correct code. This suggests a hypothesis, which we have still to investigate fully, that systematic bias in GSAT's search may improve performance. In an attempt to prevent this happening again, we advocate a strategy of coding due to Boyer and Moore. That is, every line of code should be read and approved of by an independent source. Sometimes we even do this.

DO MAKE IT FAST ENOUGH
Surprisingly, it is not always necessary to have optimal code. But it does have to be fast enough. For example, our initial implementation of GSAT had an update time between moves so large as to make it practically useless. For standard problem sets, we eventually reduced this by the square of the number of variables in the problem, and how to do this was known at the time we wrote our code. It was a year before we got to this point, having reinvented several wheels. (Suggesting the advice DO REPORT IMPORTANT IMPLEMENTATION DETAILS, though it is very hard to follow this in papers with tight page limits.) In the meantime we had published results about GSAT [Gent and Walsh 1993a, Gent and Walsh 1993b] using code of

intermediate efficiency. Your code has got to be fast *enough* to do what you need: it need not be the fastest in the world. Occasionally we have inadvertently gone too far in this direction. For example, when studying the scaling of search cost [Gent *et al.* 1997] we accidentally ran interpreted rather than compiled Lisp, for a fifty-fold slowdown. So DO COMPILE CODE.

DO USE VERSION CONTROL
The software systems we use to conduct our empirical studies are inevitably large and complex. A typical experimental system might be made up of fifteen or more program modules. These modules are constantly evolving, thanks to the addition of new algorithms and methods of problem generation. Especially for versions of software used to produce published results, we suggest that you DO PRESERVE YOUR SOFTWARE. But we have found to our cost that doing this on an ad hoc basis can lead to the recurrence of old faults. In reporting the values of a constrainedness parameter, $\kappa$, we miscalculated values by a few percent, but did not notice it until the report had been published [Grant and Smith 1996]. The cause was a crass error involving a defective `for` loop. This error had been spotted and fixed once, but had reappeared. Some other modifications to the relevant module had been attempted at the same time as the bug-fix. When these were abandoned, the previous version of this module was recalled, and the error re-introduced. Although the underlying data was unaffected, principled use of a source control system such as RCS or SCCS [Bolinger and Bronson 1995] would have prevented the problem recurring at all.

## 3 Experimental Design

Once you have played with your debugged code, you may notice interesting behaviour, or implement interesting variants of the algorithm. From this, you may form various hypotheses about the algorithm's performance. In devising experiments to test such hypotheses, we have made many mistakes.

DO MEASURE WITH MANY INSTRUMENTS
Is it wise to only use one instrument to take measurements? A study of binary constraint satisfaction problems (CSPs) focussed on the nature of the phase transition, the crossover point, and the complexity peak [Prosser 1996]. In our eagerness we decided that we needed to look at many problems, using large sample sizes and varying as many problem parameters as possible. To do this we naturally used our best and fastest algorithm. Essentially, the algorithm was used as an instrument to take measures of characteristics of the problems. At that time Tad Hogg told us he had seen some unusual behaviour, namely hard problems where they were least ex-

pected. But we did not, no matter how hard we tried. The algorithms were different; Tad used a chronological backtracker, and we used a backjumper. Tad observed what later became known as exceptionally hard problems [Hogg and Williams 1994], something that is very difficult to find with smart backjumping. Our own observation of exceptionally hard constraint satisfaction problems was delayed by some time.

Do vary all relevant factors

We have reported on many series of experiments using the forward checking algorithm. It is well-known that the smallest-domain-first variable ordering heuristic [Haralick and Elliott 1980] gives good results for random CSPs, so we used this heuristic. Eventually we carried out some experiments using a static, random, variable ordering, and found that some of the effects we had reported were due to the heuristic rather than to the algorithm. For instance, we claimed that forward checking does not suffer from such exceptionally hard problems when the constraint density is high [Smith and Grant 1995a]. Later experiments showed that this is only when using the smallest-domain-first heuristic. We realised that we should vary both the heuristic and the algorithm in this kind of experiment only after carrying out a new series of experiments comparing forward checking with simple backtracking. It was only when we were some way into the experiments, and after some results had already appeared in print [Smith and Grant 1995b], that we started using forward checking with the same static variable ordering as the backtracking algorithm. We had, in fact, broken another rule (Don't change two things at once): some of the effects we had previously seen were due to the change in algorithm, and some to the change in variable ordering heuristic.

Do measure cpu time

Using cpu time as a measure of performance is often looked down upon, rightly, as it can lead to unproductive efficiency wars between researchers [Hooker 1995]. However, comparing cpu times within different versions of your own code can be very valuable. This came to light in (unpublished) experiments on variable ordering heuristics for the CSP. We were using various heuristics, one a complex version of a simple heuristic with a complex (and computationally very expensive) tie-breaking rule added. During search, evaluation of the tie-breaking rule alone used three to five times as many consistency checks as the search using the simple heuristic. Using total consistency checks as the sole measure of performance turned out to be a bad idea, since in many cases, and especially on the hardest instances, cpu time was reduced. It turned out that the code in which the heuristic was evaluated was very simple and needed to do less additional work per consistency check than the main search. To find this out we had to rerun our experiments, measuring CPU time.

Do collect all data possible

Usually, there are several obvious statistics to collect as well as CPU time. For example, for backtracking procedures there is the number of branches searched while for local search procedures there is the number of moves made. However, you should collect as many meaningful aspects of data as you can think of. For a long time, we did not record the number of branching points for backtracking procedures, a statistic subtly different from the number of branches. In investigating the satisfiability constraint gap [Gent and Walsh 1996], the most meaningful statistic turned out to be the ratio of constraint propagations to branching points. We therefore had to rerun many experiments. To have produced this data in the first place would have involved almost no extra expense. It pays to collect everything you can think of, whether or not you expect it to be important. When rerunning these experiments, we collected for the first time the minimal search depth, and this also turned out to be a very important statistic.

Do be paranoid

How do you know your instruments are working correctly? What hope is there for independent calibration when we are applying algorithms that we have invented to problems that we have randomly generated? One way round this is to replicate experiments independently. We often encode CSP algorithms [Prosser 1993b] in Scheme initially and then in C, often doing this at different sites. We can then replicate large scale experiments, and also look in detail at specific instances to make sure that we are obtaining identical results. Inevitably, apparently minor implementation details influence behaviour. For example, we encoded in Scheme (once) and C (twice) forward checking with conflict-directed backjumping using the smallest-domain-first heuristic. All three implementations gave different behaviour. We had not agreed what to do when many variables had the same heuristic value, i.e. in tie breaking situations. One of us took the first variable with best heuristic value, another the last variable, and one chose at random. While this did not make a significant difference on average, when we were examining an individual problem we got different results. Naturally our confidence went out the window. Enter the 'paranoid flag.' We now have two modes of running our experiments, one with the paranoid flag on. In this mode, we put efficiency aside and make sure that the algorithms and their heuristics do exactly the same thing, as far as we can tell.

Do check your solutions

A feature of NP-complete problems is the ability to verify any solution within polynomial time. A sensible and inexpensive security device is therefore to check every solution produced.

Do it all again
Or at least, be able to do it all again. Reproducibility should always be an important aim. Even if different random number generators preclude other researchers literally duplicating your results, you should be able to run the same experiment on the same seeds if necessary. Keeping random seeds has a second great benefit. Testing different procedures on identical problems, not merely problems generated in the same way, greatly reduces the variance in the difference in performance of the two procedures, and makes it much easier to obtain statistically significant results (Do use the same problems). We have tried to do this as much as possible, but occasionally our own stupidity has got the better of us. When a process going through a supposedly easy problem class had not written to the file for several hours, we thought the program had crashed so we killed the job and deleted the file. Fortunately, the next day the same thing happened with some new data, and we realised that some problems were remarkably difficult in this otherwise easy problem class [Gent and Walsh 1994a]. If the behaviour had been slightly rarer, we might have never seen it again, and would not have been able to go back to the original hard problem (Don't ignore crashes).

Do it often and do it big
One reason for making your code efficient (Do make it fast enough) is so that you can perform lots of experiments on large problems. This is important since emergent behaviour is often not apparent with small problems. In addition, running lots of experiments will reduce noise and may uncover rare but important hard cases. For example, we ran many experiments on random 3-SAT looking for bad worst case performance [Gent and Walsh 1994a]. We tried many different backtracking procedures at many different problem sizes using experiments with 1000 randomly generated problems per data point. We found nothing. So we tried again with 10,000 randomly generated problems at each data point. We still found nothing. With other problem classes, we had had no difficulty in finding bad worst case performance using just 1000 randomly generated problems. Persistence eventually paid off. In a month long experiment with 100,000 randomly generated problems at each data point, we finally observed bad worst case performance. Sometimes getting an interesting result requires a lot of perspiration and very little inspiration.

Don't kill your machines
Our previous piece of advice needs to be treated with some care. Because of the exponential nature of search algorithms for NP problems, it is very easy for a small misjudgement in experimental parameters to result in experiments that will take a very long time. When using shared resources such as cpu cycles donated from other projects, the temptation is to use more than your fair share to finish the experiment. As a result we have sometimes run into

trouble with colleagues or system administrators. The irony is that it is often on just these occasions that the results were unimportant anyway. This is particularly true where scaling methods such as 'finite-size scaling' are available, enabling results from smaller sizes to be extrapolated accurately [Gent *et al.* 1997]. So Do LOOK FOR SCALING RESULTS. They can often be found from data involving surprisingly small problem sizes and samples.

Do BE STUPID

Apparently stupid experiments can sometimes give fascinating results. For example, one of the fundamental features of GSAT is the greediness of its hill-climbing. Through perversity, we implemented a variant which was not greedy but indifferent to upwards or sideways moves [Gent and Walsh 1993b]. We did not expect this variant to perform at all well. To our surprise, its performance was similar to GSAT. Indeed it was even able to outperform GSAT when combined with a tabu-like restriction. Such chance discoveries help us understand the nature of GSAT's hill-climbing and will, we expect, guide future theoretical analyses. There is, however, a limit to stupidity. We also implemented a variant of GSAT which didn't even bother with the hill-climbing. Of course, this was stupid and ran very poorly.

## 4 Problems with Random Problems

Following Cheeseman, Kanefsky & Taylor (1991) and Mitchell, Selman & Levesque (1992) , many authors have performed many experiments on randomly generated problems: or rather *pseudo*-randomly generated problems. The use and misuse of pseudo-random number generators has led us into many problems.

DON'T TRUST YOUR SOURCE OF RANDOM NUMBERS

Although problems with the low bits of the standard C library `rand()` are well known, any random number generator can cause you problems, including the rather better `random()`. At one stage, we modified code which had been previously used to generate results for a publication [Gent *et al.* 1995]. The change meant that constraints or conflicts could be added to each problem in a given sample, but without affecting the next problem. This was done by repeatedly re-seeding the random number generator before generating each constraint, thus using many short streams of random numbers rather than one long stream. This in turn interacted with the way random numbers from `random()` were used. In general this interaction does not seem to have affected results, but when performing new experiments on the random n-queens problem [Gaschnig 1979] we observed anomalous behaviour on a board of size 16. Extensive investigations eventually showed that the combination of using a power of 2 and short streams of random numbers from `random()` had led to a significant bias in the way problems were generated. The problem only arose because we took the random number modulus the board size, so we were simply examining the lowest four bits. The result was problems where the frequency of conflicts occurring with certain combinations of values could be systematically higher than others. So the problems were not truly random. One general lesson is clear: Do check changes made to core code.

Do understand your problem generator
Assuming your random numbers are sound, there are many pitfalls on the way to generating random problems, to the extent that you can even fail to generate any problems at all. For many applications, where we seek a subset of problems with a certain property, it is appropriate to use a 'generate-and-test' method. To do otherwise might lead to subtle biases you are not aware of. For example, in investigating bin-packing problems [Gent and Walsh 1997] we generated bags of numbers and discarded those with a sum outside a certain range. We then proceeded to waste hundreds of hours of CPU time over a holiday. We had asked some machines to use this generator with a set of parameters which make it impossible ever to generate a valid problem.

Do check your health regularly
An even more elementary mistake shows up a general lesson: that wherever possible you should check your results against those published by others. A few days before submitting a paper to a conference [Gent *et al.* 1995], we realised that our data was entirely inconsistent with some reported by Frost & Dechter (1994) . We quickly realised the cause: when we sought to obtain 3 conflicts out of a possible 9, our generator took the integer part of the floating point calculation $9 \times (3/9) = 9 \times 0.333\ldots = 2.999\ldots$ and obtained 2. Ironically, every line of every result file contained the correct conflict density of $0.222\ldots$, showing that we had not looked at our raw data.

Do control sources of variation
Chance features of randomly generated problems can obscure the features of the problem you are trying to control. For example, when generating satisfiability problems from the 'constant probability model' we allowed tautologies to occur [Gent and Walsh 1994a]. A clause which is a tautology is irrelevant, so a 100 clause problem with 50 tautologies is really a 50 clause problem. So in effect we have lost control of the number of clauses the problem contains. Fortunately in our experiments the number of tautologies was never large and so the effect was negligible. A similar issue arose in experimenting on random binary CSPs, in a model containing the parameter $p_2$ indicating the average number of conflicts per constraint [Prosser 1996, Smith and Dyer 1996]. In each constraint we include *exactly* $p_2$ times the total possible number of

conflicts: if we included each conflict with probability $p_2$ then the number of conflicts in each constraint would vary outside our control. Controlling sources of variation is not the same as eliminating them. We later wished to experiment on non-uniform problems in which $p_2$ varies within a problem. To do this we designed a random problem generator to allow for this explicitly and under our control [Gent *et al.* 1996].

## 5 Analysis of Data

Having run your computational experiments, you are now in a position to analyse the data. Somewhat surprisingly, it is often quite hard to determine the actual outcome of the experiments. One reason is the mega-bytes of data generated. Here are some of the lessons we have learnt in sifting through such mountains of data.

DO LOOK AT THE RAW DATA
Summaries of the data inevitably present an approximate view. By looking at the raw data, you can often spot trends, and interesting odd cases which are hidden in the summaries. For example, we couldn't miss the worst case problem that needed a week of CPU whilst most other problems in the region took seconds [Gent and Walsh 1994a]. However, we had failed to see this effect at smaller problem sizes. Although our experiments had come across other such problems, orders of magnitude harder than the typical problems in that region, graphs of median behaviour gave no hint of their existence. At best, graphs of mean behaviour for smaller problem sizes only showed more noise within this region.

DO LOOK FOR GOOD VIEWS
Almost all the insights we have had into the behaviour of our algorithms have come from finding a good view of the data. For example, for hill-climbing procedures like GSAT, we tried plotting the number of unsatisfied clauses (the "score") against the number of moves performed. Such graphs were not very illuminating. We therefore looked for a better view. We tried many possibilities before arriving at a simultaneous plot of the number of variables offered at each flip and the derivative of the score. ¿From this, we were able to see clearly the very different stages in the search [Gent and Walsh 1993a]. Sometimes an experiment will suggest a good view of old data. It's important therefore not to throw away data (DON'T DISCARD DATA).

DO FACE UP TO THE CONSEQUENCES OF YOUR RESULTS
In testing out a new algorithm, which we expected to reduce search, we found 2 cases out of 450 where it *increased* search. The obvious explanation for this was bugged code. We re-coded the algorithm from scratch, but got the same

results. We couldn't understand it, and thought that we should just forget the algorithm once and for all. But sometimes it's not easy to forget. We resorted to a detailed analysis of the two problems, tracing every feature of the algorithm as it progressed through the search space. A visual analysis took days, and resulted in an explanation of the phenomenon and an ability to replicate it [Prosser 1993a].

Don't reject the obvious

Several times we have looked at some result and rejected – or not considered – an obvious interpretation. For example, we plotted graphs of the score for GSAT decaying with the number of moves performed, but did not consider the obvious possibility of exponential decay. This would have been too simple for such a complex system. Several weeks later, with the aid of a statistics package, we discovered it was indeed a simple exponential decay [Gent and Walsh 1993a]. It would, however, have only taken a quick log plot to have found this out.

## 6 Presentation of Results

Having found a good view of the data which supports or rejects your hypotheses, you now will want to present your results to a wider audience. There are still many mistakes to make.

Do present statistics

All too often, we have presented our results by simple tables of mean performance. With experience, we have learnt that even the very simplest of statistics can provide considerably more information, giving your audience some feel for the spread of values or the accuracy of a fit. When we have presented talks without this data, it has on occasion frustrated audiences considerably. As a matter of course, we advise giving the minimum, mean, maximum, median and standard deviation. Such statistics are easy to compute, yet give a much more complete picture.

Do report negative results

It is tempting just to report your successes. This should be resisted at all costs. Reporting negative results can be just as valuable as reporting positive results. You will save other people wasting time on dead-ends. For example, we reported that adding memory of where you had previously been in the search space did not significantly improve the performance of GSAT [Gent and Walsh 1993b]. It was not an exciting result but it does save others from exploring that fruitless avenue. Negative results may also suggest important new hypotheses. For example, we reported in one short paragraph our inability to observe with GSAT the very variable behaviour seen with backtracking procedures in mostly satisfiable regions [Gent and Walsh 1994b]. We are happy to applaud Davenport (1995) for reporting more detailed results, also failing to find any such behaviour. We do not know if exceptionally hard problems do not occur, or if they just occur less frequently or with larger problem sizes.

Don't push deadlines
A deadline does, as Dr Johnson said, concentrate the mind wonderfully. Experiments we report in most of our papers have been run in the last two weeks before a deadline, because we find in writing a first draft that the experimental data we have is incomplete or does not cover all the cases we need. To date, we have very rarely succeeded in following this advice, and have learnt from it only to the extent of giving ourselves deadlines where none is imposed from outside.

Do check your references
This advice is easily given but harder to follow. For example, in an earlier version of this very paper [Gent and Walsh 1994c], we gave the incorrect page numbers for a classic in our field. This was because, to our shame, we had cribbed the reference from another paper rather than the source itself.

## 7 Conclusions

We have presented some of the lessons we have learnt in studying NP-complete problems experimentally. With hindsight, most of these lessons now seem obvious. Indeed, most of them *are* obvious. However, this did not stop us making many mistakes along the way. Perhaps the use of appropriate quality assurance techniques would have prevented them occurring. We should stress that this list of lessons is far from comprehensive. It was not intended to be. There are still many mistakes for us to make in the future. To distort a famous saying, those unable to learn from their mistakes are destined to repeat them. We therefore hope you can benefit a little from our mistakes and stupidity.

**Acknowledgements**

We thank our colleagues in the APES Research Group, of which the authors are all members, especially Philip Kilby for particularly helpful comments.

We wish to thank the various funding bodies that have supported our research over the years, and the various colleagues we have discussed our work with. Our mistakes, however, remain our own.

## References

[Bolinger and Bronson 1995] D. Bolinger and T. Bronson. 1995. *Applying RCS and SCCS*. O'Reilly & Associates.

[Cheeseman *et al.* 1991] P. Cheeseman, B. Kanefsky and W. Taylor. 1991. Where the really hard problems are. In *Proceedings of the 12th IJCAI*, 331–337.

[Cohen 1995] P. Cohen. 1995. *Empirical methods for Artificial Intelligence.* MIT Press.

[Davenport 1995] A. Davenport. 1995. A comparison of complete and incomplete algorithms in the easy and hard regions. In *Proceedings, Workshop on Studying and Solving Really Hard Problems, CP-95*, 43–51.

[Dechter 1992] R. Dechter. 1992. Constraint networks. In *Encyclopedia of Artificial Intelligence*, 276–286. Wiley. 2nd Edition.

[Frost and Dechter 1994] D. Frost and R. Dechter. 1994. In search of the best search: an empirical evaluation. In *Proceedings AAAI-94*, 301–306.

[Gaschnig 1979] J. Gaschnig. 1979. Performance measurement and analysis of certain search algorithms. Technical report CMU-CS-79-124, Carnegie-Mellon University.

[Gent and Walsh 1992] I. Gent and T. Walsh. 1992. The enigma of SAT hill-climbing procedures. Research Paper 605, Dept. of Artificial Intelligence, University of Edinburgh.

[Gent and Walsh 1993a] I. P. Gent and T. Walsh. 1993a. An empirical analysis of search in GSAT. *Journal of Artificial Intelligence Research* 1:47–59.

[Gent and Walsh 1993b] I. P. Gent and T. Walsh. 1993b. Towards an understanding of hill-climbing procedures for SAT. In *Proceedings of AAAI-93*, 28–33.

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

[Gent and Walsh 1994b] I. Gent and T. Walsh. 1994b. The hardest random SAT problems. In B. Nebel and L. Dreschler-Fischer., eds., *KI-94: Advances in Artificial Intelligence. 18th German Annual Conference on Artificial Intelligence*, 355–366. Springer-Verlag.

[Gent and Walsh 1994c] I. Gent and T. Walsh. 1994c. How not to do it. Research Paper 714, Dept. of Artificial Intelligence, Edinburgh.

[Gent and Walsh 1996] I. Gent and T. Walsh. 1996. The satisfiability constraint gap. *Artificial Intelligence* 81:59–80.

[Gent and Walsh 1997] I. Gent and T. Walsh. 1997. From approximate to optimal solutions: Constructing pruning and propagation rules. In *Proceedings of IJCAI 97*. In press.

[Gent *et al.* 1995] I. Gent, E. MacIntyre, P. Prosser and T. Walsh. 1995. Scaling effects in the CSP phase transition. In *Principles and Practice of Constraint Programming*, 70–87. Springer.

[Gent *et al.* 1996] I. Gent, E. MacIntyre, P. Prosser, B. Smith and T. Walsh. 1996. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Proceedings of CP-96*, 179–193. Springer.

[Gent *et al.* 1997] I. Gent, E. MacIntyre, P. Prosser and T. Walsh. 1997. The scaling of search cost. In *Proceedings of AAAI-97*, to appear.

[Grant and Smith 1996] S. Grant and B. Smith. 1996. The arc and path consistency phase transitions. Report 96.09, Research Report Series, School of Computer Studies, University of Leeds.

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

[Hayes 1997] B. Hayes. 1997. Can't get no satisfaction. *American Scientist* 85:108–112.

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

[Hogg *et al.* 1996] T. Hogg, B. Huberman and C. Williams., eds. 1996. *Artificial Intelligence*, volume 81. Elsevier. Special Volume on Frontiers in Problem Solving: Phase Transitions and Complexity.

[Hooker 1994] J. N. Hooker. 1994. Needed: An empirical science of algorithms. *Operations Research* 42:201–212.

[Hooker 1995] J. N. Hooker. 1995. Testing heuristics: We have it all wrong. *Journal of Heuristics* 1:33–42.

[Johnson 1996] D. S. Johnson. 1996. A theoretician's guide to the experimental analysis of algorithms. Invited talk at AAAI-96. Partial draft available at `http://www.research.att.com/~dsj/papers/exper.ps`.

[Kumar 1992] V. Kumar. 1992. Algorithms for constraint satisfaction problems: a survey. *AI Magazine* 13:32–44.

[McGeoch 1986] C. McGeoch. 1986. *Experimental Analysis of Algorithms*. Ph.D. Dissertation, Carnegie Mellon University. Also available as CMU-CS-87-124.

[McGeoch 1996] C. McGeoch. 1996. Toward an experimental method for algorithm simulation. *INFORMS Journal on Computing* 8:1–15.

[Mitchell and Levesque 1996] D. G. Mitchell and H. J. Levesque. 1996. Some pitfalls for experimenters with random SAT. *Artificial Intelligence* 81:111–125.

[Mitchell *et al.* 1992] D. Mitchell, B. Selman and H. Levesque. 1992. Hard and easy distributions of SAT problems. In *Proceedings, 10th National Conference on Artificial Intelligence*, 459–465. AAAI Press/The MIT Press.

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

[Prosser 1993b] P. Prosser. 1993b. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence* 9:268–299.

[Prosser 1996] P. Prosser. 1996. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence* 81:127–154.

[Selman *et al.* 1992] B. Selman, H. Levesque and D. Mitchell. 1992. A new method for solving hard satisfiability problems. In *Proceedings of AAAI-92*, 440–446.

[Smith and Dyer 1996] B. Smith and M. Dyer. 1996. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence* 81:155–181.

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

[Smith and Grant 1995b] B. Smith and S. Grant. 1995b. Where the *exceptionally* hard problems are. Report 95.35, Research Report Series, School of Computer Studies, University of Leeds.

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