How Not To Do It*

Ian P. Gent
Department of Artificial Intelligence,
University of Edinburgh,
80 South Bridge,
Edinburgh EH1 1HN, United Kingdom
I.P.Gent@edinburgh.ac.uk

Toby Walsh
INRIA-Lorraine,
615, rue du Jardin Botanique,
B.P. 101,
54602 Villers-les-Nancy, France
walsh@loria.fr

Presented at the AAAI Workshop on Experimental Evaluation of Reasoning and Search Methods, July 31, 1994

Abstract

We give some dos and don'ts for those analysing algorithms experimentally. We illustrate these with many examples from our own research. 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 shall not address the many technical problems (*e.g.* determining appropriate performance measures and representative problem samples) but will instead look at some of the scientific problems. We support the calls of McGeoch [10], Hooker [9] and others for a more scientific approach to the empirical study of algorithms. Our contribution here is colloquial. With time and more such tales from the front-line, we hope that standard experimental practices in the study of algorithms will develop which represent "good science".

Our own research has been into satisfiability (SAT), concentrating on two main areas. First, we have looked at hill-climbing procedures for SAT [8, 12]. Using many variants of such hill-climbing procedures, we have uncovered several interesting and unexpected features [2, 3, 5]. Second, we have analysed the phase transition for SAT: that is, the very sharp transition from satisfiable to unsatisfiable problems associated with the hardest random problems [1, 11]. In particular, we have concentrated on the worst case performance, and have shown this is very different from median performance [4, 6, 7]. In all, we have invested over a quadrillion CPU cycles looking at SAT. Not all of these cycles have been productive as we now explain.

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.

*The first author was supported by a SERC Postdoctoral Fellowship and the second by a HCM Postdoctoral fellowship. We thank Alan Bundy, Pierre Lescanne, and the members of the Mathematical Reasoning Group at Edinburgh (supported by SERC Grant GR/H 23610), the Department of Computer Science at Cornell University and the Eureca group at INRIA-Lorraine. The second author's current address is IRST, Location Panté di Povo, I38100 Trento, Italy. toby@irst.it

Don't trust yourself

Bugs always find their way into the most harmful parts of your code. Having implemented the algorithm GSAT and written a version of [3], 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 under two different compilers (from this we learnt, Do use different hardware). As a result, 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 flips) of O(kNL) making it practically useless. This can be reduced to $O(k\frac{L}{N})$ for the usual problem sets, and this was known at the time we wrote our code. It was a year before we got to this point, having reinvented several wheels. In the meantime we had published results about GSAT [2, 5] 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.

3 Experimental Design

Once you have played with your debugged code, you may start to notice some interesting aspects of the behaviour, or to 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 collect all data possible

Usually, there are several obvious statistics to collect as well as CPU time. For example, for GSAT there are the number of 'tries' and 'flips', while for backtracking procedures there is the number of branches searched. However, you should collect as many meaningful aspects of data as you can think of, even if you don't know what to do with the data currently. 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 [7], the most meaningful statistic turned out to be the ratio of constraint propagations (*ie.* unit propagations and pure eliminations) 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 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 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 [4]. 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) 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 [7]. 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.

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 [2]. 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 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 very difficult worst case problem described in [4] since it took about a week of CPU whilst most other problems in the region took seconds. 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 flips 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 [5]. Sometimes an experiment will suggest a good view of old data. It's important therefore not to throw away data (DON'T DISCARD DATA).

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 flips 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 [5]. It would, however, have only taken a quick log plot to have found this out.

5 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 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, even though we had the data in the papers supporting the talk. As a matter of course, we now give 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, in [2], we reported that adding memory of where you had previously been in the search space did not significantly improve the performance of GSAT. 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, in [7] we report our inability to observe with GSAT the very variable behaviour seen with backtracking procedures in mostly satisfiable regions. We do not know if it does not occur, or if it just occurs more infrequently or with larger problem sizes. We very much hope someone else will find out.

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 fortnight 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 *never* succeeded in following this advice, and have learnt from it only to the extent of giving ourselves deadlines where none is imposed from outside.

6 Conclusions

We have presented some of the lessons we have learnt in studying SAT 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. 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.

References

[1] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In *Proceedings of the 12th IJCAI*, pages 163–169. International Joint Conference on Artificial Intelligence, 1991.

[2] I. P. Gent and T. Walsh. Towards an Understanding of Hill-climbing Procedures for SAT. In *Proceedings of the 11th National Conference on Artificial Intelligence*. AAAI Press, 1993.

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

[4] I. P. Gent and T. Walsh. Easy problems are sometimes hard. Research Paper 642, Dept. of Artificial Intelligence, Edinburgh, June 27 1993. Revised version to appear in *Artificial Intelligence*.

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

[6] I. P. Gent and T. Walsh. The SAT phase transition. Research Paper 679, Department of Artificial Intelligence, University of Edinburgh, 1994. Revised version in Proceedings of the European Conference on AI, ECAI-94.

[7] I. P. Gent and T. Walsh. The Hardest Random SAT Problems. Research Paper 680, Department of Artificial Intelligence, University of Edinburgh, 1994. Revised version in Proceedings of the 18th German Conference on AI, K-94.

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

[9] J. N. Hooker. Needed: An Empirical Science of Algorithms. Research Paper, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh PA, January 1993.

[10] C. C. McGeoch. *Experimental Analysis of Algorithms*. PhD thesis, Carnegie Mellon University, 1986. Also available as CMU-CS-87-124.

[11] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proceedings, 10th National Conference on Artificial Intelligence*. AAAI Press, 1992.

[12] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings, 10th National Conference on Artificial Intelligence*. AAAI Press, 1992.