CNF Symmetry Breaking Options in Conflict Driven SAT Solving

Alexander Keur, Coen Stevens, and Mark Voortman

Department of Software Technology
Faculty of Electrical Engineering,
Mathematics and Computer Sciences
Delft University of Technology

a.a.a.keur@ewi.tudelft.nl, beatlevic@gmail.com, mark@voortman.name

Abstract. Many CNF formulas representing real-world problems exhibit symmetries. Various efforts have been made to deal with these symmetries efficiently. After Crawford *et al* [2], which gives a theoretical framework for detecting and exploiting symmetries, different suggestions have been posed to prevent the exponential blow up of overhead costs which shows up performing so-called full symmetry breaking. Aloul *et al* [1] give an overview of these preprocessing methods and they present a new device (Shatter) which tries to eliminate the costs of overhead substantially. We propose an alternative way of symmetry breaking in cases where conflict-clause driven SAT solvers are used in solving the CNF at hand. Instead of preprocessing the formula involved we investigate the effect of applying permutations of the symmetry group on the emerging conflict-clauses only and to add them to the conflict-clause list additionally. We use random graph k-colourings in this experimental study, first because difficult examples can be generated easily and second because of the fact that the size of the full symmetry group grows exponentially in k. In doing so we are able to obtain a clear picture of the trade-off between costs of overhead and gains of dealing with smaller search trees. We compare our results with the Shatter preprocessing option and we selected the conflict driven zChaff SAT solver [3] as a standard representative of the family of conflict driven solvers.

1 Introduction

Given a CNF $\mathcal{F}$ on variables $x_1, \ldots, x_n$ and a permutation π from the symmetry group S_n, $\mathcal{F}$ is said to have symmetry π whenever for any assignment σ for $\mathcal{F}$ we have: σ satisfies $\mathcal{F}$ if and only if $\pi(\sigma)$ satisfies $\mathcal{F}$. Here $\pi(\sigma)$ is the assignment obtained from σ by permuting its coordinates according to π. Alternatively, one might characterize symmetry through the following: for any clause C we have: π implies C if and only if $\mathcal{F}$ implies $\pi(C)$. Here $\pi(C)$ is obtained from C by permuting its variables according to π.

Many CNF formulas stemming from real-world problems - like planning and scheduling problems and EDA applications - exhibit many symmetries, either

intrinsically inherited from the combinatorial nature of the problem at hand or introduced by the CNF transformation process, or by both.

Solving such formulas by SAT solvers which are only based on resolution is a troublesome attempt. Urquhart [4] gives a theoretical insight into this feature. Crawford *et al* [2] present a framework which detects symmetries in a given CNF and also give a method for so-called full-symmetry breaking: for each symmetry a list of clauses is added to the original formula which prevent SAT solvers doing "duplicate" (under the given symmetry) search. The complexity of detecting symmetries is still open, yet in practice this fact seems to be less important than the question how to deal with the symmetries efficiently. This is due to the observation that, in many cases, the symmetry groups grow exponentially in problem size and, making it even worse, full symmetry breaking requires adding an exponential number of clauses. The fundamental question here to be answered is to find an effective trade-off between the overhead costs caused by symmetry breaking and the savings of pruning search trees. Various compromises are suggested to deal with this problem. Aloul *et al* [1] enumerate some of these attempts and present an alternative; Shatter, a preprocessing method of which they claim to be rather effective.

In this paper we propose an alternative for breaking known symmetries in cases where the CNF involved is tackled by a so-called conflict driven solver. During their search, conflict driven solvers generate clauses which are globally implied by the CNF. These so-called conflict-clauses are added to a list which is consulted and managed dynamically during the search. In the presence of symmetries it is obvious that in these cases one has the opportunity to add the permutations of the generated conflict-clauses as well. Of course, this method is subject to the same observations as made above: when the symmetry group is large, adding all permutations of all generated conflict-clauses will cause an immediate overflow and obstruction of the conflict-clause database management of the solver used.

The underlying experimental study tries to gain a first insight into the trade-off questions raised above when applying the method suggested. We selected graph k-colouring problems as test environment because they can be considered as the ultimate abstraction of planning and scheduling problems. Another reason for this selection is that the symmetry groups involved are transparent and that their sizes grow exponentially with k. More specifically, we restricted ourselves to random k-colouring because intrinsically hard problems can be generated easily.

Further we selected zChaff [3] as a default conflict driven solver because of reasons explained later. In this first exploration we deliberately chose not to interfere with the implemented strategies of zChaff's conflict-clause database management. We compare our results with Shatter's preprocessing method.

2 Preliminaries

A formula (denoted as $\mathcal{F} = c_1 \wedge c_2 \wedge \ldots \wedge c_m$) in *Conjunctive Normal Form* (CNF) is a conjunction of clauses; each clause (denoted as $c_i = l_1 \vee l_2 \vee \cdots \vee l_n$)

being a disjunction of literals; and each literal is an atomic Boolean variable x_i or its negated form $\neg x_i$. The satisfiability (SAT) problem deals with the question whether a CNF formula is satisfiable (has a Boolean solution) or not. A formula is satisfiable if an assignment satisfies all clauses. A clause is satisfied if at least one of its literals is satisfied.

The graph k-colouring problem asks whether a given graph $G = (V, E)$ can be coloured using at most k colours, such that connected vertices have different colours. A general transformation of a k-colouring problem to CNF consists of (1) $|V|$ clauses of length k forcing each vertex to have at least one colour, and (2) $k \times |E|$ binary clauses that prevent connected vertices from having the same colour. Graph k-colouring formulas have a transparent symmetry group, namely S_k. In any solution, colours can be permuted arbitrarily, resulting in another solution.

3 zChaff_fullSym

We used the zChaff[1] solver by Moskewicz *et al* [3] as environment to experiment with the permutations of conflict-clauses. The main reasons to use zChaff are its performance, its free accessibility and the readability of its source code. In algorithm 1, a simplified version of the zChaff solver is shown.

Algorithm 1 zChaff

```
1: loop
2:   if decide_next_branch() then
3:     while deduce() == CONFLICT do
4:       blevel := analyze_conflicts()
5:       if blevel == 0 then
6:         return UNSATISFIABLE
7:       else
8:         backtrack(blevel)
9:       end if
10:    end while
11:  else
12:    return SATISFIABLE
13:  end if
14: end loop
```

Because we chose not to interfere with zChaff's conflict-clause database management, adding the permutations only required an adjustment of the analyze_conflicts() procedure. Besides performing the addition of conflict-clauses, this procedure returns the backtrack level. The analyze_conflicts() procedure is modified in such a way that for each conflict-clause zChaff wants to add, all its permutations are added too. This resulted in our algorithm zChaff_fullSym.

[1] version 2004.5.13 available at `http://www.princeton.edu/~chaff/zchaff.html`

4 Experimental Results

To evaluate the performance of zChaff_fullSym we experimented on random graph k-colouring instances. For comparison we used the original zChaff as well as the combination Shatter and zChaff. In this combination, Shatter is used for preprocessing.

Both zChaff and zChaff_fullSym were compiled with GCC 3.3.5 using the `-O3` option. Tests were run on a AMD Athlon 1800+ machine with 1GB of main memory, running Linux kernel 2.4.29. The random k-colouring graphs were generated by the Graph Colouring Generator Version 2 by Joseph Culberson[2] and then converted to the DIMACS CNF format.

We restricted ourselves to 3-, 4-, and 5-colouring instances, when we noticed that the relative performance of zChaff_fullSym decreased significantly on the random 5-colouring graphs. For each of the three random graph k-colouring experiments, we generated about 5000-7500 instances, of which approximately 1000 were around the phase transition density. For the 3-, 4-, and 5-colouring experiments we generated instances with 400, 100, and 70 vertices, respectively.

The scatterplots below show the speed-up factor realized by zChaff_fullSym with respect to the combination Shatter_zChaff. Since it is unclear how much time Shatter spends for detection of the symmetries - which we took for granted - we decided to neglect the computational time used by Shatter. Notice that we use a $\log_2$ scale to express this speed-up factor.

The histograms below show the average time in seconds that the three different solvers required to determine whether instances around a certain density were satisfiable or not. The results in both the scatterplots and the histograms are presented separately for satisfiable and unsatisfiable instances.

5 Analysis and Conclusions

Analyzing the data concerning the unsatisfiable instances leaves us with a few clear conclusions. First, zChaff's performance is improved by both approaches. For the 3-colouring case (6 symmetries) our proposed method gives an observably better improvement compared to the Shatter preprocessing option. For the 4-colouring case (24 symmetries) the same conclusion can be drawn, albeit already less convincing. However, it must be said that the results are a bit biased in favour of the Shatter approach, since we did not invoke its preprocessing time. The data on the 5-colourings (120 symmetries) clearly shows that our method breaks down compared to the Shatter preprocessing option. A first tentative overall conclusion is that our method could be a promising alternative to the preprocessing options in cases where the symmetry group has modest size. For larger symmetry groups our method must be adapted: as in the preprocessing options it seems reasonable to restrict to a generating set of permutations but one could also think of aligning permutation selection with the conflict-clause

[2] freely available at `http://www.cs.ualberta.ca/~joe/Coloring/`

database management: since the permuted conflict-clauses are added to avoid duplicate search it is not unreasonable to keep them somewhat longer in the dynamically managed conflict-clause database.

An approach of combining preprocessing options with our method could be very tricky: the preprocessing options destroy the original symmetries and emerging conflicts therefore can not be permuted straightforwardly.

When analyzing the data on satisfiable instances (3- and 5-colourings) the reader should notice that this is almost irrelevant compared to the above: computational times show that generally they are solved orders of magnitude faster than the unsatisfiables. Still one might conclude that satisfiable instances with a relative high density benefit slightly from our approach for the 3-colouring case. The 4-colouring instances, where we took a number of vertices in such away that solving times for unsatisfiables are comparable, show something similar. It must be said that, in general, analyzing data on satisfiable CNF's is hard. Certainly in our experiments, in cases where zChaff finds a solution fast, any symmetry breaking method invoked will clearly only result in costs of overhead. This effect is visible from our data.

An overall conclusion from our experiments must undoubtedly be that Shatter is a fairly effective symmetry breaking preprocessor. It is elegant and its use is independent of the solver in sequence. However, we also feel confident with concluding that our approach is worth to be explored further. It has the advantage that symmetry breaking and conflict-clause management can be tuned to obtain optimal performance.

References

1. F. Aloul, I. Markov, and K. Sakallah.
   Shatter: Efficient Symmetry-Breaking for Boolean Satisfiability,
   in Design Automation Conference (2003), 836–839.
2. J. Crawford, M. Ginsberg, E. Luks, and A. Roy. *Symmetry-breaking predicates for search problems*, in Proc. of the Intl. Conference Principles of Knowledge Representation and Reasoning (1996), 148–159.
3. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik.
   Chaff: Engineering an efficient SAT solver,
   in Design Automation Conference (2001).
4. A. Urquhart. *Hard Examples for Resolution*,
   in Journal of the ACM **34**(1) (1987), 209–219.

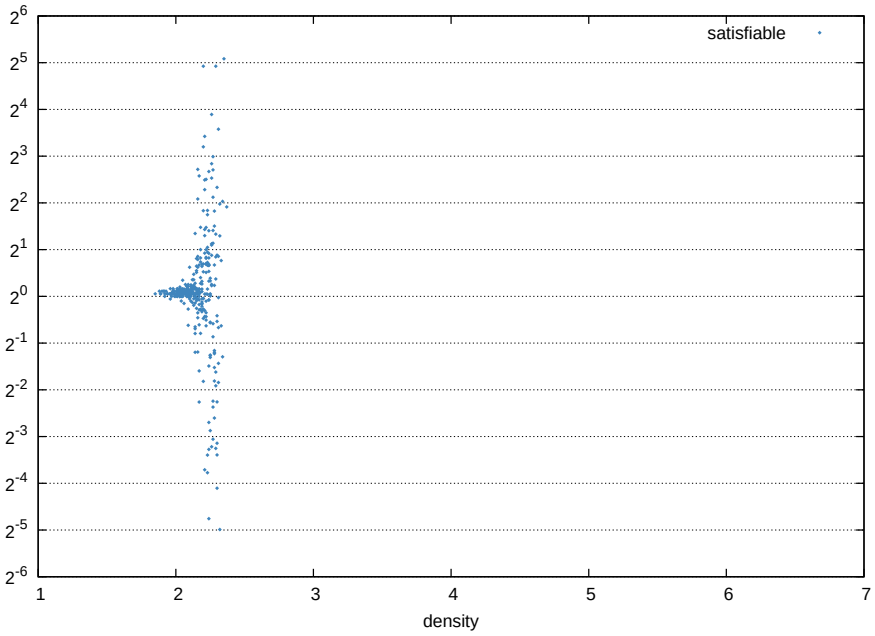

Fig. 1. Scatterplot of $\log_2$ (time(Shatter_zChaff) / time(zChaff_fullSym)) on satisfiable random 3-colouring graphs with 400 vertices.

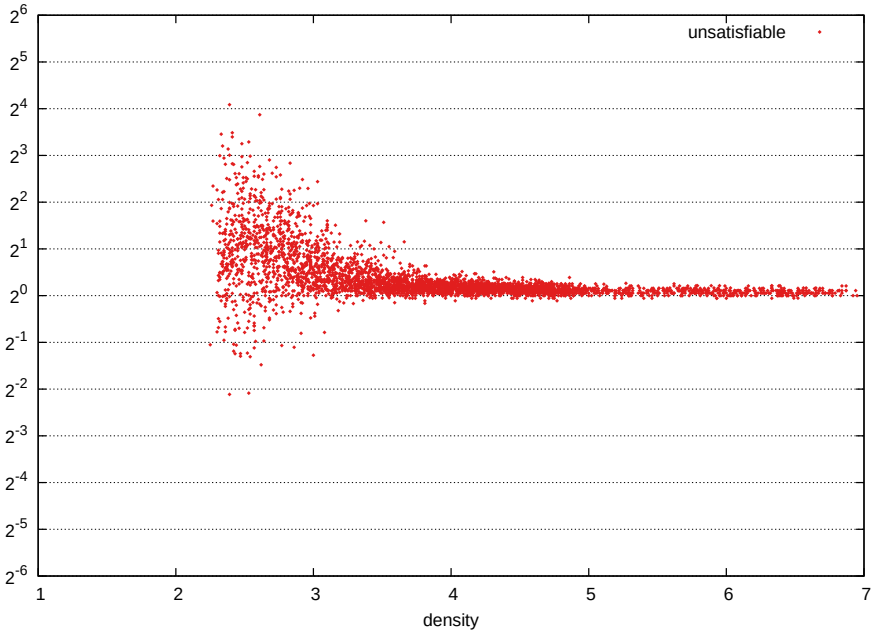

Fig. 2. Scatterplot of $\log_2$ (time(Shatter_zChaff) / time(zChaff_fullSym)) on unsatisfiable random 3-colouring graphs with 400 vertices.

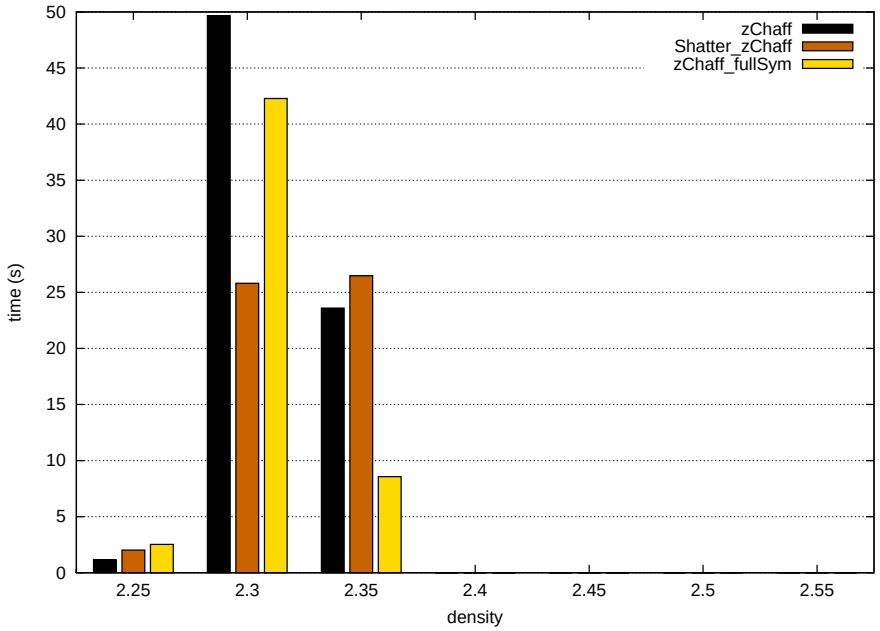

Fig. 3. Histogram of the average time required by the three algorithms to solve satisfiable random 3-colouring graphs near the phase transition density.

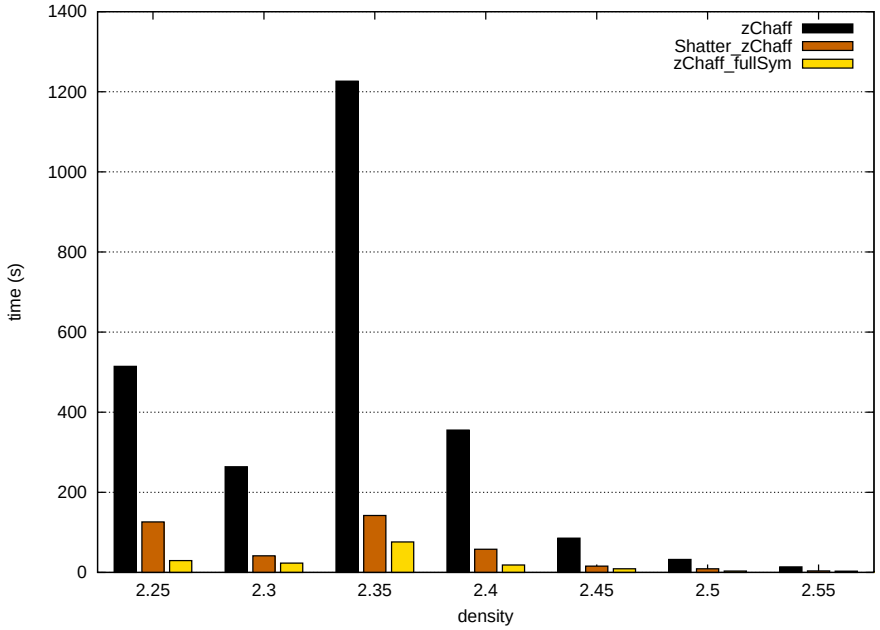

Fig. 4. Histogram of the average time required by the three algorithms to solve unsatisfiable random 3-colouring graphs near the phase transition density.

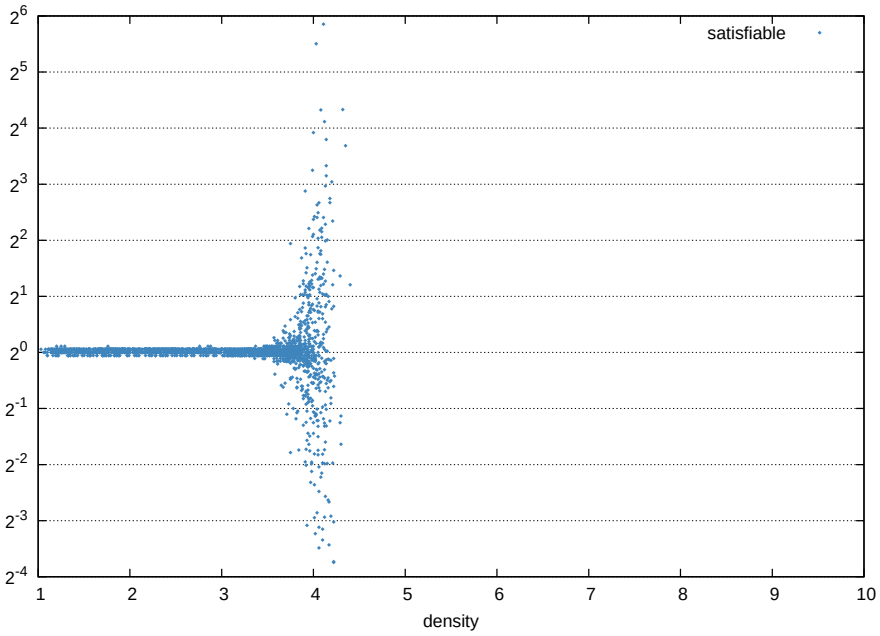

Fig. 5. Scatterplot of $\log_2$ (time(Shatter_zChaff) / time(zChaff_fullSym)) on satisfiable random 4-colouring graphs with 100 vertices.

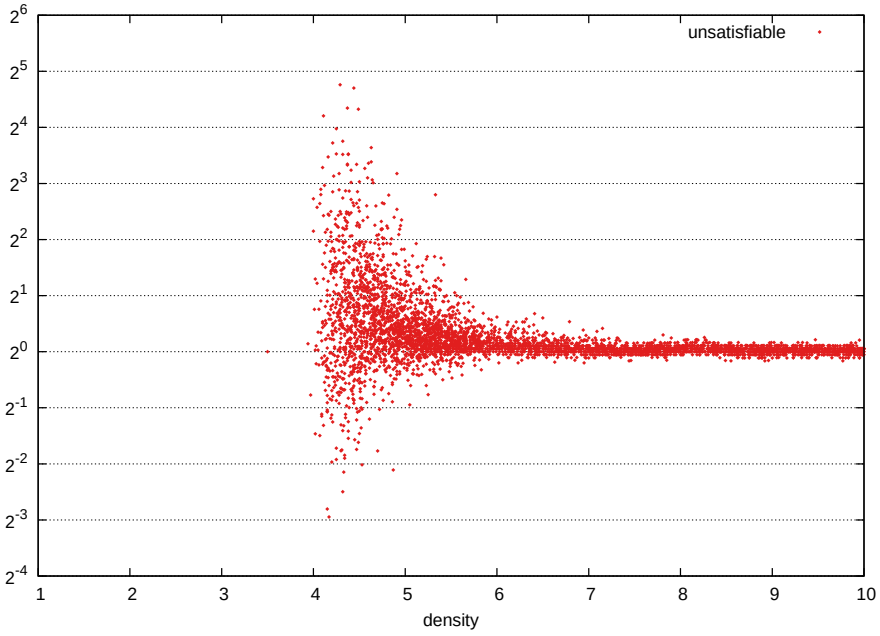

Fig. 6. Scatterplot of $\log_2$ (time(Shatter_zChaff) / time(zChaff_fullSym)) on unsatisfiable random 4-colouring graphs with 100 vertices.

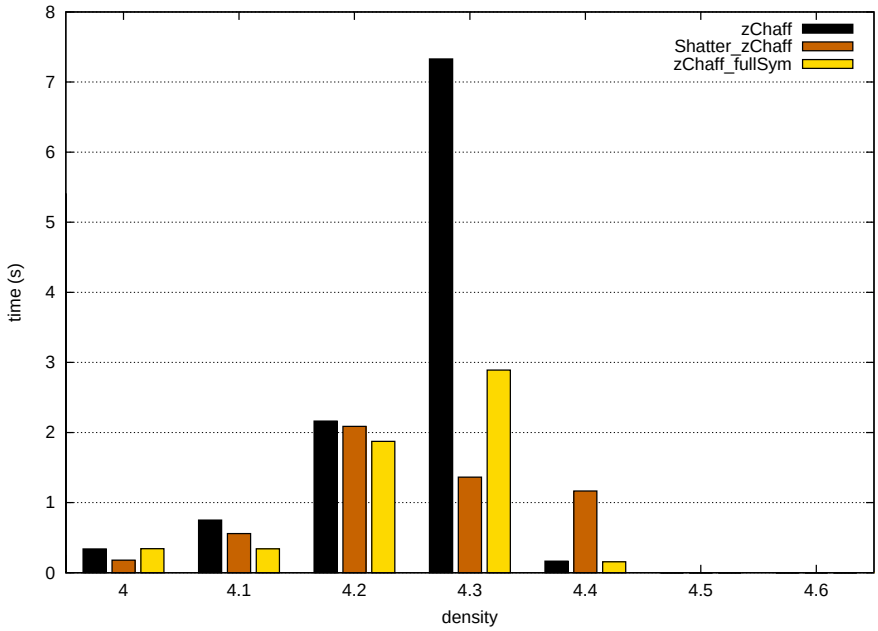

Fig. 7. Histogram of the average time required by the three algorithms to solve satisfiable random 4-colouring graphs near the phase transition density.

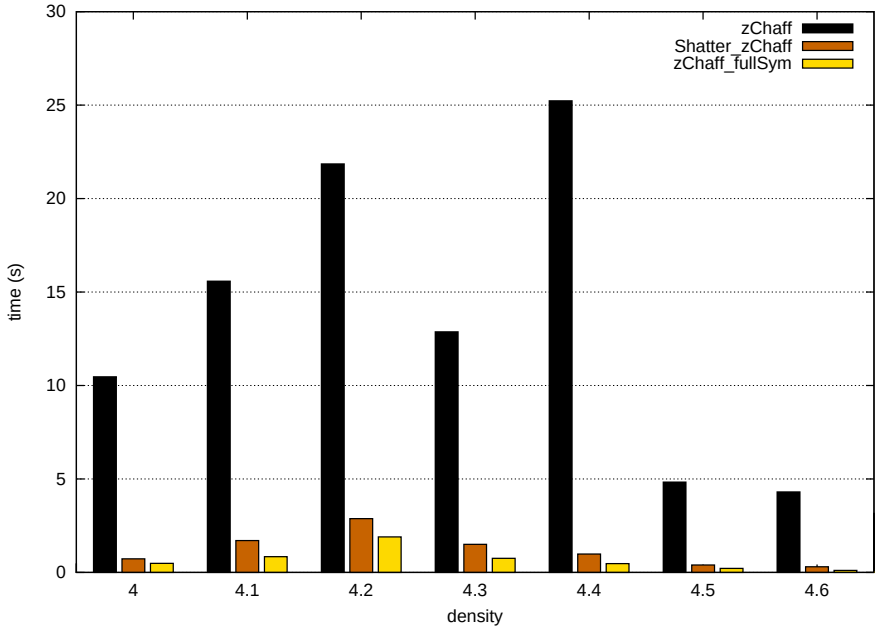

Fig. 8. Histogram of the average time required by the three algorithms to solve unsatisfiable random 4-colouring graphs near the phase transition density.

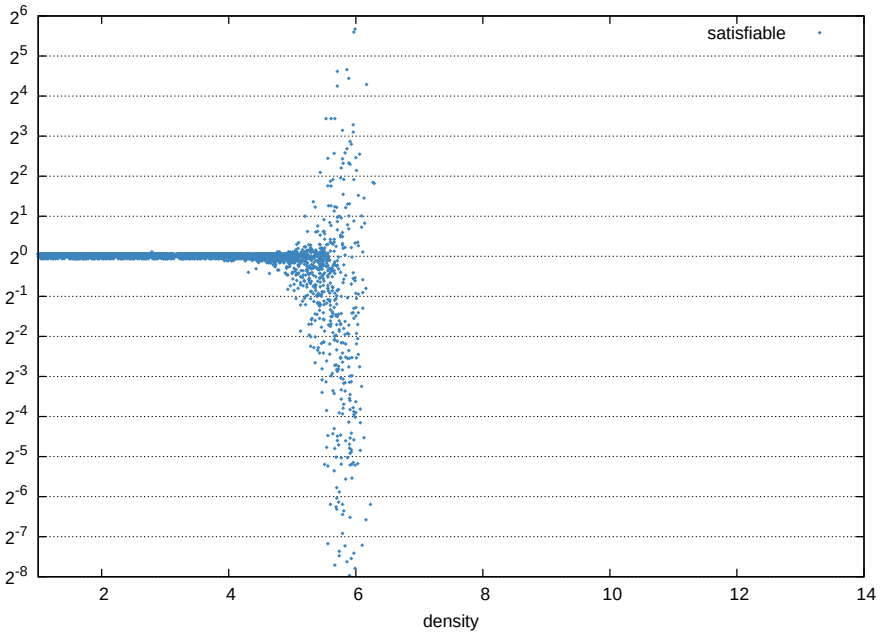

Fig. 9. Scatterplot of $\log_2$ (time(Shatter_zChaff) / time(zChaff_fullSym)) on satisfiable random 5-colouring graphs with 70 vertices.

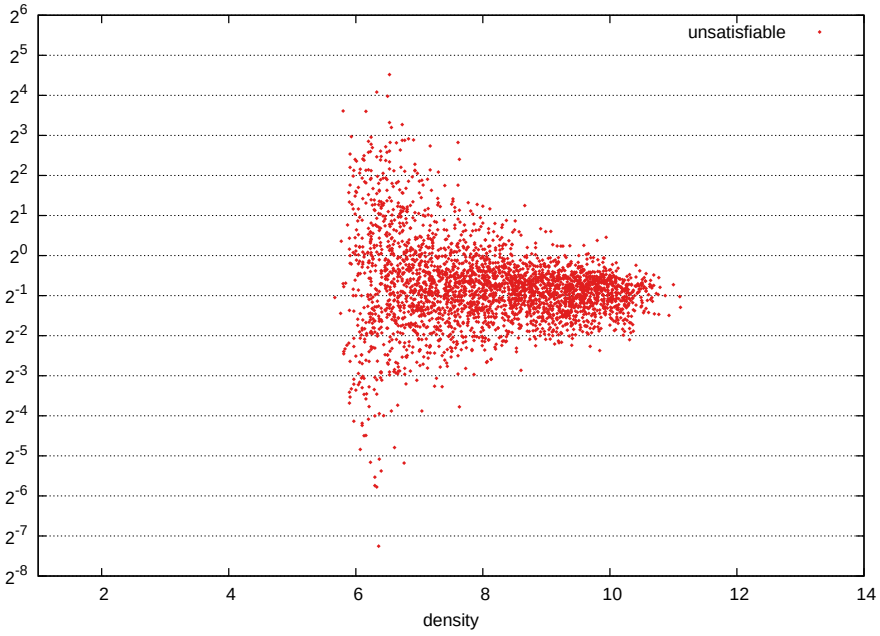

Fig. 10. Scatterplot of $\log_2$ (time(Shatter_zChaff) / time(zChaff_fullSym)) on unsatisfiable random 5-colouring graphs with 70 vertices.

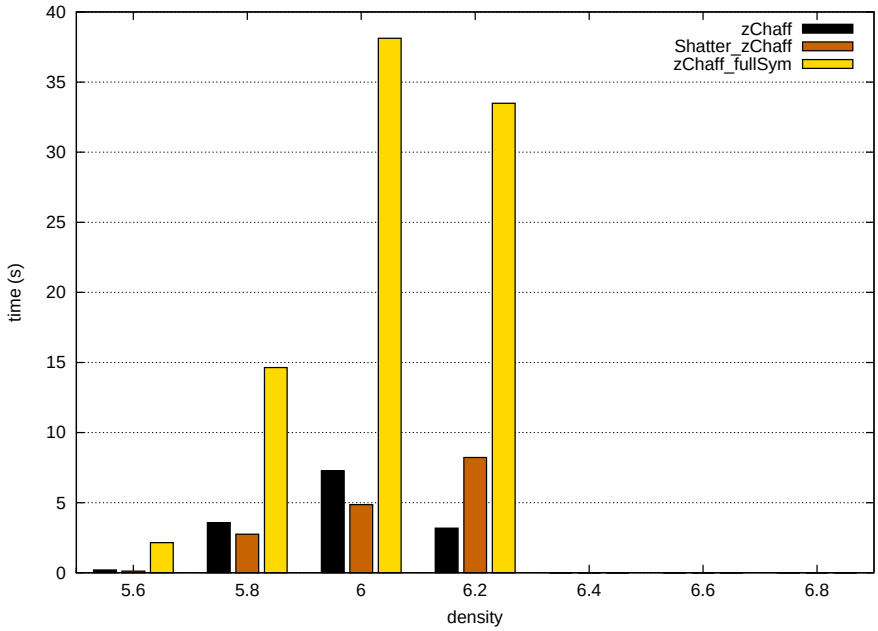

Fig. 11. Histogram of the average time required by the three algorithms to solve satisfiable random 5-colouring graphs near the phase transition density.

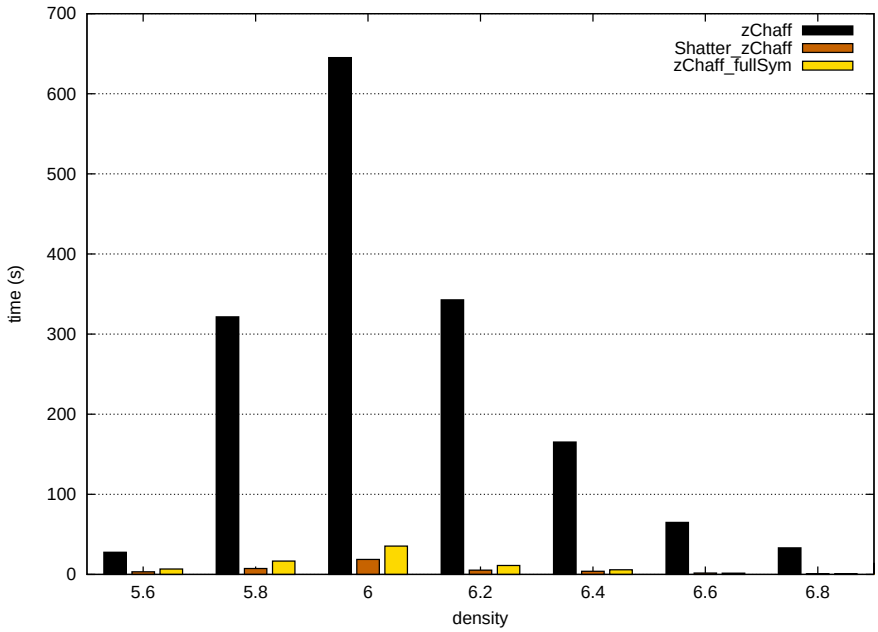

Fig. 12. Histogram of the average time required by the three algorithms to solve unsatisfiable random 5-colouring graphs near the phase transition density.