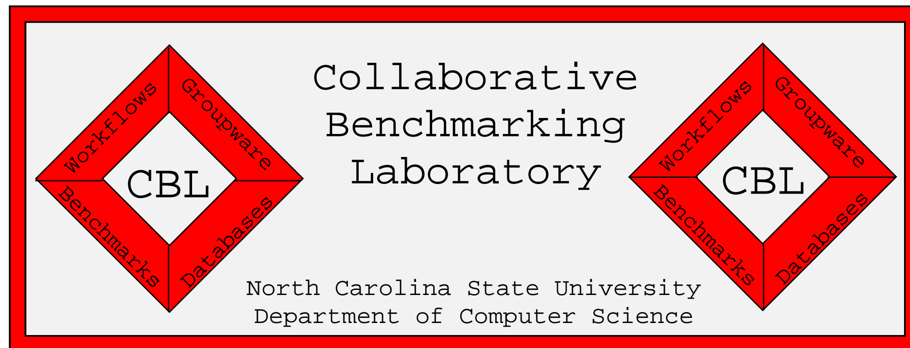


Exploring Satisfiability: Instance Families and Experimental Design

Franc Brglez   Xiao Yu Li   Matthias Stallmann

November 2001

2001-TR@CBL-02-Brglez, Version 1.0

Reprinted, with permission, from

`http://www.cbl.ncsu.edu/publications/`

Publications at this site are occasionally revised,
please check for the latest version under the same title.

For more information about CBL, visit

`http://www.cbl.ncsu.edu/`

or write to

`info@cbl.ncsu.edu`

©2001 authors and CBL, All Rights Reserved

About This Document

This is an interim technical report on the work performed by authors during Summer 2001. Some of the experiments, introduced in this report, are still in progress and an updated version of this report will be published before the end of December 2001. For a preview of web-based organization of data sets and the software environment used in these experiments, see `http://www.cbl.ncsu.edu/OpenExperiments/SAT`.

ABSTRACT

Experimental evaluation of SAT algorithms has been practiced and debated with considerable intensity over the last decade. Typical comparisons involve a collection of unrelated benchmark instances. Algorithms are evaluated on the basis of their total or average execution time for the collection.

We propose a significant departure from this approach. By introducing *equivalence classes* of closely related instances for each *reference formula*, we can experimentally deduce the *95% confidence interval of the mean time-to-solve* and other significantly correlated metrics. Instances derived from the reference formula that are perceived 'hard' by the algorithm under test may exhibit max/min ratios of a metric that can range anywhere from 2 to 1000 and beyond. In such cases, comparisons based on single formulas have no statistical merit.

A total of four class types are formalized and experiments are performed on at least 32 instances in each class. We introduce an experimental design environment and present experimental results that reveal startling, and statistically significant, differentiations between three state-of-the art algorithms and a vanilla DPLL algorithm. As a side benefit, these results provide a number of insights to improve each of these algorithms.

Note. This document has been published for viewing on the Web in PDF and Postscript formats.

"Permission to make digital/hard copy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of CBL. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee."
©2001 authors and CBL, All Rights Reserved

Citation. If you choose to cite this report, please add the following entry to your bibliography database:

```
@techreport{2001-TR@CBL-02-Brglez,
author = "F. Brglez and X.Y. Li and M.F. Stallmann",
title  = "{Exploring Satisfiability:
          Instance Families and Experimental Design}",
institution = "{CBL, Computer Science Dept., NCSU, Raleigh, NC 27695}",
number = "2001-TR@CBL-02-Brglez",
month  = "November",
year   = "2001",
note   = "{Also available at
{\tt http://www.cbl.ncsu.edu/\-publications/\-\#2001-TR@CBL-02-Brglez}}",
}
```

Technical Report 2001-TR@CBL-02-Brglez (November 2001)

Exploring Satisfiability: Instance Families and Experimental Design *

Franc Brglez
Dept. of Computer Science
NC State University
Raleigh, NC 27695, USA
brglez@cbl.ncsu.edu

Xiao Yu Li
Dept. of Computer Science
NC State University
Raleigh, NC 27695, USA
xyli@unity.ncsu.edu

Matthias F. Stallmann
Dept. of Computer Science
NC State University
Raleigh, NC 27695, USA
Matt_Stallmann@ncsu.edu

ABSTRACT

Experimental evaluation of SAT algorithms has been practiced and debated with considerable intensity over the last decade. Typical comparisons involve a collection of unrelated benchmark instances. Algorithms are evaluated on the basis of their total or average execution time for the collection.

We propose a significant departure from this approach. By introducing *equivalence classes* of closely related instances for each *reference formula*, we can experimentally deduce the *95% confidence interval of the mean time-to-solve* and other significantly correlated metrics. Instances derived from the reference formula that are perceived 'hard' by the algorithm under test may exhibit max/min ratios of a metric that can range anywhere from 2 to 1000 and beyond. In such cases, comparisons based on single formulas have no statistical merit.

A total of four class types are formalized and experiments are performed on at least 32 instances in each class. We introduce an experimental design environment and present experimental results that reveal startling, and statistically significant, differentiations between three state-of-the art algorithms and a vanilla DPLL algorithm. As a side benefit, these results provide a number of insights to improve each of these algorithms.

1. INTRODUCTION

The propositional satisfiability problem, SAT, is at the core of the NP-hard problems and has been studied in the context of automated reasoning, computer-aided design, computer-aided manufacturing, machine vision, database, robotics, scheduling, integrated circuit design, computer architecture design, computer networking, etc. The Web has become the universal resource to access large and diverse directories of SAT problem instances [1], SAT discussion forums [2], and SAT experiments [3], each with links to SAT-solvers that can be readily downloaded and installed. Up-to-date survey articles on the SAT problem are also readily available on the Web, e.g. [4, 5].

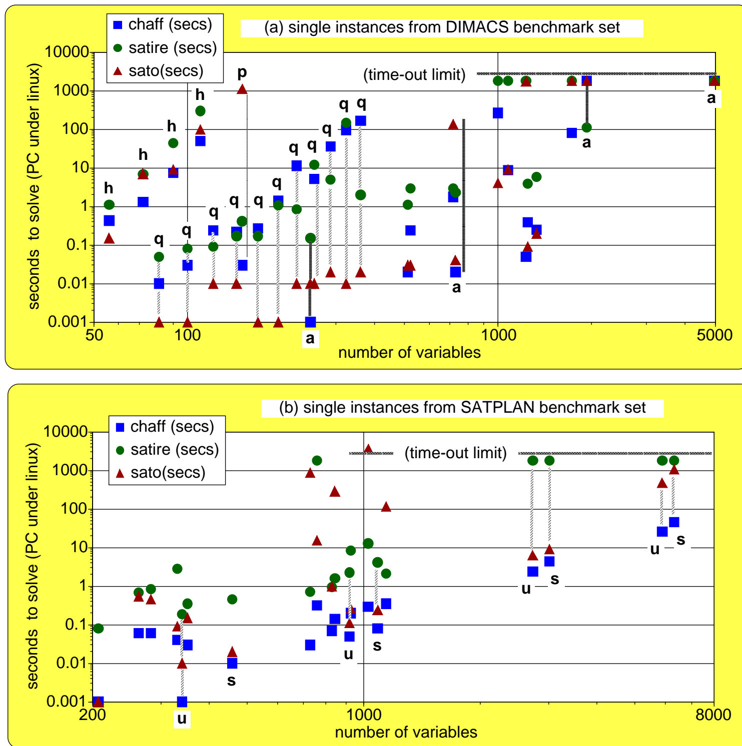
Traditionally, the performance of SAT algorithms has been evaluated experimentally either in terms of randomly generated instances of SAT problems, e.g. [6, 7], or structured instances, such as the instances from the DIMACS set [8] or the SATPLAN set [9]. Merits of either approach are subject to on-going critique and examination [10], [11], [12] in particular, and [13], [14], [15] in general. The papers [13] and [14] succinctly articulate the case for careful experimental design – an approach adopted for the experiments with SAT problems in this paper.

The experimental design methodology presumes availability of well-defined classes of experimental subjects. In our earlier work we identified and created classes of experimental subjects from VLSI circuits and applied them to testing the performance of algorithms ranging from optimization of logic, BDD variable ordering, partitioning, routing and placement, and crossing number in bigraphs [16, 17, 18, 19]. For the SAT problems, formulated as cnf formulas, we introduce in this paper four distinct equivalence classes of instances, each derived by applying well-defined variable permutation and complementation rules. These rules were originally introduced to define equivalence classes of Boolean functions [20], and more recently, to *experimentally* classify intrinsic properties of Boolean graphs in the context of BDD-based variable ordering [21].

The experimental methodology proposed in this paper marks a significant departure from the traditional approach to SAT problem testing, where experimental results are reported for single instances of unrelated formulas. By introducing *equivalence classes* of closely related instances for each *reference formula*, we can experimentally deduce the *95% confidence interval of the mean time-to-solve* and other significantly correlated metrics. Instances derived from the reference formula that are perceived 'hard' by the algorithm under test may exhibit max/min ratios of a metric that can range anywhere from 2 to 1000 and beyond. In such cases, comparisons based on single formulas have no statistical merit.

For each reference formula, we currently create at least two distinct classes with 32 instances in each class. To manage large volumes of data and automated generation

*The experiments, as reported in this paper, could not have taken place without SAT-solvers such as *chaff*, *satire*, and *sato*. We thank authors for the ready access and the exceptional ease of installation of these software packages.



One of the traditional approaches to reporting results of SAT algorithm is the time-to-solve performance of single instances of mostly unrelated benchmarks, such as the ones from the DIMACS set [8] or the SATPLAN set [9]. In addition, the tabulated results are averaged to argue merits of each algorithm.

This paper marks a significant departure from the traditional approach. We apply the experimental design methodology to deduce the most likely average case performance of each algorithm for each reference benchmark, such as the instances marked as 'a' (hanoi), 'h' (hole), 'p' (pret), 'q' (queen) in DIMACS graph. The unmarked instances refer to a subset from the 'ii' and 'quasi' data sets. In the SATPLAN graph, we marked only the instances of 'u' (bw_large_u) and 's' (bw_large_s).

The example of instance `hole10` (110 variables) in this figure reports 48.8, 300.2, and 98.6 seconds for *chaff*, *sato*, and *satire*. Later in the paper, we report the averages, at 95% confidence level, of 451.1 and 182.3 seconds for *chaff* and *sato*, while *satire* times out at 1800 seconds (*satire* starts timing out at 1800 seconds already for the `hole` class instances with 72 variables). In addition, the min/max values for *chaff* and *sato* are 48.8/736.9 and 98.6/207.9 seconds, respectively.

More startling results follow.

Figure 1: Performance experiments with SAT algorithms on *single instances* of mostly unrelated benchmarks.

of *uniform reports* from any SAT algorithm under test, we created a cross-platform test bed, 'doeSAT', that can be readily accessed and installed by other researchers, just as is the case for current data sets and SAT-solvers. By identifying metrics that closely correlate with platform-specific time-to-solve metric, the environment supports repeatability and universal portability of SAT experiments, given that input data sets, and the algorithms under test have not been changed. The new experimental results to date reveal startling, and statistically significant, differentiations between three state-of-the art algorithms *chaff* [22], *satire* [23], and *sato* [24] as well as a vanilla DPLL algorithm [25, 26], implemented as per pseudo-code in [27]. As a side benefit, these results provide (1) a number of insights to potentially improve each of these algorithms, and (2) motivation to design a truly new generation of SAT algorithms in the future.

Paper organization.
(2) Background and Motivation;
(3) Equivalence Classes in CNF;
(4) Experimental Design and SAT;
(5) Reports of Experiments;
(6) Summary and Conclusions.

2. BACKGROUND AND MOTIVATION

One of the traditional approaches to reporting results of SAT algorithms is the *time-to-solve* performance of single instances of a *reference formula* in a conjunctive normal form (cnf). At least two columns are tabulated for each instance in a row: the time-to-solve for a newly proposed algorithm, and the time-to-solve for the next best algorithm from an earlier publication. In this paper we are not proposing a new SAT algorithm, rather we are motivated to critically examine, and to the extent possible, improve the reliability of reporting results on the performance of SAT algorithms.

We begin this study by repeating a number of experiments on well-known subsets of benchmark formulas with three recently reported and readily available programs that implement the following algorithms: *chaff* [22], *satire* [23], and *sato* [24]. All programs have been installed on a PC under the Linux operating system and have been executed without modification of the code or the benchmark instances posted on the web. The time-out limit on how long a program can run on a single instance has been set at 1800 seconds. Rather than reporting time-to-solve results of experiments in the traditional tabular format, we depict them in two graphs in Figure 1. This representation is strictly for the convenience of visualizing the presence of asymptotic trends that suggest grouping a set of benchmarks into a *family*, such as `hole` (marked with 'h'), `queen` (marked with 'q'), `hanoi` (marked with 'a'), `bw_large_u` (unsat, marked with 'u'), and *bw_large_s* (sat, marked with 's').[1]

A quick view of the graph in Figure 1 may suggest the following:

- For the `hole` family, *satire* appears outranked by the other two algorithms; there is a cross-over of *chaff* and *sato*.
- For the `queen` family, *sato* appears to significantly outrank other the two algorithms; there are cross-overs of

[1]Instances of `queen` and `quasi` formulas are not part of the DIMACS set, they are available as part of the *sato* distribution.

chaff and *satire*.

- For the `hanoi` family (marked with 'a'), *satire* is the only algorithm that does not time out on the instance of `hanoi5` (1931 variables), while all three algorithms time out on the instance of `hanoi6` (4968 variables).
- For the `bw_large_u` family, *chaff* appears to outrank the other two algorithms, and *satire* times out for the last two instances (2729 and 5886 variables).

Now, these quick visual observations should not be interpreted as ones that hold any statistical significance. What should be of interest is the *most likely average* time-to-solve each formula in the respective families, and the confidence intervals associated with each of such averages. Such measurements have not been done in the past since there was no well-defined notion nor existence of an *equivalence class* for each formula under test. A simple addition of all reported time-to-solve values, as practiced by tradition in most publications as well as presently on the Web [3] can produce inconclusive results. For example, the `hole` family consists of instances hole6, hole7, hole8, hole9, and hole10. The simple sum-total for the `hole` family

	time-to-solve (secs)		
	chaff	*satire*	*sato*
	...	...	...
sum-total	58.1	352.7	114.8

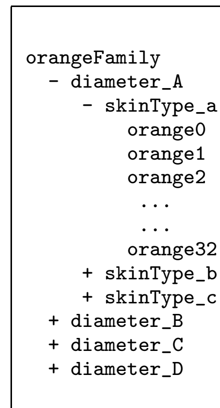
is based on adding five time-to-solve values reported by each algorithm and as such, has no statistical significance since the vales reported, *by the same algorithm*, are orders of magnitude apart. The analysis in this paper reveals that for 32 instances of the reference formula for hole10, time-to-solve can vary, *at least*, from 48.8 to 736.9 seconds for *chaff* , and from 98.6 to 207.9 seconds for *sato*. For *satire*, we can only provide an estimated range of 300.2 to 1800 seconds, since it times out at 1800 seconds. These results do not represent an isolated case of a cnf formula and its equivalence class. Rather, they are representative of a universal phenomenon that will manifest itself each and every time we design the experiments with instances from the same equivalence class.

A significant benefit of testing with equivalence classes, as defined in this paper, will become apparent as we proceed: consistency in cross-platform repeatability of SAT experiments, measured not only with time-to-solve metric but also other, highly correlated, metrics such as the total number of implications.

3. EQUIVALENCE CLASSES IN CNF

Most readers are familiar with the metaphor of 'apples and oranges': one simply is not expected to make a fair comparison between the two – they are 'too different'. We borrow from this metaphor before moving on to the notion of cnf instances from the same equivalence class and an illustrative performance evaluation of four SAT algorithms on these classes.

Oranges and Equivalence Classes. Suppose we want to evaluate two treatments, one in form of a gas, the other in form of a liquid spray, that will extend the shelf life of oranges. One thing is clear, we'll need creates of oranges. *Before* initiating a comprehensive series of experiments, we shall separate oranges into crates in accordance with a well-defined classification scheme, e.g. such as the one shown in Figure 4. Furthermore, we shall require three crates for each diameter and skin type: one for treatment0 (no treatment), one for treatment1 (gas), and one for treatment2 (liquid spray). The significance of treatment0 cannot be overstated, we'll define its counterpart when designing experiments with SAT algorithms. All crates are stored under identical climatic conditions, and all oranges are tested for freshness at periodic intervals. Statistical methods and tools are used to evaluate the effectiveness of each treatment.



An experimental design using oranges may consider the classification scheme on the left. First, we separate oranges by *diameter* and consider diameters $\{A, B, C, D\}$. Within each diameter class, we separate oranges by the *skin type*, e.g. $\{a, b, c\}$. We label a crate with a given diameter and skin type, mark a *reference orange*, orange0, that meets all the requirements of its specified diameter and skin type, and fill the crate with some 32 oranges, each within the specified tolerance range for its diameter and skin type, also marked by its instance number.

Figure 4: An orange family classification scheme and the process of creating experimental subjects.

While we may consider the same treatments on apples, we most likely will need to develop a distinctive classification scheme, specific to apple brands, before designing the experiments with apples.

A CNF Formula and Its Equivalence Classes. To assemble a crate of oranges for a specific equivalence class, we rely on monitors that evaluate the diameter and the skin type, so that oranges in the crate are sufficiently 'similar' with respect to a *reference orange* before the experiment. Still, small variations in diameter and skin type are expected to occur naturally – most likely, they will have a normal distribution.

The hardness of a cnf formula is encoded in the structure its representation itself and the total number of satisfying assignments implied by the function represented by the formula. Both may require exponential time and space to characterize exactly. However, to create an equivalence class of cnf formulas, all of the same hardness as the given *reference formula*, is surprisingly simple. All we need is a mechanism that perturbs the formula while maintaining its structure and the number of satisfying assignments invariant. Such mechanisms are provided by a few simple rules that involve permutation, complementation, and renaming of variables, introduced and illustrated in Figure 2, giving rise to four equivalence classes: **I**-class (identity), **C**-class (complement), **P**-class (permutation), and **PC**-class.

As shown in Figure 2, though the number of satisfying assignments remains the same as that of the reference formula, the solutions for each instance are different, except for instances in the **I**-class. The question, whether each perturbed instance is actually perceived as 'different' by the algorithm applied to the instance, is examined next.

Below is an illustration of the classes used in our experiments. Each class is an equivalence class with respect to the number of satisfying assignments and, to a degree, the structure of the *reference formula* — the original instance from which the class is derived. This reference formula is chosen to represent a heterogeneous distribution of clauses. It has six variables and induces four satisfying assignments. The first two lines show the formula and a solution vector for each of the four satisfying assignments (indexed from left to right starting with variable 1).

Each formula is shown as a list of clauses and each clause is a set of literals. A literal is either a positive integer i, denoting the variable x_i, or $-i$, denoting $\overline{x_i}$, the complement of x_i.

The four solution vectors for each instance are lined up in the same column as the corresponding solution vector for the reference formula.

Descriptions of the four classes follow.

- **I**-class (identity) — variable names and function values are preserved; however, the order of appearance of clauses and of literals within a clause may be permuted arbitrarily.
- **C**-class (complement) — variable names are preserved; a subset of the variables is complemented (each occurrence of such a variable is complemented) and both variables and clauses are permuted. The first formula in the example complements 1 and 5; the others complement $\{3, 5\}$, $\{1, 2, 6\}$, and $\{1, 2, 4\}$, respectively, as can be seen from the solution vectors.
- **P**-class (permutation) — variable names are permuted arbitrarily, as are clauses and literals. The four permutations are 462513, 561342, 531642, and 456312, as reflected in the solution vectors (each vector has the same number of true variables as the one it is derived from).
- **PC**-class formulas — variable names are permuted *and* variables are complemented; clauses and literals are also permuted. The first instance uses permutation 246531 and then complements $\{2, 4\}$; the remaining instances use 416253 with $\{2, 4, 5, 6\}$, 614532 with $\{1, 3, 5, 6\}$, and 512346 with $\{2, 3, 4\}$, respectively.

Reference formula and solution vectors

```
0: {-1 -2} {-1 -3} {-1 -4} {-1 -5} {-1 -6} {-2 -3} {-2 -4} {-3 -4} {-5 -6} {1 2 3} {4 5 6} {-1 2 -3 4}
```

```
0:    001001    001010    010001    010010
```

I-class formulas and solution vectors

```
1: {-1 -5} {-4 -2} {-1 -3} {-1 -2} {-4 -1} {2 3 1} {-2 -3} {5 4 6} {2 -1 4 -3} {-4 -3} {-5 -6} {-1 -6}
2: {-4 -2} {4 -1 -3 2} {1 2 3} {-3 -2} {-6 -5} {-1 -3} {-1 -2} {-4 -1} {-6 -1} {-3 -4} {6 4 5} {-1 -5}
3: {6 5 4} {-3 -2} {-4 -3} {-1 -5} {-1 -3} {-6 -5} {2 -3 -1 4} {3 1 2} {-1 -4} {-2 -4} {-6 -1} {-1 -2}
4: {-1 -2} {-1 -5} {-3 -4} {5 6 4} {-5 -6} {4 -1 -3 2} {-1 -4} {-2 -4} {-1 -3} {-3 -2} {3 1 2} {-6 -1}
   ..........
```

```
1:    001001    001010    010001    010010
2:    001001    001010    010001    010010
3:    001001    001010    010001    010010
4:    001001    001010    010001    010010
```

C-class formulas and solution vectors

```
1: {4 -5 6} {1 -4} {-6 1} {-2 -3} {-6 5} {-4 -2} {2 -3 4 1} {-1 3 2} {1 -3} {-3 -4} {5 1} {1 -2}
2: {-2 -1} {-1 -4} {-6 -1} {-2 3} {2 -1 4 3} {-4 3} {-1 5} {-6 5} {-2 -4} {-1 3} {6 -5 4} {1 2 -3}
3: {-4 -3} {-3 1} {1 2} {6 1} {-5 1} {-3 2} {1 -4} {6 -5} {4 -6 5} {2 -4} {1 4 -3 -2} {-1 3 -2}
4: {2 -3} {4 -3} {-6 1} {-3 1} {3 -1 -2} {2 1} {-4 -3 1 -2} {-5 1} {2 4} {-6 -5} {6 5 -4} {1 4}
   ..........
```

```
1:    101011    101000    110011    110000
2:    000011    000000    011011    011000
3:    111000    111011    100000    100011
4:    111101    111110    100101    100110
```

P-class formulas and solution vectors

```
1: {-4 -2} {-5 -6} {-3 -1} {-5 -4} {-6 -2} {1 3 5} {-4 -1} {6 2 4} {5 -4 -2 6} {-4 -3} {-5 -2} {-4 -6}
2: {-2 -4} {-3 -6} {6 1 5} {2 4 3} {-3 -1} {-5 -4} {-1 -5} {-1 -6} {6 3 -5 -1} {-6 -5} {-5 -2} {-3 -5}
3: {-4 -2} {-6 -1} {-2 -5} {3 -5 6 -1} {-6 -5} {-5 -3} {-3 -1} {6 2 4} {-1 -5} {-6 -3} {-4 -5} {5 1 3}
4: {-4 -2} {-5 -3} {-3 -4} {-4 -5} {3 2 1} {-6 -5} {5 6 4} {-6 -3} {-4 -6} {-4 3 5 -6} {-2 -1} {-4 -1}
   ..........
```

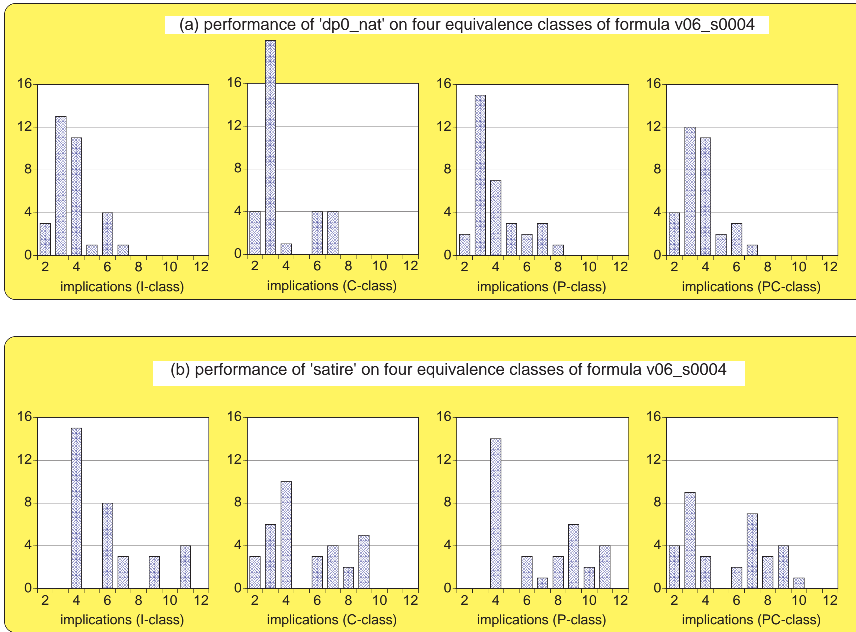
```
1:    011000    110000    001001    100001
2:    110000    100100    010001    000101
3:    110000    100100    011000    001100
4:    010001    100001    010010    100010
```

PC-class formulas and solution vectors

```
1: {-5 4} {-6 5 2 -4} {2 -3} {-6 2} {2 -5} {4 2} {-6 -5} {-2 6 -4} {-1 2} {-3 -1} {5 3 1} {4 -6}
2: {4 2} {5 4} {1 -4 -6} {4 -3} {2 6} {-5 -2 3} {6 4} {-2 6 4 1} {-3 5} {4 -1} {2 -1} {-1 6}
3: {2 -3 -5} {1 5} {-4 -5 6 -1} {5 -4} {6 1} {4 -6 -1} {-4 1} {6 3} {6 5} {-4 6} {-2 6} {-2 3}
4: {-3 -4 6} {-1 2} {4 -6} {-2 5 1} {-1 3} {-5 3} {3 2} {-5 -6} {-5 -1} {-5 2} {4 -5} {-3 -5 1 2}
   ..........
```

```
1:    110101    011101    110000    011000
2:    011110    000101    111111    110101
3:    001001    010001    101101    110101
4:    001101    001000    111101    100100
```

Figure 2: Reference formula for function v06_s000004 and formula instances from four equivalence classes.



When we invoke a SAT algorithm on a 6-variable cnf formula, such as on instances from classes of v06_0004 in Figure 2, a time-to-solve metric will not differentiate between the various class instances.

As shown here, measuring the number of implications clearly differentiates between the two algorithms even on this small illustrative data set. The algorithm 'dp0_nat' is a vanilla implementation of the DPLL algorithm [25, 26], also briefly discussed in this paper. The algorithm 'satire' is described in [23].

Noticeably, even a set of identical or closely related 6-variable cnf formulas can induce variability in performance: from 2 implications/solution to 11 implications/solution. In our experiments, we discovered, on much larger formulas, comparable and *much larger* max/min performance ratios, both in time-to-solve and number-of-implications.

Figure 3: Performance histograms of two SAT algorithms applied to instances in four equivalence classes.

CNF Equivalence Classes and SAT Algorithms. To perform experiments with oranges, we require a *new crate* of oranges, from the same class, for each treatment. Once treated, the same crate of oranges cannot be re-used for freshness experiments. In contrast, different SAT algorithms can be applied to instances from the same equivalence class any number of times, with no concern of 'contaminating' the data.

In addition to the three SAT algorithms introduced earlier, *chaff*, *satire*, and *sato*, we introduce one more: *dp0_nat*. The latter is based on a vanilla DPLL algorithm [25, 26], implemented in tcl [28] as a recursive procedure per pseudocode in [27]. In order to make the implementation as unbiased as possible, we deliberately implemented a simple branching rule for its variables: each variable is selected in the order it is encountered in the clause list, if the variable is complemented, it is assigned a value of 'false', and 'true' otherwise. Since variables and clauses are permuted in each instance, the variable selection by *dp0_nat* is thus in random order. As such, the application of *dp0_nat* is analogous to treatment0 (no treatment) in the experiments described for the equivalence classes of oranges. Here, we have decided that the vanilla DPLL algorithm represents the 'control' against which all other algorithms are to be compared. Similar

All four algorithms have been applied to all four classes of the simple cnf formula introduced in Figure 2. The results are summarized in Figure 3 and Table 1. It is important to note that, except for the algorithm *chaff*, responses foretell the responses we observe for instances of much larger formulas in subsequent experiments with equivalence classes. Overall, we make the following brief observations:

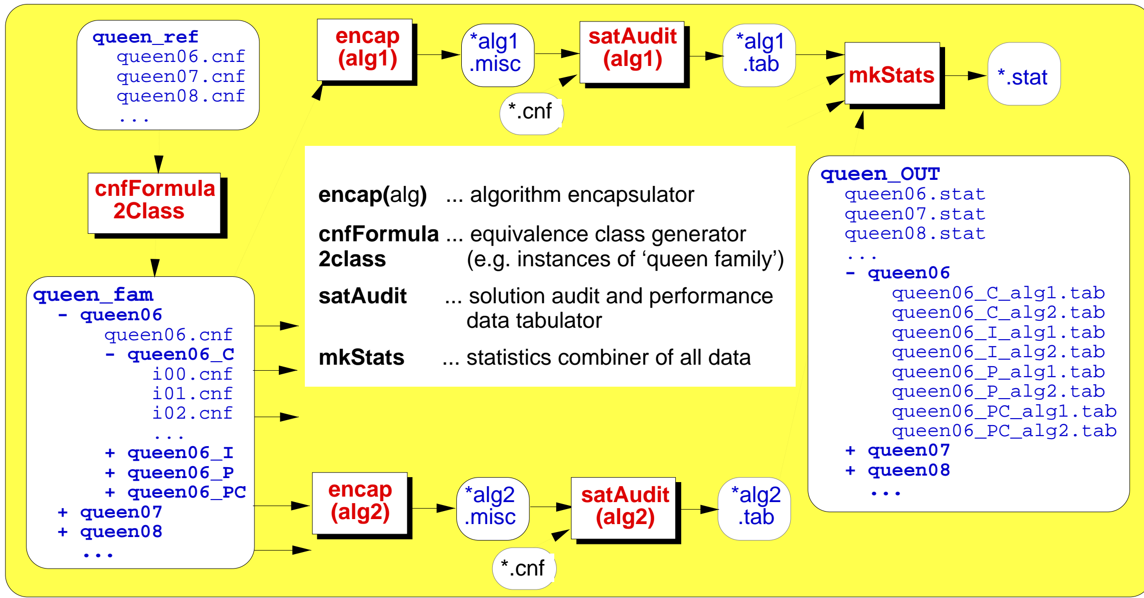
- For the algorithm *chaff*, the variance of 0 for **P**- and **PC**-class is uncharacteristic; large variances are observed for both of these classes in general, compared to variances for **I**- and **C**-class.
- For the algorithm *satire*, the near-equal variance for all four classes is characteristic.
- For the algorithm *sato*, the variance of 0 for **I**- and **P**-class is characteristic; relatively small variances are observed for both of these classes in general, compared to much larger variances for **C**- and **PC**-class.
- For the algorithm *dp0_nat*, the near-equal variance for all four classes is characteristic, and contrary to this example, it can exceed the variance of other algorithms.

Table 1: Number-of-implications statistics for equivalence classes of formula v06_0004.

equiv. class	*chaff* mean/std	*satire* mean/std	*sato* mean/std	*dp0_nat* mean/std
I	6.00/0.00	5.97/2.42	3.00/0.00	3.81/1.26
C	6.00/0.00	5.06/2.29	4.84/2.03	3.78/1.66
P	6.00/0.00	6.69/2.73	3.00/0.00	4.06/1.56
PC	6.00/0.00	5.28/2.56	4.66/2.07	3.75/1.24

Since some algorithms are clearly designed to suppress the variability of performance induced by instances from some of the classes, the only class where all algorithms can be compared fairly is the **PC**-class. However, all four classes are necessary to completely describe the properties of the algorithm under test. The presence of bias in the algorithm can decrease its performance potential – both *chaff* and *sato* exhibit erratic behavior on some of the larger formula classes described later, when compared to either *satire* or *dp0_nat*.

On SAT Cost Distribution. Since instances from the four equivalence classes introduced in this section effectively



A single cross-platform tcl script 'doeSAT' creates an experimental design environment to execute all experiments reported in this paper. The standard components include: (1) equivalence class generator of cnf formulas (cnfFormula2Class), (2) an encapsulator for each program that invokes the algorithm under test (encap), (3) an audit program to parse and verify the solution report generated by each algorithm (satAudit), (4) a summary report generator that generates statistics summary of all experiments (mkStats). Notably, the only component specific to the doeSAT environment is satAudit. For each algorithm added to the doeSAT environment, we appropriately extend the parsing capability of the satAudit program.

The primary user-defined input is a set of reference cnf formulas such as shown under 'queen_ref'. The class instances such as i00.cnf, i01.cnf under queen06_C remain archived for universal access on the Web. Similarly archived on the Web are the experimental result, shown here under queen_OUT. Histograms and correlation plots in this paper are based on tabbed data columns in *.tab, graphs of asymptotic performance are based on statistics summaries in *.stat. Any number of experiments can be repeated without re-invoking the equivalence class generator.

Figure 5: Experimental design flow and data structures used to evaluate the performance of SAT algorithms.

represent the same formula, the ideal algorithm should return a solution with the same or nearly the same cost, regardless of which instance has been chosen, i.e. the cost distribution should be normal with zero or near-zero variance. An exceedingly large variance is an indicator that the *algorithm is behaving erratically* on few or several instances from the class – and *not* that the choice of the instance is inappropriate!

4. EXPERIMENTAL DESIGN AND SAT

The experimental design methodology presumes availability of well-defined classes of experimental subjects. Four such classes have been defined in the preceding section. As noted in Figure 1, to depict asymptotic behavior of an algorithm for a family of formulas, we may need to consider some five or more appropriate reference formulas and create up to four equivalence classes for each formula. For statistical significance, we may consider at least 32 instance in each class. Thus, the number of experiments to statistically analyze the asymptotic behavior of a family based on 5 reference formulas across 4 classes requires a total of $5 \times 4 \times (32 + 1) = 660$ experiments per algorithm.[2] For routine testing, we may perform experiments with PC-class only, reducing the number of experiments to more manageable 135 per algorithm.

To manage large volumes of data and automated generation of *uniform reports* from any SAT algorithm under test, we created a cross-platform test bed, 'doeSAT'. Its major components and the flow of execution are introduced in Figure 5. The standard components include:

cnfFormula2Class: equivalence class generator of cnf formulas. Currently, up to four classes described in the previous section are generated. The primary user-defined input is a set of reference cnf formulas such as shown under `queen_ref`. It should not come as a surprise that the directory structure of classes generated under 'queen_fam' resembles the one introduced for the hypothetical classification of oranges in Figure 4. Once a family of classes is generated, a compressed directory is maintained on the Web.

encap: an encapsulator for each program that invokes the algorithm under test. This is a standardized script that can be readily customized for the I/O of the algorithm invoked within the doeSAT environment.

satAudit: an audit program to parse and *verify* the solution report generated by each algorithm. This is the only component specific to the doeSAT environment. For each algorithm added to the doeSAT en-

[2]The number of experiments per class of 32 is actually 33 since, for each class, the experiment always includes the reference formula itself (in its original form).

vironment, we appropriately extend the parsing capability of the satAudit program. After parsing the algorithm-specific output report, including the solution (if found), the solution is verified and a report generated in a standardized format that can be read by a plotting program or processed further by a statistics generator program mkStats, described below. While each SAT solver outputs a report in a solver-specific format, the major feature of the satAudit program is to generate a report for each algorithm that contains parameters reported by all other algorithms. Currently, we report in a common tabular format, for each instance and each algorithm the following common parameters: solution status, solution (if generated), time to solve, number of implications, number of decisions, max decision level (chaff), number of backtracks (others). Our experimental reports demonstrate that time to solve and number of implications correlate closely and consistently.

mkStats: a report generator that generates a statistics summary of all experiments. This is a standard program that reads the standardized data generated by various instances of satAudit program and generates a statistics summary in of all experiments, for all equivalence classes and for all algorithms under test. Statistics are based on the number of instances in each equivalence class and are reported for all common parameters described above: median value, mean value, standard deviation, minimum value, maximum value, 95% confidence interval of the mean (based on the *t*-statistics [29]), and the standard coefficient of variation. This list can be extended in the future.

The report also includes the *initial value* as a statistics that refers to the parameters evaluated for the reference formula. Ideally, the initial value will always be bounded by the minimum and maximum statistics reported for the class. However, as shown in tabulated summaries of experiments later, a SAT algorithm under test can induce a min/max range that does not contain the initial value. Since the reference formula and all instances are in the same equivalence class, we extend the min/max range by including the initial value in the range. Occasionally, an erratic behavior of the algorithm may induce a distribution that is far from normal, exhibits a large variance, and renders the confidence interval of the mean invalid. It should be clear that such distributions are a reflection on the algorithm under test, *not* on the instances in the equivalence class!

The above components form a system that supports the experimental design methodology. The key goals of that system are

- to facilitate the use of the experimental design methodology by us and other researchers,
- to allow easy replication of our current and future experimental results,
- to encourage other researchers to participate by contributing reference formulas and algorithms, and
- to automate the experiments in a way that minimizes the potential for human error.

5. REPORTS OF EXPERIMENTS

Our series of SAT experiments started by replicating a number of experiments on well-known subsets of benchmark formulas, using three readily available programs. Results of these experiments are discussed in Section 2 and summarized in Figure 1. These experiments also identify reference formulas of structurally-related instances of increasing size that can be grouped into *families* so we can study the asymptotic performance of SAT algorithms. Ideally, reference formulas in each family should be either all unsatisfiable or all satisfiable, the basic structure of clauses should be the same, and size increase from one formula to the next should be (roughly) the same multiplicative constant. Whenever possible, at least five of such reference formulas should define a family.

As part of this paper, we also created two families (unsatisfiable and satisfiable) of well-structured formulas, both suitable to study the asymptotic performance of SAT algorithms. Overall, the families of reference formulas we use in the creation of the equivalence classes in this paper include:

- `hole` (6 unsatisfiable formulas from DIMACS)
- `queen` (5 satisfiable formulas from *sato* distribution)
- `hanoi` (4 satisfiable formulas from DIMACS)
- `bw_large_u` (4 unsatisfiable formulas from SATPLAN)
- `bw_large_s` (4 satisfiable formulas from SATPLAN)
- `sched_u` (5 unsatisfiable formulas, this paper)
- `sched_s` (5 satisfiable formulas, this paper)

Having created and analyzed instances of **I**-, **C**-, **P**-, and **PC**-equivalence classes from `hole` and `queen` families, we recognized that in order to compare all algorithms on an equal basis, it is sufficient to create only **P**- and **PC**-classes, hence for the remainder of families, only these two classes have been created and analyzed.

We now proceed to brief description of key parameters that define reference formulas in each family, and present the experimental results. Without exception, there are always 32 instances in each equivalence class that we analyze. Asymptotic trends in this paper are reported with respect to average performance within the **PC**-classes of the instances of a family, and where we need to bring out an important point, also **P**-classes. All families we investigated illustrate key differences among algorithms and trends for further study.

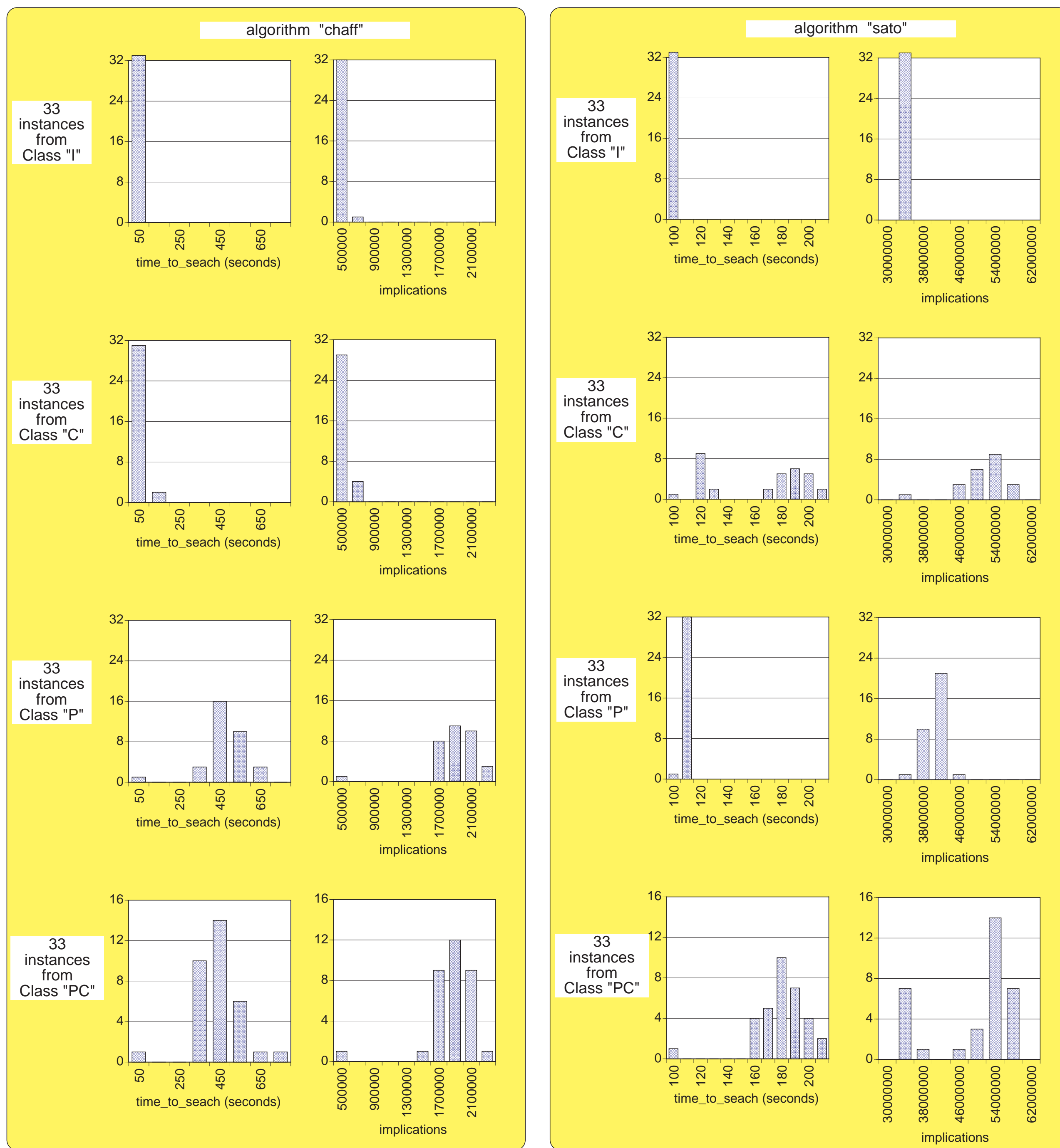
The hole family. The `hole` family consists of formulas that represent the pigeon hole principle. All formulas are unsatisfiable and the instances derived from the formula for `hole14` time out at 1800 seconds for all algorithms. As noted here, an instance need not represent thousands of variables. Instances with only 210 variable provide a non-trivial challenge.

instance	vars	clauses
`hole6`	42	133
`hole7`	56	204
`hole8`	72	297
`hole9`	90	415
`hole10`	110	561
`hole14`	210	1485

The unsatisfiable reference formulas in the `hole` family originated with the DIMACS distribution [8].

The formula for hole10 has 110 variables and is the largest

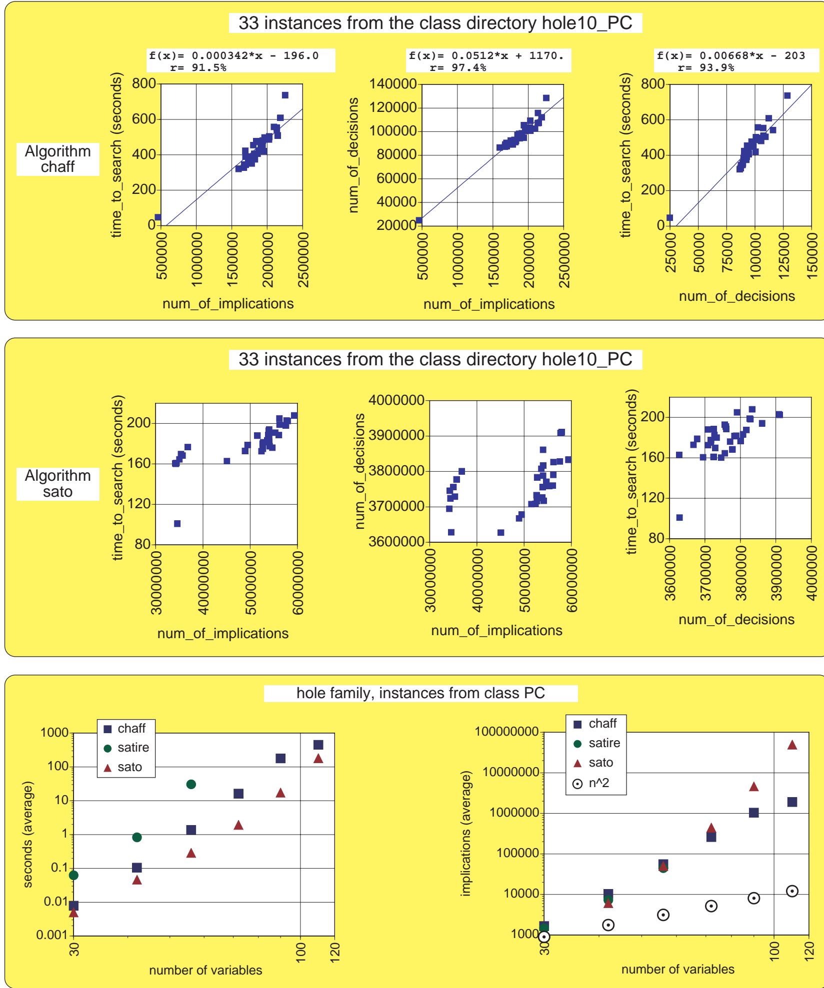
The formula for hole10 has 110 variables and is the largest reference formula from the ‘hole family’ where algorithms chaff and sato, applied to instances of classes ‘P’ and ‘PC’, do not time out at 1800 seconds. The algorithm satire starts timing out at 1800 seconds already for the ’hole’ class instances with 72 variables. Observing the histograms below, it is clear that the only class where algorithms chaff and sato *should* be compared is the ‘PC’-class. No t-test is required to confirm that, for hole10, the algorithm sato has the better average case performance, in term of time-to-search or number-of-implications. Such conclusions cannot be derived on basis of results in Figure 1 alone.



Only the instances in the P and PC class induce significant variability in the performance of the chaff algorithm. A fair comparison with the sato algorithm can only be made on basis of experiments with the PC class.

Only the instances in the C and PC class induce significant variability in the performance of the sato algorithm. A fair comparison with the chaff algorithm can only be made on basis of experiments with the PC class.

Figure 6: Performance histograms of two SAT algorithms applied to instances in four equivalence classes of ’hole10’ from the ’hole’ family.



Histograms in Figure 6 suggest a correlation between time-to-solve and number-of-implications metrics. This is indeed the case, not only for the example of the hole10 PC-class as shown here, but universally for any class. Moreover, correlation coefficients for the time-to-solve and the number-of-implications metrics are even better for algorithms dp0 and satire than the ones shown for chaff – above 99% or approaching 99%. Correlations for algorithm sato are split between two regions; a phenomenon that may be attributed to its trie data structure.

Note also: (1) the min/max ranges in time-to-solve (48.8/736.9 seconds for chaff vs 98.6/207.9 seconds for sato) mirrored by number-of-implications (458,227/2,253,460 for chaff vs 34,571,370/59,355,590 for sato); (2) about 4000 implications/second and 20 implications/decision for chaff, and a range of 200,000 – 350,000 implications/second and 9–13 implications/decision for sato.

The asymptotic performance of chaff, satire, and sato is shown for the average time-to-solve and the average number-of-implications. Significantly, the average number-of-implications statistics brings out more clearly the significant difference between the three algorithms, sato and chaff in particular (satire times out for cases where number of variables is equal to or exceeds 72). With the average number-of-implications for sato rising faster than for chaff, we can anticipate a cross-over in time-to-solve performance of the two algorithms (for the hole family class) – if we raise the time-out limit above 1800 seconds.

Figure 7: Performance correlations of two SAT algorithms applied to instances from the hole10_PC class and the asymptotic performance statistics for 'hole' family of PC equivalence classes.

reference formula from the 'hole family' where algorithms *chaff* and *sato*, applied to instances of classes 'P' and 'PC', do not time out at 1800 seconds. The algorithm satire starts timing out at 1800 seconds already for the 'hole' class instances with 72 variables. Observing the histograms in Figure 6, it is clear that the only class where *chaff* and *sato* can be compared is the 'PC'-class. No *t*-test is required to confirm that, for `hole10`, the algorithm *sato* has the better average case performance, in term of time-to-search or number-of-implications. Such conclusions cannot be derived on basis of results in Figure 1 alone.

Moreover, the histograms in Figure 6 suggest a correlation between time-to-solve and number-of-implications metrics. This is indeed the case, not only for the example of the hole10 PC-class as shown in Figure 7, but universally for any class. With very few exceptions, noted in the figure, correlation coefficients approaching 99% are the norm. Figure 7 also shows the asymptotic performance of *chaff*, *satire*, and *sato* for the average time-to-solve and the average number-of-implications, with important observation in listed in the text box.

For a complete statistical summary about this family, see also Table 2.

The queen family. The `queen` family consists of formulas that represent the classical queen problem. All formulas are satisfiable.

instance	vars	clauses
queen9	81	1065
queen10	100	1480
queen14	196	4200
queen16	256	6336
queen19	361	10735

The satisfiable reference formulas in the `queen` family originated with the *sato* distribution [24].

In Figure 8 we show the asymptotic performance of this family not only for *chaff*, *satire*, and *sato* but also for *dp0_nat*. With few, but significant exceptions, the asymptotic performance of these algorithms follows trends that are similar to ones described for the `hole` family in Figure 7. The sig-

nificant exceptions are all listed in the text box, the most startling one being the extreme sensitivity of sato to variable complementation – a fact observed already on the 6-variable example in Figure 3.

For a complete statistical summary about this family, see also Table 3.

The hanoi family. The `hanoi` family consists of formulas that represent the classical tower of Hanoi problem. All formulas are satisfiable.

instance	vars	clauses
hanoi3	249	1512
hanoi4	718	4934
hanoi5	1931	14468
hanoi6	4968	39666

The satisfiable reference formulas in the `hanoi` family originated with the DIMACS distribution [8].

The really challenging formulas are those of `hanoi5` and `hanoi6`. We were surprised to find that *satire*, one of the slower programs overall, found a solution to `hanoi5` in some 100 seconds while both *chaff* and *sato* timed out at 1800 seconds. However, running all three algorithms on instances of **P**- and a **P**-class, *not a single solution* was found by any of the three algorithms.

The bw_large families. The unsatisfiable and the satisfiable families of `bw_large` are a subset of the SATPLAN benchmarks, a collection of satisfiability problems based on AI planning scenarios developed in [9]. The most difficult of these come from the well-known blocks-world domain in AI (see, e.g. [30]). We used two families of blocks-world instances in our experiments, one satisfiable, one unsatisfiable. Only three instances of each family were within reasonable size range. Sizes of these are shown in the table below.

instance	vars	clauses
bw_large_a_u	340	3,294
bw_large_b_u	920	11,491
bw_large_c_u	2729	45,368
bw_large_d_u	5886	122,412
bw_large_a_s	459	4,675
bw_large_b_s	1087	13,772
bw_large_c_s	3016	50,457
bw_large_d_s	6325	131,973

The reference formulas in the `bw_large` family originated with the SATPLAN benchmark set [9].

The satisfiable blocks-world instances illustrate the sensitivity of *sato* to complementation more starkly than any other family we tested. See Table 4. For the P class of the largest instance *chaff* does only slightly better than *sato*. But *sato* completely loses control over the number of implications (and execution time) on the PC class. The reference formula, whose data point is well within the confidence interval of the P class, is two orders of magnitude better than a typical PC class instance. For the unsatisfiable instances, even the P class appears to confound *sato* – again, it does atypically well on the reference formula.

The sched families. The unsatisfiable and the satisfiable families of `sched` have been created as part of the experiments reported in this paper. These formulas are based on unit-length-task scheduling instances. Scheduling problems with unit-length tasks have numerous applications in computer science, management, and industrial engineering (see [31] and [32] for a survey). Many variations involving release times, deadlines, resource constraints, and precedence constraints are NP-complete (see, e.g. [31] and [33]). It is relatively simple to formulate the existence of a feasible schedule under these kinds of constraints as a cnf formula — each variable represents assignment of a task to a specific slot, two-literal clauses rule out conflicting assignments, long positive clauses guarantee that each task is assigned, and Horn clauses are used to express precedence constraints. Unit-task scheduling therefore offers a rich domain of future satisfiability benchmarks. We created two families, one satisfiable and one not, of scheduling instances with deadlines and precedence constraints. The precedence graphs were the same for both families, hierarchical structures based on *N-graphs*, forbidden subgraphs of vertex-series-parallel dags — see, e.g. [34] for further discussion. Deadlines were designed so that the satisfiable instances had only a small number of feasible solutions based on scheduling along specific critical paths first. The unsatisfiable instances differed only in the deadlines of two tasks, making them "barely infeasible". Sizes of the scheduling instances are shown below.

instance	vars	clauses
sched03u	93	493
sched04u	138	890
sched05u	410	4,336
sched06u	826	12,022
sched07u	1,384	25,669
sched03s	95	495
sched04s	140	892
sched05s	412	4,338
sched06s	828	12,024
sched07s	1,386	25,671

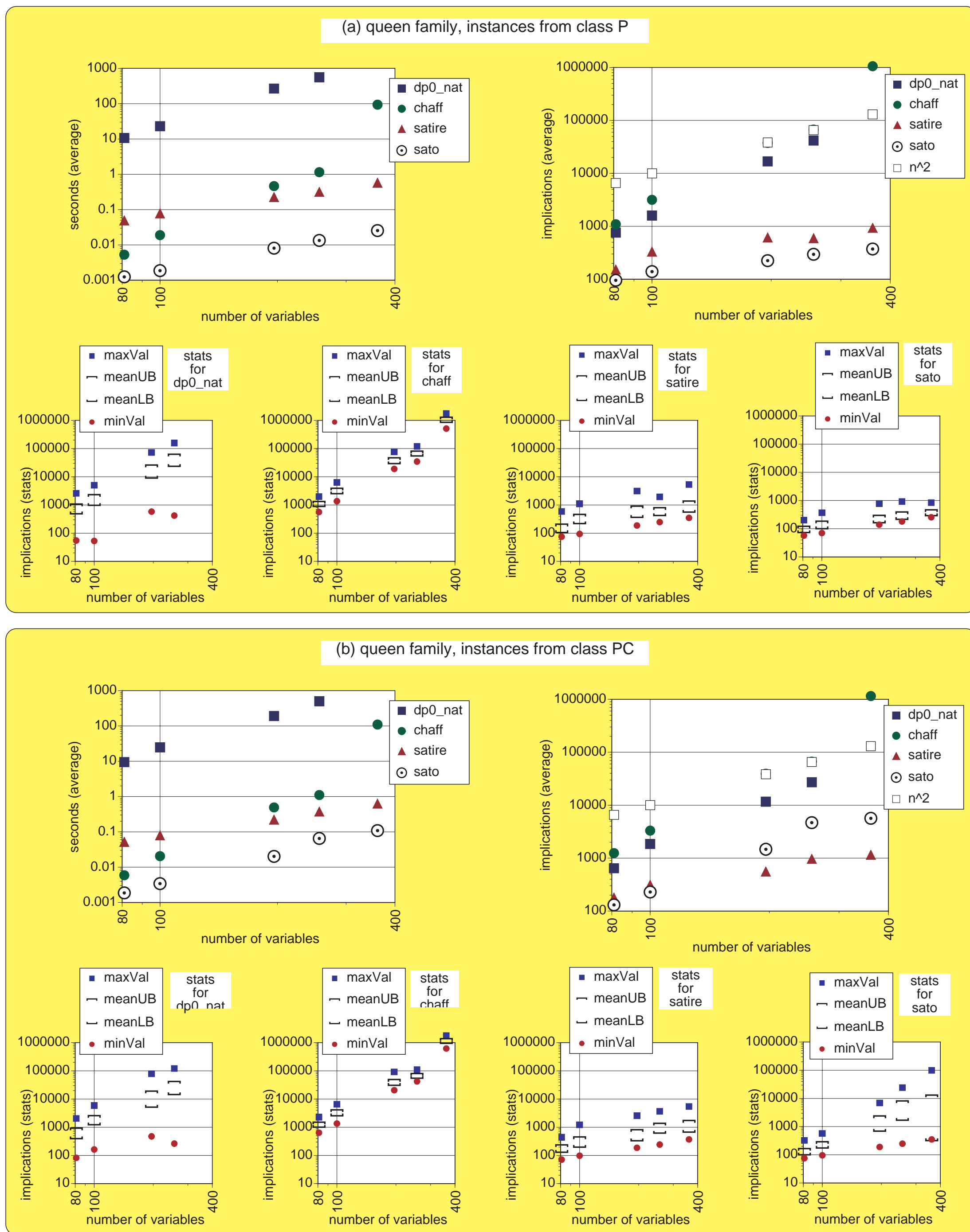
In contrast to the blocks-world and hole instances, the scheduling instances are a major success story for *sato*. See Table 5. On the P classes of satisfiable instances, *sato*'s heuristics for selecting variables were foolproof — the same solution was always identified in the same small number of steps. Other classes also show little variance – so neither complementation nor the lack of a solution have much effect. The chaff heuristic, on the other hand, not only performs orders of magnitude worse than sato. It also exhibits two orders of magnitude difference in its best and worst performance on the larger instances. Clearly, an algorithm that performs relatively well in one domain may do quite poorly in another.

6. CONCLUSIONS

As already noted, our experiments have revealed significant differences in relative algorithm performance for different instance families, different instance classes within the same family, and even for different instances within the same class. These differences, dramatic enough to raise questions about much of the previous experimental work on the satisfiability problem, could not have been detected in any traditional single-instance benchmark-driven environment.

What conclusions can we draw and how shall we proceed?

- For the algorithm engineer the bad news is that there is no simple answer to the question, "what is the best overall algorithm/strategy for solving the satisfiability problem?"
- There is good news, however — our results and our experimental setup provide a clear path to the design and testing of future enhancements. Specifically, the weaknesses we have demonstrated in each of the leading algorithms are likely to be ones that their designers



With few exceptions as noted below, the asymptotic performance of *dp0_nat*, *chaff*, *satire*, and *sato* for the ‘queen family’ follows trends that are similar to ones described for the ‘hole family’ in Figure 7. Significant observations for this series of experiments include:

- asymptotic performance in number-of-implications of *dp0_nat* (coded in interpreted language tcl [28]) is better than that of *chaff* (a well-designed C-implementation of *dp0_nat* could therefore yield a performance faster than *chaff* on problems in ‘queen family’).
- the variability in min/max performance is significantly larger than already large variability found for the non-satisfiable problems in ‘hole family’. The same observation applies to minUB/minLB, the 95% confidence level interval of the respective mean values.
- *sato* is extremely sensitive to variable complementation – the mean number of implications changes from 373.3 in the P-class to 5649.4 in the PC class in the 361-variable queen problem.

Figure 8: Asymptotic performance statistics for several SAT algorithms on ’queen’ family equivalence classes.

can identify and fix. And we have provided an environment in which to validate such improvements.

- Our focus in this paper is on experimental methodology and what it reveals about externally measurable behavior of algorithms. An important next step is the analysis of the mechanisms in these algorithms. All are based on the DPLL algorithm and differ only in three main aspects: their heuristics for

  1. choosing branching variables,
  2. choosing which branch to explore first, and
  3. backtracking after failure of the current branch.

  Evaluating the importance of each of these heuristics in isolation would yield major insights for the next generation of satisfiability algorithms and we are now in a position to do so.

- Finally, there is much work to be done
  - developing new families of instances,
  - exploring new modalities for creating equivalence classes, and
  - improving the usability of our experimental design scaffolding.

We hope to use our insights to develop better satisfiability algorithms, to improve overall understanding of what makes the problem 'hard', and to identify new 'easy' subclasses of cnf formulas and efficient algorithms to solve them.

Note. Some of the experiments, introduced in this report, are still in progress, notably on a complementary version of *sato* and additional formula classes. An updated version of this report will be published before the end of December 2001. For a preview of web-based organization of data sets and the software environment used in these experiments, see `http://www.cbl.ncsu.edu/OpenExperiments/SAT`.

7. REFERENCES

[1] SATLIB - The Satisfiability Library. For more information, see `http://www.satlib.org`.

[2] SAT Live! Up-to-date links for the SATisfiability problem. For more information, see `http://www.satlive.org`.

[3] Sat-Ex: The experimentation web site around the satisfiability . For more information, see `http://www.lri.fr/ simon/satex/satex.php3`.

[4] Ian P. Gent and Toby Walsh. The search for satisfaction. Available at `http://dream.dai.ed.ac.uk/group/tw/sat/-sat-survey2.ps`.

[5] J. Gu, P. Purdom, J. Franco, and B. Wah. Algorithms for the satisfiability (SAT) problem: A survey. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–152, 1997. Available at `http://dream.dai.ed.ac.uk/group/tw/sat/-sat-survey.ps`.

[6] Bart Selman, David G. Mitchell, and Hector J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81(1-2):17–29, 1996.

[7] S. Cook and D. Mitchell. Finding hard instances of the satisfiability problem: A survey, 1997. Available at `http://dream.dai.ed.ac.uk/group/tw/sat/-sat-survey3.ps`.

[8] Michael A. Trick. Second dimacs challenge test problems. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 26:653–657, 1993. The SAT benchmark sets are available at `ftp://dimacs.rutgers.edu/pub/challenge/-satisfiability`.

[9] Henry Kautz, David McAllester, and Bart Selman. Encoding plans in propositional logic. *KR'96: Principles of Knowledge Representation and Reasoning*, pages 374–384, 1996. The SATPLAN benchmark set is available from `http://sat.inesc.pt/benchmarks/cnf/satplan/`.

[10] Cristian Coarfa, Demetrios D. Demopoulos, Alfonso San Miguel Aguirre, Devika Subramanian, and Moshe Y. Vardi. Random 3-SAT: The plot thickens. In *Principles and Practice of Constraint Programming*, pages 143–159, 2000.

[11] Joao P. Marques-Silva. On selecting problem instances for evaluating satisfiability algorithms. *ECAI Workshop on Empirical Methods in Artificial Intelligence (ECAI-EMAI)*, 2000.

[12] David Mitchell. A remark on benchmarks and analysis. In *IJCAI-99 Workshop on Empirical AI*, 1999.

[13] J.N. Hooker. Needed: An empirical science of algorithms. *Operations Research*, pages 42(2):201–212, 1994.

[14] J. Hooker. Testing heuristics: We have it all wrong. *Journal of Heuristics*, pages 1:33–42, 1996.

[15] C. C. McGeoch. Experimental Analysis of Algorithms. In P. Pardalos and E. Romeijn, editor, *Handbook of Global Optimization, Volume 2: Heuristic Approaches*. Kluwer Academic Publishers, 2001.

[16] F. Brglez. Design of Experiments to Evaluate CAD Algorithms: Which Improvements Are Due to Improved Heuristic and Which Are Merely Due to Chance? Technical Report 1998-TR@CBL-04-Brglez, CBL, CS Dept., NCSU, Box 7550, Raleigh, NC 27695, April 1998. Also available at `http://www.cbl.ncsu.edu/publications/-#1998-TR@CBL-04-Brglez`.

[17] J. E. Harlow and F. Brglez. Design of Experiments and Evaluation of BDD Ordering Heuristics. *International Journal on Software Tools for Technology Transfer (STTT): Special Issue on BDDs*, 2001. To appear. A preprint also available under `http://www.cbl.ncsu.edu/publications/` and `http://link.springer.de/link/service/-journals/10009/first/papers/s100090100052.pdf`.

[18] D. Ghosh. *Generation of Tightly Controlled Equivalence Classes for Experimental Design of Heuristics for Graph–Based NP-hard Problems*. PhD thesis, Electrical and Computer Engineering, North Carolina State University, Raleigh, N.C., May 2000. Also available at `http://www.cbl.ncsu.edu/-publications/#2000-Thesis-PhD-Ghosh`.

[19] M. Stallmann, F. Brglez, and D. Ghosh. Heuristics, Experimental Subjects, and Treatment Evaluation in Bigraph Crossing Minimization. *Journal on*

Experimental Algorithmics, 2001. To appear. Also available at http://www.cbl.ncsu.edu/-publications/#2001-JEA-Stallmann.

[20] S.W. Golomb. On the classification of boolean functions. *IRE Trans. Inf. Theory*, pages IT–5:176–186, 1959.

[21] F. Brglez and J.E. Harlow. On Classification of Boolean Functions and Experimental Design. Technical Report 2001-TR@CBL-03-Brglez, CBL, Computer Science Dept., NCSU, Raleigh, NC 27695, 2001. (In preparation).

[22] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *IEEE/ACM Design Automation Conference (DAC)*, 2001. Version 1.0 of Chaff is available from http://www.ee.princeton.edu/ chaff/zchaff/-zchaff.2001.2.17.src.tar.gz.

[23] Jesse Whittemore, Joonyoung Kim, and Karem Sakallah. SATIRE: a new incremental satisfiability engine. In *IEEE/ACM Design Automation Conference (DAC)*, 2001. Version 1.0.0 of Satire is available from http://andante.eecs.umich.edu/satire/-Satire.tgz.

[24] Hantao Zhang. SATO: An efficient propositional prover. In *Conference on Automated Deduction*, pages 272–275, 1997. Version 3.2 of SATO is available from ftp://cs.uiowa.edu/pub/hzhang/sato/sato.tar.gz.

[25] S. Davis and M. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7(3):201–215, 1960.

[26] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[27] Hantao Zhang and Mark E. Stickel. Implementing the Davis-Putnam method. *Kluwer Academic Publisher*, 2000.

[28] Home Page of Tcl Developer Xchange, 2001. See http://tcl.activestate.com.

[29] G. E. P. Box, W. G. Hunter, and J. S. Hunter. *Statistics for experimenters: An Introduction to Design, Data Analysis, and Model Building*. John Wiley & Sons, 1978.

[30] Naresh C. Gupta and Dana S. Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, 56(2-3):223–254, 1992.

[31] Peter Brucker. *Scheduling Algorithms*. Springer Verlag, 2001.

[32] Xiuli Chao and Michael Pinedo. *Operations Scheduling with Applications in Manufacturing and Services*. McGraw Hill, 1998.

[33] Michael R. Garey and David S. Johnson. *Computers and Intractability*. Freeman, 1979.

[34] J. Valdes, R.E. Tarjan, and E.L. Lawler. The recognition of series parallel digraphs. *SIAM Journal on Computing*, 11:298 – 313, 1982.

Table 2: Comparison between chaff and sato on hole instances.

P class data (32 unsatisfiable instances)

class	chaff execution time				sato execution time			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
`hole06`	1.0e-02*	9.8e-02	1.1e-01	1.4e-01	3.0e-02	3.1e-02	3.5e-02	4.0e-02
`hole07`	4.3e-01*	1.2e+00	1.5e+00	2.6e+00	1.5e-01	1.6e-01	1.6e-01	1.6e-01
`hole08`	1.3e+00*	1.4e+01	1.7e+01	2.3e+01	5.8e+00	6.2e+00	6.3e+00	7.0e+00†
`hole09`	7.5e+00*	1.7e+02	2.1e+02	3.5e+02	9.1e+00*	9.8e+00	1.0e+01	1.0e+01
`hole10`	4.8e+01*	4.5e+02	5.1e+02	7.0e+02	9.9e+01*	1.1e+02	1.1e+02	1.1e+02

class	chaff implications				sato implications			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
`hole06`	2331*	9993	10580	11850	4815	5277	5472	5920†
`hole07`	28,708*	52,250	59,590	82,020	54,424	58,690	60,080	67,164†
`hole08`	59,504*	239,500	269,000	319,345	453,655	473,800	482,800	536,738†
`hole09`	160,729*	1,011,000	1,136,000	1,633,656	3,323,446*	3,813,000	3,975,000	4,346,236
`hole10`	458,227*	1,905,000	2,036,000	2,321,743	34,571,367*	40,350,000	41,630,000	45,496,910

PC class data (32 unsatisfiable instances)

class	chaff execution time				sato execution time			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
`hole06`	1.0e-02*	9.9e-02	1.1e-01	1.4e-01	3.0e-02	4.2e-02	4.8e-02	6.0e-02
`hole07`	4.3e-01*	1.2e+00	1.5e+00	2.0e+00	2.2e-01	2.8e-01	3.0e-01	3.4e-01
`hole08`	1.3e+00*	1.5e+01	1.8e+01	2.6e+01	1.5e+00	1.8e+00	2.0e+00	7.0e+00†
`hole09`	7.5e+00*	1.6e+02	1.9e+02	2.7e+02	9.1e+00*	1.7e+01	1.8e+01	2.2e+01
`hole10`	4.8e+01*	4.2e+02	4.8e+02	7.4e+02	9.9e+01*	1.8e+02	1.9e+02	2.1e+02

class	chaff implications				sato implications			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
`hole06`	2331*	9974	10760	12576	3663	5757	6542	7928
`hole07`	28,708*	52,440	58,900	70,605	40,547	47,970	52,880	67,164†
`hole08`	59,504*	247,500	275,100	344,997	323,497	416,400	476,200	576,465
`hole09`	160,729*	986,200	1,095,000	1,388,678	3,312,732	4,333,000	4,919,000	5,614,894
`hole10`	458,227*	1,835,000	1,961,000	2,253,460	34,157,700	46,750,000	52,860,000	59,355,590

*Minimum was achieved by the reference formula rather than by a randomly selected class member.
†Maximum was achieved by the reference formula rather than by a randomly selected class member.

Table 3: Comparison between chaff and sato on queen instances.

P class data (32 satisfiable instances)

class	chaff execution time				sato execution time			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
queen09	0.0e+00	3.5e-03	7.2e-03	1.0e-02	0.0e+00	2.4e-05	2.5e-03	1.0e-02
queen10	0.0e+00	1.6e-02	2.2e-02	4.0e-02	0.0e+00	4.3e-04	3.3e-03	1.0e-02
queen14	1.8e-01	3.6e-01	5.7e-01	1.4e+00†	0.0e+00	6.7e-03	9.6e-03	1.0e-02
queen16	4.5e-01	9.6e-01	1.3e+00	5.2e+00†	1.0e-02	1.2e-02	1.5e-02	2.0e-02
queen19	3.1e+01	7.9e+01	1.1e+02	1.8e+02	1.0e-02	2.4e-02	2.8e-02	3.0e-02

class	chaff implications				sato implications			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
queen09	562	968	1223	1995	56*	79	111	201
queen10	1363	2752	3583	6323	70	109	169	368
queen14	19,046	32,070	43,360	87,291†	139	178	271	777
queen16	34,745	59,910	74,210	187,414†	180	234	357	919
queen19	513,386	931,700	1,186,000	1,761,087	254	317	428	836

PC class data (32 satisfiable instances)

class	chaff execution time				sato execution time			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
queen09	0.0e+00	3.9e-03	8.0e-03	2.0e-02	0.0e+00	4.3e-04	3.3e-03	1.0e-02
queen10	1.0e-02	1.6e-02	2.5e-02	5.0e-02	0.0e+00	1.7e-03	5.2e-03	1.0e-02
queen14	1.8e-01	3.8e-01	6.1e-01	1.8e+00	0.0e+00	1.2e-02	2.8e-02	8.0e-02
queen16	5.2e-01	9.3e-01	1.3e+00	5.2e+00†	1.0e-02	3.2e-02	9.8e-02	2.9e-01
queen19	3.8e+01	9.4e+01	1.3e+02	1.9e+02	2.0e-02	1.8e-02	2.0e-01	1.4e+00

class	chaff implications				sato implications			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
queen09	640	1104	1386	2271	56*	109	153	324
queen10	1353	2797	3813	6542	72*	190	267	571
queen14	20,604	33,750	45,000	91,742	142*	758	2182	6910
queen16	42,332	59,880	73,200	187,414†	180*	1,847	7,522	24,065
queen19	620,688	1,034,000	1,271,000	1,776,732	260*	—‡	—‡	98,911

*Minimum was achieved by the reference formula rather than by a randomly selected class member.
†Maximum was achieved by the reference formula rather than by a randomly selected class member.
‡Variance was too large for meaningful confidence-interval calculation.

Table 4: Comparison between chaff and sato on blocks-world instances.

P class data (32 satisfiable instances)

class P	chaff execution time				sato execution time			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
bw_large_a_s	0.0e+00	7.0e-03	1.1e-02	2.0e-02	1.0e-02	1.2e-02	1.6e-02	2.0e-02
bw_large_b_s	4.0e-02	7.3e-02	9.2e-02	1.4e-01	5.6e-01	8.5e-01	1.2e+00	2.0e+00
bw_large_c_s	1.7e-01	3.3e+00	4.8e+00	1.0e+01	8.0e+00	2.4e+01	4.5e+01	9.0e+01

class	chaff implications				sato implications			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
bw_large_a_s	1026	1996	2624	4032	447	470	509	579
bw_large_b_s	9,835	18,070	23,140	36,536	31,648	50,740	71,650	117,582
bw_large_c_s	29,820	656,600	922,400	1,798,201	241,045	695,000	1,284,000	2,507,629

PC class data (32 satisfiable instances)

class	chaff execution time				sato execution time			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
bw_large_a_s	0.0e+00	7.8e-03	1.1e-02	2.0e-02	1.5e-02	2.3e-02	2.8e-02	4.0e-02
bw_large_b_s	3.0e-02	7.1e-02	9.5e-02	1.6e-01	9.0e-02	7.3e-01	1.2e+00	2.8e+00
bw_large_c_s	3.9e-01	3.4e+00	5.0e+00	8.7e+00	9.0e+00*	1.5e+03	1.9e+03	2.7e+03

class	chaff implications				sato implications			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
bw_large_a_s	1103	1961	2499	3898	447*	1717	2212	3677
bw_large_b_s	5,420	17,840	24,330	41,050	2,528	36,080	65,150	146,742
bw_large_c_s	73,663	618,500	895,200	1,613,697	745,247*	37,670,000	47,900,000	69,891,960

P class data (32 unsatisfiable instances)

class	chaff execution time				sato execution time			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
bw_large_a_u	0.0e+00	1.7e-03	5.2e-03	1.0e-02	1.0e-02	1.0e-02	1.0e-02	1.0e-02
bw_large_b_u	3.0e-02	4.7e-02	5.9e-02	9.0e-02	3.6e-01	4.5e-01	5.2e-01	7.5e-01
bw_large_c_u	1.7e+00	2.7e+00	3.2e+00	4.8e+00	6.3e+00*	6.2e+02	7.0e+02	9.3e+02

class	chaff implications				sato implications			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
bw_large_a_u	686	971	1205	1937	736	914	1003	1168
bw_large_b_u	6,835	10,950	13,630	22,479	17,429	22,370	26,220	38,263
bw_large_c_u	299,245	470,900	564,300	827,043	526,369*	15,640,000	17,980,000	23,825,580

PC class data (32 unsatisfiable instances)

class	chaff execution time				sato execution time			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
bw_large_a_u	0.0e+00	0.0e+00	0.0e+00	0.0e+00	0.0e+00	1.1e-02	1.5e-02	2.0e-02
bw_large_b_u	3.0e-02	4.7e-02	5.5e-02	7.0e-02	3.4e-01	4.9e-01	6.0e-01	8.3e-01
bw_large_c_u	1.6e+00	2.6e+00	3.2e+00	5.3e+00	6.3e+00*	5.8e+02	6.4e+02	8.7e+02

class	chaff implications				sato implications			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
bw_large_a_u	727	900	1074	1604	541	890	1002	1190
bw_large_b_u	6,505	10,960	12,770	17,005	16,775	23,600	28,590	38,370
bw_large_c_u	325,518	487,000	585,000	891,037	526,369*	15,030,000	16,600,000	20,967,660

*Minimum was achieved by the reference formula rather than by a randomly selected class member.

Table 5: Comparison between chaff and sato on scheduling instances.

P class data (32 satisfiable instances)

class	chaff execution time				sato execution time			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
sched03s	0.0e+00	3.8e-03	7.5e-03	1.0e-02	0.0e+00	0.0e+00	2.0e-03	1.0e-02
sched04s	1.0e-02	2.0e-02	2.6e-02	4.0e-02	0.0e+00	4.3e-04	3.3e-03	1.0e-02
sched05s	1.0e-01	1.5e+00	9.7e+00	5.7e+01	0.0e+00	8.6e-03	1.1e-02	2.0e-02
sched06s	1.2e+00	9.5e+00	2.1e+01	5.9e+01	3.0e-02	4.1e-02	4.4e-02	5.0e-02
sched07s	7.4e-01	1.1e+01	2.0e+01	5.9e+01	1.0e-01	1.2e-01	1.3e-01	1.3e-01

class	chaff implications				sato implications			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
sched03s	1206*	1568	1846	2706	90	90	90	90
sched04s	4194	6636	8018	12321	135	135	135	135
sched05s	23,177	208,800	439,000	1,454,082	405	405	405	405
sched06s	221,343	611,300	959,200	2,098,808	819	819	819	819
sched07s	154,277	887,700	1,268,000	2,627,722	1377	1377	1377	1377

PC class data (32 satisfiable instances)

class	chaff execution time				sato execution time			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
sched03s	0.0e+00	2.2e-03	5.9e-03	1.0e-02	0.0e+00	0.0e+00	2.0e-03	1.0e-02
sched04s	1.0e-02	2.2e-02	2.7e-02	4.0e-02	0.0e+00	2.4e-05	2.5e-03	1.0e-02
sched05s	8.0e-02	1.9e+00	9.3e+00	4.0e+01	0.0e+00	5.2e-03	8.6e-03	1.0e-02
sched06s	2.8e-01	9.5e+00	2.3e+01	6.9e+01	2.0e-02	2.4e-02	2.9e-02	4.0e-02
sched07s	3.5e+00	9.6e+00	2.1e+01	6.9e+01	6.0e-02	6.8e-02	7.8e-02	1.1e-01

class	chaff implications				sato implications			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
sched03s	1206*	1490	1714	2468	90	91	92	93
sched04s	4408	6657	7950	10808	135	137	137	138
sched05s	24,429	220,900	447,000	1,299,721	405	408	409	410
sched06s	91,729	591,000	933,900	1,831,905	819*	824	825	826
sched07s	410,889	819,300	1,144,000	2,178,409	1377*	1382	1384	1384

P class data (32 unsatisfiable instances)

class	chaff execution time				sato execution time			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
sched03u	0.0e+00	2.2e-03	5.9e-03	1.0e-02	0.0e+00	0.0e+00	1.5e-03	1.0e-02
sched04u	1.0e-02	1.9e-02	2.6e-02	4.0e-02	0.0e+00	2.0e-03	5.5e-03	1.0e-02
sched05u	1.1e-01	2.4e+00	5.8e+00	2.0e+01	1.0e-02	2.0e-02	2.4e-02	3.0e-02
sched06u	1.6e+00	1.2e+01	2.4e+01	6.2e+01	9.0e-02	9.9e-02	1.0e-01	1.1e-01
sched07u	9.6e-01	1.1e+01	2.2e+01	5.8e+01	2.6e-01	2.9e-01	2.9e-01	3.2e-01

class	chaff implications				sato implications			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
sched03u	1117*	1618	1977	3037	327	349	354	362
sched04u	3946	6244	7726	10921	529	548	554	564
sched05u	33,399	247,600	408,400	849,282	2350	2378	2390	2409
sched06u	297,822	704,400	1,051,000	1,827,430	6201	6231	6245	6277
sched07u	206,388	812,900	1,263,000	3,013,723	10,819	10,900	10,910	10,935

PC class data (32 unsatisfiable instances)

class	chaff execution time				sato execution time			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
sched03u	0.0e+00	2.0e-03	5.5e-03	1.0e-02	0.0e+00	4.3e-04	3.3e-03	1.0e-02
sched04u	1.0e-02	2.2e-02	2.8e-02	4.0e-02	0.0e+00	8.9e-04	4.1e-03	1.0e-02
sched05u	1.3e-01	1.2e+00	8.8e+00	5.3e+01	2.0e-02	2.1e-02	2.4e-02	3.0e-02
sched06u	9.7e-01	1.0e+01	2.1e+01	6.1e+01	1.0e-01	1.0e-01	1.0e-01	1.1e-01
sched07u	5.8e-01	1.1e+01	2.1e+01	4.8e+01	2.8e-01	2.9e-01	2.9e-01	3.0e-01

class	chaff implications				sato implications			
	min	mean LB	mean UB	max	min	mean LB	mean UB	max
sched03u	1117*	1512	1789	2490	341	350	355	364
sched04u	4316	6811	8404	13709	539	552	558	573
sched05u	37,182	195,100	427,400	1,485,493	2349	2375	2389	2413
sched06u	244,819	649,800	918,200	1,574,753	6176	6232	6248	6276
sched07u	140,533	863,500	1,262,000	2,471,675	10,846	10,900	10,920	10,965

*Minimum was achieved by the reference formula rather than by a randomly selected class member.