SAT 2002 – Fifth Int. Symp. on the Theory and Applications of Satisfiability Testing, May 6–9, 2002, Cincinnati, Ohio, USA

The Role of a Skeptic Agent in Testing and Benchmarking of SAT Algorithms *

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

This paper introduces a persistent agent called *skeptic* who supplies instances from well-defined equivalence classes to test and benchmark SAT solvers. On such classes, metrics such as max/min ratio of time-to-solve should approach the value of 1.0. Experiments suggested by the skeptic on the instances of the *same* class show (1) the time-to-solve max/min ratios for a given solver can exhibit a range from 2 to 1000 and beyond, and (2) max/min ratios for another solver may be several orders of magnitude better, including a significantly lower time-to-solve average value. Both of these factors point out that (1) SAT solvers can not only be much improved but also more reliably tested for any such improvement, and (2) the intrinsic complexity or 'hardness' of SAT instances cannot be gauged reliably with the current generation of SAT solvers.

The role of the skeptic is *not* to declare any particular SAT solver as the 'best'. Rather, the main goal of this paper is to have skeptic force imperfectly designed solvers to exhibit erratic statistical behavior on input classes that should induce uniform behavior. Ultimately, the role of the skeptic is to be subsumed in a new methodology of experimental design and performance evaluation of combinatorial 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 and problem instances are also readily available on the Web, e.g. [4, 5, 6].

New SAT solvers continue to be introduced, most recently at DAC'2001 [7, 8, 9]. Two of these state-of-the-art solvers, *chaff* [7] and *satire* [8], are publicly accessible and have been installed within a benchmarking environment to be described later in the paper. Some of the earlier SAT solvers are similarly available; we have added the frequently-cited *sato* [10] and an unbiased textbook-level SAT-solver *dp_nat*, implemented as per pseudo-code in [11], which itself is based on the well-known DPLL algorithm [12, 13].

We perform a number of experiments with these solvers. The motivation for these experiments is *not* to declare any particular SAT solver as the 'best'. Rather, these experiments are driven by an agent called *skeptic* who, in experimental evaluation of algorithms, plays a role analogous to that of an adversary in the worst-case complexity analysis. An adversary devises an input that forces an algorithm to perform badly enough to prove a lower bound (see, e.g., [14]). A skeptic, as the name suggests, is neither as malicious nor as rigorous as an adversary. The skeptic's purpose is to force imperfectly designed algorithms to exhibit poor statistical behavior (large confidence intervals) on input classes that should induce uniform behavior. Ultimately, the role of the skeptic is to be subsumed in a new methodology of experimental design and performance evaluation of combinatorial algorithms. The case for such methods has already been succinctly articulated in [15, 16] as well as demonstrated experimentally in our earlier work [17, 18, 19, 20].

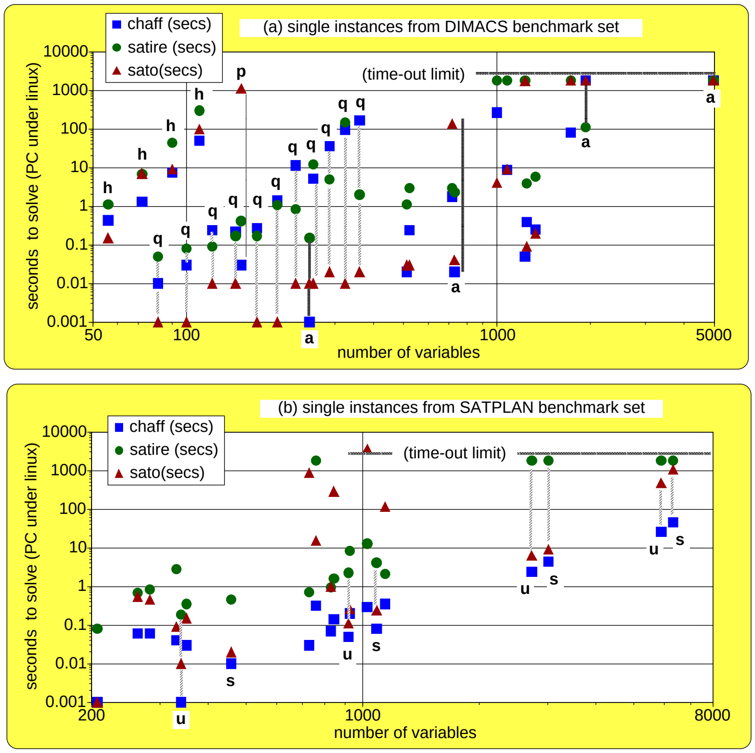
We first contrast the proposed approach with the up-to-date benchmarking experiments with SAT solvers that are being reported on the Web [3], the figure of merit being 'time-to-solve' on a particular PC. A sample of such a posting is reproduced in the two left-most columns in Figure 1. Three additional columns show 'time-to-solve' on our PC. Given the tabulated results, it is tempting to jump to conclusions such as (1) our PC is slower (relatively unimportant), (2) *chaff* significantly outperforms both *sato* and *satire* on the class of `hole` formulas [21] (very important – if true as an average).

The skeptic in this paper views results in Table 1 only as a starting point – no conclusions are to be drawn until a sufficient number of experiments with instances from the

Table 1: Traditional benchmarking formats.

reference	web-based report [3]		local host report		
	time-to-solve(secs)		time-to-solve(secs)		
benchmark	*chaff*	*sato*	*chaff*	*sato*	*satire*
hole6	0.01	0.04	0.01	0.04	0.20
hole7	0.32	0.11	0.42	0.16	1.12
hole8	0.95	5.40	1.25	6.76	6.31
hole9	5.49	6.44	7.23	9.07	44.40
hole10	36.04	69.65	47.10	98.80	288.0
sum-total	42.81	81.64	56.01	114.83	340.03

*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 [21] or the SATPLAN set [22]. In addition, the tabulated results are averaged to argue merits of each algorithm.

This paper marks a significant departure from the traditional approach. We engage a skeptic agent to devise equivalence classes for each benchmark of interest, and repeat the experiments for up to 32 such instances. An observer can thus 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 Table 1 reports 47.1, 98.8, and 288.0 seconds for *chaff*, *sato*, and *satire*. In Table 2, skeptic reports the averages of 451 seconds for *chaff*, 182 seconds for *sato*, while *satire* times out at 1800 seconds. In addition, the max/min values for *chaff* and *sato* are 736./47.1=15.6 and 206./98.8=2.09, respectively.

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

equivalence class of `hole`-like formulas have been repeated and analyzed. The simple sum-totals as shown presently have no statistical significance, since values reported for the same solver are single, unrelated measurements.

The primary role of the skeptic is to enforce statistical significance and repeatability. Each *reference* problem instance, such as `hole8` in Table 1, is used by the skeptic to generate one or more equivalence classes of closely related instances, such as the isomorphism classes introduced later in the paper. Within each isomorphism class, instances share a common *structure* and *solution landscape*, so that the level of variability of solver performance wrt to all instances from such equivalence class signifies the level of erratic behavior of the solver on such a class.

We expect to see wide variability in execution time for the same solver across SAT instances having the same size (assuming $P \neq NP$). It would be natural to attribute this variability to fundamental characteristics that make some instances inherently easy and others inherently hard, if not in general then at least for the algorithm in question. The results obtained by our skeptic-based approach suggest otherwise: SAT solvers exhibit wide variability even within narrowly-defined classes based on a single instance. We have encountered this phenomenon in problem domains other than SAT: logic optimization, BDD variable ordering, partitioning, routing and placement, and crossing number in bigraphs [17, 18, 19, 20]. For the SAT problems, formulated as cnf formulas, we introduce four isomorphism equivalence classes, each derived by applying well-defined variable permutation and complementation rules. These rules were originally introduced to define equivalence classes of Boolean functions [23].

In the remainder of the paper we illustrate how a simple skeptic challenges up to five SAT solvers. Our observations and the expectation of more elaborate skeptics in the future raise questions such as: (a) How good are the state-of-the-art solvers? and (b) What really distinguishes a hard instance from an easy one? Significant opportunities are created in SAT and other problem domains for (a) improving solvers, and (b) establishing more rigorous benchmarks.

The paper is organized into five sections as follows: (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

Traditionally, the performance of SAT solvers has been evaluated experimentally either in terms of randomly generated instances of SAT problems, e.g. [24, 25], or structured instances, such as the instances from the DIMACS set [21] or the SATPLAN set [22]. Merits of either approach are subject to on-going critique and examination [15], [16], [26], [27], [28].

The traditional way to report results of SAT solvers is the *time-to-solve* performance of single instances of a *reference formula* in conjunctive normal form (cnf). Table 1 represents the traditional organization of such an experimental report. Results of more comprehensive experiments, repeated on a subset of well-known benchmark formulas are shown in the two graphs in Figure 1. Rather than reporting time-to-solve results of experiments in the traditional tabular format, we depict them in two graphs. This representation, strictly for the convenience of visualizing the presence of asymptotic trends, groups 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

Table 2: An example of a report on experimental results (in a format introduced by the skeptic).

costID = timeToSolve (seconds)

Class labels: *name*=hole08, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
sato	6.76	1.43	1.92	2.60	4.72
chaff	1.25	9.49	16.2	26.3	21.1
satire	6.31	411	1310	t'out	> 285

Class labels: *name*=hole10, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
sato	98.8	158	181	206	2.09
chaff	47.1	320	451	736	15.6
satire	288	t'out	t'out	t'out	> 6.25

costID = numberOfImplications

Class labels: *name*=hole08, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
satire	2.80e4	1.57e5	2.45e5	t'out	> 11.3
chaff	5.95e4	1.79e5	2.61e5	3.45e5	5.80
sato	5.37e5	3.23e5	4.46e5	5.76e5	1.78

Class labels: *name*=hole10, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
chaff	4.58e5	1.60e6	1.89e6	2.25e6	1.41
sato	3.46e7	3.42e7	4.98e7	5.94e7	1.74
satire	1.84e5	t'out	t'out	t'out	>> 1

NOTES:
(1) For each *class* and for each *costID*, the solverID ordering is induced by sorting on *meanV*.
(2) The value of *initV* is included in computation of *max/min* ratio.
(3) The value of *t'out* (timeout) in these experiments has been set to 1800 seconds.
(4) A pointer to the original data and additional statistics, such as median, standard deviation,
confidence intervals of the mean, etc. is available under http://www.cbl.ncsu.edu/OpenExperiments/SAT/

with 'u'), and *bw_large_s* (sat, marked with 's').[1] All programs *chaff, satire, sato* 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. An inspection of graphs in Figure 1 suggests the following:

- For the `hole` family, *satire* appears out-ranked by the other two solvers; there is a cross-over of *chaff* and *sato*.
- For the `queen` family, *sato* appears to significantly outrank the other two solvers; there are cross-overs of *chaff* and *satire*.
- For the `hanoi` family (marked with 'a'), *satire* is the only solver that does not time out on the instance of `hanoi5` (1931 variables), while all three solvers time out on the instance of `hanoi6` (4968 variables).
- For the `bw_large_u` family, *chaff* appears to out-rank the other two solvers, and *satire* times out for the last two instances (2729 and 5886 variables).

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 for each formula in the respective families, and the confidence interval associated with each average. 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.

Consider now the experimental summary in Table 2 as organized by the skeptic, after repeating 32 additional experiments, one for each equivalence class instance. We report solutions for class 'PC' in terms of two cost functions (costIDs): time-to-solve and number-of-implications. As we show later, both of these costs are highly correlated for all solvers we tested. For each cost, we report results in five columns: *initV*, *minV*, *meanV*, *maxV*, *max/min*. The column *initV*, represents the value reported by the solver for the initial (reference) instance from which all others are to be derived; all other columns report statistics for the equivalence class. Results in the four rightmost columns, unlike conclusions based on the initV's only, *are* significant as they provide a number of platform-independent conclusions (for this class of problems):

- *sato* definitely has fastest implementation of the implication engine (most implications per unit time).
- *sato* definitely requires the most implications to reach a (non-SAT) decision, hence when the problem size (in this family) increases, there will be a cross-over in the timing performance when compared to *chaff*.
- *satire* requires the least number of implications to reach a (non-SAT) decision, however the slow implementation of its implication engine limits the experiments to instances with no more than 72 variables (here the class of `hole8` in the `hole` family).

These illustrative experiments 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 every time we let the skeptic agent design the experiments with instances from the same equivalence class.

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 five SAT solvers 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 crates of oranges. *Before* initiating a comprehensive series of experiments, we shall separate oranges into crates in accordance with a well-defined classification schema, e.g. such as the one shown in Figure 2. 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). All crates are stored under identical climatic conditions, and all oranges are tested for freshness at periodic

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

```
orangeFamily
  - diameter_A
     - skinType_a
        orange0
        orange1
        orange2
         ...
         ...
        orange32
     + skinType_b
     + skinType_c
  + diameter_B
  + diameter_C
  + diameter_D
```

An experimental design using oranges may consider the classification schema 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 2: An orange family classification schema and the process of creating experimental subjects.

intervals. Statistical methods and tools are used to evaluate the effectiveness of each treatment. The significance of treatment0 cannot be overstated; we'll define its counterpart when designing experiments with SAT solvers.

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

A CNF Formula and Its Equivalence Classes. The hardness of a SAT problem in a cnf formula is encoded in the structure of the cnf representation and the total number of satisfying assignments for the function the formula represents. Both may require exponential time and space to characterize exactly. However, it is simple to create an *isomorphism* class of cnf formulas, all of the same hardness as the given *reference formula*.

Our approach is based on the notion of equivalence classes of Boolean functions $\{F_j\}$ as defined in [23] and illustrated in Figure 4(a). Each cnf formula F_j is consists of a set 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. While the problem of deciding whether a pair of functions F_{j1} and F_{j2} belongs to $\{F_j\}$ is NP-hard, the problem of generating any number of functions in $\{F_j\}$ is simple. Here, we designate F_j as the *reference formula* and *re-write* it in accordance with well-defined transformations (variable permutation and complementation), giving rise to four distinct equivalence class types formulated in Figure 3: **I**, **P C**, and **PC**.

An Example. A specific example of the 6-variable, 12-clause formula in Figure 4(b) illustrates the simple relationship between the four solution vectors of the reference formula and the solution vectors of each formula in the respective equivalence class.

The classes we define are based on isomorphisms in the following sense. For any formula F having n variables, let $H(F)$ be the n-dimensional hypercube with nodes corresponding to all possible variable assignments (with the usual graph-theoretic interpretation — two nodes are adjacent if their labels differ in one bit), and with nodes whose labels represent satisfying assignments colored. We say that two formulas F_1 and F_2 are *structurally equivalent* if $H(F_1)$ and $H(F_2)$ have an automorphism [31] that preserves the coloring.

Skeptic's Claim. If F_1 and F_2 are structurally equivalent, then a good SAT solver should take roughly the same amount of time to solve F_1 as F_2 (and the same applies to other measures as well).

- **I**-class (identity) — variable names are preserved. Clauses and literals within a clause are randomly permuted.
  Property: each solution vector is identical to the reference formula solution vector.
- **P**-class (permutation) — variable names are permuted randomly, as are clauses and literals.
  Property: each solution vector is a permutation of the reference formula solution vector.
- **C**-class (complement) — variable names are preserved; variables are complemented randomly. Clauses and literals within a clause are randomly permuted.
  Property: each solution vector is the reference formula solution vector, with the same bits complemented.
- **PC**-class (permutation and complement) — variable names are permuted randomly *and* variables are complemented randomly. Clauses and literals within a clause are randomly permuted.
  Property: each solution vector is a permutation of the reference formula solution vector, with the same bits complemented.

Figure 3: Rewriting rules to create four isomorphism classes of cnf formulas.

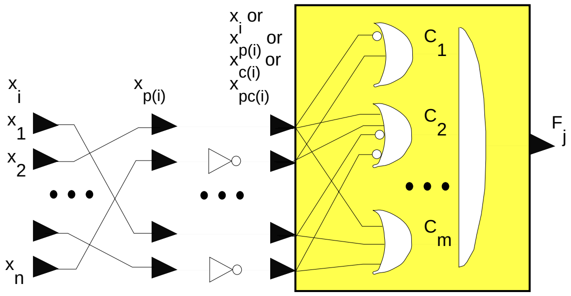
We next describe an experiment with five SAT solvers.

SAT Solvers and Equivalence Class Instances. A total of five SAT solvers, *chaff*, *satire*, *sato*, *satoL* and *dp0_nat*, were applied to 32 instances in each of the four equivalence classes of the reference formula in Figure 4(b).

Each solver is a variation of DPLL algorithm [12, 13] – they differ only in choice of variable ordering and backtracking strategy. The sato solver can be run under two solverIDs: *sato* that relies on a data structure of tries, and *satoL* that relies on a data structure of linked lists. The solver *dp0_nat* orders variables based solely on the input appearance and uses the simplest possible backtracking strategy; it thus corresponds to treatment0 (no treatment) in the analogy of the experiments with oranges. The results of all experiments are summarized in Figure 4(c). It is important to note that, except for the solver *chaff*, these results 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 solver *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 solver *satire*, the near-equal variance for all four classes is characteristic.
- For solver *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 solver *satoL*, 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. As shown here, *satoL* will typically have smaller standard deviation

(a) Equivalence classes: Boolean functions and cnf formulas.



A cnf formula consists of a set 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. As shown on the left, the formula also represents a Boolean function F_j as a 2-level dag with inputs $\{x_1, \ldots, x_n\}$ at level 0, OR-gates $\{C_1, \ldots, C_m\}$ with inverting/non-inverting inputs at level 1, and a single AND gate with output F_j.

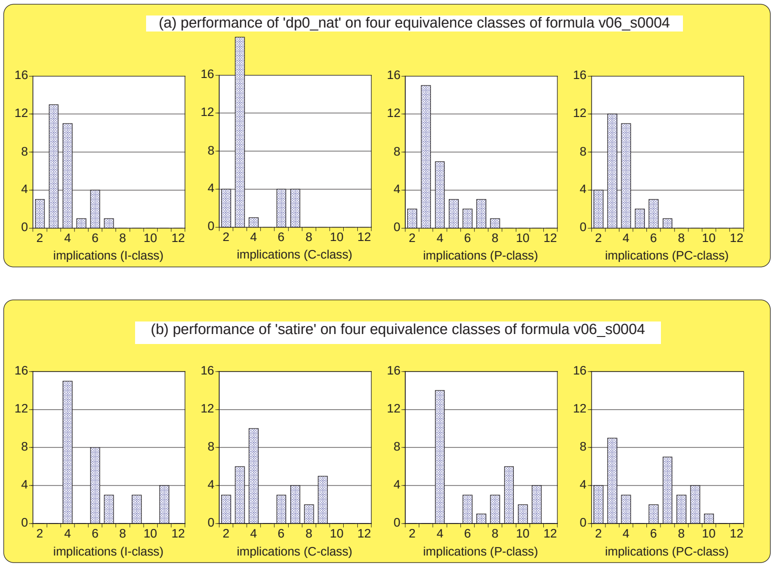
Equivalence classes $\{F_j\}$ of Boolean functions, each of size $n!2^n$, have been analyzed for their invariants in [23]. Here, we illustrate the construction of such classes with the cascaded instances of a *permutation* and a *complementation* network.

(b) Example: instances from four equivalence classes of a 6-variable, 12-clause formula v06_0004.

Four satisfying assignments of each instance are simple transformations of the reference formula assignments.

Reference formula:	{-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}
solution vectors:	(001001, 001010, 010001, 010010)
I-class formula:	{-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}
solution vectors:	(001001, 001010, 010001, 010010)
transformations:	none, solutions *are* reference formula solutions.
P-class formula:	{-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}
solution vectors:	(011000, 110000, 001001, 100001)
transformations:	variable permutation 462513 is applied to reference formula solutions.
C-class formula:	{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}
solution vectors:	(101011, 101000, 110011, 110000)
transformations:	variable complementation of (1, 5) is applied to reference formula solutions.
PC-class formula:	{-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}
solution vectors:	(110101, 011101, 110000, 011000)
transformations:	variable permutation 246531 and complementation of (2, 4) is applied to reference solutions.

(c) Experiment: number-of-implications statistics for equivalence classes of formula v06_0004.



As we demonstrate in the experiment section of this paper, the number of implications is a measure closely correlated with execution time for DPLL-based SAT solvers.

Even on this small illustrative data set, counting the number of implications clearly differentiates between all five solvers: *chaff* [7], *satire* [8], *sato* and *satoL* [10], and *dp0_nat*, our own vanilla implementation of the DPLL algorithm [12, 13].

Here, instances of isomorphic 6-variable, 12-clause cnf formulas can induce large variability in performance: from 2 implications/solution to 11 implications/solution. We discovered, on much larger formulas, comparable and much larger max/min performance ratios, both in time-to-solve and number-of-implications.

equiv.	*chaff*	*satire*	*sato*	*satoL*	*dp0_nat*
class	mean/std	mean/std	mean/std	mean/std	mean/std
I	6.00/0.00	5.97/2.42	3.00/0.00	3.00/0.00	3.81/1.26
C	6.00/0.00	5.06/2.29	7.40/1.34	4.84/2.03	3.78/1.66
P	6.00/0.00	6.69/2.73	3.00/0.00	3.00/0.00	4.06/1.56
PC	6.00/0.00	5.28/2.56	6.56/1.43	4.66/2.07	3.75/1.24

Figure 4: On generation of equivalence classes of cnf formulas – a method and an experiment.

```
benchm_SATcnf
  + bw_large_s
  + bw_large_u
     ...
     ...
  + queen_medium
  - queen_small
    - @references
        queen04_v00016.cnf
        queen04_v00025.cnf
        queen04_v00036.cnf
         ...
         ...
    - queen_04_v00016
       - queen04_v00016_C
            i0000.cnf
            i0001.cnf
            i0002.cnf
             ...
             ...
       + queen04_v00016_I
       + queen04_v00016_P
       + queen04_v00016_PC
    + queen_04_v00025
    + queen_04_v00036
       ...
       ...
  + sched_s
  + sched_u
     ...
     ...
```

```
results_SATcnf
  + chaff
  + dp0_nat
     ...
     ...
  + satire
  - sato
    + bw_large_s
    + bw_large_s
       ...
       ...
    + queen_medium
    + queen_small
        - @references
            sato.raw
        - queen_04_v00016
           - queen04_v00016_C
                sato.raw
           + queen04_v00016_I
           + queen04_v00016_P
           + queen04_v00016_PC
        + queen_04_v00025
        + queen_04_v00036
           ...
           ...
    + sched_s
    + sched_u
       ...
       ...
  + satoL
     ...
```

Two components of the experimental design schema (EDS) specific to SATcnf problems are shown on the left: benchm_SATcnf contains families of cnf reference formulas and corresponding equivalence class formulas; results_SATcnf contains any number of solver-specific directories of results, each organized as per benchm_SATcnf schema, except that in place of formula instance files, solver-reported results are filed in their native 'raw' formats. This organization allows for simple additions of new benchmark families and archival and retrieval of existing and new experimental results, generated by new solvers.

The comprehensive SATcnf experiment can now be invoked with few lines of code in Tcl [29, 30]:

```
# inputs:
set timeOutSeconds "1800"
set benchmList "bw_large_s bw_large_u queen_small"
set solverList "chaff sato satoL"
# outputs (coordinated by encap_SATcnf)
set resultsDir "results_SATcnf"
foreach benchmID $benchmList {
   set filePaths [findFiles $benchmID *.cnf]
   foreach solverID $solverList {
      encap_SATcnf   $timeOutSeconds $solverID \
       $benchmID $filePaths $resultsDir
   }
}
```

For each solverID and each class directory, 'raw' results of experiments such as above are saved in files labeled as 'solverID.raw' in appropriate directories under results_SATcnf. These results are post-processed by utilities for a number of different objectives (see text).

Figure 5: SATcnf experimental design schema (EDS) and the comprehensive SATcnf experiment.

and smaller mean in number-of-implications as well as time-to-solve.

- For solver *dp0_nat*, the near-equal variance for all four classes is characteristic, and contrary to this example, it can exceed the mean and the variance of other solvers.

Since some solvers appear designed to suppress the variability of performance induced by instances from some of the classes, the only class where all solvers can be compared fairly is the **PC**-class. However, all four classes are necessary to completely describe the properties of the solver under test. The presence of bias in the solver can decrease its performance potential – *chaff*, *satire*, *sato*, and *satoL* have been noted to exhibit erratic behavior on some of the larger formula classes.

On Solver Cost Distribution. Since instances from the four equivalence classes in this section effectively represent the same formula, the ideal solver 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. A large variance is an indicator that the *solver is behaving erratically* on few or several instances from the class – and *not* that the choice of the instance is inappropriate or 'hard'!

4. EXPERIMENTAL DESIGN AND SAT

A well-defined schema is required to manage large volumes of input data sets and repeated executions of several solvers, each writing results in solver-specific formats. Two components of the *experimental design schema* (EDS) that evolved in this work are shown in Figure 5: benchm_SATcnf archives all input data sets, results_SATcnf archives results of every experiment generated by each solver. The initial results are in solver-specific formats and are readily accessible for any number of post-processing steps by various utilities that perform tasks such as verification of reported solutions and tabulation of results in a uniform format that can be read by most spreadsheet and charting programs, displayed on the Web, etc.

Input Data Sets. The structure of benchm_SATcnf is similar to the classification schema illustrated for oranges in Figure 2. We group reference formulas of structurally-related instances of increasing size into *families* so we can study the asymptotic performance of SAT algorithms. 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. At least three or more of such reference formulas should define a family. However, as we note in Figure 6, there can also be exceptions to this rule. For example, the family 'aTutorialCNF' has a tutorial purpose only, and the families under 'random3sat' represent different instances of the same size, and as such cannot be used to study solver asymptotic behavior. See next section for detailed description of archives in Figure 6 and associated experiments.

For each reference formula, e.g. queen_04v00016.cnf, we may generate a number of equivalence classes such as queen_04v00016_C, with instances *i0000.cnf*, *i0001.cnf*, *i0002.cnf*, . . . , etc. By convention, we maintain the instance *i0000.cnf* strictly as a copy of the reference formula, e.g. queen_04v00016.cnf. We can thus readily access the value of *initV* (under results_SATcnf) that we report in the first column of tabulated results, such as shown in Table 2.

Comprehensive SATcnf Experiment. The comprehensive SATcnf experiment can be invoked with a few lines of

As per schema in Figure 5, there are currently 32 to 128 instances of the 'PC' isomorphism (equivalence) class for *each* reference instance listed below. For some formulas, there are up to four equivalence classes, each of size at least 32. A pointer to the compressed archive for each family is available under http://www.cbl.ncsu.edu/OpenExperiments/SAT/.

`aTutorialCNF/` (family directory)

`@references/`	vars	clauses	sat?	notes
`v06_s0004.cnf`	6	12	yes	See paper.
`v10_s0000_sched.cnf`	10	23	no	”
`v10_s0012_ramsey.cnf`	10	25	yes	See [10].

`bw_large_s/` (family directory)

`@references/`	vars	clauses	sat?	notes
`bw_large_a_s.cnf`	459	4,675	yes	See [22].
`bw_large_b_s.cnf`	1,087	13,772	yes	”
`bw_large_c_s.cnf`	3,016	50,457	yes	”
`bw_large_d_s.cnf`	6,325	131,973	yes	”

`bw_large_u/` (family directory)

`@references/`	vars	clauses	sat?	notes
`bw_large_a_u.cnf`	340	3,294	no	See [22].
`bw_large_b_u.cnf`	920	11,491	no	”
`bw_large_c_u.cnf`	2,729	45,368	no	”
`bw_large_d_u.cnf`	5,886	122,412	no	”

`hole_medium/` (family directory)

`@references/`	vars	clauses	sat?	notes
`hole06_v00042.cnf`	42	133	no	See [21].
`hole07_v00056.cnf`	56	204	no	”
`hole08_v00072.cnf`	72	297	no	”
`hole09_v00090.cnf`	91	415	no	”
`hole10_v00110.cnf`	110	561	no	”
`hole14_v00210.cnf`	210	1,685	no	”

`hole_small/` (family directory)

`@references/`	vars	clauses	sat?	notes
`hole02_v00006.cnf`	6	9	no	See [21].
`hole03_v00012.cnf`	12	22	no	”
`hole04_v00020.cnf`	20	45	no	”
`hole05_v00030.cnf`	30	81	no	”

`hanoi/` (family directory)

`@references/`	vars	clauses	sat?	notes
`hanoi03.cnf`	249	1,512	yes	See [21].
`hanoi04.cnf`	718	4,934	yes	”
`hanoi05.cnf`	1,931	14,468	yes	”
`hanoi06.cnf`	4,968	39,666	yes	”

`hanoi_trim/` (family directory)

`@references/`	vars	clauses	sat?	notes
`hanoi03_v00171.cnf`	171	1,068	yes	See paper.
`hanoi04_v00541.cnf`	541	3,912	yes	”
`hanoi05_v01500.cnf`	1,500	12,063	yes	”
`hanoi06_v03864.cnf`	3,864	33,753	yes	”

`queen_medium/` (family directory)

`@references/`	vars	clauses	sat?	notes
`queen09_v00081.cnf`	81	1,065	yes	See [10].
`queen10_v00100.cnf`	100	1,480	yes	”
`queen14_v00196.cnf`	196	4,200	yes	”
`queen16_v00256.cnf`	256	6,336	yes	”
`queen19_v00361.cnf`	361	10,735	yes	”

`queen_small/` (family directory)

`@references/`	vars	clauses	sat?	notes
`queen04_v00016.cnf`	16	80	yes	See [10].
`queen05_v00025.cnf`	25	165	yes	”
`queen06_v00036.cnf`	36	296	yes	”
`queen07_v00049.cnf`	49	483	yes	”
`queen08_v00064.cnf`	64	736	yes	”

`rand3sat_250-1065_s/` (family directory)

`@references/`	vars	clauses	sat?	notes
`uf250-1065_027.cnf`	250	1,065	yes	See [6]
`uf250-1065_034.cnf`	250	1,065	yes	and
`uf250-1065_087.cnf`	250	1,065	yes	this paper.

`rand3sat_250-1065_u/` (family directory)

`@references/`	vars	clauses	sat?	notes
`uuf250-1065_046.cnf`	250	1,065	no	See [6]
`uuf250-1065_074.cnf`	250	1,065	no	and
`uuf250-1065_090.cnf`	250	1,065	no	this paper.

`sched_medium_s/` (family directory)

`@references/`	vars	clauses	sat?	notes
`sched03s_v00095.cnf`	95	495	yes	See paper.
`sched04s_v00140.cnf`	140	892	yes	”
`sched05s_v00412.cnf`	412	4,338	yes	”
`sched06s_v00828.cnf`	828	12,024	yes	”
`sched07s_v01386.cnf`	1,386	25,671	yes	”

`sched_medium_u/` (family directory)

`@references/`	vars	clauses	sat?	notes
`sched03u_v00093.cnf`	93	493	no	See paper.
`sched04u_v00138.cnf`	138	890	no	”
`sched05u_v00410.cnf`	410	4,336	no	”
`sched06u_v00826.cnf`	826	12,022	no	”
`sched07u_v01384.cnf`	1,384	25,669	no	”

`sched_small_s/` (family directory)

`@references/`	vars	clauses	sat?	notes
`sched00s_v00012.cnf`	12	26	yes	See paper.
`sched01s_v00031.cnf`	31	98	yes	”
`sched02s_v00059.cnf`	59	255	yes	”

`sched_small_u/` (family directory)

`@references/`	vars	clauses	sat?	notes
`sched00u_v00010.cnf`	10	23	no	See paper.
`sched01u_v00029.cnf`	29	96	no	”
`sched02u_v00057.cnf`	57	253	no	”

Figure 6: Family directories and reference cnf formulas from which isomorphism classes have been generated.

code in Tcl [29, 30], shown in Figure 5. The essential input variables are *timeOutSeconds, benchmList* and *solverList.* Once file paths to all formula instances (in a given family) have been generated, a solver encapsulation program *encap_SATcnf* is invoked and the designated solver repeatedly executed on all formula instances.

While not shown here, additional input variables expand the features of the SATcnf experiment. For example, experiments may be conducted only on user-specified subsets of class instances under the `benchm_SATcnf` schema. For each solverID and each class directory, 'raw' results of experiments in solver-specific formats are written to files in designated directories under `results_SATcnf`, each one labeled as 'solverID.raw'. These results are post-processed by utilities for a number of objectives we describe next.

Results Archives. In any SATcnf experiment, *benchmList* may contain the (top-level) name of the family directory or any of its subdirectories. However, items in the *solverList* can only reference a specific *solverID.* The schema of `results_SATcnf` represents the home directory of solver-reported results, each under its own *solverID.* Under each *solverID*, results are organized parallel to schema of `benchm_SATcnf` – with exception that in place of formula instance files, solver-reported results are filed in their native 'raw' formats – as 'sato.raw', shown for *solverID=sato* in Figure 5. This organization allows for simple additions of new benchmark families and archival and retrieval of existing and new experimental results, generated by new solvers.

Essential utilities that we routinely invoke to generate equivalence classes, to execute any SAT solver, and to post-process results of experiments include:

- *cnfFormula2Class*, a program that reads a cnf reference file and class designation parameters. The program creates the designated directories and formula class instances in cnf format.
- *encap_SATcnf*, a solver encapsulation program that reads solverID and class-specific formula instances, executes the solver, and returns for each class directory visited, a solver-specific file of results as 'solverID.raw'.
- *solverVer*, a verification program that reads the formula instance, the reported solution in 'solverID.raw', and outputs a verification report as 'solverID.ver'. Here, we find confirmation that the reported solution indeed satisfies the formula.
- *solverPP*, a program that reads a file 'solverID.raw' and returns a standard spreadsheet report of raw data as a file 'solverID.tab', and essential statistics of raw data as a file 'solverID.stat'. For each costID such as time-to-solve, number-of-implications, number-of-backtracks, etc., results listed in the 'stat' file currently include: initial value (associated with the reference formula) median value, mean value, standard deviation, minimum value, maximum value, 95% confidence interval of the mean (based on the t-statistics [32]), the standard coefficient of variation, and the max/min ratio. The computation of max/min ratio always includes not only the minimum and maximum values generated from class instances but also the initial value associated with the reference formula.
- *solverSum*, a program that reads, from a designated location in the `results_SATcnf` schema, any number of statistics summary files with extension 'stat' and outputs a file as a statistical summary generically named as 'solvers.summ'. This file may contain data such as shown in Table 2.

Additional utilities may be added in the future, such as post-processors to write data directly for a commercial statistics packages such as JMP [33] or post-processors to prepare data for multiple-comparison analyses such as discussed in [34].

The experimental design schema for the SATcnf problem described here enabled us to develop a cross-platform environment that

- facilitates the use of the experimental design methodology,
- supports replication of our current and future experimental results,
- may encourage other researchers to participate by contributing reference formulas and solvers,
- automates the experiments in a way that minimizes the potential for human error.

To access compressed archives of input data sets under `benchm_SATcnf`, raw results and statistical summary under `results_SATcnf`, and the cross-platform utilities, reader is invited to follow the links posted on the home page
http://www.cbl.ncsu.edu/OpenExperiments/SAT/

5. REPORTS OF EXPERIMENTS

Our initial SAT experiments replicated the experiments on well-known subsets of benchmark formulas, using three readily available SAT solvers. Results are discussed in Section 2 and summarized in Figure 1. Following suggestions from the skeptic, we devised four equivalence classes for each reference formula, and repeated experiments on formula instances from each class, at least 32 formulas per class. Experiments with up to five solvers are reported in great detail in the illustrative example of a 6-variable, 12-clause reference formula and its class instances in Figure 4. These small-scale experiments provide a basis for the experimental design strategy and experiments described in this section.

For completeness, we list the five solvers in terms of respective solverIDs:

- *chaff*, as described in [7].
- *dp0_nat*, implemented as per pseudo-code in [11], which itself is based on the well-known DPLL algorithm [12, 13]. This solver orders variables solely on the order in which they appear in the input and uses the simplest possible backtracking strategy. It is implemented in Tcl [29, 30] (an interpreted language), so only its number-of-implications as costID may be compared to other implementations.
- *satire*, as described in [8].
- *sato*, as described in [10], implemented with a data structure based on tries.
- *satoL*, as described in [10], implemented with a data structure based on linked lists.

The grouping of reference formulas into families and the organization of equivalence classes has been discussed in Figure 5. The actual formulas and classes we used in the experiments reported in this paper are shown in Figure 6. While families in Figure 6 are listed in the alphabetical order (as they appear on the Web site), we describe each family and related experiments in a logical order that follows the flow of the 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 PC-classes, hence

Table 3: Results reported for random and isomorphism class instances from uf250-1065 and uuf250-1065.

costID = timeToSolve (seconds)

(a) * * * * * * * sat instances * * * * * * * *

Class labels: *name*=uf250-1065, *type*=R, *size*=100

solverID	initV	minV	meanV	maxV	max/min
satoL	–	0.13	17.1	65.5	504
sato	–	0.07	105	1240	17800
chaff	–	0.10	116	1320	13200

Class labels: *name*=uf250-1065-027, *type*=PC, *size*=128

solverID	initV	minV	meanV	maxV	max/min
satoL	46.3	0.23	26.4	56.0	243
chaff	115	9.11	69.6	355	39.0
sato	23.1	0.13	92.7	678	5220

Class labels: *name*=uf250-1065-034, *type*=PC, *size*=128

solverID	initV	minV	meanV	maxV	max/min
satoL	6.26	0.15	26.3	56.6	377
chaff	552	3.29	199	1310	398
sato	224	1.19	223	1000	840

Class labels: *name*=uf250-1065-087, *type*=PC, *size*=128

solverID	initV	minV	meanV	maxV	max/min
satoL	51.6	0.580	30.9	66.4	114
sato	94.5	0.160	148	1100	6880
chaff	1320	114	655	1530	13.4

(b) * * * * * * * unsat instances * * * * * * *

Class labels: *name*=uuf250-1065, *type*=R, *size*=100

solverID	initV	minV	meanV	maxV	max/min
satoL	–	14.9	45.3	95.1	6.38
sato	–	58.2	494	1770	30.4
chaff	–	103	900	t'out1	> 17.5

Class labels: *name*=uuf250-1065-090, *type*=PC, *size*=128

solverID	initV	minV	meanV	maxV	max/min
satoL	22.4	21.8	22.7	24.0	1.10
sato	55.8	21.4	84.4	308	14.4
chaff	103	89.2	131	245	2.75

Class labels: *name*=uuf250-1065-074, *type*=PC, *size*=128

solverID	initV	minV	meanV	maxV	max/min
satoL	36.1	34.6	36.4	38.1	1.10
sato	431	81.0	304	1140	14.7
chaff	504	400	609	780	1.95

Class labels: *name*=uuf250-1065-046, *type*=PC, *size*=128

solverID	initV	minV	meanV	maxV	max/min
satoL	75.1	65.0	75.9	83.2	1.28
sato	952	440	1210	3420	7.77
chaff	2078	1912	2424	t'out2	> 1.88

costID = numberOfImplications

(a) * * * * * * * sat instances * * * * * * * *

Class labels: *name*=uf250-1065, *type*=R, *size*=100

solverID	initV	minV	meanV	maxV	max/min
sato	–	7900	1.95e6	9.55e6	1210
satoL	–	1.49e4	2.17e6	8.18e6	549
chaff	–	1.92E3	2.99e6	1.87e7	9740

Class labels: *name*=uf250-1065-027, *type*=PC, *size*=128

solverID	initV	minV	meanV	maxV	max/min
sato	1.05e6	1.40e4	2.00e6	6.81e6	486
chaff	4.48e6	8.14e5	2.84e6	9.40e6	11.5
satoL	5.93e6	2.79e4	3.36e6	7.09e6	254

Class labels: *name*=uf250-1065-034, *type*=PC, *size*=128

solverID	initV	minV	meanV	maxV	max/min
sato	3.88e6	1.07e5	3.25e6	8.19e6	76.5
satoL	7.88e5	1.87e4	3.48e6	7.18e6	383
chaff	1.18e7	4.11e5	4.40e6	1.74e7	42.3

Class labels: *name*=uf250-1065-087, *type*=PC, *size*=128

solverID	initV	minV	meanV	maxV	max/min
sato	2.42e6	1.56e4	2.42e6	8.85e6	567
satoL	6.56e6	7.57e4	3.90e6	8.32e6	110
chaff	1.81e7	4.45e6	1.19e7	2.04e7	4.58

(b) * * * * * * * unsat instances * * * * * * *

Class labels: *name*=uuf250-1065, *type*=R, *size*=100

solverID	initV	minV	meanV	maxV	max/min
sato	–	1.65e6	5.32e6	1.08e7	6.55
satoL	–	1.85e6	5.70e6	1.22e7	6.59
chaff	–	4.06e6	1.44e7	t'out1	> 6.0

Class labels: *name*=uuf250-1065-090, *type*=PC, *size*=128

solverID	initV	minV	meanV	maxV	max/min
sato	1.60e6	9.37e5	2.08e6	4.16e6	4.44
satoL	2.74e6	2.65e6	2.78e6	2.92e6	1.10
chaff	4.06e6	3.70e6	4.51e6	6.05e6	1.64

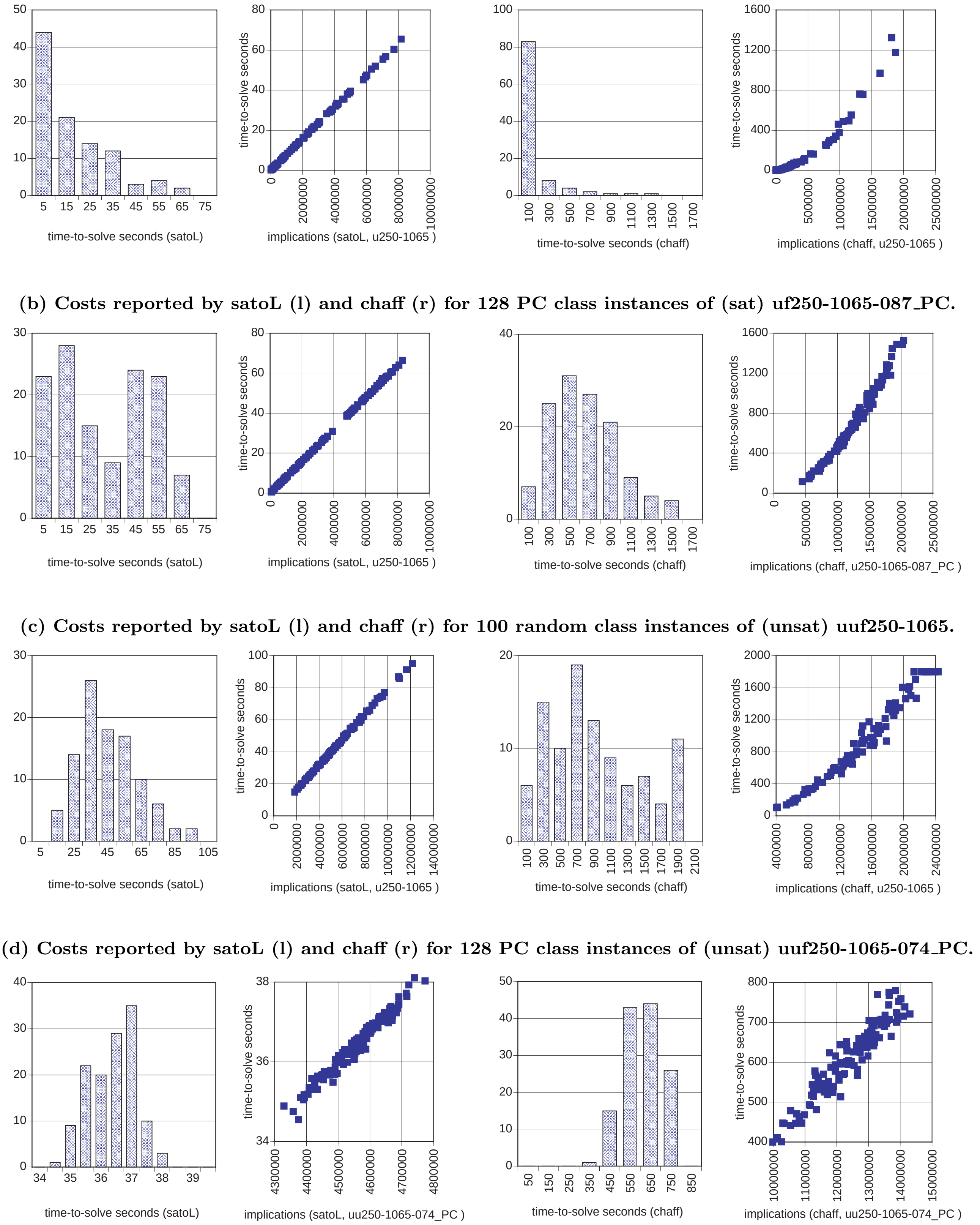
Class labels: *name*=uuf250-1065-074, *type*=PC, *size*=128

solverID	initV	minV	meanV	maxV	max/min
sato	5.13e6	2.13e6	4.11e6	8.10e6	3.80
satoL	4.54e6	4.33e6	4.55e6	4.77e6	1.10
chaff	1.08e7	3.98e6	1.24e7	1.43e7	3.59

Class labels: *name*=uuf250-1065-046, *type*=PC, *size*=128

solverID	initV	minV	meanV	maxV	max/min
sato	8.29e6	5.71e6	8.97e6	1.53e7	2.68
satoL	9.44e6	8.29e6	9.57e6	1.05e7	1.27
chaff	2.52e7	2.32e7	2.67e7	t'out2	> 1.18

NOTES:
(1) For each *class* and for each *costID*, the solverID ordering is induced by sorting on *meanV*.
(2) The value of *initV* is included in computation of *max/min* ratio.
(3) The *timeout* values have been set to t'out1 = 1800 seconds and t'out2 = 3600 seconds.
(4) For both random classes, values of *initV* have been set to '–' since instances from these classes relate to no reference formula.



(a) Costs reported by satoL (l) and chaff (r) for 100 random class instances of (sat) uf250-1065.

(b) Costs reported by satoL (l) and chaff (r) for 128 PC class instances of (sat) uf250-1065-087_PC.

(c) Costs reported by satoL (l) and chaff (r) for 100 random class instances of (unsat) uuf250-1065.

(d) Costs reported by satoL (l) and chaff (r) for 128 PC class instances of (unsat) uuf250-1065-074_PC.

Figure 7: Histograms and correlations: random/isomorphism class instances from uf250-1065 & uuf250-1065.

unless stated otherwise, only instances from the PC-class will be analyzed. Without exception, there are always 32 instances (or more) 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 the P-classes. All families we investigated illustrate key differences among solvers and offer insights for further study.

The aTutorialCNF family. We devised this family to illustrate the construction of equivalence classes and a number of insights that can be gained already on such simple formulas when various SAT solvers are applied. We argue the merits of the approach in Section 3, using the case of formula `v06_s0004.cnf`.

The hanoi families. The *hanoi* families consists of formulas that represent the classical tower of Hanoi problem. All formulas are satisfiable. Two directories are listed in Figure 6: `hanoi` and `hanoi_trim`. The first one contains reference formulas and equivalence classes of the original instances [21]. For the second directory, we applied unit-clause propagation to the reference formulas and removed a significant number of clauses and variables before generating equivalence classes.

The really challenging formulas are those of `hanoi05` and `hanoi06`. We were surprised to find that *satire*, one of the slower programs overall, found a verified solution to `hanoi05` in some 100 seconds while both *chaff* and *sato* timed out at 1800 seconds. However, running all three solvers on instances of the P- and PC-class, *not a single solution* was found by any of the three solvers. We made the same observations when trying out the three solvers on comparable instances from `hanoi_trim` family.

The random3sat_250-1065 families. The families of *random3sat_250-1065* consist of formulas selected from the two classes of largest random-3-sat satisfiable and unsatisfiable formulas from SATLIB [6]. Two directories are listed in Figure 6: `random3sat_250-1065_s`, designating the family of satisfiable formulas, and `random3sat_250-1065_u`, designating the family of unsatisfiable formulas.

The three formulas under `random3sat_250-1065_s` have been selected by examining time-to-solve results reported by *chaff* on 100 satisfiable formula instances in uf250-1065:

- `uf250-1065_027.cnf` is the instance '27', chosen as being the closest in time-to-solve value (115 seconds) to the mean value reported for *chaff* for all 100 instances (116 seconds). We have chosen an instance closest to the mean value rather than the minimum value since the minimum is 0.13 seconds and the instance may not be as interesting.
- `uf250-1065_034.cnf` is the instance '34', chosen since its time-to-solve reported by *chaff* (552 seconds) is closest to the value of 5x115 = 575 seconds.
- `uf250-1065_087.cnf` is the instance '87', chosen since its time-to-solve reported by *chaff* (1320 seconds) is the maximum value reported.

The three formulas under `random3sat_250-1065_u` have been selected by examining time-to-solve results reported by *chaff* on 100 unsatisfiable formula instances in uuf250-1065:

- `uuf250-1065_090.cnf` is the instance '90', chosen since its time-to-solve reported by *chaff* (103 seconds) is the minimum value reported.
- `uuf250-1065_074.cnf` is the instance '74', chosen since its time-to-solve reported by *chaff* (504 seconds) is closest to the value of 5x103 = 515 seconds.
- `uuf250-1065_046.cnf` is the instance '46', chosen since its time-to-solve reported by *chaff* (1800 seconds) is the time-out value.

Next, we generated 128 PC-class instances for each formula selected above and placed them into the respective family directories according to the schema `benchm_SATcnf` in Figure 5. In addition, we placed the 'original' 2 × 100 random class instances under uf250-1065_R and uuf250-1065_R as the random class directories next to PC-class instance directories. We then ran experiments with *chaff*, *sato*, and *satoL* on all class instances. A comprehensive statistical summary of these experiments is shown in Table 3.

The results displayed in Table 3 are surprising and revealing. Due to limited space, our discussion is focused on the reported time-to-solve statistics only. For the four classes that are satisfiable, we observe:

- The mean time-to-solve reported by *satoL* is by far superior to both *sato* and *chaff* – uniformly across all four classes.
- For the original random class only, the difference of the means of time-to-solve reported by *sato* and *chaff* is *not* significant (a *t*-test finds $t = 1.995$). This would be a statistically valid conclusion – had the skeptic accepted instances from the randomly generated class, even under conditions as described in [6], as a fair test case for the two solvers.
- There is an interesting situation for the three cases of the PC-class: the difference of the means of time-to-solve reported by *sato* and *chaff* is *not* significant wrt to instances in the class of uf250-1065_034_PC only (a *t*-test finds $t = 0.0346$). However, the difference is significant for instances in the other two classes: *chaff* appears more efficient for the class of uf250-1065_027_PC while *sato* appears significantly more efficient for the class of uf250-1065_087_PC.
- The most interesting result however is that, as far as *satoL* is concerned, instances in all of the three PC-classes are of comparable difficulty – the differences of the means of time-to-solve reported by *satoL* for each class are *not* significant (a *t*-test finds $t = 0.230, 0.917$, and 0.774, when comparing values of all pairs).
- Finally, the max/min ratios reported for the original 'random class' represent substantially different problem instances, unlike those of the carefully controlled classes introduced by the skeptic. Therefore, these ratios must be interpreted as resulting from the variability of problem instances rather than that of the heuristics.

For the four classes that are unsatisfiable, we observe:

- Uniformly across all four classes, the mean time-to-solve reported by *satoL* is consistently better than both *sato* and *chaff*, and the mean time-to-solve reported by *sato* is consistently better than *chaff*.
- While the mean time-to-solve reported by *satoL* is clearly different from class to class, the maximum time-to-solve increases to only 83.2 seconds from the corresponding value of 66.4 seconds for the satisfiable class instances. On the other hand, note that *chaff* times out at 3600 seconds on a number of instances.

Table 4: Three solver comparisons on PC class instances of the queen and hole families.

(a) * * * * * queen instances (sat) * * * * * *

costID = timeToSolve (seconds)

Class labels: *name*=queen10_v00100, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
sato	0.00	0.00	0.00	0.01	–
satoL	0.02	0.00	0.00	0.01	–
chaff	0.02	0.01	0.02	0.05	5.00

Class labels: *name*=queen14_v00196, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
sato	0.01	0.00	0.02	0.08	–
satoL	0.09	0.01	0.06	0.18	18.0
chaff	1.36	0.18	0.49	1.80	10.0

Class labels: *name*=queen16_v00256, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
sato	0.01	0.01	0.07	0.30	30.0
satoL	0.11	0.01	0.25	0.65	65.0
chaff	4.99	0.52	1.11	2.42	9.60

Class labels: *name*=queen19_v00361, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
sato	0.02	0.02	0.11	1.39	69.5
satoL	2.22	0.03	3.35	14.3	477
chaff	161	38.2	109	188	4.94

(a) * * * * * queen instances (sat) * * * * * *

costID = numberOfImplications

Class labels: *name*=queen10_v00100, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
sato	72	96	229	571	7.93
satoL	1122	88	276	583	12.75
chaff	4424	1353	3304	6542	4.84

Class labels: *name*=queen14_v00196, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
sato	142	190	1470	6910	48.7
satoL	3991	329	5024	1.64e4	49.7
chaff	8.73e4	2.06e4	3.94e4	9.17e4	4.45

Class labels: *name*=queen16_v00256, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
sato	180	248	4684	2.41e4	133
satoL	309	266	1.83e4	5.16e4	193
chaff	1.87e5	4.23e4	6.65e4	1.10e5	4.42

Class labels: *name*=queen19_v00361, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
sato	260	351	5649	9.89e4	380
satoL	1.35e5	512	2.08e5	9.09e5	1776
chaff	1.76e6	6.21e5	1.15e6	1.78e6	2.86

(b) * * * * * hole instances (unsat) * * * * * *

costID = timeToSolve (seconds)

Class labels: *name*=hole06_v00042, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
satoL	0.01	0.02	0.02	0.03	3.00
sato	0.04	0.03	0.05	0.06	2.00
chaff	0.01	0.07	0.11	0.14	14.0

Class labels: *name*=hole07_v00056_v00056, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
satoL	0.14	0.15	0.17	0.19	1.36
sato	0.16	0.21	0.29	0.34	2.12
chaff	0.42	0.86	1.36	2.04	4.86

Class labels: *name*=hole08_v00072, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
satoL	1.31	1.48	1.68	1.85	1.41
sato	6.76	1.43	1.92	2.60	4.72
chaff	1.25	9.49	16.2	26.3	21.1

Class labels: *name*=hole09_v00090, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
sato	9.07	14.5	17.3	21.4	2.36
satoL	13.2	16.5	17.9	19.5	1.48
chaff	7.23	85.4	178	274	37.9

Class labels: *name*=hole10_v00110, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
sato	98.8	158	181	206	2.09
satoL	152	202	217	227	1.49
chaff	47.1	320	451	736	15.6

(b) * * * * * hole instances (unsat) * * * * *

costID = numberOfImplications

Class labels: *name*=hole06_v00042, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
sato	5920	3663	6149	7928	2.16
satoL	9058	7438	8278	9191	1.24
chaff	2331	7930	1.04e4	1.26e4	5.40

Class labels: *name*=hole07_v00056_v00056, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
sato	6.72e4	4.05e4	5.04e4	6.26e4	1.66
chaff	2.87e4	4.12e4	5.57e4	7.06e4	2.46
satoL	7.03e4	6.00e4	6.69e4	7.29e4	1.22

Class labels: *name*=hole08_v00072, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
chaff	5.95e4	1.79e5	2.61e5	3.45e5	5.80
sato	5.37e5	3.23e5	4.46e5	5.76e5	1.78
satoL	6.17e5	5.20e5	6.05e5	6.79e5	1.30

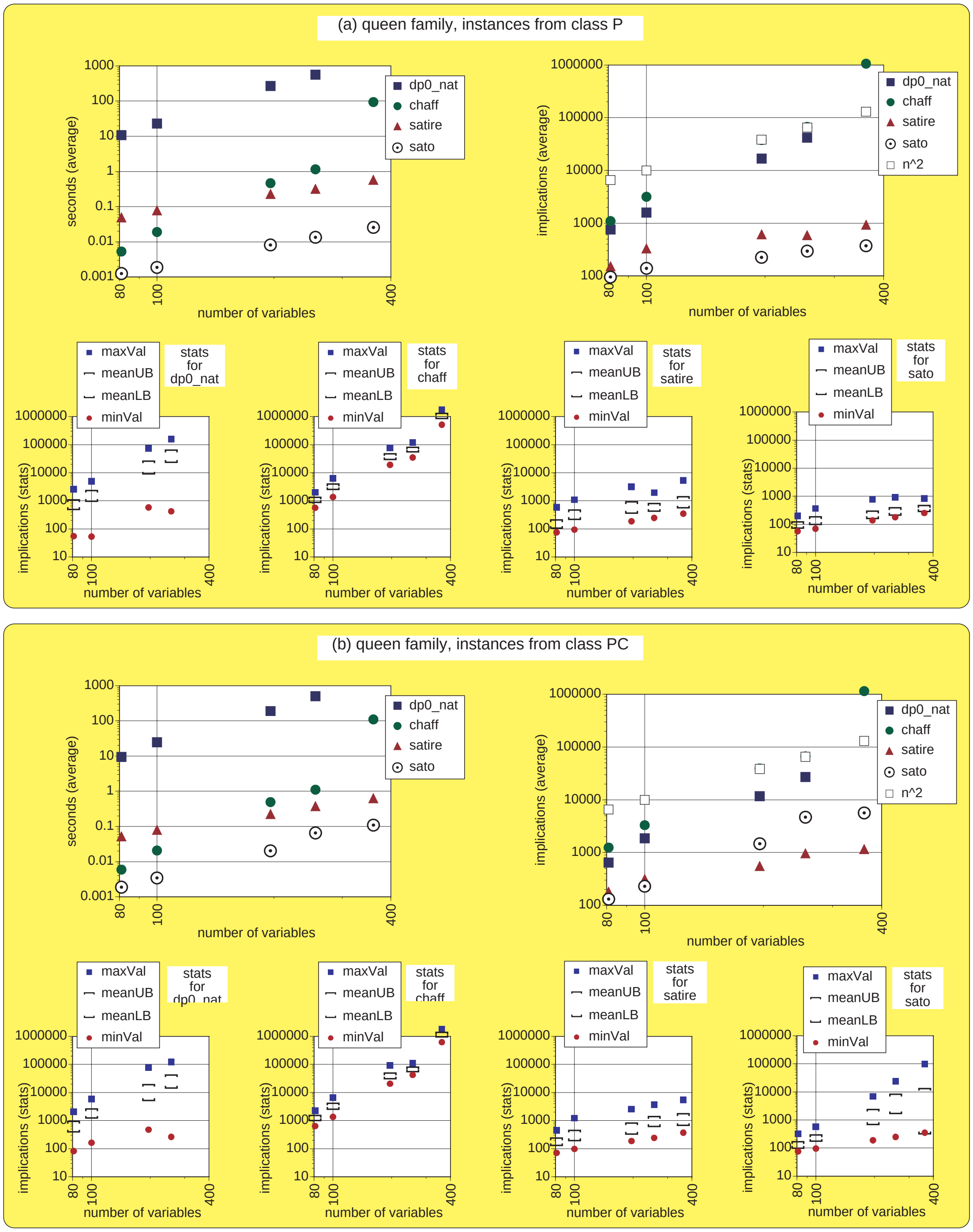
Class labels: *name*=hole09_v00090, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
chaff	1.61e5	6.61e5	1.04e6	1.39e6	8.64
sato	3.32e6	3.31e6	4.63e6	5.61e6	1.69
satoL	6.05e6	5.45e6	6.01e6	6.61e6	1.21

Class labels: *name*=hole10_v00110, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
chaff	4.58e5	1.60e6	1.90e6	2.25e6	4.92
sato	3.46e7	3.42e7	4.98e7	5.94e7	1.74
satoL	6.54e7	6.20e7	6.77e7	7.08e7	1.14

NOTES: (1) For each *class* and for each *costID*, the solverID ordering is induced by sorting on *meanV*.
(2) The value of *initV* is included in computation of *max/min* ratio.

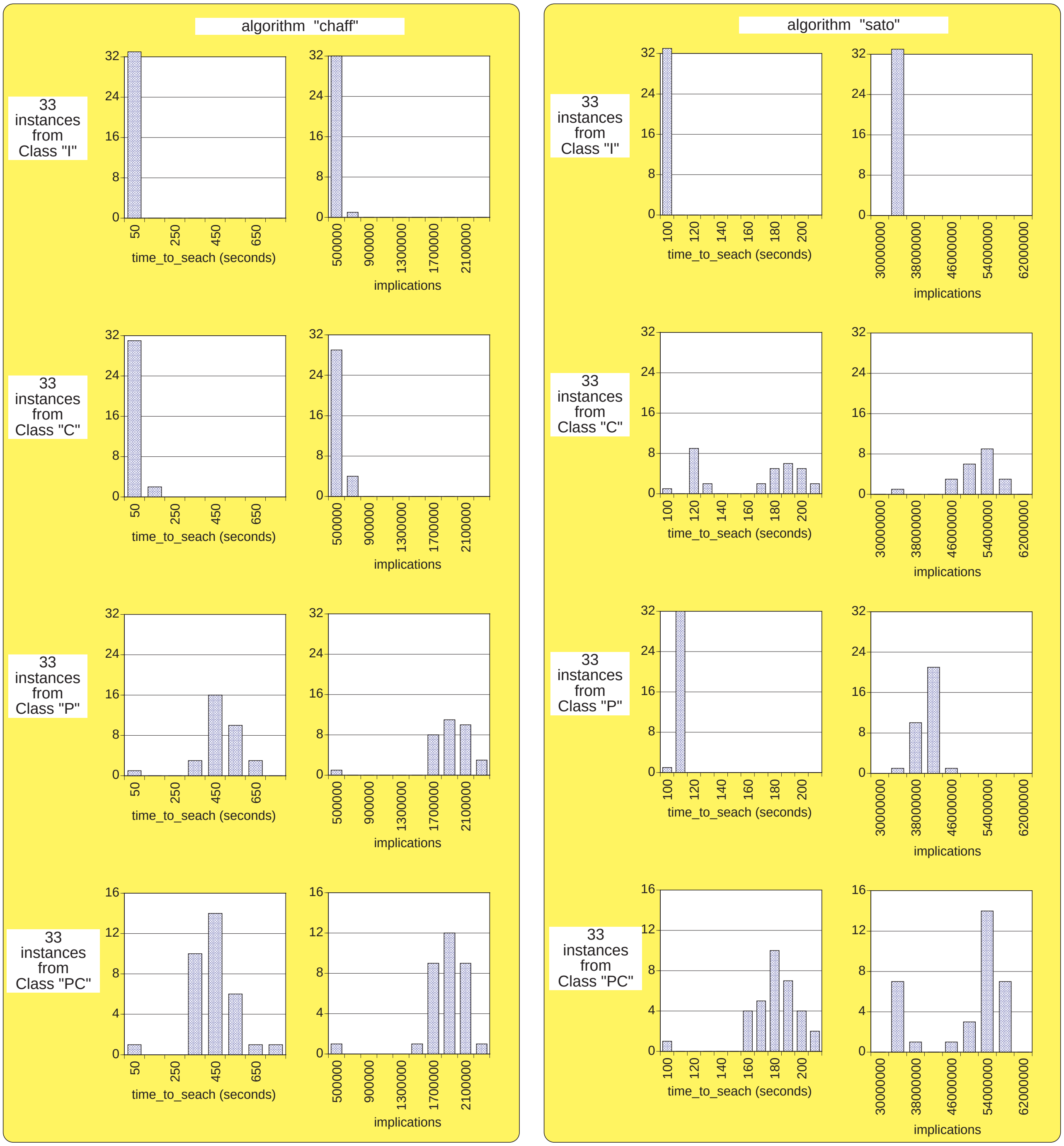


It is important to view the solver performance in the context of asymptotic trends, both in terms of time-to-solve and the platform-independent number-of-implications. Clearly, instances from the PC-class are the only choice to reliably test the performance of various solvers. Significant observations for this series of experiments also include:

- asymptotic performance in number-of-implications of *dp0_nat* (coded in interpreted language Tcl [30]) 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 max/min performance is significantly larger than the already large variability found for the non-satisfiable problems such as the 'hole family'. The same observation applies to [meanLB, meanUB], 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.

The formula for hole10 has 110 variables and is the largest reference formula from the 'hole family' where solvers *chaff* and *sato*, applied to instances of classes 'P' and 'PC', do not time out at 1800 seconds. The solver *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 solvers *chaff* and *sato should* be compared is the 'PC'-class. No *t*-test is required to confirm that, for hole10, the solver 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 instances in the P and PC class induce significant variability in the performance of the *chaff* solver. A fair comparison with the *sato* solver can only be made on basis of experiments with the PC class.

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

Figure 9: Histograms of two SAT solvers applied to instances in four equivalence classes of 'hole10'.

- When compared to variability of max/min ratios observed for satisfiable instances, the variability observed for the unsatisfiable instances is significantly lower, which is a result we would expect.

In Figure 7 we show histograms and correlations for satisfiable and unsatisfiable random-class and the PC-class instances whose statistics we discussed in Table 3. A few brief observations follow:

- There is a dramatic difference in distributions shown for the satisfiable and unsatisfiable random-class instances.
- Regardless of the solver, the distribution for the satisfiable instances from the random-class tends to follow a Poisson distribution, while the distribution for the unsatisfiable instances from the random-class tends to follow a Gaussian distribution.
- In contrast, regardless of the solver, the distributions from both satisfiable and unsatisfiable instances of the PC-class tend to follow a Gaussian distribution.
- There is near perfect linear correlation of the time-to-solve versus the number-of-implications for the *satoL* solver, while the correlation for *chaff* is also high but not as linear.

The queen families. The *queen* families consists of formulas that represent the classical queen problem. All formulas are satisfiable. Two directories are listed in Figure 6: `queen_small` and `queen_medium`.

A comprehensive statistical summary of experiments with *chaff*, *sato*, and *satoL* on instances from the PC-class of `queen_medium` is shown in Table 4(a). The superior performance of *sato* versus *satoL* and *chaff* can be readily ascertained, both in terms of time-to-solve and number-of-implications. Note also the variability of the max/min ratio, peaking at 1776 for *satoL*.

In Figure 8 we show the asymptotic performance not only for *chaff*, *satire*, and *sato* but also for *dp0_nat*. It is clear that while *sato* is indeed a very competitive solver, its performance is significantly degraded when we consider the PC-class rather than P-class – this startling sensitivity to variable complementation has already been observed on the 6-variable example in Figure 4(b). In contrast, the performance of *chaff*, *dp0_nat*, and *satire* is basically the same for both classes.

It is important to view the solver performance in the context of asymptotic trends, both in terms of time-to-solve and the platform-independent number-of-implications. Clearly, instances from the PC-class are the only choice to reliably test the performance of various solvers. Thus, for the instances from the PC-class of `queen_medium` family, ordering by the mean value of number-of-implications (a platform-independent comparison) induces the following order on the solvers: *satire*, *sato*, *dp0_nat*, and *chaff*. It is unfortunate that the current implementation of *satire* and *dp0_nat* makes the two solvers too slow to run experiments next to *sato* and *chaff* on larger problem instances in most other families.

The hole families. The *hole* families consist of formulas that represent the pigeon hole principle. All formulas are unsatisfiable. Two directories are listed in Figure 6: `hole_small` and `hole_medium`. Instances derived from the formula for `hole14`, the largest formula in the `hole_medium` family, time out at 1800 seconds for all solvers. As noted here, an instance need not represent thousands of variables. Instances with only 210 variables provide a non-trivial challenge.

A comprehensive statistical summary of experiments with *chaff*, *sato*, and *satoL* on instances from the PC-class of `hole_medium` is shown in Table 4(b). There is a crossover in time-to-solve performance between *sato* and *satoL*, whereas for number-of-implications *chaff* starts to dominate already for instances from class hole08_v00072. Note that compared to the extremely large max/min ratios for instances from the (satisfiable) queen problem, the peak max/min ratios for instances from the (unsatisfiable) hole problem amount to at most 37.9 (for chaff). The histograms in Figure 9 illustrate a number of universal observation we made throughout these experiments:

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

Clearly, the PC-class is the class in which we should compare the performance of all SAT solvers.

The bw_large families. The satisfiable and the unsatisfiable families of `bw_large`, listed in Figure 6, are a subset of the SATPLAN benchmarks, a collection of satisfiability problems based on AI planning scenarios developed in [22]. The most difficult of these come from the well-known blocks-world domain in AI (see, e.g. [35]).

A comprehensive statistical summary of experiments with *chaff*, *sato*, and *satoL* on instances from the PC-class of satisfiable and unsatisfiable instances from `bw_large` is shown in Table 5. Here, no $t-$tests are required to determine that *chaff* outranks *sato*, which significantly outranks *satoL*. The max/min ratios are much larger for the satisfiable instances again, peaking at 39.3 for *satoL*. The satisfiable blocks-world instances illustrate the sensitivity of *sato* to complementation more starkly than any other family we tested. While for the P-class (results posted on the Web), *sato* performance appears close to that of *chaff*, this clearly is not the case for the PC-class instances.

The sched families. The satisfiable and the unsatisfiable 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 [36] and [37] for a survey). Many variations involving release times, deadlines, resource constraints, and precedence constraints are NP-complete (see, e.g. [36] and [38]). 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. [39] 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

Table 5: Three solver comparisons on PC class instances of the bw_large families.

costID = timeToSolve (seconds)

(a) * * * * * * sat instances * * * * * *

Class labels: *name*=bw_large_a_s, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
chaff	0.01	0.00	0.01	0.02	–
satoL	0.01	0.02	0.03	0.04	4.00
sato	0.03	0.03	0.04	0.06	2.00

Class labels: *name*=bw_large_b_s, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
chaff	0.08	0.03	0.08	0.16	5.33
sato	0.24	0.34	0.79	1.58	6.58
satoL	1.34	0.09	0.97	2.78	30.9

Class labels: *name*=bw_large_c_s, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
chaff	4.34	0.39	4.18	8.71	22.3
sato	9.14	78.5	123	235	9.12
satoL	68.3	616	1712	2686	39.3

(b) * * * * * * unsat instances * * * * * *

Class labels: *name*=bw_large_a_u, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
chaff	0.00	0.00	0.00	0.00	–
sato	0.01	0.00	0.01	0.02	–
satoL	0.00	0.00	0.01	0.02	–

Class labels: *name*=bw_large_b_u, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
chaff	0.05	0.03	0.05	0.07	2.33
sato	0.06	0.21	0.37	0.56	9.33
satoL	0.48	0.34	0.55	0.83	2.44

Class labels: *name*=bw_large_c_u, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
chaff	2.23	1.57	2.90	5.31	3.38
sato	6.29	25.8	46.5	68.1	10.8
satoL	663	485	609	870	1.79

costID = numberOfImplications

(a) * * * * * * sat instances * * * * * *

Class labels: *name*=bw_large_a_s, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
satoL	447	757	1964	3677	8.23
chaff	3085	1103	2229	3898	3.53
sato	1780	2538	3465	4796	2.69

Class labels: *name*=bw_large_b_s, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
chaff	2.39e4	5420	2.11e4	4.10e4	7.57
sato	2.88e4	1.54e4	3.67e4	7.90e4	5.13
satoL	1.01e5	2528	5.06e4	1.47e5	58.0

Class labels: *name*=bw_large_c_s, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
chaff	1.10e6	7.37e4	7.57e5	1.61e6	21.9
sato	7.45e5	2.40e6	3.85e6	6.98e6	9.37
satoL	2.90e6	1.56e7	4.28e7	6.99e7	24.1

(b) * * * * * * unsat instances * * * * * *

Class labels: *name*=bw_large_a_u, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
satoL	862	541	946	1190	2.20
chaff	1080	727	987	1604	2.21
sato	672	567	1036	2772	5.91

Class labels: *name*=bw_large_b_u, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
chaff	1.44e4	6505	1.19e4	1.70e4	2.61
sato	1.43e4	9518	1.86e4	2.95e4	3.10
satoL	3.48e4	1.68e4	2.61e4	3.84e4	2.29

Class labels: *name*=bw_large_c_u, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
chaff	6.16e5	3.26e5	5.36e5	8.91e5	2.74
sato	5.26e5	8.72e5	1.55e6	2.42e6	4.60
satoL	2.51e7	1.24e7	1.58e7	2.10e7	2.02

NOTES: (1) For each *class* and for each *costID*, the solverID ordering is induced by sorting on *meanV*.
(2) The value of *initV* is included in computation of *max/min* ratio.

the deadlines of two tasks, making them "barely infeasible". Sizes of the scheduling instances are listed in Figure 6.

A comprehensive statistical summary of experiments with *chaff*, *sato*, and *satoL* on instances from the PC-class of satisfiable and unsatisfiable instances of sched is shown in Table 6. In contrast to the blocks-world and hole instances, the scheduling instances are a major success story for *satoL*, with *sato* yielding to *chaff* on larger instances. Most surprisingly, here the complementation, intrinsic to the PC-class, has negligible effect on the max/min ratios reported by *satoL*, whereas these ratios peak at 410 for *chaff* and 156,301 for *sato*! Moreover, the average-case performance of *satoL* is two orders of magnitude better than the performance reported by *chaff*, and three orders of magnitude better than the performance reported by *sato*! Major reason reason for the degradation in performance of *sato* is that it times out or nearly times out on several instances that represent no problem to other solvers. Clearly, a solver that performs extremely well in one domain (e.g. *sato* on the queen problem) may perform quite unpredictably in another.

Variability – Final Observations. Recent experiments that addressed variability of solver performance attributed the variability to 'easy' problems turning out to be 'hard' or equivalently, to 'heavy-tail' phenomena [40, 41, 42]. The experiments with four distinct isomorphism classes formulated in this paper point out that more attention should be paid to the current generation of SAT solvers, not the problem instances per se. For example, the PC-class induces high variability on *chaff* performance because the P-class induces such variability, not the C-class. However, in most cases for *sato* and *satoL*, the high variability demonstrated for the PC-class is due to variability induced by the C-class.

To put the time-to-solve variability in perspective, we note that for *chaff*, the maximum reported max/min ratio is 398 with mean of 199 seconds for class uf250-1065-034_PC whereas *satoL* reports a ratio of 377 and a mean of 26.3 seconds (see Table 3). On the other hand, for *sato*, the maximum reported max/min ratio is 39,713 with mean of 238 seconds for class sched07s_v01286_PC whereas *satoL* reports a ratio of 1.83 and a mean of 0.07 seconds (see Table 6).

Table 6: **Three solver comparisons on PC class instances of the sched families.**

costID = timeToSolve (seconds)

(a) * * * * * * sat instances * * * * * *

Class labels: *name*=sched05s_v00412, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
satoL	0.01	0.00	0.01	0.01	–
sato	0.01	0.00	0.01	0.02	–
chaff	0.12	0.08	5.59	40.5	505

Class labels: *name*=sched06s_v00828, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
satoL	0.03	0.02	0.03	0.04	2.00
sato	0.04	0.04	4.09	81.9	2046
chaff	5.74	0.28	16.4	69.2	247

Class labels: *name*=sched07s_v01386, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
satoL	0.09	0.06	0.07	0.11	1.83
chaff	22.3	3.51	15.5	69.5	19.8
sato	0.11	0.09	238	t'out	$> 39,713$

(b) * * * * * * unsat instances * * * * * *

Class labels: *name*=sched05u_v00410, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
satoL	0.02	0.02	0.02	0.03	1.50
sato	0.02	0.01	0.03	0.22	22.0
chaff	0.14	0.13	5.01	53.3	410

Class labels: *name*=sched06u_v00826, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
satoL	0.07	0.10	0.10	0.11	1.57
sato	0.10	0.08	0.91	26.1	326
chaff	6.31	0.97	15.4	60.5	62.4

Class labels: *name*=sched07u_v01384, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
satoL	0.22	0.28	0.29	0.30	1.36
chaff	7.39	0.58	16.1	47.8	82.5
sato	0.18	0.23	385	t'out	$> 20,086$

costID = numberOfImplications

(a) * * * * * * sat instances * * * * * *

Class labels: *name*=sched05s_v00412, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
sato	410	297	393	1366	4.60
satoL	405	405	409	410	1.01
chaff	3.90e4	2.44e4	3.34e5	1.30e6	53.2

Class labels: *name*=sched06s_v00828, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
satoL	819	821	824	826	1.01
sato	826	673	3.40e5	7.30e6	10842
chaff	5.22e5	9.17e4	7.62e5	1.83e6	20.0

Class labels: *name*=sched07s_v01386, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
satoL	1377	1379	1383	1384	1.01
chaff	1.51e6	4.11e5	9.82e5	2.18e6	5.30
sato	1384	1220	1.21e7	t'out	$> 156,301$

(b) * * * * * * unsat instances * * * * * *

Class labels: *name*=sched05u_v00410, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
satoL	2387	2349	2382	2413	1.03
sato	2387	2083	2927	2.50e4	12.0
chaff	4.32e4	3.72e4	3.11e5	1.49e6	40.0

Class labels: *name*=sched06u_v00826, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
satoL	6254	6176	6239	6276	1.02
sato	6254	5640	8.30e4	2.48e6	438
chaff	5.33e5	2.45e5	7.84e5	1.57e6	6.43

Class labels: *name*=sched07u_v01384, *type*=PC, *size*=32

solverID	initV	minV	meanV	maxV	max/min
satoL	1.09e4	1.08e4	1.09e4	1.10e4	1.01
chaff	8.17e5	1.41e5	1.06e6	2.47e6	17.6
sato	1.09e4	1.02e4	1.92e7	t'out	$> 18,887$

NOTES: (1) For each *class* and for each *costID*, the solverID ordering is induced by sorting on *meanV*.
(2) The value of *initV* is included in computation of *max/min* ratio.
(3) The *timeout* values have been set to t'out = 3600 seconds.

Significantly, for the entire sched family in Table 6, *satoL* dramatically outperforms *chaff* and *sato* while max/min ratio remains under 2.0 for time-to-solve *and* is no more than 1.05 for number-of-implications!

The variability and the Poisson distribution exhibited for results on a class of satisfiable instances (see Figure 7 and Table 3) suggest that experiments on sets of random instances are not a viable strategy for *statistically significant* comparisons among heuristics. Our skeptic approach, however, leads to statistically reasonable data (Gaussian with lower variance) when applied to single random instances as well as to structured benchmark instances.

6. CONCLUSIONS

As already noted, our experiments used the skeptic to reveal significant differences in relative SAT solver 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 solvers are likely to be the ones that their designers can identify and fix. And the skeptic, along with supporting infrastructure, is a tool for validating such improvements.

- In this paper we focus on experimental methodology and what it reveals about externally measurable behavior of SAT solvers. An important next step is the analysis of the heuristics in these solvers. 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, i.e. developing more sophisticated skeptics,
  - improving the usability of our experimental design scaffolding, and
  - extending the skeptic to other problem domains in CAD and industrial engineering.

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. A home page using the `SATcnf` structure in Figure 5 has been initiated under

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

All reference formulas and equivalence classes listed in Figure 6 reside as compressed archives under `benchm_SATcnf` and experimenters are invited to try them out. Raw results and statistical summaries of all experiments reported in this paper can be found under `results_SATcnf`. By the end of June 2002, links will be provided to the cross-platform experimental design environment and utilities we used in these experiments.

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] H. Hoos and T. Stuetzle. Satlib: An online resource for research on sat, 2000.

[7] 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`.

[8] 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`.

[9] Aarti Gupta, Anubhav Gupta, Zijiang Yang, and Pranav Ashar. Dynamic detection and removal of inactive clauses in SAT with application in image computation. In *IEEE/ACM Design Automation Conference (DAC)*, 2001.

[10] 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`.

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

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

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

[14] Sara Baase and Allen Van Gelder. *Computer Algorithms*. Addison-Wesley, third edition, 2000.

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

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

[17] 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`.

[18] J. E. Harlow and F. Brglez. Design of Experiments and Evaluation of BDD Ordering Heuristics. In *Software Tools for Technology Transfer (STTT), Special Issue on BDDs*. Springer-Verlag, 2001. A reprint also available under `http://www.cbl.ncsu.edu/publications/`.

[19] 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`.

[20] 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`.

[21] 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`.

[22] 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/`.

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

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

[25] 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`.

[26] 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.

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

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

[29] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

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

[31] D. B. West. *Introduction to Graph Theory*. Prentice-Hall, 1996.

[32] 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.

[33] JMP, The Statistical Discovery Software. April 2002. See http://www.jmpdiscovery.com/.

[34] J. C. Hsu. *Multiple Comparisons: Theory and Methods*. Chapman & Hall, 1996.

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

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

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

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

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

[40] Ian P. Gent and Toby Walsh. Easy problems are sometimes hard. *Artificial Intelligence*, 70(1-2):335–345, 1994.

[41] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proc. of AAAI-98, Madison/WI, USA*, pages 431–437, 1998.

[42] Erica Melis Andreas Meier, Carla Gomes. Heavy-tailed behavior and randomization in proof planning. In *Workshop on Model-Based Validation of Intelligence on AAAI Spring Symposium*, 2001.