sgen1: A Generator of Small but Difficult Satisfiability Benchmarks

IVOR SPENCE
Queen's University Belfast

The satisfiability problem is known to be NP-Complete; therefore, there should be relatively small problem instances that take a very long time to solve. However, most of the smaller benchmarks that were once thought challenging, especially the satisfiable ones, can be processed quickly by modern SAT-solvers. We describe and make available a generator that produces both unsatisfiable and, more significantly, satisfiable formulae that take longer to solve than any others known. At the two most recent international SAT Competitions, the smallest unsolved benchmarks were created by this generator. We analyze the results of all solvers in the most recent competition when applied to these benchmarks and also present our own more focused experiments.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Complexity of Proof Procedures; Computation on Discrete Structures*; F.4.1 [**Mathematical Logic**]: Logic and Constraint Programming

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Satisfiability benchmarks, SAT-solvers

ACM Reference Format:

Spence, I. 2010. sgen1: A generator of small but difficult satisfiability benchmarks. ACM J. Exp. Algor. 15, Article 1.2 (January 2010), 15 pages.
DOI = 10.1145/1671970.1671972 http://doi.acm.org/10.1145/1671970.1671972

1. INTRODUCTION

The boolean satisfiability problem is of considerable interest and importance, both because of its position as arguably the best-known NP-Complete problem [Cook 1971] and also as an ever more popular intermediate step in solving a range of practical problems [Batory 2005; Smith et al. 2003; Safarpour et al. 2006; Rintanen et al. 2006]. This is evidenced by the considerable efforts which are expended on the (now biennial) SAT Competitions [le Berre 2009] in which typically 50 solvers compete to solve more than 1,000 formulae.

Author's address: Ivor Spence, School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, University Road, Belfast BT7 1NN, United Kingdom; email: i.spence@qub.ac.uk.

DOI 10.1145/1671970.1671972 http://doi.acm.org/10.1145/1671970.1671972

Modern satisfiability solvers can process many of the formulae that were once regarded as challenging and, as noted by Aloul et al. [2003], advances in solvers had exceeded those of benchmark creators [Haanpää et al. 2006; Xu et al. 2007; Haixia et al. 2005; Hirsch 2002; Crawford et al. 1994], especially for satisfiable benchmarks.

1.1 Propositional Formulae

The satisfiability problem is to take a boolean proposition (formula) in conjunctive normal form [Whitesitt 1995] and determine whether there is an assignment of the values true and false to the variables such that the whole proposition has the value true. If this is possible, then the formula is said to be satisfiable and the corresponding values of the variables constitute a satisfying model, otherwise the formula is said to be unsatisfiable. The typical mathematical notation is to use p to denote a variable, $\overline{p}$ to denote the negation of p, and $\wedge$, $\vee$ to denote logical and and or, respectively. p are $\overline{p}$ are known as literals. A clause is a sequence of literals separated by $\vee$ operators and enclosed in parenthesis. A formula is a sequence of clauses separated by $\wedge$ operators.

Thus, $(p \vee q)$ is a clause, $(p \vee q) \wedge (\overline{p} \vee \overline{q})$ is a formula that is satisfiable with $\{p, \overline{q}\}$ being a satisfying model, and the formula $(p) \wedge (q) \wedge (\overline{p} \vee \overline{q})$ is unsatisfiable.

The standard for representing formulae by text files is the DIMACS [1993] format and files have the standard extension `.cnf`—this is the format used during the SAT Competitions. In such a file, the variables are represented by non-zero integers, with a positive value indicating the literal p and a negative value indicating $\overline{p}$. The first line of the file is of the form

```
p cnf num-of-variables num-of-clauses,
```

and thereafter, clauses are expressed as zero-terminated sequences of integers (conventionally each clause is on a separate line). For example, if p is represented by 1 and q by 2, then the file to represent $(p \vee q) \wedge (\overline{p} \vee \overline{q})$ is

```
p cnf 2 2
1 2 0
-1 -2 0
```

1.2 SAT-Solvers

Given the interest in solving the satisfiability problem it is not surprising that many programs (SAT-solvers) have been written to do this [Zhang and Malik 2002]. These can be divided into two categories:

—A complete solver can solve both satisfiable and unsatisfiable formulae. Most complete solvers are based at least loosely on the Davis-Putnam-Logemann-Loveland Procedure (DPLL) [Davis and Putnam 1960; Davis et al. 1962] and involve a systematic search that terminates when the search space is exhausted if no satisfying model is found.

—An incomplete solver cannot guarantee to determine that a formula is unsatisfiable—many are not able to report unsatisfiable as an answer.

However, when applied to a satisfiable formula, such a solver may well discover a satisfying model much more quickly than a complete solver. Many are based on the concept of a "random walk" [Krishnamachari et al. 2000], which does not terminate unless a satisfying model is found.

1.3 Formula Size

There are two popular metrics for assessing the size of a formula and thereby its relative difficulty, namely the number of variables n and the number of literals l. Within a particular series of formulae, there is often a simple linear relationship between these values and it does not particularly matter which is chosen, but when comparing very different formulae, this relationship breaks down. Consider for example the series `unsat-`n of pathological unsatisfiable formulae in which the nth element has n variables and 2^n clauses. Each clause has n literals (one occurrence of each variable, either positive or negative), and there is a clause to refute each possible assignment of the variables. For example, the formula `unsat-2` is

$$(\overline{p} \vee \overline{q}) \wedge (\overline{p} \vee q) \wedge (p \vee \overline{q}) \wedge (p \vee q)$$

and the corresponding `cnf` file is,

```
p cnf 2 4
-1 -2 0
-1 2 0
1 -2 0
1 2 0
```

The number of literals in `unsat-`n is $n2^n$, so for a comparatively small number of variables, a formula with an enormous number of literals can be created. Each one of these literals must be examined to determine the satisfiability of the formula because, although `unsat-`n is unsatisfiable, if any single literal were to change (even just the sign) it would become satisfiable. Thus, the formula takes at least $O(l) = O(n2^n)$ time to process. For example, `unsat-23` exceeds 600MB and takes the best solvers more than 1,200 seconds to solve even though it has only 23 variables.

This series can also be used to derive difficult satisfiable formulae. Deleting a random clause from `unsat-`n yields a similarly large but satisfiable formula containing $n(2^n - 1)$ literals, with the single satisfying model given by negating every literal in the clause which was deleted. For example, deleting the clause $(p \vee \overline{q})$ from `unsat-2` yields $(p \vee q) \wedge (\overline{p} \vee q) \wedge (\overline{p} \vee \overline{q})$, which is satisfied by $\{\overline{p}, q\}$. To solve one of these satisfiable formulae, at least each clause has to be examined, so the complexity is at least $O(2^n)$, and in practice, it is hard to see how a solver could avoid having to read every literal.

In short, if we use "number of variables" as the measure of size, the satisfiability problem is clearly exponential and it is easy to generate some very difficult instances. We focus instead on "number of literals" as the more interesting measure of size. This is consistent with the theoretical considerations, which lead to the problem being considered NP-Complete with its exponential complexity being still an open question [Fortnow 2009].

1.4 Sources of Formula

Formulae typically arise either as a result of converting a problem from a different domain into an equivalent satisfiability problem, or as the output of a generator. In the SAT Competitions, three categories of formula are considered:

—Crafted: Formulae that are carefully constructed to be difficult to solve, often based on a known problem from a different domain.
—Random: Formulae that are generated using a random number generator but satisfying some constraints, such as ratio of number of variables to number of clauses.
—Industrial/Application: Formulae that result from the conversion of some other problem. These are often much larger than the formulae from the other categories but, for their size, can be surprisingly easy to solve.

We are here interested particularly in the "crafted" category, where the smallest and most difficult problems have been found to lie. According to the results of the SAT Competitions, the nearest competitors for `sgen1` are `hgen2` (satisfiable) and `hgen8` (unsatisfiable) [Hirsch 2002], and all the formulae generated by these three programs lie in the "crafted" category.

2. THE METHODS USED BY `sgen1`

We describe two methods of generating formulae of a requested size, for unsatisfiable and then for satisfiable formulae. Notwithstanding the argument in Section 1.3, for historical reasons the size of a formula is requested in terms of the number of variables. The generation process requires a random seed as input, meaning that even for a given number of variables, it is possible to generate different formulae. In both the unsatisfiable and satisfiable cases, repeated use is made of the idea of partitioning the variables into (nearly) equally sized disjoint sets and how this is done is described first. Then, we explain the two different ways in which clauses are constructed from these sets as required.

2.1 Grouping Previously Unrelated Variables

A DPLL-based solver typically makes partial assignments, that is assignments to some of the variables, in an attempt to satisfy as many clauses as possible. If a partial assignment is found to refute one of the clauses in the formula, this can be recorded and used subsequently to prune the search tree. One factor that is likely to make a formula hard to solve is when partial assignments include many variables before any clause is refuted. This happens more often when the same variables do not repeatedly occur in the same clauses (two variables which occur in the same clause are said to be *neighbors*), and so one goal of formula generation is to reduce the occurrences of this. Although this goal is based on an analysis of the DPLL procedure, it should be noted that all the solvers participating in the SAT2007 and SAT2009 Competitions find the resulting formulae difficult.

In his hgen2 generator, Hirsch [2002] used the idea of constructing clauses in turn by randomly choosing variables and only placing two variables in the same clause if they had not previously appeared in the same clause, or in fact only if they did not even share a neighbor. If it proved impossible to find such a variable, generation of the current benchmark was abandoned and a new benchmark was begun. This restarting was more likely to be necessary when using fewer variables, and a lower limit of 250 was imposed on the number of variables.

In sgen1, this idea is adapted by generating a sequence of partitions of the variables into disjoint sets. Two variables which are members of the same set are said to be neighbors in the partition. The way in which clauses are generated from partitions (see later discussion) means that this is equivalent to saying that they occur in the same clause. The fact that two variables are neighbors in the the first partition means that we want to reduce the probability of them being neighbors in a subsequent partition. To achieve this, we define a fitness function based on the neighborhood relationships of the variables and use simulated annealing [Kirkpatrick et al. 1983] to search for a partition that minimizes this function. A move in this method consists of selecting two variables at random and deciding whether or not to swap them. If the swap results in a better partition, then we do make it. Even if it would result in a worse partition, we still consider making the swap, using a probability that is based on the penalty arising from the swap and becomes smaller as the method proceeds. Overall, this makes it less likely that the algorithm will get stuck in a local minimum in the search to get close to a global minimum.

The first partition is on the basis of numerical order, so that for example to obtain sets of cardinality four we would use $\{1, 2, 3, 4\}$, $\{5, 6, 7, 8\}$, and so on. The fitness function applied to a potential second partition gives a weighting of 16 for every pair of variables that are in the same set for a second time, that is, when neighbors in the first partition are also neighbors in the second one. An additional weighting of 1 is added when two pairs of neighbors in the second partition have been interchanged from the first, that is, when $\{a, b, \ldots\}$, $\{c, d, \ldots\}$ in the first partition becomes $\{a, c, \ldots\}$, $\{b, d, \ldots\}$ in the second. In principle, the fitness function should be the sum of all these weightings. In practice, to reduce the execution time, we approximate this fitness function when deciding whether or not to make a swap. We consider only the changes resulting directly from the variables being swapped, which results in some indirect changes not being taken into account.

In contrast to hgen2, using simulated annealing will always give a partition, and there is never a need to restart the algorithm, but it is not guaranteed to yield a global minimum of the fitness function. The partitioning process does take a noticeable amount of time, but even so, any of the formulae mentioned in this article can be generated within a few seconds. The fitness function and its approximation have been chosen empirically and give good results with an acceptable formula generation time, but of course there may well be even better functions that could be used.

2.2 Generating Unsatisfiable Formulae

Generating difficult unsatisfiable formulae with a small number of literals requires a balance between the following observations:

—to keep the formula short, each clause should eliminate a large number of possible models;
—to make the formula hard to solve, the variables in each clause should not be "related," that is, they should not occur together in multiple clauses.

Unfortunately, these requirements conflict. The easiest way for a clause to eliminate a large number of possible models is for it to contain the same variables as other clauses. For example, the presence of the clauses $(p \vee q)$, $(p \vee \overline{q})$, and $(\overline{p} \vee q)$ eliminates all assignments not containing $\{\overline{p}, \overline{q}\}$, which might be expected to be $\frac{3}{4}$ of all possible assignments. In the spirit of Hirsch's `hgen8` program, the compromise we have used is to partition the variables into small sets and, for each set, have a limited number of clauses containing exactly those variables. We use two partitions and generate clauses that impose incompatible limits on the overall number of variables that can be true and the number that can be false. The sets are all of cardinality four, except that in each partition there is exactly one set of cardinality five. To permit such a partition, it will not always be possible to create a formula with exactly the requested number of variables. If necessary, the number of variables is increased until it is of the form $4g+1$ where $g \in \mathbb{N}$, $g > 0$. The generated formula will, therefore, contain at least the requested number of variables but may contain up to three more. Let n be this actual number of variables and assume that we have partitioned the variables into $(g-1)$ sets of cardinality four and one set of cardinality five.

For each set $\{a, b, c, d\}$ of cardinality four, we generate all possible 3-clauses of negative literals, that is,

$$(\overline{a} \vee \overline{b} \vee \overline{c}) \wedge (\overline{a} \vee \overline{b} \vee \overline{d}) \wedge (\overline{a} \vee \overline{c} \vee \overline{d}) \wedge (\overline{b} \vee \overline{c} \vee \overline{d})$$

This permits at most two variables from the set $\{a, b, c, d\}$ to be *true*, because any set of three that were all *true* (e.g., $\{a, b, c\}$) would fail to satisfy one of the clauses (in this case $(\overline{a} \vee \overline{b} \vee \overline{c})$).

For the set of cardinality five, again we generate all possible 3-clauses of negative literals (10 clauses this time), meaning that from this set also at most two variables can be *true*. In total, therefore, at most $2(g-1)+2 = 2g = (n-1)/2$ variables can be true.

Now we repeat the process, partitioning the variables into a different collection of $(g-1)$ sets of cardinality four and one of cardinality five and again for each set of variables generating all possible 3-clauses, except this time, all the literals used are positive. This second collection of clauses guarantees that, at most, $(n-1)/2$ variables can be false. Taking these two sets of clauses together it can be seen that it is not possible to assign a value to every variable. At most, $(n-1)/2$ can be *true* and also, at most, $(n-1)/2$ can be false, giving a total of $(n-1)/2 + (n-1)/2 = n-1$ variables that can be assigned values. Thus,

Table I. Example Instances

Unsatisfiable (9 variables) `sgen1-unsat-9-1`	Satisfiable (10 variables) `sgen1-sat-10-1`
p cnf 9 28	p cnf 10 24
−2 −3 −4 0	−1 −2 0
−1 −3 −4 0	−1 −3 0
−1 −2 −4 0	−1 −4 0
−1 −2 −3 0	−1 −5 0
−5 −6 −7 0	−2 −3 0
−5 −6 −8 0	−2 −4 0
−5 −6 −9 0	−2 −5 0
−5 −7 −8 0	−3 −4 0
−5 −7 −9 0	−3 −5 0
−5 −8 −9 0	−4 −5 0
−6 −7 −8 0	−6 −7 0
−6 −7 −9 0	−6 −8 0
−6 −8 −9 0	−6 −9 0
−7 −8 −9 0	−6 −10 0
	−7 −8 0
4 2 6 0	−7 −9 0
7 2 6 0	−7 −10 0
7 4 6 0	−8 −9 0
7 4 2 0	−8 −10 0
3 8 5 0	−9 −10 0
3 8 9 0	
3 8 1 0	9 6 3 5 1 0
3 5 9 0	4 2 8 10 7 0
3 5 1 0	
3 9 1 0	7 5 3 9 2 0
8 5 9 0	10 4 8 1 6 0
8 5 1 0	
8 9 1 0	
5 9 1 0	

there is at least one variable that cannot be assigned a value and the generated instance is, therefore, known to be unsatisfiable.

If the number of variables n is $4g+1$, then the number of clauses is $8g+12 = 2n+10$, so as n increases the number of clauses is approximately $2n$. Each clause has three literals, so the number of literals is approximately $6n$.

We denote a formula of this form that has n variables by `sgen1-unsat-`n. Note that this is not unique, as a random seed is used in the simulated annealing process. If we wish to specify the random seed that was used, we add it at the end, giving the name `sgen1-unsat-`n`-`s. As an example, `sgen1-unsat-9-1` is shown in Table I, where the blank lines serve merely to distinguish the partitions for this article and are not part of the formula.

2.3 Generating Satisfiable Formulae

The overall goals in generating difficult satisfiable formulae are still (as in Section 2.2) to have "unrelated" variables and to eliminate many possible models with each clause. This time, however, we have to be careful to leave as few satisfying models as possible (ideally one) and yet not eliminate them all.

Section 1.3 explained how deleting a random clause from the formula `unsat-`n yields a satisfiable formula with only one satisfying model that is similarly difficult for solvers. It is also possible to delete a random clause from a formula of the form `sgen1-unsat-`n and obtain a satisfiable result. However, in this case, there are multiple satisfying models and the resulting formula is not difficult to solve. For example, a typical formula of the form `sgen1-unsat-201` has 1,236 literals and is much too difficult for any solver, whereas deleting a random clause from such a formula yields a satisfiable formula that can be solved by `Minisat` [Eén and Sörensson 2003] in a fraction of a second.

Instead, we use the following approach. First, we use a partition and corresponding clauses to ensure a maximum of $n/5$ true variables overall. Note that these clauses are not the same as those generated for unsatisfiable formulae. Then, we use multiple distinct partitions and yet more corresponding clauses, that each require $n/5$ true values overall but restrict which of the variables these can be.

To achieve this, we first (if necessary) increase the number of variables until it is a multiple of five and then partition the variables into g sets, all having cardinality five. For each set, we generate all possible binary clauses of negative literals (10 clauses for each set). For example, from the set $\{a, b, c, d, e\}$ we would generate

$$(\overline{a} \vee \overline{b}) \wedge (\overline{a} \vee \overline{c}) \wedge (\overline{a} \vee \overline{d}) \wedge (\overline{a} \vee \overline{e}) \wedge (\overline{b} \vee \overline{c}) \wedge (\overline{b} \vee \overline{d}) \wedge \ldots \wedge (\overline{d} \vee \overline{e})$$

This permits at most one true variable per set, since any two being true (e.g., $\{a, b\}$) would refute one of these clauses, (in this case $(\overline{a} \vee \overline{b})$), and so there is a maximum of $g = n/5$ true variables overall.

To ensure that $n/5$ `true` variables are required, we repartition the variables into different sets of cardinality five, and for each set generate one 5-clause of all the positive literals. For example, from the set $\{a, b, c, d, e\}$, we would generate $(a \vee b \vee c \vee d \vee e)$. There are g such clauses and each requires at least one true variable to satisfy it. Thus, there must be at least $g = n/5$ true variables overall. A model containing the $n/5$ true variables permitted by the first partition might be enough to satisfy these clauses if they can be allocated as one true variable per set in the partition.

We repeat this with another partition. Again, it might be possible to allocate one true variable per set. Consider the satisfiable formula on the right in Table I. The first partition is $\{1, 2, 3, 4, 5\}$,$\{6, 7, 8, 9, 10\}$, and choosing the assignment $\{-1, -2, 3, -4, -5, -6, -7, 8\}$, that is, with only 3 and 8 to be positive, satisfies all the binary clauses of negative literals. It can be seen that the clauses from the second and third partitions are also satisfied by this assignment because, in each case, 3 in is one clause and 8 is in the other.

If more random collections of 5-clauses of positive literals are added, the formula becomes harder to solve in absolute terms and also less and less likely to remain satisfiable. However, because this would also increase the number of literals, the difficulty of the formula relative to its size would not necessarily increase.

The compromise that has been determined empirically to give good results is:

—add two partitions of positive literals;
—choose $n/5$ variables in advance and restrict the swaps that can be made during the simulated annealing process to ensure that these are always distributed one per set of variables in the partition. This guarantees at least one satisfying model that has these variables true and all others false.

Thus, it can be seen that in the satisfiable formula in Table I, the variables 3 and 8 occur in the same positions in each of the partitions of positive literals, so it is thereby ensured that $\{-1, -2, 3, -4, -5, -6, -7, 8\}$ is a satisfying model.

The instance will contain $10g$ binary clauses and $2g$ 5-clauses, giving a total of $12g = 12n/5$ clauses and $6n$ literals.

We denote a formula of this form which has n variables by `sgen1-sat-`n or, if the random seed is specified, `sgen1-sat-`n`-`s. As an example, `sgen1-sat-10-1` is shown in Table I.

3. RESULTS

The generator is a single C source file and is available with this article. A short description of how to run the program is given later in the text. The program was submitted to the SAT2009 Competition as a benchmark generator and we analyze the performance of all the solvers on the formulae which it generated, both satisfiable and unsatisfiable. We then give the results of some of our own experiments, which were carried out using a computer running Linux with a 3GHz processor and 2GB of RAM.

3.1 Running `sgen1`

The mandatory parameters are:

-sat or -unsat. Indicates whether a satisfiable or unsatisfiable formula is to be generated.

-n integer-value. Specifies the minimum number of variables to be created. As explained previously, the generator may increase this value slightly.

The optional parameters are:

-s integer-value. Specifies a seed for the random number generator. Repeatedly using the same seed generates the same formula. If not specified, the value used is 1.

-m string-value. Specifies the name of a file into which a satisfying model will be written. This is applicable only when a satisfiable formula is being generated.

-min-variables. Specifies that a formula with a minimum number of variables is to be generated, that is, of the form `unsat-`n described in Section 1.3, or the corresponding satisfiable formula with a random clause deleted. The default is to minimize the number of literals, that is, generate a formula of the form `sgen1-unsat-`n or `sgen1-sat-`n

The generated formula is written to standard output, for example,

```
sgen1 -unsat -n 70 -s 100 >sgen1-unsat-70-100.cnf
```

Table I gives an example of a 9-variable unsatisfiable formula and a 10-variable satisfiable one. In the unsatisfiable formula, the second partition (used for the positive literals) is {2, 4, 6, 7}, {1, 3, 5, 8, 9}. In the satisfiable formula, the second and third partitions are {1, 3, 5, 6, 9}, {2, 4, 7, 8, 10}, and {2, 3, 5, 7, 9}, {1, 4, 6, 8, 10}, respectively.

3.2 SAT2009 Competition

The SAT Competition has been running since 2002 in conjunction with the annual conference on the Theory and Applications of Satisfiability Testing [Hoos 2009]. There have been six competitions to date, and in addition, there have been two related SAT Races [Sinz 2008].

Unsatisfiable and satisfiable formulae generated by `sgen1` were among those used as benchmarks in the SAT2009 competition (so a brief description has already appeared on the competition Web site). The competition was in two phases, with CPU time limits of 1200 and 5000 seconds respectively, and only the best solvers in each of the three categories (Crafted, Random, and Application) qualified for the second phase. The solvers which generally demonstrated the best performance on the `sgen1` formulae in the first phase (the author's `tts-5-0` [Spence 2008] for unsatisfiable formulae, `gnovelty+2` [Pham et al. 2007] for satisfiable) did not qualify to be run on them in the second phase. In general, incomplete solvers (e.g., `gnovelty+2`) had much better performance on the satisfiable formulae than complete solvers had.

The formula `sgen1-unsat-121-100` with 756 literals (121 variables) was the shortest formula not solved during the competition. `sgen1-sat-230-100`, with 1,380 literals, (230 variables) was the shortest known satisfiable formula not solved in the second phase, and `sgen1-sat-300-100`, with 1,800 literals (300 variables), was the shortest known satisfiable formula not solved in the first phase. This surprising result, that the formula for the second phase was shorter than for the first phase, can be explained by the fact that the incomplete solvers were not run on `sgen1` formulae during the second phase.

So, for consistency, only the first phase results are considered here, which means that the times for every solver against every formula are included. Figure 1 shows the smallest execution time of any solver against each of the `sgen1` unsatisfiable benchmarks, and Figure 2 shows the corresponding times for satisfiable formulae. All the results are plotted up to a timeout of 1,200 seconds. It can be seen that the unsatisfiable times increase steadily, whereas the satisfiable ones are more irregular. In both cases, the largest formula (756 literals for unsatisfiable, 1,800 for satisfiable) was not solved by any solver within the 1,200 seconds limit. For comparison, the smallest unsolved unsatisfiable formula from any other source was `hgen8-n320-01-S1432474240.sat05-483` with 1,102 literals. There were only three other unsatisfiable formulae with fewer then 800 literals (`instance_n2_i2_pp`, `mod2-3cage-unsat-9-12.sat05-2587.reshuffled-07`, and

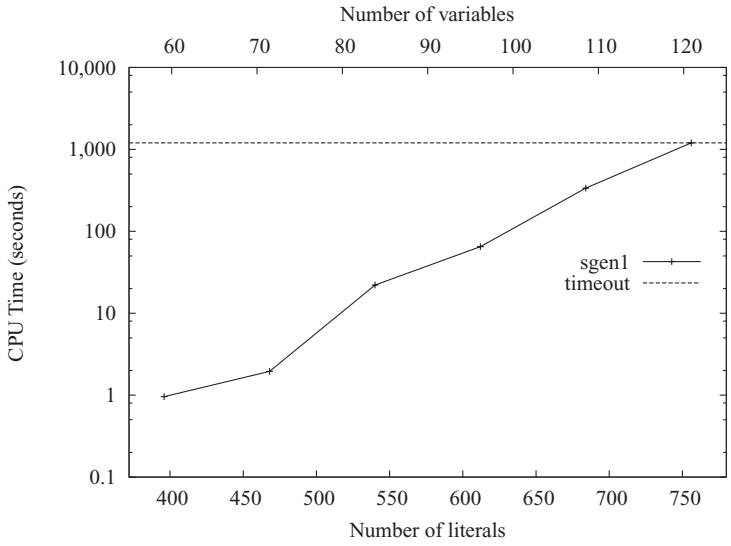

Fig. 1. SAT2009 Competition. Smallest execution times for sgen1 UNSAT formulae.

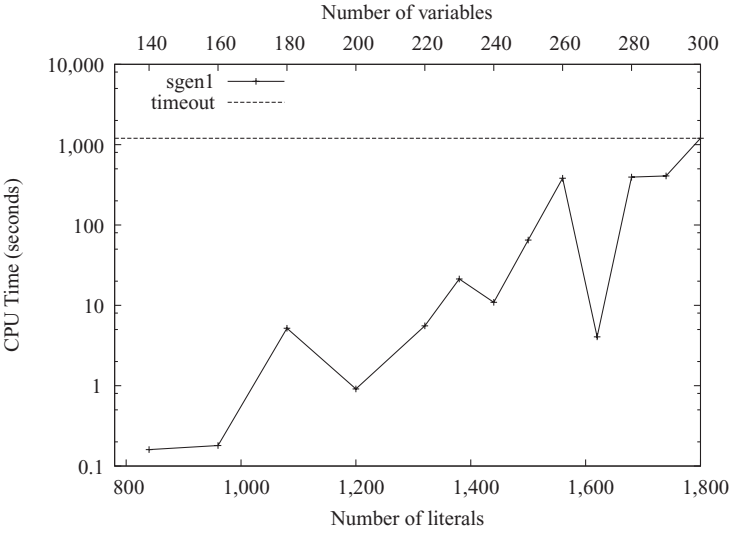

Fig. 2. SAT2009 Competition. Smallest execution times for sgen1 SAT formulae.

`mod2-3cage-unsat-9-14.sat05-2589.reshuffled-07`), and these were all solved within 0.01 seconds. There were no other satisfiable formulae submitted within the "Number of literals" range of Figure 2. The next smallest had 2,520 literals and was solved within 0.01 seconds.

It could be argued that the `hgen8` generator was not shown to its best advantage in the competition because the smallest formula included had 1,102 literals and was not solved, so we conducted our own experiments to compare smaller `hgen8` formula with `sgen1`. Also, `hgen2` was the only other generator

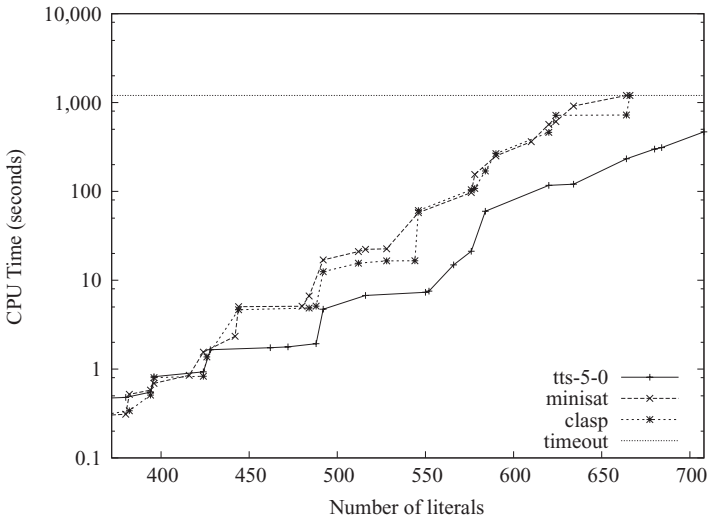

Fig. 3. Greatest execution times for hgen8 benchmarks—various solvers.

of satisfiable formula known to be even comparable (it had previously generated the smallest satisfiable formula in the SAT2002 competition), so it was considered in more detail as well.

3.3 Additional Experiments

Having seen that the formula used in the competition did not really give a good comparison between `hgen8` and `sgen1`, we conducted further experiments to compare the most difficult formulae that each of them could generate. Like `sgen1`, `hgen8` takes a random seed as a parameter. As different random seeds were used, the time taken by the solver varied, even for formulae of the same size. For each value of n within the given range, a formula was generated for a range of random seed values. We are interested in the most difficult formulae generated, so we plot the greatest execution time over all the formula for a given number of literals. Also, for clarity, we only plot monotonically increasing values. If F is the set of formulae generated, then for $f \in F$, we write $L(f)$ for the number of literals in f and $T(f)$ for the time taken to solve f. The set of points plotted is then

$$\{(L(f), T(f))\} \bullet f \in F \wedge (\nexists f' \in F \bullet L(f') \leq L(f) \wedge T(f') > T(f))$$

In principle, we could have run all the solvers on all these formulae, but as shown by both our previous results [Spence 2008] and those of the SAT 2009 Competition, for the formulae generated by `hgen8` and `sgen1`, the beste solver is `tts-5-0`. For example, see Figures 3 and 4, which plot the results for the solvers `tts-5-0`, `Minisat` [Eén and Sörensson 2003] and `clasp` [clasp 2009] on `hgen8` and `sgen1` formulae, respectively. `clasp` has been chosen as an example because it was the winner of two of the SAT2009 Competition gold medals in the Crafted category, and `Minisat` has been chosen because it is so well known. For each value of n, the random seeds used were $\{1, 11, 21, \ldots, 191\}$.

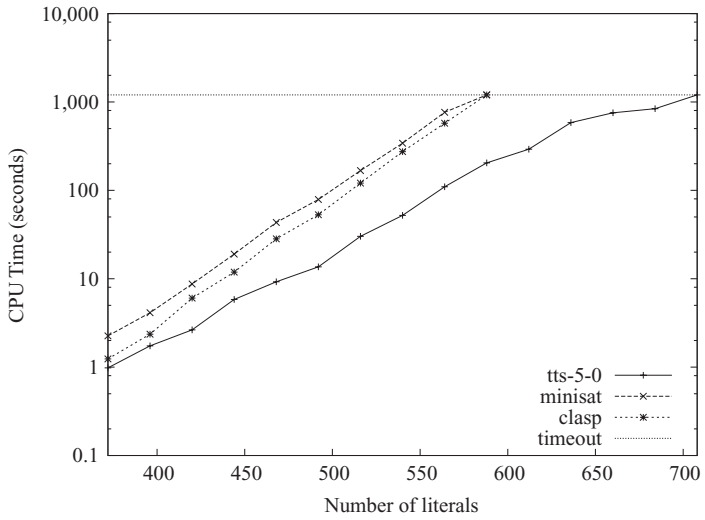

Fig. 4. Greatest execution times for `sgen1` benchmarks—various solvers.

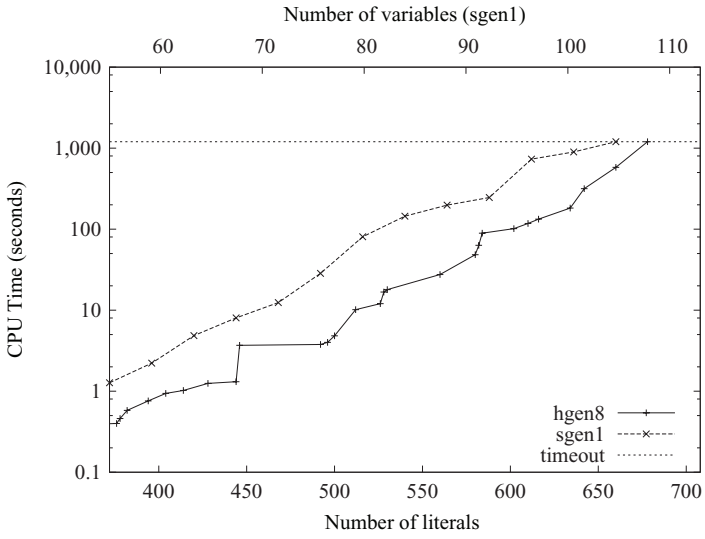

Fig. 5. Greatest execution times for `hgen/sgen1` benchmarks.

It can be seen that for both collections of formulae `tts-5-0` is the fastest solver.

Therefore, for the main comparison, we ran just `tts-5-0` on a large number of formulae generated by `hgen8` and `sgen1`. For each value of n, the random seeds used were $\{1, \ldots, 200\}$. It should be noted that `tts-5-0` is not a good general-purpose solver—it is only competitive on formulae such as these that other solvers find difficult.

The results are showne in Figure 5. Note that the "Number of variables" axis refers only to `sgen1`; for a given number of literals, the formulae from different

sources will have different numbers of variables. It can be seen that the execution times for formulae generated by `sgen1` are greater than the corresponding ones for `hgen8`. It should perhaps be noted that for other solvers that take longer on both `hgen` and `sgen1` formula, the difference is even more marked.

The smallest satisfiable formulae that can be generated by `hgen2` have 2,625 literals (250 variables). They can be solved in less than 1 second by, for example, the incomplete solver adaptg2sat0 [Li and Huang 2005], so there was no need of any further comparison between `sgen1` and `hgen2`.

4. CONCLUSIONS

Using the results of the SAT 2009 competition and also the results of our own experiments, we have demonstrated that `sgen1` generates the most difficult formulae (both unsatisfiable and satisfiable) for state-of-the-art solvers. The increase in difficulty for unsatisfiable formula has been incremental (approximately a factor of 2-3) but that for satisfiable formula has been greater. It has been particularly striking that we have produced satisfiable formulae that all known incomplete solvers find difficult.

SAT-solver performance is notoriously difficult to predict, and there is always the possibility that a new one can be written specially for solving a particular series of formulae. It could well be possible to check for the specific formulae generated here and so misleadingly give small execution times. However, for the current generation of solvers, the formulae generated are the most difficult known.

We finish by comparing the number of literals in the smallest unsolved formulae of the SAT2009 Competition with the corresponding ones from the first SAT Competition in 2002.

	SAT2002	SAT2009
UNSAT	844	756
SAT	5,250	1,800

The fact that the unsolved formulae are now smaller suggests that perhaps formula generators have more than kept up with solvers.

ACKNOWLEDGMENTS

We would like to thank Daniel le Berre and Olivier Roussel, the organizers of the SAT 2009 Competition. We would also like to thank the reviewers of the first draft of this article for their helpful comments.

REFERENCES

ALOUL, F. A., RAMANI, A., MARKOV, I., AND SAKALLAH, K. 2003. Solving difficult instances of Boolean satisfiability in the presence of symmetry. *IEEE Trans. CAD 22*, 9, 1117–1137.

BATORY, D. 2005. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Conference on Software Product Lines*. Springer, Berlin, 7–20.

CLASP. 2009. A conflict-driven no good learning answer set solver. http://www.cs.uni-potsdam.de/clasp/.

COOK, S. A. 1971. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*. ACM, New York, 151–158.

CRAWFORD, J. M., KEARNS, M. J., AND SHAPIRE, R. E. 1994. The minimal disagreement parity problem as a hard satisfiability problem. Tech. rep., Computational Intelligence Research Laboratory and AT&T Bell Labs.

DAVIS, M., LOGEMANN, G., AND LOVELAND, D. 1962. A machine program for theorem-proving. *Commun. ACM 5*, 7, 394–397.

DAVIS, M. AND PUTNAM, H. 1960. A computing procedure for quantification theory. *J. ACM 7*, 3, 201–215.

DIMACS. 1993. Satisfiability suggested format. ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/ doc/satformat.tex.

EÉN, N. AND SÖRENSSON, N. 2003. An extensible sat-solver. In *Proceedings of the 6th Annual Conference on Theory and Applications of Satisfiability Testing*. Springer, Berlin, 502–518.

FORTNOW, L. 2009. The status of the P versus NP problem. *Commun. ACM 52*, 9, 78–86.

HAANPÄÄ, H., JÄRVISALO, M., KASKI, P., AND NIEMELÄ, I. 2006. Hard satisfiable clause sets for benchmarking equivalence reasoning techniques. *J. Satisfiability Boolean Model. Comput. 2*, 1-4, 27–46.

HAIXIA, J., MOORE, C., AND SELMAN, B. 2005. From spin glasses to hard satisfiable formulas. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*. Springer, Berlin, 199–210.

HIRSCH, E. 2002. Random generator hgen2 of satisfiable formulas in 3-CNF. http://logic.pdmi.ras.ru/∼hirsch/benchmarks/.

HOOS, H. H. 2009. The international conferences on theory and applications of satisfiability testing (sat). http://www.satisfiability.org.

KIRKPATRICK, S., GELATT, C. D., AND VECCHI, M. P. 1983. Optimization by simulated annealing. *Science 220*, 4598, 671–680.

KRISHNAMACHARI, B., XI XIE, A. B. S., AND WICKER, S. 2000. Analysis of random noise and random walk algorithms for satisfiability testing. In *Proceedings of the 6th Annual Conference on Principles and Practice of Constraint Programming*. Springer, Berlin, 278–290.

LE BERRE, D. 2009. The sat competitions. http://www.satcompetition.org.

LI, C. M. AND HUANG, W. Q. 2005. Diversification and determinism in local search for satisfiability. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing*. Springer, Berlin, 158–172.

PHAM, D. N., THORNTON, J. R., GRETTON, C., AND SATTAR, A. 2007. Advances in local search for satisfiability. In *Proceedings of the 20th Australian Joint Conference on Artificial Intelligence*. Springer, Berlin, 213–222.

RINTANEN, J., HELJANKO, K., AND NIEMEL, I. 2006. Planning as satisfiability: Parallel plans and algorithms for plan search. *Artif. Intell. 170*, 12-13, 1031–1080.

SAFARPOUR, S. A., BAECKLER, G., YUAN, R., AND VENERIS, A. 2006. Efficient sat-based Boolean matching for FPGA technology mapping. In *Proceedings 43rd Design Automation Conference*. ACM, New York, 466–471.

SINZ, C. 2008. SAT-Race 2008. http://baldur.iti.uka.de/sat-race-2008.

SMITH, A., VENERIS, A. G., ALI, M. F., AND VIGLAS, A. 2003. Fault diagnosis and logic debugging using Boolean satisfiability. *IEEE Trans. CAD 24*, 10, 1606–1621.

SPENCE, I. 2008. tts: A SAT-solver for small, difficult instances. *J. Satisfiability Boolean Model. Comput. 4*, 173–190.

WHITESITT, J. E. 1995. *Boolean Algebra and Its Applications*. Dover Publications, Mineola, NY.

XU, K., BOUSSEMART, F., HEMERY, F., AND LECOUTRE, C. 2007. Random constraint satisfaction: Easy generation of hard (satisfiable) instances. *Artif. Intell. 171*, 514–534.

ZHANG, L. AND MALIK, S. 2002. The quest for efficient Boolean satisfiability solvers. In *Proceedings of the 14th International Conference on Computer-Aided Verification*. Springer-Verlag, Berlin, 17–36.

Received November 2009; revised November 2009; accepted December 2009