Weakening Cardinality Constraints Creates Harder Satisfiability Benchmarks

Ivor Spence, Queen's University Belfast

For some time the satisfiability formulae which have been the most difficult to solve for their size have been crafted to be unsatisfiable by the use of cardinality constraints. Recent solvers have introduced explicit checking of such constraints rendering previously difficult formulae trivial to solve. A family of unsatisfiable formulae is described which is derived from the `sgen4` family but which cannot be solved using cardinality constraints detection and reasoning alone. These formulae were found to be the most difficult during the SAT2014 competition by a significant margin and include the shortest unsolved benchmark in the competition, `sgen6-1200-5-1.cnf`.

Categories and Subject Descriptors: F.2.2 [**Nonnumerical Algorithms and Problems**]: Complexity of proof procedures

General Terms: Satisfiability, Benchmarks

Additional Key Words and Phrases: SAT-solvers, Small Hard Benchmarks

ACM Reference Format:
Ivor Spence, 2014. Weakening Cardinality Constraints for Harder Satisfiability Benchmarks. V, N, Article A (January YYYY), 15 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

The boolean satisfiability problem is to determine whether it is possible to assign truth values to the Boolean variables in a propositional expression (formula) in such a way that the overall expression has the value `true`. The theoretical and practical importance of this problem mean that much work has gone into developing effective solvers and every year an international competition [Järvisalo et al. 2012] is held in conjunction with the annual conference on the Theory and Applications of Satisfiability Testing [1]. There are frequently significant improvements in solver technology from one competition to the next, and there are formulae with millions of literals which can be solved within the allowed CPU time which was 5,000 seconds in 2014.

However this decision problem is NP-Complete [Cook 1971], inspiring the search for small difficult formula which the best solvers find exponentially hard, and we focus in this paper on formulae with fewer than 1500 literals which were not solved in the 2014 competition. The most difficult formulae for their size have always been unsatisfiable and to date have relied on cardinality constraints to ensure that they cannot be satisfied. For example the formulae in the pigeon-hole series [Haken 1984] encode the allocation of n pigeons into $n-1$ pigeon-holes with no more than one pigeon per hole,

[1] http://www.satisfiability.org/

Author's address: I. Spence, School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, University Road, Belfast BT7 1NN United Kingdom

DOI:http://dx.doi.org/10.1145/0000000.0000000

which is of course impossible. Related series include Hirsch's hgen [Hirsch 2002] and in recent years (including 2011) the most difficult formulae for their size at the SAT competitions [Järvisalo et al. 2012] have come from the `sgen` [Spence 2010] family of generators.

Some solver developers [Ostrowski et al. 2002; Li 2000; Warners and Maaren 1998] have targeted particular classes of formula which has enabled very efficient solving of formulae previously found to be difficult. In particular analysis using cardinality constraints [Biere et al. 2014] means that solvers which can solve instances of these series up to 1200 literals in less than one second are now available. Motivated by this the approach which was used in the `sgen4` benchmark generator has been modified so that cardinality constraints alone are not sufficient to solve the generated formulae.

The main contributions from this paper are:

- A demonstration that the satisfiability instances which were previously the most difficult are not any more and an explanation of why this is the case.
- A description of a new approach for generating unsatisfiable formulae with an explanation of why these should be more difficult.
- The presentation of results from the SAT 2014 competition and further experiments indicating that that the new formulae are now by a significant margin the most difficult known, and suggesting that they are exponentially difficult for even the best solvers.

We first describe briefly the satisfiability problem and the structure of the DIMACS standard input format. We explain the reasons for trying to create difficult instances, showing how instances previously found to be difficult were constructed. After explaining how recent solvers are able to solve such instances easily we introduce our new family of formulae. We define our measure of difficulty and present results from the 2014 SAT Competition indicating the success of this approach and finally we present more detailed experimental results which suggest the exponential times required by the best solvers for these benchmarks.

2. THE SATISFIABILITY PROBLEM

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 uses 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. A formula which is used to test the performance of a solver is often called a *benchmark* or an *instance*.

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

2.1. DIMACS Input Format

The standard for representing formulae by text files is the DIMACS format [DIMACS 1993] 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 such a 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
```

3. CARDINALITY CONSTRAINTS

The most effective techniques for creating small difficult formulae have to date involved partitioning variables into small groups and, within each group, generating clauses which impose constraints on the number of variables which can take a particular value. For example, at least one variable must take the value `true`, or at most two must take the value `false` [Spence 2010]. When all the constraints for a formula are considered together they can be inconsistent (for unsatisfiable formula) or at least only permit a small number of solutions (for satisfiable formula) but previous solvers did not explicitly use this information.

3.1. Unsatisfiable Cardinality Constraints

In an unsatisfiable formula generated by `sgen4` the partitions contain one group of five variables with the remaining groups containing four variables. There are clauses which guarantee that at most two variables in each group are `true` meaning that overall at most (N-1)/2 variables can be `true`. A second and similar partition is used to guarantee that at most (N-1)/2 variables can be `false`. Taken together it is only possible to allocate values to N-1 of the variables and so the formula is unsatisfiable. Simulated annealing is used to create the second partition in such a way as to minimise any similarity to the first, so that for example variables which appear in the same group in the first partition are not in the same group in the second. This shuffling reduces the number of times that the same variables appear together in the same clause which increases the difficulty of finding a solution [Spence 2010].

Formulae of this kind have until recently proved to be the most difficult for state-of-the-art solvers, indeed it has been shown [Mikša and Nordström 2014] that they are exponentially difficult for resolution-based approaches and in 2010 a challenge was issued [Van Gelder and Spence 2010] for a particular 1060-literal formula to be solved in less than a day. It is apparent that a formula-specific approach could have been used to recognise this structure and declare unsatisfiability quickly, but the target was not met until 2014. Solvers have now been written [Biere et al. 2014] which incorporate detection and analysis of cardinality constraints (see Section 4.2) meaning that the previously difficult formulae, including the challenge mentioned above, can be solved in a fraction of a second. These solvers are based on cuttings planes which allow the solver to produce a short proof of unsatisfiability [Cook et al. 1987].

3.2. Weaker Cardinality Constraints

In satisfiable formulae generated by `sgen4` weaker constraints are used so that not all possible assignments are immediately precluded. Such a formula is based on three partitions of the variables, each consisting of groups of g variables (for the SAT 2013 competition a value of $g = 5$ was used, see section 4.3). Simulated annealing is again used to minimise any similarity amongst the partitions. The groups in the second and third partitions all intersect the the groups of the first partition in at most one variable, except for very small benchmarks where this is impossible to achieve. The constraints generated for the first partition guarantee that at most p variables from each group are

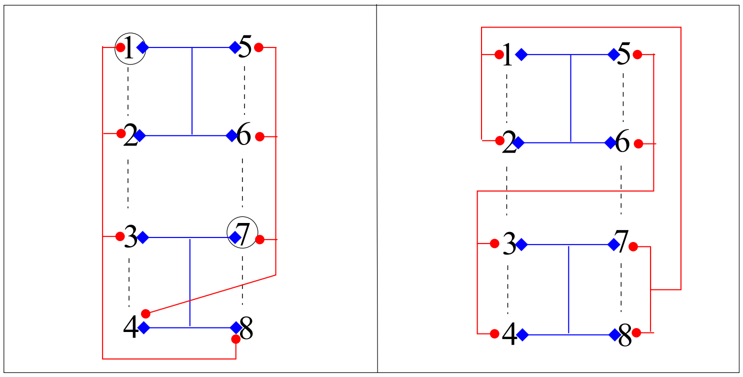

Fig. 1. Constraints corresponding to satisfiable (left) and unsatisfiable formulae

true (for the SAT 2013 competition a value of $p = 1$ was used, see section 4.4), those for each of the other two partitions guarantee that at least p variables from each group are true. Thus potentially there is a solution with exactly np/g variables being true. The question then becomes whether it is possible to identify a set of np/g variables such that, in each of the three partitions, exactly p of the chosen variables occurs in each group. The sgen4 generator generates the partitions in such a way that this is forced to be true by choosing in advance what this set of variables will be and then not shuffling them in the simulated annealing process. Analysing cardinality constraints does not give a quick solution to these formulae, which are amongst the more difficult satisfiable benchmarks, but using incomplete solvers such as sattime, formulae with more than 1000 literals could be solved in the order of 10 seconds in 2011. The cutting planes approach does not help here because there can be no proof of unsatisfiability for a satisfiable formula.

3.3. sgen6 - Unsatisfiability With Weaker Constraints

Finally we describe the improvement which has led to the sgen6 unsatisfiable formulae which cannot be solved quickly even using cardinality constraints. In order to provide the challenge of having an unsatisfiable formula but yet avoid having cardinality constraints which on their own guarantee unsatisfiability, we investigated removing the restriction on partitioning which sgen4 enforced. This means satisfiability of generated formulae is not guaranteed but depends in the random partitioning process. If it is possible to choose a set of np/g variables in such a way that in every partition there are exactly p such variables in each group then the formula is satisfiable, otherwise it is not. As the size of the formula increases it can be seen empirically that unsatisfiability is more likely. For example, with a group size of 5 the probability of a satisfiable formula is approximately 50% at 540 literals. Within the sgen6 tool an exhaustive (computationally expensive) search can be carried out on request to ensure that only unsatisfiable benchmarks are generated and these are the kinds of formula which proved to be so difficult in the 2014 competition.

Figure 1 represents the constraints corresponding to one satisfiable (left) and one unsatisfiable formula in which $g = 4$ and $p = 1$. The vertical black dashed lines represent the at-most-one constraints and it can be seen that in both formulae these constraints refer to the unshuffled partition $\{\{1, 2, 3, 4\}, \{5, 6, 7, 8\}\}$. The solid red (with dots) and blue (with diamonds) lines represent the at-least-one constraints which are applied to shuffled partitions. For each example choosing one positive variable from each of the two elements of the first partition, with the at-most-one constraints, means choosing

Satisfiable with the model {1,-2,-3,-4,-5,-6,7,-8}	Unsatisfiable
p cnf 8 16 -1 -2 0 First partition -1 -3 0 {1,2,3,4},{5,6,7,8} -1 -4 0 "at most one" -2 -3 0 black dashed lines -2 -4 0 -3 -4 0 -5 -6 0 -5 -7 0 -5 -8 0 -6 -7 0 -6 -8 0 -7 -8 0 $\underline{1}$ 2 3 8 0 Second partition 4 5 6 $\underline{7}$ 0 {1,2,3,8},{4,5,6,7} "at least one" red lines with circles $\underline{1}$ 2 5 6 0 Third partition 3 4 $\underline{7}$ 8 0 {1,2,5,6},{3,4,7,8} "at least one" blue lines with diamonds	p cnf 8 16 -1 -2 0 First partition -1 -3 0 {1,2,3,4},{5,6,7,8} -1 -4 0 "at most one" -2 -3 0 black dashed lines -2 -4 0 -3 -4 0 -5 -6 0 -5 -7 0 -5 -8 0 -6 -7 0 -6 -8 0 -7 -8 0 1 2 7 8 0 Second partition 3 4 5 6 0 {1,2,7,8},{3,4,5,6} "at least one" red lines with circles 1 2 5 6 0 Third partition 3 4 7 8 0 {1,2,5,6},{3,4,7,8} "at least one" blue lines with diamonds

Fig. 2. cnf files for the example satisfiable and unsatisfiable formulae

a maximum of two positive variables in total. The second and third partitions both lead to two at-least-one constraints so a minimum of two positive variables have to be chosen. Thus overall there must be exactly two. As can be seen choosing 1 and 7 (circled) satisfies all the constraints in the left diagram. The right diagram however cannot be satisfied. For example choosing 1 means that the second positive variable must come from the column $\{5, 6, 7, 8\}$. Choosing 5 or 6 leaves $\{3, 4, 7, 8\}$ (blue with diamonds) unsatisfied and choosing 7 or 8 leaves $\{3, 4, 5, 6\}$ (red with dots) unsatisfied.

Figure 2 gives the cnf files corresponding to these two sets of constraints. The two formula have the same structure with respect to cardinality constraints and their satisfiability depends on the exact partitions, demonstrating that cardinality detection alone is not sufficient to solve these formulae. We have not proved the impossibility of a short proof of unsatisfiability, but no known solver has been found to take advantage of one.

4. RESULTS

We present published results from the SAT2014 competition, and also results from our own experiments. The platform for SAT2014 was

— Operating System: Linux Centos 5.5
— Processor(s): 2 Hex-core Xeon 5680
— Memory: 24GB

— Timeout: 5,000 seconds

For our experiments the platform was

— Operating System: Ubuntu 13.04
— Processor(s): 2 Quad-core Xeon 5430
— Memory: 8GB
— Timeout: 30,000 seconds

All the programs are sequential.

4.1. Smallest Known Time Metric

In the absence of any absolute measure of difficulty we introduce a technique for comparing the relative difficulty of two formulae by considering for each its size, measured by the number of literals it contains, and the shortest time to solution of any known solver applied to that formula (the execution platform is implicit). This is also known as the execution time of the Virtual Best Solver (VBS) [Xu et al. 2012]. Note that we do not insist on the same solver being used in both cases - we are looking for formulae which are resistant to solution by any solver. We define a partial order which indicates whether it is possible to compare the difficulty of two given benchmarks. Clearly the value depends on the set of solvers used and for any given formula the inclusion of a new solver can decrease but cannot increase the shortest time to solution.

Suppose that S is a set of solvers and F is a set of formulae. Given $s \in S, f \in F$, we denote by $t(s, f)$ the execution time of the solver s applied to the formula f. We define $\mathrm{vbs}_S(f) = \min_{s\in S} t(s, f)$, i.e. the smallest time of any solver $s \in S$ when applied to f. We are interested in the difficulty of formulae in terms of their length, so we denote the number of literals in a formula f by $\mathrm{lits}(f)$.

We then define a partial order $\preceq_S$ on formulae which is intended to formalise the notion of one formula being less difficult than another with respect to the set of solvers S. If two formula are the same size, then the one with smaller VBS execution time is said to be easier. If the formulae have the same VBS execution time then the one containing more literals is said to be easier. If the longer of two formulae has the smaller VBS time then it is clearly easier. Finally if the longer formula also has a greater VBS time then we cannot compare the difficulty of the two formulae.

Formally the relation $f_1 \preceq_S f_2$ indicates that f_1 is easier then f_2 (or equally diffficult) where

$$f_1 \preceq_S f_2 = \{(f_1, f_2) \mid \mathrm{lits}(f_1) \geq \mathrm{lits}(f_2) \wedge \mathrm{vbs}_S(f_1) \leq \mathrm{vbs}_S(f_2)\}$$

$$f_1 =_S f_2 = \{(f_1, f_2) \mid \mathrm{lits}(f_1) = \mathrm{lits}(f_2) \wedge \mathrm{vbs}_S(f_1) = \mathrm{vbs}_S(f_2)\}$$

For this relation to be useful the set of solvers should be as large as possible, or should at least include the best known solvers for the formulae in question. In this paper we use the set of solvers submitted to the SAT Competition 2014 together with a small number of others which have been found to be particularly effective on small difficult benchmarks. No other solvers are known whose inclusion would materially alter the conclusions.

Figure 3 illustrates the different possibilities when we attempt to compare two formulae. Each part of this figure represents a scatterplot of VBS time against number of literals. The first two sketches are hypothetical and the third illustrates the situation for the four sample formulae listed in Table I with data from the SAT Competition 2014. Formulae A,B and D are unsatisfiable and formula C is satisfiable. In the first sketch f_1 is easier than any of f_2, f_3, f_4. It is longer than f_2 for the same execution

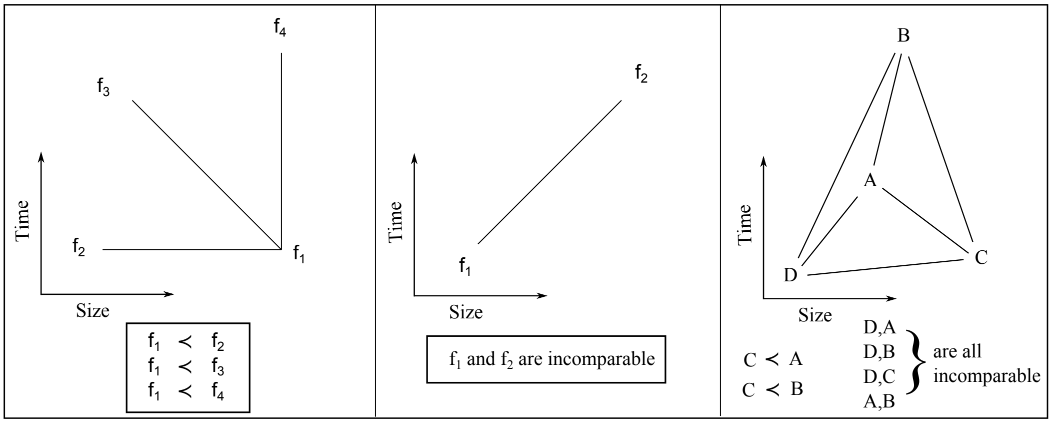

Fig. 3. Relative difficulty of two formulae

Table I. Example results

Id	Formula	Lits	Best Solver	VBS Time(s)
A	sgen6-840-5-1	840	BFS-Glucose_mem_8_85	279
B	sgen6-960-5-1	960	glueSplit_clasp	692
C	sgen4-sat-220-8	1320	RSeq2014	11.1
D	edges-024-4-5667555-1-00	612	Lingeling	0.004

time, takes less time than f_4 for the same number of literals, and is both longer and faster than f_3. In the second sketch f_1 is shorter than f_2 but also takes less time and so their difficulties cannot be compared. In the third sketch it can seen that for example that edges-024-4-5667555-1-00 has a very short minimum execution time (0.004 seconds) but because it is also smaller than the other formula we cannot infer that it is easier. By contrast since sgen4-sat-220-8 (C) is larger than sgen6-960-5-1 (A) as well have having a shorter execution time we say that sgen4-sat-220-8 is easier than sgen6-960-5-1.

4.2. Solver incorporating cardinality detection

First we demonstrate that the use of a solver incorporating cardinality detection can quickly solve formulae previously found to be very difficult. sgen4 formulae were used in the SAT 2014 Competition but pigeon-hole and hgen8 were not, so we ran the solver sat4j-detectcards [Biere et al. 2014] on a collection of pigeon-hole, hgen8 and sgen4 formulae from 500 to 1200 literals. Figure 4 shows the execution times for this solver for the two pigeon-hole formulae within this range and for 20 randomly generated hgen8 and sgen4 formulae for each of a range of requested numbers of literals. Note that hgen8 does not generate formula with exactly a given number of literals, hence the horizontal blurring of each cluster of values. It can be seen that in every case the execution time was less than one second. By contrast this solver takes more than 10,000 seconds to solve a 540 literal sgen6 formula.

4.3. Group Size

We now address the issue of determining the optimum values for the size of the groups of variables (g) and by the number of variables within each group which can and must be positive (p). The values $g = 5$ and $p = 1$ had been found empirically to be the best for the satisfiable benchmarks in sgen4 and it was anticipated that these values

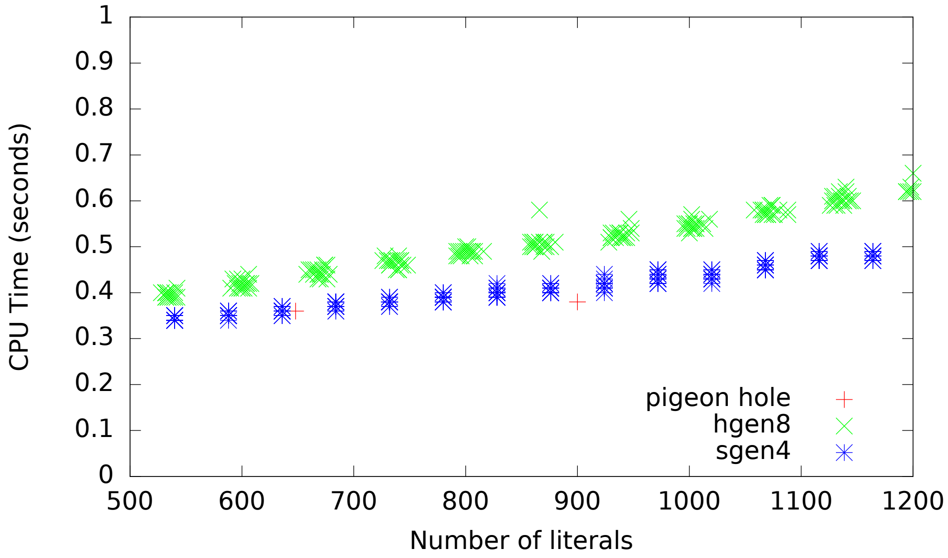

Fig. 4. Performance of sat4j-detectcards on previously difficult instances

would also be best for the unsatisfiable benchmarks of sgen6. For a group of size g, of which p variables are positive, there will be $\frac{g!}{p!(g-p)!} = \binom{g}{p}$ possible ways of assigning these positive variables. It might be anticipated that, for a given number of groups, the solver execution time will increase as $\binom{g}{p}$ increases. The experiments described in sections 4.3 and 4.4 use the solver glueSplit_clasp. If p is fixed at for example 1, then $\binom{g}{p} = g$ and execution time might be expected to increase with group size. Figure 5, where each candlestick shows the minimum, lower quartile, upper quartile and maximum execution times, plots glueSplit_clasp against 20 sgen6 benchmarks for each group size from 2 to 7 with 20 groups and with 30 groups. It can be seen that execution time does broadly increase with group size.

In addition to increasing the execution time for a fixed number of groups however, increasing the value of g also increases the total number of literals. Given that we are interested in execution time for given number of literals, we must take this into count. Instead of fixing the number of groups we must fix the number of literals (at least approximately). Figure 6 shows execution times for glueSplit_clasp against 20 benchmarks for each group size from 2 to 7 with limits of approximately 700 and 900 literals. Note that the number of literals is not fixed exactly because of the constraint imposed by the relationship amongst g, p, number of groups and number of literals.

It can be seen that a group size of 5, as used previously, does provide the greatest execution times for a given number of literals. It offers the best combination of providing sufficient complexity per group without requiring too many literals per group.

4.4. Positive Variables per Group

If g is fixed at for example 5 then $\binom{g}{1} = 5$, $\binom{g}{2} = 10$ and $\binom{g}{3} = 10$ and execution time might be expected to increase at least as p increases from 1 to 2. Figure 7 shows the execution times for glueSplit_clasp against 20 benchmarks with $g = 5$ and $p = 1, 2, 3$ with 20 groups and with 30 groups. It can be seen in practice that, even with a fixed number of groups, the execution time does not increase with p. Given that the number

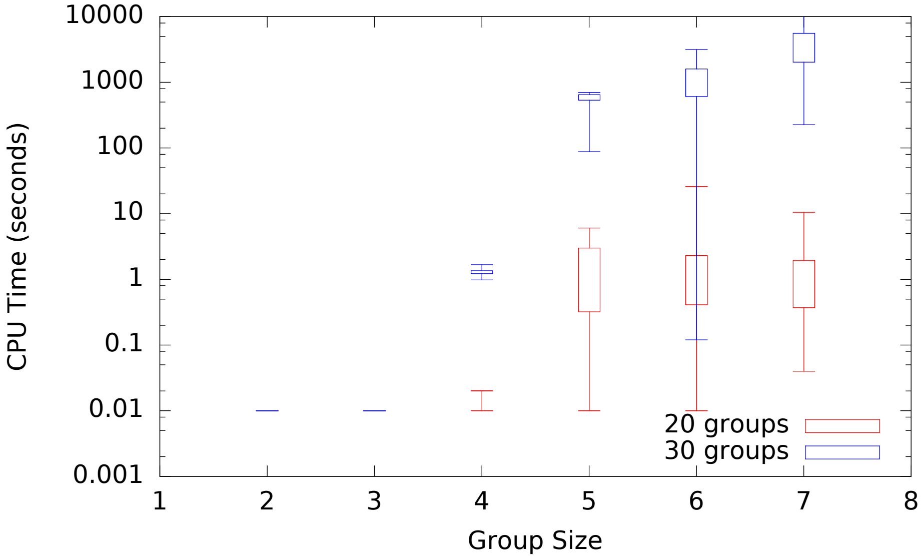

Fig. 5. Execution time against group size for fixed number of groups

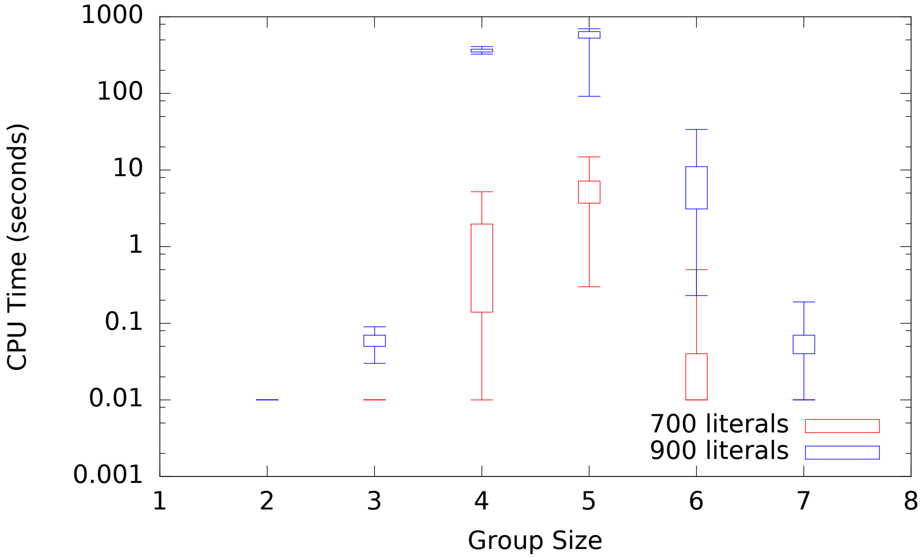

Fig. 6. Execution time against group size for fixed number of literals

of literals <u>does</u> increase with p it is clear that using $p = 1$ gives the best results. It should be noted that with $p > 1$ it was not possible to generate any unsatisfiable benchmarks and so the results reported are for satisfiable ones. It is possible that an incomplete solver could give even smaller execution times when $p > 1$ but this would only confirm the result that $p = 1$ is the best choice.

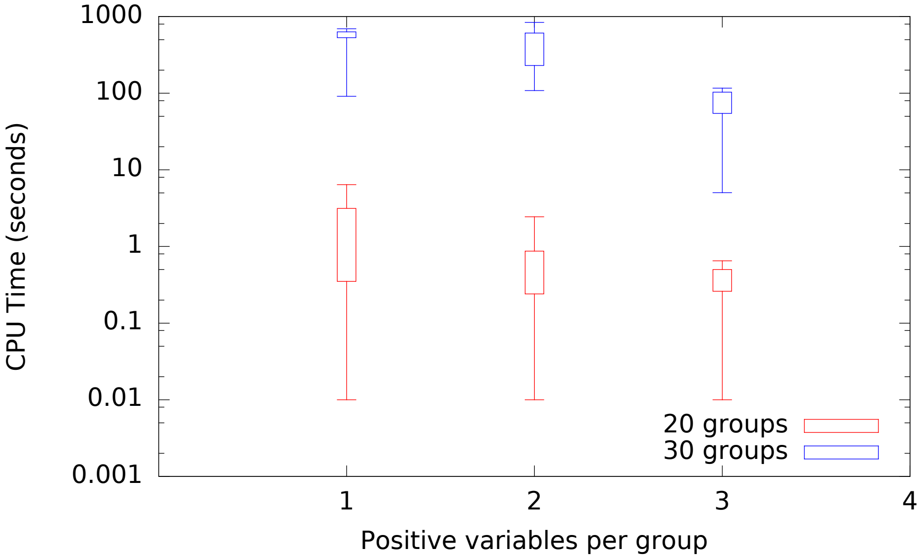

Fig. 7. Execution time against positive variables per group for fixed number of groups

4.5. SAT2014 Competition

Now we present the results obtained by submitting our formulae to be used as benchmarks in the SAT 2014 Competition. As in previous years, the competition was divided into three sections based on the origin of the corresponding benchmarks:

Application.
These benchmarks reflect the expression of a real-world problem as a satisfiability instance. They are typically large but can sometimes be surprisingly easy for solvers.

Random.
These benchmarks are randomly generated but typically with carefully controlled ratios of numbers of clauses to numbers of literals.

Hard-combinatorial (crafted).
These are typically the smallest benchmarks and are constructed purely in order to be difficult to solve or demonstrate some principle. The `sgen6` benchmarks were included in this category.

The benchmarks in the Application and Random categories were all the longer than the 500-1200 literal range of interest here, so only the Crafted results are considered. There was a limit of 5,000 seconds on the CPU time permitted for any solver and the results are available via the competition website [Belov et al. 2014].

Figure 8 displays the results from the Hard-combinatorial SAT+UNSAT track in a form suitable for evaluating the relation $\preceq_{SC}$ where SC is the set of solvers used in the competition. Each point on the plot gives the size (number of literals) of a formula and the VBS execution time for that formula over all the solvers in the competition. If a formula f_1 appears below and to the right of f_2 in Figure 8 then $f_1 \preceq_{SC} f_2$, i.e. f_1 is easier than f_2. To preserve legibility both axes use logarithmic scales and execution times of less than 0.1 seconds are recorded as 0.1 (which improves the case for formulae other than `sgen6`).

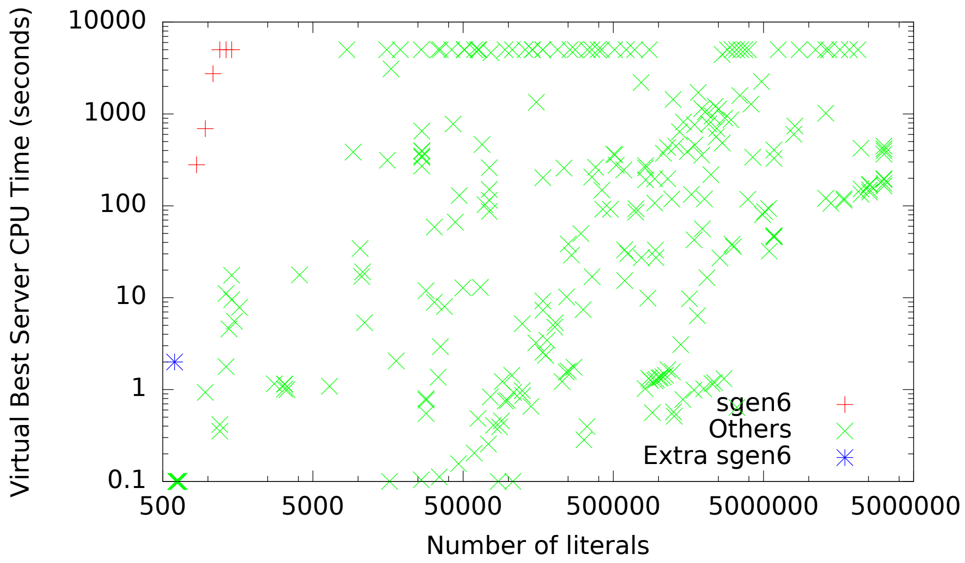

Fig. 8. Minimum Times for Solution of Crafted Benchmarks - SAT2014

One additional point, labelled `Extra sgen6` has been included. The smallest `sgen6` formula submitted to the competition was 840 literals and so could not be compared with the `Z1` formulae which were of size between 600 and 700 literals with minimum execution times less than 0.01 seconds. We want every other formula to be comparable with at least one `sgen6` formula so on our own platform we ran `glueSplit_clasp` (the best solver for `sgen6`) on a 600-literal `sgen6` formula and found the execution time to be 3.93 seconds. Comparison of other results suggests that execution times on our computer are approximately twice those of the competition environment so we included the point (600 literals ,2 seconds). Clearly this value is not particularly accurate but it is convincingly more than than the 0.004 seconds of the `Z1` formulae.

From the SAT Competition results the solvers which give the best results for the `sgen6` formulae were `glueSplit_clasp` and `lingeling` so we focused our final experiments on these solvers. `glueSplit_clasp` incorporates `clasp` so we included that as well.

4.6. Further Results

For every multiple of 60 from 540 to 1200 literals we generated 20 `sgen6` formulae with $g = 5, p = 1$, and ran each of these three solvers on the formulae with a timeout of 30,000 seconds. The results are shown in Figure 9. It can be seen that `glueSplit_clasp` gives the best results, with a complexity which appears to be of order $5^{n/120}$.

5. ANALYSIS

We consider first Figure 8 which shows the Virtual Best Server CPU time plotted against size in literals for all the entries in the Hard Combinatorial track of the SAT Competition 2014, together with a single added point for `sgen6` as explained above. The largest Time value which appears corresponds to the competition time-out of 5,000 seconds. On this chart the gradient of the lining joining two points can be used to determine the relative difficulties of the corresponding formulae. If the gradient is

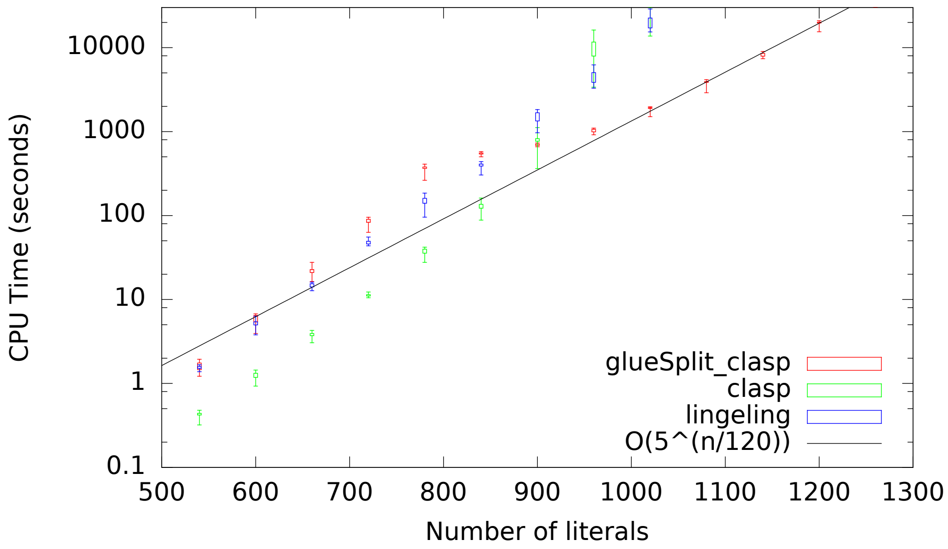

Fig. 9. Minimum execution times for sgen6 formulae

positive the difficulties cannot be compared, whereas if the gradient is negative, zero or infinite (vertical) then the formula appearing to the left/higher is the more difficult.

From this plot it can be seen that there is no formula which is harder than (i.e to left and above) any sgen6 formula. That is if F_s is the set of sgen6 formulae and F_o is the set of all other formulae in the competition,

$$\forall f_s \in F_s, f_o \in F_o \mid f_s \not\preceq f_o$$

It can also be seen that for every formula which is not from sgen6 there is a sgen6 formula which is harder than it. That is

$$\forall f_o \in F_o \exists f_s \in F_s \mid f_o \preceq f_s$$

There is a caveat to these comments which is that if two formulae both result in a vbs timeout value then there is not enough information to compare them (which is not the same as saying that they are incomparable). However if only one formula results in a timeout the comparison is valid.

Both axes of Figure 8 use logarithmic scales and it can be seen that the sgen6 formulae are not just the most difficult but significantly so. Their vbs times for formula of even nearly comparable size are ten times greater, and other formula whose vbs times are similar are nearly ten times as long.

Figure 9 demonstrates the increase in execution time with formula size of the three best solvers for sgen6 formulae. For 1000 literals and over the best solver is glueSplit_clasp and its complexity appears to be $O(5^{n/120})$ where n is the number of literals. Given that the addition of a group of 5 variables increases the number of literals by 30, this means that the addition of four groups of 5 variables multiplies the execution time by 5.

6. CONCLUSIONS

On the basis of the results of the SAT 2014 competition and confirmed by our own experiments we have demonstrated that sgen6 generates the most difficult unsatisfiable formulae for state-of-the-art solvers. In particular solvers which use cardinality constraint detection and are able quickly to solve what were previously regarded as the most difficult formulae cannot quickly solve sgen6 formulae. For an unsatisfiable formula generated by sgen4 the relaxation of cardinality constraints to real-valued variables yields an unsatisfiable problem whereas sgen6 formulae relax to problems which can be satisfied by the assignment of 1/g to each variable. Therefore no solver for the real-valued relaxation, including one using linear programming, can solve these formulae. The increase in difficulty for unsatisfiable formulae is significant, with execution times for the smallest formulae at least ten times those for other formulae of comparable length.

There remains the possibility that a new solver could be written which is targeted specifically for these formulae, but (unlike the situation previously) it is not apparent how this could be done. There are both satisfiable and unsatisfiable sgen6 formulae with similar cardinality structures and it would appear that this is what increases the difficulty so much.

ACKNOWLEDGMENTS

The author would like to thank the organisers of the SAT Competition 2014 and the anonymous referees who made helpful comments on earlier versions of this paper.

REFERENCES

Anton Belov, Daniel Diepold, Marijn Heule, and Matti Järvisalo. 2014. SAT Competition Website. http://www.satcompetition.org/2014/, accessed 2nd Dec. 2014. (2014).

Armin Biere, Daniel Le Berre, Emmanuel Lonca, and Norbert Manthey. 2014. Detecting Cardinality Constraints in CNF. In *Theory and Applications of Satisfiability Testing SAT 2014*, Carsten Sinz and Uwe Egly (Eds.). Lecture Notes in Computer Science, Vol. 8561. Springer International Publishing, 285–301. DOI:http://dx.doi.org/10.1007/978-3-319-09284-3_22

Stephen A. Cook. 1971. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*. ACM, New York, NY, USA, 151–158. DOI:http://dx.doi.org/10.1145/800157.805047

W. Cook, C.R. Coullard, and Gy. Turn. 1987. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics* 18, 1 (1987), 25 – 38. DOI:http://dx.doi.org/10.1016/0166-218X(87)90039-4

DIMACS. 1993. Satisfiability Suggested Format. ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/satformat.tex, accessed 15th October 2009. (1993).

A. Haken. 1984. *The Intractability of Resolution*. University of Illinois at Urbana-Champaign.

Edward Hirsch. 2002. Random generator hgen2 of satisfiable formulas in 3-CNF. http://logic.pdmi.ras.ru/~hirsch/benchmarks/, accessed 22nd Oct. 2009. (2002).

Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. 2012. The International SAT Solver Competitions. *AI Magazine* 33, 1 (2012), 89–92.

Chu Min Li. 2000. Integrating Equivalency Reasoning into Davis-Putnam Procedure. In *The Seventeenth National Conference on Artificial Intelligence - AAAI-00*. 291–296.

Mladen Mikša and Jakob Nordström. 2014. Long Proofs of (Seemingly) Simple Formulas. In *Theory and Applications of Satisfiability Testing SAT 2014*, Carsten Sinz and Uwe Egly (Eds.). Lecture Notes in Computer Science, Vol. 8561. Springer International Publishing, 121–137. DOI:http://dx.doi.org/10.1007/978-3-319-09284-3_10

Richard Ostrowski, Éric Grégoire, Bertrand Mazure, and Lakhdar Sais. 2002. Recovering and Exploiting Structural Knowledge from CNF Formulas. In *Principles and Practice of Constraint Programming - CP2002*, Pascal Van Hentenryck (Ed.). 185–199.

Ivor Spence. 2010. sgen1: A generator of small but difficult satisfiability benchmarks. *ACM Journal of Experimental Algorithmics* 15 (2010). DOI:http://dx.doi.org/10.1145/1671970.1671972

Allen Van Gelder and Ivor Spence. 2010. Zero-One Designs Produce Small Hard SAT Instances. In *Theory and Applications of Satisfiability Testing SAT 2010*, Ofer Strichman and Stefan Szeider (Eds.). Lecture Notes in Computer Science, Vol. 6175. Springer Berlin Heidelberg, 388–397. DOI:http://dx.doi.org/10.1007/978-3-642-14186-7_37

Joost P. Warners and Hans Van Maaren. 1998. A Two Phase Algorithm for Solving a Class of Hard Satisfiability Problems. *Operations Research Letters* 23 (1998), 81–88.

J. Eldon Whitesitt. 1995. *Boolean algebra and its applications*. Dover Publications.

Lin Xu, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. 2012. Evaluating Component Solver Contributions to Portfolio-Based Algorithm Selectors. In *Theory and Applications of Satisfiability Testing SAT 2012*, Alessandro Cimatti and Roberto Sebastiani (Eds.). Lecture Notes in Computer Science, Vol. 7317. Springer Berlin Heidelberg, 228–241. DOI:http://dx.doi.org/10.1007/978-3-642-31612-8_18

APPENDIX

A.1. Running sgen6

The source code `sgen.c` for the generator is available from the JEA library. It can be compiled with the command

```
gcc -o sgen sgen.c -lm
```

and writes the generated formula to standard output so that a typical execution would be

```
./sgen -unsat -lits 600 >sgen6-600.cnf
```

The arguments are:

-version integer-value.
specifies a version number which can be either 4 or 6 and defaults to 6.

-sat | *-unsat* | *-unknown.*
indicates whether a satisfiable, unsatisfiable or unknown formula is to be generated. A satisfiable formula is generated by constraining the permutation process as described above. An unsatisfiable formula is generated by testing and repeated generation using successive seeds. Exactly one of these argument must be given for version 6. For version 4 `unknown` is not valid. -unsat is only supported if -pos-per-group is 1.

-vars integer-value.
specifies the maximum number of variables to be created. For version 6, the number of variables will be a multiple of 5.

-lits integer-value.
specifies the maximum number of literals to be created. For version 6, with default values for group-size and pos-per-group, the number of variables will be a multiple of 30. Exactly one of -lits and -vars must be specified.

-group-size integer-value.
specifies the value of g, the size of the groups of variables. Defaults to 5.

-pos-per-group.
specifies the value of p, the number of variables in each group which must be positive. Defaults to 1.

-s integer-value.
specifies a seed for the random number generator. Repeatedly using the same seed generates the same formula. Defaults to 1.

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

-min-variables.
specifies that a formula with a minimum number of variables is to be generated.

-reorder.
as a final generation step the variables and clauses are randomly permuted.

Received ; revised ; accepted