Evaluating Instance Generators by Configuration

Sam Bayless[1], Dave A. D. Tompkins[2], and Holger H. Hoos[1]

[1] Department of Computer Science
University of British Columbia, Canada
{sbayless,hoos}@cs.ubc.ca

[2] David R. Cheriton School of Computer Science
University of Waterloo, Canada
dtompkins@uwaterloo.ca

Abstract. The propositional satisfiability problem (SAT) is one of the most prominent and widely studied NP-hard problems. The development of SAT solvers, whether it is carried out manually or through the use of automated design tools such as algorithm configurators, depends substantially on the sets of benchmark instances used for performance evaluation. Since the supply of instances from real-world applications of SAT is limited, and artificial instance distributions such as Uniform Random k-SAT are known to have markedly different structure, there has been a long-standing interest in instance generators capable of producing 'realistic' SAT instances that could be used during development as proxies for real-world instances. However, it is not obvious how to assess the quality of the instances produced by any such generator. We propose a new approach for evaluating the usefulness of an arbitrary set of instances for use as proxies during solver development, and introduce a new metric, Q-score, to quantify this. We apply our approach on several artificially generated and real-world benchmark sets and quantitatively compare their usefulness for developing competitive SAT solvers.

Keywords: SAT, benchmark sets, instance generation, automated configuration

1 Introduction & Background

The Boolean satisfiability problem (SAT) is perhaps the most widely studied $\mathcal{NP}$-complete problem; as many advances in SAT have direct implications for solving other important combinatorial problems, SAT has been a focus of intense research in algorithms, artificial intelligence and several other areas for several decades. State-of-the-art SAT solvers have proven to be effective in real-world applications – particularly, Conflict-Driven Clause Learning (CDCL) solvers in the area of hardware and software verification. This has been one of the driving forces in the substantial progress on practical SAT solvers, as witnessed in the well-known SAT competitions, where instances from applications are often referred to as *industrial* instances. The SAT competitions

also feature a separate track for randomly generated instances, with a particular focus on the prominent class of uniform k-CNF instances at or near the solubility phase transition [10] (henceforth, *random* instances). The competitions separate industrial and random instances into distinct tracks, because they tend to have very different structures [3], and because SAT solvers that perform well on random instances (*e.g.*, KCNFS [11]) tend to perform poorly on industrial instances, and vice-versa. The industrial instances used in SAT competitions are often large, routinely containing millions of variables, whereas challenging random instances are significantly smaller. Industrial instances are typically vastly easier than random instances of comparable size.

Many of the industrial instances available to the public belong to the sets used in prior competitions. Developers of SAT solvers targeting industrial instances tend to configure and test their solvers on this limited supply of instances, which can lead to overfitting (see, *e.g.*, [19]) and test set contamination. Furthermore, the instances used in competitions can be very large, and are often unsuitable for performing the more extensive experiments carried out during the design and optimisation of solvers. Therefore, developers of SAT solvers would benefit from access to a large quantity of industrial instances spanning a large range of sizes and difficulty. Ideally, smaller or easier instances would satisfy the criterion that improvements made in solving them can be expected to scale to larger (competition-sized) or harder instances. Such smaller or easier instances would effectively act as *proxies* for the *target* instances that are ultimately to be solved.

The development of solvers targeted for hard, random k-CNF instances has benefited for a long time from the availability of generators that can easily produce large quantities of instances of varying sizes and difficulty. The development of generators for instances bearing close resemblance to real-world SAT instances has been one of the "ten challenges in propositional reasoning and search" posed in 1997 by Selman *et al.* [26] and was reaffirmed as an important goal by Kautz & Selman in 2003 [21]. The challenge calls for the automated generation of SAT instances that "have computational properties that are more similar to real-world instances" [26], and it remains somewhat unclear how to assess the degree to which a generator meets this goal. Despite this, many generators have been proposed as more realistic alternatives to k-CNF. These include several instance generators derived from graph theory problems([25,27,13]), and the quasigroup completion problem (QCP) [12,1].

More recently, Ansótegui *et al.* [2] proposed a set of instance generators, including one which was specifically designed to produce 'industrial-like' instances that exhibit some of the same statistical properties as real-world industrial instances. Another industrial-like instance generator was presented by Burg *et al.* [9], which combines small segments of instances from real-world instances to produce new instances.[3] Finally, Järvisalo *et al.* [20] proposed an instance generator derived from finding optimal circuits for simultaneously computing ensembles of Boolean functions. While this last generator makes no specific claims of industrial-like properties, its instances are de-

[3] Unfortunately, this instance generator is not publicly available.

rived from (random) circuits and so we speculate that they may share properties with industrial instances derived from (real-world) circuits.

Here, we propose a new approach for assessing instance generators – and, indeed, arbitrary sets of instances – in terms of how useful they are as proxies for real-world instances during the development SAT solvers. (Actually, our approach is not specific to SAT, and generalises to other problems in a straightforward manner). In particular, we motivate and define a new metric, Q*-score*, to measure the extent to which optimising the performance on a given instance set results in performance improvements on a set of target instances used for testing purposes (*e.g.*, in the context of a competition or real-world application). Q-score is particularly useful in situations where benchmark sets that are known *a priori* to be good proxies for the target instances are either not available (*e.g.*, because the supply of target instances is too limited) or not usable for performance optimisation (*e.g.*, because they are too difficult). We note that this premise provides the core motivation for developing random generators for 'industrial-like' SAT instances. It also stands in contrast to standard situations in machine learning, where the data used for training a prediction or classification procedure is typically representative of the testing data used for assessing the performance of the trained procedure. This latter observation is relevant, because the development of a SAT solver resembles a training process in machine learning in that both aim at optimising performance on certain classes of input data. This aspect of solver development is captured in the notion of automated algorithm configuration, an approach that has proven to be very effective for the development of high-performance SAT solvers [17,22,30,29].

We define the Q-score in Section 2, and then use it to measure the usefulness of benchmark sets obtained from four instance generators with respect to three industrial target sets using two different highly parametric solvers. In Section 3, we describe the three target sets and the four generators used in our experiments: an 'industrial-like' generator proposed by Ansótegui *et al.* [2], the ensemble-circuit generator from [20], a 'fuzzing' tool for debugging SAT solvers [8], and a reference uniform random 3-CNF generator [10]. Also in Section 3, we provide details on the two highly parametric algorithms LINGELING [7] and SPEAR [6], and on the way in which we configured these using two fundamentally different configurators: PARAMILS [19,18], and SMAC [16]. The results from our experiments, reported in Section 4, indicate that the 'industrial-like' generator proposed in [2] is not generally suitable as a proxy during SAT solver development, while the ensemble-circuit generator from [20] can indeed produce useful instances. In Section 5 we summarise the insights gained from our work and propose some avenues for future research.

2 Quantitative Assessment of Instance Set Utility

In the following, we introduce our new metric for determining the utility of using a given instance set S as a proxy for a target instance set T during the training and development of new solvers. Our primary motivation is to aid the development of a new

algorithm that we wish to perform well on the target instance set T, when using T itself during development is infeasible (*e.g.*, the instances in T are prohibitively large, too costly, or not available in sufficient numbers). Under such circumstances, we would like to use some other instance set, S, to develop and train our algorithm. In particular, we might wish to use randomly generated instances with 'realistic' properties as proxies for the target instances.

Our metric requires a *reference algorithm* A, which is typically not the same algorithm we are interested in developing. For example, we may choose A as one of the current state-of-the-art algorithms for solving instances in the target set T, or A may be a previous version of an algorithm that we are trying to improve upon. The configuration space of A, which we denote as Θ, is ideally quite large and sufficiently rich to permit effective optimisation of A for many different types of instances. Algorithms that have been designed to have a large configuration space are known as *highly parametric* algorithms [6,23,7,30,29,15]. The primary criterion for selecting A is the quality of its parameter configuration space Θ; ideally, to solve instances both in T and outside of T, and with significantly different optimal configurations for each.

Our metric also requires a cost statistic c to measure the performance of the algorithm with a given configuration θ. We use the notation $c(A(\theta), X)$ to represent the cost of running configuration θ of A on each instance in set X. Cost statistics used in the literature include the average run-time, average run-length, percent of instances not solved within a fixed time, and PAR10, which we describe in Section 3. For convenience, we will assume that c is to be minimized and is greater than zero; otherwise, a simple transformation can be used to ensure this is true. We use the notation θ^*_X to represent the optimal configuration of A for an instance set X for the given cost statistic c. The cost statistic used in the context of assessing instance set utility should reflect the way performance is assessed when running the algorithm of interest on T.

We now define our metric, $Q(S, T, A, c)$ as the ratio of the performance of algorithm A in its optimal configuration for target instance set T, and the performance of A in its optimal configuration for the proxy instance set S, both evaluated on instance set T according to cost statistic c. Formally,

$$Q(S, T, A, c) = \frac{c(A(\theta^*_T), T)}{c(A(\theta^*_S), T)}.$$

We use $Q_T(S)$ as a shorthand for $Q(S, T, A, c)$ if A and c are held fixed and are clear from the context, and we refer to $Q_T(S)$ as the *Q-score of S given T*. The closer $Q_T(S)$ is to one, the more suitable set S is as a proxy for target set T and conversely, the lower $Q_T(S)$, the less suitable S is as a proxy for T. Intuitively, $Q_T(S)$ can be interpreted directly as the percentage of optimal performance that can be obtained through optimising algorithm A based on the proxy instances in S.

In practice, the optimal configuration θ^*_T will typically be unknown. We can approximate it by θ'_T, the *best known* configuration of A on target set T. This best known configuration can be drawn from any source, and represents an upper bound on the cost

of the real optimal configuration. Conveniently, an approximate Q-score computed using θ'_T will still always be ≤ 1 (as otherwise, some other known configuration would be better than the best known configuration). Similarly, the optimal configuration θ^*_S (optimal in terms of performance on the proxy set, not the target set) will also be unknown; one convenient way to obtain an approximation θ'_S is by applying automatic configuration of A on the proxy set S.

The approximate Q-score for a given algorithm, proxy set, and target set can then be calculated as follows:

1. Obtain θ'_S by configuring algorithm A on the proxy set S using some method (such as one of the automatic configurators discussed in Section 3).
2. Evaluate this configuration on some instances from the target set, T, using a cost metric such as PAR10, to obtain $c(A(\theta'_S, T))$.
3. Evaluate some other, known configurations of A (for example, the default configuration) on those same target set instances.
4. Let θ'_T be the configuration with the lowest cost from any of the evaluations above (including θ'_S), and let $c(A(\theta'_T, T))$ be the corresponding cost.
5. Compute $Q_T(S) = \frac{c(A(\theta'_T),T)}{c(A(\theta'_S),T)}$

This process entails collecting a set of good, known configurations of A for T to find a good approximation θ'_T of the optimal configuration θ^*_T. One way to improve that approximation is to generate new configurations by applying automatic configuration of A on T (as we do in this work). This may not always be possible, nor is it strictly necessary to compute an approximate Q-score; but we recommend it where practical. However, if automated configuration is applied to T, it is critical to use a set of training instances for configuration that does not contain any of the instances from T that are used for evaluating the Q-score. The reasons for this are somewhat subtle, but worth discussing in some more detail.

Particularly when a given target set T consists of a small set of available real-world instances, these instances are assumed to be representative of a larger set of real-world instances inaccessible to the algorithm designer (or experimenter), and our goal is to improve performance on that larger set, rather than on the specific representative instances we have available. Under these circumstances, applying automatic configuration on the same instances as we use for evaluation may result in *over-tuning* - that is, it may produce a configuration that performs well on those exact instances, but generalizes poorly to the larger set of (unavailable) instances.

Ideally, we would have enough instances available from T that we can afford to split them into two disjoint sets, and use one for configuration, and the other for evaluation and Q-score computation. However, since a motivating factor for producing instance generators in the first place is having access to only limited numbers of instances from T, there may not be sufficiently many instances to split T into disjoint training and testing sets that can both be seen as representative of T. We encounter precisely this

dilemma in Section 3, where we resolve it by configuring instead on a set of instances that we believe to be similar to the target set. As the approximate Q-score does not define where the best known configuration comes from, this is entirely safe to do: in the worse case, configuring on other instance families may simply produce bad configurations that fail to improve on the best known configuration.

We also ensure that the instances in our candidate proxy sets S can be solved efficiently enough for purposes of algorithm configuration. In particular, for some cost statistics, Q-scores can be pushed arbitrarily close to 1 simply by adding a large number of unsolvable instances to the target set; to avoid this possibility, we exclude from the target set T any instances that were unsolved by any configuration of a given solver. Even after this consideration, if the performance differences between different configurations of A on the instance sets of interest, and in particular on T, are small, then all Q_T values will be close to one and their usefulness for assessing instance sets (or the generators from which the instance sets were obtained) will be limited.

Below, we will provide evidence that for algorithms with sufficiently large and rich configuration spaces, the differences in Q_T measures for different candidate proxy sets tend to be consistent, so that sets that are better proxies w.r.t. a given algorithm A tend to also be better proxies w.r.t. a different algorithm A', as long as A and A' are not too different. This latter argument implies that instance sets (or generators) determined to be useful given some target set T (*e.g.*, industrial instances or, more specifically, hardware verification instances) for some baseline solver can be reused, without costly recomputation of Q-scores, for the development of other solvers. Tompkins *et al.* [29] used a metric analogous to Q-score, although their purpose was significantly different than that underlying our work presented here, and observed configurations where the best known configuration for a set was found while configuring for a different set.

3 Experimental Setup

We now turn to the question of how useful various types of SAT instances are as proxies for typical industrial instances. For this purpose, we used four instance generators (Double-Powerlaw, Ensemble, Circuit-Fuzz and 3-CNF), three industrial target sets (SWV, HWV and SAT Race), two high-performance, highly parametric SAT solvers (SPEAR and LINGELING), and the automatic algorithm configurators PARAMILS and SMAC.

The first generator we selected is the Double-Powerlaw generator from Ansótegui *et al.* [2]. Of the five generators introduced in that work, Double-Powerlaw was identified as the most 'industrial-like' by its authors, as it was the only one that they found to produce instances on which a typical CDCL SAT solver known to perform well on industrial instances, MINISAT (version 2 [28]), consistently out-performed the solvers MARCH [14] and SATZ [24], which perform much better on random and *crafted (hand-made)* instances than on industrial instances. Using the software provided by Ansótegui

et al. with the same parameters used in their experiments ($\beta = 0.75, m/n = 2.650, n = 500\,000$), we generated 600 training instances at the solubility phase-transition of the Double-Powerlaw instance distribution.

The second generator is the random ensemble-circuit generator, GENRANDOM, from Section 5.2 of Järvisalo *et al.* [20]. This generator takes parameters (p, q, r), two of which (p and q) were set to 10 in their experiments. Our own informal experiments suggested that a value of $r = 11$ produces a mix of satisfiable and unsatisfiable instances that are moderately difficult for LINGELING (requiring between 10 and 200 seconds to solve); larger and smaller values produce easier instances that are dominated by either satisfiable or unsatisfiable instances. We make no claim that these are optimal settings for this generator, but we believe that they are reasonable and produce interesting instances. We used the script provided by Järvisalo *et al.* to generate 600 training instances with $p = 10, q = 10, r = 11$.

The third generator is adapted from the circuit-based CNF fuzzing tool FUZZSAT [8] (version 0.1). FUZZSAT is a *fuzzing* tool, intended to help the designers of SAT solvers test their code for bugs by randomly generating many instances. It randomly constructs combinational circuits by repeatedly applying the operations AND, OR, XOR, IFF and NOT, starting with a user-supplied number of input gates. The tool then applies the Tseitin transformation to convert the circuit into CNF. Finally, a number of additional clauses are added to the CNF, to further constrain the problem. While not designed with evaluation or configuration in mind, these instances are structured in ways that resemble (at least superficially) real-world, circuit-derived instances, and hence might make good proxies for such instances. However, the instances generated by the tool are usually very easy and typically solved within fractions of a second. This is a useful property for testing, but not for configuration, since crucial parts of a modern CDCL solver might not be sufficiently exercised to realistically assess their efficacy. Adjusting the number of starting input gates allows the size of the circuit to be controlled, but even for moderately sized random circuits, most generated instances remain very easy. In order to produce a set of instances of representative difficulty, we randomly generated $10\,000$ instances with exactly 100 inputs using FUZZSAT (with default settings), and then filtered out any instances solvable by the state-of-the-art SAT solver LINGELING (described below) in less than 1 CPU second. This yielded a set of 387 instances, of which 85 could be proved satisfiable by LINGELING, 273 proved unsatisfiable, and the remaining 29 could not be solved within 300 CPU seconds. We make no claim that these instances are near a solubility phase-transition, or that these are the optimal settings for producing such instances; however, they do represent a broad range of difficulty for LINGELING, which makes them potentially useful for configuration. We selected 300 of these instances to form a training set.

The fourth generator we selected is the random instance generator used in the 2009 SAT Competition. There is strong evidence that these instances are *dissimilar* to typical industrial instances [3], and we included them in our experiments primarily as a control.

We generated a set of 600 training instances composed of 100 instances each at 200, 225, 250, 275, 300, and 325 variables at the solubility phase transition [10]. While other solvers can solve much harder random instances than these, they are an appropriate size for experimentation with the reference algorithms we selected (SPEAR and LINGELING, see below).

We picked two classes of industrial instances known from the literature as our target instance sets. The first is a set of hardware verification instances (HWV) sampled from the IBM Formal Verification Benchmarks Library [31], and the second consists of software verification instances (SWV) generated by CALYSTO [5]. Both of these sets have been employed previously by Hutter *et al.* in the context of automatically configuring the highly parametric SAT solver SPEAR, and we used the same disjoint training and testing sets as they did [17].

The third target instance set is from the 2008 SAT Race and includes a mix of real-world industrial problems from several sources (including the target sets we selected). This is the same set used by Ansótegui *et al.* for evaluating their instance generators [2]. The SAT Race 2008 organizers used a separate set of instances to qualify solvers for entry into the main competition. As there are only 100 instances from the main competition, instead of splitting them into training and testing sets, we used these qualifying instances to train the solvers and tested on the complete set of main competition instances. This qualifying set is comprised of real industrial instances, but from different sources than the instances used in the actual SAT Race. Still, as we will show below, configuring on the qualifying instances produced the best configurations for each solver.

We selected two highly parametric, high-performance CDCL SAT solvers for our experiments. The first is LINGELING [7] (version 276), which won third place in the application category of the 2011 SAT Competition. The second is SPEAR [6] (version 32.1.2.1), one of the first industrial SAT solvers designed to be highly parametric, which won the QF_BV category of the 2007 SMT Competition. These two solvers were chosen based on their performance on industrial instances and their large configuration space ($\approx 10^{17}$ and $\approx 10^{46}$ configurations, respectively[4]). Furthermore, these solvers were developed entirely independently from each other, with very different configuration spaces. LINGELING has many parameters controlling the behaviour of its preprocessing/in-processing and memory management mechanisms, while SPEAR features several alternative decision and phase heuristics.

Both LINGELING and SPEAR were configured for each of our five training sets using two independent configurators: PARAMILS [17,18], and SMAC [16]. Both configurators optimised the Penalised Average Runtime (PAR10) performance, with a cut-off of 300 seconds for each run of the solver to be configured. PAR10 measures the average runtime, treating unsolved instances as having taken 10 times the cut-off time.

[4] PARAMILS can only configure over finite, discretized configuration spaces. Parameters taking arbitrary integers or real numbers were manually discretized to a small number of representative values (< 10), from which the spaces above were computed.

Table 1. PARAMILS configurations of LINGELING running on the target instances. Best known configurations are shown in boldface. Q-scores closer to 1 are better.

LINGELING	HWV			SWV			SAT Race 2008		
Config.	Q	PAR10	#Solved	Q	PAR10	#Solved	Q	PAR10	#Solved
3-CNF	0.043	182.8	286/291	0.005	32.3	301/302	0.175	3512.4	58/93
Double-Powerlaw	0.095	82.9	289/291	0.005	34.6	301/302	0.209	2944.7	64/93
Circuit-Fuzz	0.175	45.2	290/291	0.132	1.2	302/302	0.437	1404.5	80/93
Ensemble	0.766	10.3	291/291	0.079	2.1	302/302	0.562	1092.8	83/93
HWV	**1.000**	**7.9**	**291/291**	0.078	2.1	302/302	0.621	989.8	84/93
SWV	0.095	83.2	289/291	0.879	0.2	302/302	0.217	2825.2	65/93
SAT-Race Qualifying	0.624	12.6	291/291	0.203	0.8	302/302	**1.000**	**614.3**	**88/93**
Default	0.724	10.9	291/291	0.054	3.0	302/302	0.624	984.0	84/93

Configuration remains a compute-intensive step. Following a widely used protocol for applying PARAMILS, we conducted ten independent runs for each of our fourteen pairs of solvers and training sets, allocating 2 CPU days to each of those runs. For each solver and training set combination we then evaluated the ten configurations thus obtained on the full training set and selected the one with the best PAR10 score; this second stage required as much as three additional days of CPU time. The same protocol was used for SMAC. Carried out on a large compute cluster using 100 cores in parallel, this part of our experiments took five days of wall clock time and resulted in seven configurations for each configurator on both SAT solvers (in addition to their default configurations), which we refer to as SAT-Race, HWV, SWV, 3-CNF, Circuit-Fuzz, Ensemble, and Double-Powerlaw. We then evaluated each configuration on each target testing set using a cut-off time of 15 CPU minutes per instance. On the HWV and SAT Race target sets, there were some instances that were not solved by any configuration of each solver. We have excluded those instances from the results for the respective solvers, to avoid inflating the Q-scores, as discussed above. We note that, unlike in a competition scenario, this does not distort our results, as the purpose of our study was not to compare solver performance.

All experiments were performed on a cluster of machines equipped with six-core 2.66GHz 64-bit CPUs with 12 MB of L3 cache running Red Hat Linux 5.5; each configuration and evaluation run had access to 1 core and 4GB of RAM.

4 Results & Analysis

Results for each configuration against the three target instance sets (HWV, SWV, and SAT Race 2008) are presented in Tables 1-4; for reference, the performance of each respective solver's default configuration is shown in the bottom rows. As seen from these data, in all cases the best known configuration of each solver was found through automatic configuration (sometimes by SMAC, and sometimes by PARAMILS).

Table 2. SMAC configurations of LINGELING running on the target instances. Best known configurations are shown in boldface. Q-scores closer to 1 are better.

LINGELING	HWV			SWV			SAT Race 2008		
Config.	Q	PAR10	#Solved	Q	PAR10	#Solved	Q	PAR10	#Solved
3-CNF	0.065	121.5	288/291	0.005	32.7	301/302	0.183	3349.2	60/93
Double-Powerlaw	0.100	78.7	289/291	0.090	1.8	302/302	0.273	2250.3	71/93
Circuit-Fuzz	0.053	150.1	287/291	0.932	0.2	302/302	0.215	2852.4	65/93
Ensemble	0.177	44.6	290/291	0.073	2.3	302/302	0.551	1114.6	83/93
HWV	0.720	11.0	291/291	0.158	1.0	302/302	0.471	1303.5	81/93
SWV	0.100	79.3	289/291	**1.000**	**0.2**	**302/302**	0.327	1879.8	75/93
SAT-Race Qualifying	0.178	44.3	290/291	0.177	0.9	302/302	0.439	1399.2	80/93
Default	0.724	10.9	291/291	0.054	3.01	302/302	0.624	984.0	84/93

Table 3. PARAMILS configurations SPEAR running on the target instances. Best known configurations are shown in boldface. Q-scores closer to 1 are better.

SPEAR	HWV			SWV			SAT Race 2008		
Config.	Q	PAR10	#Solved	Q	PAR10	#Solved	Q	PAR10	#Solved
3-CNF	0.265	376.3	279/290	0.001	881.0	273/302	0.487	4336.6	45/78
Double-Powerlaw	0.083	1111.7	255/290	< 0.001	1737.2	244/302	0.298	6712.4	23/78
Circuit-Fuzz	0.211	435.8	276/290	0.001	907.4	273/302	0.615	3589.0	52/78
Ensemble	0.084	1097.9	256/290	0.001	1389.5	256/302	0.499	4251.0	46/78
HWV	**1.000**	**91.8**	**287/290**	0.001	695.4	279/302	0.687	3292.2	55/78
SWV	0.045	2058.2	224/290	0.641	1.19	302/302	0.300	6585.8	23/78
SAT-Race Qualifying	0.800	114.7	286/290	0.469	1.62	302/302	**1.000**	**1909.3**	**63/78**
Default	0.213	430.2	277/290	0.012	64.5	300/302	0.591	3706.7	51/78

The Q-scores provide us with quantitative insight regarding the extent to which the instance generators can serve as proxies for the three sets of real-world instances considered here. For example, overall, there are only two cases where a configuration on generated instances produced a SAT-solver that scored above 0.75 (*i.e.*, was within 25% of the best known configuration's performance). Both of these involve LINGELING: once when configured by PARAMILS on the Ensemble instances and running on the HWV target set, and once when configured by SMAC on the Circuit-Fuzz instances and running on the SWV target set. However, in both cases this very strong performance of a generated instance configuration seems to be an isolated occurrence, one that is not replicated with SPEAR or the other configurator. This suggests that none of the four generated instance sets could be considered excellent matches to any of the three industrial instance sets considered here (for the purposes of developing SAT solvers).

However, there are still substantial differences between the generators: Considering the Q-scores in Tables 1–4, we observe that, as expected, the 3-CNF instances did not

Table 4. SMAC configurations of SPEAR running on the target instances. Best known configurations are shown in boldface. Q-scores closer to 1 are better.

SPEAR	HWV			SWV			SAT Race 2008		
Config.	Q	PAR10	#Solved	Q	PAR10	#Solved	Q	PAR10	#Solved
3-CNF	0.199	462.1	276/290	0.001	1496.9	252/302	0.533	3579.3	48/78
Double-Powerlaw	0.061	1497.6	243/290	< 0.001	1981.4	236/302	0.262	7300.2	15/78
Circuit-Fuzz	0.271	338.9	280/290	0.001	613.6	282/302	0.633	3015.1	53/78
Ensemble	0.163	563.8	273/290	0.001	823.7	275/302	0.611	3125.6	52/78
HWV	0.787	116.6	287/290	0.289	2.63	302/302	0.662	2885.7	54/78
SWV	0.029	3122.0	190/290	**1.000**	**0.762**	**302/302**	0.280	6818.2	19/78
SAT-Race Qualifying	0.188	487.7	275/290	0.003	231.0	295/302	0.572	3338.2	50/78
Default	0.213	430.2	277/290	0.012	64.5	300/302	0.591	3231.7	51/78

provide effective guidance towards good configurations for real-world instances: in only one case we obtained performance within 50% of the best known configuration, and in most cases the Q-scores are well below 25% of optimal.

On the other hand, solvers configured on the Circuit-Fuzz instances showed better performance, especially on the SAT Race instances. SPEAR always improved its performance on the SAT Race instances relative to the default configuration when configured using the Circuit-Fuzz instances. This provides evidence that the Circuit-Fuzz instances can make reasonable proxies for real-world SAT Race instances. However, we also observe that these are at best imperfect proxies: LINGELING, a SAT solver that has been more heavily optimized for performance on SAT Race instances, always performed worse than the default on the SAT Race instances after configuring on the Circuit-Fuzz instances (however, configuring LINGELING on the Circuit-Fuzz instances was still better than configuring on 3-CNF).

The evidence for the utility of the Ensemble instances is much stronger. In three out of four cases, configuring on the Ensemble instances lead to a solver that obtained a runtime $> 50\%$ of best known configuration on the SAT Race 2008 target set, and even in the remaining case its runtime was only just barely less than 50% of the best known configuration. This is not stellar performance – but it is not dismal, either: we can conclude that the Ensemble instance are moderately effective proxies for the SAT Race target set.

Our results also provide a clear answer to the question whether the Double-Powerlaw instances can serve as useful proxies in solver design for the types of industrial instances considered here. Neither LINGELING nor SPEAR when configured on these instances performed well on any of our three target sets; not once did configuring on the Double-Powerlaw instances produce a solver that was within 50% of the best known configuration. Strikingly, we can see that in 7 of 12 cases, the Double-Powerlaw configurations performed worse than the 3-CNF configurations, and even in the remaining cases, it was better than 3-CNF by less than 10%.

Finally, we compared results between LINGELING and SPEAR to assess of the robustness of our Q-score measure. As we observed for the Circuit-Fuzz instances, there are certainly differences between these solvers, and an instance set may be more useful for one than for the other. However, our results indicate that even for these very different solvers (in terms of configuration space, design and implementation), Q-scores are generally quite consistent, especially if the same configurator is used. For example, configuring either SPEAR or LINGELING on the HWV training instances always produced reasonably good results on the SAT Race 2008 target set. Conversely, training either solver on the SWV training instances always produced poor results on the HWV and SAT Race 2008 instances. Training either solver on 3-CNF or Double-Powerlaw always produced poor results on HWV and SWV; as observed above, training on the Ensemble instances always produced reasonably good results on the SAT Race 2008 instances.

That said, PARAMILS and SMAC sometimes produced inconsistent results. For example, using PARAMILS to configure either solver on the SAT Race Qualifying instances produced good performance on the HWV target set, whereas poor performance was observed on the same set for configurations observed from SMAC. We speculate that this could be due to the way in which the model-based search approach underlying SMAC reacts to characteristics of the given instances and configuration spaces. Nevertheless, these inconsistencies were quite rare, and even when using different configurators, results were usually highly consistent between solvers.

Closer examination of the Double-Powerlaw instances provided strong evidence that, despite sharing some statistical similarities with actual industrial instances, they give rise to very different behaviour in standard SAT solvers. In particular, we found that the Double-Powerlaw instances (both satisfiable and unsatisfiable) are without exception extremely easy for industrial SAT solvers to solve. A typical industrial instance of medium difficulty tends to require a modern CDCL solver to resolve tens or hundreds of thousands of conflicts; these conflicts arise from bad decisions made by the decision heuristic while searching for a satisfying assignment of literals. In contrast, the Double-Powerlaw instances can typically be solved (by MINISAT [28], which reports this information conveniently) with less than 100 conflicts – even though these instances are very large (containing $500\,000$ variables and 1.3 million clauses).

Unfortunately, there is not much room to make these instances larger without causing solvers to run out of memory (though we have experimented informally with generating Double-Powerlaw instances that are 10 times larger, and found that they are not substantially harder to solve). Moreover, we found these instances to be *uniformly* easy to solve – even out of thousands of generated instances, none took more than 10 seconds to solve using SPEAR or LINGELING. For this reason, filtering by difficulty, as we did with the Circuit-Fuzz instances, would not be effective.

5 Conclusions & Future Work

We have introduced a new configuration-based metric, the Q-score, for assessing the utility of a given instance set for developing, training and testing solvers. The fundamental approach underlying this metric is based on the idea of using the automated configuration of highly parametric solvers as a metaphor for a solver development process aimed at optimising performance on particular classes of target instances. Although the notion of Q-score applies to highly parametric solvers for arbitrary problems, our motivation for developing it was to assess how actual instance generators can serve as proxies for a range of SAT instances as considered in the literature.

We found strong evidence that the Double-Powerlaw instances do not fulfill that role well, as indicated by robust, consistent results obtained for two high-performance CDCL SAT solvers with very different configuration spaces, LINGELING and SPEAR, across three separate sets of industrial target instances, and using two different configurators, PARAMILS and SMAC. We also presented evidence that the generated Ensemble instances are moderately effective for configuring for the SAT Race 2008 competition instances. Along with our results for the Circuit-Fuzz instances, this suggests that generating random instances in the original problem domain (circuits, in these two cases) might be a promising area for future industrial-like instance generators.

Because our metric does not depend on any specific properties of the generators or target domains, it should be widely applicable for evaluating the usefulness of many different types of instance generators, and on any target instance set for which there is an appropriate parametric solver (one whose design space includes good configurations for those target instances). As argued by Selman *et al.* [26], generators that can produce instances resembling real-world instances would be valuable in the development of SAT solvers. By providing an approach to evaluate candidates for such generators, we hope to spur further research in this direction. We see the work by Ansótegui *et al.* [2] as a valuable first step in this direction, but as indicated by our findings reported here (and also reflected in the title of their publication), much work remains to be done.

Finally, as our metric can be evaluated automatically, we can in principle use it to configure instance generators themselves. Generators are typically parametrized; it may not be known in advance what settings produce the most appropriate instances for training. Instead of finding generator settings that produce difficult instances or that correspond to a phase transition, automatic algorithm configuration based on Q-score could identify generator settings that produce instances that make good proxies for interesting classes of real-world SAT problems.

Acknowledgments

This research has been enabled by the use of computing resources provided by WestGrid and Compute/Calcul Canada, and funding provided by the NSERC Canada Graduate Scholarships and Discovery Grants Programs.

References

1. D. Achlioptas, C. Gomes, H. Kautz, and B. Selman. Generating satisfiable problem instances. In *Proceedings of the national conference on artificial intelligence*, pages 256–261. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2000.
2. Carlos Ansótegui, María Luisa Bonet, and Jordi Levy. Towards industrial-like random SAT instances. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 387–392, 2009.
3. Carlos Ansótegui, María Luisa Bonet, Jordi Levy, and Felip Manyà. Measuring the hardness of SAT instances. In *Proceedings of the Twenty-Third National Conference on Artificial Intelligence (AAAI-08)*, pages 222–229, 2008.
4. Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Constraint Programming (CP-09)*, volume 5732 of *Lecture Notes in Computer Science*, pages 142–157, 2009.
5. Domagoj Babić and Alan J. Hu. Structural abstraction of software verification conditions. In *Proceedings of the Nineteenth International Conference on Computer Aided Verification (CAV-07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 371–383, 2007.
6. Domagoj Babić and Frank Hutter. Spear theorem prover. Solver Description, SAT Race 2008, 2008.
7. Armin Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT race 2010. Technical Report 10/1, FMV Report Series, Institute for Formal Models and Verification, Johannes Kepler University, 2010.
8. R. Brummayer, F. Lonsing, and A. Biere. Automated testing and debugging of SAT and QBF solvers. *Theory and Applications of Satisfiability Testing–SAT 2010*, pages 44–57, 2010.
9. S. Burg, S. Kottler, and M. Kaufmann. Creating industrial-like sat instances by clustering and reconstruction. *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 471–472, 2012.
10. Vašek Chvátal and Endre Szemerédi. Many hard examples for resolution. *Journal of the ACM*, 35(4):759–768, 1988.
11. Gilles Dequen and Olivier Dubois. An efficient approach to solving random k-SAT problems. *Journal of Automated Reasoning*, 37(4):261–276, 2006.
12. C.P. Gomes, B. Selman, et al. Problem structure in the presence of perturbations. In *Proceedings of the National Conference on Artificial Intelligence*, pages 221–226. JOHN WILEY & SONS LTD, 1997.
13. H. Haanpää, M. Järvisalo, P. Kaski, and I. Niemelä. Hard satisfiable clause sets for benchmarking equivalence reasoning techniques. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):27–46, 2006.
14. Marijn J. H. Heule and Hans van Maaren. Whose side are you on? finding solutions in a biased search-tree. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:117–148, 2008.
15. Holger H. Hoos. Programming by optimization. *Communications of the ACM*, 55:70–80, February 2011.
16. F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proc. of LION-5*, page 507523, 2011.
17. Frank Hutter, Domagoj Babić, Holger H. Hoos, and Alan J. Hu. Boosting verification by automatic tuning of decision procedures. In *Proceedings of the Seventh International Conference on Formal Methods in Computer-Aided Design (FMCAD-07)*, pages 27–34, 2007.

18. Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.
19. Frank Hutter, Holger H. Hoos, and Thomas Stützle. Automatic algorithm configuration based on local search. In *Proceedings of the Twenty-Second National Conference on Artificial Intelligence (AAAI-07)*, pages 1152–1157, 2007.
20. M. Järvisalo, P. Kaski, M. Koivisto, and J. Korhonen. Finding efficient circuits for ensemble computation. *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 369–382, 2012.
21. Henry Kautz and Bart Selman. Ten challenges redux: Recent progress in propositional reasoning and search. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP-03)*, volume 2833 of *Lecture Notes in Computer Science*, pages 1–18, 2003.
22. Ashiqur R. KhudaBukhsh. SATenstein: Automatically building local search SAT solvers from components. Master's thesis, University of British Columbia, 2009.
23. Ashiqur R. KhudaBukhsh, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. SATenstein: Automatically building local search SAT solvers from components. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 517–524, 2009.
24. Chu Min Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP-97)*, volume 1330 of *Lecture Notes in Computer Science*, pages 341–355, 1997.
25. I. Rish and R. Dechter. Resolution versus search: Two strategies for sat. *Journal of Automated Reasoning*, 24(1):225–275, 2000.
26. Bart Selman, Henry Kautz, and David McAllester. Ten challenges in propositional reasoning and search. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 50–54, 1997.
27. A. Slater. Modelling more realistic sat problems. *AI 2002: Advances in Artificial Intelligence*, pages 591–602, 2002.
28. Niklas Sörensson and Niklas Eén. Minisat v1.13 – a SAT solver with conflict-clause minimization. Poster, Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT-05), 2005.
29. Dave A. D. Tompkins, Adrian Balint, and Holger H. Hoos. Captain Jack: New variable selection heuristics in local search for SAT. In *Proceedings of the Fourteenth International Conference on Theory and Applications of Satisfiability Testing (SAT-11)*, volume 6695 of *Lecture Notes in Computer Science*, pages 302–316, 2011.
30. Dave A. D. Tompkins and Holger H. Hoos. Dynamic scoring functions with variable expressions: New SLS methods for solving SAT. In *Proceedings of the Thirteenth International Conference on Theory and Applications of Satisfiability Testing (SAT-10)*, volume 6175 of *Lecture Notes in Computer Science*, pages 278–292, 2010.
31. Emmanuel Zarpas. Benchmarking SAT solvers for bounded model checking. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT-05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 340–354, 2005.