Structure-Preserving Instance Generation

Yuri Malitsky[1], Marius Merschformann[2], Barry O'Sullivan[3], and Kevin Tierney[2]

[1] IBM T.J. Watson Research Center, New York, USA yuri.malitsky@gmail.com
[2] Decision Support & Operations Research Lab, University of Paderborn, Germany
{tierney,merschformann}@dsor.de
[3] Insight Centre for Data Analytics, University College Cork, Ireland
b.osullivan@insight-centre.org

Abstract. Real-world instances are critical for the development of state-of-the-art algorithms, algorithm configuration techniques, and selection approaches. However, very few true industrial instances exist for most problems, which poses a problem both to algorithm designers and methods for algorithm selection. The lack of enough real data leads to an inability for algorithm designers to show the effectiveness of their techniques, and for algorithm selection it is difficult or even impossible to train a portfolio with so few training examples. This paper introduces a novel instance generator that creates instances that have the same structural properties as industrial instances. We generate instances through a large neighborhood search-like method that combines components of instances together to form new ones. We test our approach on the MaxSAT and SAT problems, and then demonstrate that portfolios trained on these generated instances perform just as well or even better than those trained on the real instances.

1 Introduction

One of the largest problems facing algorithm developers is a distinct lack of industrial instances with which to evaluate their approaches. Yet, it is the use of such instances that helps ensure the applicability of new methods and procedures to the real-world. Algorithm configuration and selection techniques are particularly sensitive to the lack of industrial instances and are prone to overfitting, as it is difficult to build valid learning models when little data is present. Although a plethora of random instance generators exist, the structure of industrial instances tends to be different than that of randomly generated instances, as has been shown for the satisfiability (SAT) problem [4, 3, 20].

In this work, we therefore present a novel framework for instance generation that creates new instances out of existing ones through a large neighborhood search-like iterative process of destruction and reconstruction [28] of structures present in the instances. Specifically, given an instance to modify, m, and a pool of similar instances, P, we destroy elements of m that fit certain properties (such as variable connectivity) and merge portions of the instances in P into m, to create a set of new instances. We compute the features of each new generated instance and accept the instance if it falls into the cluster of instances defined by P. To the best of our knowledge, our framework is the first approach able to generate industrial-like instances directly from real data.

Aside from the immediate benefits of providing a good training set for portfolio techniques, this type of instance generation has the potential of opening new avenues for future research. In particular, the underlying assumption of most portfolio techniques is that a representative feature vector can be used to identify the best solver to be

employed on that instance. Techniques like ISAC [19] take this idea further by claiming that instances with similar features are likely to have the same underlying structure, and can therefore be solved using the same solver. Structure-preserving instance generation uses and furthers this notion, that in order to predict the best solver for a given instance, one should create a plethora of instances with very similar features, train a model on them, and then make a prediction for the original instance. Additionally, given recent results regarding the benefits of having multiple, correlated instances for CSPs [14], our instance generator may be able to help solvers more quickly find solutions, as it can provide such correlated instances.

In theory, structure-preserving instance generation can even be used to generate instances significantly different from those observed before. This could in turn allow the targeted creation of portfolios that can anticipate any novel instances not part of the original training set. It would also allow for a systematic way of studying the problem space to identify regions of hard and easy problems, as in [31] but with a stronger basis in real instances. This would allow algorithm designers to create new approaches to specifically target challenging problems.

In this work we primarily focus on the well-known SAT problem, as well as its optimization version, maximum SAT (MaxSAT). These two problems pose ideal test beds for our approach, as although the number of industrial instances is low, it is still larger than what is available in most other domains. For example, the MaxSAT competition in 2013 [7] had only 55 industrial instances in the unweighted category, as opposed to 167 crafted and 378 random instances. We show that the instances generated by our method have similar runtime profiles as the industrial instances they are based on, that they have similar features, and that they can be used in algorithm selection techniques with no loss of performance and, in some cases, even provide small gains. We then show that our technique will even help for problems with larger available datasets, as is the case with SAT and the 300 available industrial instances from the 2013 SAT Competition [8]. We also evaluate our generator using the Q-score from [11], which is specifically designed for evaluating instance generators, and receive near perfect scores. Finally, our source code is available in a public repository at: `https://bitbucket.org/eusorpb/spig`.

2 Related Work

Numerous random instance generators exist for SAT/CSP problems, such as [16, 9, 32], to name a few[4]. Some generators try to hide solutions within the problem or generate a specific number of solutions ([21] and [27], respectively), whereas others convert problems from other fields to SAT/CSP problems (e.g., [1]). The approach of Slater (2002) [29] creates instances by connecting "modules" of 3SAT instances with a shared component, a structure that is often present in industrial instances. For MaxSAT, several generators exist, e.g. [12], which generates bin packing-like problems. The generator from Motoki [24] can create MaxSAT problems with a specific number of unsatisfied clauses. However, in all of these generators the structures inherent in industrial problems are not present.

[4] An extended version of this work provides a more extensive literature review; see: `https://bitbucket.org/eusorpb/spig/`

The most industry-like SAT/MaxSAT instances are generated by Ansótegui et al. [2] through the modification of a random instance generator to use a power-law distribution. In contrast, our framework is able to specifically target certain types of industrial instances. Our approach is similar to instance morphing [15], the primary difference being our focus on instance features and a destroy-repair paradigm. Furthermore, morphing is meant to "connect" the structures of instances, while our goal is to also find new combinations of structures leading to new areas of the instances' feature space.

Burg et al. propose a way of generating SAT instances by clustering the variables based on their degree in a weighted variable graph in Burg et al. [13]. Several approaches are tested to try to "re-wire" the instance by adding and removing connections between the variables. The authors compute the features from Nudelman et al. [26] for the original instances and the generated ones, noting that several features of the generated instances no longer resemble the original instances. In contrast, the instances we generate have similar features to their original instances and stay in the same cluster as the original instance pool.

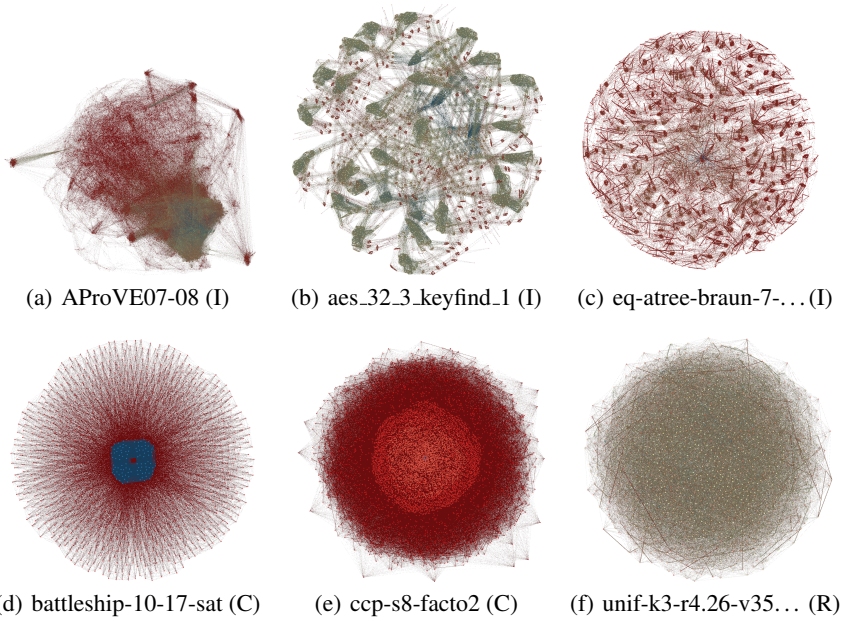
An evolutionary algorithm approach is used by Smith-Miles and van Hemert [31] to generate traveling salesman problem instances that are uniquely hard or easy for a set of algorithms. This approach starts from a random instance under some assumptions about the size and structure of the resulting instance. In Lopes and Smith-Miles [22], real-world-like instances for a timetabling problem are generated using an existing instance generator. While similar to our approach, their work focuses mainly on generating instances that are able to discriminate between solvers in terms of performance. Furthermore, our approach does not require an existing instance generator. TSP instances are also evolved in Nallaperuma et al. [25] for a parameter prediction model for an ant colony optimization model. However, all these works focus on creating hard instances for particular solvers, rather than instances that resemble industrial instances.

The most similar work to ours is from Smith-Miles and Bowly [30], in which instances are generated for the graph coloring problem targeting specific instance features. The authors project their features into a two dimensional space with a principal component analysis, and then check if the instance features to be generated are in a feasible region of the space. A genetic algorithm is then used to try to find an instance matching the input features. This approach is more general than ours, but it is not known how well it works with industrial instances, or other problem types.

3 Structure-preserving Instance Generation

Our instance generation algorithm is motivated by the differing structures found in SAT instances, especially between the industrial, crafted and random categories of instances. Figure 1 provides some visualizations of SAT instances based on their clause graphs. In these graphs, each node represents a clause in the formula, with an edge specifying that the two clauses share at least one variable. The nodes are also color coded from red to blue, where nodes with only a few edges are colored red and those with the most edges are colored blue. A force-based algorithm is used to spread nodes apart. In this way, nodes that share many edges between each other are pulled together into clumps, while the others are pushed away.

Note that the industrial instances, which in Figure 1 are (a), (b) and (c), tend to contain a core set of clauses that share at least one variable with many other clauses.



(a) AProVE07-08 (I) (b) aes_32_3_keyfind_1 (I) (c) eq-atree-braun-7-... (I)

(d) battleship-10-17-sat (C) (e) ccp-s8-facto2 (C) (f) unif-k3-r4.26-v35... (R)

Fig. 1. Visualizations of industrial (I), crafted (C) and random (R) instances made with Gephi [10]. Nodes are clauses, and an edge exists if the corresponding clauses share a variable.

In addition, a large number of small subsets of clauses are built on the same set of variables. A few variables link the subsets of clauses to the common core. In contrast, instances (d) and (e), which are from the crafted category from the SAT competition, and (f), which is from the random category, show significantly more connectivity between clauses and less modules or groupings of nodes within the graph.

Given a pool of instances, P, and an instance to modify, m, structure-preserving instance generation works as shown in Algorithm 1 to create a set of generated instances gen. The instance pool should be a set of homogeneous instances, such as the instances in a particular cluster from the ISAC method [19]. While using a heterogeneous pool would still result in instances, recall that in this work we aim to create instances with similar properties. In cases where an industrial instance has no similar instances, our method can still be used with a pool consisting of only the instance to modify. The parameters $\alpha \in [0, 1]$ and $\beta \in [1, \infty]$ define the minimum and maximum of the size of the generated instances in proportion to m, respectively. We use these values as general guidelines rather than hard constraints in order to prevent the instance from growing too large or too small.

Our proposed algorithm can be thought of as a modified large neighborhood search [28], in which the incumbent solution, in this case the instance to modify, is iteratively destroyed and repaired. The DESTROY function identifies and removes a particular structure or component of m. The destroy process is run at least once, and is continued until the instance drops below its maximum size. The REPAIR function then identifies and extracts structures from one or more randomly chosen instances from P and inserts

Algorithm 1 Structure-preserving instance generation algorithm.

```
1: function SPIG(m, P, α, β)
2:     gen ← ∅, m′ ← m
3:     repeat
4:         do
5:             m′ ← DESTROY(m′, SELECT-STRUCT(m′))
6:         while SIZE(m′) > β · SIZE(m)
7:         do
8:             i ← random instance in P
9:             d ← max(0, VARS(m) − VARS(m′))
10:            m ← REPAIR(m′, SELECT-STRUCT(i), d)
11:        while SIZE(m′) < α · SIZE(m)
12:        if ACCEPT(m′, P ∪ {m}) then
13:            gen ← gen ∪ {m′}
14:    until TERMINATE
15: return gen
```

them into m. This is repeated until m is larger than the minimum instance size. The d parameter taken by the repair method makes sure that the total number of variables in the problem also stays constant. An acceptance criterion determines whether or not the modified instance should be added to the dataset of instances being built. We base this acceptance on the features of the instance and check whether each individual feature is close to the features of the cluster formed by $P \cup \{m\}$. For problems like SAT or CSP, where an unsatisfiable component or tautology could be introduced, a check can be performed to ensure that the instance did not become trivial to solve. The algorithm terminates when enough instances are generated.

Here it may be argued that an alternate search strategy may also work as well or even better than the one outlined in this section. While alternatives are certainly possible, they are beyond the scope of this work (although some have been tried). For example, one can imagine a combination of local search strategies where each method adjusts an instance to match some combination of features. This modifies the internals of an instance while keeping the instance features relatively unchanged, or moves them back if they change too much. The issue with this method is that the features of interest for problems like SAT are highly interdependent, making any fine grained control over them an arduous task at best. Furthermore, it is frequently very easy to make an instance trivial to solve by introducing an infeasibility, a position that is very difficult to remedy.

Alternatively, one can argue that as long as the provided acceptance criteria is utilized as is, it is possible to employ a local search to just try a number of instantiations. While possible in theory, this approach can take a considerable amount of time before stumbling over even a single seemingly useful instance. The problem space of instance generation is simply too vast. Therefore, while we do not claim that the approach presented here is the only way of generating instances or even the best way, it is a systematic procedure that allows rich datasets to be quickly generated that we can empirically demonstrate works well in practice.

4 Application to SAT and MaxSAT

We present an instantiation of the structure-preserving instance generation framework on the NP-complete SAT problem and NP-hard MaxSAT problem. A SAT problem consists of a propositional logic formula F given in conjunctive normal form. The goal of the SAT problem is to find an assignment to the variables of F such that F evaluates to true. MaxSAT is the optimization version of SAT, in which the goal is to find the largest set of clauses of F that have some satisfying assignment. A version of MaxSAT can also have a weight associated with each clause, with the objective then being to maximize the sum of satisfied clauses. In this work, however, we concentrate only on the unweighted variant of MaxSAT and describe our structure identification routines (SELECT-STRUCT), destroy, repair and acceptance operators. Due to the similarity of the SAT and MaxSAT problems, our instance generation procedure is the same with the exception of the acceptance criteria, which we modify to avoid trivial SAT instances.

Structure Identification Many industrial SAT/MaxSAT instances consist of a number of connected components that are bound together through a core of common variables (see Figure 1). Our goal is to identify one of these components in an instance at random and remove it. To this end, we propose two heuristics for identifying such structures that we use in both the destroy and repair functions with 50% probability in each iteration. The first heuristic identifies a set of clauses shared by a particular variable, whereas the second identifies a clause and selects all of the clauses it shares a variable with.

Our goal in the *variable-based selection* heuristic is to identify components of instances with a shared variable, as shown in Algorithm 2. We first calculate the mean number of clauses that a variable is in. Variables in many clauses are likely to be a part of the "core" of an instance that connects various sub-components, whereas variables in the average number of clauses are more likely to be part of the sub-components themselves. The algorithm selects a set of variables, E, in the average number of clauses, if there are any. If E is empty, the algorithm relaxes its strictness of how many clauses a variable should be in until some clauses are found. Finally, the algorithm selects a variable from E at random and returns all of the clauses that variable is present in.

In contrast to our previous heuristic, the *clause-based selection* heuristic focuses on clauses with an average out degree. The out degree of a clause is defined as the number of clauses sharing at least one variable in common with the clause. This corresponds to the out degree of the clause's node in the clause-variable graph. Algorithm 3 accepts an instance i and a parameter σ, described below. The heuristic selects a random clause and compares its out degree to the average out degree of all the clauses. If the clause's out degree is within σ standard deviations of the average clause out degree, we accept the clause and return all of the clauses it is connected to in the clause-variable graph.

Destroy Our destroy function accepts an instance i and a set of clauses C selected by SELECT-STRUCT-VAR or SELECT-STRUCT-CLAUSE that are to be removed from the instance. First, all clauses in C are removed, i.e., $\text{CLAUSES}(i) \leftarrow \text{CLAUSES}(i) \setminus C$, and then all variables that no longer belong to any clause are removed from $\text{VARS}(i)$.

Repair Our repair procedure maps the variables contained within a previously selected set of clauses to the variables present in the instance to modify, and adds new variables

Algorithm 2 Variable based structure selection heuristic.

```
1: function SELECT-STRUCT-VAR(i)
2:     E ← ∅, f ← 0
3:     a ← MEAN-VAR-IN-CLAUSES(i)
4:     while E = ∅ do
5:         E ← {v ∈ VARS(i) | |CLAUSES(v) − a| ≤ f}
6:         f ← f + 1
7: return IN-CLAUSES(random variable in E)
```

Algorithm 3 Clause based structure selection heuristic.

```
1: function SELECT-STRUCT-CLAUSE(i, σ)
2:     C = ∅
3:     mcod ← MEAN-CLAUSE-OUT-DEGREE(i)
4:     scod ← STD-CLAUSE-OUT-DEGREE(i)
5:     while C = ∅ do
6:         c ← random clause in i
7:         cod ← CLAUSE-OUT-DEGREE(c)
8:         if |cod − mcod| < σ · scod then
9:             C ← {c′ ∈ CLAUSES(i) | c and c′ share at least one variable.}
10: return C
```

with some probability. To avoid confusion, we refer to the instance being modified as the *receiver*, and the instance providing clauses as the *giver*. Algorithm 4 shows the repair process, which is initialized with the receiving instance r, the set of clauses to add, C, and some number of variables to add to the instance, d. The parameter d is used to increase the size of the receiver if too many variables are deleted during the destruction phase. Additionally, we note that C contains clauses from the giver, meaning the variables in those clauses do not match those in the receiving instance. Thus, the main action of the repair method is to find a mapping, M, that allows us to convert the variables in C into similar variables in the receiver.

We map the variables in C into the variables of r by computing the following three features for each variable in the VAR-FEATURES function. We use these features because our goal is to map variables with similar connectivity to other parts of the instance with each other and they are easy to compute.

1. Number of clauses the variable is in divided by the total number of instance clauses.
2. Percent of clauses the variable is in, in which the variable is positive.
3. Average of the number of variables of each clause the variable v is in.

On Line 7 of Algorithm 4 we decide whether to map v_g to an existing variable in r or to a new variable. The function TRIVIAL(v_g) returns true if v_g is not (*i*) positive in at least one clause in C, and (*ii*) negated in at least one clause C. This ensures that if we map v_g to a new variable, a valid assignment of v_g is not entirely obvious. We assign v_g to a new variable with probability $d/|V_g|$, as with this probability we add roughly d new variables in the absence of trivial variables.

We compute the $L2$ norm between the giver variables and the receiver's variables on line 12, and should we decide not to add a new variable to r, we map v_g to the variable in

Algorithm 4 SAT/MaxSAT instance repair procedure.

function REPAIR(r, C, d)
  $V_g \leftarrow \bigcup_{c \in C}$ VARS(c)
  $F_r \leftarrow$ VAR-FEATURES(r)
  $F_g \leftarrow$ VAR-FEATURES(V_g)
  $M \leftarrow \emptyset$
  for $v_g \in V_g$ **do**
    if $\neg$TRIVIAL(v_g) and RND$(0, 1) < d/|V_g|$ **then**
      $v' \leftarrow$ new variable
      VARS(r) $\leftarrow$ VARS(r) $\cup \{v'\}$
      $M \leftarrow M \cup \{v_g \mapsto v'\}$
    else
      $dists \leftarrow \{\|F_r(v) - F_g(v_g)\|^2, \forall v \in \text{VARS}(r)\}$
      $v' \leftarrow \text{argmin}_{v \in \text{VARS}(r)}\{dists(v) \mid v \notin M\}$
      $M \leftarrow M \cup \{v_g \mapsto v'\}$
  CLAUSES(r) $\leftarrow$ CLAUSES(r) $\cup$ MAP-VARS(C, M)
return r

r that most closely resembles its features that is not yet assigned to a different variable. This is a greedy procedure, that finds the best match for each variable individually. Finally, the algorithm performs the variable mapping and merges the clauses of C into r. We omit the details of the merging process as it is straight forward.

Acceptance Criteria We compute a set of well known features for SAT and MaxSAT[5] problems from [26] in order to determine whether to accept a modified instance. We compute the average and standard deviation for each feature across the entire pool of instances (including the instance to modify). An instance is accepted if all of its features are within three standard deviations of the mean. That is, we compare a feature to the cluster center on a feature by feature basis. However, some features do not vary at all in a cluster, meaning they have a standard deviation of 0. In such cases, even small changes to an instance can result in a rejection of all generated instances, although the instance is for the most part within the cluster. Thus, when absolutely no instance could be generated we relax the conditions for features that do not vary within the cluster, and allow them to vary by some epsilon value. We note that in our experiments such instances were still well situated within clusters when measured with the Euclidean distance to the cluster center.

For SAT problems, we extend this acceptance criteria with an execution of the instance with a SAT solver. If the instance is solvable in under 30 seconds it is discarded. We do this because our generation procedure sometimes introduces unsatisfiable components to satisfiable problems that are easily found and exploited by solvers. This clearly breaks the structure of the instance that we are striving to preserve, thus the instance must be discarded. Note that this does not guarantee that the instance will be satisfiable, all we are checking is that the generated instance is not trivially solvable. This issue generally only happens to a couple of instances per generation procedure.

[5] We do note use local search probing features in this work.

5 Computational Evaluation

We perform an evaluation of structure-preserving instance generation on instances from the MaxSAT and SAT competitions. We show that the instances generated using our method preserve structure well enough such that they are effectively solved using the same algorithms as the original instances. To evaluate this, we train an algorithm selection approach on the generated data and evaluate it on the subset of original test instances that were neither part of the training nor the generation. It is assumed that if our generated instances can allow us to train a portfolio to identify the most appropriate solver for the instance at hand, then they successfully embody the same key structures as the original industrial data. For SAT, we also use the Q-score method of [11] to show the quality of our instance generator. All experiments were performed on a cluster of Intel Xeon E5-2670 processors with 4GB of RAM for random instances and 12GB for industrial instances for MaxSAT (as industrial MaxSAT instances are very large), and 4GB of RAM for all SAT instances.

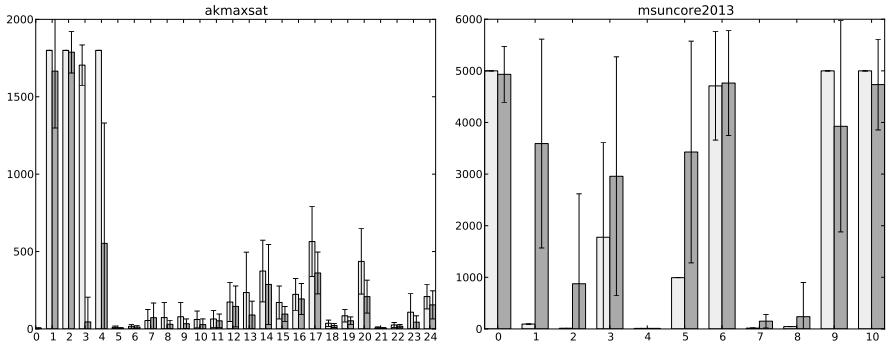
5.1 MaxSAT

For MaxSAT, we evaluate our technique on both the random as well as industrial instances from the MaxSAT 2013 competition [7]. Using the random dataset in addition to the industrial dataset shows that our method can be used for any group of similar instances, even though our main target is industrial instances. Here we generate our datasets according to a manually established similarity measure based on the filenames of the instances. For each pool of instances, we perform 10 instance generations for each instance of the pool with a different random seed. For the experiments presented in this section, we limit each generation attempt to 25 destroy/repair iterations. This process generated 5,306 instances based off of 378 random instances, and 2,606 instances based off of 42 industrial instances[6]

One measure to ensure the quality of the generated instances is to compare the runtime of a solver on both the original and the new dataset. Should the runtime performance of an algorithm be similar on both the original and the new dataset, we can conclude that the new dataset is similar to the old one. This is a desirable property for our instance generator, and is based on a fundamental argument on which algorithm portfolios are built and what makes them so successful in practice: that a solver/algorithm performs analogously well or poorly on instances that are similar.

Figure 2 shows the average solution time of the original instances and their generated counterparts for the akmaxsat and msuncore2013 solvers for several clusterings of instances on the random and industrial datasets, respectively. The clusters were generated based on the categories of the instances. The solutions times are comparable for both random and industrial instances, with the exception of clusters 2 and 5 on the industrial dataset in which the generated instances are too hard. This runtime performance similarity strongly indicates that our generator preserves the structure of instances during generation.

[6] The MaxSAT 2013 dataset contains 55 instances, but we remove instances over 110 MB after performing unit propagation, as SpIG cannot fit them in RAM.



(a) Akmaxsat on the random dataset. (b) Msuncore2013 on the industrial dataset.

Fig. 2. The average solution time in CPU seconds and standard deviation for each cluster in terms of original (left bar, light gray) and generated (right bar, dark gray) instances.

Note that by comparable runtimes, we do not mean identical, which would be an undesirable quality since generated instances should be slightly different from the originals. Furthermore, even when running a solver on the same instance runtimes can vary. The results displayed for the industrial dataset are somewhat noisy, due to the fact that very few original instances exist as a basis for comparison. For example, if we had more instances for cluster 2 (in reality we only have a single instance), it could very well be the case that they are hard to solve as well, but the one instance we have turned out to be solved through a smart (or lucky) branching decision by msuncore2013.

Another important result of our CPU runtime experiments is that "industrial" solvers perform well on our generated industrial instances, whereas "random" solvers tend to timeout. The opposite is also true; when we run an industrial solver on our generated random instances the industrial solvers tend to timeout, but the random solvers perform well. This means that our instance generation framework is able to preserve instance structure nearly regardless of what type of instance it is used on. We note, however, that we do not intend for our instance generator to be used on random or crafted instances, as perfectly good generators already exist for these categories of instances. We show results from these categories only to serve as an evaluation of the overall approach.

One might not even expect our generator to work at all on random instances, as they tend to have little structure. We believe the effectiveness of our approach for such instances is simply due to random changes to a random instance not having a huge effect. Industrial instances (or any instance with some kind of global/local structure), however, require an approach like the one we provide so that generated instances do not get malformed through completely random changes.

Our final experimental comparison on the MaxSAT dataset observes the effect of training a simple portfolio on only the generated instances as opposed to the original ones. Table 1 shows the performance of a portfolio trained and evaluated using leave-one-out cross validation. Note here that for evaluating the generated dataset, none of the instances generated from the test instances were included in the training set. We compare the performance of only using the overall best solver to a portfolio that uses either a random forest or a support vector machine (SVM) to predict the runtime of each

	Original		Generated	
Model	Average time	Unsolved	Average time	Unsolved
Best Single	735	2	735	2
Random Forest	988	5	599	2
SVM (radial)	734	2	591	1
VBS	184	0	184	0

Table 1. Comparison of a portfolio trained (leave-one-out) on only the original MaxSAT instances, and one that is trained on the generated instances. The average time is given in seconds.

solver, selecting the solver with the best expected performance. There are of course a plethora of other popular and more powerful portfolio techniques that could be used and compared, but random forests and SVMs are readily available to anyone and have been previously shown to be effective for runtime prediction, and here we show that even they are able to perform well in our case. The virtual best solver (VBS) gives the performance of an oracle that always picks the fastest solver. Due to the limited training set, the best the portfolio can do is match the performance of a single solver. However, if we train the same solvers on the generated instances and evaluate on the original instances, we are able to see improvements over the best solver, meaning the extra generated instances provide value to the portfolio approach. Our generator is able to help fill in gaps between training instances in the feature space, allowing learning algorithms to avoid having to make a guess as to which algorithm will work best within such gaps. Instead, learning approaches have data on the instances in these gaps and can make informed decisions for their portfolio.

5.2 SAT

We next use the instances from the 2013 SAT competition [8] to conduct further experiments. For the competition, these instances are split into three categories each consisting of 300 instances: application (industrial), crafted, and random. And although this competition has taken place annually for the last decade, it is important to note that the majority of the industrial instances repeat each year, which means our experiments use most of the instances available.We first evaluate our generated dataset using the Q-score technique from [11]. We then use an algorithm selection approach to confirm the usefulness of our generator.

Algorithm selection with CSHC Due to the increased number of available instances over the Max-SAT scenario, we apply a more automated technique for grouping SAT instances for generation. In MaxSAT, the instances were grouped based on a manually defined similarity metric associated with the filenames. For SAT, however, the industrial instances are clustered based on their features using the g-means algorithm from [17], an approach that automatically determines the best number of clusters based on how Gaussian distributed each cluster is. Limiting the minimum size of a cluster to 50 resulted in a total of 7 clusters. We typically use 50 instances for a cluster to ensure we have a reasonable statistical evidence that a particular solver works better than another. We sample 15% of the instances from each cluster to compose our training set, and to form the subsets of similar instances for generation.

We perform a more standard portfolio evaluation of the generator for SAT since so many instances are available. For this evaluation we use the top solvers from the

	Average	PAR10	Solved
Best Single	453	3,872	157
Random (300)	541	5,107	144
Crafted (300)	386	4,090	154
Industrial (46)	348	3,426	161
Generated (300)	502	3,463	162
Generated (1,500)	437	3,049	166
VBS	364	364	195

Table 2. Comparison of a portfolio evaluated on 195 Industrial SAT instances when trained on either 300 randomly generated instances, 300 crafted instances, a subset of 46 industrial instances, 300 generated instances, or 1500 generated instances.

2013 SAT Competition: glucose, glue bit, lingeling 587f, lingeling aqw, MIPSat, riss3g, strangenight, zenn, CSHCapplLC, and CSHCapplLG.

Our test set for all the subsequent experiments is the collection of 254 industrial instances remaining after the subset of training instances is removed. This list is then further reduced by removing those instances for which no solver finds a solution within 1,800 seconds. This leaves a total of 195 instances. We compare the portfolios based on three metrics: average time without timeouts (Average), PAR10, and number of instances solved (Solved). PAR10 is a penalized average where a timeout counts as having taken 10 times the timeout time.

For our underlying portfolio technique, we utilized CSHC [23], the technique that won the "Open Track" at the 2013 SAT Competition and was behind the portfolio of ISAC++ [6] that won the MaxSAT Evaluation in 2013 and 2014. The core premise of this portfolio technique is a branching criteria for a tree that ensures that after each partition, the instances assigned to sub-node maximally prefer a particular solver other than the one used in the parent node. Training a forest of such trees then ensures the robustness of the algorithm.

The results of the experiments are presented in Table 2. The best standalone solver is Lingeling aqw, which solves a total of 157 instances. We trained the portfolio on a variety of training sets: 300 random instances, 300 crafted instances, 46 industrial instances, 300 generated (industrial) instances and 1500 generated (industrial) instances. Not surprisingly, training on random or crafted instances does not perform well. In both cases, less instances can be solved than just using the best single solver, and the PAR10 scores are significantly higher. This further confirms a well-known result in the algorithm selection literature that training and test sets need to be similar in order for the learning algorithm to be successful. We include these results to emphasize the fact that if our generated instances were significantly different from the datasets they were generated from, we would expect similarly bad performance on the test set. Indeed, using even just 46 industrial instances already results in better performance than the best single solver in terms of average time, PAR10 and the number of instances solved.

Using 300 generated industrial instances shows similar performance to the original industrial instances in terms of the number of the PAR10 score and number of instances solved, although the average runtime is higher. This is already enough evidence to further confirm that our instance generation routine is successful at preserving instance structures in SAT, as in Max-SAT. However, because we are not limited by the number

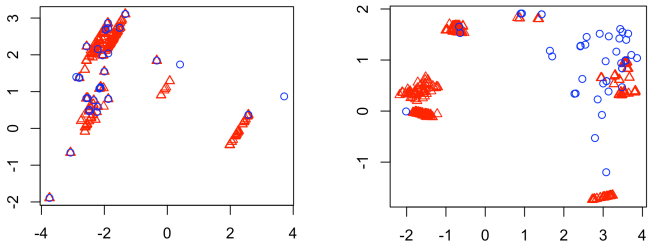
of instances we generate, we can create much larger training samples. We therefore evaluate our portfolio trained on 1,500 generated instances and observe that 166 instances can be solved, 5 more than with the original training set. In a competition setting, this improvement is often the difference between first place and finishing outside the top three solvers. This provides further support that our approach fills in gaps in the instance feature space, and that this provides critical information to selection algorithms that improves their performance.

***Q*-score** The Q-score, introduced by Bayless et al. in [11], provides a mechanism for assessing whether or not a dataset of instances can act as a proxy for some other set of instances. In other words, using the Q-score we can check whether the instances we generate share similar properties with the original dataset of industrial instances. The score is based on the performance of parameters found through algorithm configuration using a method like [5] or [18]. We use SMAC [18] as it was previously used for calculating the Q-score in [11]. We configure the Lingeling and Spear solvers each three times for five days on the same 1500 generated instances used in our algorithm selection experiments and all 300 original industrial instances, which we label S and T, respectively. Adopting the notation of [11] (which we refer to for full details), the Q-score is computed by $c(A(\theta'_T), T)/c(A(\theta'_S), T)$, where c is the PAR10 score of a parameterization on the specified dataset, A specified an algorithm configuration, and θ'_T and θ'_S are the best performing configurations (on the test set) of all tuned configurations and on the generated set, respectively.

We found the Q-score 0.9177 for lingeling and 0.9978 for Spear on our generated instances. We note that 1.0 is the best possible score. This indicates that the datasets we generate lead to high quality parameter configurations that generalize to the original instances. Interestingly the best parameter configuration for Lingeling on the test set was one of the parameterizations trained on the generated instances. However, its training set evaluation was beaten by another parameterization, thus we do not use it in the calculation of the Q-score for the set S. This is especially noteworthy given that our generated instance set is not even based on all of the industrial instances, but is nonetheless being compared to parameters specifically tuned on all 300 industrial instances.

5.3 Structure Comparison

As a final evaluation of our instance generation methodology we present a comparison of the original and generated instances when their features are projected into a two dimensional space. We do this using a standard principal component analysis (PCA). Figure 3 presents the results for both the MaxSAT and SAT datasets. The figure shows that there is not a perfect matching between the generated and original instances. While future work can focus on reducing the spread between these instances, we note that a perfect matching is not desirable as we do not want exact replicas of our instance pool. Instead, we want to cover a range of scenarios of similar instances, which can be seen in many parts of the projection. This subsequently leads to a better trained portfolio. Furthermore, note that the generated instances tend to be close to their original counterparts in this projected space. This means that although they are not completely identical, the generated instances are still fairly representative of their originals.



(a) MaxSAT (b) SAT

Fig. 3. Projection of the instances into 2D using PCA on their features. Original training instances are represented as blue circles, the generated instances are represented by red triangles.

6 Conclusion and Future Work

One of the current key problems in solver development is the limited number of instances on which algorithms can be compared. This is especially the case for industrial instances, where datasets are extremely limited and difficult to expand. To remedy this, this paper presented a novel methodology for generating new instances with structures similar to a given dataset. We then demonstrated the quality of the generated datasets by training portfolios on them and evaluating them on the original instances. This showed that not only do the instances have similar structures as the originals, but that those structures also allow a portfolio (and algorithm configuration) to correctly learn the best solver for provided instances. For future work, will evaluate our instance generation framework on other types of problems, such as CSPs and MIPs, as well as explore how to improve the generated instances' coverage of the feature space.

Acknowledgements We thank the Paderborn Center for Parallel Computing for the use of the OCuLUS cluster for the experiments in this paper. Barry O'Sullivan was supported in part by Science Foundation Ireland (SFI) under Grant Number SFI/12/RC/2289.

References

1. Ansótegui, C., Béjar, R., Fernández, C., Mateu, C.: Edge matching puzzles as hard sat/csp benchmarks. In: CP. LNCS, vol. 5202, pp. 560–565 (2008)
2. Ansótegui, C., Bonet, M.L., Levy, J., Li, C.M.: Analysis and generation of pseudo-industrial maxsat instances. In: CCIA. FAIA, vol. 248, pp. 173–184. IOS Press (2012)
3. Ansótegui, C., Giráldez-Cru, J., Levy, J.: The community structure of SAT formulas. In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing (SAT 2012), pp. 410–423. No. 7317 in LNCS, Springer (Jan 2012)
4. Ansótegui, C., Levy, J.: On the modularity of industrial sat instances. In: F., C., Geffner, H., Manyà, F. (eds.) CCIA. FAIA, vol. 232, pp. 11–20. IOS Press (2011)
5. Ansotegui, C., Sellmann, M., Tierney, K.: A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms. In: Gent, I. (ed.) Principles and Practice of Constraint Programming (CP-09). LNCS, vol. 5732, pp. 142–157. Springer (2009)
6. Ansotegui, C., Malitsky, Y., Sellmann, M.: MaxSAT by Improved Instance-Specific Algorithm Configuration. AAAI (2014)
7. Argelich, J., Li, C.M., Manyà, F., Planes, J.: Eighth Max-SAT evaluation (2013)
8. Balint, A., Belov, A., Heule, M., Järvisalo, M.: Proceedings of SAT competition 2013; solver and benchmark descriptions. Tech. rep., University of Helsinki (2013)

9. Barták, R.: On generators of random quasigroup problems. In: Recent Advances in Constraints, LNCS, vol. 3978, pp. 164–178. Springer (2006)
10. Bastian, M., Heymann, S., Jacomy, M.: Gephi: An open source software for exploring and manipulating networks. In: AAAI Conference on Weblogs and Social Media (2009)
11. Bayless, S., Tompkins, D., Hoos, H.: Evaluating instance generators by configuration. In: Pardalos, P.M., Resende, M.G.C., Vogiatzis, C., Walteros, J.L. (eds.) LION 8. LNCS, vol. 8426, pp. 47–61. Springer (2014)
12. Bejar, R., Cabiscol, A., Manya, F., Planes, J.: Generating hard instances for MaxSAT. In: International Symposium on Multiple-Valued Logic (ISMVL 2009). pp. 191–195 (May 2009)
13. Burg, S., Kottler, S., Kaufmann, M.: Creating industrial-like sat instances by clustering and reconstruction. In: SAT 2012, pp. 471–472. Springer (2012)
14. Dinur, I., Goldwasser, S., Lin, H.: The computational benefit of correlated instances. In: Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science. pp. 219–228. ACM (2015)
15. Gent, I.P., Hoos, H.H., Prosser, P., Walsh, T.: Morphing: Combining structure and randomness. In: Hendler, J., Subramanian, D. (eds.) AAAI. pp. 654–660 (1999)
16. Gomes, C.P., Selman, B.: Problem structure in the presence of perturbations. In: Kuipers, B., Webber, B.L. (eds.) AAAI. pp. 221–226 (1997)
17. Hamerly, G., Elkan, C.: Learning the K in K-Means. In: Neural Information Processing Systems (NIPS) (2003)
18. Hutter, F., Hoos, H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: LION 5, pp. 507–523. Springer (2011)
19. Kadioglu, S., Malitsky, Y., Sellmann, M., Tierney, K.: ISAC – Instance-Specific Algorithm Configuration. In: ECAI. FAIA, vol. 215, pp. 751–756. IOS Press (2010)
20. Katsirelos, G., Simon, L.: Eigenvector centrality in industrial sat instances. In: Milano, M. (ed.) CP. LNCS, vol. 7514, pp. 348–356. Springer (2012)
21. Krzakala, F., Zdeborová, L.: Hiding quiet solutions in random constraint satisfaction problems. Physical review letters 102(23), 238701 (2009)
22. Lopes, L., Smith-Miles, K.: Generating applicable synthetic instances for branch problems. Operations Research 61(3), 563–577 (2013)
23. Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Algorithm Portfolios Based on Cost-Sensitive Hierarchical Clustering. IJCAI (2013)
24. Motoki, M.: Test instance generation for MAX 2SAT. In: van Beek, P. (ed.) CP05, pp. 787–791. No. 3709 in LNCS, Springer (Jan 2005)
25. Nallaperuma, S., Wagner, M., Neumann, F.: Parameter prediction based on features of evolved instances for ant colony optimization and the traveling salesperson problem. In: PPSN XIII, LNCS, vol. 8672, pp. 100–109. Springer (2014)
26. Nudelman, E., Leyton-Brown, K., Hoos, H.H., Devkar, A., Shoham, Y.: Understanding random sat: Beyond the clauses-to-variables ratio. In: CP 2004, pp. 438–452. Springer (2004)
27. Pari, P.R., Lin, J., Yuan, L., Qu, G.: Generating 'random' 3-sat instances with specific solution space structure. In: McGuinness, D.L., Ferguson, G. (eds.) AAAI. pp. 960–961 (2004)
28. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: CP 1998, pp. 417–431. Springer (1998)
29. Slater, A.: Modelling more realistic sat problems. In: McKay, B., Slaney, J.K. (eds.) Australian Joint Conference on AI. LNCS, vol. 2557, pp. 591–602. Springer (2002)
30. Smith-Miles, K., Bowly, S.: Generating new test instances by evolving in instance space. Computers & Operations Research 63, 102 – 113 (2015)
31. Smith-Miles, K., van Hemert, J.: Discovering the suitability of optimisation algorithms by learning from evolved instances. Ann Math Artif Intell 61(2), 87–104 (2011)
32. Van Gelder, A., Spence, I.: Zero-one designs produce small hard SAT instances. In: Strichman, O., Szeider, S. (eds.) SAT 2010, pp. 388–397. No. 6175 in LNCS, Springer (Jan 2010)