Characteristic Subsets of SMT-LIB Benchmarks

Jan Jakubův*[1,2], Mikoláš Janota[2] and Andrew Reynolds[3]

[1]*University of Innsbruck, Innsbruck, Austria*

[2]*Czech Technical University in Prague, Prague, Czechia*

[3]*The University of Iowa, Iowa City, USA*

Abstract

A typical challenge faced when developing a parametrized solver is to evaluate a set of strategies over a set of benchmarking problems. When the set of strategies is large, the evaluation is often done with a restricted time limit and/or on a smaller subset of problems in order to estimate the quality of the strategies in a reasonable time. Firstly, considering the standard SMT-LIB benchmarks, we ask the question how much the time evaluation limit and benchmark size can be restricted to still obtain reasonable performance results. Furthermore, we propose a method to construct a benchmark characteristic subset which faithfully characterizes all benchmark problems. To achieve this, we collect problem performance statistics and employ clustering methods. We evaluate the quality of our benchmark characteristic subsets on the task of the best cover construction, and we compare the results with randomly selected benchmark subsets. We show that our method achieves smaller relative error than random problem selection.

Keywords

Benchmarking, Satisfiability Modulo Theories, Machine Learning, Clustering, Benchmark Characteristic Subsets

1. Introduction

Optimizing the performance of a Satisfiability Modulo Theories (SMT) solver, like CVC4 [1] or Z3 [2], on a large set of problems, like the SMT-LIB library [3, 4], is a time-consuming task. This is because of a large number of problems in SMT-LIB (over 100000). Even with the help of parallelization, it takes several hours to evaluate a single strategy over all benchmark problems with a standard time limit (like 60 seconds). The situation gets even worse when more than one strategy, or a parametrized strategy with different arguments, needs to be evaluated. Then it is common practice to restrict the evaluation to a subset of problems and/or to decrease the evaluation time limit. In this paper, we try to address the question how much can the evaluation be restricted so that the results are still plausible. Furthermore, we propose a method to construct a small *benchmark characteristic subset* that would faithfully represent all benchmark problems for the sake of evaluation.

Within a large problem library, one can expect a large number of syntactically similar problems or problems with similar performance with respect to many strategies. Identification of similar problems could help us to speed up the evaluation, as we can select a single representative

---

from each similarity class. To identify classes of similar problems, we propose to perform short evaluation runs of several CVC4 strategies and to collect run-time statistics. Then, problems with similar statistics might be considered similar. To evaluate this approach, we experiment with the best cover construction, that is, a selection of strategies with the best performance on a given set of problems. We construct the best cover on a smaller benchmark subset and we compare its performance with the optimal cover constructed on all problems. This gives us a quality measure of problem subset selection.

The paper is structured as follows. Section 2 describes the proposed method of identification of similar problems based on performance statistics, and how to construct a benchmark characteristic subset. Section 3 describes the best cover construction task which is used for empirical evaluation of the quality of constructed subsets in Section 4. We conclude in Section 5.

2. Benchmark Characteristic Subsets by Clustering

Restricting the evaluation of a solver to a smaller random subset of all benchmark problems is a common and effective method used by developers. Considering a large set of benchmark problems, like the SMT-LIB library [3, 4], one can expect a significantly large number of very similar or even duplicate problems. Random selection does not take problem similarities into account and thus can lead to unnecessary and duplicate computations. In this section, we propose a simple method for selecting a characteristic subset of benchmark problems based on *performance features* and *clustering algorithms*. The desired property of this characteristic subset is that it faithfully characterizes all benchmark problems. That is, that any development, like parameter tuning or scheduler construction, performed on this subset yields similar results as the same development performed on all benchmark problems, but faster.

Clustering algorithms [5] are capable of dividing a set of entities into disjoint subsets, called *clusters*, such that similar entities end up in the same cluster. We propose to apply clustering algorithms to create a benchmark characteristic subset. Bearing in mind that the benchmark characteristic subset should allow us to avoid duplicate computations, we cluster the benchmark problems and we create the characteristic subset by selecting one problem from each cluster.

To employ clustering algorithms, the entities, in our case benchmark problems, need to be represented by numeric *feature vectors*. We could create the feature vectors from the syntactic structure of problems, similarly to ENIGMA [6, 7, 8] clause and problem features for first-order logic, but in this work we propose an alternative to compute the feature vectors from runtime statistics of short probing solver runs. In particular, we run CVC4 solver on all SMT-LIB benchmark problems with small abstract resource limit[1]. Independently on whether the problem is solved or not, we collect the runtime statistics, and we extract selected statistic values (see Appendix A) to construct the *performance feature* vectors of the problem. We use only statistic values which always lead to the same value with repeated runs, in particular, we do not use any real-time measurements. Moreover, we repeat this process with different CVC4 strategies to obtain longer and more descriptive feature vectors. Our performance features are similar to *dynamic features* found in research literature [9, 10].

---

[1]We run CVC4 with argument `-rlimit=10000` to limit the resources (roughly the number of SAT conflicts).

Once the performance feature vectors for all benchmark problems are constructed, we employ the k-means [11] clustering algorithm, one of the most common clustering methods. The k-means algorithm divides benchmark problems into k disjoint clusters, hence we set k to the desired size of the benchmark characteristic subset. Briefly, the k-means algorithm works as follows. Firstly, k points in the vector space called *cluster centroids* are selected. Different variants of the k-means algorithm use different centroid initialization methods, with random initialization being one of the common variants. In the next step, all distances between centroids and feature vectors are computed, and each feature vector is assigned to the closest cluster centroid. After that, the average vector of the vectors assigned to the same cluster is computed for each cluster, and the cluster centroid is moved to the computed average. This process is iterated until the centroids stop moving.

The above gives us the following method for the construction of the benchmark characteristic set. We construct the performance feature vectors for all benchmark problems by extracting runtime statistics from short probing solver runs. As different features might have values of different units, we normalize the vectors by dividing each feature by the standard deviation across all feature values. Then we employ the k-means algorithm and construct k problem clusters and their centroids. From each cluster, we select the problem whose performance feature vector is the closest to the cluster centroid. Thusly selected problems form the benchmark characteristic subset of size k.

3. Evaluation Tasks: Greedy and Exact Covers

To evaluate the quality of the benchmark characteristic subsets from Section 2, we consider the task of the best cover construction. Let P be the set of benchmark problems and let S be a set of strategies. Suppose we evaluate all strategies S over problems P. The *coverage* of S over P, denoted coverage(S,P), is the count of problems solved by strategies S.

$$\text{coverage}(S,P) = \left| \bigcup_{s \in S} \{p \in P : \text{strategy } s \text{ solves problem } p\} \right|$$

The *best cover* of strategies S over P of size n is the subset of $S_0 \subseteq S$ with $|S_0| \leq n$ and with the highest possible coverage(S_0, P).

Suppose we construct the best cover $S_0 \subseteq S$ based only on a partial evaluation of strategies S over a subset P_0 of problems P. This is typically done when the set of strategies and the set of problems are large. We can then compare the performance of the best cover constructed on P_0 with the optimal performance of the best cover constructed with the full evaluation over P. The performance of the two covers should be similar, if P_0 faithfully characterizes P. More formally, let us fix strategies S and the cover size n. Let S_0 be the best cover over P_0 and let S_{best} be the best cover over P. We will measure the difference between their coverages coverage(S_0, P) and coverage(S_{best}, P) over all problems P. We usually do not have all strategies evaluated over all problems, but for the sake of the evaluation in this paper we shall compute them.

We consider two methods to construct the best covers. First, an approximative but fast method called *greedy cover* (Section 3.1). Second, an exact but a bit slower best cover construction using an ILP solver (Section 3.2).

```
1  FUNCTION greedy(P, S, n)
      INPUT   : set of problems P, set of strategies S, size n
      OUTPUT : a subset of S of size n approximating the best cover
2     G ← ∅
3     WHILE |G| < n and ((∃p ∈ P)(∃s ∈ S) : strategy s solves problem p) DO
4        FOR s ∈ S DO              // compute the problems solved by each strategy s ∈ S
5           ⌊ R_s ← {p ∈ P : strategy s solves problem p}
6        g ← argmax_{s∈S} (|R_s|)        // obtain the best strategy on current problems P
7        P ← P \ R_g                              // remove the problems solved by g …
8        G ← G ∪ {g}
9     RETURN G
```

Algorithm 1: Greedy cover algorithm.

3.1. Greedy Cover

The greedy cover algorithm is an approximative method for the construction of the best cover of strategies S over problems P of size n. The algorithm greedy(*P,S,n*) (see Algorithm 1) proceeds as follows. It starts with an empty cover G and it finds the strategy $g \in S$ which solves most problems from P. The strategy g is added to G and the problems solved by g are removed from P. This process is iteratively repeated until G reaches the desired size n or no strategy can solve any of the remaining problems.

We will measure the relative error of constructing the greedy cover (of size n) by a partial evaluation of strategies S over a subset P_0 of all problems P using the standard formula below. The experimental evaluation of greedy cover construction on random subsets and on benchmark characteristic subsets based on performance features is given in Section 4.

$$\text{error}(P_0,n) = 100 \cdot \left| 1 - \frac{\text{coverage(greedy}(P_0,S,n),P)}{\text{coverage(greedy}(P,S,n),P)} \right|$$

3.2. Exact Cover

Apart from the approximative greedy cover construction, we also consider an exact best cover construction method based on the representation of the best cover problem by an Integer Linear Programming (ILP) formulation. The ILP problems are solved by the Gurobi solver [12].

We consider a maximal set cover formulation, i.e., without taking into account the solving time. This means that given a set of strategies S, a set of solved problems P_s by each strategy $s \in S$, and the size of the cover n, the goal is to select a set of strategies $S' \subseteq S$ with $|S| = n$ that maximizes the total number of instances solved, i.e., maximizing the cardinality of $\bigcup_{s \in S'} P_s$.

We remark that the exact cover maximization is typically only slightly better than the greedy cover. However, interestingly, Gurobi has proven to be extremely efficient on these problems; typically we observe solving times around 1 s.
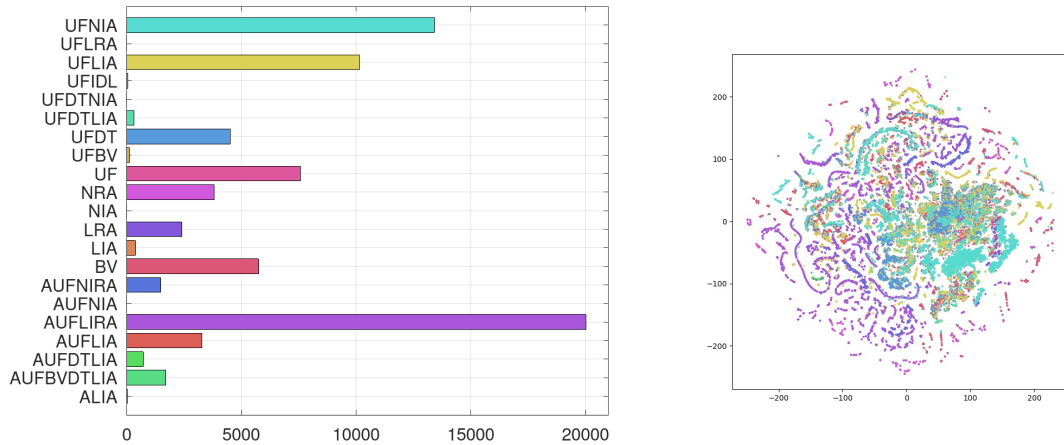
Figure 1: Representation of different logics in SMT-LIB benchmark problems (left) and visualization of problem similarities based on performance features (right).

4. Experimental Evaluation

In this section, we describe the experimental setup in Section 4.1. We experiment with the best cover construction of random benchmark subsets in Section 4.2. In Section 4.3, we discuss reasonable restrictions of the evaluation time limit. Mainly, in Section 4.4, we evaluate the quality of benchmark characteristic subsets constructed using performance features and the k-means clustering method introduced in Section 2.

4.1. SMT-LIB Experiments Setup

For our experiments, we consider the quantified fragment of the SMT-LIB library [3, 4]. This gives us altogether 75814 problems from 21 different logics. We consider 23 different CVC4 [1] strategies (see Appendix B) and Z3 [2] in the default mode as another strategy. We evaluate all strategies over all problems with a 60 seconds time limit. Any solver run terminating before the time limit with the result sat or unsat is considered successful and the problem is considered solved by the respective strategy. Computing this database of results took around 6 days of real time.[2] Once we have this large database of results, we can easily compute partial results on problem subsets or with a smaller time limit.

Figure 1 graphically depicts benchmark problem properties. The bar graph on the left shows the count of problems from different logics. The plot on the right visualizes the performance feature vectors of the problems described in Section 2. The plot is produced by dimensionality reduction to 2 dimensions by the t-SNE [13] algorithm. Each problem is represented by one point colored with the color of its logic corresponding to the left bar graph. Problems with similar feature vectors should appear close to each other in the plot. Here we just note that

---

[2]All the experiments are done on a server with 28 hyperthreading Intel Xeon CPU @ 2.3GHz cores and 256 GB of memory.
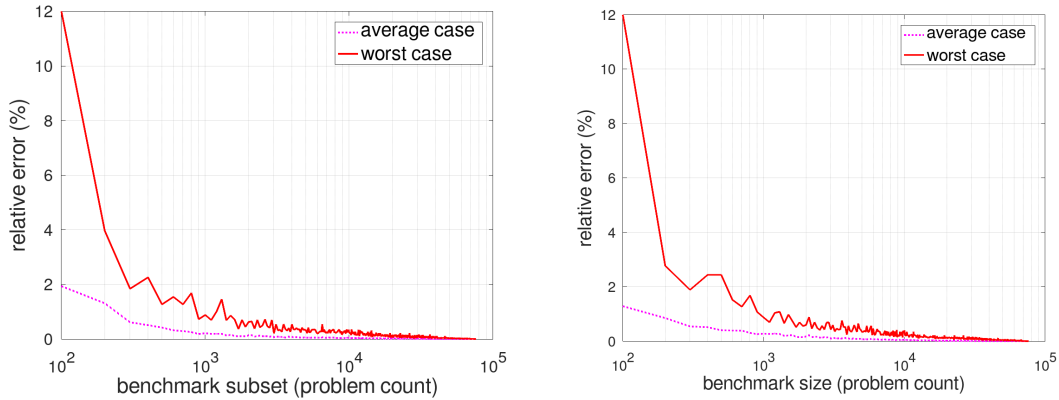
Figure 2: Relative error of greedy (left) and exact (right) covers on random subsets.

nearby points tend to have the same color. That is, the problems from one logic have similar performance feature vectors, which is to be expected.

4.2. Random Benchmark Subsets

In this section, we present an experiment on the best cover construction on randomly selected subsets of benchmark problems. From all 75814 benchmark problems, we construct random subsets of different sizes, ranging from 100 up to 75800 with step 100. Moreover, we construct 10 different collections of these subsets to measure the effect of random selection. On every benchmark subset, we construct the greedy and exact covers of the 14 different strategies, called *validation strategies* (see Section 4.1). We construct covers of different sizes ($n \in \{1, 2, \ldots, 10\}$). We evaluate the constructed covers on all benchmark problems and we measure the error as described in Section 3.1 and Section 3.2.

Figure 2 presents the error for the greedy cover (left) and exact cover (right) construction. For every random subset size, we plot the worst case error (red solid line), and the average error (dotted line). Given 10 random subset collections and 10 different cover sizes, each value is computed from 100 relative error values.

We can see that with benchmark subsets bigger than 1000 problems (roughly 1.3% of benchmark problems), we obtain less than 2% error even in the worst case. With subsets bigger than 10000, the worst case error drops below 0.5%. From the relatively big difference between the worst case and the average, we can conclude that the accuracy of the benchmark approximation by a random subset depends considerably on the coincidence of random selection. In Section 4.4 we show that performance features can help us reduce this dependence by constructing better benchmark characteristic problems.

4.3. Time Limit Evaluation

In this section, we present basic statistics on the solver runtime. In particular, we are interested in how many problems are solved with a decreased evaluation time limit. This information can help developers to set time limits for evaluation experiments.
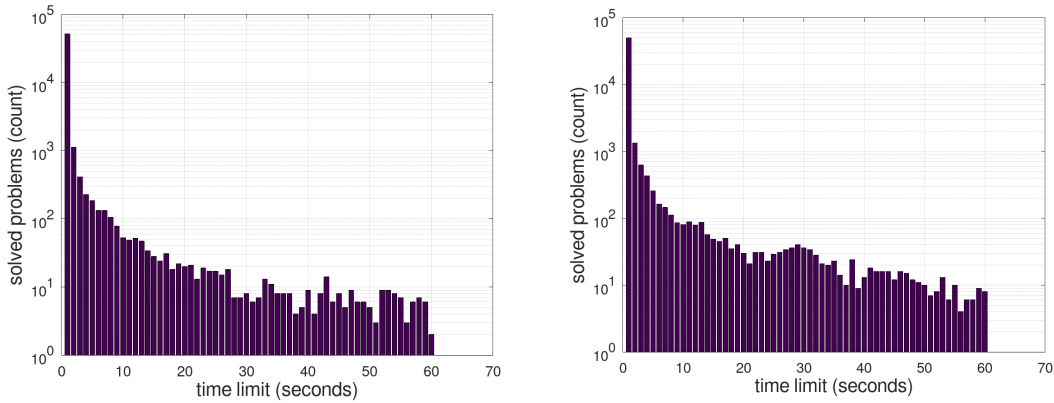
Figure 3: Number of problems solved in specific time by cvc_1 (left) and cvc_{16} (right).

For each of the 24 considered strategies, we measure how many problems are solved in the n-th second, that is, the count of problems solved with runtime between $n - 1$ and n. Figure 3 shows the results for two representative strategies, namely, the strategy that solves the most problems in the first second (left) and the strategy that solves the most problems with runtime greater than 1 second. The corresponding strategies (cvc_1 and cvc_{16}) can be found in Appendix B.

From Figure 3, we can see that the majority of problems are solved within the first second of runtime, and this is the case with all evaluated strategies. Note the logarithmic y-axis in the figure. We have conducted experiments with greedy cover construction, similar to the experiments in Section 4.2. The experiments show that the greedy cover constructed with the evaluation restricted to the time limit of 1 second shows less than 0.25% error, when compared with greedy covers constructed with the full time limit of 60 seconds. With the time limit of 10 seconds, we reach the error less than 0.15% and with the limit of 30 seconds the error drops below 0.05%.

4.4. Performance Features Benchmark Characteristics

In this section, we evaluate the quality of benchmark characteristic subsets constructed with performance features and k-means clustering from Section 2. We compare the relative error of the construction of the best covers on benchmark characteristic subsets and on random benchmark subsets. Out of the 24 strategies considered in this experiment (see Section 4.1), we use 10 CVC4 strategies to construct the performance features of benchmark problems. We use only the remaining 14 strategies to construct the best covers to provide an independent validation set of strategies.

The performance features are computed by short CVC4 runs with limited resources as described in Section 2. Computing the performance features took less than 2 hours. We construct benchmark characteristic subsets[3] of different sizes corresponding to the sizes of

---

[3]We use `scipy.org`'s implementation of k-means, namely function `scipy.cluster.vq.kmeans2` with parameter `minit` set to `points`.
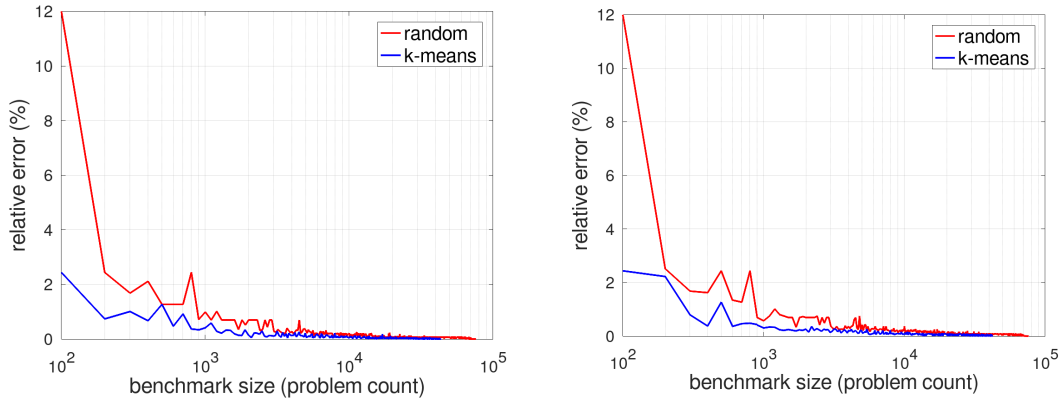
Figure 4: Worst case relative errors when constructing the greedy (left) and the exact (right) cover on random subsets and on benchmark characteristic subsets (k-means).

random benchmark subsets from Section 4.2. In this case, however, we do not construct 10 different collections but only one subset for each size. On the benchmark characteristic subsets, we compute both greedy and exact covers of different sizes ($n \in \{1, 2, \ldots, 10\}$), and we measure the relative error as in Section 4.2.

Figure 4 presents the worst case relative errors for greedy cover (left) and exact cover (right) constructions. The red lines correspond to the relative error on random benchmark subsets, that is, they correspond to the red lines from Figure 2, but this time the results are computed only on the 14 validation strategies. The blue lines correspond to relative errors on the benchmark characteristic subsets. We can see that in both cases, the relative errors on the benchmark characteristic subsets are significantly lower, especially for smaller benchmark subsets. Even with the smallest subset of size 100, we get almost 2% worst case error. Moreover, from size 600 we obtain the worst case error below 1%. Furthermore, the error on benchmark characteristic subsets approaches the average error on random subsets (see the dotted lines in Figure 2). This suggests that the construction of benchmark characteristic subsets is less coincidental than random selection.

5. Conclusions and Future Work

We propose and evaluate a method of constructing a benchmark characteristic subset that represents the whole benchmark. We provide empirical evidence that our method, based on clustering of problem performance feature vectors, gives better results than a random benchmark sampling. To evaluate our method, we use the task of best cover construction. However, we believe that our method shall prove useful for many other tasks. Furthermore, we provide an experimental evaluation of how restricting the benchmark size and evaluation time limits affect the solver performance. The characteristic subsets of quantified problems from SMT-LIB library computed using the k-means clustering method are available for download.

https://github.com/ai4reason/public/blob/master/SMT2021

Hence SMT users can directly benefit from the results published in this paper by speeding up their solver evaluation.

As future work, we would like to experiment with different methods of measuring the quality of benchmark characteristic subsets, other than the best cover construction. Moreover, we would like to employ different clustering algorithms than k-means. Finally, we would like to propose different methods of constructing problem characteristic feature vectors, for example, extracting features directly from the problem syntax.

Acknowledgments

References

[1] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, C. Tinelli, CVC4, in: CAV, volume 6806 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 171–177.

[2] L. M. de Moura, N. Bjørner, Z3: an efficient SMT solver, in: TACAS, volume 4963 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 337–340.

[3] C. Barrett, P. Fontaine, C. Tinelli, The Satisfiability Modulo Theories Library (SMT-LIB), www.SMT-LIB.org, 2016.

[4] C. Barrett, A. Stump, C. Tinelli, The SMT-LIB Standard: Version 2.0, in: A. Gupta, D. Kroening (Eds.), Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK), 2010.

[5] M. Omran, A. Engelbrecht, A. Salman, An overview of clustering methods, Intell. Data Anal. 11 (2007) 583–605. doi:10.3233/IDA-2007-11602.

[6] J. Jakubův, K. Chvalovský, M. Olšák, B. Piotrowski, M. Suda, J. Urban, ENIGMA anonymous: Symbol-independent inference guiding machine (system description), in: IJCAR (2), volume 12167 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 448–463.

[7] J. Jakubův, J. Urban, Enhancing ENIGMA given clause guidance, in: CICM, volume 11006 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 118–124.

[8] J. Jakubův, J. Urban, ENIGMA: efficient learning-based inference guiding machine, in: CICM, volume 10383 of *Lecture Notes in Computer Science*, Springer, 2017, pp. 292–302.

[9] J. P. Bridge, S. B. Holden, L. C. Paulson, Machine learning for first-order theorem proving - learning to select a good heuristic, J. Autom. Reason. 53 (2014) 141–172.

[10] M. Rawson, G. Reger, Dynamic strategy priority: Empower the strong and abandon the weak, in: PAAR@FLoC, volume 2162 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2018, pp. 58–71.

[11] S. P. Lloyd, Least squares quantization in PCM, IEEE Trans. Inf. Theory 28 (1982) 129–136.

[12] L. Gurobi Optimization, Gurobi optimizer reference manual, 2021. URL: http://www.gurobi.com.

[13] G. E. Hinton, S. T. Roweis, Stochastic neighbor embedding, in: NIPS, MIT Press, 2002, pp. 833–840.

A. CVC4 Statistics Used as Performance Features

We use the following statistic keys, obtained by running CVC4 with option `-stats`, to construct problem performance feature vectors.

```
sat::conflicts
sat::decisions
Instantiate::Instantiations_Total
SharedTermsDatabase::termsCount
resource::PreprocessStep
resource::RewriteStep
resource::resourceUnitsUsed
```

B. CVC4 Strategies Used in Experiments

The following 23 CVC4 strategies, described as CVC4 command line arguments, are used in the experiments. The 24th strategy is Z3 in the default mode. The first ten $\{\text{cvc}_1, \ldots, \text{cvc}_{10}\}$ are used to construct the performance feature vectors of problems. The remaining 14 are used as validation strategies.

cvc_1	`–simplification=none –full-saturate-quant`
cvc_2	`–no-e-matching –full-saturate-quant`
cvc_3	`–relevant-triggers –full-saturate-quant`
cvc_4	`–trigger-sel=max –full-saturate-quant`
cvc_5	`–multi-trigger-when-single –full-saturate-quant`
cvc_6	`–multi-trigger-when-single –multi-trigger-priority –full-saturate-quant`
cvc_7	`–multi-trigger-cache –full-saturate-quant`
cvc_8	`–no-multi-trigger-linear –full-saturate-quant`
cvc_9	`–pre-skolem-quant –full-saturate-quant`
cvc_{10}	`–inst-when=full –full-saturate-quant`
cvc_{11}	`–no-e-matching –no-quant-cf –full-saturate-quant`
cvc_{12}	`–full-saturate-quant –quant-ind`
cvc_{13}	`–decision=internal –simplification=none –no-inst-no-entail \\`
	`–no-quant-cf –full-saturate-quant`
cvc_{14}	`–decision=internal –full-saturate-quant`
cvc_{15}	`–term-db-mode=relevant –full-saturate-quant`
cvc_{16}	`–fs-interleave –full-saturate-quant`
cvc_{17}	`–finite-model-find –mbqi=none`
cvc_{18}	`–finite-model-find –decision=internal`
cvc_{19}	`–finite-model-find –macros-quant –macros-quant-mode=all`
cvc_{20}	`–finite-model-find –uf-ss=no-minimal`
cvc_{21}	`–finite-model-find –fmf-inst-engine`
cvc_{22}	`–finite-model-find –decision=internal`
cvc_{23}	`–full-saturate-quant`