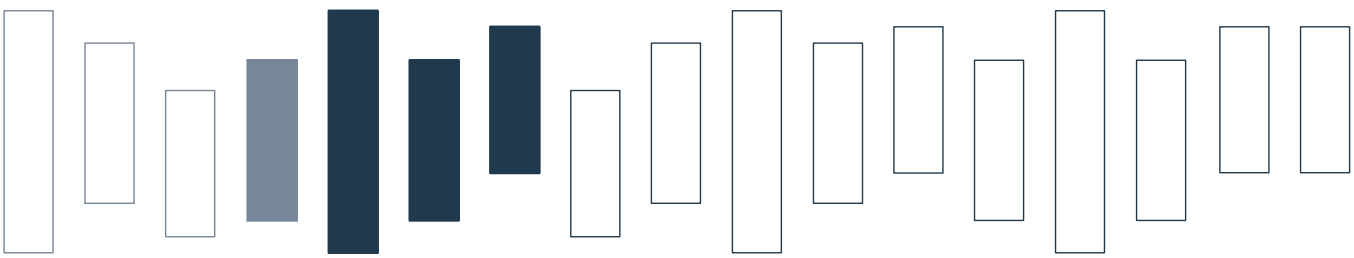

Technical Report AC-TR-18-004

July 2018

Portfolio-Based Algorithm Selection for Circuit QBFs

Holger H. Hoos, Tomáš Peitl, Friedrich Slivovsky, and Stefan Szeider

This is the authors' copy of a paper that is to appear the proceedings of CP 2018, the 24th International Conference on Principles and Practice of Constraint Programming, Lille, France, August 27–31, 2018. LNCS, Springer Verlag, 2018.

www.ac.tuwien.ac.at/tr

Portfolio-Based Algorithm Selection for Circuit QBFs*

Holger H. Hoos[1], Tomáš Peitl[2], Friedrich Slivovsky[2], and Stefan Szeider[2]

[1]Leiden Institute of Advanced Computer Science, Leiden University, Leiden, The Netherlands
hh@liacs.nl
[2]Algorithms and Complexity Group, TU Wien, Vienna, Austria
{peitl,fslivovsky,sz}@ac.tuwien.ac.at

Abstract. Quantified Boolean Formulas (QBFs) are a generalization of propositional formulae that admits succinct encodings of verification and synthesis problems. Given that modern QBF solvers are based on different architectures with complementary performance characteristics, a portfolio-based approach to QBF solving is particularly promising.
While general QBFs can be converted to prenex conjunctive normal form (PCNF) with small overhead, this transformation has been known to adversely affect performance. This issue has prompted the development of several solvers for circuit QBFs in recent years.
We define a natural set of features of circuit QBFs and show that they can be used to construct portfolio-based algorithm selectors of state-of-the-art circuit QBF solvers that are close to the virtual best solver. We further demonstrate that most of this performance can be achieved using surprisingly small subsets of cheaply computable and intuitive features.

1 Introduction

The advent of modern satisfiability (SAT) solvers has established propositional logic as the low-level language of choice for encoding hard combinatorial problems arising in domains such as formal verification [4,27] and AI planning [23]. However, since the computational complexity of these problems usually outstrips the complexity of SAT, propositional encodings of such problems can be exponentially larger than their original descriptions. This imposes a limit on the problem instances that can be feasibly solved even with extremely efficient SAT solvers, and has prompted research on decision procedures for more succinct logical formalisms such as Quantified Boolean Formulas (QBFs).

QBFs augment propositional formulas with existential and universal quantification over truth values and can be exponentially more succinct. The flip side of this conciseness is that the satisfiability problem of QBFs (QSAT) is PSPACE-complete [25], and in spite of substantial progress in solver technology, practically relevant instances remain hard to solve. The complexity of QSAT is also reflected in the fact that there is currently no single best QBF solver—in fact, state-of-the-art solvers are based on fundamentally different paradigms whose underlying proof systems are known to be exponentially separated [3,10]. In particular, it has been observed that expansion-based

* This research was partially supported by FWF grants P27721 and W1255-N23.

solvers work better than search-based solvers on formulas with few quantifier alternations, while search-based solvers tend to be better suited to formulas with many quantifier alternations [17].

Thus, even more so than in the case of SAT, portfolio-based approaches that leverage the complementary strength of multiple QBF solvers, such as per-instance algorithm selection, have the potential to achieve significant speedups over individual solvers, as demonstrated for QBF formulae in the prenex CNF (PCNF) format [20]. Although any QBF can be converted to PCNF with small overhead, this transformation is known to adversely affect solver performance [1]; moreover, it can obscure features of the original instance that might be strong predictors of solver performance. In light of the first issue, researchers have developed a new standard, QCIR, for representing *quantified circuits*, or *circuit QBFs* [12],[1] while the second issue is potentially relevant to per-instance algorithm selection.

In this work, we present the first per-instance algorithm selector for QCIR formulae, built from four state-of-the-art QBF solvers, and demonstrate that it achieves performance substantially better than any of the individual solvers and close to the theoretical upper bound given by the virtual best solver (VBS) both in terms of overall runtime and number of solved instances. Following common practice, we developed and used a large set of static and dynamic instance features for this purpose. To our surprise, we discovered that, different from the situation for SAT, probing features are not helpful, and a set of only three static instance features are sufficient to achieve 99% of the performance gain obtained using our full set of features. Interestingly, these features are simple, cheaply-computable and intuitively characterize the quantification and circuit structure of the instance. Therefore, our work provides evidence that, different from what might gather from the literature, to effectively leverage per-instance algorithm selection, at least in some cases, a small set of easily implemented and computed features is sufficient. This is a significant finding, since it further lowers the barrier for researchers to effectively apply algorithm selection.

2 Related Work

For many problems in AI, there is no single algorithm that is clearly superior to all other algorithms. This may be due to algorithms implementing heuristics that work well on some instance type but not on others. Per-instance algorithm selection (as originally introduced by Rice [22]) attempts to mitigate this issue by choosing the algorithm that is expected to solve a given instance most efficiently.

In recent years, algorithm selection tools have been successfully applied to a variety of AI problems, such as SAT, CSP, ASP, and QBF [29,18,5,20]. The most common approach to algorithm selection involves picking an algorithm from a set of algorithms called a portfolio. Since the relationship between properties of a problem instance and algorithm performance is typically opaque and hard to capture formally, the construction of a portfolio normally involves training a machine learning model to predict performance and choose an algorithm [14].

[1] We only consider "cleansed" QCIR instances in prenex normal form supported by the current generation of solvers.

In the context of QBF, multinomial logistic regression has been used to switch between different branching heuristics in a search-based QBF solver based on instance features, even at runtime [24]. The (PCNF) portfolio solver AQME incorporates several models such as decision trees and nearest neighbor classification[20]. Moreover, it is "self-adaptive" in the sense that it can modify its performance prediction model to accommodate for instance types not seen during initial training. HORDEQBF is a massively parallel QBF solver [2] that implements a parallel portfolio by running multiple instances of the solver DEPQBF [16] with different parameter settings.

Automated parameter tuning is an area that is gaining popularity due to algorithms increasingly having a large number of parameters that are virtually impossible to tune by hand [6,7]. Parameter tuning can be combined with portfolio construction in order to find algorithm configurations that complement each other well [28]. Algorithm selectors typically have many options themselves (such as the choice of machine learning model and its corresponding hyperparameters), and parameter tuning can also be used to configure the selector [15].

3 Preliminaries

3.1 Circuit QBF Solvers

Our portfolios comprise the QBF solvers that participated in the prenex non-CNF track of the 2017 QBF Evaluation[2] (with the exception of CQesto, which is not publicly available; for all solvers, the default configurations provided by their authors were used). Their performance on the corresponding benchmark set was fairly similar, with the number of solved instances ranging from 89 (GHOSTQ) to 117 (QFUN) out of a total 320.

1. QUABS [26] generalizes the concept of "clause selection" (as implemented in QESTO [11] and CAQE [21]) from clauses to subformulas. An abstraction is maintained for each quantifier block, and so-called interface literals communicate whether a subformula is satisfied or falsified at a lower (or higher) level.
2. QFUN [8] generalizes counterexample-guided abstraction refinement (CEGAR) solving [9] to circuit QBFs and uses decision tree learning to "guess" counter(models) based on recent truth assignments.
3. QUTE [19] is a search-based solver that implements a technique called dependency learning to ignore artificial syntactic dependencies induced by nested quantifiers.
4. GHOSTQ [13] is a search-based solver that utilizes so-called ghost literals for dual propagation.

3.2 AutoFolio

AUTOFOLIO is an algorithm selector that alleviates the burden of manually choosing the right machine learning model for a problem domain and hand-tuning hyperparameters

[2] See http://www.qbflib.org.

by using algorithm configuration tools to automatically make design choices and find hyperparameter settings that work well for a particular scenario [15].

AUTOFOLIO allows us to construct a portfolio from the above solvers with little effort. In particular, it quickly lets us create portfolios that are tuned to particular subsets of features (see Section 6). Our main design choice consists in defining the set of features described in the next section.

4 QCIR Instance Features

We consider circuit Quantified Boolean Formulas (QBFs) in prenex normal form encoded according to the "cleansed" QCIR standard [12]. Each such formula is a pair $\mathcal{F} = \mathcal{Q}.\varphi$ consisting of a *quantifier prefix* $\mathcal{Q}$ and a Boolean circuit φ called the *matrix* of $\mathcal{F}$. The quantifier prefix $\mathcal{Q}$ is a sequence $Q_1 X_1 \dots Q_k X_k$ where each $Q_i \in \{\forall, \exists\}$ is a *quantifier* for $1 \leq i \leq k$ such that $Q_i \neq Q_{i+1}$ for $1 \leq i < k$, and the X_i are pairwise disjoint sets of variables called *quantifier blocks*.

The matrix φ is a Boolean circuit encoded as a sequence of gate definitions of the form

$$g = \circ(l_1, \dots, l_r)$$

where $\circ \in \{\wedge, \vee\}$, each *gate literal* l_i is either an unnegated gate variable g' (a *positive* gate literal) or a negated gate variable $\neg g'$ (a *negative* gate literal), and g' is a previously defined gate or an input gate $g' \in \bigcup_{i=1}^{k} X_i$. We refer to r as the *size* of gate g. The *depth* of a gate g is 0 if g is an input gate, and otherwise the maximum depth of a gate occurring in the definition of g plus one. A unique gate literal is identified as the output of the circuit φ.

We consider the following *static features* of QCIR instances:

1. The number n_e of existential variables.
2. The number n_u of universal variables.
3. The balance $n_e/n_u + n_u/n_e$ of existential and universal variables.
4. The number k of quantifier blocks.
5. The minimum size min_{b} of a quantifier block.
6. The maximum size max_{b} of a quantifier block.
7. The average size μ_{b} of a quantifier block.
8. The standard deviation σ_{b} of the quantifier block size.
9. The relative standard deviation $\sigma_{\mathrm{b}}/\mu_{\mathrm{b}}$ of the quantifier block size.
10. The total number pos of positive gate literals.
11. The total number neg of negative gate literals.
12. The balance $pos/neg + neg/pos$ of positive and negative gate literals.
13. The number $n_\wedge$ of AND gates.
14. The number $n_\vee$ or OR gates.
15. The maximum gate size max_{gs}.
16. The average gate size μ_{gs}.
17. The standard deviation σ_{gs} of the gate size.
18. The relative standard deviation $\sigma_{\mathrm{gs}}/\mu_{\mathrm{gs}}$ of the gate size.
19. The maximum gate depth max_d.

20. The average gate depth μ_d.
21. The standard deviation σ_d of the gate depth.
22. The relative standard deviation σ_d/μ_d of the gate depth.
23. The number n_p of gates all of whose gate literals have the same polarity (all positive or all negative).

Features that only depend on the quantifier prefix can be computed just as well for PCNF instances, and indeed some of the features 1–9 were already used in constructing the portfolio solver AQME [20]. The main difference between PCNF and QCIR is in the representation of the matrix and accordingly, this is where new features are required. Some of the above features (such as the numbers of AND/OR gates) can be seen as generalizations of PCNF features (number of clauses). Others, such as the maximum gate depth, only make sense for circuits.

In addition to these static features, we use several *probing features* computed by a short run of QUTE (probing features are crucial for the performance of portfolios for SAT [29]):

1. The number of learned clauses.
2. The number of learned tautological clauses.
3. The number of learned terms.
4. The number of learned contradictory terms.
5. The fraction of variable assignments made by branching (the remaining assignments are due to propagation).
6. The total number of backtracks.
7. The number of backtracks due to dependency learning (a feature of QUTE).
8. The number of learned dependencies as a fraction of the trivial dependencies.

5 Per-instance Algorithm Selection for QCIR

The experiments were conducted on a cluster where each node is equipped with 2 Intel Xeon E5-2640 v4 processors (25M Cache, 2.40 GHz) and 160GB of RAM. The machines are running 64-bit Ubuntu in version 16.04.3.

We work with the set of QCIR benchmark instances from the 2016 and 2017 QBF evaluations solved by at least one of the above solvers within 900 seconds of CPU time and 4GB of memory usage, a total of 731 instances. Figure 1 illustrates that there is a lot of complementarity between the component solvers. We split the 731 instances into a training set of 549 instances and a test set of 182 instances, uniformly at random. On the training set we fixed a cross-validation split into 10 folds of the same size. When we report performance of a selector on the *training* set, we in fact report cross-validation performance on this fixed split. This means that the selector was trained once on each subset of 9 folds and evaluated on the 10th one, and the results were combined. On the other hand, when we report performance on the test set, the respective selector is trained on the *entire* training set, disregarding the CV-split, and then evaluated on the entire test set. The reason why we use this setup for our evaluation is the following. The standard way to evaluate the performance of AUTOFOLIO is by using cross-validation. However, if AUTOFOLIO is tuned to the specific CV-split, the CV performance may be an overly

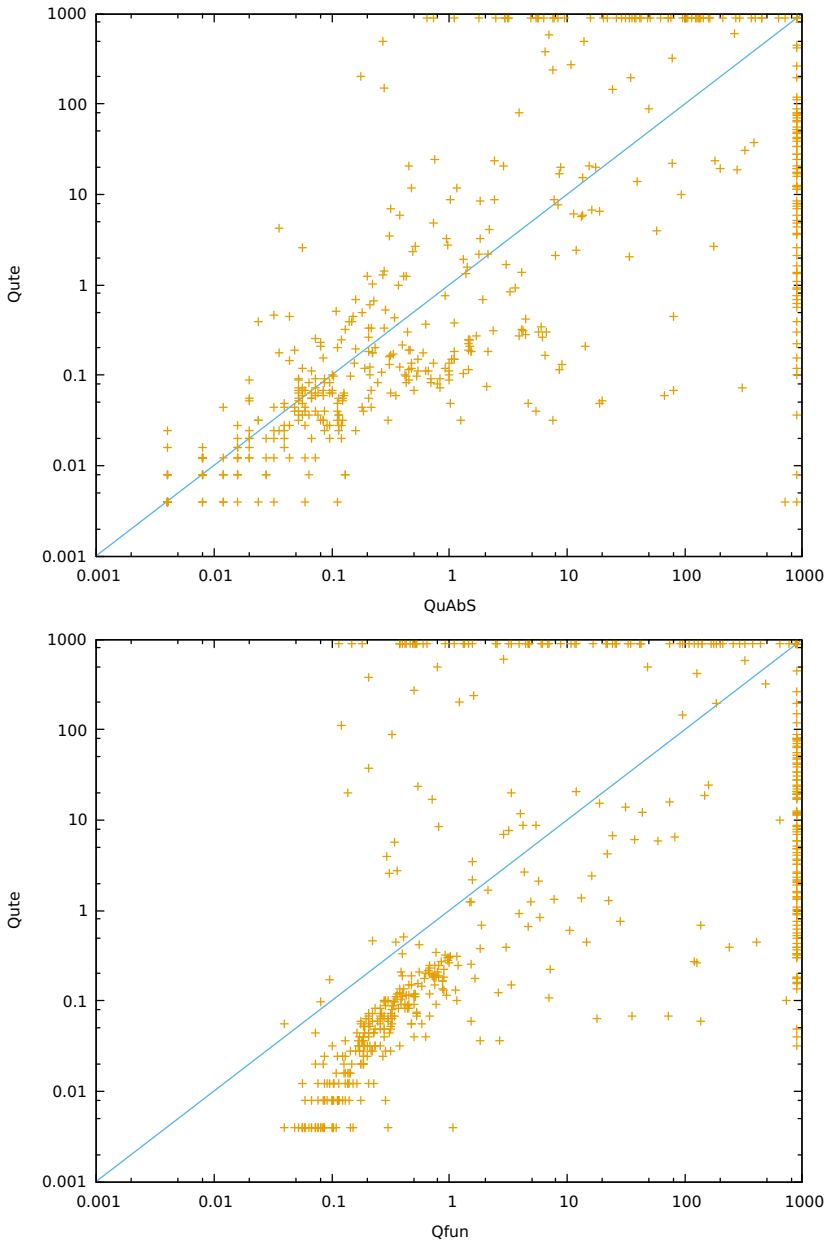

Fig. 1. Comparisons of high-performance QBF solvers on our instance set; performance is measured as PAR 10 (penalized running times with penalty factor 10) on our reference machines. This shows that there is quite a lot of complementarity between the solvers.

optimistic estimate of how well the model will generalize. Even though cross validation should still protect us from overfitting, we decided to hold out a test set even on top of that, in order to perform a sanity check of the experiment afterwards.

Each of the selectors PF* mentioned in Table 1 was trained using AUTOFOLIO in self-tuning mode, with a budget of $42\,000$ wall-clock seconds and a bound of $50\,000$ runs for the algorithm configuration tool SMAC, and with a specific subset of features (see the next section and caption of Table 1 for details). For the SMAC-configuration phase we used the CV-split as mentioned earlier. The selectors PFA, PFS, and PF3 use an XGBoost classifier, while PF2 uses a random-forest regressor.

	Training set (549)			Test set (182)		
solver	PAR10	#solved	%closed	PAR10	#solved	%closed
GhostQ	2228.92	414	—	2492.61	132	—
Qfun	1922.07	433	—	2384.68	134	—
QuAbS	1641.90	450	—	1747.40	147	0%
Qute	1458.09	461	0%	1845.48	145	—
PFA	71.93	546	96.35%	171.03	179	91.01%
PF2	57.58	547	97.35%	217.16	178	88.34%
PF3	55.78	547	97.47%	165.97	179	91.30%
PFS	55.65	547	97.48%	167.53	179	91.21%
VBS	19.46	549	100%	15.35	182	100%

Table 1. Performance of component solvers and selectors on the training and test sets in terms of penalized average runtime (PAR10), the number of solved instances, and for selectors the extent to which they match the virtual best solver (VBS) measured as the percentage of the PAR10 gap between the single best solver (SBS) and the VBS that is closed by the selector. Training performance of selectors is CV-performance. Selectors were configured using AUTOFOLIO in self-tuning mode for each of the feature subsets reported. PF2 is the selector configured for the best subset of 2 features, similarly PF3, PFS uses static features only, and PFA uses all features.

6 Which Features Matter?

It is common wisdom that high-performance per-instance algorithm selectors should have access to a large and rich set of features (see, e.g., [29]). While earlier selector designs based on ridge regression required feature selection to work well, state-of-the-art per-instance selectors make use of sophisticated machine learning techniques, such as random forests, that are less sensitive to uninformative or correlated features. However, defining and computing features requires substantial domain expertise and often involves significant amount of work, especially since feature computation must be efficient in order to achieve good selector performance. Furthermore, selectors based on large sets of complex features can be far more difficult to understand than ones based on few and simple features. Since our full feature set for QCIR formulae, as described previously,

gave rise to excellent selector performance, we decided to investigate whether similarly good performance could be obtained with fewer features.

We first trained a selector using only our static features, using AUTOFOLIO, as described in the previous section. The resulting selector, denoted PFS in Table 1, performed slightly better than the selector trained using the full set of static and probing features (PFA). This was a great surprise to us in light of previous work on algorithm selection in which probing features were found to be helpful (see, e.g., [14]). Since our full selector is already very close in performance to the VBS, it cannot be the case that we simply failed to come up with the right probing features, but rather that in the scenario we consider, static features are sufficient. Prompted by this finding, we decided to investigate the effect of further reducing our static features set.

In order to test what feature subsets might work well, we used the following setup. We configured AUTOFOLIO using the static features, and we saved the resulting configuration of hyperparameters. Then, with this configuration of AUTOFOLIO, we performed forward and backward selection on the set of static features. In forward selection, we started with the empty set of features, and at each step added a single feature, while in backward selection we started with the full set of static features, and at each step removed a feature. In both cases, the feature to be added/removed was chosen so that the resulting portfolio would have maximum performance. It is important to note here that we *did not* configure AUTOFOLIO for each of the subsets searched in this process—instead we used the configuration that we computed as described at the beginning of this paragraph. The reason for that was to avoid the huge computational cost of configuring AUTOFOLIO over and over again. In retrospect, this was indeed justified, as we obtained well-performing selectors for the feature subsets even this way, and we saved months of CPU time. However, note that once we found promising subsets of features by forward/backward selection, we configured AUTOFOLIO for these subsets again, and the results of those specifically configured selectors are reported in Table 1.

Figure 2 shows the performance curve along forward/backward selection. The values of PAR10 and the number of solved instances were obtained by performing cross validation on the fixed CV-split mentioned earlier. In particular, we can see that forward selection achieves very good performance with two or three features already. The first three features picked by forward selection are *circuit depth*, *number of quantifier blocks*, and *average block size*. Since so few features turned out to yield such good selectors, we performed a brute-force search of all subsets of size 2 or 3 (again, evaluating performance with the fixed AUTOFOLIO configuration used for forward/backward selection). This search confirmed that the size-2 subset found by forward selection was almost optimal (second best, equal number of solved instances as with the optimal set, difference of 1.2 in PAR10), while the size-3 subset was optimal. We decided to continue the experiment with the size-2 subset found by forward selection (instead of the "optimal" one), for two reasons. Firstly, it contains the feature *circuit depth*, which is the best single predictor, but which is replaced in the optimal subset by *relative standard deviation of gate depths*, a feature that is somewhat harder to interpret. Secondly, we need to keep in mind, that not even this exhaustive search was perfect, as we did not (and could not) configure AUTOFOLIO for each subset searched. Therefore, its results only served as a sanity check, to make sure that forward selection did not miss some great feature set, which

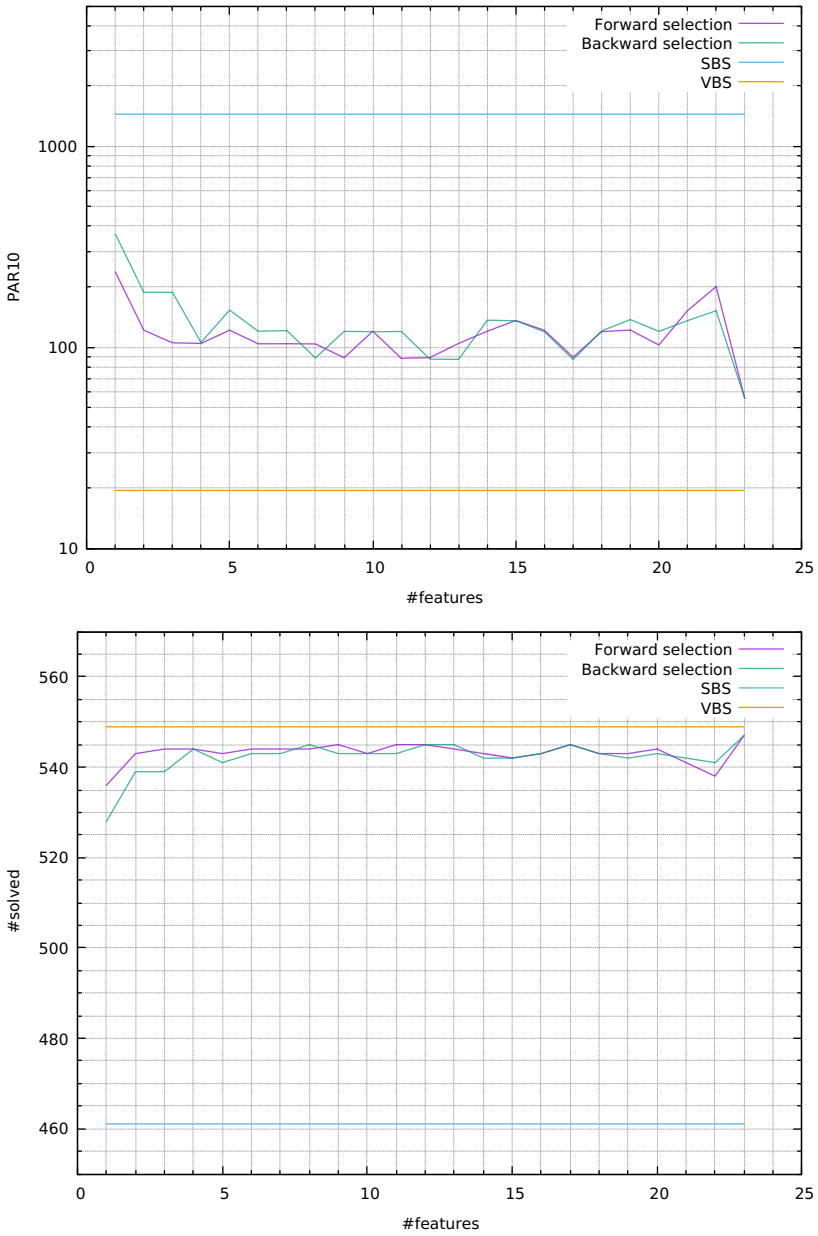

Fig. 2. Forward and backward selection on the static features; the plots show performance based on the number of features included. Note that for the performance evaluation during forward/backward selection, AUTOFOLIO was not automatically configured for each subset of features, but instead was once configured for the full set of static features at the beginning, and this configuration of hyperparameters was subsequently used for all feature subsets.

turned out not to be the case. Hence, we went on to configure AUTOFOLIO for the subsets {*circuit depth*, *number of quantifier blocks*}, and {*circuit depth*, *number of quantifier blocks*, *average block size*}, the results of which are shown in Table 1 (entries PF2 and PF3). As Table 1 shows, PF2 achieves virtually the same good performance as PFS, and closes almost all of the gap between SBS and VBS. This holds whether we look at the CV-evaluation on the training set, or the additional evaluation on the test set.

As a final sanity check, we evaluated the performance of selectors trained using these small sets of features on the same set of instances, but using only 3 out the 4 participating solvers (for each subset of 3 solvers). We set this experiment up in the following way: for each subset of features corresponding to one of the selectors PF*, we saved the configuration of AUTOFOLIO that was optimized for the particular subset of features using all four solvers. We then evaluated the performance of selectors built using the saved configurations for each of the 4 size-3 solver subsets (a total of 16 selectors), in the same way as we did for Table 1. In order to get the theoretically best AUTOFOLIO performance, we would have had to reconfigure AUTOFOLIO for every pair of (solver subset, feature subset), but as before we simplified things to save computational resources. This experiment confirmed that even for different solver sets, the features *circuit depth*, and *number of quantifier blocks* are fairly robust predictors of solver performance. However, naturally, features must be tied to solvers whose performance they predict, so we cannot expect that a fixed set of features will be a universal predictor for all solver sets.

In a sense, these results are not surprising, as one would expect from complexity theory as well as from previous work that the number of quantifier blocks indeed plays an important role. Similarly, circuit depth seems to be a prominent property of circuits. However, it is indeed striking that only two, and moreover the most straightforward features of circuit QBF suffice to build such robust portfolios. We believe that this opens up a new path of thinking for both solver users and developers. Users can classify their benchmarks and pick a suitable solver more easily, while developers can take advantage of this information to build portfolios within their solvers. Believing many features are necessary to learn anything meaningful about a given instance can be discouraging from even trying. With just two features, the options are much wider—they can be understood intuitively, or even plotted. In fact, to demonstrate how we can gain additional insight into the problem, we visualize the solver choices made both by the portfolio, as well as by the VBS. When plotting the VBS in Figure 4, we ignore instances where the solvers perform too similarly, because they contain more noise than information. On the other hand, we plot the portfolio choices in Figure 5 as a grid (of hypothetical instances), in order to discover the decision boundaries. These figures show very clearly which solvers are good for which instances. Incidentally, Figure 4 also reveals the fact that the QCIR instances that are available either have many quantifier blocks, or deep circuits, or neither, but not both (strictly speaking, to see that, we would need to plot all instances, but the picture has the same shape, only more noise). This should serve as a challenge to the QBF community to come up with a more complete distribution of benchmark instances.

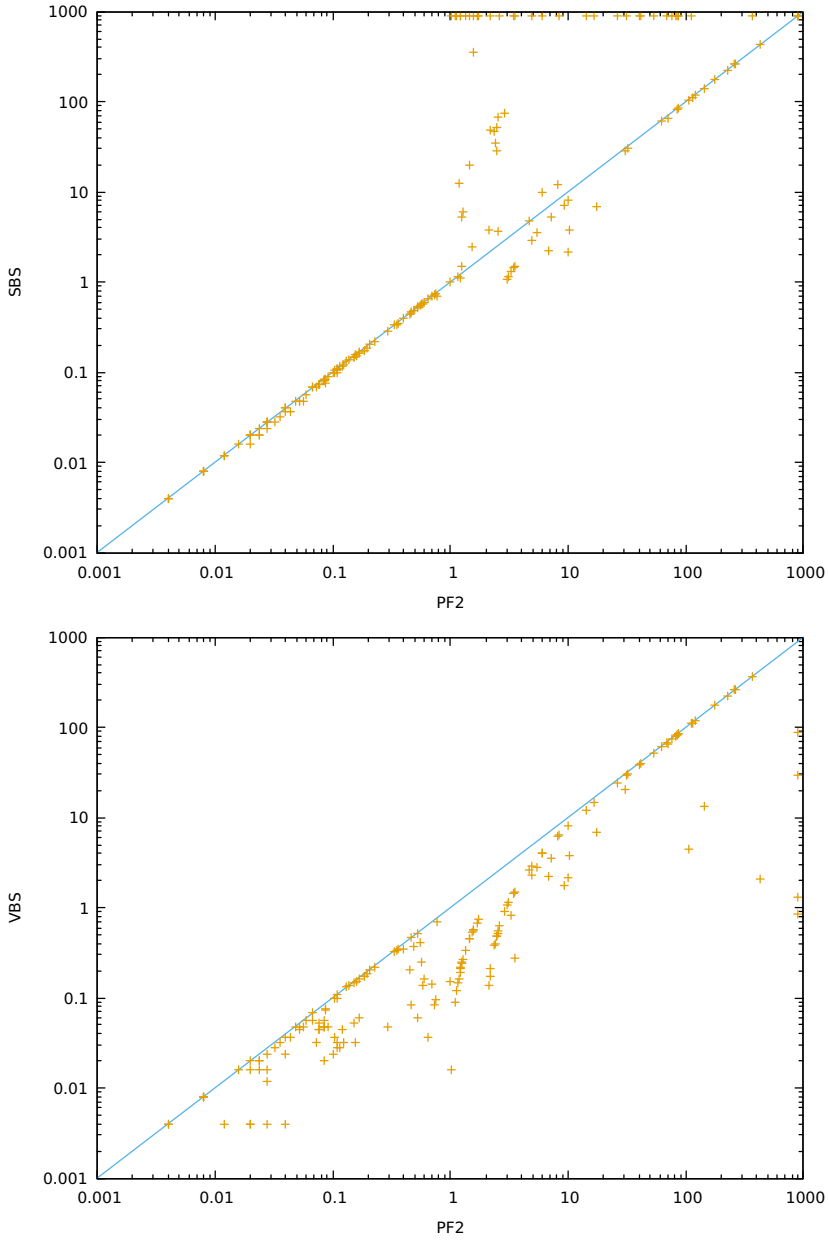

Fig. 3. Performance of PF2 with all four solvers vs SBS and VBS.

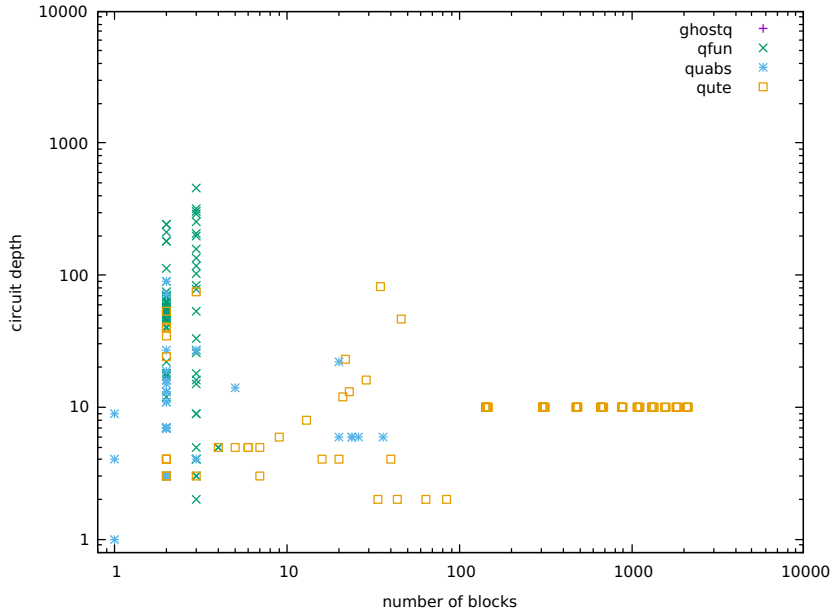

Fig. 4. Best solver choices based on instance features. Each point represents an instance/solver pair; the coordinates correspond to the number of quantifier blocks and circuit depth of the instance, the shape and color indicate the solver that is fastest on that instance. Only instances where the fastest solver is either the only one to solve the instance, or at least ten times faster than the second fastest, are shown. This is to ensure that the figure shows only solver choices that are crucial, and to avoid instances where the solver choice is unimportant, because all of them run in similar time.

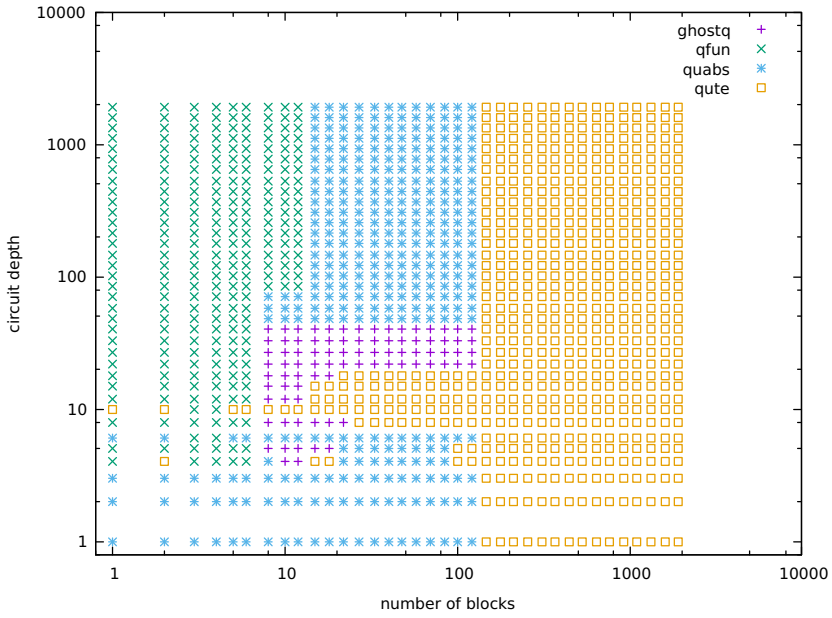

Fig. 5. Points indicate solver choices of PF2 based on feature values.

7 Conclusions and Future Work

With the availability of tools such as AUTOFOLIO, the task of constructing effective per-instance algorithm selectors essentially boils down to designing and implementing features that (jointly) permit to effectively identify which solver to run on any given problem instance. This can still seem daunting in view of the fact that certain domains require rich sets of quickly computable features, with a combination of static and dynamic features, in order to achieve good selector performance [29]. Our results show that this need not be the case: for circuit QBFs, two or three cheaply computable instance features are sufficient to realize most of the performance potential of a (hypothetical) perfect selector. Moreover, these features include properties of QBFs such as the number of quantifier blocks that are known to affect solver performance. Apart from corroborating the notion that quantifier alternations matter, our results show that circuit depth seems to be important. This warrants further investigation.

Our finding that simple feature sets can be effective likely applies to other problems and encourages an incremental design philosophy: start with a few simple features and add features as needed. As part of future work we hope to find other domains where this approach works well and, more generally, identify the circumstances under which this is the case.

References

1. Ansótegui, C., Gomes, C.P., Selman, B.: The Achilles' heel of QBF. In: Veloso, M.M., Kambhampati, S. (eds.) The Twentieth National Conference on Artificial Intelligence - AAAI 2005. pp. 275–281. AAAI Press / The MIT Press (2005)
2. Balyo, T., Lonsing, F.: Hordeqbf: A modular and massively parallel QBF solver. In: Creignou, N., Berre, D.L. (eds.) Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9710, pp. 531–538. Springer Verlag (2016)
3. Beyersdorff, O., Chew, L., Janota, M.: Proof complexity of resolution-based QBF calculi. In: Mayr, E.W., Ollinger, N. (eds.) 32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany. LIPIcs, vol. 30, pp. 76–89. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015)
4. Biere, A.: Bounded model checking. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 457–481. IOS Press (2009)
5. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., Schneider, M.T., Ziller, S.: A portfolio solver for answer set programming: Preliminary report. In: Delgrande, J.P., Faber, W. (eds.) Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6645, pp. 352–357. Springer Verlag (2011)
6. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Coello, C.A.C. (ed.) Learning and Intelligent Optimization - 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers. Lecture Notes in Computer Science, vol. 6683, pp. 507–523. Springer Verlag (2011)
7. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: Paramils: An automatic algorithm configuration framework. J. Artif. Intell. Res. **36**, 267–306 (2009)
8. Janota, M.: Towards generalization in QBF solving via machine learning. In: McIlraith, S.A., Weinberger, K.Q. (eds.) Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence - AAAI 2018. AAAI Press (2018)
9. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.M.: Solving QBF with counterexample guided refinement. In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing - SAT 2012. Lecture Notes in Computer Science, vol. 7317, pp. 114–128. Springer Verlag (2012)
10. Janota, M., Marques-Silva, J.: Expansion-based QBF solving versus q-resolution. Theoretical Computer Science **577**, 25–42 (2015)
11. Janota, M., Marques-Silva, J.: Solving QBF by clause selection. In: Yang, Q., Wooldridge, M. (eds.) Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015. pp. 325–331. AAAI Press (2015)
12. Jordan, C., Klieber, W., Seidl, M.: Non-cnf QBF solving with QCIR. In: Darwiche, A. (ed.) Beyond NP, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 12, 2016. AAAI Workshops, vol. WS-16-05. AAAI Press (2016)
13. Klieber, W., Sapra, S., Gao, S., Clarke, E.M.: A non-prenex, non-clausal QBF solver with game-state learning. In: Strichman, O., Szeider, S. (eds.) Theory and Applications of Satisfiability Testing - SAT 2010. Lecture Notes in Computer Science, vol. 6175, pp. 128–142. Springer Verlag (2010)
14. Kotthoff, L.: Algorithm selection for combinatorial search problems: A survey. In: Bessiere, C., Raedt, L.D., Kotthoff, L., Nijssen, S., O'Sullivan, B., Pedreschi, D. (eds.) Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach, Lecture Notes in Computer Science, vol. 10101, pp. 149–190. Springer (2016)

15. Lindauer, M.T., Hoos, H.H., Hutter, F., Schaub, T.: Autofolio: An automatically configured algorithm selector. J. Artif. Intell. Res. **53**, 745–778 (2015)
16. Lonsing, F., Egly, U.: Depqbf 6.0: A search-based QBF solver beyond traditional QCDCL. In: de Moura, L. (ed.) Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10395, pp. 371–384. Springer Verlag (2017)
17. Lonsing, F., Egly, U.: Evaluating QBF solvers: Quantifier alternations matter. CoRR **abs/1701.06612** (2017), http://arxiv.org/abs/1701.06612
18. OMahony, E., Hebrard, E., Holland, A., Nugent, C., OSullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In: Irish conference on artificial intelligence and cognitive science. pp. 210–216 (2008)
19. Peitl, T., Slivovsky, F., Szeider, S.: Dependency learning for QBF. In: Gaspers, S., Walsh, T. (eds.) Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10491, pp. 298–313. Springer Verlag (2017)
20. Pulina, L., Tacchella, A.: A self-adaptive multi-engine solver for quantified Boolean formulas. Constraints **14**(1), 80–116 (2009)
21. Rabe, M.N., Tentrup, L.: CAQE: A certifying QBF solver. In: Kaivola, R., Wahl, T. (eds.) Formal Methods in Computer-Aided Design - FMCAD 2015. pp. 136–143. IEEE Computer Soc. (2015)
22. Rice, J.R.: The algorithm selection problem. Advances in Computers **15**, 65–118 (1976)
23. Rintanen, J.: Planning and SAT. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 483–504. IOS Press (2009)
24. Samulowitz, H., Memisevic, R.: Learning to solve QBF. In: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada. pp. 255–260. AAAI Press (2007)
25. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time: Preliminary report. In: Aho, A.V., Borodin, A., Constable, R.L., Floyd, R.W., Harrison, M.A., Karp, R.M., Strong, H.R. (eds.) Proceedings of the 5th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1973, Austin, Texas, USA. pp. 1–9. Assoc. Comput. Mach., New York (1973)
26. Tentrup, L.: Non-prenex QBF solving using abstraction. In: Creignou, N., Berre, D.L. (eds.) Theory and Applications of Satisfiability Testing - SAT 2016. Lecture Notes in Computer Science, vol. 9710, pp. 393–401. Springer Verlag (2016)
27. Vizel, Y., Weissenbacher, G., Malik, S.: Boolean satisfiability solvers and their applications in model checking. Proceedings of the IEEE **103**(11), 2021–2035 (2015)
28. Xu, L., Hoos, H., Leyton-Brown, K.: Hydra: Automatically configuring algorithms for portfolio-based selection. In: Fox, M., Poole, D. (eds.) Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010. AAAI Press (2010)
29. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Satzilla: Portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. **32**, 565–606 (2008)