# 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 (LIACS), Universiteit Leiden, The Netherlands
hh@liacs.nl
[2] Algorithms and Complexity Group, TU Wien, 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.
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

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].

Thus 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],[3] while the second issue is potentially relevant to per-instance algorithm selection.

In this work, 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

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

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, at least in some cases, a small set of easily implemented features is sufficient. This is a significant finding, since it further lowers the barrier for researchers to effectively apply algorithm selection.

## 2 Setup

Our portfolios comprise the QBF solvers that participated in the prenex non-CNF track of the 2017 QBF Evaluation[4] (with the exception of the CQesto, which is not publicly available; for all solvers, the default configurations provided by their authors were used): QuAbS [26], QFun [8], Qute [19], and GhostQ [13].

We use AutoFolio [15] to construct a portfolio from these solvers. 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 by using algorithm configuration tools to automatically to make design choices and find hyperparameter settings that work well for a particular scenario. 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 5). Our main design choice consists in defining a set of features.

## 3 QCIR Instance Features

We consider circuit Quantified Boolean Formulas (QBFs) in prenex normal form encoded according to the "cleansed" QCIR standard [12]. We compute 23 *static features* of QCIR instances, such as:

1. The number of existential variables.
2. The number of universal variables.
4. The number of quantifier blocks.
7. The average quantifier block size.
13. The number of AND gates.
14. The number or OR gates.
19. The maximum gate depth.
20. The average gate depth.

Features that only depend on the quantifier prefix can be computed just as well for PCNF instances, and indeed some of these features 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

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

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]).

## 4 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. 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, i.e., the selector trained once on each subset of 9 folds and evaluated on the 10th one, results 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.

Results are shown in Table 1. Each of the selectors 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 algorithm configuration. 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 (SBS) | 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 and PF3 are selectors configured for the best feature subsets of sizes 2 and 3, and PFS only uses static features. PFA uses all features.

## 5 Which Features Matter?

Since our full feature set for QCIR formulae 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. This was surprising in light of previous work on algorithm selection in which probing features were found to be necessary (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 didn't 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.

Forward/backward selection indicated that very good performance can be achieved already with two or three features. Accordingly, we performed a brute-force search of all subsets of size 2 or 3. This search confirmed that both the size-2 and size-3 subsets found by forward selection were almost optimal (equal number of solved instances as with the optimal set, PAR10 within 1%).

As an additional 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). This revealed that even for different solver sets, these features are robust predictors of solver performance, closing anywhere between 80 and 97% of the gap between SBS and VBS for each solver subset. Afterwards, we configured AUTOFOLIO for these subsets, the results of which are shown in Table 1 (entries PF2 and PF3). It is safe to say that especially these two features (we note that due to the nature of forward selection, the 2-subset is a subset of the 3-subset) are crucial for prediction of solver performance. These features are *number of quantifier blocks* and *maximum gate depth*. Although one would expect these features to play an important role, it is still striking that these two features suffice to build robust portfolios.

## 6 Conclusions and Future Work

With the availability of tools such as AUTOFOLIO [15] 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 featuresin 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.

## 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.: QFUN: towards machine learning in QBF. CoRR **abs/1710.02198** (2017), http://arxiv.org/abs/1710.02198
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)

## A 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].

In the context of QBF, multinomial logistic regression has been used to switch between different branching heuristics in a search-based QBF solver based in 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].

## B Detailed List of 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 $\varphi$ 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 $\varphi$ 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 $\varphi$.

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 $min_{\mathrm{b}}$ of a quantifier block.
6. The maximum size $max_{\mathrm{b}}$ of a quantifier block.
7. The average size $\mu_{\mathrm{b}}$ of a quantifier block.
8. The standard deviation $\sigma_{\mathrm{b}}$ 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 $max_{\mathrm{gs}}$.
16. The average gate size $\mu_{\mathrm{gs}}$.
17. The standard deviation $\sigma_{\mathrm{gs}}$ 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 $\mu_d$.
21. The standard deviation $\sigma_d$ of the gate depth.
22. The relative standard deviation $\sigma_d/\mu_d$ 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).

In addition to these static features, we use several *probing features* computed by a short run of QUTE:

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.

## C Additional Figures

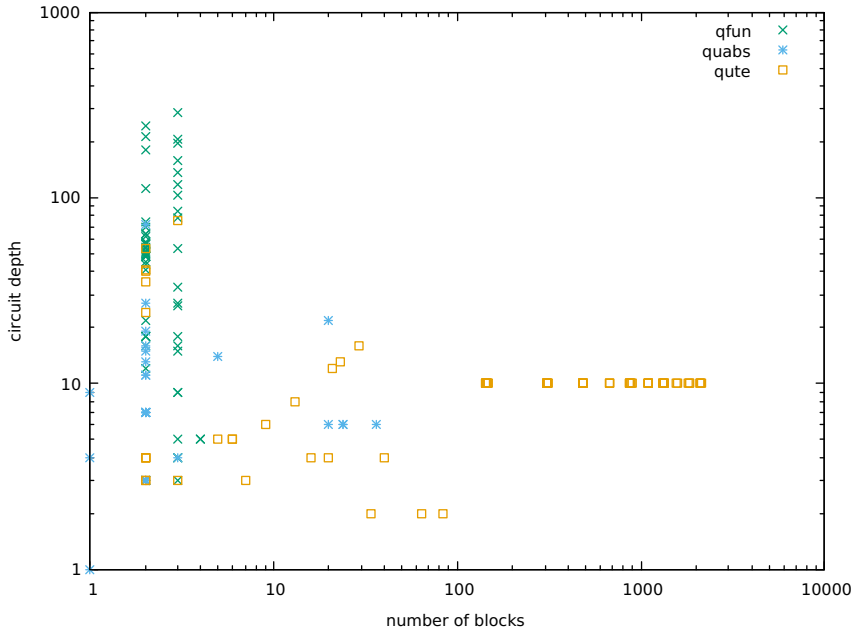


**Fig. 1.** Dependence of solver performance on circuit depth and number of quantifier blocks. Each point represents an instance/solver pair; the coordinates correspond to the number of quantifier blocks and circuit depth of the instance, the shape and color of the data point indicate the solver that 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. Also note that GHOSTQ is not present in this figure, because it is never substantially better than the other solvers.

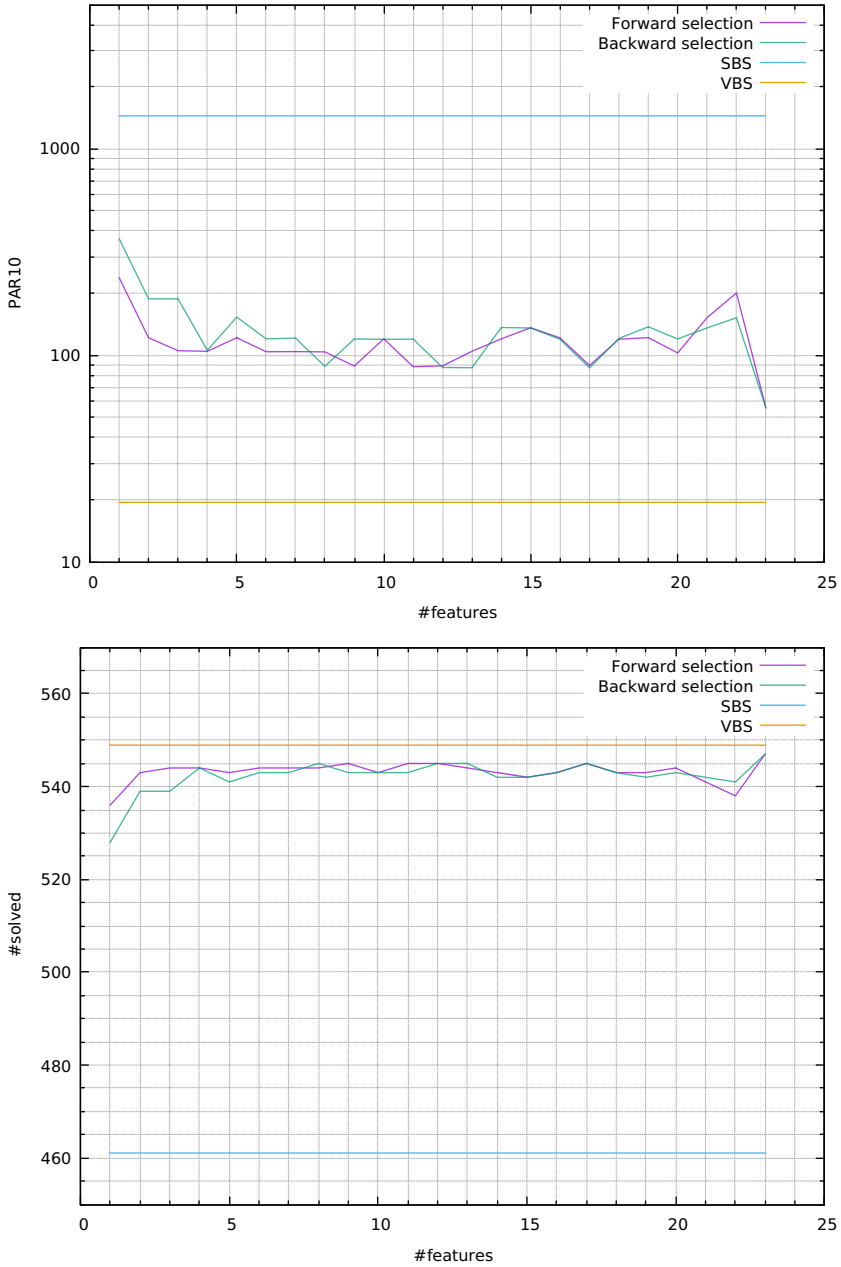


**Fig. 2.** Forward and backward selection on the static features; the plots show performance based on the number of features included. Note that for the performance evaluation during forward/backward selection, AUTOFOLIO was not automatically configured for the particular set of features, but instead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