

Learning to Solve QBF

Horst Samulowitz and Roland Memisevic

Department of Computer Science
University of Toronto
Toronto, Canada
[horst|roland]@cs.toronto.edu

Abstract

We present a novel approach to solving Quantified Boolean Formulas (QBF) that combines a search-based QBF solver with machine learning techniques. We show how classification methods can be used to predict run-times and to choose optimal heuristics both within a portfolio-based, and within a dynamic, online approach. In the dynamic method variables are set to a truth value according to a scheme that tries to maximize the probability of successfully solving the remaining sub-problem efficiently. Since each variable assignment can drastically change the problem-structure, new heuristics are chosen dynamically, and a classifier is used online to predict the usefulness of each heuristic. Experimental results on a large corpus of example problems show the usefulness of our approach in terms of run-time as well as the ability to solve previously unsolved problem instances.

Introduction

Quantified boolean formulas (QBF) are a powerful generalization of the satisfiability problem (SAT) in which the variables are also allowed to be universally as well as existentially quantified. The ability to nest universal and existential quantification in arbitrary ways makes QBF considerably more expressive than SAT. While any NP problem can be encoded in SAT, QBF allows us to encode any PSPACE problem: QBF is PSPACE-complete.

This expressiveness opens a much wider range of potential application areas for a QBF solver, including areas like automated planning (particularly conditional planning), non-monotonic reasoning, electronic design automation, scheduling, model checking and verification. The difficulty, however, is that QBF is in practice a much harder problem to solve than SAT. (It is also much harder theoretically, assuming that PSPACE \neq NP). One indication of this practical difficulty is the fact that current QBF solvers are typically limited to problems that are about 1-2 orders of magnitude smaller than the instances solvable by current SAT solvers (1000's of variables rather than 100,000's).

Nevertheless, this limitation in the size of the instances solvable by current QBF solvers is somewhat misleading. In particular, many problems have a much more compact encoding in QBF than in SAT. For example, (Ali *et al.* 2005)

give an innovative application of QBF to hardware debugging, showing that the QBF encoding of the problem is many times smaller than an equivalent SAT encoding. Results like this demonstrate the potential of QBF and the importance of continuing to improve QBF solvers.

In this paper we develop a framework to solving QBF that combines a search-based QBF solver with machine learning techniques. We use statistical classification to predict optimal heuristics within a portfolio-, as well as in a dynamic, online-setting. Our experimental results show that it is possible to obtain significant gains in efficiency over existing solvers in both settings.

While a few preliminary approaches exist that apply machine learning methods to solve SAT, no such approaches have been reported yet for QBF. For SAT, (Nudelman *et al.* 2004) describe a methodology that starts with a fixed set of pre-chosen solvers and uses learning to determine which solvers to use for given problem instances. Similarly, (Hutter *et al.* 2006) describe an approach to choosing optimal solvers along with optimal parameter settings for a set of problem instances, by trying to predict their run-times. Common to these and similar approaches is, that they make use of a fixed, pre-determined set of solvers. Learning methods are used to make an optimal assignment *ahead of time*, as a pre-processing step.

The main motivation behind such portfolio-based approaches is that they can make use of already developed and already highly optimized machinery, since they merely need to solve a simple assignment problem. A potential disadvantage is that they give up on the opportunity to obtain entirely novel methods that show a qualitatively different behavior than previous methods. Our dynamic approach therefore make use of a portfolio-based scheme only on the level of sub-components and uses classification to choose online among a set of heuristics to solve sub-instances.

Among existing work on using learning to solve SAT the method that comes closest to our approach is probably (Lagoudakis & Littman 2001). The approach uses reinforcement learning to dynamically adjust the branching behavior of a solver, but does not make use of the properties of sub-instances that need to be solved in each step, and it failed to show an improvement over non-learning based approaches. In this paper, in contrast we use discriminative learning in order to dynamically *predict* optimal branching heuristics

from the (sub-)instances a solver encounters at each step.

Background

A quantified boolean formula has the form $\vec{Q}.F$, where F is a propositional formula expressed in CNF and \vec{Q} is a sequence of quantified variables ($\forall x$ or $\exists x$). We require that no variable appear twice in \vec{Q} and that all variables in F appear in \vec{Q} (i.e., F contains no free variables).

For example, $\exists e_1 e_2. \forall u_1 u_2. \exists e_3 e_4. (e_1, \neg e_2, u_2, e_4) \wedge (\neg u_1, \neg e_3)$ is a QBF with $\vec{Q} = \exists e_1 e_2. \forall u_1 u_2. \exists e_3 e_4$ and $F = (e_1, \neg e_2, u_2, e_4) \wedge (\neg u_1, \neg e_3)$. The quantifier blocks in order are $\exists e_1 e_2$, $\forall u_1 u_2$, and $\exists e_3 e_4$, the u_i variables are universal while the e_i variables are existential, and $e_1 \leq_q e_2 <_q u_1 \leq_q u_2 <_q e_3 \leq_q e_4$. A QBF solver has to respect the quantifier ordering when branching on variables.

QBF solvers are interested in answering the question of whether or not $\vec{Q}.F$ expresses a true or false assertion, i.e., whether or not $\vec{Q}.F$ is true or false. The **reduction** of a CNF formula F by a literal ℓ is the new CNF $F|_\ell$ which is F with all clauses containing ℓ removed and $\neg\ell$, the negation of ℓ , removed from all remaining clauses. For example, let $\vec{Q}.F = (\forall xz. \exists y. (\bar{y}, x, z) \wedge (\bar{x}, y))$. Then, $\vec{Q}.F|_x = \forall z. \exists y. (\bar{y}, z)$. The semantics of a QBF can be defined recursively in the following way:

1. If F is the empty set of clauses then $\vec{Q}.F$ is *true*.
2. If F contains an empty clause then $\vec{Q}.F$ is *false*.
3. $\forall v \vec{Q}.F$ is *true* iff both $\vec{Q}.F|_v$ and $\vec{Q}.F|_{\neg v}$ are true.
4. $\exists v \vec{Q}.F$ is *true* iff at least one of $\vec{Q}.F|_v$ and $\vec{Q}.F|_{\neg v}$ is true.

Dynamic Prediction

In this section we present our approach to integrate machine learning techniques within a search-based QBF solver. First, we briefly explain how a search-based solver works in general. Second, we point out the main idea behind our approach and how it differs from the standard way of solving QBF.

Search-Based QBF Solver

Search-based QBF solvers are based on a modification of the Davis-Putnam-Longman algorithm (DPLL, (Davis, Logemann, & Loveland 1962)). DPLL works on the principle of assigning variables, simplifying the formula to account for that assignment and then recursively solving the simplified formula. The main difference to the original algorithm used to solve SAT is the fact that with QBF it is not only necessary to backtrack from a conflict but also from a solution in order to verify both settings of each universally quantified variable. A recursive version of this algorithm can be stated as follows (ignoring Lines 4 to 5).

- 1 *bool* **Learn-QBF**($\vec{Q}.F$)
- 2 **if** F contains an [empty clause/is empty] **then**
- 3 **return**([FALSE/TRUE])
- 4 [Compute features of F]
- 5 [Predict best heuristic h among h_1, \dots, h_n]
- 6 Select variable v according to heuristic function h
- 7 $\vec{Q}.F' = \vec{Q}.F|_{l_v}$
- 8 *bool* $bValue = \mathbf{Learn-QBF}(\vec{Q}.F')$
- 9 **if** $bValue == \mathbf{false}$ **and** v is universal **then**
- 10 **return** *false*
- 11 **else if** $bValue == \mathbf{true}$ **and** v is existential **then**
- 12 **return** *true*
- 13 $\vec{Q}.F' = \vec{Q}.F|_{\neg l_v}$
- 14 **return** **Learn-QBF**($\vec{Q}.F'$)

The algorithm simply reflects the earlier stated semantic definition of QBF. Modern backtracking QBF solvers employ conflict as well as solution learning to achieve a better performance (e.g., (Zhang & Malik 2002), (Letz 2002)). Furthermore, several degrees of reasoning at each search node have been proposed. In addition to the standard closure under unit propagation, stronger inference rules like hyper-binary resolution were introduced (Samulowitz & Bacchus 2006). Consequently, at each node the theory is not simply reduced by a single literal only, but further reasoning is applied.

This iterative application of reduction steps is likely to change the structural properties of the initial theory in an essential way. And since it is a well-known fact that the performance of a heuristic varies essentially across different instances it is one of our purposes in this work to show that the performance of a heuristic can also change dynamically as we descend into the search tree.

Our change to the recursive algorithm is depicted in the scheme shown above (Lines 4 to 5). Before selecting a new variable we compute the properties of the current theory, which has been dynamically generated. Then, we use a previously trained classifier to determine which heuristic is suited best for this theory.

In the following subsections we describe the different heuristics we designed and how we capture the properties of a theory.

Heuristics

In order to achieve a wide variety of solver characteristics we developed 10 different heuristics. All heuristics are crafted so that each of them tries to be orthogonal to the others. The next branching literal was mainly selected based on one or a combination of the VSIDS score and cube score, the number of implied unit propagations and satisfied clauses. While the VSIDS score (Moskewicz *et al.* 2001) is mainly based on recent conflicts occurred during the search, the cube score (Zhang & Malik 2002) is based on the recently encountered solutions. Stated differently, branching according to VSIDS tries to discover another conflict space while the cube score guides the search towards the solution space. The other two measures behave in a similar dual fashion. For instance, picking a literal with a high number of implied literals is

likely to reduce and constrain the remaining theory maximally. In contrast, a literal that satisfies the highest number of clauses reduces the theory as well, but it does not necessarily reduce the length of the remaining clauses in an essential way. We also use the inverse of these measures in several heuristics.

We also employ other measures like the weighted sum between the number of literals forced by a literal and its corresponding VSIDS score in order to provoke a conflict even more drastically. In addition, there exist several other parameters (e.g., detect pure literals), but space restrictions prevent us from going into great detail about all the different heuristics. However, it is worth mentioning that we also use a heuristic that simply employs a static variable ordering which performs surprisingly well on a subset of benchmarks (see heuristic *H10*). In SAT it is provable that an inflexible variable ordering can cause an exponential explosion in the size of the backtracking search tree. However, the given displayed results might indicate that the behaviour of a static variable ordering is more complex in the QBF setting. One reason for this could be the employed solution learning with QBF which could potentially benefit from a static variable ordering.

Feature Choice

In this section we point out the basic measures and give an intuition on how we generated other features from these. Again, the limited amount of space precludes a detailed listing and description of all features we used in our approach.

In total, we selected 78 features to capture the structure contained within an instance. All features are mainly based on the following basic properties of a QBF instance:

- # Variables (# Existentials, # Universals)
- # Clauses (# binary, # ternary, # k-ary)
- # Quantifier Alternations
- # Literal Appearances (# in binary, # in ternary, # in k-ary)

Based on these fundamental attributes we additionally compute several ratios between combinations of these attributes, like the clause/variable ratio. Again, we also compute this ratio in the context of binary, ternary, and n-ary clauses.

While we compute many features that are also applicable with SAT (see e.g., (Nudelman *et al.* 2004)) we also take into account properties that are specific to QBF. For instance, based on the the number of binary clauses that contain existentially quantified variables we compute the ratio of existentials and binary clauses further weighted by the number of universals. This weighted ratio tries to capture the degree of constrainedness of an instance: The lower the ratio the more constrained are the existentials.

While this ratio focuses on the two different quantification types, we also took the number of quantifier alternations into account. For instance, we weighted the number of literal appearances by the number of the corresponding quantifier block. This is motivated by the fact that variables from inner quantifier blocks are often less constrained than variables from outer quantifier blocks.

Classification

In the following we describe the approach that we use for predicting optimal heuristics for problem (sub-)instances. A

key requirement for the predictor is run-time efficiency, because it can potentially get applied very often when solving a given problem instance. Furthermore, it is important to obtain well-calibrated outputs that reflect the confidences in the classification decisions over all possible heuristics. If calibrated correctly, these confidences allow us to determine at run-time, when it is worth switching the heuristic, and when not.

A simple classifier that satisfies both these requirements is multinomial logistic regression (see e.g., (Hastie, Tibshirani, & Friedman 2001)): We maximize the probability $p(h|x)$ of predicting the correct heuristic h from an instance x based on a set of training cases. Here, x denotes the feature-representation of an instance. Using a linear response function, the probability can be defined as

$$p(h|x) = \frac{\exp(w_h^T x)}{\sum_{h'} \exp(w_{h'}^T x)}. \quad (1)$$

For training, we can maximize the regularized (log-) probability of training data:

$$\sum_i \log p(h^i|x^i) + \lambda \|w\|^2 \quad (2)$$

wrt. the parameters $w := (w_h)_{h=1, \dots, n}$. Any gradient based optimization method can be used for this purpose. Since the objective is convex, there are no local minima. At test-time, classification decisions can be made by finding the maximum of Eq. 1, or equivalently of $w_h^T x$, wrt. the heuristic h .

To obtain the training set we applied each heuristic as the top-level decision on a set of benchmark datasets and recorded the run times resulting from using each heuristic. We defined as the winning heuristic for each problem instance the one whose overall runtime is smallest. This dataset is certainly suboptimal because of its limited size, and to obtain even better performance additional training data could be gathered from the running system by collecting sub-instances. However, we obtained very good results already using this limited dataset, which shows that there is a sufficient degree of 'self-similarity' present in the problem-instances: The features of top-level problem instances have properties that are comparable to those of sub-instances and are good enough for generalization.

Experimental Evaluation

To evaluate the empirical effect of our new approach we considered all of the non-random benchmark instances from QBFLib 2005 and 2006 (Narizzano, Pulina, & Tacchella 2006) (723 instances in total). In order to increase the number of non-random instances we applied a version of static partitioning on all instances. A QBF can be divided in to disjoint sub-formulas as long as each of these sub-formulas do not share any existentially quantified variables (see, e.g., (Samulowitz & Bacchus 2007)). This way we obtained a total of 1647 problem instances. However, among these instances there existed several duplicates (instances that fall into symmetric sub-theories) and instances that were solved by all approaches in 0 seconds. We discarded all of these cases and ended up with 897 instances across 27 different

benchmark families. To obtain a larger dataset for training we used 800 additional random instances, that were not used for testing. On all test runs the CPU time limit was set to 5,000 seconds. All runs were performed on 2.4GHz Pentiums with 4GB of memory (SHARCNET 2007).

Variable Elimination vs. Search

To see whether QBF can gain from statistical approaches similarly as previously SAT, we first experimented with a pure portfolio-approach, where the goal is to predict which method from a set of pre-defined methods is the one best for a given instance.

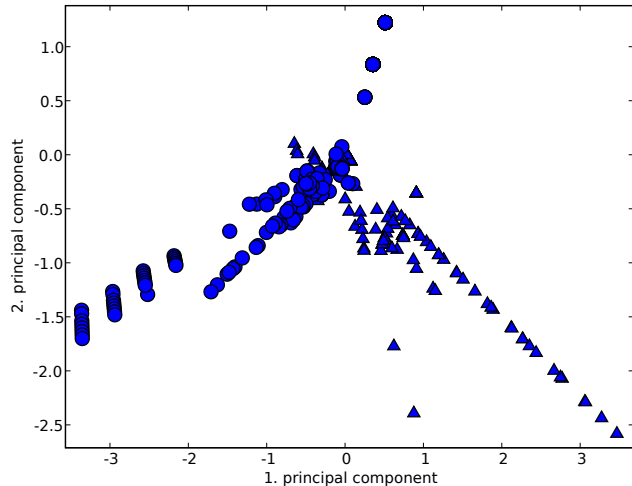


Figure 1: Low dimensional projection of problem instances that could be solved either by variable elimination (circles) or the search-based method (diamonds).

It is often easy to construct specialized methods that are *very good* at solving a *very small* set of problems, but it is much more difficult to develop a method that shows consistently good performance across a large set of problems. The advantage of using learning based approaches is that they allow us to *combine* highly specialized methods, since they can predict the right method for each (sub-)instance.

To illustrate that the features of instances can capture this orthogonality, we visualize a subset of the above mentioned data using principal components analysis (Hastie, Tibshirani, & Friedman 2001). Figure 1 shows instances that could be solved by exactly one of two solvers (but not by both simultaneously), using different symbols to indicate which of the two solvers was able to solve each instance. The two solvers are based on (i) variable elimination (Biere 2004) and (ii) search as described earlier.

The plot shows that there is a quite clear separation of these instances already in two dimensions, and suggests that a linear classifier using all available features should indeed yield good performances in practice. As described above, we used logistic regression to predict which of the two methods to use in each case. We estimated λ using cross-validation on the training set. All reported final performances were computed on an independent test set. Each of the two methods has a time-out of 5000 seconds, after

Table 1: Performances: Variable elimination vs. search.

	PROBLEMS SOLVED	TIME SPENT (IN SEC)
ONLY VARIABLE ELIMINATION	595	7918.58
ONLY SEARCH-BASED	423	31116.24
AUTOMATIC PREDICTION	666	27923.81

which we declare it as unable to solve the instance within a reasonable amount of time.

Table 1 displays the results on the test set and shows that choosing the best heuristic based on the data on an item-by-item basis yields much better performance than each of the two methods alone. In fact, the automatic prediction compared to variable elimination achieves a 12% improvement while the performance of the search based solver can be improved by more than 50%.

These results further underline the orthogonality of these two approaches. The two distinct characteristics were exploited also by the winning QBF solver of the 2006 QBF-competition (Narizzano, Pulina, & Tacchella 2006). The winning search-based solver used Quantor (a solver based on variable elimination (Biere 2004)) as a time-limited pre-processor. Here we are able to uncover this difference and use learning to exploit it in an automated fashion.

Predicting Heuristics

In this section we take the top-scoring search-based solver *2clsQ* (Samulowitz & Bacchus 2006) from the QBF competition (Narizzano, Pulina, & Tacchella 2006). We add 9 new heuristics to the original heuristic. Furthermore, we add the functionality to compute features on the fly for the current theory, and enable the solver to compute the linear classification decision in order to determine the most suitable heuristic for this theory.

We tested all heuristics on all instances and recorded their CPU times. This data was used in part to train the classifier off-line. Therefore, the data was split into a training set (628 instances, including random instances) and test set (576 instances across 20 benchmark families). All parameters were set on the training set (using cross-validation for λ).

We show a summary of the results for each heuristic (H_1, \dots, H_{10}) on each benchmark family in Table 2 contained in the test set. The heuristic H_1 is the original heuristic employed in *2clsQ* and consequently the performance displayed under H_1 is a reference to the state of the art. In addition, we show the performance of the portfolio version in this context. Finally, we also include the results of the solver that dynamically alters its heuristic. In the table we show in the first row the CPU time in seconds required on solved instances. As shown, all versions of the search-based solvers are roughly comparable in terms of CPU time over their solvable instances. We consider instances to be solvable, if they were solved by at least one approach. In the second row we display the average percentage of solvable instances among all solved instances per benchmark family and approach. On this measure all approaches using a fixed heuristic for all instances vary between 70% and 83%.

	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10	PF	DYN
CPU(s)	26,178	29,255	22,757	14,781	26,732	26,722	31,136	25,570	28,079	18,485	23,816	30,530
% SOL	81%	74%	81%	81%	70%	77%	83%	79%	80%	79%	90%	93%

Table 2: Summary of the results achieved by the 10 different heuristics, the portfolio solver (PF), and the dynamic version (DYN) on instances across 20 benchmark families contained in the test set. For each approach the total CPU time required for the solved instances amongst all benchmark families is shown. For each approach the average percentage of solved instances amongst all solved instances per family is shown in the last row.

This variability in performance reflects the degree of variance induced by changes in heuristic only. The table also shows that the portfolio approach – choosing a heuristic on a per-instance basis – is able to significantly outperform any approach employing a fixed heuristic. More importantly, the strategy to dynamically adjust the heuristic performs best among all approaches. In fact, it is able to outperform the best fixed heuristic by 10% on average. Furthermore, it is able to perform better than choosing the best heuristic on a per-instance basis.

In Table 3 and 4 we display more detailed results. Again we show the percentage of solved instances among all instances solved by any approach per benchmark family and approach as well as the CPU times for each approach and benchmark family. Also on these more detailed results the dynamic approach displays a robust performance (e.g., being the best method on 5 benchmark families). Table 3 also shows that our approach of dynamically adjusting the heuristic choice is able to solve instances not solvable by any other approach (see e.g., the K benchmark). Furthermore, Table 4 also shows that the overhead introduced by the dynamic feature extraction and classification is negligible.

In total, our empirical results show that the portfolio approach as well as dynamically adjusting the variable branching heuristics can be a very effective tool for solving QBF.

Conclusions and Future Work

We believe that machine learning can be helpful to a much larger degree when solving hard search-based problems than it is already shown here. With QBF there exist many additional choices besides heuristics (e.g., whether to apply stronger inference/partitioning or not) that, if selected automatically, could drastically improve the performance of a QBF solver. Since the problem of predicting multiple labels simultaneously can entail a combinatorial explosion, recent work on structure prediction (see e.g., (Memisevic 2006) for an overview) could be useful for this purpose.

Further directions for future work include optimal feature selection, and the use of non-linear prediction models. The hardest challenge for the latter will be run-time efficiency, which might rule out kernel-based, and other non-parametric, methods.

References

- Ali, M.; Safarpour, S.; Veneris, A.; Abadir, M.; and Drechsler, R. 2005. Post-verification debugging of hierarchical designs. In *International Conf. on Computer Aided Design (ICCAD)*, 871–876.
- Biere, A. 2004. Resolve and expand. In *Seventh Interna-*

tional Conference on Theory and Applications of Satisfiability Testing (SAT), 238–246.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem-proving. *Communications of the ACM* 4:394–397.

Hastie, T.; Tibshirani, R.; and Friedman, J. 2001. *The Elements of Statistical Learning*. Springer.

Hutter, F.; Hamadi, Y.; Hoos, H. H.; and Leyton-Brown, K. 2006. Performance prediction and automated tuning of randomized and parametric algorithms. In *12th International Conference on Constraint Programming, CP 06*.

Lagoudakis, M., and Littman, M. 2001. Learning to select branching rules in the dpll procedure for satisfiability. In *Workshop on Theory and Applications of Satisfiability Testing (SAT 01)*.

Letz, R. 2002. Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In *Proceedings of Tableaux 2002*, LNAI 2381, 160–175. Springer.

Memisevic, R. 2006. An introduction to structured discriminative learning. Technical report, University of Toronto, Toronto, Canada.

Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient sat solver. In *Proc. of the Design Automation Conference (DAC)*.

Narizzano, M.; Pulina, L.; and Tacchella, A. 2006. QBF solvers competitive evaluation (QBFEVAL). <http://www.qbflib.org/qbfeval>.

Narizzano, M.; Pulina, L.; and Tacchella, A. 2006. The third QBF solvers comparative evaluation. *Journal on Satisfiability, Boolean Modeling and Computation* 2:145–164.

Nudelman, E.; Leyton-Brown, K.; Devkar, A.; Shoham, Y.; and Hoos, H. 2004. Satzilla: An algorithm portfolio for sat. In *Seventh International Conference on Theory and Applications of Satisfiability Testing*, 13–14.

Samulowitz, H., and Bacchus, F. 2006. Binary clause reasoning in qbf. In *Ninth International Conference on Theory and Applications of Satisfiability Testing (SAT 06)*.

Samulowitz, H., and Bacchus, F. 2007. Dynamically partitioning for solving qbf. In *Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT)*.

SHARCNET. 2007. Shared hierarchical academic research computing network. <http://www.sharcnet.ca>.

Zhang, L., and Malik, S. 2002. Towards symmetric treatment of conflicts and satisfaction in quantified boolean satisfiability solver. In *Principles and Practice of Constraint Programming (CP2002)*, 185–199.

BENCHMARK FAMILY	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10	PORTFOLIO	DYNAMIC
ADDER	83	33	83	78	33	83	77	56	50	100	94	67
BLOCKS	75	100	75	75	50	100	100	75	75	75	100	100
C	89	89	100	89	33	100	100	100	100	100	100	89
TOILET	67	67	83	33	67	33	66	100	67	67	83	83
ADDER	88	88	88	80	88	88	92	84	96	88	100	88
COUNTER	75	75	75	75	75	75	75	100	75	75	75	100
EIJK	100	100	100	100	100	100	100	100	100	100	100	100
EV	57	57	57	71	57	57	84	57	100	57	71	100
IRST	100	100	100	100	100	100	100	100	100	100	100	100
K	95	95	95	90	95	90	95	90	90	86	95	100
KEN	100	100	100	100	100	50	100	100	100	50	100	100
LUT	67	67	67	67	67	67	66	67	100	33	67	67
MUTEX	25	25	25	25	25	25	25	25	25	100	75	75
NUSMV	100	100	100	100	100	100	100	100	80	100	100	100
QSHIFTER	57	60	57	57	60	60	60	100	100	100	100	100
S	100	100	100	100	100	100	100	80	100	100	100	100
SORT	94	96	100	92	94	96	94	96	94	94	98	94
SZYMANSKI	100	0	100	100	0	100	100	0	0	13	100	100
TEXAS	50	25	50	100	50	50	50	50	50	50	50	100
TOILET	100	100	71	88	100	65	94	94	100	100	94	88
SUMMARY	81	74	81	81	70	77	83	79	80	79	90	93

Table 3: Success rates achieved by the 10 different heuristics, the portfolio solver, and the dynamic version on instances across 20 benchmark families contained in the test set. For each benchmark and approach the percentage of solved instances among all solved instances per family is shown. For each family the solver with highest success rate is shown in bold, where ties are broken by time required to solve these instances. The summary line shows the average success rate over all benchmark families.

BENCH	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10	PF	DYN
ADDER	2754	513	4041	387	246	5012	669	7366	6844	2412	2090	1121
BLOCKS	1	3077	3	0	1	161	2002	1	1	4	3077	4413
C	0	0	1	0	0	1	1	1	1	1	1	1
TOILET	450	424	55	116	579	206	749	832	1344	1972	1973	4159
ADDER	6234	8835	2790	938	9767	788	5332	2187	5652	2355	5332	4478
COUNTER	10	10	0	227	10	9	38	903	16	10	16	732
EIJK	255	244	1621	386	191	1782	89	20	4	5	89	255
EV	68	104	35	4423	183	66	3859	24	22	149	96	39
IRST	4	4	2	8	4	2	1	1	1	1	1	1
K	7207	5794	6664	5709	5872	5409	9270	9274	10567	4306	5042	11410
KEN	0	0	0	0	0	0	0	0	0	0	0	0
LUT	0	0	207	4	0	302	3	4	89	0	0	0
MUTEX	0	0	0	0	0	0	0	0	0	1	0	0
NUSMV	22	21	1149	21	21	936	1678	606	342	175	749	938
QSHIFTER	13	2785	13	13	2763	3430	2602	48	41	19	20	23
S	0	0	0	0	0	0	0	0	0	0	0	0
SORT	2216	2640	3252	58	26	4809	2620	3589	1225	57	3250	31
SZYMANSKI	1805	0	1944	1122	0	1133	1103	0	0	4492	1133	1900
TEXAS	0	0	0	0	0	0	0	0	0	0	0	0
TOILET	4090	3847	1	438	6014	1630	3	3	923	1768	15	16
SUMMARY	26178	29255	22757	14781	26732	26722	31136	25570	28079	18485	23816	30530

Table 4: CPU times achieved by the 10 different heuristics, the portfolio solver (PF), and the dynamic version (DYN) on instances across 20 benchmark families contained in the test set. For each benchmark and approach the CPU times used on solved instances is shown. The summary line shows the total CPU time over all benchmark families.