Machine Learning of Strategies in QBF Solving

Ricardo Joel M. dos Santos Silva[1] and Mikoláš Janota[1,2][0000−0003−3487−784X]

[1] IST/INESC-ID, Universidade de Lisboa
[2] Czech Technical University in Prague

Abstract. `QFun` is a QBF solver based on Counterexample Guided Abstraction Refinement (CEGAR) that uses machine learning to learn strategies for variables during solving.This paper focuses on the research, development, implementation and evaluation of alternative machine learning algorithms for `QFun`. We have implemented six alternative algorithms for the learning part of the solver and have experimentally evaluated their performance.

Keywords: QBF · Machine Learning · decision list.

1 Introduction

The *Quantified Boolean Formula (QBF)* [3] is a generalization of SAT, in which either existential or universal quantifier is applied to a variable. There has been a growing interest in QBF solving over the last two decades, motivated by the practical success of SAT solvers. There are two main families of QBF-solvers: *DPLL-based* [7,8,13] and *expansion based* [1,2,14]. Expansion-based solvers eliminate quantifiers by expanding into Boolean connectives and use a SAT solver to solve the resulting propositional formula.

Recent expansion-based QBF solvers expand to SAT *gradually* to mitigate the exponential blow up size. Gradual expansion is achieved by using the paradigm of *Counter-Example Guided Abstraction Refinement (CEGAR)*. In the CEGAR approach the formula to solve is approximated by an abstraction. The CEGAR loop begins by solving the abstraction with the use of a SAT solver. If the abstraction has no solution, then the problem also has no solution. If the abstraction has a solution, that solution is a candidate solution to the general problem, and we test it with a new SAT call. If it is a solution for the original formula, the algorithm stops. If it is not, then the SAT call provides us with a counterexample, which is used to refine the abstraction to be used in the next iteration of the loop.

This CEGAR approach underlies the 2-QBF solver AReQS [12], which was later generalized to arbitrary number of quantifiers in RAReQS [11,10]. The solver `QFun` [9] improves RAReQS by using *machine learning* during solving. `QFun` enables refining the abstraction with not only a single counterexample but by a strategy learned from *multiple* counterexamples. The contribution of this paper is the investigation of the performance of different machine learning approaches within `QFun`. [3]

[3] This paper is based on the results obtained in the MSc thesis of the first author [18].

Algorithm 1: `QFun`²: 2-level QBF Refinement with Learning

input : $\exists X \forall Y. \phi$ where ϕ is propositional.
output : a winning move for $\exists X$ if exists, $\bot$ otherwise

```
1  E ← ∅                                    // start with no samples
2  α ← true                                 // empty abstraction
3  while true do
4      τ ← SAT(α)                           // candidate for X
5      if τ = ⊥ then return ⊥               // Q loses
6      μ ← SAT(¬φ[τ])                       // countermove
7      if μ = ⊥ then return τ               // τ is winnning
8      E ← E ∪ {(τ, μ)}                     // record sample
9      if ¬ShouldLearn() then
10         α ← α ∧ φ[μ]                     // refine with μ
11     else
12         S ← Learn(E)                     // learn a strategy
13         α ← α ∧ φ[S]                     // refine with S
14         E ← ∅                            // reset samples
```

2 QFUN Algorithm

Algorithm 1 shows pseudocode for `QFun`²—`QFun` restricted to the form $\exists X \forall Y. \phi$, with variable vectors X and Y. The general algorithm `QFun` applies `QFun`² recursively just as RAReQS [11,10]. The algorithm is anchored in the game-perspective on QBF, where $\exists$ aims to make the formula true and $\forall$ false.

The winning move for $\exists X$ is sought for by picking a *candidate move*, which is a satisfying assignments to the abstraction α. Any candidate move that is not a winning move, has a *counter-move*, which is used to strengthen (refine) α by substituting the counter-move into ϕ. If learning is employed, refinement is based on what was learned from multiple counter-moves.

Without learning, the algorithm can easily have poor behavior. For instance, $\exists x_1 \ldots x_n \forall y_1 \ldots y_n. \bigvee_{i \in 1..n} x_i = y_i$ requires 2^n iterations of the loop, even though the formula is obviously false once y_i plays $\neg x_i$. Hence, we are looking for good strategies for Y. More formally, a *strategy* for $\forall Y$ is a multi-valued Boolean function $S : X \to Y$, giving values to Y based on the move played by $\exists X$. In practice, a strategy is represented by a set of Boolean functions—one for each variable $y \in Y$; each of these functions is represented by a Boolean formula. *Refinement by a strategy* then consists of substituting the strategy formula S_y for the corresponding variable $y \in Y$. This enables enables solving the formula above in linear number of SAT calls.

New strategies are obtained by learning on previous candidates and counter-moves. At each learning moment, the collected pairs of candidate and counter-moves are used as training set of a machine learning algorithm. The original implementation of `QFun` [9] uses the `ID3` algorithm [16], which learns strategy formulas in the form of decision trees.

Algorithm 2: Pseudo-code for ID3

```
   Function ID3(E, domain)
   input  : E = {(τ_1, y_1), (τ_2, y_2), ..., (τ_m, y_m)}, domain is a subset of X
   output : decision tree, representing a strategy for y
1  if E = E^+ then return leaf node with label 1
2  if E = E^- then return leaf node with label 0
3  if domain = ∅ then
4  |  return leaf node labeled with the most common value of y in E
5  sv ← arg max_{x∈domain}(IG(E, x))
6  for b ∈ {0, 1} do
7  |  let E_b = {(τ_i, y_i) ∈ E : τ_i(sv) = b}
8  |  DT_b ← ID3(E_b, domain\{sv})
9  return sv ? DT_1 : DT_2
```

3 Learning Algorithms

The general QFun receives $\Psi = Qx_1, \ldots, x_{n_1}\overline{Q}y_1, \ldots, y_{n_2}\Phi$, with $Q \in \{\exists, \forall\}$, which is solved following the logic in Algorithm 1. Each iteration of the loop produces a move assigning values to $X = \{x_1, \ldots, x_{n_1}\}$ and a counter-move assigning values to $Y = \{y_1, \ldots, y_{n_2}\}$.

After m iterations of the loop, m move/counter-move pairs are stored by the solver in the variable E. The function Learn then takes E as input and computes a strategy for each of the $y \in Y$ (line 12).

Since strategies are computed one variable at a time, we consider learning algorithms that obtain samples in the form (τ, v) where τ is an assignments to X and v a a value for some target variable $y \in Y$. The strategy is then learned by calling learning for each $y \in Y$. After the abstraction is refined with the learned strategy, the set of samples is emptied, and the solver continues with solving.

We will call each element $e \in E = \{(\tau_1, y_1), (\tau_2, y_2), \ldots, (\tau_m, y_m)\}$ an *example*. We will say that an example is positive if $y_i = 1$ and a negative if $y_i = 0$, denoting the sets of positive and negative examples by E^+ and E^-, respectively.

One of the findings of [9, p. 10] was that QFun's performance improves significantly if previous strategies are kept, while they correctly classify the new set of examples. We maintain this approach.

3.1 Decision trees and ID3

In the original implementation of the solver, the function Learn uses the ID3 algorithm [16], presented in Algorithm 2.

The ID3 algorithm has a recursive nature. The base case of the recursion is when all the given examples are in the same class. The general case picks a variable sv which splits the examples on those where sv is 1 and those where sv is 0, reducing thus the domain. These two sets are then classified recursively.

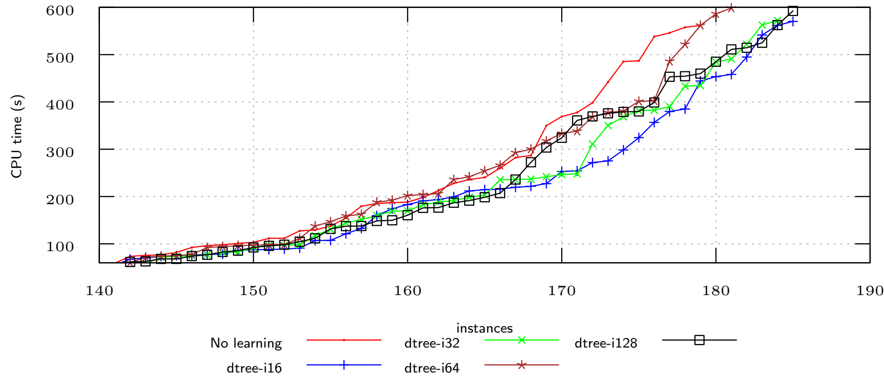

Fig. 1. ID3 results

The algorithm needs to account for the possibility that runs out of variables to split on. In this case there is no perfect solution, and the typical solution is to label the root with the most common value of the y_i in E. However, in the context of `QFun` this does cannot happen.

The `ID3` algorithm chooses the splitting variable `sv` by maximizing the *information gain* of the split of E. Information gain derives from *entropy*, which measures the observed uncertainty. Given a set of samples E, the minimum value for the entropy of E is 0, when all elements in E are all positive or all negative. The maximum value for the entropy of E is 1, if exactly half of its elements are positive and half negative. Information gain measures the reduction in uncertainty, i.e., the reduction of entropy, caused by the observation of a variable.

Definition 1. ***(Entropy)*** *Given a set of samples E, the entropy of E is given by:* $\mathtt{H}(E) = \sum_{b \in \{0,1\}} -p(y = b) \log_2 p(y = b)$.

Definition 2. ***(Information Gain)*** *Given a set of samples E, and a variable $x \in X$, the information gain of E for variable x is the difference between the entropy of E and the entropy of E given x:* $\mathtt{IG}(E, x) = \mathtt{H}(E) - \mathtt{H}(E|x)$.

After generating a decision tree that correctly classifies all the examples, `QFun` translates it into a Boolean formula.

We tested the solver both without learning, and learning with `ID3`. The results obtained, presented in Figure 1, will serve us as a benchmark against which our other algorithms will be compared.

All tests were conducted on 4 identical Linux machines, with Intel Xeon 5160 3GHz processors and 4GB of memory. The memory limit used was 2 GB, and the time limit used for each test was 600 seconds. We have used a set of 484 QBF formulae obtained by joining the two sets used for the evaluation of the competing solvers in QBFEval'17 and in QBFEval'18 (Prenex Non-CNF track) and filtering out the formulae we identified to be randomly-generated.

The reason for removing randomly generated benchmarks is that the motivation for learning in QBF is to identify *structure in practical problems*, whereas random problems will inherently have none. The set obtained was used consistently throughout this section, for all evaluation purposes.

`QFun` with `ID3` is more effective than `QFun` without learning. This had been observed in [9] and was confirmed by our own results. Overall, we believe the choice of `ID3` is sensible, and there are a number of arguments in favor of it. The conversion between trees and formulas can be done efficiently. Learning itself is not too heavy, in the sense that the time spent in learning is generally small (between 1.4% and 2.3%) compared to the total time of execution of the solver. Learning reduced considerably the number of refinements. However, the improvement in results with learning versus without learning is small. The limited improvement in efficiency is not related to a trade-off between time spent learning and time saved in resolution. More important than the computational time spent learning, there is a trade-off between the benefit of learning, which often reduces the number of refinements needed to reach the solution to a formula, and the drawback of learning, in terms of the increased complexity of the abstractions we use, making the computation of new refinements heavier and more time-consuming.

The results achieved by `QFun` with `ID3` do not mean that better alternatives for learning could not be found. One possibility that we decided to explore in order to improve the solver was to use decision lists instead of decision trees. This approach is the one we will develop in the subsequent sections.

3.2 Decision Lists

Decision lists, introduced by Rivest [17], have the aim of avoiding the *replicated subtree problem* [15,6]: in some situations identical subtrees have to be learned repeatedly at various places in a decision tree. Rivest further proved that k-decision lists are PAC learnable and provided an algorithm for constructing a decision list consistent with a given set of data [17].[4] We have altered `QFun`, implementing 6 different algorithms using decision lists, that we present next:

1. Rivest's original algorithm [17, p. 243-244];
2. Pagallo and Haussler's `Greedy3` algorithm, introduced in [15];
3. Pagallo and Haussler's `Grove` algorithm [15];
4. `Laplace`, a `separate-and-conquer` algorithm, based on Pagallo and Haussler's `Grove`, but using Laplace estimate instead of entropy;
5. `Simple`, a `separate-and-conquer` algorithm with a `select-rule` auxiliary function designed for our specific case;
6. `CN2`, a `separate-and-conquer` algorithm relying on beam-search.[5,4]

Rivest

We applied Rivest's original algorithm for learning decision lists.

[4] We recall that PAC-learning was introduced by Valiant [19].

Algorithm 3: Pseudo-code for `Rivest`

```
  Function Rivest(E, max)
  input  : E = {(τ_1, y_1), (τ_2, y_2), ..., (τ_m, y_m)}
           max is a positive integer
  output : DL, a strategy for y
1 for t ∈ C^n_max do
2     T ← {(τ_i, y_i) : τ_i ⊨ t}
3     if T = ∅ then continue
4     if T ⊂ E^+ then return {(t, 1)} . Rivest(E \ T, max)
5     if T ⊂ E^- then return {(t, 0)} . Rivest(E \ T, max)
6 let (τ_i, y_i) belong to the smaller class of E
7 return (τ_i, y_i) . Rivest(E \ {(τ_i, y_i)}, max)
```

The algorithm works recursively: it iterates over all terms of length at most max, until it finds a term t such that all the examples satisfying it are of the same class. We start our decision list with a rule consisting of t and the corresponding class. The remaining rules are computed by recursively calling the algorithm.

We modified the original algorithm, by introducing a *fall-back rule* (steps 6 and 7): when no t satisfies the requirement, we add a rule corresponding to an example from the less common class in E (Algorithm 3).

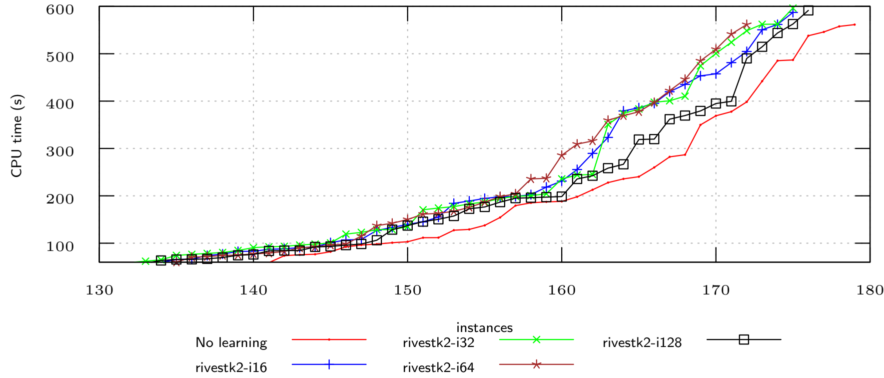

Fig. 2. Rivest results, with max $= 2$

`QFun` with `Rivest` performs considerably worse than with `ID3` and even worse than `QFun` without any learning (Fig. 2).

Interestingly, the fall-back rule was very seldom used. That means there is no point in evaluating `Rivest` with max $= 3$. We therefore tested `Rivest` with max $= 1$ instead, with similar, slightly worse results.

Algorithm 4: Loop of `separate-and-conquer` algorithms

Function `separate-and-conquer`(E)
input : $E = \{(\tau_1, y_1), (\tau_2, y_2), \ldots, (\tau_m, y_m)\}$
output : DL, a strategy for y

1. $C \leftarrow E$
2. $DL \leftarrow \emptyset$
3. **while** $C^+ \neq \emptyset$ ***and*** $C^- \neq \emptyset$ **do**
4. $rule \leftarrow$ `select-rule`(C)
5.   **Append** $rule$ **to** DL
6. $C \leftarrow C \setminus \{e \in C \mid e \text{ is covered by } rule\}$
7. $c \leftarrow$ class common to all examples in C
8. **Append** (true, c) **to** DL
9. **return** DL

The algorithm does not apply any heuristic to guide the search for new terms to add to our decision list. This has two major drawbacks: at each step we will learn the first term that is coherent with our examples, not necessarily the best one; and for a large n or max, considering all the possible terms is too inefficient. We conclude that Rivest's algorithm is not efficient for practical purposes, and we will try different algorithms in the upcoming sections.

Greedy3

There are a number of different search strategies that can be used to avoid generating all possible rules and checking them one by one, while also striving to learn the best rule possible. Pagallo and Haussler [15] have used a top-down greedy approach. For each new rule, the term is built by starting with `true` and adding one literal at a time, using a heuristic to choose the best literal. They used a new strategy, called `separate-and-conquer`, and `Greedy3` became the first example of a large family of learning algorithms sharing that same strategy. We begin by presenting the `separate-and-conquer` general strategy before looking at the more specific `Greedy3`.

All `separate-and-conquer` algorithms share the same top-level loop: a `separate-and-conquer` algorithm selects a rule that classifies a part of the examples, removes (*separates*) these examples from the set of examples to classify, and recursively classifies (*conquers*) part of the remaining examples until no examples remain.

The pseudo-code for the `separate-and-conquer` family of algorithms is presented as Algorithm 4. The algorithm starts with an empty decision list, DL and a set of unclassified examples C. As long as C includes both positive and negative examples, the algorithm adds rules to DL, and eliminates the examples covered with these rules from C. Once C has only positive or only negative examples left, the default rule is added to DL and the algorithm ends.

The difference between the specific algorithms of the `separate-and-conquer` paradigm is in the way they select the next rule to be added to the decision list

Algorithm 5: `select-rule` for hill-climbing `separate-and-conquer`

Function `select-rule`(C)
input : C is a set of examples
output : a rule

```
1  Pot ← ∅
2  term ← true
3  domain ← X
4  while C⁺ ≠ ∅ and C⁻ ≠ ∅ do
5  |   l ← select-literal(C, domain)
6  |   term ← (term ∧ l)
7  |   domain ← domain \ {var(l)}
8  |   remove from C all examples with value 0 for l and add them to Pot
9  c ← class of remaining examples in C
10 C ← Pot
11 return (term, c)
```

(step 4). Most `separate-and-conquer` algorithms use *hill-climbing*, building the term of the new rule by adding one literal at a time, selecting the best literal according to some criterion, and stopping when no improvement is possible.

The pseudo-code for the `select-rule` auxiliary function using this hill-climbing search strategy is presented as algorithm 5. Every time we want to learn a new rule, an auxiliary set of examples Pot is initialized as empty, the *term* to be learned is initialized as `true`, and the set of variables we can use, domain, is set to X. We then repeatedly select a literal l, update *term* to be the intersection of *term* and l, exclude l's variable from the domain and move all the examples that don't satisfy l from C to Pot. We stop selecting literals when all examples in C belong to the same class c. At that point, we append $(term, c)$ to the decision list and the examples stored in Pot become the new C.

Within this hill-climbing strategy, learning algorithms differ only in the way they select the next literal to be used in a rule (step 5). This is usually done using a `select-literal` function that assigns a value to each literal and then chooses the literal that maximizes that value. `Greedy3`'s select-literal function chooses the next literal using as a criterion the measure of *validity* or *purity* of a literal, i.e., the probability that an example is a member of a class given that it satisfies that literal.

The results obtained by `QFun` with `Greedy3` can be seen in Fig. 3. Overall, it works better than `Rivest`, but does not match the performance of `ID3`. Due to its `select-literal` function, `Greedy3`'s decision lists consist of a sequence of rules with class 1, ending with a default rule with class 0. `Greedy3` was designed with the goal to allow efficient representation of small DNF formulae (which was a limitation of decision trees). However, the fact that `Greedy3` is always trying to maximize the positive elements leads to inefficiency in some instances.

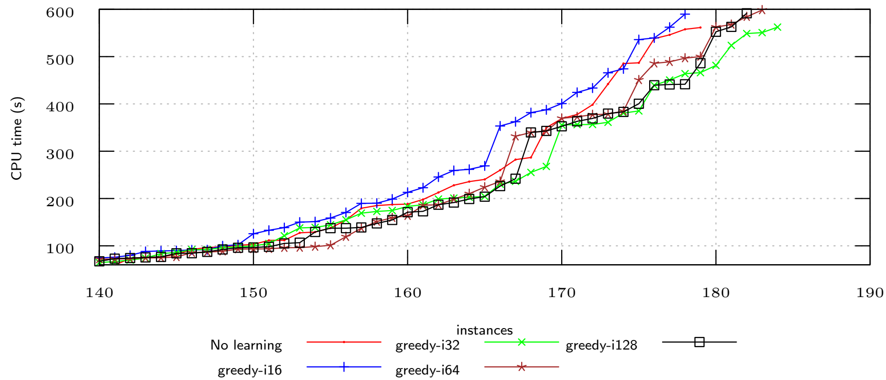

Fig. 3. Greedy3 results

Grove

The `Grove` algorithm is another `separate-and-conquer` algorithm, introduced with the goal of efficiently learning a decision list without `Greedy3`'s bias. The difference between them is only in the `select-literal` auxiliary function. `Grove` chooses the next variable to be used by maximizing the Information Gain of C for x, and then chooses between the positive and the negative literal of that variable, according to which one has a smaller entropy.

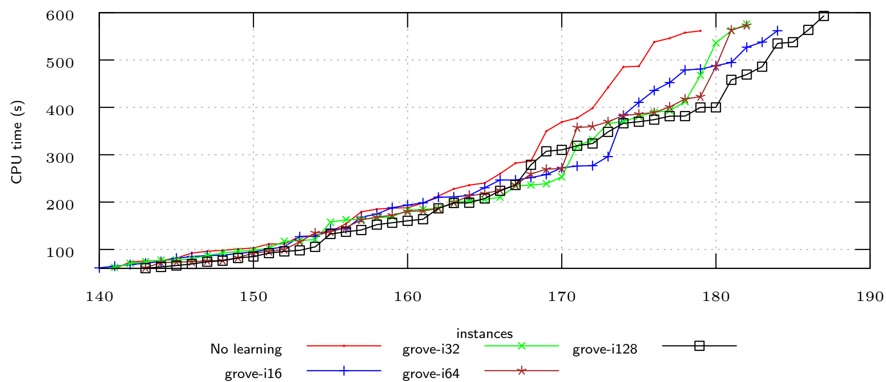

Fig. 4. Grove results

Overall, results were better than the ones obtained with `Greedy3`, and very similar to the ones obtained with `ID3`. They can be seen in Figure 4.

Using entropy as heuristic created a bias towards learning rules that are too specific. This problem has been identified in the literature, sugesting the replacement of entropy by the `Laplace` estimate. We therefore decided to create a version of `Grove` using `Laplace` estimate instead of entropy.

Laplace

To counter the bias found in the previous section, we have adapted `Grove`'s algorithm to use *Laplace Estimate* instead of Entropy, and we have called this algorithm `Laplace`.

Definition 3. ***(Laplace Estimate)*** *Given a set of samples* $E = \{(\tau_1, y_1), (\tau_2, y_2), \ldots, (\tau_m, y_m)\}$*, a literal* l*, and a Boolean* b*, the Laplace Estimate is given by:* $\texttt{Laplace}(E, l, b) = \frac{p+1}{p+n+2}$*, where* p *and* n *are the number of examples that satisfy* l *in* $\{e \in E : y_i = b\}$ *and* $\{e \in E : y_i \neq b\}$*, respectively.*

The results obtained with `Laplace` were better than the results obtained with `Grove` and with `ID3`. It was the best of the algorithms used so far, and these results can be seen in Figure 5. In terms of the number of formulae solved, results were better than for any algorithm previously tested. `Laplace` performed better than `Grove` and `ID3` at most of the possible metrics considered: solved instances, total solver time, learning time.

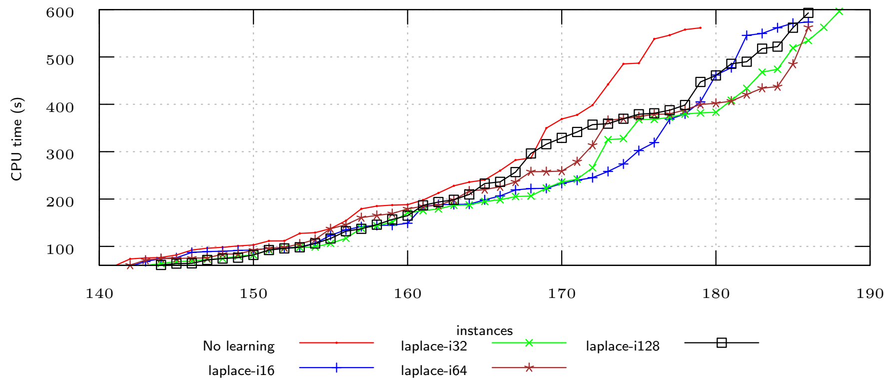

Fig. 5. Laplace results

Simple

During the implementation of `Laplace`, the idea of creating a heuristic tailored to our specific problem was developed. It was clear to us how it should evaluate literals. Given a set of examples E, divided in positive and negative examples, and a literal l, we wanted to attribute a value to l based on the result of subdividing E^+ and E^- according to the evaluation of l:

– ideally, the set of examples that satisfy l, E_l, coincides with E^+ or E^-;
– alternatively, $E_l \subset E^+$ or $E_l \subset E^-$;
– the remaining literals should be ordered by how unmixed E_l is;
– an additional ordering criterion, should be to maximize the number of examples covered.

We defined a function that for every set of examples and a literal returns a pair. The first element is -1, 0 or a positive integer, depending on whether l belongs to the first, second or third of the 3 groups described above. In the third case, the integer measures how far E_l is from being unmixed. We decided to use the minimum value between $\#E_l^+$ and $\#E_l^-$. The second element in the pair is the number of examples covered. We want to choose the literal with the lowest possible value in the first element, and for literals with the same first value, we want the highest possible second value.

Definition 4. ***(Simple)*** *For a set of samples E, and literal l, the function Simple is defined as:* $\texttt{Simple}(E, l) = \textbf{\textit{if}}(E_l = E^+ \ \textbf{\textit{or}}\ E_l = E^-)\ \textbf{\textit{then}}\ (-1, p+n)$ $\textbf{\textit{else}}\ (min(p, n)\ ,\ p+n)$
where p and n are the number of examples satisfying l in E^+ and E^-, resp.

We choose the ordering of our pairs by defining: $(a, b) < (c, d)$ iff $(a < c) \vee ((a = c) \wedge (b > d))$.

The `select-literal` function of `Simple` is the minimization of the Simple function defined above. Simple is in reality a particular member of the lexicografic evaluation functionals [6, p. 35].

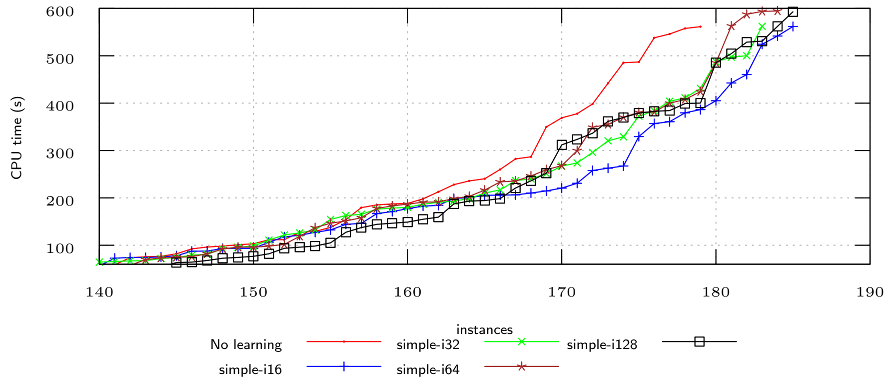

Fig. 6. Simple results

`Simple` did not perform better than `Laplace`, but at the level of `Grove` and `ID3`. The results obtained by `Simple` and `Grove` were very similar, in all the metrics considered, except the learning time, where `Simple` performed clearly

better. We conclude that our heuristic led to a faster learning, but not to a more relevant learning.

`Simple` has a basic limitation, inherent to the hill-climbing strategy: by eliminating all possible one-step choices but one, hill-climbing may be unable to reach the optimal solution in the search space, because it gets stuck in a local maximum. There are at least two possible ways to address this problem. One is *look ahead*. In our concrete case, this would mean to evaluate all possible combinations of up to n literals, instead of choosing them one by one. The other is *beam-search*: instead of remembering only the best solution, we keep in memory a fixed number of alternatives, the so-called *beam*. While hill-climbing has to make a choice of a single solution, and look ahead causes an exponential growth in the search space, beam-search causes only a multiplication of the search space by a constant factor, the size of the beam. From these two approaches, we chose beam-search, since keeping the size of the search space controlled is a relevant constraint in our problem. For that reason, we decided to implement the `CN2` algorithm, which we cover in the next section.

CN2

The `CN2` algorithm is another `separate-and-conquer` algorithm, but does uses beam-search instead of hill-climbing. The pseudo-code for `CN2`'s `select-rule` auxiliary function is presented as Algorithm 6.

Algorithm 6: `select-rule` for `CN2`

```
   Function select-rule(C)
   input  : C is a set of examples
   output : a rule
1  let s, ns, bt ← {true}, ∅, true
2  while s ≠ ∅ do
3      foreach t ∈ s do
4          if term is coherent then continue
5          let dom ← {x ∈ X | x ∉ t}
6          foreach l ∈ {x, ¬x | x ∈ dom} do
7              let nt ← t ∧ l
8              if Laplace(nt) > Laplace(bt) and nt is coherent then
9                  bt ← nt
10             ns ← ns ∪ {nt}
11             if size of nt > maxs then
12                 remove the worst term from ns
13     s ← ns
14     ns ← ∅
15 c ← class of examples that satisfy bt
16 return (bt, c)
```

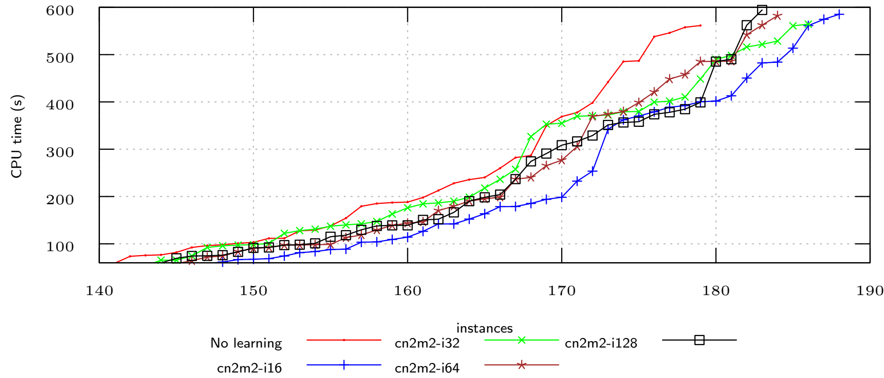

Fig. 7. CN2 results, beam size 2

We begin by initializing 3 variables: s, ns and bt. The variable s holds the beam, i.e., the best terms found so far, and is initialized as the singular set containing only the `true` term. Throughout the execution of `CN2`, the size of s cannot exceed a constant `maxs`. The variable ns holds an auxiliary set so that we can iteratively update s, and is initialized as the empty set. The variable bt identifies the best term found so far and is initialized to `true`.

We then start a loop that will only stop when s is empty. In each iteration, we expand every term in s (unless it is already coherent) with every possible literal whose variable is not yet in t.

For every expansion of a t with a new literal, the resulting nt is evaluated with the `Laplace` heuristic. If nt is better than the current bt and nt is coherent, we update bt. If ns is smaller than `maxs` we add nt to it, otherwise, if nt is better than the worst term in nt, we replace that worst term with nt.

After iterating through all terms in s and all possible literals for each t, we update s to ns and reset ns to $\emptyset$, and we move onto the next iteration of the external cycle. We stop the cycle, once s is empty, which means that we have not expanded any terms in the last cycle. At this point we return a rule consisting of bt and the class of its examples.

One concern we had was whether using beam search would make the learning process computationally heavier. We have tested `CN2` for the same learning intervals as the previous algorithms, and for beam sizes of 2, 3 and 4. Overall, the results with `CN2` were slightly better than with `Laplace`, and therefore it was the best of all the algorithms we tried. Once again the difference was modest, but in the end by these small increments we got observably far from the results obtained without learning.

Figures 7, 8, and 9, show results obtained with the `CN2` algorithm, for a beam size of 2, 3, and 4, respectively. In Figure 10, we can see the results obtained with all algorithms, combining the best version of each. We have also added the results

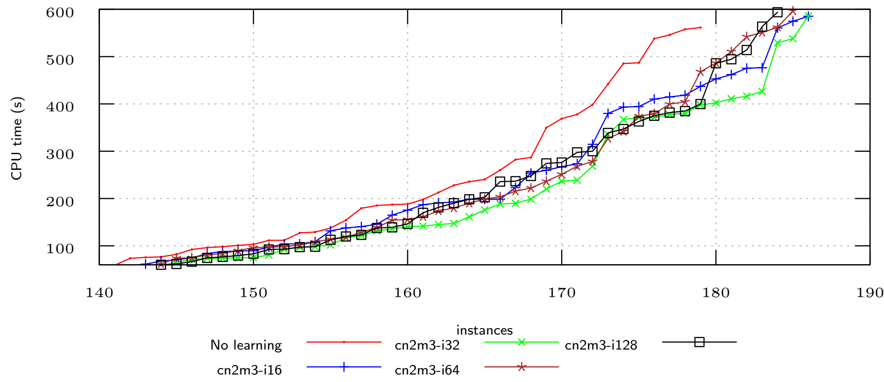

Fig. 8. CN2 results, beam size 3

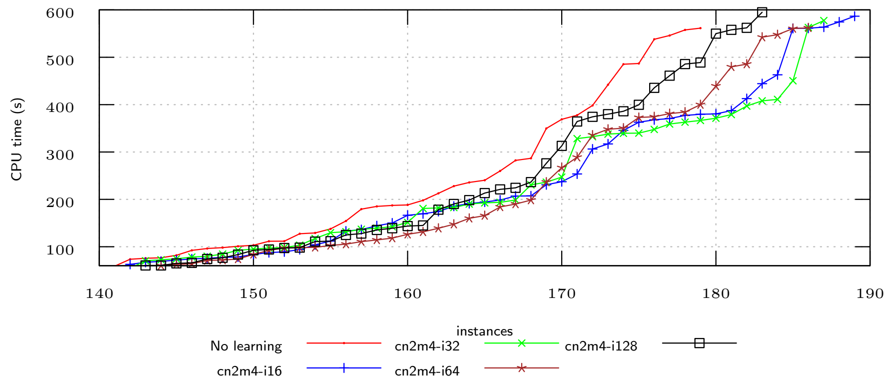

Fig. 9. CN2 results, beam size 4

obtained with the QuAbS solver, the winner of the QBFEval18 competition.[5] A detailed table with all the results obtained is publicly available.[6] Further experimental details can be found in the MSc thesis of the first author [18].

4 Summary and Future Work

We have amply confirmed the main result of [9], that consists in claiming that machine learning of strategies during the solving of QBF with CEGAR enables improvements in the solver's performance. The learning of strategies results in a smaller number of refinements and therefore, in principle, in faster solving. We

[5] https://github.com/ltentrup/quabs

[6] http://sat.inesc-id.pt/~rjs/qbf/qbfEval17-18_non-random_600.html

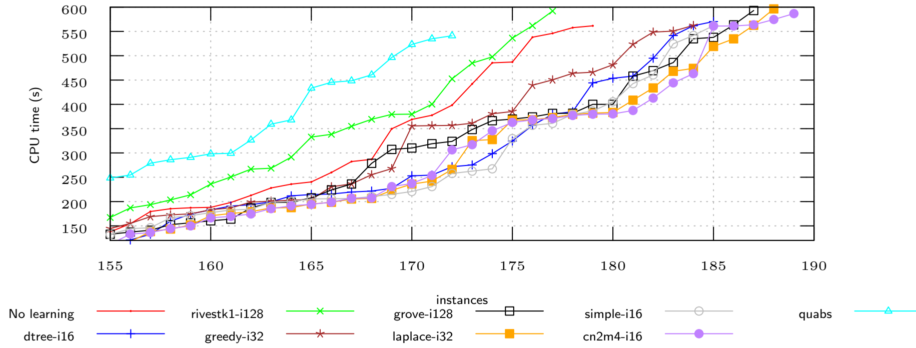

Fig. 10. All algorithms (best version of each)

conclude that whether the reduction in the number of refinements necessary to solve a QBF materializes into a faster solving time is formula-specific.

We conclude that the quality of the strategies learned is the crucial factor, and not so much the efficiency in learning. Efficiency of the learning algorithm is also important, but in our implementation, the learning process only takes a small fraction of the total solving time. `QFun` with learning can result in computationally heavier refinements, causing the solver to slow down. In many of the instances that `QFun` solved without learning and failed to solve with learning, the reason for failure was a timeout and `QFun` with learning did not reach the number of refinements at which it was able to solve the formula without learning.

The results obtained with `CN2` show that using beam search to select literals is feasible. This in turn leads us to conclude that more complex learning algorithms might be suitable candidates to improve `QFun`.
Possible future work includes:

- Learn strategies for multiple variables at once.
- Implement a look-ahead algorithm
- Dynamic learning intervals
- Incremental learning
- Improve the analysis of QBF families

Acknowledgments. This work was supported by national funds through FCT — Fundação para a Ciência e a Tecnologia with reference UID/CEC/50021/2019 and the project INFOCOS with reference PTDC/CCI-COM/32378/2017. The work was supported by the European Regional Development Fund under the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).

References

[1] Benedetti, M.: Evaluating QBFs via symbolic Skolemization. In: LPAR (2004)
[2] Biere, A.: Resolve and expand. In: SAT (2004)
[3] Büning, H.K., Bubeck, U.: Theory of quantified boolean formulas. In: Handbook of Satisfiability. IOS Press (2009)
[4] Clark, P., Boswell, R.: Rule induction with CN2: Some recent improvements. In: Proceedings of the European Working Session on Learning (EWSL). pp. 151–163. Springer (1991)
[5] Clark, P., Niblett, T.: The CN2 induction algorithm. Machine Learning **3**(4), 261–283 (1989)
[6] Fürnkranz, J.: Separate-and-Conquer Rule Learning. Artificial Intelligence Review **13**(1), 3–54 (1999). https://doi.org/10.1023/A:1006524209794
[7] Giunchiglia, E., Marin, P., Narizzano, M.: QuBE 7.0 system description. Journal on Satisfiability, Boolean Modeling and Computation **7** (2010)
[8] Goultiaeva, A., Bacchus, F.: Exploiting QBF duality on a circuit representation. In: AAAI (2010)
[9] Janota, M.: Towards Generalization in QBF Solving via Machine Learning. In: AAAI Conference on Artificial Intelligence. pp. 1–14 (2018)
[10] Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.: Solving QBF with counterexample guided refinement. Artificial Intelligence **234**, 1–25 (2016)
[11] Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.M.: Solving QBF with counterexample guided refinement. In: Theory and Applications of Satisfiability Testing (SAT). pp. 114–128 (2012)
[12] Janota, M., Marques-Silva, J.: Abstraction-based algorithm for 2QBF. In: SAT (2011)
[13] Klieber, W., Sapra, S., Gao, S., Clarke, E.M.: A non-prenex, non-clausal QBF solver with game-state learning. In: SAT (2010)
[14] Lonsing, F., Biere, A.: Nenofex: Expanding NNF for QBF solving. In: SAT (2008)
[15] Pagallo, G., Haussler, D.: Boolean Feature Discovery in Empirical Learning. Machine Learning **5**(1), 71–99 (1990). https://doi.org/10.1023/A:1022611825350
[16] Quinlan, J.R.: Induction of Decision Trees. Machine learning **1**(1), 81–106 (1986). https://doi.org/10.1023/A:1022643204877
[17] Rivest, R.L.: Learning Decision Lists. Machine Learning **2**(3), 229–246 (1987). https://doi.org/10.1023/A:1022607331053
[18] dos Santos Silva, R.J.M.: Machine learning of strategies for efficiently solving QBF with abstraction refinement. Master's thesis, Universidade de Lisboa (2019), `http://sat.inesc-id.pt/~mikolas/R-Silva-MSc19.pdf`
[19] Valiant, L.G.: A Theory of the Learnable. Communications of the ACM **27**(11), 1134–1142 (1984). https://doi.org/10.1145/1968.1972