Machine learning of strategies for efficiently solving QBF with abstraction refinement

Ricardo Joel Marques dos Santos Silva

Instituto Superior Técnico, Universidade de Lisboa, Portugal

July 23, 2019

Abstract

`QFUN` is a QBF solver based on Counterexample Guided Abstraction Refinement (CEGAR) that uses machine learning to learn strategies for variables during QBF solving, opening new possibilities for improvement in solver efficiency.

Our work 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. Additionally, we evaluate the performance of the solver `QFUN`, employing the different learning algorithms, on specific QBF families and identify families where this approach is efficient, analyzing also which learning algorithms perform better on which families.

We conclude from the experimental results that learning improves considerably the solver's performance, for some families of formulae.

1 Introduction

The Quantified Boolean Formula Problem, abbreviated QBF, is a generalization of SAT, in which either the existential or the universal quantifiers can be applied to each variable. There has been a growing interest in QBF solving over the last two decades, motivated by the practical success of SAT solvers.

Modern QBF algorithms build on the success of SAT solvers, while implementing specific techniques. There are two main families of QBF-solvers: DPLL-based and expansion based. Expansion-based solvers expand the quantified formula, eliminating the quantifiers, and then use a SAT solver to solve the resulting propositional formula.

Expansion-based QBF solvers, must do the expansion gradually, otherwise the formula will grow exponentially. One possible way to expand the formula carefully is by using the paradigm of Counter-Example Guided Abstraction Refinement, developed by Clarke et al [3]. In the CEGAR paradigm, the formula to solve is represented 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 foprmula, our work is finished. If it is not, then the SAT call provides us with a counterexample, that we can use to refine the abstraction and start the next iteration of the loop. This is the idea underlying to the expansion-based solver for 2-QBF AReQS [7], and the solver RAReQS [6], which generalises AReQS to an arbitrary number of quantifiers through recursion, therefore creating a complete and sound algorithm for QBF.

The QBF solver `QFUN` [5] improves RAReQS by using machine learning during the solving: `QFUN` interleaves refinement with a counterexample and refinement with a formula obtained by learning.

`QFUN`'s general algorithm is presented as Algorithm 1. The input to the algorithm is a multi-game, and its output is a winning move for the multi-game, if there is one, or $\perp$ in case there isn't. The algorithm works recursively, and the base of recursion is a multi-game of the form $\exists X.\{\phi_1 \dots \phi_n\}$ or $\forall X.\{\phi_1 \dots \phi_n\}$, where each of the ϕ_i is propositional.

In the base case, the multi-game is expanded into a propositional formula, which can then be evaluated with a single `SAT` call.

In the general case, it initializes the sets of samples and the abstraction and then enters a cycle. The cycle ends when a winning move has been found for the input, or when it was proven that such move does not exist.

In each iteration of the cycle, the following steps are taken:

1. a winning move to the abstraction is computed, (if none exists the algorithm returns $\perp$),
2. a move for the input multi-game, τ, is created

Algorithm 1: Pseudo-code for QFUN

Function QFUN($QX.\ \{\Phi_1, \ldots, \Phi_n\}$)
input : Each Φ_i is propositional or begins with $\bar{Q}Y$.
output: a winning move for QX, if it exists, $\bot$ otherwise

1 **if** *all* Φ_i *propositional* **then**
2 $\quad \alpha \leftarrow (Q = \exists)\ ?\ \bigwedge_{i \in 1..n} \Phi_i\ :\ \bigwedge_{i \in 1..n} \neg \Phi_i$
3 $\quad$ **return** SAT(α)
4 $\mathcal{E}_i \leftarrow \emptyset$, $i \in 1..n$
5 $\alpha \leftarrow QX.\emptyset$
6 **while** true **do**
7 $\quad \tau' \leftarrow$ QFUN(α)
8 $\quad$ **if** $\tau' = \bot$ **then return** $\bot$
9 $\quad \tau \leftarrow \{l \mid l \in \tau' \wedge \mathsf{var}(l) \in X\}$
10 $\quad$ **if** QFUN($\Phi_i[\tau]$) $= \bot$ *for all* i **then**
11 $\quad\quad$ **return** τ
12 $\quad$ **let** j be s.t. QFUN($\Phi_j[\tau]$) $= \mu \neq \bot$
13 $\quad \mathcal{E}_j \leftarrow \mathcal{E}_j \cup \{(\tau, \mu)\}$
14 $\quad$ **if** ShouldLearn() **then**
15 $\quad\quad S \leftarrow$ Learn($\mathcal{E}_j$)
16 $\quad\quad \alpha \leftarrow$ Refine(α, Φ_j, S)
17 $\quad\quad \mathcal{E}_j \leftarrow \emptyset$
18 $\quad$ **else**
19 $\quad\quad \alpha \leftarrow$ Refine(α, Φ_j, μ)

3. a counter-move, μ, is computed for τ (if none exists, τ is a winning move, and the algorithm returns it); and a game j is chosen such that μ is a winning move for $\Phi_j[\tau]$,
4. the pair (τ, μ) is appended to $\mathcal{E}_j$, the set of samples of game j,
5. the decision is taken whether learning should happen at this step or not,
   - if we decide to learn at this iteration
     - learning occurs, producing a new strategy from $\mathcal{E}_j$
     - the abstraction is refined with the learned strategy,
     - $\mathcal{E}_j$ is reset to $\emptyset$.
   - if we decide not to learn at this iteration, the abstraction is refined with μ.

At each learning moment, the previous moves and counter-moves are the training set of the machine learning algorithm. The original implementation of QFUN [5] used the ID3 algorithm [9], working on decision trees. The purpose of our work was to clarify if QFUN could be improved by choosing another learning algorithm. Another goal was to verify if there are any QBF families for which QFUN with learning is particularly useful. In Section 2 we describe the alternative learning algorithms for QFUN we implemented and analyze the results obtained. In Section 3 we extend that analysis to complete QBF families in order to identify the families for which learning improves the solver's efficiency. In Section 4 we summarize the conclusions reached and review the open questions that can be addressed in future work.

2 Learning Algorithms

QFUN receives $\Psi = Qx_1, \ldots, x_n \overline{Q} y_1, \ldots, y_l \Phi$, and starts solving it, according to algorithm 1, producing a list of moves to $X = \{x_1, \ldots, x_n\}$ and respective counter-moves.

After a number of attempts, m, have been tried for a winning move to X, a set $\mathcal{E}$ of m samples has been stored by the solver. The function Learn, in step 15 of Algorithm 1 then takes $\mathcal{E}$ as its input and computes a strategy for each of the $y \in Y$.

Since the strategies are computed one variable at a time, the only input needed are the assignments to X and the values of the target variable y in the respective counter-moves. 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 a positive example if $y_i = 1$ and a negative example if $y_i = 0$, denoting the sets of positive and negative examples by E^+ and E^-, respectively.

After learning, the resulting strategies are refined into the abstraction, the set of samples is emptied, and the solver resumes its resolution of the formula.

2.1 Decision trees and ID3

In the original implementation of the solver, the function Learn uses the ID3 algorithm [9], which pseudo-code we present as Algorithm 2.

Algorithm 2: Pseudo-code for ID3

Function ID3($E, domain$)
input : $E = \{(\tau_1, y_1), (\tau_2, y_2), \ldots, (\tau_m, y_m)\}$
$domain$ is a subset of X
output: DT, a strategy for y

1 $DT \leftarrow$ single-node tree Root
2 **if** $E = E^+$ **then return** DT with label 1
3 **if** $E = E^-$ **then return** DT with label 0
4 **if** $domain = \emptyset$ **then return** DT labeled with the most common value of y in E
5 $sv \leftarrow \arg\max_{x \in domain} (\mathtt{IG}(E, x))$
6 **for** $b \in \{0, 1\}$ **do**
7 $\quad$ **let** $E_b = \{(\tau_i, y_i) \in E : \tau_i(sv) = b\}$
8 $\quad$ Add to DT a new tree branch for $sv = b$
9 $\quad$ add ID3(E_b, $domain \backslash \{sv\}$) below the new branch
10 **return** DT

The ID3 algorithm has a recursive nature. We begin by creating a decision tree having only the

root node. Then, we have three possibilities. The first is that all the examples we need to classify are in the same class, and in that case we label the root with the value of the y_i and we are done.

The second possibility is that we have run out of variables to label our nodes with, and in this case there is no perfect solution, and the best option is to label the root with the most common value of the y_i in E.

The third possibility is that we have positive and negative examples in E. In this case, we choose a split variable $sv \in domain$, separate the examples where $sv = 1$ from those where $sv = 0$ and then create a positive and a negative branch from the root, applying the ID3 algorithm recursively.

The algorithm chooses the best variable sv, by maximizing the information gain of E for $x \in domain$. Entropy measures the uncertainty observed. 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.

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

Information gain measures the reduction in uncertainty, i.e., the reduction of entropy, caused by the observation of a variable.

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 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 [5] and was confirmed by our own results.

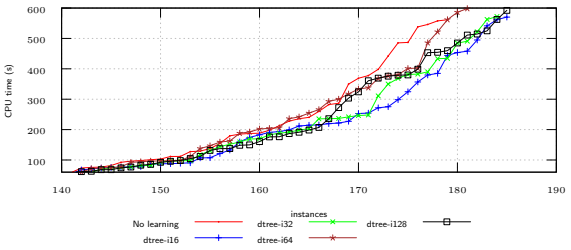

Figure 1: ID3 results

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 next Sections.

2.2 Decision Lists

Decision lists were introduced by Rivest [10], and we have chosen to use them, hoping to avoid the *replicated subtree problem* [8, 4]: in some situations identical subtrees have to be learned repeatedly at various places in a decision tree.

Rivest proved that decision lists are polynomially learnable, in the sense of PAC-learning introduced by Valiant [11], and provided an algorithm for constructing a decision list consistent with a given set of data [10].

We have altered `QFUN`, implementing 6 different algorithms using decision lists, that we present in Sections 2.2.1 to 2.2.6:

1. Rivest's original algorithm [10, p. 243-244];
2. Pagallo and Haussler's `Greedy3` algorithm, introduced in [8];
3. Pagallo and Haussler's `Grove` algorithm [8];

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.[2, 1]

2.2.1 Rivest

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

Algorithm 3: Pseudo-code for `Rivest`

Function `Rivest`(E, max)
input : $E = \{(\tau_1, y_1), (\tau_2, y_2), \ldots, (\tau_m, y_m)\}$
max is a parameter
output: DL, a strategy for y

1 **for** $t \in C^n_{max}$ **do**
2 $\quad$ T $\leftarrow \{(\tau_i, y_i) : \tau_i \models t\}$
3 $\quad$ **if** $T = \emptyset$ **then continue**
4 $\quad$ **if** $T \subset E^+$ **then**
5 $\quad\quad$ **return** $\{(t, 1)\}$ **.** `Rivest`$(E \setminus T, max)$
6 $\quad$ **if** $T \subset E^-$ **then**
7 $\quad\quad$ **return** $\{(t, 0)\}$ **.** `Rivest`$(E \setminus T, max)$
8 **let** (τ_i, y_i) belong to the smaller class of E
9 **return** (τ_i, y_i) **.** `Rivest`$(E \setminus \{(\tau_i, y_i)\}, max)$

The algorithm works recursively: we iterate over all terms of length at most max, until we find a term t such that all the examples that satisfy 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 8 and 9): when no t satisfies the requirement, we add a rule corresponding to an example from the less common class in E.

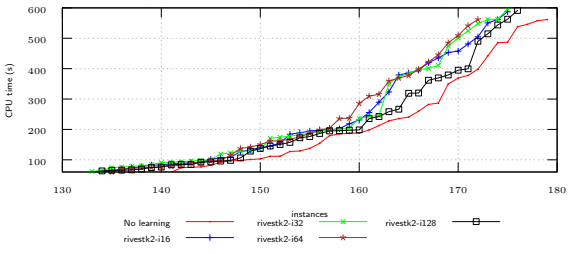

Figure 2: Rivest results, with $max = 2$

`QFUN` with `Rivest` performs considerably worse than with `ID3` and even worse than `QFUN` without any learning.

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.

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.

2.2.2 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 [8] 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.

Algorithm 4: Pseudo-code for the top-level 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 $\quad$ $rule \leftarrow$ `select-rule`(C)
5 $\quad$ **Append** $rule$ **to** DL
6 $\quad$ $C \leftarrow C \setminus \{e \in C \mid e$ is covered by $rule\}$
7 $c \leftarrow$ class common to all examples in C
8 **Append** (true, c) **to** DL
9 **return** DL

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 (step 4). Most `separate-and-conquer` algorithms use *hill-climbing*, building the term of the new rule by adding one literal at a time, at each moment 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.

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

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

1 $Pot \leftarrow \emptyset$
2 $term \leftarrow$ `true`
3 $domain \leftarrow X$
4 **while** $C^+ \neq \emptyset$ ***and*** $C^- \neq \emptyset$ **do**
5 $\quad l \leftarrow$ `select-literal`$(C, domain)$
6 $\quad term \leftarrow (term \wedge l)$
7 $\quad domain \leftarrow domain \setminus \{var(l)\}$
8 $\quad$ remove from C all examples with value 0 for l and add them to *Pot*
9 $c \leftarrow$ class of remaining examples in C
10 $C \leftarrow Pot$
11 **return** $(term, c)$

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 Figure 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.

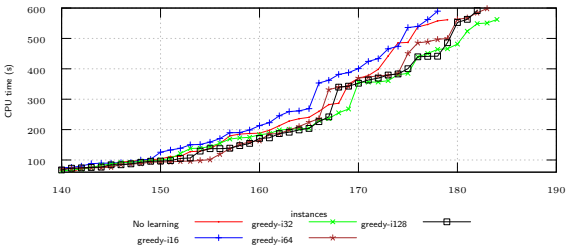

Figure 3: Greedy3 results

2.2.3 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.

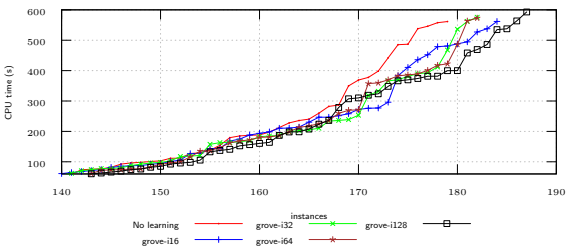

Figure 4: Grove results

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.

2.2.4 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.

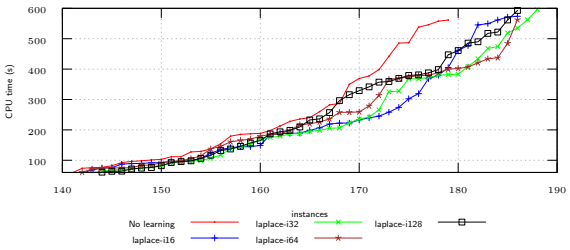

Figure 5: Laplace results

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.

2.2.5 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 either 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) Given a set of samples $E = \{(\tau_1, y_1), (\tau_2, y_2), \ldots, (\tau_m, y_m)\}$, and a literal l, we define the function *Simple* as:

$$\begin{aligned}\texttt{Simple}(E, l) = \ & \textbf{if}(E_l = E^+ \textbf{ or } E_l = E^-) \\ & \textbf{then return}(-1, p+n) \\ & \textbf{else return}(min(p, n)\ ,\ p+n),\end{aligned}$$

where p and n are the number of examples that satisfy l in E^+ and E^-, respectively.

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 [4, p. 35].

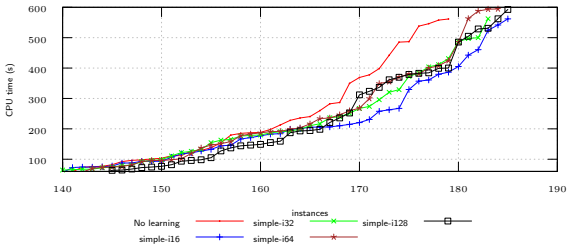

Figure 6: Simple results

`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.

2.2.6 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

let $s \leftarrow \{\text{true}\}$
let $ns \leftarrow \emptyset$
let $bt \leftarrow \text{true}$
while $s \neq \emptyset$ **do**
  foreach $t \in s$ **do**
    if *term is coherent* **then continue**
    let $dom \leftarrow \{x \in X \mid x \text{ not in } t\}$
    foreach $l \in \{x, \neg x \mid x \in dom\}$ **do**
      let $nt \leftarrow t \wedge l$
      if Laplace(nt) > Laplace(bt)
        and *nt is coherent* **then**
        $bt \leftarrow nt$
      $ns \leftarrow ns \cup \{nt\}$
      if *size of* $nt > maxs$ **then**
        remove the worst term from ns
  $s \leftarrow ns$
  $ns \leftarrow \emptyset$
$c \leftarrow$ class of examples that satisfy bt
return (bt, c)

We begin by initializing 3 variables: s, ns and bt. s will be used to hold 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 $maxs$. ns will be used to hold an auxiliary set, so that we can iteratively update s, and is initialized as the empty set. bt will be used to identify the best term found so far, and is initialized to true.

We then start a cycle, 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.

In Figure 7, we can see the results obtained with the CN2 algorithm, for a beam size of 2.

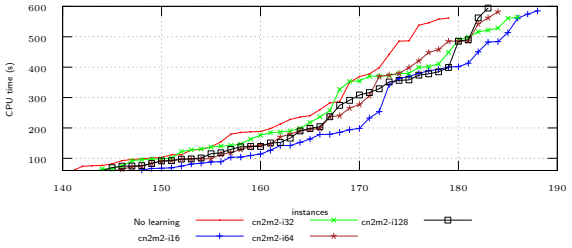

Figure 7: CN2 results, beam size 2

In Figure 8, we can see the results obtained with the CN2 algorithm, for a beam size of 3.

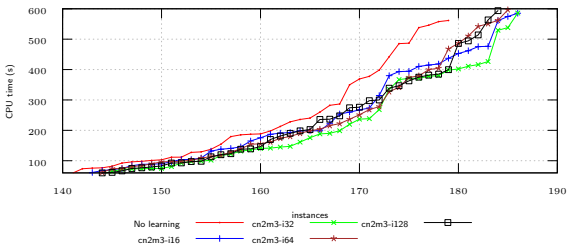

Figure 8: CN2 results, beam size 3

In Figure 9, we can see the results obtained with the CN2 algorithm, for a beam size of 4.

The results obtained with CN2, made us want to look into more detail at the differences in performance of the different algorithms, for different families of formulae. This will be the subject of Section 3.

In Figure 10, we can see the results obtained with all algorithms, combining the best version of each. We have also added the results obtained with the QuAbS solver, the winner of the QBFEval18 competition.[1]

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

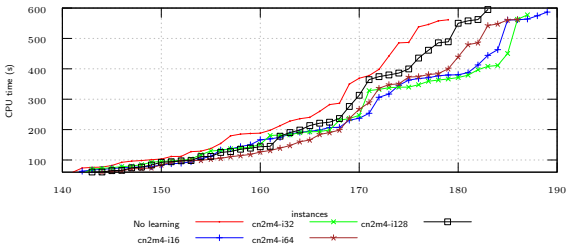

Figure 9: CN2 results, beam size 4

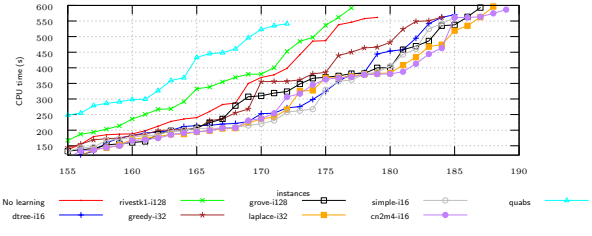

Figure 10: All algorithms (best version of each)

A detailed table with all the results obtained is publicly available.[2]

3 Specific families of formulae

In Section 2, we have tested QFUN on a set of QBFs obtained from instances used in international competitions. To analyze the behavior of the solver and its learning algorithms on specific families, we decided to use sets of QBFs that are available for download at QBFLIB, The Quantified Boolean Formulas Satisfiability Library[3], an open online collection of instances.

We have kept the same time and memory constraints that we have used in Section 2: 600 seconds and 2GB, and used the same machines to perform the tests, please refer to page 2 for the details.[4]

Family toy is a set of QBFs consisting of 270 instances of QBF encodings for a number of basic building blocks of circuits. Results are presented in Figure 11.

QFUN without learning was able to solve 205 QBFs, while QFUN with learning solved between 217 and 229. QuAbS solved exactly the same number of QBFs as QFUN without learning (not the same QBFs, though).

These results are interesting, because they show clearly a much better performance of QFUN with learning.

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

[3] http://www.qbflib.org/index.php

[4] The complete and detailed set of results can also be accessed at http://sat.inesc-id.pt/~rjs/qbf/familias_completas/.

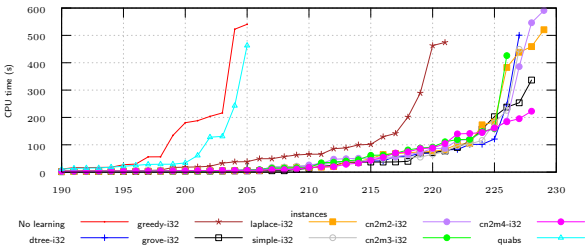

Figure 11: Results for the **toy** family of QBFs

Family genbuf is another family of QBFs: 126 instances of encodings for generalized buffer specification, and we can see the results obtained in Figure 12.

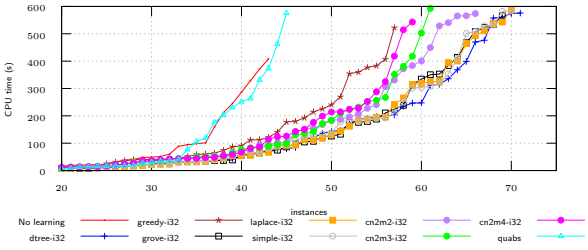

Figure 12: Results for the **genbuf** family of QBFs

QFUN without learning solved 41 instances, and QuAbS did a little better, with 43. QFUN with learning solved between 58 and 71 instances. The effect of learning on the solver's performance is even more impressive in this family than it was with **toy**. Hill-climbing works better than beam search for this family of QBFs. We can see a pattern in the results of CN2, whereby the performance degrades as the size of the beam increases.

Family driver is a smaller family of 48 QBFs encoding specifications of a driver for a hard disk controller. We can see the results obtained in Figure 13.

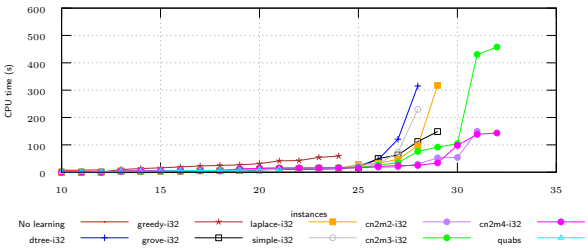

Figure 13: Results for the **driver** family of QBFs

QFUN without learning could only solve 12 QBFs, QuAbS did better and solved 21. QFUN with Greedy3 solved 24. QFUN with the other algorithms except for CN2 solved between 27 and 29. QFUN with

CN2 solved between 31 and 33. So, in a complete contrast with what we saw in **genbuf**, **driver** is a family for which not only learning improves the solver, but specifically learning with CN2 is better compared to the other algorithms, so beam search pays off.

Family mult-matrix is a larger set of 522 QBFs encoding the specifications for circuits that perform matrix multiplication and the results we have obtained for it show that learning degrades the performance of the solver. Graphics of the results are shown in figure 14.

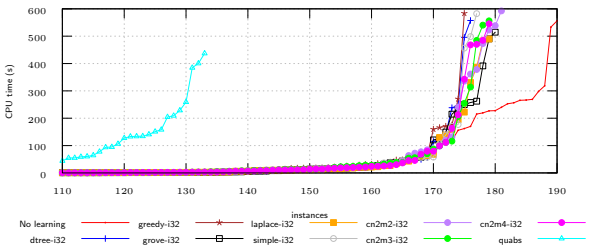

Figure 14: Results for the **mult-matrix** family of QBFs

Family cycle-sched is a family of 88 instances, encoding a controller. QuAbS only solved 2 while QFUN without learning solved 18. QFUN with learning solved between 19 and 23 instances. In this family, similarly to what we saw with **genbuf**, CN2 performs worse than the hill-climbing algorithms.

Conclusion of Family Analysis We have identified four QBF families where learning improves the performance of QFUN noticeably. We have also clearly identified another family where learning degrades the solver's performance and should therefore be avoided.

A more in-depth analysis, in order to understand why the behaviors observed emerge, which features of the formulae create these differences in learning, is a very important open question.

4 Conclusions

We have amply confirmed the main result of [5], 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 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 (Section 2.2.6) showed that, contrary to our concerns, using beam search to select literals is quite feasible. This in turn leads us to conclude that more complex learning algorithms might be suitable candidates to improve QFUN.

Concerning the analysis of the QBF families, the experimental evidence confirms that for some families QFUN with learning is particularly useful, while for others it is preferable to use QFUN without learning. Much is certainly to be done in terms of understanding which formulae benefit from this kind of learning during QBF solving, but we have taken the first steps. The analysis of families seems more promising than analyzing the sets from QBFEval.

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

References

[1] Peter Clark and Robin Boswell. Rule induction with CN2: Some recent improvements. In *Proceedings of the European Working Session on Learning (EWSL)*, pages 151–163. Springer, 1991.

[2] Peter Clark and Tim Niblett. The CN2 induction algorithm. *Machine Learning*, 3(4):261–283, 1989.

[3] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided Abstraction Refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

[4] Johannes Fürnkranz. Separate-and-Conquer Rule Learning. *Artificial Intelligence Review*, 13(1):3–54, 1999.

[5] Mikoláš Janota. Towards Generalization in QBF Solving via Machine Learning. In *AAAI Conference on Artificial Intelligence*, pages 1–14, 2018.

[6] Mikoláš Janota, William Klieber, João Marques-Silva, and Edmund Clarke. Solving QBF with counterexample guided refinement. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 114–128. Springer, 2012.

[7] Mikoláš Janota and João Marques-Silva. Abstraction-based algorithm for 2QBF. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 230–244. Springer, 2011.

[8] Giulia Pagallo and David Haussler. Boolean Feature Discovery in Empirical Learning. *Machine Learning*, 5(1):71–99, 1990.

[9] J. Ross Quinlan. Induction of Decision Trees. *Machine learning*, 1(1):81–106, 1986.

[10] Ronald L. Rivest. Learning Decision Lists. *Machine Learning*, 2(3):229–246, 1987.

[11] Leslie G Valiant. A Theory of the Learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.