QFUN: Towards Machine Learning in QBF

Mikoláš Janota

IST/INESC-ID, Portgual
mikolas.janota@gmail.com

Abstract. This paper reports on the QBF solver QFUN that has won the non-CNF track in the recent QBF evaluation. The solver is motivated by the fact that it is easy to construct Quantified Boolean Formulas (QBFs) with short winning strategies (Skolem/Herbrand functions) but are hard to solve by nowadays solvers. This paper argues that a solver benefits from generalizing a set of individual wins into a strategy. This idea is realized on top of the competitive RAReQS algorithm by utilizing machine learning. The results of the implemented prototype are highly encouraging.

1 Introduction

Against all odds posed by computational complexity, logic-based problem solving had a remarkable success at research but also industrial level. One of the impressive success stories is the Boolean satisfiability problem (SAT). Quantified Boolean formulas (QBF) go one step further and extend SAT with quantification. This enables targeting a larger class of problems [6,39,41,14]. However, success of QBF solvers comparable to SAT still seems quite far. Nevertheless, we have recently seen a significant progress in the area almost every year, e.g. [52,8,5,44,15,16,21,37,19,32,43,36]. This paper aims to make a case for the use of machine learning *during* QBF solving.

It has been observed that search is often insufficient. A well-known example is the formula $\forall X \exists Y. \bigwedge x_i \leftrightarrow y_i$ with $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_n\}$ [28]. Traditional search will easily find an assignment (valuation) to X and Y satisfying the matrix (the propositional part). However, to prove that there is an assignment for Y given *any* assignment to X is difficult. Traditional search, even with various extensions, will try exponentially many assignments. A human can easily see why the formula is true. Indeed, given an arbitrary assignment to X, setting each y_i to x_i gives a witness for the validity of the formula.

It is useful to see QBFs as two-player games, where the existential player tries to make the formula true and the universal false. A winning strategy for the existential player shows that it is true. The formula above is a good example of a *small winning strategy*—the strategy for y_i is the function $s_{y_i}(x_1, \ldots, x_n) \triangleq x_i$. The million dollar question here is, where do we get the strategies? This paper builds on the following idea: *Observe a set of assignments and learn from them strategies using machine learning. In another words, rather than looking at individual assignments, collect a set of them and generalize them into a strategy.*

Learning a strategy is not enough—it must also be incorporated into a solving algorithm. A straightforward approach would be to test for the learned strategy whether it is a winning one (that is possible with a SAT call [26]). However, this would put a lot of strain on the learning since we would have to be quite lucky to learn the right strategy and eventually we would have to deal with large training sets.

The algorithm presented in this paper takes inspiration in the existing algorithm RAReQS, which gradually expands the given formula by plugging in the encountered assignments [20]. Instead of plugging in assignments, we will be plugging in the learned strategies. This forms the second main idea of the paper: *Expand the formula using strategies learned from collected samples and then start collecting a new set of samples.*

2 Preliminaries

A *literal* is a Boolean variable or its negation. The literal complementary to a literal l is denoted as $\bar{l}$, i.e. $\bar{x} = \neg x$, $\overline{\neg x} = x$. For a literal $l = x$ or $l = \neg x$, we write $\mathsf{var}(l)$ for x. Analogously, $\mathsf{vars}(\phi)$ is the set of all variables in formula ϕ. An assignment is a mapping from variables to Boolean constants $0, 1$. For a formula ϕ and an assignment τ, we write $\phi[\tau]$ for the substitution of the variables in the domain of τ with their respective constants.

2.1 Quantified Boolean Formulas

Quantified Boolean Formulas (QBFs) [25] extend propositional logic by enabling quantification over Boolean variables. Any propositional formula ϕ is also a QBF with all variables *free*. If Φ is a QBF with a free variable x, the formulas $\exists x.\,\Phi$ and $\forall x.\,\Phi$ are QBFs with x *bound*, i.e. not free. Note that we disallow expressions such as $\exists x.\exists x.\,x$, i.e., each variable is bound at most once. Whenever possible, we write $\exists x_1 \ldots x_k$ instead of $\exists x_1 \ldots \exists x_k$; analogously for $\forall$. For a QBF $\Phi = \forall x.\,\Psi$ we say that x is *universal* in Φ and is existential in $\exists x.\,\Psi$. Analogously, a literal l is universal (resp. existential) if $\mathsf{var}(l)$ is universal (resp. existential).

Assignments also can be applied to QBF with $(Qx.\,\Phi)[\tau]$ defined as $\Phi[\tau]$ if x is in the domain of τ with $Q \in \{\forall, \exists\}$.

A QBF correspond to a propositional formula: $\forall x.\,\Psi$ corresponds to $\Psi[x{\to}0] \wedge \Psi[x{\to}1]$ and $\exists x.\,\Psi$ to $\Psi[x{\to}0] \vee \Psi[x{\to}1]$. Since $\forall x \forall y.\,\Phi$ and $\forall y \forall x.\,\Phi$ are semantically equivalent, we allow QX for a set of variables X, $Q \in \{\forall, \exists\}$. A QBF with no free variables is *false* (resp. *true*), iff it is semantically equivalent to the constant 0 (resp. 1).

A QBF is *closed* if it does not contain any free variables. A QBF is in *prenex form* if it is of the form $\mathcal{Q}_1 X_1 \ldots \mathcal{Q}_k X_k.\,\phi$, where $\mathcal{Q}_i \in \{\exists, \forall\}$, $\mathcal{Q}_i \neq \mathcal{Q}_{i+1}$, ϕ propositional, and X_i pairwise disjoint sets of variables. The propositional part ϕ is called the *matrix* and the rest *prefix*. For a variable $x \in X_i$ we say that x is at *level* i and write $\mathsf{lv}(x) = i$; we write $\mathsf{lv}(l)$ for $\mathsf{lv}(\mathsf{var}(l))$. Unless specified otherwise, QBFs are assumed to be closed and in prenex form.

2.2 Games and Strategies

For most of the paper, QBFs are seen as two-player games. The existential player tries to make the matrix true and conversely the universal player to make it false. A player assigns value only to variables that belong to the player and may assign a variable only once all variables that precede it in the prefix are assigned. In other words, the two players assign values following the order of the prefix. The game semantic perspective on QBF has the advantage that mostly we do not need to distinguish between the player $\exists$ and $\forall$. Instead, we will be talking about a player and its opponent.

Notation. We write Q for either of the players, and, $\bar{Q}$ for its opponent.

Given a QBF $\mathcal{Q}_1 X_1, \ldots, \mathcal{Q}_n X_n. \phi$ the *domain* $\mathsf{dom}(x)$ of a variable $x \in X_k$ are all the variables in the preceding blocks, i.e. $\mathsf{dom}(x) = \bigcup_{i \in 1..k-1} X_i$.

A *play* is a sequence of assignments $\tau_1, \ldots, \tau_n$ where τ_i is an assignment to X_i.

Definition 1. *For a QBF $\mathcal{Q}_1 X_1, \ldots, \mathcal{Q}_n X_n. \phi$ a* strategy for a variable $x \in X_k$ *is a boolean function s_x whose arguments are the variable's domain, i.e.* $\mathsf{dom}(x)$*.*

A strategy for a player $\mathcal{Q}$ *is a set of strategies s_x for each of the variables $x \in QX_i$. Whenever clear from the context, we simply say strategy for either of the concepts.*

Notation. For the sake of succinctness, a strategy for a variable x is conflated with a Boolean formula whose truth value represents the value of the strategy. In another words, a strategy represents both some function $s_x : 2^{\mathsf{dom}(x)} \mapsto \{0, 1\}$ and some formula ψ_x with $\mathsf{vars}(\psi_x) \subseteq \mathsf{dom}(x)$. This convention lets us also treat a set of strategies S for some variables X as a substitution. Hence, $\xi[S]$ represents the formula that results from simultaneously replacing in ξ each variable $x \in X$ with its strategy ψ_x.

Definition 2 (winning strategy). *Let Ψ be a closed QBF $(QX \ldots \phi)$ with ϕ propositional. A strategy S for $\exists$ is* winning in Ψ *if $\phi[S]$ is a tautology. A strategy S for $\forall$ is* winning in Ψ *if $\phi[S]$ is unsatisfiable.*

In particular, for a formula $\exists X. \phi$ a winning strategy for $\exists$ corresponds to a satisfying assignment of ϕ.

Observation 1 *A closed QBF Φ is true iff there exists a winning strategy for $\exists$; it is false iff there exists a winning strategy for $\forall$.*

Definition 3 (winning/counter move). *For a closed QBF $QX. \Phi$ an assignment τ to X is a* winning move *if there exists a winning strategy for Q in $\Phi[\tau]$.*

For a closed QBF $QX\bar{Q}Y. \Phi$ and an assignment τ to X, an assignment μ to Y is a counter-move *to τ if μ is a winning move for $\bar{Q}Y. \Phi[\tau]$.*

Observation 2 *There exists some winning move for QX in a formula $QX. \Phi$, if and only if there exists a winning strategy for Q in the formula.*

Observation 3 *For a formula $QX\bar{Q}Y. \Phi$, an assignment to X is a winning move if and only if there does not exist a counter-move to it.*

3 Algorithm qfun

As to make it a more pleasant read, this section comes in three installations, each bringing in more detail. The first part quickly overviews the existing algorithm (R)AReQS and sketches the main ideas of the proposed approach, which we will simply call the algorithm qfun. The second part presents qfun for the two-level case, i.e., formulas with one quantifier alternation. Finally, the third part details out the algorithm for the general case, i.e., formulas with arbitrary number of quantifier alternations.

3.1 Exposition

Let us quickly review the existing algorithm RAReQS [20]. For a formula $QX\bar{Q}Y.\Phi$ RAReQS aims to decide whether there exists a winning move for Q. To that end, the algorithm keeps on constructing a sequence of pairs $(\tau_1, \mu_1), \ldots, (\tau_k, \mu_k)$. Each τ_i is an assignment to X and μ_i is a counter-move to τ_i (see Def. 3). In each iteration, RAReQS constructs a partial expansion (called *abstraction*) of the original QBF such that no existing μ_i is a counter-move in the original formula to any winning move of the abstraction. In another words, if Q draws the next move so that it wins the abstraction, he is guaranteed not to be beaten by any of the existing counter-moves.

If there is no winning move for the abstraction, there isn't one for the original formula either and therefore there is no winning strategy for Q (we are done). If there is some winning move τ_{k+1} for the abstraction, we check whether the opponent still comes up with a counter-move μ_{k+1}. If he does not, τ_{k+1} is a winning move for Q and we are again done (see Observation 2). If a counter-move is found, the pair (τ_{k+1}, μ_{k+1}) is added to the sequence and the process repeats.

This setup inspires the use of machine learning. Since each μ_i is a counter-move to τ_i, the constructed sequence of pairs (τ_i, μ_i) can be conceived as a *training set* for the strategies for the variables Y (belonging to the player $\bar{Q}$). More specifically, for each variable $y \in Y$, the pair (τ_i,μ_i) represents a training sample for the function s_y prescribing that $s_y(\tau_i) = \mu_i(y)$. Observe that there might be other good strategies for the opponent $\bar{Q}$. However, the pairs (τ_i, μ_i) have already *proven* to be good for $\bar{Q}$ and therefore we will stick to them.

It is tempting to learn a strategy for $\bar{Q}Y$ from such samples and then verify that it is a winning one. If it is a winning one, we would be done. If it is not a winning one, we could just learn a better one once we have more samples. However, this approach is unlikely to work. The problem with this approach is twofold. Firstly, it is overly optimistic to hope to hit the right strategy given a set of samples whose number is likely to be much smaller than the full truth table of the strategy. Secondly, it is putting too much strain on machine learning because the set of samples keeps on growing. Instead, this paper proposes the following schema.

1. Collect some suitable set of samples $\mathcal{E}$.

Algorithm 1: Playing 1-move multi-game

Function `Wins1` ($QX.\ \{\phi_1, \ldots, \phi_n\}$)
input : All ϕ_i propositional.
output : a winning move for all $QX.\,\phi_i$, if there is one; $\bot$ otherwise.
$\alpha \leftarrow (Q = \exists)\,?\ \bigwedge_{i\in 1..n} \phi_i\ :\ \bigwedge_{i\in 1..n} \neg\phi_i$
return `SAT`(α)

2. Learn strategies S for the opponent variables.
3. Strengthen the current abstraction using the strategies S.
4. Reset the set of samples $\mathcal{E}$
5. Repeat.

3.2 qfun^2: 2-level QBF

Let us look at the two-level case, i.e., a QBF of the form $QX\bar{Q}Y.\,\phi$ with ϕ propositional. This form is particularly amenable to analysis since both the abstraction and candidate-checking is solvable by a SAT solver. Also, 2-level QBF has a number of interesting applications (cf. [6,39,41]).

A slight generalization of a game called a *multi-game* [20] is useful in the following presentation. A multi-game is a set of sub-games where the top-level player must find a move that is winning for all these sub-games at once. Note that a multi-game can be converted to a standard QBF by prenexing. However, it is useful to maintain this form (see [20, Sec. 4.1]).

Definition 4 (multi-game). *A multi-game is written as $QX.\{\Phi_1, \ldots, \Phi_k\}$. An assignment τ to X is a winning move for it iff it is a winning move for all $QX.\,\Phi_i$. Each Φ_i is called a* sub-game *and is either propositional or begins with $\bar{Q}$.*

When all sub-games are propositional, the multi-game is solvable by a single SAT call. For such we introduce a function `Wins1` (Algorithm 1). The function calculates a winning move for the multi-game or returns $\bot$ if it does not exist (the function `SAT` has the same behavior). Observe that if the set of sub-games is *empty*, the formula α in `Wins1` is the empty conjunction, which is equivalent to true, i.e., the SAT call then returns an arbitrary assignment.

Just as the existing algorithm AReQS, qfun^2 (Algorithm 2) maintains an *abstraction* α. The abstraction corresponds to a partial expansion of the inner quantifier. This means that for a formula $QX\bar{Q}Y.\phi$, the abstraction has the form $QX.\ \{\Phi[S] \mid S \in \omega\}$, where ω is some set of strategies. Observe that the abstraction is trivially equivalent to the original formula if ω contains all possible constant functions.

For instance, $\forall u \exists e.\,\phi$ is equivalent to $\forall u.\ \phi[e{\to}0] \vee \phi[e{\to}1]$, which is equivalent to the multi-game $\forall u.\,\{\phi[e{\to}0], \phi[e{\to}1]\}$.

Example 1. Consider $\forall uw\exists xy.\,\phi$ with $\phi = (u \Rightarrow (\neg w \Leftrightarrow x \wedge w \Leftrightarrow y)) \wedge (\neg u \Rightarrow (w \Leftrightarrow x \wedge \neg w \Leftrightarrow y))$. The following abstractions of this formula are both losing

Algorithm 2: qfun^2: 2-level QBF Refinement with Learning

Function qfun^2($QX\bar{Q}Y.\,\phi$)
input : ϕ is propositional.
output : a winning move for QX if there exists one, $\perp$ otherwise

$\mathcal{E} \leftarrow \emptyset$ `// start with no samples`
$\alpha \leftarrow \emptyset$ `// empty abstraction`
while true **do**
  $\tau \leftarrow$ `Wins1`$(QX.\,\alpha)$ `// candidate`
  if $\tau = \perp$ **then return** $\perp$ `// loss`
  $\mu \leftarrow$ `Wins1`$(\bar{Q}Y.\,\{\phi[\tau]\})$ `// countermove`
  if $\mu = \perp$ **then return** τ `// win`
  $\mathcal{E} \leftarrow \mathcal{E} \cup \{(\tau, \mu)\}$ `// record sample`
  if `ShouldLearn`() **then**
    $S \leftarrow$ `Learn`$(\mathcal{E})$ `// learn`
    $\alpha \leftarrow \alpha \cup \{\phi[S]\}$
    $\mathcal{E} \leftarrow \emptyset$ `// reset samples`
  else
    $\alpha \leftarrow \alpha \cup \{\phi[\mu]\}$ `// refine`

for $\forall$. With two sub-games: $\forall uw.\,\{\phi[x{\to}\neg w, y{\to}w], \phi[x{\to}w, y{\to}\neg w]\}$; with single sub-game: $\forall uw.\,\{\phi[x{\to}(u\,?\,\neg w\,:\,w), y{\to}(u\,?\,w\,:\,\neg w)]\}$.

The abstraction α is *refined* with every play losing for Q, which effectively means adding a subgame to the current abstraction. Additionally, qfun^2 maintains a set of samples $\mathcal{E}$. The samples are pairs (τ_i, μ_i) such that τ_i, μ_i is a losing play for Q, i.e., a winning play for $\bar{Q}$. So for instance, if $Q = \forall$ then $\bar{Q} = \exists$ and $\tau_i \cup \mu_i \models \phi$.

Both the abstraction α and samples $\mathcal{E}$ are initialized as empty. In each iteration, qfun^2 calls `Wins1` to calculate a *candidate* for a winning move τ. Subsequently, another call to `Wins1` is issued to calculate a counter-move μ. If either candidate or counter-move does not exist, one of the player has lost without recovery.

Machine learning is invoked only ever so often. To decide when, the pseudocode queries the function `ShouldLearn`. Whenever `ShouldLearn` is true, new strategies are learned for Y-variables based on the samples $\mathcal{E}$. These strategies are plugged into the formula ϕ and recorded in the abstraction. Then, and that is crucial, the set of samples is reset back to the empty set.

In terms of soundness, the set of samples $\mathcal{E}$ need not be reset after each learning. However, it is crucial in terms of performance. If the set is always augmented, learning will become overly time consuming. Recall that a strategy needs to be learned for each opponents' variable and further, the number of iterations can go to millions.

So what does the abstraction represent and what is the role of the learned strategies? The original AReQS adds a sub-game $\phi[\mu_i]$ for each existing counter-move μ_i. Intuitively, this means that the player Q never plays before making sure that he can successfully defend himself against all the existing counter-moves. Once strategies are also included, the player Q also defends himself against all the strategies devised so far. Strategy-based refinement is a generalization of the traditional one—the traditional refinement corresponds to a set of strategies comprising constant functions.

What do we require from the strategy learning? The good news is that in fact very little. A strategy must be learned in the form of a formula so that it can be plugged into the original formula. This means that the algorithm does not easily allow for neural networks, for instance. The current implementation uses decision-trees (see Section 4). For the sake of soundness, the learned strategy formula *must follow* the definition of a strategy (Definition 1). In practice this means that a strategy formula ξ_y for a variable y must only contain variables from $\mathsf{dom}(y)$.

If the function `ShouldLearn` triggers traditional refinement only finitely many times, the learning method also needs to guarantee termination of the whole algorithm. A natural minimal requirement for this is that the learned strategies will correspond to at least one sample $(\tau_i, \mu_i) \in \mathcal{E}$, i.e. $s_y(\tau_i) = \mu_i(y)$, for each $y \in Y$. This requirement guarantees that τ_i will not appear as a candidate for a winning move in the upcoming iterations. Nevertheless, if `ShouldLearn` alternates between traditional and learning-based refinement, termination is already guaranteed by the traditional refinement and we do not need to worry about what is learned as long as it is sound.

3.3 qfun: General Case

The general case qfun generalizes the two-level case qfun^2 using recursion (just as RAReQS generalizes AReQS). The basic ideas remain, even though we are faced with a couple of technical complications. The pseudocode is presented as Algorithm 3. Since the abstraction is a multi-game, the recursive call also needs to handle a multi-game. For this purpose, we maintain a set of sequences of samples—each sequence for each given sub-game. Candidates for a winning-move are drawn from the abstraction α by a recursive call. The small technical difficulty here is that the abstraction may return a winning move containing some extra fresh variables coming from refinement. Hence, these need to be filtered out (ln. 8).

If, the candidate move τ is a winning move, it is returned. If, however, there is some counter-move μ_i obtained by playing the sub-game Φ_i, it is used for refinement. This means inserting the pair (τ, μ_i) into the sample sequence pertaining to this sub-game, i.e. sequence $\mathcal{E}_i$. And subsequently, performing refinement. In order to ensure that quantifiers alternate, refinement introduces fresh variables for formulas with more than 2 levels. The refinement function is defined as follows.

$\mathtt{Refine}\big(QX.\{\Psi_1, \ldots, \Psi_n\},\ \bar{Q}YQX_1.\Psi,\ S\big) :=$
$QXX_1'.\{\Psi_1, \ldots, \Psi_n, \Psi'[S]\}$

Algorithm 3: QBF Refinement with Learning

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 there exists one, $\perp$ otherwise

if *all* Φ_i *propositional* **then**
  return `Wins1`($QX.\ \{\Phi_1, \ldots, \Phi_n\}$)
$\mathcal{E}_i \leftarrow \emptyset$, $i \in 1..n$ `// samples`
$\alpha \leftarrow QX.\emptyset$ `// empty abstraction`
while `true` **do**
  $\tau' \leftarrow$ qfun(α) `// candidate`
  if $\tau' = \perp$ **then return** $\perp$ `// loss`
  $\tau \leftarrow \{l \mid l \in \tau' \wedge \mathsf{var}(l) \in X\}$ `// filter`
  if *all qfun*($\Phi_i[\tau]) = \perp$ **then return** τ `// win`
  let l be s.t. qfun($\Phi_l[\tau]) = \mu$ for some $l \in \{1..n\}$, $\mu \neq \perp$
  $\mathcal{E}_l \leftarrow \mathcal{E}_l \cup \{(\tau, \mu)\}$ `// record sample`
  if `ShouldLearn`() **then**
    $S \leftarrow$ `Learn`($\mathcal{E}_l$) `// learn`
    $\alpha \leftarrow$ `Refine`(α, Φ_l, S)
    $\mathcal{E}_l \leftarrow \emptyset$ `// reset samples`
  else
    $\alpha \leftarrow$ `Refine`(α, Φ_l, μ) `// refine`

$\texttt{Refine}\big(QX.\{\Psi_1, \ldots, \Psi_n\},\ \bar{Q}Y.\,\psi,\ S\big)\ :=$
$QX.\{\Psi_1, \ldots, \Psi_n, \psi[S]\}$
where X_1' are fresh duplicates of the variables X_1 and Ψ' is Ψ with X_1 replaced by X_1' and where ψ is a propositional formula.

4 Implementation

4.1 Formula representation

The algorithm requires nontrivial formula manipulation to achieve refinement. Performing these operations directly on a CNF representation is difficult and further, CNF representation as input has well-known pitfalls [1]. Hence, the implementation represents formulas as *And-Inverter graphs* (AIG) [18], which are simplified by trivial non-invasive simplifications [11]. All the logical operations (e.g. substitution/conjunction) are performed on AIGs. Only when the time comes to call a SAT solver, the AIG is translated into CNF. This is done in straightforward fashion. Each sub-AIG is mapped to an encoding Boolean variable in the SAT solver. Since the AIGs are hash-consed, each sub-AIG also corresponds to just one variable. All the and-gates are binary. The input to the solver is the circuit-like format for QBF called *QCIR* [22].

4.2 Learning

Recall that learning is invoked with the sequence of pairs of assignments $\mathcal{E} = (\tau_1, \mu_1), \ldots, (\tau_k, \mu_k)$, where each τ_i is an assignment to some block of variables X in the prefix and μ_i is an assignment to variables Y, which is the adjacent block in the prefix, belonging to the opposing player.

The objective is to learn a strategy (a function) for each of the variables in Y. A Boolean function can be seen as a *classifier* with two classes: the input assignments where the strategy should return 1 (true) and the input assignments where the strategy should return 0 (false). The implementation uses the popular classifier *Decision trees* [40]. These are constructed by the standard *ID3* algorithm [35].

For each variable in $y \in Y$, construct the training set $\mathcal{E}_y$ from $\mathcal{E}$ by ignoring all the other Y variables. Subsequently invoke ID3 on $\mathcal{E}_y$ thus obtaining a decision tree conforming to the sample assignments. Once a decision-tree is constructed, the Boolean formula is constructed as follows.

1. Construct the sets of conjunctions of literals $\mathcal{I}_p$ and $\mathcal{I}_n$ corresponding to the positive and negative branches of the tree, respectively. Hence, if $t \in \mathcal{I}_p$ is true, the tree gives 1.
2. Repeatedly apply subsumption and self-subsumption on each set $\mathcal{I}_p$ and $\mathcal{I}_n$, until a fixed point is reached.
3. If $|\mathcal{I}_p| < |\mathcal{I}_n|$ return $\bigvee \mathcal{I}_p$, otherwise return $\neg \bigvee \mathcal{I}_n$.

Step 2 would not necessarily be needed but since we are substituting the constructed functions into the input formula, it is desirable to maintain them small. Analogously, either set could be chosen in step 3 but a smaller is preferable.

When to learn? It is a bad idea to learn too frequently since this would produce poor sample-sets to learn from. However, learning too *infrequently* has two main pitfalls:

1. Learning on large sample-sets will be too costly (recall that a learning algorithm is run for each opponent variable upon refinement).
2. There is a risk of very complicated and therefore large functions to be learned from complicated samples

A straightforward approach was taken to implement the function `ShouldLearn`: learning is triggered every K iterations of the loop, where K is a parameter of the solver. The number of iterations is considered local for each recursive call of qfun. The experimental evaluation examines the solver's behavior for several values of K (see Section 5).

4.3 Strategy accumulation

Upon each refinement the set of samples is reset. Also, whatever is learned is forgotten in the next rounds—learning starts from scratch on a new set of

Solver	Quabs	GQ	RAReQS	L-16	L-64	L-128	L-64-f
Solved (320)	103	75	105	110	**111**	**111**	104
Wins	63	11	**67**	55	63	62	60

Table 1. Result summary. A win is counted also for solvers that are not worse than the best time by 1s.

samples. This might be disadvantageous. The current implementation uses a simple but important improvement. The algorithm records for each variable y the last learned strategy. This strategy is then evaluated on the next batch of samples when learning is invoked again. If it still fits the data, it is kept. Otherwise it is discarded and a new strategy is learned.

Example 2. Consider the formula from the introduction of the paper: $\forall x_1, \ldots, x_n \exists y_1, \ldots, y_n. \bigwedge x_i \leftrightarrow y_i$, *and, the following sequence of samples.*

x_1	x_2	$\ldots$	x_n	y_1	y_2	$\ldots$	y_n
0	0	$\ldots$	0	0	0	$\ldots$	0
1	0	$\ldots$	0	1	0	$\ldots$	0
0	0	$\ldots$	1	0	0	$\ldots$	1
0	1	$\ldots$	1	0	1	$\ldots$	1

If $K = 2$, the first application of learning gives $y_1 \triangleq x_1$ and the rest of the strategies are constantly 0. In the second refinement, learning gives $y_2 \triangleq x_2$ and the rest constants. If, however, we keep the information from the previous learning, we get both $y_1 \triangleq x_1$, $y_2 \triangleq x_2$. Hence, accumulating the individual strategies will eventually yield the right strategy.

4.4 Incrementality

The recursive structure of the algorithm is very elegant but might be too forgetful. If one is to solve $\Phi[\mu_i]$, it could be useful to maintain the abstraction from that solving in order to solve $\Phi[\mu_{i+1}]$. The issue is that then the solvers tend to occupy too much space. Currently, the solver maintains only abstractions that are purely propositional.

5 Experimental Evaluation

The RAReQS algorithm has proven to be highly competitive as it have placed first in several tracks of the recent QBF competitions.[1] So the key question is whether RAReQS benefits, or may benefit, from the proposed learning.

The success of the machine learning techniques can be assessed at various levels. The lowest bar is whether the technique is at all *computationally feasible*. Indeed, it might be that the learning is impractically time-consuming. Second step is whether the *number of iterations decreases when learning is applied*. The

[1] http://www.qbflib.org/

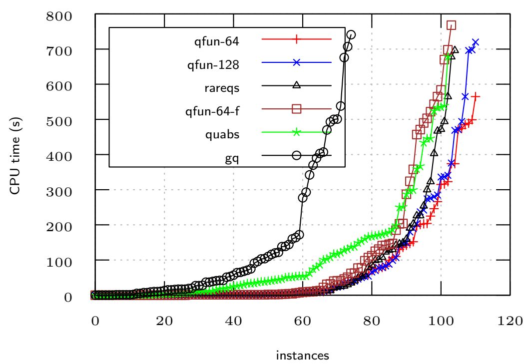

Fig. 1a Cactus plot. A point at (x, y) means that the solver solved x instances each within y sec.

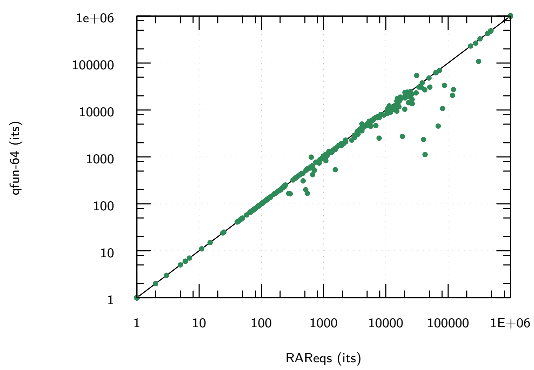

Fig. 1b Log-scale scatter plot for #iterations.

third step is whether also *solving time decreases when learning is applied*. Finally, we are interested in variations of the algorithm. Namely, the effect of the learning interval and the effect of the technique of accumulating strategies (see Section 4.3).

The evaluation considers the following configurations of algorithm qfun (Algorithm 3): qfun *without* any learning, which is in the fact RAReQS; versions qfun-16, qfun-64, and qfun-128 where learning is triggered every 16/64/128 iterations, respectively; qfun-64-f **f**orgetful version of qfun where previously learned strategies are not used in the future. All the other versions accumulate strategies as described in Section 4.3.

Additionally we compare to the highly competitive non-CNF solvers GhostQ [27] and QuAbS [43].

The prototype used for the evaluation is implemented in C^{++} and the SAT solver minisat 2.2 [12] is used as the backend solver. The experiments were carried out on Linux machines with Intel Xeon 5160 3GHz processors and 4GB of memory with the time limit 800 s and memory limit 2GB.

For the evaluation we used the non-CNF suite from the *2017 QBF Competition* counting 320 instances. [2]

The overall results are summarized in Table 1. The cactus plot in Fig. 1a summarizes the performance. For the sake of readability, the cactus plots omits qfun-16, whose performance is quite similar to qfun-64 and qfun-128, which are already quite close. Fig. 2b is a scatterplot comparing the total number of refinements for qfun-64 and RAReQS, i.e., machine learning every 64 iterations versus no learning. Detailed results are provided as supplementary material.

5.1 Results Discussion

Overall, learning gives improvement both in terms of number of solved instances as well as number of iterations. Admittedly, in terms of number of solved instances the gain is modest. However, the difference in performance between RAReQS and QuAbS is even smaller despite each representing a completely different algorithm. Also recall that Fig. 2b is in logarithmic scale so the number of iterations saved are in number of cases in orders of magnitude. Overall this suggests that adding learning in the brings about a new quality in the solver.

The effect of frequency of learning on the performance is relatively small. The best configuration is with learning every 64 refinements (qfun-64), while qfun-16 and qfun-128 perform slightly worse. This is not surprising as too frequent learning will slow down the solving and too infrequent does not give enough opportunity to learn.

The biggest effect has strategy accumulation. Indeed, without it, learning in fact performs worse then without any learning. This suggests that at least for some variables it is important to learn a certain strategy and maintain it. This observation clearly opens opportunities for further investigation as the techniques of accumulating strategies can be further developed.

6 Related Work

The research on QBF solving has been quite active in the last decades and an array of approaches exists. It appears that these different approaches also give us a different classes of instances where they are successful. One of the oldest approaches is conflict/solution learning [52,31,13,29], which essentially generalizes clause learning in SAT. Then there are solvers that perform quantifier expansion into Boolean connectives [8,5,30,34,44]; solvers that target non-CNF inputs [51,27,15,16,47,4,43]; and solvers that calculate blocking clauses using a SAT solver [38,21,37]. Recently we have also seen integration of inprocessing with conflict/solution learning [32].

[2] http://www.qbflib.org/event_page.php?year=2017

This paper builds on the algorithm RAReQS [20], which expands quantifiers gradually by substituting them one by one into the formula. This approach is conceptually akin to the *model-based quantifier instantiation* [48].

It is known that QBF solvers *implicitly* trace strategies because a winning strategy can be extracted once the formula is solved [17,2,3,7]. However, to our best knowledge there are currently only two QBF solvers that *explicitly* target strategy computation. In [10] the authors fused clause learning and RAReQS by refining abstractions with strategies calculated from clause learning—with not very promising results. The second solver by Rabe and Seshia works in the context of 2QBF and gradually adds variables to a winning strategy of the inner quantifier [36].

It is hard to do justice to the work that has been done in machine learning, the reader is directed to standard literature [40]. It should be mentioned that strategy learning is a very specific type of learning because we need the result in the form of a formula. This is closely related to function synthesis/learning cf. [46,24,33,42]. Machine learning has also been used in portfolio solvers e.g. [50] or to predict formulas' value [49].

Last but not least, machine learning has been used at a higher level of inference to discover lemmas in the context of first order or higher order reasoning [45,23].

7 Conclusion and Future Work

This paper presents a QBF solver that periodically *generalizes* a set of observations (plays) into a strategy by machine learning. These strategies are plugged into the original formula in order to gradually strengthen a partial expansion of the formula. The results show that this is feasible and it also helps to reduce the number of refinement iterations but also the solving time. The fact that this results in a competitive QBF solver is already compelling. Indeed, machine learning is invoked many times during solving on a number of variables separately. However, the design of the algorithm enables us to curb the computational burden of machine learning by limiting the size of the training set.

As discussed in Section 4, the current prototype is rather straightforward in its implementation decisions. There is a lot of room for making the solver more intelligent. Besides inprocessing and other implementation issues, number of things are to be investigated for the machine learning part. What kind of machine learning methods are good for this purpose? When to trigger machine learning? Can we improve the training sets (e.g. introduction of don't-cares)?

Another interesting question for future work is whether machine learning can be beneficial in other type of QBF solving. There are opportunities for this. Even if the solver is *not* performing expansion-based refinement (e.g. CAQE [37], QESTO [21], CADET [36]), it can for instance use a learned strategy to predict the behavior of the opponent.

At the theoretical level, the paper touches a fundamental question: how difficult is it to learn the right strategies? Here, PAC-learnability could give some answers [46].

References

1. Ansótegui, C., Gomes, C.P., Selman, B.: The Achilles' heel of QBF. In: The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference (AAAI). pp. 275–281 (2005)
2. Balabanov, V., Jiang, J.H.R.: Unified QBF certification and its applications. Formal Methods in System Design 41(1), 45–65 (2012)
3. Balabanov, V., Jiang, J.R., Janota, M., Widl, M.: Efficient extraction of QBF (counter)models from long-distance resolution proofs. In: Conference on Artificial Intelligence (AAAI). pp. 3694–3701 (2015)
4. Balabanov, V., Jiang, J.R., Mishchenko, A., Scholl, C.: Clauses versus gates in CEGAR-Based 2QBF solving. In: Beyond NP, AAAI Workshop (2016)
5. Benedetti, M.: sKizzo: a suite to evaluate and certify QBFs. In: International Conference on Automated Deduction (CADE). pp. 369–376 (2005)
6. Benedetti, M., Mangassarian, H.: QBF-based formal verification: Experience and perspectives. Journal on Satisfiability, Boolean Modeling and Computation (JSAT) 5(1-4), 133–191 (2008)
7. Beyersdorff, O., Chew, L., Janota, M.: On unification of QBF resolution-based calculi. In: International Symposium on Mathematical Foundations of Computer Science (MFCS). pp. 81–93. Springer (2014)
8. Biere, A.: Resolve and expand. In: Theory and Applications of Satisfiability Testing (SAT). pp. 238–246 (2004)
9. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
10. Bjørner, N., Janota, M., Klieber, W.: On conflicts and strategies in QBF. In: International Conferences on Logic for Programming LPAR-20, Short Presentations (2015)
11. Brummayer, R., Biere, A.: Local two-level and-inverter graph minimization without blowup. Proc. MEMICS 6, 32–38 (2006)
12. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Theory and Applications of Satisfiability Testing (SAT) (2003)
13. Giunchiglia, E., Marin, P., Narizzano, M.: QuBE 7.0 system description. Journal on Satisfiability, Boolean Modeling and Computation 7, 83–88 (2010)
14. Giunchiglia, E., Marin, P., Narizzano, M.: Reasoning with quantified boolean formulas. In: Biere et al. [9], pp. 761–780
15. Goultiaeva, A., Bacchus, F.: Exploiting QBF duality on a circuit representation. In: Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI) (2010)
16. Goultiaeva, A., Seidl, M., Biere, A.: Bridging the gap between dual propagation and CNF-based QBF solving. In: Design, Automation and Test in Europe, (DATE). pp. 811–814 (2013)
17. Goultiaeva, A., Van Gelder, A., Bacchus, F.: A uniform approach for generating proofs and strategies for both true and false QBF formulas. In: International Joint Conference on Artificial Intelligence (IJCAI). pp. 546–553 (2011)
18. Hellerman, L.: A catalog of three-variable or-invert and and-invert logical circuits. IEEE Trans. Electronic Computers 12(3), 198–223 (1963)

19. 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)
20. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.: Solving QBF with counterexample guided refinement. Artificial Intelligence 234, 1–25 (2016)
21. Janota, M., Marques-Silva, J.: Solving QBF by clause selection. In: International Conference on Artificial Intelligence (IJCAI). pp. 325–331. AAAI Press (2015)
22. Jordan, C., Klieber, W., Seidl, M.: Non-CNF QBF solving with QCIR. In: Proceedings of BNP (Workshop) (2016)
23. Kaliszyk, C., Urban, J.: Learning-assisted automated reasoning with Flyspeck. Journal of Automated Reasoning 53(2), 173–213 (2014)
24. Kamath, A.P., Karmarkar, N., Ramakrishnan, K.G., Resende, M.G.C.: A continuous approach to inductive inference. Math. Program. 57, 215–238 (1992)
25. Kleine Büning, H., Bubeck, U.: Theory of quantified boolean formulas. In: Biere et al. [9], pp. 735–760
26. Kleine Büning, H., Subramani, K., Zhao, X.: Boolean functions as models for quantified boolean formulas. J. Autom. Reasoning 39(1), 49–75 (2007)
27. Klieber, W., Sapra, S., Gao, S., Clarke, E.M.: A non-prenex, non-clausal QBF solver with game-state learning. In: SAT. pp. 128–142 (2010)
28. Letz, R.: Lemma and model caching in decision procedures for quantified boolean formulas. In: Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX). pp. 160–175 (2002), https://doi.org/10.1007/3-540-45616-3_12
29. Lonsing, F.: Dependency Schemes and Search-Based QBF Solving: Theory and Practice. Ph.D. thesis, Johannes Kepler Universität (2012), http://www.kr.tuwien.ac.at/staff/lonsing/diss/
30. Lonsing, F., Biere, A.: Nenofex: Expanding NNF for QBF solving. In: Theory and Applications of Satisfiability Testing (SAT). pp. 196–210 (2008)
31. Lonsing, F., Biere, A.: DepQBF: A dependency-aware QBF solver. Journal on Satisfiability, Boolean Modeling and Computation (JSAT) 7(2-3), 71–76 (2010)
32. Lonsing, F., Egly, U., Seidl, M.: Q-resolution with generalized axioms. In: Theory and Applications of Satisfiability Testing (SAT). pp. 435–452 (2016)
33. Oliveira, A.L., Sangiovanni-Vincentelli, A.L.: Learning complex boolean functions: Algorithms and applications. In: Advances in Neural Information Processing Systems NIPS. pp. 911–918 (1993)
34. Pigorsch, F., Scholl, C.: An AIG-based QBF-solver using SAT for preprocessing. In: Design Automation Conference, DAC. pp. 170–175 (2010)
35. Quinlan, J.R.: Induction of decision trees. Machine learning 1(1), 81–106 (1986)
36. Rabe, M.N., Seshia, S.A.: Incremental determinization. In: Theory and Applications of Satisfiability Testing (SAT). pp. 375–392 (2016)
37. Rabe, M.N., Tentrup, L.: CAQE: A certifying QBF solver. In: Formal Methods in Computer-Aided Design, FMCAD. pp. 136–143 (2015)
38. Ranjan, D.P., Tang, D., Malik, S.: A comparative study of 2QBF algorithms. In: Theory and Applications of Satisfiability Testing (SAT). pp. 292–305 (2004)
39. Rintanen, J.: Asymptotically optimal encodings of conformant planning in QBF. In: AAAI Conference on Artificial Intelligence. pp. 1045–1050. AAAI Press (2007)
40. Russell, S.J., Norvig, P.: Artificial intelligence: a modern approach. Prentice Hall (2010)
41. Schaefer, M., Umans, C.: Completeness in the polynomial-time hierarchy: A compendium. SIGACT news 33(3), 32–49 (2002)

42. Su, G., Wei, D., Varshney, K.R., Malioutov, D.M.: Learning sparse two-level boolean rules. In: IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP). pp. 1–6 (Sept 2016)
43. Tentrup, L.: Non-prenex QBF solving using abstraction. In: Theory and Applications of Satisfiability Testing (SAT). pp. 393–401 (2016)
44. Tu, K., Hsu, T., Jiang, J.R.: QELL: QBF reasoning with extended clause learning and levelized SAT solving. In: Theory and Applications of Satisfiability Testing (SAT). pp. 343–359 (2015)
45. Urban, J., Sutcliffe, G., Pudlák, P., Vyskočil, J.: MaLARea SG1- machine learner for automated reasoning with semantic guidance. In: International Joint Conference on Automated Reasoning (IJCAR). pp. 441–456 (2008)
46. Valiant, L.G.: A theory of the learnable. Commun. ACM 27(11), 1134–1142 (1984), http://doi.acm.org/10.1145/1968.1972
47. Van Gelder, A.: Primal and dual encoding from applications into quantified boolean formulas. In: CP. pp. 694–707 (2013)
48. Wintersteiger, C.M., Hamadi, Y., de Moura, L.M.: Efficiently solving quantified bit-vector formulas. Formal Methods in System Design 42(1), 3–23 (2013)
49. Xu, L., Hoos, H.H., Leyton-Brown, K.: Predicting satisfiability at the phase transition. In: Twenty-Sixth AAAI Conference on Artificial Intelligence (AAAI) (2012), http://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/5098
50. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. 32, 565–606 (2008), https://doi.org/10.1613/jair.2490
51. Zhang, L.: Solving QBF by combining conjunctive and disjunctive normal forms. In: The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference (AAAI). AAAI Press (2006)
52. Zhang, L., Malik, S.: Conflict driven learning in a quantified Boolean satisfiability solver. In: International Conference On Computer Aided Design (ICCAD). pp. 442–449 (2002)