Towards Generalization in QBF Solving via Machine Learning

Mikoláš Janota
INESC-ID / Instituto Superior Técnico,
Universidade de Lisboa, Portugal
mikolas.janota@gmail.com

Abstract
There are well known cases of Quantified Boolean Formulas (QBFs) that have short winning strategies (Skolem/Herbrand functions) but that 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, which enables learning shorter strategies. The implemented prototype QFUN has won the first place in the non-CNF track of the most recent QBF competition.

1 Introduction
Against all odds posed by computational complexity, logic-based problem solving had a remarkable success at research but also at 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 (Benedetti and Mangasearan 2008; Rintanen 2007; Schaefer and Umans 2002; Giunchiglia, Marin, and Narizzano 2009). 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. (Zhang and Malik 2002; Benedetti 2005; Tu, Hsu, and Jiang 2015; Gouliaeva and Bacchus 2010; Gouliaeva, Seidl, and Biere 2013; Janota and Marques-Silva 2015; Rabe and Tentrup 2015; Janota et al. 2012; Lonsing, Egly, and Seidl 2016; Tentrup 2016; Rabe and Seshia 2016). 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 \( Y = \{y_1, \ldots, y_n\} \) (Letz 2002). 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 (Kleine Büning, Subramani, and Zhao 2007)). 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 (Janota et al. 2016). 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; complementary literal is denoted as \( l \), i.e. \( \bar{x} = \neg x \), \( \neg \bar{x} = x \). For a literal \( l = x \) or \( l = \neg x \), we write \( \text{var}(l) \) for \( x \). Analogously, \( \text{vars}(\phi) \) is the set of all variables in formula \( \phi \). An assignment is a mapping from variables to Boolean constants 0, 1. For a formula \( \phi \) and an assignment \( \tau \), we write \( \phi[\tau] \) for the application of the assignments to \( \phi \).

2.1 Quantified Boolean Formulas
Quantified Boolean Formulas (QBFs) (Kleine Büning and Bubeck 2009) extend propositional logic by enabling quantification over Boolean variables. Any propositional formula \( \phi \) is also a QBF with all variables free. If \( \Phi \) 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 $\exists x. \exists x. \ldots$, 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 $\Phi$ and is existential in $\exists x. \Psi$. Analogously, a literal $l$ is universal (resp. existential) if $\text{var}(l)$ 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 $\tau$ with $Q \in \{\forall, \exists\}$.

A QBF corresponds to a propositional formula: $\forall x. \Psi$ corresponds to $\Psi[x \rightarrow 0] \land \Psi[x \rightarrow 1]$ and $\exists x. \Psi$ to $\Psi[x \rightarrow 0] \lor \Psi[x \rightarrow 1]$. 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 $Q_1X_1 \ldots Q_nX_n. \phi$, where $Q_i \in \{\exists, \forall\}$, $Q_1 \neq Q_{i+1}$, $\phi$ propositional, and $X_i$ pairwise disjoint sets of variables. The propositional part $\phi$ 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 $\text{lv}(x) = i$; we write $\text{lv}(l)$ for $\text{lv}(\text{var}(l))$. 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 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 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, $Q$ for its opponent.

Given a QBF $Q_1X_1 \ldots Q_nX_n. \phi$ the domain $\text{dom}(x)$ of a variable $x \in X_k$ are all the variables in the preceding blocks, i.e. $\text{dom}(x) = \bigcup_{i \in 1..k-1} X_i$. A play is a sequence of assignments $\tau_1, \ldots, \tau_n$ where $\tau_i$ is an assignment to $X_i$.

**Definition 1** For a QBF $Q_1X_1 \ldots Q_nX_n. \phi$ a strategy for a variable $x \in X_k$ is a Boolean function $s_x$ whose arguments are the variable’s domain, i.e. $\text{dom}(x)$.

A strategy for a player $Q$ 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 $s_x : \text{dom}(x) \rightarrow \{0, 1\}$ and some formula $\psi_x$ with $\text{vars}(\psi_x) \subseteq \text{dom}(x)$. 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 $\xi$ each variable $x \in X$ with its strategy $\psi_x$.

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

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

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

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

For a closed QBF $QXQY. \Phi$ and an assignment $\tau$ to $X$, an assignment $\mu$ to $Y$ is a counter-move to $\tau$ if $\mu$ is a winning move for $QY. \Phi[\tau]$.

**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 $QXQY. \Phi$, an assignment to $X$ is a winning move iff there is not 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 overview 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 RARE-QS (Janošta et al. 2016). For a formula $QXQY. \Phi$ RARE-QS 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 $\tau_i$ is an assignment to $X$ and $\mu_i$ is a counter-move to $\tau_i$ (see Def. 3). In each iteration, RARE-QS constructs a partial expansion (called abstraction) of the original QBF such that no existing $\mu_i$ 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 $\tau_{k+1}$ for the abstraction, check whether the opponent still comes up with a counter-move $\mu_{k+1}$. If he does not, $\tau_{k+1}$ is a winning move for $Q$ and we are done (see Observation 2). If a counter-move is found, the pair $(\tau_{k+1}, \mu_{k+1})$ is added to the sequence and the process repeats.

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

It is tempting to learn a strategy for $QY$ 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 $E$.
2. Learn strategies $S$ for the opponent variables.
3. Strengthen the current abstraction using the strategies $S$.
4. Reset the set of samples $E$.
5. Repeat.

3.2 QFUN$^2$: 2-level QBF

Let us look at the two-level case, i.e., a QBF of the form $Q XQY. φ$. 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. (Benedetti and Mangassarian 2008; Rintanen 2007; Schaefer and Umans 2002)).

A slight generalization of a game called a multi-game (Janota et al. 2016) 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 (Janota et al. 2016, Sec. 4.1)).

Definition 4 (multi-game) A multi-game is written as $Q X. Φ_i$. An assignment $τ$ to $X$ is a winning move for it if it is a winning move for all $Q X. Φ_i$. Each $Φ_i$ is called a sub-game and is either propositional or begins with $Q$.

When all sub-games are propositional, the multi-game is solvable by a single SAT call. For such we introduce a function Winsl (Algorithm 1). The function calculates a winning move for the multi-game or returns ⊥ 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 Winsl 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. Hence, for a formula $Q XQY. φ$, the abstraction has the form $Q X. Φ[S] | S ∈ ω$, 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, $∀ u. φ[e=0] ∨ φ[e=1]$, which is equivalent to the multi-game $∀ u. {φ[e=0], φ[e=1]}$.

Example 1 Consider the formula $∀ u w x y. φ$ with $φ = (u ⇒ (¬ w \equiv x ∧ w \equiv y)) ∧ (¬ u ⇒ (w ⇒ x ∧ ¬ w \equiv y)))$. The following abstractions of this formula are both losing for $∀$. With two sub-games: $∀ u w. \{φ[x⋅¬ w, y⇒ w], φ[x w, y⇔ ¬ w]\}$; with single sub-game: $∀ u w. \{φ[x⇒ u ? ¬ w : w], y⇒ u ? w : ¬ w]\}$.

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 $E$. 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 $Q$. So for instance, if $Q = ∃ τ$ then $Q = ∃ τ$ and $τ_i ∪ μ_i = φ$.

Both the abstraction $α$ and samples $E$ are initialized as empty. In each iteration, QFUN$^2$ calls Winsl to calculate a candidate for a winning move $τ$. Subsequently, another call to Winsl 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 pseudo-code queries the function ShouldLearn. Whenever ShouldLearn is true, new strategies are learned for $Y$-variables based on the sam-
ment guarantees that he can successfully defend himself against all the ex-
isting counter-moves. Once strategies are also included, the
strategy only finitely many times, the learning method also
be filtered out (ln. 8).
So what does the abstraction represent and what is the role
from the strategy learning? The good
news is that in fact very little. A strategy must be learned in
each opponent’s variable and further, the
number of iterations can go to millions.
What do we require from the strategy learning? The good
news is that the player never plays before making sure
that a strategy formula \( \Phi \) will not appear as a candidate for
a variable \( y \).

### 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. Candi-
ates for a winning-move are drawn from the abstraction \( \alpha \) 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 \( \tau \) is a winning move, it is returned. If, however, there is some counter-move \( \mu \), obtained by playing a sub-game \( \Phi_\mu \), it is used for refinement. This means in-
serting the pair \((\tau, \mu)\) into the sample sequence pertaining to this sub-game, i.e. sequence \( \mathcal{E}_i \), and subsequently, per-
forming refinement. In order to ensure that quantifiers alter-
ate, refinement introduces fresh variables for formulas with more than 2 levels. The refinement function is defined as follows.

\[
\text{Refine}(QX_1, \Psi, S) := QX_1', \Psi' \psi[S]
\]

where \( X_1' \) are fresh duplicates of the variables \( X_1 \) and \( \Psi' \) is \( \Psi \) with \( X_1 \) replaced by \( X_1' \) and where \( \psi \) is a proposi-
tional 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 repre-
sentation as input has well-known pitfalls (Ansótegui, Gomes, and Selman 2005; Janota and Marques-Silva 2017). Hence, the implementation represents formulas as And-
Inverter graphs (AIG) (Hellerman 1963), which are simpli-
fied by trivial non-invasive simplifications (Brummayer and Biere 2006). 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-coned, each sub-AIG also corresponds to just one variable. All the and-gates are bi-

<table>
<thead>
<tr>
<th>Algorithm 3: QBF Refinement with Learning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong> QFUN(QX, {\Phi_1, ..., \Phi_n})</td>
</tr>
<tr>
<td><strong>input</strong>: Each ( \Phi_i ) is propositional or begins with ( \bar{Q}Y ).</td>
</tr>
<tr>
<td><strong>output</strong>: a win. move for QX if exists, ( \perp ) otherwise</td>
</tr>
<tr>
<td>1 if all ( \Phi_i ) propositional then</td>
</tr>
<tr>
<td>2 [ return \text{Wins}(QX, {\Phi_1, ..., \Phi_n}) ]</td>
</tr>
<tr>
<td>3 ( \mathcal{E}_i \leftarrow \emptyset, i \in {1..n} ) // samples</td>
</tr>
<tr>
<td>4 ( \alpha \leftarrow QX, \emptyset ) // empty abstraction</td>
</tr>
<tr>
<td>while true do</td>
</tr>
<tr>
<td>5 ( \tau' \leftarrow \text{QFUN}(\alpha) ) // candidate</td>
</tr>
<tr>
<td>6 if ( \tau' = \perp ) then return ( \perp ) // loss</td>
</tr>
<tr>
<td>7 ( \tau \leftarrow {l \mid l \in \tau' \land \text{var}(l) \in X} ) // filter</td>
</tr>
<tr>
<td>8 if all ( \text{QFUN}(\Phi_i[\tau]) = \perp ) then return ( \tau ) // win</td>
</tr>
<tr>
<td>9 let ( \bar{l} ) be s.t. ( \text{QFUN}(\Phi_i[\tau]) = \mu ) for some</td>
</tr>
<tr>
<td>( l \in {1..n}, \mu \neq \perp )</td>
</tr>
<tr>
<td>10 ( \mathcal{E}_i \leftarrow \mathcal{E}_i \cup {(\tau, \mu)} ) // record sample</td>
</tr>
<tr>
<td>11 if ShouldLearn() then</td>
</tr>
<tr>
<td>12 ( S \leftarrow \text{Learn}(\mathcal{E}_i) ) // learn</td>
</tr>
<tr>
<td>13 ( \alpha \leftarrow \text{Refine}(\alpha, \Phi_i, S) )</td>
</tr>
<tr>
<td>14 ( \mathcal{E}_i \leftarrow \emptyset ) // reset samples</td>
</tr>
<tr>
<td>15 else</td>
</tr>
<tr>
<td>16 ( \alpha \leftarrow \text{Refine}(\alpha, \Phi_i, \mu) ) // refine</td>
</tr>
</tbody>
</table>
The input to the solver is the circuit-like format for QBF called QCIR (Jordan, Klieber, and Seidl 2016).

### 4.2 Learning

Recall that learning is invoked with the sequence of pairs of assignments \( E = (\tau_1, \mu_1), \ldots, (\tau_k, \mu_k) \), where each \( \tau_i \) is an assignment to some block of variables \( X \) in the prefix and \( \mu_i \) 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 (Russell and Norvig 2010). These are constructed by the standard ID3 algorithm (Quinlan 1986).

For each variable in \( y \in Y \), construct the training set \( E_y \) from \( E \) by ignoring all the other \( Y \) variables. Subsequently invoke ID3 on \( E_y \) 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 \( I_p \) and \( I_n \) corresponding to the positive and negative branches of the tree, respectively. Hence, if \( t \in I_p \) is true, the tree gives 1.
2. Repeatedly apply subsumption and self-subsumption on each set \( I_p \) and \( I_n \), until a fixed point is reached.
3. If \( |I_p| < |I_n| \), return \( \lor I_p \), otherwise return \( \lor I_n \).

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. Some heavier methods could be considered here, cf. (Ignatiev, Previti, and Marques-Silva 2015).

**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 QFUNCTION. 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 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: \( \forall x_1, \ldots, x_n. \exists y_1, \ldots, y_n. (x_1 \leftrightarrow y_1) \wedge \text{and, the following sequence of samples.} \)

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( \ldots )</th>
<th>( x_n )</th>
<th>( y_1 )</th>
<th>( y_2 )</th>
<th>( \ldots )</th>
<th>( y_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>\ldots</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>\ldots</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>\ldots</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>\ldots</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>\ldots</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>\ldots</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>\ldots</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>\ldots</td>
<td>1</td>
</tr>
</tbody>
</table>

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

Some similar functions were in fact learned during the experimental part. For instance, I have observed the solver learn \( y \equiv x \) for \( \forall x \ldots \exists y \ldots (F \wedge (y \Rightarrow (x \wedge G))) \wedge (H \vee \neg x \vee y) \) for some larger formulas \( F, G, H \). Theoretically, learning can be worse than traditional refinement, e.g., \( \ldots \exists x_1 \ldots x_{100}. (x_1 \wedge \ldots \wedge x_{100}) \vee G \) it is clearly better refine with \( x_i \equiv 1 \) rather than some complicated functions.

### 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 has placed first in several tracks of the recent QBF competitions. 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 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 forgetful 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 (Klieber et al. 2010) and QuAbS (Tentrup 2016).

The evaluated prototype is implemented in $C^{\dagger\dagger}$ and minisat 2.2 (Eén and Sörensson 2003) 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.\footnote{\url{www.qbplib.org/event_page.php?year=2017}}

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. 1b is a scatterplot comparing the total number of refinements for QFUN-64 and RAReQS, i.e., machine learning every 64 iterations versus no learning; all instances are displayed and time/mem-outs are placed on the edges. The learning time observed was the relatively small, for instance on CM-sat-07-01-07-3 with 425+917 variables, each learning round took 0.8 s with 200 s being the total solving time.

### 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. 1b 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 (Zhang and Malik 2002; Lonsing and Biere 2010; Giunchiglia, Marin, and Narizzano 2010; Lonsing 2012), which essentially generalizes clause learning in SAT. Then there are solvers that perform quantifier expansion into Boolean connectives (Biere 2004; Benedetti 2005; Lonsing and Biere 2008; Pigorsch and Scholl 2010; Tu, Hsu, and Jiang 2015); solvers that target non-CNF inputs (Zhang 2006; Klieber et al. 2010; Goultiaeva and Bacchus 2010; Goultiaeva, Seidl, and Biere 2013; Van Gelder 2013; Balabanov et al. 2016; Tentrup 2016); and solvers that calculate blocking clauses using a SAT solver (Ranjan, Tang, and Malik 2004; Janota and Marques-Silva 2015; Rabe and Tentrup 2015). Recently we have also seen integration of inprocessing with conflict/solution learning (Lonsing, Egly, and Seidl 2016).

This paper builds on the algorithm RAReQS (Janota et al. 2016), which expands quantifiers gradually by substituting them one by one into the formula. This approach is conceptually akin to the *model-based quantifier instantiation* (Wintersteiger, Hamadi, and de Moura 2013).

It is known that QBF solvers *implicitly* trace strategies because a winning strategy can be extracted once the formula is solved (Goultiaeva, Van Gelder, and Bacchus 2011;
Balabanov and Jiang 2012; Balabanov et al. 2015; Beyersdorff, Chew, and Janota 2014). However, to our best knowledge there are currently only two QBF solvers that explicitly target strategy computation. In (Bjørner, Janota, and Klieber 2015) 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 (Rabe and Seshia 2016).

It is hard to do justice to the work that has been done in machine learning, the reader is directed to standard literature (Russell and Norvig 2010). 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. (Valiant 1984; Kamar et al. 1992; Oliveira and Sangiovanni-Vincentelli 1993; Su et al. 2016). Machine learning has also been used in portfolio solvers e.g. (Xu et al. 2008).

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 (Urban et al. 2008; Kaliszyk and Urban 2014).

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 (Rabe and Tentrup 2015), QESTO (Janota and Marques-Silva 2015), CADET (Rabe and Seshia 2016)), 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 (Valiant 1984).

Acknowledgments. This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013.

References


Goultiaeva, A.; Van Gelder, A.; and Bacchus, F. 2011. A uniform approach for generating proofs and strategies for both
true and false QBF formulas. In International Joint Conference on Artificial Intelligence (IJCAI), 546–553.


