A structural approach to reasoning with quantified Boolean formulas *

Luca Pulina and Armando Tacchella

DIST, Università di Genova, Viale Causa, 13 – 16145 Genova, Italy
Luca.Pulina@unige.it Armando.Tacchella@unige.it

Abstract

In this paper we approach the problem of reasoning with quantified Boolean formulas (QBFs) by combining search and resolution, and by switching between them according to structural properties of QBFs. We provide empirical evidence that QBFs which cannot be solved by search or resolution alone, can be solved by combining them, and that our approach makes a proof-of-concept implementation competitive with current QBF solvers.

1 Introduction

The development of the first practically efficient QBF solver can be traced back to [4], wherein a search-based solver is proposed. Since then, researchers have proposed improvements to the basic search algorithm including, e.g., backjumping and learning [7], as well as different approaches to the problem including, e.g., skolemization [1], and resolution [2, 12]. However, in spite of such efforts, current QBF reasoners still cannot handle many instances of practical interest. This is witnessed, e.g., by the results of the 2008 QBF solver competition (QBFEVAL'08) where even the most sophisticated solvers failed to decide encodings obtained from real-world applications [14].

In this work, we consider search and resolution for QBFs, and we propose a new approach that combines them dynamically. The key point of our approach is to implicitly leverage graph abstractions of QBFs to yield structural features which support the decision between search and resolution. As for the choice of a particular feature, it is known, see, e.g., [16], that resolution on plain Boolean formulas may require computational resources that are exponential in the treewidth.[1] This connection has been studied also for quantified formulas in [5, 8, 13]. In particular, in [5], an extension of treewidth is shown to be related to the efficiency of reasoning about quantified Boolean constraints, of which QBFs are a subclass. This result tells us that treewidth is a promising structural parameter to gauge resolution and search: Resolution is the choice when treewidth is relatively small, and search is the alternative when treewidth is relatively large. Switching between the two

*The results herein presented are also detailed in a paper in proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI'09).

[1] Induced width, as defined in [16], is shown to be equivalent to treewidth in [9].

Proceedings of the 16th International RCRA workshop (RCRA 2009):
Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion
Reggio Emilia, Italy, 11–12 December 2009

options may occur as long as search is able to obtain subproblems whose treewidth is smaller than the one of the original problem. In practice, QBFs of some interest for applications may have thousands of variables, and thus even approximations of treewidth – a well-known NP-complete problem – are too expensive to compute on-the-fly [15]. Also, while deciding if treewidth is bounded by a constant can be done in asymptotic linear time [3], an actual implementation of such algorithm would be too slow to be of practical use. In this paper we show how these issues can be overcome by considering alternative parameters indicating whether resolution is bound to increase the size of the initial set of clauses beyond "reasonable" limits. To this purpose, we compute estimates of the number of new clauses generated during resolution, and we require that the number of such clauses is less than the number of old clauses used to generate them. Also, we require that the expected number of clauses is less than a predefined threshold. Both these conditions can be checked without a prohibitive overhead, and respecting them can be as informative as tracking treewidth in many situations.

To test our approach, we implemented it in STRUQS (for "Stru"ctural "Q"BF "S"olver), a proof-of-concept tool consisting of less than 2K lines of C++ code. We have experimented with STRUQS on the QBFEVAL'08 dataset [14], which is comprised of more than three thousands QBF encodings in various application domains – the largest publicly available collection of QBFs to date. Here, we show that exploiting dynamic combination of search and resolution enables STRUQS to solve QBFs which, all other things being equal, cannot be solved by the search and resolution components of STRUQS alone. Moreover, STRUQS is competitive with respect to current QBF solvers, as it would have ranked third best among QBFEVAL'08 competitors if considering the number of problems solved within a given amount of resources.

While approaches similar to ours have been proposed for plain Boolean satisfiability in [16], and for non-quantified constraint satisfaction in [11], we notice that this is the first time that a structural approach combining search and resolution is designed, implemented and empirically tested for QBFs. Learning in search-based QBF solvers such as, e.g., QUBE [7] and YQUAFFLE [18], can also be seen as a way to integrate resolution and search. However, as shown in [7], this is true of a particular form of control strategy for general resolution known as tree resolution, while here we focus on variable elimination instead. Moreover, since STRUQS features backjumping, and backjumping is essentially space-bounded tree resolution, we could say that our work extends the above contributions in this sense. The solver 2CLSQ [17] is another example of integration between search and resolution. Also in this case, the contribution is limited to a specific form of resolution, namely hyper-binary resolution, which is seen as a useful complement to the standard unit clause resolution step performed by most search-based QBF solvers, and not as an alternative resolution-based decision procedure as we do in STRUQS. Also QUANTOR [2] provides some kind of integration between resolution and search, but with two fundamental differences with respect to STRUQS. The first one is that QUANTOR intertwines resolution with the expansion of univer-

sally quantified variables, whereas in STRUQS no such expansion is performed. The second one is that QUANTOR uses search only at the very end of the decision process, when the input QBF has been reduced to an equi-satisfiable propositional formula.

The paper is structured as follows. In Section 2 we lay down the foundations of our work considering relevant definitions and known results. In Section 3 we define the basic STRUQS algorithms, including search, resolution and their combination. In Section 4 we describe our experiments and in Section 5 we summarize our current results and future research directions.

2 Groundwork

A *variable* is an element of a set P of propositional letters and a *literal* is a variable or the negation thereof. We denote with $|l|$ the variable occurring in the literal l, and with $\bar{l}$ the *complement* of l, i.e., $\neg l$ if l is a variable and $|l|$ otherwise. A *clause* C is an n-ary ($n \geq 0$) disjunction of literals such that, for any two distinct disjuncts l, l' of C, it is not the case that $|l| = |l'|$. A *propositional formula* is a k-ary ($k \geq 0$) conjunction of clauses. A *quantified Boolean formula* is an expression of the form

$$Q_1 z_1 \dots Q_n z_n \Phi$$

where, for each $1 \leq i \leq n$, z_i is a variable, Q_i is either an existential quantifier $Q_i = \exists$ or a universal one $Q_i = \forall$, and Φ is a propositional formula in the variables $Z = \{z_1, \dots, z_n\}$; the expression $Q_1 z_1 \dots Q_n z_n$ is the *prefix* and Φ is the *matrix*; a literal l is *existential* if $|l| = z_i$ for some $1 \leq i \leq n$ and $\exists z_i$ belongs to the prefix, and it is *universal* otherwise.

A prefix $p = Q_1 z_1 \dots Q_n z_n$ can be viewed as the concatenation of h *quantifier blocks*, i.e., $p = Q_1 Z_1 \dots Q_h Z_h$, where the sets Z_i with $1 \leq i \leq h$ are a partition of Z, and consecutive blocks have different quantifiers. To each variable z we can associate a *level* $lvl(z)$ which is the index of the corresponding block, i.e., $lvl(z) = i$ for all the variables $z \in Z_i$. We also say that variable z *comes after* a variable z' in p if $lvl(z) \geq lvl(z')$.

The semantics of a QBF φ can be defined recursively as follows. A QBF clause is *contradictory* exactly when it does not contain existential literals. If φ contains a contradictory clause then φ is false. If φ has no conjuncts then φ is true. If $\varphi = Qz\psi$ is a QBF and l is a literal, we define φ_l as the QBF obtained from ψ by removing all the conjuncts in which l occurs and removing $\bar{l}$ from the others. Then we have two cases:

- If φ is $\exists z\psi$, then φ is true exactly when φ_z or $\varphi_{\neg z}$ are true.
- If φ is $\forall z\psi$, then φ is true exactly when φ_z and $\varphi_{\neg z}$ are true.

The QBF decision problem (QSAT) can be stated as the problem of deciding whether a given QBF is true or false.

Given a QBF φ on the set of variables $Z = \{z_1, \ldots, z_n\}$, its *Gaifman graph* has a vertex set equal to Z with an edge (z, z') for every pair of different elements $z, z' \in Z$ that occur together in some clause of φ. A *scheme* for a QBF φ having prefix p is a supergraph (Z, E) of the Gaifman graph of φ along with and ordering $z'_1, \ldots, z'_n$ of the elements of Z such that

1. the ordering $z'_1, \ldots, z'_n$ preserves the order of p, i.e., if $i < j$ then z'_j comes after z'_i in p, and

2. for any z'_k, its lower numbered neighbors form a clique, that is, for all k, if $i < k$, $j < k$, $(z'_i, z'_k) \in E$ and $(z'_j, z'_k) \in E$, then $(z'_i, z'_j) \in E$.

The *width* $w_p(\varphi)$ of a scheme is the maximum, over all vertices z_k, of the size of the set $\{i : i < k, (z_i, z_k) \in E\}$, i.e., the set containing all lower numbered neighbors of z_k. The *treewidth* $tw_p(\varphi)$ of a QBF φ is the minimum width over all schemes for φ.

Clause resolution [10] for a QBF φ is an operation whereby given two clauses $Q \vee x$ and $R \vee \neg x$ of φ, the clause $\min(Q \vee R)$ can be derived, where

1. Q and R are disjunctions of literals,

2. x is an existential variable,

3. Q and R do not share any literal l such that l occurs in Q and $\bar{l}$ occurs in R, and

4. $\min(C)$ is obtained from C by removing the universal literals coming after all the existential literals in C.

Variable elimination is a control strategy for resolution defined as follows. Given a QBF φ on the set of variables $Z = \{z_1, \ldots, z_n\}$, we consider a scheme for φ and the associated *elimination ordering* $Z' = \{z'_1, \ldots, z'_n\}$. Starting from $z = z'_n$ and scanning the elimination ordering in reverse, we *eliminate* the variables as follows. If z is existential, and assuming that φ contains $k > 0$ clauses in the form $Q \vee z$ and $h > 0$ clauses in the form $R \vee \neg z$, then we add $O(kh)$ clauses to φ obtained by performing all the resolutions on z. In the case where either $k = 0$ or $h = 0$, no operation is performed. If z is universal, we simply skip to the next variable. If all the variables in φ can be eliminated without generating a contradictory clause, then φ is true, else it is false.

Backtracking search [4] for a QBF φ is a depth-first exploration of an *AND-OR tree* defined as follows. Initially, the current QBF is φ. If the current QBF is of the form $\exists x\psi$ then we create an *OR-node*, whose children are obtained by checking recursively whether φ_x is true or $\varphi_{\neg x}$ is true. If the current QBF is of the form $\forall y\psi$ then we create an *AND-node*, whose children are obtained by checking recursively whether both φ_y and $\varphi_{\neg y}$ are true. We call φ_l a *branch* (also, an *assignment*), and since we explore the tree in a depth-first fashion, a node wherein only the first branch was taken is said to be *open*, and *closed* otherwise. The leaves are of two

kinds: If the current QBF contains a contradictory clause we have a *conflict*, while if the current QBF contains no conjuncts we have a *solution*. Backtracking from a conflict amounts to reaching back to the deepest open OR-node: if there is no such node, then φ is false; backtracking from a solution amounts to reaching back to the deepest open AND-node: if there is no such node, then φ is true.

Unit propagation [4] is an optimization that can be added on top of basic variable elimination and search. A clause C is *unit* in a QBF φ exactly when (i) C contains only one existential literal l (*unit literal*) and, (ii) all the universal literals in C come after l in the prefix of φ. If φ contains a unit literal l, then φ is true if and only if φ_l is true. Unit propagation is the process whereby we keep assigning unit literals until no more such literals can be found.

Backjumping is an optimization that can be added on top of search. According to [7] the computation performed by a QBF search algorithm corresponds to a particular kind of deductive proof, i.e., a tree wherein clause resolution and *term resolution* alternate, where a *term* is a conjunction of literals, and term resolution is the "symmetric" operation of clause resolution – see [7] for details. The clause/term tree resolution proof corresponding to search for a QBF φ can be reconstructed as follows. In case of a conflict, there must be two clauses, say $Q \vee x$ and $R \vee \neg x$ such that all the literals in Q and R have been deleted by previous assignments. We can always resolve such clauses to obtain the *working reason* $\min(Q \vee R)$ – where $Q \vee x$ and $R \vee \neg x$ are the *initial* working reasons. The other clauses can be derived from the working reason when backtracking over any existential literal l such that $|l|$ occurs in the working reason. There are three cases:

1. If l is a unit literal, then there is a clause C where l occurs – the *reason* of assigning l – and we obtain a new working reason by resolving the current one with C.

2. If l is an open branch, then proof reconstruction stops, because we must branch on $\bar{l}$, and the reason of this assignment is the current working reason.

3. If l is a closed branch, then its reason was computed before, and it can be treated as in (1).

In case of a solution, the initial working reason is a term which is the conjunction of (a subset of) all the literals assigned from the root of the search tree, down to the current leaf, i.e., a Boolean implicant of the matrix of φ. The other terms can be derived when backtracking over any universal literal l such that l is in the working reason, considering cases (ii) and (iii) above, and using term instead of clause resolution.[2] Given the above, it is easy to see that closing branches over existential (resp. universal) literals that do not appear in the current working reason is useless, i.e., they are not responsible for the current conflict (resp. solution). Backjumping can thus be described as the process whereby useless branches are skipped while backtracking.

[2]Case (1) will not apply unless terms are learned, and thus unit propagation may involve universal variables as well – see [7] for the technical details.

```
SEARCH(φ, Γ)
  l ← CHOOSELIT(φ, Γ)
  push(Γ, ⟨l, LS, NIL⟩)
  DOASSIGN(φ, l)

ELIMINATE(φ, Γ)
  v ← CHOOSEVAR(φ, Γ)
  push(Γ, ⟨v, VE, NIL⟩)
  DOELIMINATE(φ, v)

BACKTRACK(φ, Γ, q)
  while Γ ≠ EMPTY do
    ⟨l, m, _⟩ ← top(Γ)
    if m = VE then
      UNDOELIMINATE(φ, l)
    else
      UNDOASSIGN(φ, l)
    pop(Γ)
    if ISBOUND(q, l, φ) then
      if m = LS then
        push(Γ, ⟨l̄, FL, NIL⟩)
        DOASSIGN(φ, l̄)
        return FALSE
  return TRUE

PROPAGATE(φ, Γ)
  ⟨l, r⟩ ← FINDUNIT(φ)
  while l ≠ NIL do
    push(Γ, ⟨l, FL, r⟩)
    DOASSIGN(φ, l)
    ⟨l, r⟩ ← FINDUNIT(φ)

BACKJUMP(φ, Γ, q)
  wr ← INITREASON(φ, Γ, q)
  while Γ ≠ EMPTY do
    ⟨l, m, r⟩ ← top(Γ)
    if m = VE then
      UNDOELIMINATE(φ, l)
    else
      UNDOASSIGN(φ, l)
    pop(Γ)
    if ISBOUND(q, l, φ) then
      if OCCURS(|l|, wr) then
        if m = LS then
          push(Γ, ⟨l̄, FL, wr⟩)
          DOASSIGN(φ, l̄)
          return FALSE
        else if m = FL then
          UPDATEREASON(wr, r)
  return TRUE
```

Figure 1: Pseudo-code of STRUQS basic routines.

3 Implementing the structural approach

In Figure 1 we present STRUQS basic routines in pseudo-code format. All the routines take as by-reference parameters the data structure φ which encodes the input QBF, and a stack Γ which keeps track of the steps performed. We consider the following primitives for φ:

- DOASSIGN(φ, l) implements φ_l as per Section 2, where clauses and literals are disabled rather than deleted.
- DOELIMINATE(φ, v) implements the elimination of a single variable as described in Section 2, with the addition that clauses subsumed by some clause already in φ are not added – a process named *forward subsumption* in [2]; as in the case of DOASSIGN, the eliminated variables and the clauses where they occur are disabled.
- FINDUNIT(φ), returns a pair $\langle l, r\rangle$ if l is a unit literal in φ and r is a unit clause in which l occurs, or a pair $\langle$NIL , NIL $\rangle$ if no such literal exists in φ.
- CHOOSELIT(φ, Γ) returns a literal l subject to the constraint that all the remaining variables of φ which do not appear in Γ come after $|l|$.
- CHOOSEVAR(φ, Γ) returns a variable v subject to the constraint that v comes after all the remaining variables of φ which do not appear in Γ.

Because updates in DOASSIGN and DOELIMINATE are not destructive, we can further assume that their effects can by "undo" functions, i.e., UNDOASSIGN and UNDOELIMINATE.

The routines SEARCH and ELIMINATE (Figure 1, top-left) perform one step of search and variable elimination, respectively. SEARCH (resp. ELIMINATE), asks CHOOSELIT (resp. CHOOSEVAR) for a literal l (resp. a variable v) and then it (i) pushes l (resp. v) in the stack and (ii) it updates φ. In both cases step (i) amounts to push a triple $\langle u, m, r\rangle$ in Γ where r is always NIL , u is the literal being assigned or the variable being eliminated, and m is the *mode* of the assignment, where LS ("L"eft "S"plit) indicates an open branch in the search tree, and VE ("V"ariable "E"limination) indicates the corresponding resolution step. The task of PROPAGATE (Figure 1, top-right) is to perform unit propagation in φ. For each unit literal l, PROPAGATE pushes a record $\langle l, \text{FL}, r\rangle$ in Γ, where l and r are the results of FINDUNIT, and the mode is always FL ("F"orced "L"iteral).

BACKTRACK (Figure 1 bottom-left) and BACKJUMP (Figure 1, bottom-right) are alternative backtracking procedures to be invoked at the leaves of the search tree with an additional parameter q, where $q = \exists$ if the current leaf is a conflict, and $q = \forall$ otherwise. BACKTRACK goes back in the stack Γ, popping records for search (LS) or deduction steps (VE, FL), and undoing the corresponding effects on φ. Notice that, if BACKTRACK is a component of a plain search solver, then VE-type records will not show up in Γ. BACKTRACK stops when an open branch (LS) corresponding to a literal l bound by the quantifier q is found (ISBOUND(q, l, φ)=TRUE). In this case, BACKTRACK closes the branch by assigning $\bar{l}$ with mode FL and returning FALSE to indicate that search should not stop. On the other hand, if all the records are popped from Γ without finding an open branch then search is over and BACKTRACK returns TRUE.

BACKJUMP implements backjumping as per Section 2 on top of the following primitives:

- INITREASON(φ, Γ, q) initializes the working reason wr. If $q = \exists$, then we have two cases: (i) If **top**(Γ) is a FL-type record, then wr is any clause in φ which is contradictory because all of its existential literals have been disabled. (ii) If **top**(Γ) is a VE-type record, this means that a contradictory clause C has been derived when eliminating a variable, and thus wr is C. If $q = \forall$, then wr is a term obtained considering a subset of the literals in Γ such that the number of universal literals in the term is minimized – VE-type records are disregarded in this process.

- UPDATEREASON(wr, r) updates wr by resolving it with r, i.e., the reason of the assignment which is being undone.

When BACKJUMP is invoked in lieu of BACKTRACK, it first initializes the working reason, and then it pops records from Γ until either the stack is empty, or some open branch is found – the return values in these cases are the same as BACKTRACK. BACKJUMP stops at open branches (LS) only if they correspond to some literal l

```
STRUQS[BJ,VE](φ, Γ)
  stop ← FALSE
  while ¬stop do
    PROPAGATE(φ, Γ)
    if ISTRUE(φ) then
      result ← TRUE
      stop ← BACKJUMP(φ, Γ, ∀)
    else if ISFALSE(φ) then
      result ← FALSE
      stop ← BACKJUMP(φ, Γ, ∃)
    else
      if PREFERSEARCH(φ) then SEARCH(φ, Γ)
      else ELIMINATE(φ, Γ)
  return result
```

Figure 2: Pseudo-code of STRUQS[BJ,VE].

which is bound by the quantifier q and such that $|l|$ occurs in the current working reason wr (OCCURS($|l|$, wr)=TRUE). Also, when popping records $\langle l, \mathrm{FL}, r\rangle$ such that l occurs in the working reason, the procedure UPDATEREASON is invoked to update the current working reason wr. As described in Section 2, all the literals bound by q that are not in the current working reason are irrelevant to the current conflict/solution and are simply popped from Γ without further ado.

On top of the basic algorithms of Figure 1, we define five different versions of STRUQS: Three "single" versions, namely STRUQS[BJ], STRUQS[BT], and STRUQS[VE], and two "blended" ones, namely STRUQS[BT,VE] and STRUQS[BJ,VE]. The single solvers implement basic reasoning algorithms and are provided for reference purposes, while the blended ones implement our structural approach.[3] The pseudo-code of STRUQS[BJ,VE] is detailed in Figure 2 where we consider some additional primitives on φ: ISTRUE(φ) returns TRUE exactly when all clauses in φ have been disabled; ISFALSE(φ) returns TRUE exactly when there is at least one contradictory clause in φ; finally, PREFERSEARCH(φ) is some heuristic to choose between search and variable elimination. The other versions of STRUQS can be obtained from the code in Figure 2 as follows. If PREFERSEARCH always returns TRUE, then we obtain the search solver STRUQS[BJ], and if it always returns FALSE, then we obtain the variable elimination solver STRUQS[VE]. Replacing BACKJUMP with BACKTRACK, we obtain STRUQS[BT,VE], and if we further assume that PREFERSEARCH always returns TRUE, then we obtain STRUQS[BT].

We now consider the implementation of PREFERSEARCH for the blended versions of STRUQS. As discussed in Section 1, PREFERSEARCH needs to be based on parameters which are informative, yet computable without a prohibitive overhead. To this end, in PREFERSEARCH we resort to the following criteria:

1. Given the current φ and Γ, check if there is any existential variable x which respects CHOOSEVAR's constraints, and such that the number of clauses potentially generated by DOELIMINATE(φ, x) is less than the number of clauses

[3] The source code of STRUQS is available at `http://www.mind-lab.it/projects`.

containing x in φ.

2. If condition (1) is met for some x, then consider the number of clauses where x occurs, say n_x, and the number of clauses where $\neg x$ occurs, say $n_{\neg x}$, and compute the *diversity* of x as the product $n_x n_{\neg x}$; check if the diversity of x is less than a predefined threshold div.

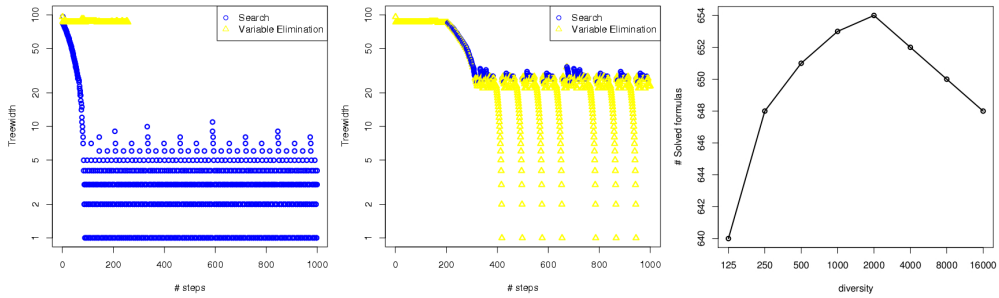
Notice that, when checking for conditions (1) we compute an upper bound on the number of clauses generated as $n_x n_{\neg x}$, and we stop checking when the first variable respecting both conditions is found. If both conditions above are met for some x, then PREFERSEARCH returns FALSE, and CHOOSEVAR is instructed to return x. If either condition is not met, then PREFERSEARCH returns TRUE. In this case, CHOOSELIT tries to pick a literal l which respects CHOOSELIT's constraints, and such that $|l|$ is a node in the neighborhood of some variable x tested during the last call to PREFERSEARCH. If no such literal can be found, then CHOOSELIT returns a literal l such that $|l|$ has the maximum degree among all the candidate literals of φ.

We mention here that condition (1) is proposed in [6] to control variable elimination during preprocessing of plain Boolean encodings, and diversity is introduced in [16] to efficiently account for structure of Boolean formulas. Intuitively, in our implementation condition (1) provides the main feedback that implicitly considers the treewidth of the current formula: If variable elimination becomes less efficient, then this means that the structure of the current QBF is probably less amenable to this kind of approach. Condition (2) fine tunes (1): Whenever some variable x meets condition (1), but $n_x n_{\neg x}$ is too large, eliminating x could be overwhelmingly slow. Finally, the search heuristic in CHOOSELIT is modified in order to use search for breaking large clusters of tightly connected variables. If this cannot be done, then CHOOSELIT falls back to assigning the variable which is most constrained.

4 Putting the structural approach to the test

All the experimental results that we present are obtained on a family of identical Linux workstations comprised of 10 Intel Core 2 Duo 2.13 GHz PCs with 4GB of RAM. The resources granted to the solvers are always 600s of CPU time and 3GB of memory. To compare the solvers we use the QBFEVAL'08 dataset, which is comprised of 3326 formulas divided into 17 suites and 71 families (suites may be comprised of more than one family). All the formulas considered are QBF encodings of some automated reasoning task [14]. In the following, we say that solver A *dominates* solver B whenever the set of problems solved by A is a superset of the problems solved by B; we say that two solvers are *incomparable* when neither one dominates the other.

Our first experiment aims to provide an empirical explanation on why alternating between search and resolution can be effective. In Figure 3 (a,b) we present two



(a) (b) (c)

Figure 3: **(a)** Number of reasoning steps (x axis) vs. approximated treewidth value (y axis) in StruQS[VE] (triangles) and StruQS[BJ] (dots). **(b)** As (a), but describing the alternation of search (dots) and variable elimination (triangles) steps inside StruQS[BJ,VE]. The number of reasoning steps corresponds to the number of LS- and VE-type records pushed on Γ. In particular, we consider the sequence of QBFs $\varphi_1, \varphi_2, \ldots$ where φ_1 is the input QBF, and φ_{i+1} is obtained from φ_i as follows. If φ_i does not correspond to a conflict or a solution, then φ_{i+1} is the result of a destructive update of φ_i corresponding to (i) unit propagation and (ii) variable elimination or search. If φ_i is a conflict or a solution, then φ_{i+1} is the formula corresponding to the node where StruQS backtracks to. For all $\varphi_1, \varphi_2, \ldots$, approximated treewidth $\hat{tw}_p(\varphi_i)$ is such that $tw_p(\varphi_i) \leq \hat{tw}_p(\varphi_i)$. **(c)** Number of formulas solved (y axis) vs. div parameter value (x axis) in StruQS[BJ,VE].

plots showing the connection between the behaviour of StruQS – as dictated by PreferSearch – and the treewidth[4] of the input formula as it is modified during the solution process. The plots have been obtained by running various versions of StruQS on the instance `test5_quant5` – family `jmc_quant`, suite `Katz` in the QBFEVAL'08 dataset. We consider `test5_quant5` because it is the smallest instance (358 variables, 941 clauses, 3 quantifier blocks) which is solved by StruQS[BJ,VE], and it is not solved by StruQS[BJ] and StruQS[VE]. Looking at Figure 3 (a), we can see that StruQS[VE] (triangle dots) is able to decrease slightly treewidth (from an initial estimate of 96 to a final estimate of 86), but it can go as far as a few hundred steps before exhausting resources. On the other hand, StruQS[BJ] (circle dots) can decrease substantially treewidth, but it gets trapped into structurally simpler subproblems, without managing to solve the input QBF after about 9×10^6 steps, when it reaches the time limit.[5] From Figure 3 (b) we can see that StruQS[BJ,VE] changes its behavior according to the current value of treewidth, i.e., PreferSearch implicitly takes into account structure. In the first 206 steps, only variable elimination is performed (triangle dots). Then, as this approach becomes untenable, search (circle dots) is exploited to obtain structurally simpler subproblems. Comparing the two plots, it is clear that PreferSearch prevents search to go "too deep", because as soon as $\hat{tw}_p(\varphi_i)$ becomes small, PreferSearch reckons that variable elimination alone can deal with φ_i. Overall, StruQS[BJ,VE] solves `test5_quant5` in less than 4×10^5

[4] More precisely, we consider an *approximation* of treewidth, computed as described in [15].

[5] Only the first 10^3 steps are shown in plots of Figure 3 (a,b). The hidden portions of StruQS[BJ] and StruQS[BJ,VE] behave as they do in the last few hundred steps shown.

Solver	Performance			
	#	Time	#LS ($\times 10^5$)	#VE ($\times 10^5$)
STRUQS[BJ,VE]	1304	63141	656	1407
STRUQS[BT,VE]	1019	41900	1744	1731
STRUQS[BJ]	614	31543	11478	–
STRUQS[VE]	528	12834	–	3
STRUQS[BT]	380	10855	35594	–

Table 1: Comparison of STRUQS versions considering the number of problems solved ("#"), the cumulative CPU seconds spent ("Time"), the cumulative number of search ("#LS") and variable elimination ("#VE") steps performed on solved problems.

steps, of which about 25% are search steps. Another interesting phenomenon is that even when search is prevailing – for $\hat{tw}_p(\varphi)$ values roughly in between 75 and 25 – PREFERSEARCH still gives a chance to variable elimination. In these cases, variable elimination plays essentially the same role of unit propagation, i.e., it stops in advance the exploration of irrelevant portions of the search space.

In the above experiment, PREFERSEARCH turns out to be effective in switching between variable elimination and search with a specific value of the threshold div (2000). Since different values of div influence the behaviour of PREFERSEARCH, it is reasonable to ask how much a particular setting of div will influence the performance of STRUQS, if at all, and which is the best setting, at least in the dataset that we consider. In figure 3 (c) we present the result of running STRUQS[BJ,VE] with various settings of the parameter div on the formulas in the QBFEVAL'08 dataset which cannot be solved by either STRUQS[VE] or STRUQS[BJ] alone (660 formulas). In particular, we consider an initial value $div = 125$ and we increase it by a $\times 2$ factor until $div = 16000$, each time running STRUQS[BJ,VE] on the above dataset. The plot in Figure 3 clearly shows that (i) $div \approx 2000$ appears to be the best setting in the QBFEVAL'08 dataset, and that (ii) only 15 instances separate the performance of the best STRUQS[BJ,VE] setting ($div = 2000$) with respect to the worst one ($div = 125$). In all the experiments that we show hereafter div is fixed to 2000, but we conjecture that while tuning div helps in making STRUQS faster in some cases, relatively small changes in the number of problems solved are to be expected.

As we anticipated, there is a number of instances that can be solved by STRUQS[BJ,VE], but cannot be solved by the corresponding single solvers. Our next experiment is to compare all STRUQS versions to see whether blended solvers have an edge over single ones. Indeed, a quick glance at the results shown in Table 1 reveals that blended solvers are *always* superior to single ones. In more detail, STRUQS[BT] can solve 11% of the dataset, STRUQS[VE] can deal with 16% of it, and STRUQS[BJ] tops at 18%. Only 273 problems (8% of the dataset) are solved by both STRUQS[BT] and STRUQS[VE], which are incomparable. On the other hand, STRUQS[BJ] dominates STRUQS[BT] and it is incomparable with STRUQS[VE]. As for the blended versions, we see that STRUQS[BT,VE] is more efficient than all the single versions: It can solve 31% of the dataset, which is more than the set of problems solved by at least one of STRUQS[BT] and STRUQS[VE] (635). STRUQS[BJ,VE]

turns out to be the most efficient among STRUQS versions: It can solve 39% of the dataset, which is also more than the set of problems solved by at least one of STRUQS[BJ] and STRUQS[VE] (800). If we normalize the number of search steps performed by the solvers considering the problems that are solved, we see a $10\times$ gap between STRUQS[BT,VE] and STRUQS[BJ], while the same gap between STRUQS[BJ] and STRUQS[BT] is only $5\times$. This means that combining search and variable elimination turns to be efficient per se, and even more efficient than boosting search with backjumping. If we look at variable elimination steps, we can see a substantial increase in the normalized number of steps: nearly $300\times$ (resp. $200\times$) gap between STRUQS[VE] and STRUQS[BT,VE] (resp. STRUQS[BJ,VE]). Indeed, variable elimination is used more in STRUQS[BT,VE] and STRUQS[BJ,VE] than in STRUQS[VE], but this happens on subproblems that are structurally simpler, as there is only a $1.7\times$ (resp. $2\times$) gap in normalized CPU time between STRUQS[VE] and STRUQS[BT,VE] (resp. STRUQS[BJ,VE]). Notice also that STRUQS[BJ,VE] performs less search and variable elimination steps than STRUQS[BT,VE], and this appears to be the net effect of backjumping.

One last set of considerations about Table 1 regards dominance relationships. In particular, STRUQS[BT,VE] is incomparable with both STRUQS[BT] and STRUQS[VE]. In the case of STRUQS[BT], there are 6 problems in the suite `Ansotegui` which can be solved by STRUQS[BT], but not by STRUQS[BT,VE]. Indeed, the size of these problems is such that variable elimination becomes prohibitive too quickly for PREFERSEARCH to notice. STRUQS[VE] can solve 162 problems that STRUQS[BT,VE] cannot solve, where the bulk of such problems (121) is from the suite `Pan`: in these formulas, condition (1) of PREFERSEARCH is almost always unmet, and yet they turn out to be very difficult for search. Relaxing condition (1) of PREFERSEARCH can have positive effects on these formulas, but it can also severely degrade the performance of STRUQS[BT,VE] on other suites. STRUQS[BJ,VE] dominates STRUQS[BT,VE] and STRUQS[BT], but is incomparable with STRUQS[BJ] and STRUQS[VE]. In particular, there are 52 problems solved by STRUQS[BJ] that STRUQS[BJ,VE] cannot solve, whereof 40 are from the family `blackbox-01X-QBF`. These instances are characterized by the largest number of quantifier blocks among formulas in the dataset, and search is already quite efficient on them. STRUQS[VE] can solve 68 formulas on which STRUQS[BJ,VE] fails, of which 67 are from the suite `Pan` and have the same peculiarities mentioned about STRUQS[BT,VE].

In Table 2 we conclude this section by comparing STRUQS[BJ,VE] with state-of-the-art QBF solvers. We have considered all the competitors of QBFEVAL'08, namely AQME, QUBE6.1, NENOFEX, QUANTOR, and SSOLVE. To them, we have added the solvers SKIZZO [1] which is representative of skolemization-based approaches, QUBE3.0 [7] which is the same as QUBE6.1 but without preprocessing, YQUAFFLE [18] and 2CLSQ [17] which are also search-based solvers, and QMRES [12] which features symbolic variable elimination. Although STRUQS[BJ,VE] is just a proof-of-concept solver, Table 2 reveals that it would have ranked third best considering only the participants to QBFEVAL'08, and it ranks as fourth best in Table 2. Moreover, there are 24 instances that can be solved only by

Solver	Solved		True		False	
	#	Time	#	Time	#	Time
AQME	2434	43987	977	19747	1457	24240
QUBE6.1	2144	32414	828	18248	1316	14166
SKIZZO	1887	40864	631	17550	1256	23314
STRUQS[BJ,VE]	1304	63141	576	32685	728	30456
QUBE3.0	1077	16700	406	6536	671	10164
NENOFEX	985	22360	459	13853	526	8507
QUANTOR	972	15718	485	10418	487	5300
SSOLVE	965	23059	450	9866	515	13193
YQUAFFLE	948	16708	389	9058	559	7650
2CLSQ	780	21287	391	13234	389	8053
QMRES	704	13576	360	7722	344	5853

Table 2: STRUQS[BJ,VE] vs. state-of-the-art QBF solvers considering the number of problems solved and the cumulative CPU seconds considering overall results ("Solved"), as well as "True" and "False" formulas separately.

STRUQS[BJ,VE]. If we consider that AQME is a multi-engine solver combining the ability of several engines (including QUBE6.1 and SKIZZO), and that both QUBE6.1 and SKIZZO are fairly sophisticated pieces of software, we can conclude that STRUQS[BJ,VE] performances are very satisfactory. Indeed, STRUQS is somewhat limited by the straigthforward implementation of operations such as, e.g., .forward subsumption. On one hand, disabling forward subsumption brings to 1114 the number of problems solved by STRUQS[BJ,VE]. On the other hand, forward subsumption accounts for 8% of the total time spent by STRUQS[BJ,VE] on the formulas that it can solve, but for 20% of the total time on the formulas that it cannot solve. Increasing the time limit to 1200 seconds allows STRUQS[BJ,VE] to solve about one hundred additional problems, indicating that the efficiency of basic operations is currently a limiting factor for STRUQS.

5 Conclusions and future work

Summing up, the combination of search and resolution featured by STRUQS seems to offer a gateway to effective reasoning in QBFs. The two key results of our work are (i) showing that the blended solvers STRUQS[BT,VE] and STRUQS[BJ,VE] outperform the single ones, and (ii) showing that STRUQS[BJ,VE] is competitive with other state-of-the-art solvers. We believe that these results can be improved by pursuing an efficient implementation of STRUQS. The fact that STRUQS, however unsophisticated, is able to solve 24 problems that cannot be solved by other state-of-the-art QBF solvers is promising in this sense. Also, STRUQS[BJ,VE] used as a SAT solver on a subset of 790 QBFEVAL'08 formulas can solve 668 problems within one hour of CPU time and it is competitive with, e.g., MINISAT which can solve 787 of them, while STRUQS[BJ] and STRUQS[VE] top at 553 and 145 problems solved, respectively. Other improvements would be leveraging different structural features and approximations thereof, and/or integrating inductive techniques to synthesize (sub)instance-based PREFERSEARCH policies.

References

[1] M. Benedetti. sKizzo: a Suite to Evaluate and Certify QBFs. In *20th Int.l. Conf. on Automated Deduction*, volume 3632 of *LNCS*, pages 369–376, 2005.

[2] A. Biere. Resolve and Expand. In *7th Intl. Conf. on Theory and Applications of Satisfiability Testing*, volume 3542 of *LNCS*, pages 59–70, 2004.

[3] H. Bodlaender. A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth. *SIAM J. of Computation*, 25(6):1305–1317, 1996.

[4] M. Cadoli, A. Giovanardi, and M. Schaerf. An algorithm to evaluate quantified boolean formulae. In *15th Nat.l Conf. on Artificial Intelligence*, pages 262–267, 1998.

[5] H. Chen and V. Dalmau. From Pebble Games to Tractability: An Ambidextrous Consistency Algorithm for Quantified Constraint Satisfaction. In *In proc. of 19th Int.l workshop on Computer Science Logic*, volume 3634 of *LNCS*, pages 232–247, 2005.

[6] N. Eén and A. Biere. Effective Preprocessing in SAT through Variable and Clause Elimination. In *8th Intl. Conf. on Theory and Applications of Satisfiability Testing*, volume 3569 of *LNCS*, pages 61–75, 2005.

[7] E. Giunchiglia, M. Narizzano, and A. Tacchella. Clause-Term Resolution and Learning in Quantified Boolean Logic Satisfiability. *Artificial Intelligence Research*, 26:371–416, 2006.

[8] G. Gottlob, G. Greco, and F. Scarcello. The Complexity of Quantified Constraint Satisfaction Problems under Structural Restrictions. In *19th Int.l Joint Conference on Artificial Intelligence*, pages 150–155, 2005.

[9] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124:243–282, 2000.

[10] H. Kleine-Büning, M. Karpinski, and A. Flögel. Resolution for Quantified Boolean Formulas. *Information and Computation*, 117(1):12–18, 1995.

[11] J. Larrosa and R. Dechter. Boosting Search with Variable Elimination in Constraint Optimization and Constraint Satisfaction Problems. *Constraints*, 8(3):303–326, 2003.

[12] G. Pan and M. Vardi. Symbolic Decision Procedures for QBF. In *10th Int.l Conf. on Principles and Practice of Constraint Programming*, pages 453–467, 2004.

[13] G. Pan and M. Vardi. Fixed-Parameter Hierarchies inside PSPACE. In *21st IEEE Symposium on Logic in Computer Science*, pages 27–36, 2006.

[14] C. Peschiera, L. Pulina, and A. Taccchella. Sixth QBF solvers evaluation (QBFEVAL), 2008. http://www.qbfeval.org/2008.

[15] L. Pulina and A. Tacchella. Treewidth: a useful marker of empirical hardness in quantified Boolean logic encodings. In *15th Int.l Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 5330 of *LNCS*, 2008.

[16] I. Rish and R. Dechter. Resolution versus search: Two strategies for sat. *Journal of Automated Reasoning*, 24(1/2):225–275, 2000.

[17] H. Samulowitz and F. Bacchus. Binary Clause Reasoning in QBF. In *In proc. of Ninth Int.l Conference on Theory and Applications of Satisfiability Testing*, volume 4121 of *LNCS*, pages 353–367, 2006.

[18] Y. Yu and S. Malik. Verifying the Correctness of Quantified Boolean Formula(QBF) Solvers: Theory and Practice. In *Conference on Asia South-Pacific Design Automation*, pages 1047–1051, 2005.