Expansion-based QBF Solving on Tree Decompositions*

Günther Charwat and Stefan Woltran

Institut für Informationssysteme, TU Wien
Favoritenstraße 9-11, 1040 Vienna, Austria
{gcharwat, woltran}@dbai.tuwien.ac.at

Abstract. In recent years, various approaches for quantified Boolean formula (QBF) solving have been developed, including methods based on expansion, skolemization and search. Here, we present a novel expansion-based solving technique that is motivated by concepts from the area of parameterized complexity. Our approach relies on dynamic programming over the tree decomposition of QBFs in prenex conjunctive normal form (PCNF). Hereby, BDDs are used for compactly storing partial solutions. Towards efficiency in practice, we integrate dependency schemes and dedicated heuristic strategies. Our experimental evaluation reveals that our implementation is competitive to state-of-the-art solvers on instances with one quantifier alternation. Furthermore, it performs particularly well on instances up to a treewidth of approximately 80. Results indicate that our approach is orthogonal to existing techniques, with a large number of uniquely solved instances.

1 Introduction

Quantified Boolean formulae (QBFs) extend propositional logic by explicit universal and existential quantification over variables. They can be used to compactly encode many computationally hard problems, which makes them amenable to application fields where highly complex tasks emerge, e.g. formal verification, synthesis, and planning. In this work we consider the problem of deciding satisfiability of QBFs (QSAT) which is, in general, PSPACE-complete [33]. We present an approach that is motivated by results from the area of parameterized complexity: many computationally hard problems are *fixed-parameter tractable* (fpt) [14], i.e., they can be solved in time $f(p) \cdot n^{\mathcal{O}(1)}$ where n is the input size, p the parameter, and f a computable function. It is known that QSAT is fpt w.r.t. the combined parameter *quantifier alternations plus treewidth* of the QBF instance (this follows from [12]), but not w.r.t. treewidth alone [4].

Intuitively, treewidth captures the "tree-likeness" of a graph. It emerged from the observation that computationally hard problems are usually easier to be solved on trees than they are on arbitrary graphs. Treewidth is defined on tree

* This work was supported by the Austrian Science Fund (FWF): Y698. Rudimentary ideas were presented at the QBF'16 workshop [11].

decompositions (TDs). Our approach employs dynamic programming (DP) over the TD of the primal graph of QBFs in prenex conjunctive normal form (PCNF). Partial solutions of the DP are obtained via locally restricted expansion. The practical feasibility of this approach rests on the following pillars. First, we make use of *binary decision diagrams* (BDDs) [10] for compactly storing information in our dedicated data structure. Second, we consider structure in the quantifier prefix by integrating dependency schemes (see, e.g., [31]) into our DP algorithm. Finally, we introduce optimization techniques such as dynamic variable removal and TD selection based on characteristics beyond treewidth. By design, our novel approach is expected suitable for QBF instances of low-to-medium treewidth and a restricted number of quantifier alternations.

The concept most closely related to ours was developed by Pan and Vardi [24]. There, variables are eliminated based on an elimination ordering that also underlies the construction of the tree decomposition in our approach. However, their approach requires the variables to be eliminated from the inner- to the outermost quantifier level, a restriction that we circumvent in this work. Treewidth and its relation to (empirical) hardness of QBFs was studied, for instance, by Pulina and Tacchella [27] and Marin *etal.* [23]. There, quantified treewidth is considered, a generalization of primal treewidth that includes the variable ordering as specified in the QBF prefix. Additionally, there exists a QBF solver that uses treewidth to dynamically decide between resolution and search during the solving process [26]. Further approaches that consider structural aspects of QBFs include search-based solving based on dependencies between variables [22], decomposition of the QBF according to the quantifier level of variables [28], and restoring structure in instances converted to PCNF [6].

The presented algorithms are implemented in the QBF solver *dynQBF* which is freely available at `https://dbai.tuwien.ac.at/proj/decodyn/dynqbf/`. We conduct an experimental evaluation along the lines of the QBF competition [25] in 2016. In comparison with state-of-the-art solvers, results show that our approach is particularly competitive on instances with one quantifier alternation. Furthermore, the implementation performs well on instances that exhibit a width of up to 80, even for instances with more quantifier alternations. Additionally, we observe a large number of instances that is uniquely solved by dynQBF. These observations underline the practical potential of parameterized algorithms in highly competitive domains and we believe that the techniques used in our system (space efficient storage via BDDs, TD selection, etc.) will also prove useful when efficient dynamic programming algorithms for other problems are to be implemented.

2 Preliminaries

As usual, a *literal* is a variable or its negation. A *clause* is a disjunction of literals. A Boolean formula in conjunctive normal form (CNF) is a conjunction of clauses. We sometimes denote clauses as sets of literals, and a formula in CNF

as a set of clauses. Herein, we consider *quantified Boolean formulae* (QBFs) in closed prenex CNF (PCNF) form.

Definition 1. *In a* PCNF QBF $Q.\psi$*,* Q *is the* quantifier prefix *and* ψ *is a CNF formula, also called the* matrix*.* Q *is of the form* $Q_1X_1Q_2X_2 \dots Q_kX_k$ *where* $Q_i \in \{\exists, \forall\}$ *for* $1 \leq i \leq k$*,* $Q_i \neq Q_{i+1}$ *for* $1 \leq i < k$*, and* $X = \{X_1, \dots, X_k\}$ *is a partition over all variables in* ψ*. For a variable* $x \in X_l$ *(*$1 \leq l \leq k$*),* l *is the* level *of* x*, and* $k - l + 1$ *the* depth *of* x*.*

We frequently use the following notation: Given a QBF $Q.\psi$ with $Q = Q_1X_1 \dots Q_kX_k$ and an index i with $1 \leq i \leq k$, $\mathit{quantifier}_Q(i) = Q_i$ gives the i-th quantifier. For a variable x, $\mathit{level}_Q(x)$ returns the level of x; $\mathit{depth}_Q(x)$ returns the depth of x in Q; and $\mathit{quantifier}_Q(x) = Q_{\mathit{level}_Q(x)}$ returns the quantifier for x. Finally, for a clause $c \in \psi$, we denote by $\mathit{variables}_\psi(c)$ the variables occurring in c. We will usually omit the subscripts whenever no ambiguity arises.

Example 1. As our running example, we will consider the QBF $Q.\psi$ with $Q = \exists ab\, \forall cd\, \exists ef$ and $\psi = (a \vee c \vee e) \wedge (\neg a \vee b) \wedge (\neg b \vee f) \wedge (d \vee \neg e)$, which is satisfiable.

A *tree decomposition* (TD) [29] is a mapping from a graph to a tree, where each node in the TD can contain several vertices of the original graph.

Definition 2. *A* tree decomposition *of a graph* $G = (V, E)$ *is a pair* $\mathcal{T} = (T, \mathit{bag}_\mathcal{T})$ *where* $T = (N, F)$ *is a (rooted) tree with nodes* N *and edges* F*, and* $\mathit{bag}_\mathcal{T} : N \to 2^V$ *assigns to each node a set of vertices, such that:*

1. *For every vertex* $v \in V$*, there exists a node* $n \in N$ *such that* $v \in \mathit{bag}_\mathcal{T}(n)$*.*
2. *For every edge* $e \in E$*, there exists a node* $n \in N$ *such that* $e \subseteq \mathit{bag}_\mathcal{T}(n)$*.*
3. *For every vertex* $v \in V$*, the subtree of* T *induced by* $\{n \in N \mid v \in \mathit{bag}_\mathcal{T}(n)\}$ *is connected.*

Intuitively, Condition 1 and 2 guarantee that the whole graph is covered by the TD, and Condition 3 is the *connectedness property*, which, roughly speaking, states that a vertex cannot "reappear" in unconnected parts (w.r.t. the bags). The *width* of $\mathcal{T}$ is defined as $\max_{n \in N} |\mathit{bag}_\mathcal{T}(n)| - 1$. The *treewidth* t of a graph is the minimum width over all its TDs. Given a graph and an integer t, deciding whether the graph has at most treewidth t is NP-complete [3]. However, the problem itself is fpt when t is considered as parameter. Additionally, there exist good polynomial-time heuristics for constructing TDs [8, 13].

A TD $\mathcal{T} = ((N, F), \mathit{bag}_\mathcal{T})$ is *weakly normalized*, if each $n \in N$ is either a *leaf* node (n has no children), an *exchange* node (n has exactly one child n_1, such that $\mathit{bag}_\mathcal{T}(n) \neq \mathit{bag}_\mathcal{T}(n_1)$), or a *join* ($n$ has children $n_1, \dots, n_m$ such that $m \geq 2$, and $\mathit{bag}_\mathcal{T}(n) = \mathit{bag}_\mathcal{T}(n_1) = \dots = \mathit{bag}_\mathcal{T}(n_m)$). Given a TD $\mathcal{T} = (T, \mathit{bag}_\mathcal{T})$ with $T = (N, F)$, for a node $n \in N$ we denote its set of children in T by $\mathit{children}_\mathcal{T}(n)$. We specify $\mathit{firstChild}_\mathcal{T}(n)$ and $\mathit{nextChild}_\mathcal{T}(n)$ to iterate over the children, and $\mathit{hasNextChild}_\mathcal{T}(n)$ to check whether further children exist. The node type is checked with $\mathit{isLeaf}_\mathcal{T}(n)$, $\mathit{isExchange}_\mathcal{T}(n)$ and $\mathit{isJoin}_\mathcal{T}(n)$. For a node n with single child node n_1, changed bag contents are accessed by $\mathit{introduced}_\mathcal{T}(n) =$

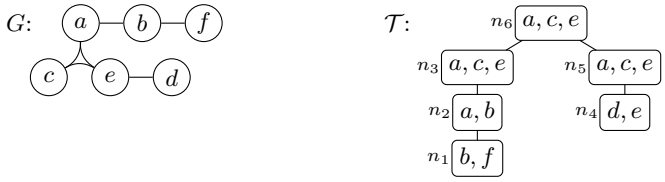

Fig. 1. Primal graph G and a possible TD $\mathcal{T}$ of G for the QBF in Example 1.

$bag_\mathcal{T}(n) \setminus bag_\mathcal{T}(n_1)$ and $removed_\mathcal{T}(n) = bag_\mathcal{T}(n_1) \setminus bag_\mathcal{T}(n)$. $isRoot_\mathcal{T}(n)$ returns true if n has no parent node.

Our algorithm for QBF solving is based on a TD of the given QBF $Q.\psi$, which is obtained from the graph $G_\psi = (V, E)$ where V are the variables occurring in ψ and each clause in ψ forms a clique in G_ψ, i.e. $E = \{(x, y) \mid x, y \in variables_\psi(c), c \in \psi, x \neq y\}$ (called *primal* or *Gaifman* graph). Given a TD $\mathcal{T} = (T, bag_\mathcal{T})$ of QBF $Q.\psi$, we define $clauses_{\mathcal{T},\psi}(n) = \{c \in \psi \mid variables_\psi(c) \subseteq bag_\mathcal{T}(n)\}$.

Example 2. Consider our running example. Figure 1 illustrates the graph representation G of ψ, and $\mathcal{T}$ is a weakly-normalized TD for G of width 2.

3 Dynamic Programming-based QBF Solving

In a nutshell, the algorithm proceeds as follows. Given a QBF instance $Q.\psi$, we heuristically construct a weakly normalized TD $\mathcal{T} = (T, bag_\mathcal{T})$ with $T = (N, F)$ of the primal graph of ψ. Then, $\mathcal{T}$ is traversed in post-order. For each $n \in N$ we compute *partial solution candidates* and store them in a dedicated data structure. In this context, *partial* means that the data structure is restricted to variables occurring in $bag_\mathcal{T}(n)$. *Candidate* refers to the fact that other parts of the QBF might not yet be considered. At the root node, the whole instance was taken into account and the problem is decided.

3.1 Data structure

We define so-called *nested sets of formulae* (NSFs) where the innermost sets contain *reduced ordered binary decision diagrams* (BDDs) [10]. A BDD compactly represents Boolean formulae in form of a rooted directed acyclic graph (DAG). For a fixed variable ordering, BDDs are canonical, i.e., equivalent formulae are represented by the same BDD, a property that is vital to our approach. Intuitively, nestings will be used to differentiate between quantifier blocks, and BDDs store parts of the QBF matrix.

Definition 3. *Given a QBF $Q.\psi$ with k quantifiers, we have a* nested set of formulae (NSF) *of depth k which is inductively defined over the depth of nestings d with $0 \leq d \leq k$: for $d = 0$, the NSF is a BDD; for $1 \leq d \leq k$, the NSF is a set of NSFs of depth $d - 1$.*

Fig. 2. Example NSF N, represented as tree, and $N[B/B \wedge c]$ applied to N.

For a QBF $Q.\psi$ with $Q = Q_1X_1 \dots Q_kX_k$ and an NSF N of depth k, for any NSF M appearing somewhere in N we denote by $\mathit{depth}(M)$ the depth of the nesting of M, $\mathit{level}_Q(M) = k - \mathit{depth}(M) + 1$ is the level of M, and $\mathit{quantifier}_Q(M) = Q_{\mathit{level}_Q(M)}$ (for $\mathit{level}_Q(M) \leq k$). We define the procedure $\mathit{init}(k, \phi)$ that initializes an NSF with k levels (and hence of depth k), such that each set contains exactly one NSF, and the innermost NSF represents ϕ. For instance, $\mathit{init}(3, \top)$ returns $\{\{\{\top\}\}\}$. Furthermore, for an NSF N we denote by $N[B/B']$ the replacement of each BDD B in N by B'. For a BDD B, *restriction* of a variable v is denoted by $B[v/\top]$ or $B[v/\bot]$. Quantification and standard logical operators are applied as usual.

Example 3. Suppose we are given an NSF $N = \{\{\{\top, \bot\}\}, \{\{\neg a \vee b\}, \{\bot\}, \{a \wedge b\}\}\}$. In the examples, we will illustrate nested sets as trees where leaves contain the formulae represented by the BDDs. Figure 2 shows the tree representing N together with the one resulting from $N[B/B \wedge c]$.

NSFs can be used to efficiently keep track of *parts* of the solution space (with respect to the TD), instead of representing the whole QBF instance at once. Internal elements of the NSF have quantifier semantics, as we will show later. Opposed to the similar concept of quantifier trees [5], NSFs are defined as recursive sets in order to automatically remove trivial redundancies. Furthermore, the depth is specified by the number of quantifiers, not by the number of variables. We remark that although CNFs of bounded treewidth can be stored entirely in a BDD of polynomial size, existential quantification can result in an exponential blowup [15]. Our NSFs mitigate this by only storing parts of the QBF's CNF in the BDDs.

3.2 Main Procedure

Algorithm 1 illustrates the recursive procedure for the post-order traversal of the TD and computing the partial solution candidates. It is called with the root node of the TD and returns an NSF that represents the overall solution. In leaf nodes, an NSF of k levels is initialized with the innermost set containing a BDD that represents the clauses associated with the current node. In an exchange node, variables are removed as well as introduced (w.r.t. the bag's contents). Removed variables are handled by "splitting" the NSF. Procedure $\mathit{split}(N, x)$ (see Algorithm 2) implements a variant of locally restricted expansion: at the level of x in N, each NSF M contained in N is replaced by two NSFs that distinguish between assignments of x to $\bot$ and $\top$. Observe that thereby any occurrence of x in the BDDs is removed. This guarantees that the size of each BDD

Algorithm 1: $solve(n)$

Input : A tree decomposition node n
Output: An NSF with partial solution candidates for n

```
1  if isLeaf(n) then N := init(k, clauses(n))
2  if isExchange(n) then
3  |  N := solve(firstChild(n))
4  |  for x ∈ removed(n) do N := split(N, x)
5  |  N := N[B/B ∧ clauses(n)]
6  if isJoin(n) then
7  |  N := solve(firstChild(n))
8  |  while hasNextChild(n) do
9  |  |  M := solve(nextChild(n))
10 |  |  N := join(N, M)
11 |  end
12 if isRoot(n) then N := evaluateQ(n, N)
13 return N
```

Algorithm 2: $split(N, x)$

Input : An NSF N and a variable x
Output: An NSF split at $level(x)$ w.r.t. assignments to x

```
if level(N) = level(x) then
 |  return {M[B/B[x/⊥]], M[B/B[x/⊤]] | M ∈ N}
else return {split(M, x) | M ∈ N}
```

Algorithm 3: $join(N_1, N_2)$

Input : NSFs N_1 and N_2 of same depth
Output: A joined NSF

```
if depth(N1) = 0 then return N1 ∧ N2
else return {join(M1, M2) | M1 ∈ N1, M2 ∈ N2}
```

is bounded by the bag's size. Furthermore, since (reduced ordered) BDDs are canonical and thanks to the set semantics of NSFs, the overall resulting NSF's size is bounded by the bag's size and depth (i.e., the number of quantifiers). Removal of variable x from the BDDs is admissible due to the connectedness property of the TD: x will never reappear somewhere upwards the TD, and therefore all clauses containing x were already considered. After splitting, the clauses associated with the current node are added to the NSF's BDDs via conjunction. In join nodes, NSFs computed in the child nodes are successively combined by procedure $join(N_1, N_2)$ (see Algorithm 3). The procedure guarantees that the structure (nesting) of the NSFs to be joined is preserved. BDDs in the NSFs are combined via conjunction, thus already considered information of both child nodes is retained.

Algorithm 4: $evaluateQ(n, N)$

Input : A tree decomposition node n and an NSF N
Output: A BDD B of N after evaluation of quantifiers

if $depth(N) = 0$ **then** $B := N$
else
  $X := \{x \mid x \in bag(n)\ and\ level(x) = level(N)\}$
  if $quantifier(N) = \exists$ **then**
    $B := \exists X \bigvee_{M \in N} evaluateQ(n, M)$
  else if $quantifier(N) = \forall$ **then**
    $B := \forall X \bigwedge_{M \in N} evaluateQ(n, M)$
return B

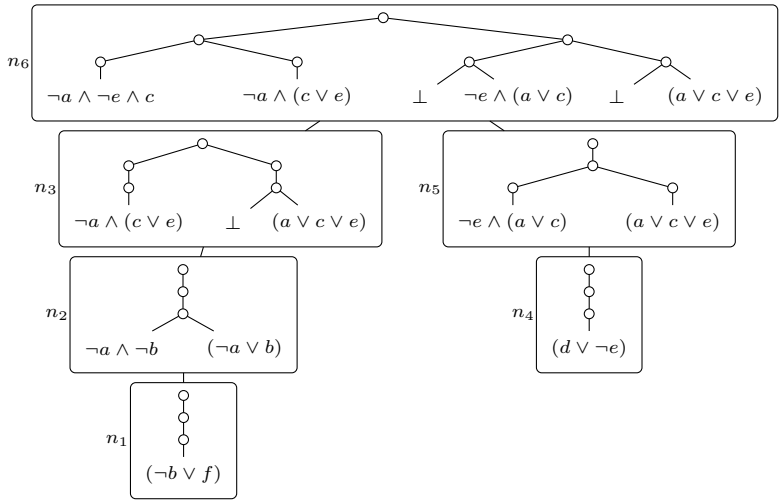

Fig. 3. Computed NSFs at the decomposition nodes of our running example.

So far, quantifiers were not taken into account in our algorithm. This is only done in the root node r of the TD, where the problem is decided by applying quantifier elimination as shown in Algorithm 4. Our approach is similar to that described by Pan and Vardi [24], but restricted to the bag contents and quantifiers are recursively evaluated over the nestings. Procedure $evaluateQ(r, N)$ combines the elements of the NSF by disjunction (for existential quantifiers) or conjunction (for universal quantifiers), starting at the innermost NSFs. Thereby, variables contained in the current bag are removed by *quantified abstraction* (i.e., they get existentially or universally quantified and thereby also removed from the BDDs). Thus, this procedure finally returns a single BDD B without variables. If $B \equiv \bot$, the QBF is unsatisfiable, otherwise it is satisfiable.

Example 4. Figure 3 shows the NSFs computed at the TD nodes of our running example (without quantifier evaluation at the root node). In n_1, an NSF of depth 3 is initialized with $(\neg b \vee f)$, i.e., the clause associated with this TD node. In n_2,

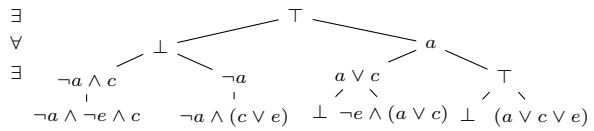

Fig. 4. (Intermediate) results for $evaluateQ(n_6, N)$ executed on the NSF of root node n_6.

variable f is removed. Hence the NSF is split at $level(f) = 3$, once by setting f to $\bot$ (left NSF branch) yielding $\neg b$, and once by $\top$ (right branch), yielding $\top$. Furthermore, the current clause $(\neg a \vee b)$ is added to these BDDs via conjunction, giving $\{\{\{\neg a \wedge \neg b, (\neg a \vee b)\}\}\}$. The algorithm proceeds similarly for nodes n_3, n_4 and n_5. In n_6, the NSFs are joined. For instance, the leftmost branches in n_3 and n_5 are joined by conjunction of $\neg a \wedge (c \vee e)$ and $\neg e \wedge (a \vee c)$, yielding $\neg a \wedge \neg e \wedge c$. Figure 4 shows the NSF N in root node n_6 together with the BDDs obtained recursively when applying $evaluateQ(n_6, N)$. The procedure returns $\top$, as the QBF from Example 1 is satisfiable.

3.3 Dependency Schemes

Quantifiers in QBFs introduce dependencies between variables. Let x and y be variables of the QBF, and assume that y is dependent on x. Then, the assignment to y is dependent on the assignment to x [30] (i.e., reordering x and y in the prefix changes satisfiability). So far, when a variable is removed splitting is applied to distinguish between variable assignments. With this, even if x is removed before y, we *implicitly* keep track of these assignments in our NSF data structure. Hence, when y is removed later, its dependency on x is accounted for, and our algorithm remains sound. However, if all variables dependent on x were already removed, the distinction between assignments is not necessary. We considered several dependency schemes (for details, see e.g., [31]). Let $Q.\psi$ be a PCNF QBF with k quantifiers and x, y be variables of $Q.\psi$. Then $(x, y) \in D^S_{Q.\psi}$ w.r.t. dependency scheme $S \in \{naive, simple, standard\}$ if:

1. naive: $level(x) < k$;
2. simple: $level(x) < level(y)$; and
3. standard: $level(x) < level(y), quantifier(x) \neq quantifier(y)$ and there is an X-path from x to y for some $X \subseteq \{z \mid z \in X_i, level(x) < i \leq k, quantifier(z) = \exists\}$. An X-path is a sequence $c_1, \ldots, c_l$ of clauses in ψ, s.t. $x \in c_1$, $y \in c_l$ and $c_j \cap c_{j+1} \cap X \neq \emptyset$ for $1 \leq j < l$ (see [21] for details).

$dependent^S_{Q.\psi}(x) = \{y \mid (x, y) \in D^S_{Q.\psi}\}$ denotes the set of variables that are dependent on x in $Q.\psi$ w.r.t. S.

Towards our adapted algorithm, for a TD node n of $\mathcal{T}$, we recursively define by $removedSub_{\mathcal{T}}(n) = removed_{\mathcal{T}}(n) \cup \bigcup_{m \in children_{\mathcal{T}}(n)} removedSub_{\mathcal{T}}(m)$ the set of removed variables in the subtree of $\mathcal{T}$ rooted at n. Let $removedBelow_{\mathcal{T}}(n) =$

Algorithm 5: S-*dependentSplit*(n, N, x)

Input : Tree decomposition node n, NSF N, variable x
Output: An NSF with abstracted or split x

if $\mathit{dependent}^S(x) \subseteq \mathit{removedBelow}(n)$ **then**
  if $\mathit{quantifier}(x) = \exists$ **then return** $N[B/\exists xB]$
  if $\mathit{quantifier}(x) = \forall$ **then return** $N[B/\forall xB]$
else return $\mathit{split}(N, x)$

$\mathit{removedSub}_{\mathcal{T}}(n) \setminus \mathit{removed}_{\mathcal{T}}(n)$ be the variables removed below n in $\mathcal{T}$. In Algorithm 1, $\mathit{split}(M, x)$ is replaced with S-*dependentSplit*(n, N, x) (see Algorithm 5). Whenever all variables dependent on x were already removed, x is removed by quantified abstraction. Otherwise, the standard $\mathit{split}(N, x)$ procedure is called.

Example 5. The NSF at node n_2 of Figure 3 reduces to $\{\{\{(\neg a \vee b)\}\}\}$ (for all dependency schemes). Furthermore, we have $D^{standard}_{Q.\psi} = \{(a, c), (a, d), (c, e), (d, e)\}$. Since $\mathit{dependent}(b) = \{\}$, b can be existentially abstracted in n_3. However, in n_5, d must be split, since $\mathit{dependent}(d) = \{e\} \not\subseteq \mathit{removedBelow}(n_5) = \{\}$.

We remark that for all considered dependency schemes variables at the innermost level can be removed by quantified abstraction. Hence our algorithms can be simplified, as the NSFs at depth 1 always only contain a single BDD. In particular, for 2-QBFs (i.e. instances of the form $\forall X_1 \exists X_2 . \psi$) the general NSF data structure could then be replaced by just a set of BDDs. Furthermore, we observed that in almost all 2-QBF instances (used in Section 5) variables at level 2 are dependent on those at level 1. For 2-QBFs, we thus apply the easily computable *naive* dependency scheme. For other instances *standard* turned out to be superior to *simple* and *naive*.

4 Towards Efficiency in Practice

Clause splitting. Given a QBF $Q.\psi$, we construct a TD of width w for the primal graph of ψ. Due to Conditions 2 and 3 of Definition 2, $w \geq \max_{c \in \psi} |c| - 1$ holds, i.e., the size of the largest clause gives a lower bound for w. To reduce this bound, we apply *clause splitting*, which is a standard technique implemented in many QBF solvers and preprocessors: a fresh variable is added (once positively, once negatively) to the parts of a split clause, and quantified existentially in the innermost quantifier block. Experiments preceding this work reveal that splitting clauses larger than 30 yields good results, without introducing too many additional variables.

TD selection. It was shown that TD characteristics besides width play a crucial role in practice [2]. In 2-QBF instances usually most computational effort is required for joining the NSFs. We consider the number of children in join nodes $\mathit{jNodes}(\mathcal{T})$ which is given as $\mathit{joinChildCount}(\mathcal{T}) = \sum_{j \in \mathit{jNodes}(\mathcal{T})} |\mathit{children}_{\mathcal{T}}(j)|$.

Algorithm 6: *removeRedundant*(N)

```
Input  : An NSF N
Output: An NSF without supersets
if depth(N) > 1 then
    for M ∈ N do  M := removeRedundant(M)
    for M1, M2 ∈ N and M1 ≠ M2 do
        if M1 ⊂ M2 then  N := N \ {M2}
    end
else
    for M1, M2 ∈ N and M1 ≠ M2 do
        if quantifier(N) = ∃ and M1 ∨ M2 = M1 then
            N := N \ {M2}
        if quantifier(N) = ∀ and M1 ∧ M2 = M1 then
            N := N \ {M2}
    end
return N
```

Additionally, we consider the following TD characteristic. Variable dependencies can be exploited more efficiently if the variables are removed in the TD from the innermost to the outermost quantifier block. Let $\mathit{removedBelowLevel}_{\mathcal{T},Q}(n,l) = \{b \mid b \in \mathit{removedBelow}_{\mathcal{T}}(n) \text{ and } \mathit{level}_Q(b) < l\}$. Now, $\mathit{removedLevel}(\mathcal{T},Q) = \sum_{n\in N}\sum_{r\in \mathit{removed}_{\mathcal{T}}(n)} |\mathit{removedBelowLevel}_{\mathcal{T},Q}(n,\mathit{level}(r))|$. We construct several TDs (using the min-fill heuristics [13]) and then select the one minimizing $\mathit{joinChildCount}(\mathcal{T})$ (for 2-QBFs) or $\mathit{removedLevel}(\mathcal{T},Q)$ (for instances with more quantifier blocks). We observe that 10 decompositions are sufficient to increase performance, despite the additional effort in the decomposition step.

Redundant NSF removal. Two BDDs in the same nesting of an NSF are redundant if they are in a subset relation w.r.t. the represented models (which is similar to subsumption checking [7]), or if two NSFs in the same nesting are in a subset relation. Algorithm 6 gives the pseudo-code for removing unnecessary elements[1]. Since the procedure includes a recursive comparison of all NSFs, checking for redundant NSFs is expensive. Nevertheless, periodic checks are required to circumvent an explosion in size in join nodes.

Example 6. Figure 5 shows an NSF N before and after $\mathit{removeRedundant}(N)$. For instance, consider the leftmost branch of the NSF at depth 1, i.e. $\{\bot, \neg a\}$. Since $\mathit{quantifier}(N_1) = \exists$ and $\bot \vee \neg a \equiv \neg a$, $\bot$ is removed. At depth 2, we subsequently have $\{\{\neg a\}, \{\neg a, c\}\}$. Since $\{\neg a\} \subseteq \{\neg a, c\}$, $\{\neg a, c\}$ is removed.

Intermediate unsatisfiability checks. Procedure $\mathit{evaluateQ}(n, N)$ can be applied to any NSF during the TD traversal. If it returns $\bot$, the QBF is unsatisfiable.

[1] When dependency schemes are considered, the NSFs at depth 1 contain only a single BDD. Then, subset checking w.r.t. models of the BDDs can be shifted by one level.

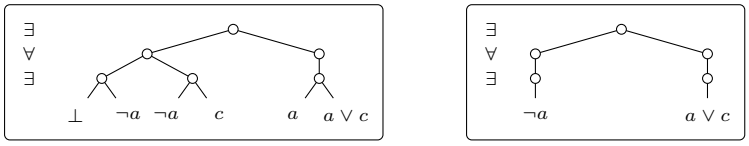

Fig. 5. Example NSF before (left) and after (right) compression.

However, if it returns $\top$, the QBF might still be unsatisfiable due to clauses that are encountered later in the traversal. In our setting, the overhead for these checks is negligible.

Estimated NSF size. For a node n of decomposition $\mathcal{T}$, let $sizeNSF(n)$ be the number of BDDs in the NSF N computed at node n, and $maxSizeBDD(n)$ be the size of the largest BDD in N. The size of a BDD is determined by the number of nodes in the DAG of the BDD. $sizeNSF(n)$ can be kept small by delaying splitting of removed variables. Instead, the variable is stored in a cache for later removal. However, this usually increases $maxSizeBDD(n)$ (since the variable is not removed from the BDDs), and the size of BDDs is no longer bounded by the bag size. Hence, NSF and BDD sizes have to be carefully balanced.

BDD variable ordering. The size of a BDD can be exponential in the number of variables. Nontheless, in practice the size may be exponentially smaller, in particular in case a "good" variable ordering is applied [16]. Since finding an optimal variable ordering is in general NP-hard [10], BDD-internal heuristics for finding such a good ordering can be used. For our purposes, we initialize the ordering with the variables' occurrence in the instance (which usually implies that the ordering corresponds to their occurrence in the QBF prefix), and apply dynamic reordering during the computation via lazy sifting.

5 Experimental Evaluation

The presented algorithms are implemented in the *dynQBF* system, which relies on HTD [1] for tree decomposition construction, CUDD [32] for BDD management, and optionally DepQBF [20] for computing the standard dependency scheme. We compare our system to publicly available QBF solvers that participated successfully in the 2016 QBF competition (QBFEval'16) [25]. Systems include the 2-QBF solver AReQS (20160702) [18], the search-based solvers DepQBF 5.0.1 [20] and GhostQ (CEGAR 2016) [17], the expansion-based system RAReQS 1.1 [17], CAQE 2 [28] that relies on variable level-based decomposition; as well as Questo 1.0 [19] and QSTS (2016) [9] that use SAT solvers. We consider the 305 2-QBF'16 and 825 PCNF'16 competition instances. Since preprocessing oftentimes influences performance, we additionally evaluate the solvers on the instances preprocessed with Bloqqer 37. Tests are performed on a single core of an Intel Xeon E5-2637 (3.5GHz) running Debian 8.3, with a time limit of 10 minutes and 16 GB of memory. For preprocessing we use the same configuration.

Table 1. 2-QBF'16: System comparison for the original (left) and preprocessed (right) instances.

2-QBF'16 (original)					
System	Solved	Time	⊤	⊥	U
AReQS	181	79K	126	55	0
dynQBF	170	86K	140	30	13
GhostQ	156	98K	108	48	0
DepQBF	120	116K	55	65	11
QSTS	97	132K	60	37	8
Qesto	78	140K	47	31	2
RAReQS	70	142K	44	26	0
CAQE	57	151K	35	22	1
dynQBF Bo10	203	68K	154	49	
dynQBF Ao10	169.9	86K	141.5	28.4	

2-QBF'16 (preprocessed)					
System	Solved	Time	⊤	⊥	U
Qesto	236	50K	160	76	0
RAReQS	232	51K	161	71	1
dynQBF	221	53K	172	49	43
DepQBF	221	56K	143	78	1
QSTS	220	58K	162	58	2
CAQE	204	65K	153	51	0
AReQS	202	66K	141	61	0
GhostQ	151	95K	123	28	0
dynQBF Bo10	225	49K	172	53	
dynQBF Ao10	221.2	53K	171.0	50.2	

Table 2. 2-QBF'16 (preprocessed, non-trivial): Influence of width w on the system performance.

$w \leq 80$ (86 instances)			$w > 80$ (89 instances)		
System	Solved	Time	System	Solved	Time
dynQBF	79	6K	RAReQS	69	17K
DepQBF	41	28K	QSTS	69	18K
Qesto	39	31K	Qesto	67	19K
RAReQS	33	34K	DepQBF	50	28K
CAQE	28	36K	AReQS	47	28K
AReQS	25	38K	CAQE	46	29K
QSTS	21	40K	**dynQBF**	12	47K
GhostQ	9	47K	GhostQ	12	49K

In the following, we report on the number of solved, solved satisfiable (⊤) and unsatisfiable (⊥) instances. The stated time is the accumulated user time in thousands of seconds (K), including a penalty of 600 seconds per instance that is not solved. Additionally, we give the number of instances uniquely solved by a single system (U). dynQBF is run with one random, fixed seed. However, the performance is influenced by the heuristically constructed TD. To gain an insight into the potential of our current implementation, we also provide a *virtual best dynQBF* analysis over 10 seeds (each running for up to 10 minutes). Best of 10 (*Bo10*) reports the number of instances solved in any of the 10 runs, as well as the minimum time required, and average of 10 (*Ao10*) reports the average case.

2-QBF'16. Table 1 shows that our system is competitive to state-of-the-art solvers on 2-QBF instances. On the original instances, only the 2-QBF solver AReQS performs better. When considering 10 different seeds, *Bo10* indicates that there is still potential for our feature-based tree decomposition selection. Regarding preprocessing, 130 out of 305 instances are directly solved by Bloqqer. Qesto and RAReQS benefit the most from preprocessing. Overall, dynQBF is particularly strong on satisfiable instances. Additionally, we report on a large

Table 3. PCNF'16: System comparison for the original (left) and preprocessed (right) instances.

Original					Preprocessed				
System	Solved	Time	⊤	⊥ U	System	Solved	Time	⊤	⊥ U
GhostQ	592	153K	300	292 14	RAReQS	633	126K	301	332 14
QSTS	548	173K	276	272 13	Qesto	618	134K	298	320 1
DepQBF	436	242K	188	248 14	DepQBF	596	144K	296	300 7
CAQE	399	268K	182	217 0	QSTS	592	149K	294	298 3
Qesto	368	287K	159	209 3	CAQE	589	155K	295	294 1
dynQBF	365	291K	184	181 14	GhostQ	571	161K	293	278 1
RAReQS	338	299K	129	209 8	**dynQBF**	494	203K	239	255 21
dynQBF Bo10	421	259K	212	209	dynQBF Bo10	515	193K	249	266
dynQBF Ao10	365.5	292K	184.6	180.9	dynQBF Ao10	494.8	202K	239.1	255.7

Table 4. PCNF'16 (preprocessed, non-trivial): Influence of width w on the system performance.

$w \leq 80$ (182 instances)			$w > 80$ (302 instances)		
System	Solved	Time	System	Solved	Time
RAReQS	137	28K	RAReQS	155	98K
dynQBF	134	32K	Qesto	148	100K
Qesto	129	34K	DepQBF	131	108K
DepQBF	124	36K	CAQE	129	114K
QSTS	123	37K	QSTS	128	112K
CAQE	119	40K	GhostQ	112	120K
GhostQ	118	41K	**dynQBF**	19	171K

number of uniquely solved instances. For the original data set, they mostly stem from QBF encodings for ranking functions ("`rankfunc*`"). Interestingly, after preprocessing we observe that 43 instances from the area of formal verification ("`stmt*`") are uniquely solved.

To study the influence of treewidth on solving, we consider the 175 preprocessed instances that are not solved directly by Bloqqer. Since computing the exact treewidth is infeasible, we use HTD [1] to heuristically obtain an over-approximation. In Table 2, the data set is partitioned based on the computed width w. Here, the influence of the width on the performance of dynQBF becomes apparent.

PCNF'16. Results for the PCNF'16 data set are summarized in Table 3. The obtained data confirms that dynQBF is indeed sensitive to the number of quantifier blocks (k). For the original instances we measure an average k of 17, and 14.8 for instances solved by dynQBF. 75 instances have 2 (or less) quantifier blocks, of which dynQBF solves the most instances (55). Of the 391 instances with $k = 3$, dynQBF solves 142 instances, while the best solver here is GhostQ with 299 instances. Of the 359 instances with $k > 3$, dynQBF solves 168 instances, but GhostQ solves 256 instances. With preprocessing, 341 instances are

solved by Bloqqer. Interestingly, all solvers except GhostQ benefit from preprocessing. Regarding the impact of quantifiers on the performance of dynQBF we obtain a similar picture as for the original instances. Overall, we again observe several instances uniquely solved by dynQBF.

As in the 2-QBF setting, we consider the width w of the preprocessed, non-trivial instances. Table 4 again shows that dynQBF performs well on instances where $w \leq 80$: here, k is 4.9 for all instances on average, and 3.7 for instances solved by dynQBF.

6 Conclusion

In this paper we introduced an alternative approach for QBF solving. Our algorithm is inspired by concepts from parameterized complexity, yielding a new expansion-based solver technique that mitigates space explosion by dynamic programming over the TD and by using BDDs. First ideas for dependency scheme integration were presented, and we discussed entry points for heuristic optimizations of our technique. We conducted a thorough experimental analysis along the lines of QBFEval'16, which shows that our approach is already competitive for 2-QBF instances as well as on instances of width up to 80 (even for more quantifier blocks). Additionally, we showed that the behavior of our system is indeed different from the diverse field of existing techniques. Seen in a broader context, our results clearly demonstrate the potential of parameterized algorithms for problems beyond NP in practice, in particular when combined with BDDs.

References

1. Abseher, M., Musliu, N., Woltran, S.: htd – a free, open-source framework for tree decompositions and beyond. Tech. Rep. DBAI-TR-2016-96, TU Wien (2016)
2. Abseher, M., Musliu, N., Woltran, S.: Improving the efficiency of dynamic programming on tree decompositions via machine learning. J. Artif. Intell. Res. 58, 829–858 (2017)
3. Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k-tree. SIAM J. Algebra. Discr. Meth. 8, 277–284 (1987)
4. Atserias, A., Oliva, S.: Bounded-width QBF is PSPACE-complete. J. Comput. Syst. Sci. 80(7), 1415–1429 (2014)
5. Benedetti, M.: Quantifier trees for QBFs. In: Proc. SAT. LNCS, vol. 3569, pp. 378–385. Springer (2005)
6. Benedetti, M.: sKizzo: A suite to evaluate and certify QBFs. In: Proc. CADE. LNCS, vol. 3632, pp. 369–376. Springer (2005)
7. Biere, A.: Resolve and expand. In: Proc. SAT. LNCS, vol. 3542, pp. 59–70. Springer (2004)
8. Bodlaender, H.L., Koster, A.M.C.A.: Treewidth computations I. Upper bounds. Inf. Comput. 208(3), 259–275 (2010)
9. Bogaerts, B., Janhunen, T., Tasharrofi, S.: Solving QBF instances with nested SAT solvers. In: Proc. Beyond NP. AAAI Workshops, vol. WS-16-05. AAAI Press (2016)

10. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers 100(8), 677–691 (1986)
11. Charwat, G., Woltran, S.: Dynamic programming-based QBF solving. In: Proc. QBF. CEUR Workshop Proceedings, vol. 1719, pp. 27–40. CEUR-WS.org (2016)
12. Chen, H.: Quantified constraint satisfaction and bounded treewidth. In: Proc. ECAI. pp. 161–165. IOS Press (2004)
13. Dechter, R.: Constraint Processing. Morgan Kaufmann (2003)
14. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Monographs in Computer Science, Springer (1999)
15. Ferrara, A., Pan, G., Vardi, M.Y.: Treewidth in verification: Local vs. global. In: Proc. LPAR. LNCS, vol. 3835, pp. 489–503. Springer (2005)
16. Friedman, S.J., Supowit, K.J.: Finding the optimal variable ordering for binary decision diagrams. In: Proc. DAC. pp. 348–356. ACM (1987)
17. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.M.: Solving QBF with counterexample guided refinement. In: Proc. SAT. LNCS, vol. 7317, pp. 114–128. Springer (2012)
18. Janota, M., Marques-Silva, J.: Abstraction-based algorithm for 2QBF. In: Proc. SAT. LNCS, vol. 6695, pp. 230–244. Springer (2011)
19. Janota, M., Marques-Silva, J.: Solving QBF by clause selection. In: Proc. IJCAI. pp. 325–331. AAAI Press (2015)
20. Lonsing, F., Bacchus, F., Biere, A., Egly, U., Seidl, M.: Enhancing search-based QBF solving by dynamic blocked clause elimination. In: Proc. LPAR. LNCS, vol. 9450, pp. 418–433. Springer (2015)
21. Lonsing, F., Biere, A.: A compact representation for syntactic dependencies in QBFs. In: Proc. SAT. LNCS, vol. 5584, pp. 398–411. Springer (2009)
22. Lonsing, F., Biere, A.: DepQBF: A dependency-aware QBF solver. J. SAT 7(2-3), 71–76 (2010)
23. Marin, P., Narizzano, M., Pulina, L., Tacchella, A., Giunchiglia, E.: An empirical perspective on ten years of QBF solving. In: Proc. RCRA. CEUR Workshop Proceedings, vol. 1451, pp. 62–75. CEUR-WS.org (2015)
24. Pan, G., Vardi, M.Y.: Symbolic decision procedures for QBF. In: Proc. CP. LNCS, vol. 3258, pp. 453–467. Springer (2004)
25. Pulina, L.: The ninth QBF solvers evaluation - preliminary report. In: Proc. QBF. CEUR Workshop Proceedings, vol. 1719, pp. 1–13. CEUR-WS.org (2016)
26. Pulina, L., Tacchella, A.: A structural approach to reasoning with quantified Boolean formulas. In: Proc. IJCAI. pp. 596–602. AAAI Press (2009)
27. Pulina, L., Tacchella, A.: An empirical study of QBF encodings: From treewidth estimation to useful preprocessing. Fundam. Inform. 102(3-4), 391–427 (2010)
28. Rabe, M.N., Tentrup, L.: CAQE: A certifying QBF solver. In: Proc. FMCAD. pp. 136–143. IEEE (2015)
29. Robertson, N., Seymour, P.D.: Graph minors. III. Planar tree-width. J. Comb. Theory, Ser. B 36(1), 49–64 (1984)
30. Samer, M., Szeider, S.: Backdoor sets of quantified Boolean formulas. J. Autom. Reasoning 42(1), 77–97 (2009)
31. Slivovsky, F., Szeider, S.: Computing resolution-path dependencies in linear time. In: Proc. SAT. LNCS, vol. 7317, pp. 58–71. Springer (2012)
32. Somenzi, F.: CU Decision Diagram package release 3.0.0. Department of Electrical and Computer Engineering, University of Colorado at Boulder (2015)
33. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time (preliminary report). In: Proc. TOC. pp. 1–9. ACM (1973)