Dynamic Programming-based QBF Solving

Günther Charwat and Stefan Woltran

Institut für Informationssysteme, TU Wien, Austria
{gcharwat,woltran}@dbai.tuwien.ac.at

Abstract. Solving Quantified Boolean Formulas (QBFs) is a challenging problem due to its high complexity. Many successful methods have been proposed, including extensions of DPLL/CDCL procedures and expansion-based approaches. In this paper, we present a novel method that is inspired by concepts from the field of parameterized complexity. More specifically, we develop a dynamic programming algorithm that traverses a tree decomposition of the QBF instance. We implemented our method using Binary Decision Diagrams as key ingredient to compactly represent the partial solutions computed during dynamic programming. Experiments indicate that our prototype shows promising results for instances with few quantifier alternations and where the treewidth of the propositional formula does not exceed 50. In fact, treewidth can be understood as a measurement of structure within the formula, and to the best of our knowledge has not been used explicitly in QBF solvers yet.

1 Introduction

Quantified Boolean Formulas (QBFs) are a powerful tool to compactly encode many computationally hard problems, since QBF satisfiability checking (QSAT) is PSPACE-complete. This makes QSAT amenable to several application fields where highly complex tasks emerge, e.g. planning, verification, and many more. Most of today's QBF solvers rely on extending the DPLL/CDCL procedures (see e.g. [21]), but also alternative methods based on Binary Decision Diagrams (BDDs) [23] or abstraction-refinement [16] proved successful.

Our approach is different and has its origin in the field of parameterized complexity [13]. Hereby, the computational costs for solving a particular problem is not solely related to the size of instance, but to some structural parameter. The goal then is to design algorithms that perform efficiently when the considered parameter is relatively small. The parameter we base our algorithm on is the treewidth [26] of the hypergraph obtained from the matrix of a given QBF in prenex CNF. Roughly speaking, treewidth measures the tree-likeness of a hypergraph, thus providing a structural parameter that reflects the shape of the CNF. Our goal is to develop a novel dynamic programming-based algorithm that is particularly efficient on tree-like QBF instances. For other logical problems, e.g. algorithms for SAT [28] and CSP [27], such treewidth-based algorithms have already been presented in the literature. However, to the best of our knowledge this method has not been applied to QSAT yet.

In a nutshell, our method is as follows. We split the QBF instance into subproblems by constructing a so-called tree decomposition of its hypergraph representation. The

QBF is then solved by dynamic programming over the tree decomposition. In contrast to standard methods where intermediate results are stored in tables (see e.g. [22]), we require a more complicated data structure in order to deal with the high complexity of QSAT. At the heart of this data structure we employ sets of BDDs. The motivation for that is two-fold: first BDDs allow for a compact representation of solutions; second, BDDs are canonical in the sense that equivalent formulas are represented by identical BDDs; thus in order to keep our data structure compact, explicit tests for duplicates are not needed. The size of each BDD is bounded by the width of the used decomposition, and the overall number of BDDs required in each node is bounded by the width and by the number of quantifiers in the instance. Thus, the runtime depends exponentially on the structural parameter instead of the size of the formula.

We provide some preliminary experiments which indicate that our method already performs well on QBFs with one quantifier alternation, while for QBFs with a higher number of alternations our system does not reach the performance of state-of-the-art tools yet. However, we encountered several instances that our solver was able to solve, but where others (we compared our system with DepQBF, RAReQS, and EBDDRES) ran into a timeout. Our method gives rise to several directions of advancements, such as QBF-tailored tree decomposition heuristics and width-reducing preprocessing.

2 Background

Quantified Boolean Formulae. 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. Depending on the context, we will sometimes denote clauses as sets of literals, and a formula in CNF as a set of clauses. Herein, we consider QBFs in closed prenex CNF (PCNF) form.

Definition 1. *A PCNF QBF instance is of the form $Q.\psi$ where Q is the quantifier prefix and ψ is a CNF formula. The* quantifier prefix *Q is of the form $Q_1X_1Q_2X_2 \ldots Q_kX_k$ where $Q_i \neq Q_{i+1}$ for $1 \leq i < k$. Furthermore, every variable in ψ occurs in exactly one set X_j for $1 \leq j \leq k$.*

The level *of a variable x is specified by its appearance in Q, i.e. if $x \in X_j$ then the level of x is j. We define the* depth *of x as k minus its level plus one.*

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

Example 1. As our running example, we will consider QBF $Q.\psi$ with $Q = \exists ab\, \forall cd\, \exists ef$ and $\psi = (a \vee c \vee e) \wedge (\neg b \vee d) \wedge (e \vee f) \wedge (c \vee \neg e) \wedge (\neg d \vee f)$, which is satisfiable (for $a = \top$, $b = \bot$). Note that this example is designed to illustrate our approach. Hence, simplifications (e.g. pure literal elimination) are not considered.

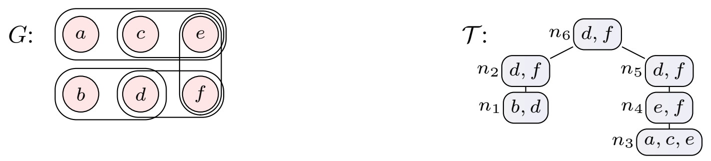

Fig. 1. Graph G and possible (weakly-normalized) tree decomposition $\mathcal{T}$ of G.

Tree Decompositions. In order to employ dynamic programming on tree decompositions (TDs) for QBF solving, we have to construct a tree decomposition from the given QBF instance. A TD $\mathcal{T}$ is a mapping from a graph to a tree, where vertices of the original graph are associated with nodes of $\mathcal{T}$. Herein, we consider hypergraphs, i.e. graphs where the edges may have multiple endpoints.

Definition 2. *A* tree decomposition *of a hypergraph $G = (V, E)$ is defined as a pair $\mathcal{T} = (T, bag_{\mathcal{T}})$ where $T = (N, F)$ is a (rooted) tree with nodes N and edges F, and $bag_{\mathcal{T}} : N \to 2^V$ assigns to each node a set of vertices, such that the following conditions are met:*

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

The *width* of $\mathcal{T}$ is $\max_{n \in N} |bag_{\mathcal{T}}(n)| - 1$. The *treewidth* of a hypergraph is the minimum width over all its tree decompositions. Constructing a tree decomposition with minimum width is intractable [2]. However there are good heuristics available [11, 12, 8]. In this paper we will consider so-called *weakly-normalized* tree decompositions. A tree decomposition can be transformed into a weakly-normalized one in linear time without increasing the width [18].

Definition 3. *A tree decomposition $\mathcal{T} = (T, bag_{\mathcal{T}})$ with $T = (N, F)$ is* weakly normalized *iff for each node $n \in N$ with children $n_1, \ldots, n_m$ such that $m \geq 2$, $bag_{\mathcal{T}}(n) = bag_{\mathcal{T}}(n_1) = \cdots = bag_{\mathcal{T}}(n_m)$ holds.*

A QBF instance $Q.\psi$ can naturally be represented as a hypergraph $G = (V, E)$ where V are the variables occurring in $Q.\psi$ and for each clause $c \in \psi$, $variables(c)$ forms a hyperedge in G. Note that this representation of CNF formulae is commonly used, e.g. for tree decomposition-based SAT solving [28]. The number of nodes in the TD is linear in the size of the QBF (i.e., its variables).

Example 2. Given formula $\psi = (a \vee c \vee e) \wedge (\neg b \vee d) \wedge (e \vee f) \wedge (c \vee \neg e) \wedge (\neg d \vee f)$ of our running example, Figure 1 illustrates its hypergraph representation G, and $\mathcal{T}$ represents a weakly-normalized tree decomposition for ψ of width 2.

Given a tree decomposition $\mathcal{T} = (T, bag_{\mathcal{T}})$ with $T = (N, F)$, for a tree decomposition node $n \in N$ we denote its set of children in T by $children_{\mathcal{T}}(n)$. In order to iterate over the children, we specify $firstChild_{\mathcal{T}}(n)$ and $nextChild_{\mathcal{T}}(n)$ as procedures to access the children, and $hasNextChild_{\mathcal{T}}(n)$ to check whether further children exist. $isLeaf_{\mathcal{T}}(n)$ returns true if n has no children, nodes with one child are called exchange nodes (tested by $\mathit{isExchange}_{\mathcal{T}}(n)$), and for nodes with more than one child, $\mathit{isJoin}_{\mathcal{T}}(n)$ returns true. For an exchange node n with child node n', we denote by $\mathit{introduced}_{\mathcal{T}}(n) = \mathit{bag}_{\mathcal{T}}(n) \setminus \mathit{bag}_{\mathcal{T}}(n')$ the variables introduced in n; and $\mathit{removed}_{\mathcal{T}}(n) = \mathit{bag}_{\mathcal{T}}(n') \setminus \mathit{bag}_{\mathcal{T}}(n)$ gives the variables removed in n. Furthermore, clauses of a QBF instance $Q.\psi$ are related to nodes in the tree decomposition by $\mathit{clauses}_{\mathcal{T},\psi}(n) = \{c \mid c \in \psi, \mathit{variables}_{\psi}(c) \subseteq \mathit{bag}_{\mathcal{T}}(n)\}$. For improved readability, we will usually omit subscript $\mathcal{T}$ and ψ.

Binary Decision Diagrams. A BDD is a well-studied and widely-used data structure that represents Boolean formulae in form of a rooted directed acyclic graph [19, 1]. Inhere, we use a special type of BDDs, so-called Reduced Ordered Binary Decision Diagrams (ROBDDs) [9] as one key ingredient for efficiently storing information. ROBDDs are a refinement of BDDs which are oftentimes particularly space-efficient. Furthermore, for a fixed ordering over variables occurring in the formula, they are canonical, i.e., equivalent formulae are represented by the same ROBDD. Beside standard logical operators ($\wedge, \vee, \neg, ...$), we assume BDDs B that support existential and universal quantification over sets of variables X, denoted by $\exists X B$ ($\forall X B$), as well as restriction of a variable x to true (denoted by $B[x/\top]$) or false ($B[x/\bot]$). In the following we will specify BDDs in form of Boolean formulae.

3 Dynamic Programming for QBFs

In a nutshell, dynamic programming on tree decompositions proceeds as follows: First, the CNF formula of the input instance, represented as hypergraph, is decomposed (using heuristics [11, 12, 8]). Then, the obtained tree decomposition is traversed in post-order. At each node, partial solution candidates for the problem at hand are computed. Solution candidates are "partial" because they are restricted to the current node's bag contents. They are computed by taking into account the partial solution candidates of the already-visited child nodes, together with the subgraph induced by the current node's bag. At the root node we obtain the solution to our problem. We will now first introduce the data structure to be used for storing partial solution candidates, and then provide our algorithm for dynamic programming-based QBF solving.

3.1 Data structure

As data structure we use so-called *nested sets of formulae* (NSFs) where the innermost sets contain Boolean formulae, represented as BDDs. Intuitively, an NSF resembles the structure of the QBF instance. The depth of the nesting in the NSF corresponds to the number of quantifiers in the QBF. The nestings are used to differentiate between variables that are at different depth in the quantifier prefix. NSFs, in relation to a QBF instance, are defined as follows.

Definition 4. *Given a QBF instance $Q.\psi$ with k quantifiers, we have a* nested set of formulae (NSF) *of depth k whose elements are 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 $d - 1$.*

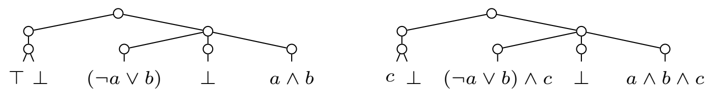

Fig. 2. Example NSF N, represented as tree, and $N[B/B \wedge c]$ applied to N.

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 $depth(M)$ the depth of the nesting of M, $level_Q(M) = k - depth(M) + 1$ is the level of M, and $quantifier_Q(M) = Q_{level_Q(M)}$ (for $level_Q(M) \leq k$). Subscripts will be again omitted in the following.

Additionally, we define the procedure $init(k, \phi)$ that initializes an NSF of depth k, such that each set contains exactly one NSF, and the innermost NSF represents ϕ. For instance, $init(3, \top)$ returns $\{\{\{\top\}\}\}$. Furthermore, for an NSF N we denote by $N[B/B']$ the replacement of each BDD B in N by some B'.

Example 3. Suppose we have given an NSF $N = \{\{\{\top, \bot\}\}, \{\{\neg a \vee b\}, \{\bot\}, \{a \wedge b\}\}\}$. For the ease of readability, we will illustrate nested sets in form of a tree where the leaves contain the Boolean formulae represented by the BDDs, and each circle denotes a non-leaf NSF in the nestings with its contents being the children in the tree. Figure 2 shows the tree representing NSF N together with the one resulting from $N[B/B \wedge c]$.

NSFs are tailored towards efficient representation of partial solution candidates. Opposed to the similar concept of quantifier trees [5], NSFs follow set semantics in order to automatically remove (trivial) redundancies. Furthermore, the depth of nestings is specified by the number of quantifiers, not by the variables in the instance. As we will see, NSFs can be used to keep track of parts of the solution space, instead of representing the whole QBF instance at once.

3.2 Dynamic programming on tree decompositions for QBF solving

Algorithm 1 illustrates the recursive procedure for the bottom-up traversal of the tree decomposition and computing the partial solution candidates. When called with the root node of the tree decomposition, it returns an NSF that represents the overall solution to the problem.

Procedure $compute(n)$ calls itself based on the child nodes of n. At each node, we distinguish between leaf, exchange and join nodes. In leaf nodes, an NSF of depth k (i.e., the number of quantifiers in the QBF instance) is initialized with the innermost set containing a BDD that represents the clauses associated with the current decomposition node. In an exchange node, we have to deal with removed as well as introduced variables. First the NSF of the child node is computed. Then, removed variables are handled by "splitting" the NSF. Procedure $split(N, x)$ (see Algorithm 2) handles this removal of a variable x. It is called recursively for the contents of the NSF until the level of x is reached. Then, for each NSF at this level, the NSF is updated once by replacing all occurrences of x in the BDDs with $\top$, and once with $\bot$. Due to the connectedness property of the tree decomposition, we know that a removed variable will never reappear somewhere upwards the tree decomposition, and therefore all clauses related to

Algorithm 1: Recursive procedure $compute(n)$ for QBF solving

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

if $isLeaf(n)$ then
    $N := init(k, clauses(n))$
if $isExchange(n)$ then
    $N := compute(firstChild(n))$
    for $x \in removed(n)$ do
        $N := split(N, x)$
    end
    $N := N[B/B \wedge clauses(n)]$
if $isJoin(n)$ then
    $N := compute(firstChild(n))$
    while $hasNextChild(n)$ do
        $M := compute(nextChild(n))$
        $N := join(N, M)$
    end
return $N$
```

Algorithm 2: Recursive procedure $split(N, x)$

```
Input  : An NSF $N$ and a variable $x$
Output: An NSF split at $level(x)$

if $level(N) = level(x)$ then
    return $\{M[B/B[x/\top]], M[B/B[x/\bot]]) \mid M \in N\}$
else
    return $\{split(M, x) \mid M \in N\}$
```

Algorithm 3: Recursive procedure $join(N_1, N_2)$

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

if $depth(N_1) = 0$ then
    return $N_1 \wedge N_2$
else
    return $\{join(M_1, M_2) \mid M_1 \in N_1, M_2 \in N_2\}$
```

the removed variable were already considered. Thereby we are also guaranteed that the size of each BDD is bounded by the bag's size. After splitting, the BDDs in the NSF are updated by adding the clauses associated with the current node via conjunction to the BDDs in the NSF. In join nodes, NSFs computed in the child nodes are successively combined by procedure $join(N_1, N_2)$ (see Algorithm 3). Observe that the procedure guarantees that the structure (nesting) of the NSFs to be joined is preserved. BDDs in the NSFs are then combined via conjunction, thus already considered information (i.e., clauses of the sub-hypergraph induced by the subtree's bag) of both child tree decom-

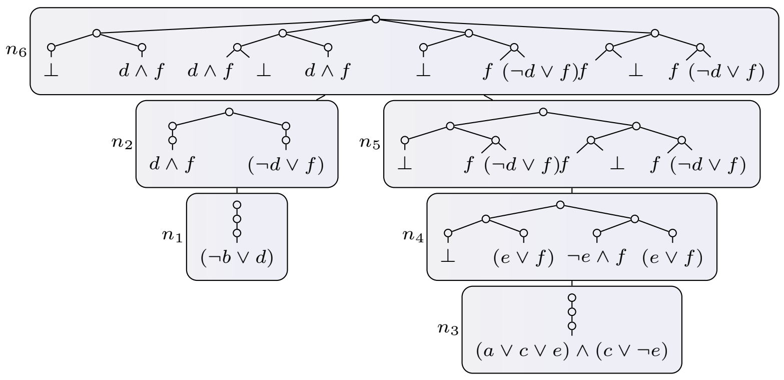

Fig. 3. Computed NSFs for our running example.

position nodes is combined. Note that this procedure does not take the quantifiers of the QBF instance into account. They will be evaluated on the NSF of the root node.

Example 4. Figure 3 shows the NSFs computed at the tree decomposition nodes of our running example. In n_1, an NSF of depth 3 is initialized with $(\neg b \vee d)$, i.e., the clause associated with this tree decomposition node. In n_2 variable b is removed. Hence the NSF is split at $\mathit{level}(b) = 1$, once by setting b to true (left NSF branch), yielding formula d and once by false (right branch), yielding $\top$. Furthermore, current clause $(\neg d \vee f)$ is added to these BDDs via conjunction. Similarly, the right branch of the tree decomposition (nodes n_3–n_5) is computed. In n_6, the NSFs are joined. For instance, the leftmost branches in n_2 and n_5 are joined by conjunction of $d \wedge f$ and $\bot$, yielding$\bot$.

3.3 Obtaining the solution

At the root node r of the tree decomposition, we can decide the problem since the whole input instance was taken into account. We apply quantifier elimination by evaluating the NSF as shown in Algorithm 4, which is similar to the approach described in [24]. Procedure $\mathit{evaluateQ}(r, N)$ recursively combines the elements of the NSF by disjunction

Algorithm 4: Recursive procedure $\mathit{evaluateQ}(n, N)$

```
Input  : A tree decomposition node $n$ and an NSF $N$
Output: A BDD $B$ of $N$, obtained by evaluating the quantifiers
if $depth(N) = 0$ then
    $B := N$
else
    $X := \{x \mid x \in bag(n) \text{ and } level(x) = level(N)\}$
    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 $B$
```

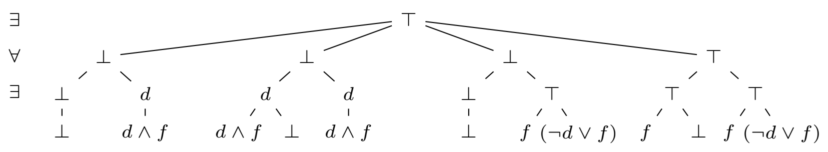
∃
∀
∃
⊤
⊥ ⊥ ⊥ ⊤
⊥ d d d ⊥ ⊤ ⊤ ⊤
⊥ $d \wedge f$ $d \wedge f$ ⊥ $d \wedge f$ ⊥ f $(\neg d \vee f)$ f ⊥ f $(\neg d \vee f)$

Fig. 4. Results for $\mathit{evaluateQ}(n_6, N)$ executed on the NSF of root node n_6.

(for existential quantifiers) or conjunction (for universal quantifiers), starting at the innermost NSFs. Furthermore, variables contained in the current bag are abstracted away from the merged BDD according to the quantifier. Thus, this procedure finally returns a single BDD B without variables. There, if $B \equiv \bot$, the QBF instance is unsatisfiable, otherwise it is satisfiable.

Example 5. Figure 4 shows the NSF N in root node n_6 of our running example, and the BDDs obtained recursively (bottom-up) when applying $\mathit{evaluateQ}(r, N)$. Note that $\mathit{bag}(n_6) = \{d, f\}$ with $\mathit{level}(d) = 2$ and $\mathit{level}(f) = 3$, which are additionally taken into account when evaluating the quantifiers. The procedure returns $\top$ for our running example, hence the QBF is satisfiable.

We omit a formal proof of the correctness of the proposed algorithm; instead we give an informal discussion about its runtime. Given a QBF $Q.\psi$ with $Q = Q_1 X_1 \dots Q_k X_k$ and a tree decomposition for ψ of width w, the algorithm determines the truth of $Q.\psi$ in time $\mathcal{O}(2^{2^{\cdot^{\cdot^{2^{w+1}}}}} \cdot |\psi|)$, where the height of the tower of exponents in $2^{2^{\cdot^{\cdot^{2^{w+1}}}}}$ is $k + 1$, since the size of each BDD is at most 2^{w+1} and we have k quantifiers[1]. Furthermore, $|\psi|$ denotes the size of ψ. We recall that the number of nodes of a tree decomposition is linear in the size of ψ. Moreover, any BDD involved in the algorithm is given over at most w variables and NSFs are just built upon such BDDs. Thus all operations on NSFs are also bound by w. Recall that due to the canonical form of BDDs, there are no duplicates at any level of an NSF, thus yielding this runtime.

3.4 Algorithm Optimizations

Although our algorithm runs in polynomial time for bounded treewidth of the QBF instance (for a fixed number of quantifiers), refinements are necessary in order to make it useful in practice. Herein, we discuss several optimizations for our algorithm.

Intermediate unsatisfiability checks. One optimization is to check for unsatisfiability of the QBF instance during the bottom-up traversal of the tree decomposition. We can directly reuse procedure $\mathit{evaluateQ}(n, N)$. Whenever it returns $\bot$, the QBF is unsatisfiable, and we can immediately abort our main procedure $\mathit{compute}(n)$. However, if it returns $\top$, the QBF might still be unsatisfiable due to clauses that are encountered later during the traversal.

[1] It is known that QSAT can be solved in FPT time when treewidth and number of quantifiers are bounded, which follows, for instance, from [10]. However, the QSAT problem is *not* fixed-parameter tractable w.r.t. parameter treewidth [3], unless the number of quantifiers is also bound or, more generally, the dependencies between variables are restricted (see [14]).

Algorithm 5: Recursive procedure $\mathit{removeRedundant}(N)$

```
Input  : An NSF $N$
Output: An NSF without supersets
if $depth(N) > 1$ then
    for $M \in N$ do
        $M := \mathit{removeRedundant}(M)$
    end
    for $M_1, M_2 \in N$ and $M_1 \neq M_2$ do
        if $M_1 \subset M_2$ then
            $N := N \setminus \{M_2\}$
    end
else
    // $N$ contains a set of BDDs
    for $M_1, M_2 \in N$ and $M_1 \neq M_2$ do
        if $\mathit{quantifier}(N) = \exists$ and $M_1 \vee M_2 = M_1$ then
            $N := N \setminus \{M_2\}$
        if $\mathit{quantifier}(N) = \forall$ and $M_1 \wedge M_2 = M_1$ then
            $N := N \setminus \{M_2\}$
    end
return $N$
```

Evaluate innermost quantifier. For any NSF N at depth one, the quantifier can be evaluated immediately: A single BDD is constructed by disjunction (for existential quantification) or conjunction (for universal quantification) of the BDDs contained in N. Thereby, the overall size can be reduced, since usually the single BDD stores models more efficiently than several BDDs. Furthermore, redundant models (i.e., models that are stored in several BDDs) are now only kept once, and for universal quantification additionally only models appearing in all BDDs are stored in the newly created BDD.

Remove redundant NSFs. Redundant NSFs can be removed by checking for subsets w.r.t. models represented by the BDDs (similar to subsumption checking [7]), and subsets w.r.t. nested sets. Procedure $\mathit{removeRedundant}(N)$ (see Algorithm 5) gives the pseudo-code for removing unnecessary elements.

Balance NSF and BDD size. By delaying the split of removed variables (and storing them in a cache), the size of the NSF can be kept small. However, this usually increases the size of the BDDs (since the variables are not abstracted away). Note that a join node can drastically increase the size of an NSF, which has to be considered already below that tree decomposition node, when vertices are removed.

Example 6. Figure 5 shows the NSFs computed at the tree decomposition nodes of our running example. Compared to Figure 3, here our algorithm optimizations were taken into account. For instance, due to immediate evaluation of the innermost quantifier, the sets at level three only contain one BDD. Another example would be the NSF in n_4, where in the left branch the NSF containing BDD $(e \vee f)$ (cf. Figure 3) is removed since its models are a superset of $\bot$.

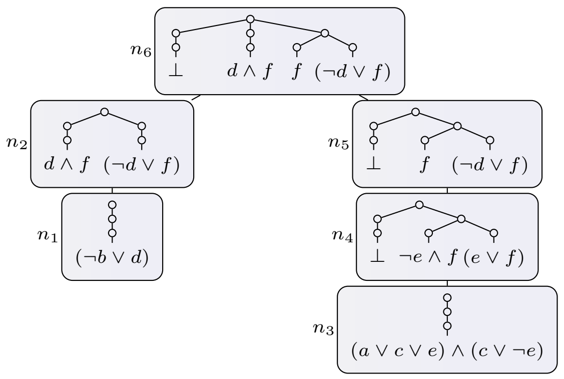
n_6
$\bot$ $d \wedge f$ f $(\neg d \vee f)$
n_2
$d \wedge f$ $(\neg d \vee f)$
n_5
$\bot$ f $(\neg d \vee f)$
n_1
$(\neg b \vee d)$
n_4
$\bot$ $\neg e \wedge f$ $(e \vee f)$
n_3
$(a \vee c \vee e) \wedge (c \vee \neg e)$

Fig. 5. Computed NSFs (including optimizations) for our running example.

4 Preliminary Experimental Analysis

We implemented our approach in a system called *dynQBF*. Internally an improved version of HTDECOMP [12] for obtaining the tree decompositions (using the *minimum fill* heuristic) and CUDD [29] for BDD management (using heuristic *lazy sifting* for dynamic variable reordering) are used. Additionally, we implemented a variant called *BDD (naive)*, where the formula is not decomposed, but instead given to a single BDD at once. This gives a hint of whether the overhead for constructing and traversing the decomposition pay off. We compare the total runtime to DepQBF [20] (v. 5.0) and RAReQS [16] (v. 1.1), which succeeded in the latest QBF competition [15]. Additionally, we consider the BDD-based system EBDDRES [17] (v. 1.2), which originally was designed as a SAT solver that provides resolution proofs. Unfortunately, we could not include the BDD-based QBF solvers QBFBDD [4] and eBDD-QBF [23], as well as the quantifier-tree based tool sKizzo [6], since, to the best of our knowledge, they are not publicly available. Tests were performed on a single core of an Intel Xeon E5-2637 processor with 3.5GHz running Debian 8.3. Each run was limited to a runtime of 10 minutes (TO) and 16 GB of memory (MO).

2-QBF Instances. We used 200 publicly available 2-QBF (i.e., QBFs with a $\forall\exists$ quantifier prefix) competition instances[2]. Table 1 reports the number of solved instances. DepQBF solved the most instances, closely followed by dynQBF, and BDD (naive). EBDDRES is only available as 32-bit library, hence it is limited to roughly 4GB of memory, which explains the high number of memouts. However, this system also requires more time for solved instances compared to the better-performing systems. Interestingly, there are several instances that are uniquely solved by one of the compared systems. This indicates that the solvers work particularly well for different types of instances. dynQBF seems to handle satisfiable instances better than DepQBF, but unsatisfiability is oftentimes not detected within the time limit. One reason could be that the intermediate unsatisfiability checks of dynQBF are less powerful than conflict-driven

[2] Available at `http://www.qbflib.org/TS2010/2QBF.tar.gz`.

Table 1. 2-QBF: System comparison

System	Solved	SAT	UNSAT	Timeout	Memout	Uniquely solved
DepQBF	109	51	58	91	0	43
dynQBF	108	85	23	92	0	26
BDD (naive)	41	40	1	159	0	0
EBDDRES	34	34	0	0	166	2
RAReQS	32	24	8	168	0	5

Table 2. QBF Gallery 2014: System comparison

System	Solved	SAT	UNSAT	Timeout	Memout	Uniquely solved
DepQBF	103	48	55	169	4	42
RAReQS	83	36	47	193	0	22
dynQBF	21	6	15	250	5	8
EBDDRES	7	5	2	4	265	2
BDD (naive)	3	1	2	273	0	0

clause learning in DepQBF. Furthermore, the decomposition width of the instances is important: All instances were decomposed within the time limit. For instances with a (heuristically obtained) tree decomposition of width up to 50, dynQBF solved 54 out of 55 instances, while DepQBF only solved 28 of these instances. This result is in line with design of our algorithm, which directly tries to exploit this structural parameter.

QBF Gallery 2014. Here we considered the 276 instances[3] of the latest QBF competition [15]. Table 2 shows the overall number of solved instances. Here, dynQBF is not yet competitive, with only 21 solved instances. We were not able to decompose 27 out of 276 instances within the time limit. On average, tree decompositions for instances solved by dynQBF have a width of 55. These instances contain (on average) 3 quantifiers and 4711 atoms and 16409 clauses. Furthermore, the data set contains a single class of instances with only 2 quantifiers and an average width of 80, named "stmt*". dynQBF uniquely solved 7 out of 12 of these instances, which again indicates that our approach is well-suited for instances of this form.

5 Discussion

Overall, while our early prototypical implementation is already quite promising for 2-QBF instances, future research will focus on how to handle more quantifier alternations. From the experimental evaluation we see that further unsatisfiability checking strategies should be investigated. Most importantly, note that our system currently does not implement any preprocessing at all. Besides standard approaches for variable and clause elimination, width-reducing preprocessing seems to be worth investigating. Additionally, problem-tailored tree decompositions can be developed, where, for instance, clauses that are likely to be responsible for unsatisfiability of the instance appear together near the leaf nodes. Furthermore, we believe that variables with a high level should be assigned to bags near the decomposition leaves.

[3] Available at `http://qbf.satisfiability.org/gallery/eval2012r2.tgz`.

Besides deciding QSAT, enumerating solutions (in case the outermost quantifier is existential) is also supported by our approach. Here, we do not split and abstract away variables contained in the outermost quantifier block. Note, however, that the BDD's size is then no longer bounded by the width of the decomposition, but additionally by the number of variables in the first quantifier block. At root node r, procedure $evaluateQ(r, N)$ is adapted to keep all variables of the first quantifier block. The resulting NSF then contains exactly one BDD representing the solutions.

Our algorithm can also be used for prenex DNF QBF solving, where the input is provided in disjunctive normal form (DNF). Here, the hypergraph is constructed by considering the terms in the DNF, and dual to introduced clauses for CNF-based solving, introduced terms are added to the BDDs by disjunction. Similar to the $evaluateQ(n, N)$ procedure, we check for *satisfiability* of the DNF QBF during the bottom-up traversal.

Since BDDs support arbitrary Boolean formulae, they are well suited for non-CNF instances. Additionally, our data structure can be extended to reflect the structure of a non-prenex QBF, which takes the nestings of quantifiers in the QBF into account. Here, an NSF no longer contains NSFs of same depth, but the depth is determined by the different nestings of quantifiers in the QBF. Here, the hypergraph is constructed by considering the outermost subformulae that are connected via conjunction (or disjunction). As long as the QBF consists of reasonably many such subformulae, our tree decomposition-based approach could pay off, without introducing much overhead compared to solving the respective PCNF QBF instance. Additionally, we note that such non-prenex non-CNF QBFs potentially reflect the structure of the original problem, which can get lost during the translation to the prenex CNF format.

6 Conclusion

In this work, we have presented the first QBF solver that relies on the method of decomposition and dynamic programming, thus taking structural properties of the instance explicitly into account. Initial experiments showed the potential of this method for $\forall\exists$-QBFs, revealing that this method appears to be well suited for particular classes of QBFs that are hard to solve for other systems. This is in line with observations for SAT solving, where resolution-based solving can outperform search (DPLL-like) solving for instances of small width [25]. However, for formulas with more alternations and higher width, further improvements are necessary to be competitive with state-of-the-art solvers. One explanation for the rather poor performance of our method on more involved formulas is that we have not considered any analysis of quantifier dependencies in our algorithm yet. Thus, incorporating information about such dependencies to reduce the size of our data structures is on top of our agenda for future work. On the other hand, since our method is in principle not restricted to normal forms, we also plan to extend our prototype to work with non-prenex non-CNF formulas. Finally, we still need a better understanding of the interplay between different variable orderings in the BDDs and the shape of the used decomposition. Since good variable orderings are crucial to keep the size of the BDDs low, we expect that such insights can also lead to significant improvements on certain instances.

Acknowledgements. We would like to thank Florian Lonsing for his hints on QBF solving and Michael Abseher for providing us with efficient tree decomposition heuristics. This work has been supported by the Austrian Science Fund (FWF): Y698, P25607, P25518.

References

1. Akers, S.B.: Binary decision diagrams. IEEE Transactions on Computers 100(6), 509–516 (1978)
2. Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k-tree. SIAM J. Algebraic Discrete Methods 8, 277–284 (1987)
3. Atserias, A., Oliva, S.: Bounded-width QBF is PSPACE-complete. J. Comput. Syst. Sci. 80(7), 1415–1429 (2014), `http://dx.doi.org/10.1016/j.jcss.2014.04.014`
4. Audemard, G., Sais, L.: SAT based BDD solver for quantified Boolean formulas. In: Proc. ICTAI. pp. 82–89. IEEE Computer Society (2004), `http://dx.doi.org/10.1109/ICTAI.2004.106`
5. Benedetti, M.: Quantifier trees for QBFs. In: Bacchus, F., Walsh, T. (eds.) Proc. SAT. LNCS, vol. 3569, pp. 378–385. Springer (2005), `http://dx.doi.org/10.1007/11499107_28`
6. Benedetti, M.: sKizzo: A suite to evaluate and certify QBFs. In: Nieuwenhuis, R. (ed.) Proc. CADE. LNCS, vol. 3632, pp. 369–376. Springer (2005), `http://dx.doi.org/10.1007/11532231_27`
7. Biere, A.: Resolve and expand. In: Hoos, H.H., Mitchell, D.G. (eds.) Proc. SAT. LNCS, vol. 3542, pp. 59–70 (2004)
8. Bodlaender, H.L., Koster, A.M.C.A.: Treewidth computations I. Upper bounds. Inf. Comput. 208(3), 259–275 (2010)
9. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers 100(8), 677–691 (1986)
10. Chen, H.: Quantified constraint satisfaction and bounded treewidth. In: de Mántaras, R.L., Saitta, L. (eds.) Proc. ECAI. pp. 161–165. IOS Press (2004)
11. Dechter, R.: Constraint Processing. Morgan Kaufmann (2003)
12. Dermaku, A., Ganzow, T., Gottlob, G., McMahan, B.J., Musliu, N., Samer, M.: Heuristic methods for hypertree decomposition. In: Proc. MICAI. LNCS, vol. 5317, pp. 1–11. Springer (2008)
13. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Monographs in Computer Science, Springer (1999)
14. Eiben, E., Ganian, R., Ordyniak, S.: Using decomposition-parameters for QBF: Mind the prefix! In: Schuurmans, D., Wellman, M.P. (eds.) Proc. AAAI. pp. 964–970. AAAI Press (2016)
15. Janota, M., Jordan, C., Klieber, W., Lonsing, F., Seidl, M., Gelder, A.V.: The QBFGallery 2014: The QBF competition at the FLoC olympic games. JSAT (special issue on SAT 2014 Competitions) 9, 187–206 (2015)
16. 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), `http://dx.doi.org/10.1007/978-3-642-31612-8_10`
17. Jussila, T., Sinz, C., Biere, A.: Extended resolution proofs for symbolic SAT solving with quantification. In: Proc. SAT. LNCS, vol. 4121, pp. 54–60. Springer (2006), `http://dx.doi.org/10.1007/11814948_8`

18. Kloks, T.: Treewidth, Computations and Approximations, LNCS, vol. 842. Springer (1994)
19. Lee, C.: Representation of switching circuits by binary-decision programs. Bell System Technical Journal 38, 985–999 (1959)
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), `http://dx.doi.org/10.1007/978-3-662-48899-7_29`
21. Lonsing, F., Biere, A.: DepQBF: A dependency-aware QBF solver. JSAT 7(2-3), 71–76 (2010), `http://jsat.ewi.tudelft.nl/content/volume7/JSAT7_6_Lonsing.pdf`
22. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford Lecture Series in Mathematics and its Applications, OUP (2006)
23. Olivo, O., Emerson, E.A.: A more efficient BDD-based QBF solver. In: Proc. CP. LNCS, vol. 6876, pp. 675–690. Springer (2011), `http://dx.doi.org/10.1007/978-3-642-23786-7_51`
24. Pan, G., Vardi, M.Y.: Symbolic decision procedures for QBF. In: Wallace, M. (ed.) Proc. CP. LNCS, vol. 3258, pp. 453–467. Springer (2004), `http://dx.doi.org/10.1007/978-3-540-30201-8_34`
25. Rish, I., Dechter, R.: Resolution versus search: Two strategies for SAT. J. Autom. Reasoning 24(1/2), 225–275 (2000), `http://dx.doi.org/10.1023/A:1006303512524`
26. Robertson, N., Seymour, P.D.: Graph minors. III. Planar tree-width. J. Comb. Theory, Ser. B 36(1), 49–64 (1984)
27. Sachenbacher, M., Williams, B.C.: Bounded search and symbolic inference for constraint optimization. In: Proc. IJCAI. pp. 286–291. PBC (2005)
28. Samer, M., Szeider, S.: Algorithms for propositional model counting. J. Discrete Algorithms 8(1), 50–64 (2010), `http://dx.doi.org/10.1016/j.jda.2009.06.002`
29. Somenzi, F.: CU Decision Diagram package release 3.0.0. Department of Electrical and Computer Engineering, University of Colorado at Boulder (2015)