Expansion-based QBF Solving on Tree Decompositions

Günther Charwat[*]
Institute of Logic and Computation
TU Wien
Favoritenstraße 9-11, 1040 Vienna, Austria
gcharwat@dbai.tuwien.ac.at

Stefan Woltran[*]
Institute of Logic and Computation
TU Wien
Favoritenstraße 9-11, 1040 Vienna, Austria
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, binary decision diagrams (BDDs) are used for compactly storing partial solutions. Towards efficiency in practice, we integrate dependency schemes and develop 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, even for more quantifier alternations. Results indicate that our approach is orthogonal to existing techniques, with a large number of uniquely solved instances.

Keywords: QBF, expansion, dynamic programming, QSAT, treewidth

Address for correspondence: `gcharwat@dbai.tuwien.ac.at`
[*]This work was supported by the Austrian Science Fund (FWF): Y698.

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 [1]. We present an approach that is motivated by results from the area of parameterized complexity: many computationally hard problems are *fixed-parameter tractable* (fpt) [2], 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 [3]), 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 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) [5] for compactly storing information in our dedicated data structure. Second, we consider structure in the quantifier prefix by integrating dependency schemes (see, e.g., [6]) 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 approach is expected suitable for QBF instances of low-to-medium treewidth and a restricted number of quantifier alternations.

The presented algorithms are implemented in the QBF solver *dynQBF*[1]. We conduct an experimental evaluation along the lines of the 2016 [7] and 2017[2] QBF competitions. 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.

This article is structured as follows. In Section 2 we summarize QBF solving techniques and give a special account to related work that considers treewidth. In Section 3 we formally introduce QBFs and tree decompositions. Our core algorithm for QBF solving is presented in Section 4. Subsequently, the algorithm is extended by dependency scheme integration. Towards efficiency in practice, in Section 5 we introduce algorithmic refinements and heuristic optimizations. Section 6 contains our experimental analysis. Finally, correctness of our core algorithm is shown in Appendix A.

This article is a significantly extended version of a paper that appeared in the proceedings of the RCRA'17 workshop [8].

[1]dynQBF is freely available at `https://github.com/gcharwat/dynqbf/`.

[2]See `http://www.qbflib.org/qbfeval17.php`.

2. Related Work

Approaches for QBF solving have been intensively investigated in the past 20 years. They are oftentimes based on ideas that were originally developed for SAT solving, but extended with techniques to handle quantification over variables. An overview is given in the survey article by Marin et al. [9].

In a nutshell, solving with *expansion* proceeds by successively eliminating variables via duplication of the (sub) formula, and replacing the eliminated variable once by *true* and once by *false*. Early systems that apply this technique are QUBOS [10] and Quantor [11]. Expansion potentially yields an exponential blow-up in the size of the formula. The solvers AReQS [12] and RAReQS [13] mitigate this problem by applying *counter-example guided refinement* (CEGAR) [14]: here, the idea is to first consider an abstract representation of the problem instance, and then to gradually refine the abstraction until a solution is obtained. In our approach, which is also expansion-based, we decompose the input instance in order to reduce space requirements during solving.

Search-based QBF solving is based on extensions of the well-known DPLL or CDCL algorithms for SAT solving to handle quantification over variables [15, 16, 17]. CDCL is, for instance, implemented in the QBF solver DepQBF [18]. DepQBF additionally analyzes (in)dependencies between variables [19]. In our approach we also rely on dependency schemes to reduce computational effort and memory consumption. Another search-based solver is GhostQ [20] which introduces *ghost variables* that help handle existential and universal variables symmetrically, and to efficiently propagate information. There also exists a version of GhostQ that integrates CEGAR-based learning [13]. The solver Qestos is based on ideas presented in [21]. It combines DPLL-based and expansion-based solving techniques, and calls a SAT solver.

The concept most closely related to our approach was developed by Pan and Vardi [22] and implemented in the QBDD system, which relies on BDDs. Variable elimination follows an *elimination ordering* that also underlies the construction of the tree decomposition in our approach. However, with our explicit approach of dynamic programming on tree decompositions, we gain additional flexibility on the order in which variables are to be eliminated. Similarly, the early BDD-based solver EBDDRES [23] implements an algorithm whose runtime is exponential only in the width of the used elimination ordering. Similar to QBDD, elimination has to follow the variable ordering of the prefix.

Pulina and Taccella studied the relation of treewidth to (empirical) hardness of QBFs [24]. They consider *quantified treewidth* that is a generalization of primal treewidth (i.e., the treewidth of the primal graph representation of the QBF matrix) by also integrating the variable ordering specified by the QBF prefix (see also [25]). They show that reducing the quantified treewidth by preprocessing consistently improves the performance of solvers [26], and incorporate the idea of quantified treewidth reduction in the (incomplete) solver QuBIS [27]. The solver may either solve the instance, or return a modified QBF whose quantified treewidth is (usually) not larger than the treewidth of the original instance. The solver StruQS [28] dynamically decides between resolution and search during the solving process. Resolution is applied when the treewidth (or an approximation thereof) is relatively small (or, in particular, if the number of clauses resulting from the elimination of a variable is less than the number of clauses it originally occurred in), and search is used otherwise. An analysis of available QBF instance characteristics, such as clause length, number of variables, but also quantified (tree)width is given in [9].

3. Preliminaries

3.1. 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. 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 3.1. In a *PCNF QBF* $\mathsf{Q}.\psi$, 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 $1 \leq i < k$, $Q_k = \exists$ 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 *block* of x, and $k - l + 1$ the *depth* of x.

We frequently use the following notation: Given a QBF $\mathsf{Q}.\psi$ with $\mathsf{Q} = Q_1X_1\dots Q_kX_k$ and an index i with $1 \leq i \leq k$, $\mathit{quantifier}_{\mathsf{Q}}(i) = Q_i$ gives the i-th quantifier. For a variable x, $\mathit{block}_{\mathsf{Q}}(x)$ returns the block of x; $\mathit{depth}_{\mathsf{Q}}(x)$ returns the depth of x in Q; and $\mathit{quantifier}_{\mathsf{Q}}(x) = Q_{\mathit{block}_{\mathsf{Q}}(x)}$ 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. For a variable x in a formula ψ, we denote by $\psi[x/\cdot]$ with $\cdot \in \{\top, \bot\}$ the assignment of true (respectively false) to x in ψ.

A PCNF QBF $\mathsf{Q}.\psi$ can be decided by *expansion* of all variables, where a subformula of the form $\exists x \psi'$ in $\mathsf{Q}.\psi$ is expanded to $\psi'[x/\top] \vee \psi'[x/\bot]$, and subformula $\forall x \psi'$ is expanded to $\psi'[x/\top] \wedge \psi'[x/\bot]$. $\mathsf{Q}.\psi$ is said to be *satisfiable* (or *valid*) if the formula resulting from expansion over all variables equals to *true*.

Example 3.2. As our running example, we will consider the QBF $\mathsf{Q}.\psi$ with $\mathsf{Q} = \exists ab\, \forall cd\, \exists ef$ and $\psi = (a \vee c \vee e) \wedge (\neg a \vee b) \wedge (\neg b \vee f) \wedge (d \vee \neg e)$, which is valid. For instance, with $a/\top, b/\top$, for any assignment to c and d, there exists an assignment to e and f such that the formula evaluates to *true*. Note that this example is designed for illustration purposes, hence simplifications (such as pure literal elimination) are not considered.

3.2. Tree Decompositions and Treewidth

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

Definition 3.3. A *tree decomposition* of a graph $G = (V, E)$ is a pair $\mathcal{T} = (S, \mathit{bag}_{\mathcal{T}})$ where $S = (T, F)$ is a (rooted) tree with nodes T and edges F, and $\mathit{bag}_{\mathcal{T}} : T \to 2^V$ assigns to each node a set of vertices, such that:

1. For every vertex $v \in V$, there exists a node $t \in T$ such that $v \in \mathit{bag}_{\mathcal{T}}(t)$.
2. For every edge $(v_1, v_2) \in E$, there exists a node $t \in T$ such that $\{v_1, v_2\} \subseteq \mathit{bag}_{\mathcal{T}}(t)$.
3. For every vertex $v \in V$, the subtree of S induced by $\{t \in T \mid v \in \mathit{bag}_{\mathcal{T}}(t)\}$ is connected.

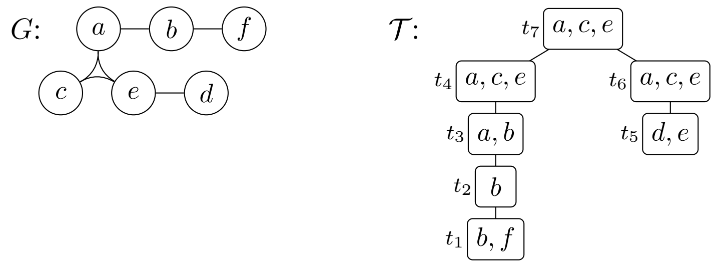

Figure 1. Primal graph G and a possible TD $\mathcal{T}$ of G for the QBF in Example 3.2.

Intuitively, Condition 1 and 2 guarantee that the whole graph is covered by the tree decomposition, and Condition 3 is the *connectedness property*, which, roughly speaking, states that a vertex cannot "reappear" in unconnected parts (w.r.t. the bags). For a tree decomposition $\mathcal{T} = (S, bag_{\mathcal{T}})$ with $S = (T, F)$ we sometimes write $t \in \mathcal{T}$ instead of $t \in T$ to access a tree decomposition node.

The *width* of $\mathcal{T}$ is defined as $\max_{t \in \mathcal{T}} |bag_{\mathcal{T}}(t)| - 1$. The *treewidth* w of a graph is the minimum width over all its tree decompositions. Given a graph and an integer w, deciding whether the graph has at most treewidth w is NP-complete [30]. However, the problem itself is fpt when w is considered as parameter [31]. Additionally, there exist good polynomial-time heuristics for constructing TDs [32, 33]. Here we consider a special type of tree decompositions.

Definition 3.4. A tree decomposition $\mathcal{T} = ((T, F), bag_{\mathcal{T}})$ is *weakly normalized*, if each $t \in T$ is

- a *leaf* node (t has no children),
- an *exchange* node (t has exactly one child t_1, such that $bag_{\mathcal{T}}(t) \neq bag_{\mathcal{T}}(t_1)$); or
- a *join* node (t has children $t_1, \dots, t_m$ such that $m \geq 2$, and $bag_{\mathcal{T}}(t) = bag_{\mathcal{T}}(t_1) = \cdots = bag_{\mathcal{T}}(t_m)$).

Given a tree decomposition $\mathcal{T} = (S, bag_{\mathcal{T}})$ with $S = (T, F)$, for a node $t \in T$ we denote its set of children in S by $children_{\mathcal{T}}(t)$. We specify $firstChild_{\mathcal{T}}(t)$ and $nextChild_{\mathcal{T}}(t)$ to iterate over the children, and $hasNextChild_{\mathcal{T}}(t)$ to check whether further children exist. The node type is checked with $isLeaf_{\mathcal{T}}(t)$, $isExchange_{\mathcal{T}}(t)$ and $isJoin_{\mathcal{T}}(t)$. For a node t with single child node t_1, changed bag contents are accessed by $introduced_{\mathcal{T}}(t) = bag_{\mathcal{T}}(t) \setminus bag_{\mathcal{T}}(t_1)$ and $removed_{\mathcal{T}}(t) = bag_{\mathcal{T}}(t_1) \setminus bag_{\mathcal{T}}(t)$. $isRoot_{\mathcal{T}}(t)$ returns true if t has no parent node.

Our algorithm for QBF solving is based on a tree decomposition of the given QBF $\mathsf{Q}.\psi$, 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 tree decomposition $\mathcal{T} = (S, bag_{\mathcal{T}})$ of QBF $\mathsf{Q}.\psi$, for $t \in \mathcal{T}$ we define $clauses_{\mathcal{T},\psi}(t) = \{c \in \psi \mid variables_\psi(c) \subseteq bag_{\mathcal{T}}(t)\}$. In the following we usually omit subscripts $\mathcal{T}$ and ψ.

Example 3.5. Consider our running example with $\psi = (a \vee c \vee e) \wedge (\neg a \vee b) \wedge (\neg b \vee f) \wedge (d \vee \neg e)$. Figure 1 shows the graph representation G of ψ, and $\mathcal{T}$ is a weakly-normalized TD for G of width 2.

4. 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 tree decomposition $\mathcal{T} = (S, bag_{\mathcal{T}})$ of the primal graph of ψ. Then, $\mathcal{T}$ is traversed in post-order. For each $t \in \mathcal{T}$ we compute *partial solution candidates* and store them in a dedicated data structure (called nested set of formulae (NSF), see below). In this context, *partial* means that the data structure is restricted to variables occurring in $bag_{\mathcal{T}}(t)$. *Candidate* refers to the fact that other parts of the QBF might not yet be considered. Intuitively, during traversal of the tree decomposition for variables that are introduced to the bag of the tree decomposition, we add the respective bag-induced clauses to the formulae stored in the NSF. If a variable is removed from the bag it is assigned either $\top$ or $\bot$, representing the step of *expansion* in traditional expansion-based QBF solvers: the NSF keeps track of the quantifier block the removed variable occurred in and stores different formulae resulting from the variable assignment. At the root node, the whole instance was taken into account and the problem is decided by finally evaluating the quantifiers.

4.1. Data structure

We define *nested sets of formulae* (NSFs) where the innermost sets contain *reduced ordered binary decision diagrams* (BDDs) [5]. 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. Nestings will be used to differentiate between quantifier blocks, and BDDs store parts of the QBF matrix.

Definition 4.1. A *nested set of formulae (NSF)* of depth k 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 $d - 1$.

For a QBF $\mathsf{Q}.\psi$ with $\mathsf{Q} = Q_1X_1 \dots Q_kX_k$ 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, $block_{\mathsf{Q}}(M) = k - depth(M) + 1$ is the block of M, and $quantifier_{\mathsf{Q}}(M) = Q_{block_{\mathsf{Q}}(M)}$ (for $block_{\mathsf{Q}}(M) \leq k$). 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 the propositional formula 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 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 4.2. 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 QBF (with respect to the TD), instead of representing the whole QBF instance at once.

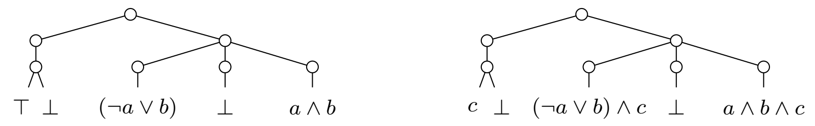

Figure 2. Tree representation of example NSF N and $N[B/B \land c]$ applied to N (each BDD B is replaced with $B \land c$).

Definition 4.3. For a PCNF QBF $\mathsf{Q}.\psi$ with k quantifier blocks and a tree decomposition $\mathcal{T}$ of ψ, for a node $t \in \mathcal{T}$ a *nested set of formulae* (NSF) N_t is of depth k and each BDD at depth 0 represents a propositional formula over variables in $bag_{\mathcal{T}}(t)$.

Internal elements of the NSF have quantifier semantics, as we will show later. Opposed to the similar concept of quantifier trees [34], 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 [35]. Our NSFs mitigate this by only storing parts of the QBF's CNF in the BDDs.

4.2. Main Procedure

In the following we present our expansion based algorithm for deciding satisfiability of QBFs and provide intuitive arguments for its correctness. The proof is given in Appendix A. Algorithm 1 illustrates the recursive procedure for the post-order traversal of the tree decomposition and computing the partial solution candidates. It is called with the root node of the tree decomposition and returns an NSF that represents the overall solution.

In leaf nodes, an NSF of depth k (i.e., the number of quantifier blocks in the QBF instance) 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 contents of the bag). Removed variables are handled by "splitting" the NSF. Procedure $split(N, x)$ (see Algorithm 2) implements a variant of locally restricted expansion: at the block 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 all occurrences of x in the BDDs are removed. This guarantees that the size of each BDD is bounded by the size of the bag. Furthermore, since (reduced ordered) BDDs are canonical and due to set semantics of NSFs, the overall resulting NSF size is bounded by the size of the bag and depth. Removal of variable x from the BDDs is admissible due to the connectedness property of the tree decomposition: x will never reappear somewhere upwards the tree decomposition, and therefore all clauses containing x were already considered. After splitting, the clauses associated with the current node are added to the BDDs of the NSF 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 preserved.

So far, quantifiers were not taken into account. In the basic algorithm they are only considered

Algorithm 1: $solve(t)$

```
Input  : A tree decomposition node $t$
Output: An NSF with partial solution candidates for $t$
if $isLeaf(t)$ then $N := init(k, clauses(t))$
if $isExchange(t)$ then
    $N := solve(firstChild(t))$
    for $x \in removed(t)$ do $N := split(N, x)$
    $N := N[B/B \wedge clauses(t)]$
if $isJoin(t)$ then
    $N := solve(firstChild(t))$
    while $hasNextChild(t)$ do
        $M := solve(nextChild(t))$
        $N := join(N, M)$
    end
if $isRoot(t)$ then $N := evaluateQ(t, N)$
return $N$
```

Algorithm 2: $split(N, x)$

```
Input  : An NSF $N$ and a variable $x$
Output: An NSF split at $block(x)$ w.r.t. assignments to $x$
if $block(N) = block(x)$ then return $\{M[B/B[x/\bot]], M[B/B[x/\top]] \mid M \in N\}$
else return $\{split(M, x) \mid M \in N\}$
```

Algorithm 3: $join(N_1, N_2)$

```
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 $\{join(M_1, M_2) \mid M_1 \in N_1, M_2 \in N_2\}$
```

in the root node r of the tree decomposition, 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 [22], 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). Thus, this procedure finally returns a single BDD B without variables. If $B \equiv \bot$, the QBF is invalid, otherwise it is valid.

Algorithm 4: $evaluateQ(t, N)$

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

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

Example 4.4. Figure 3 shows the NSFs computed at the tree decomposition nodes of our running example (without quantifier evaluation at the root node). In t_1, an NSF of depth 3 is initialized with $(\neg b \vee f)$, i.e., the clause induced by $bag(t_1)$. In t_2, variable f is removed. Hence the NSF is split at $block(f) = 3$, once by setting f to $\bot$ (left NSF branch) yielding $\neg b$, and once by $\top$ (right branch), yielding $\top$. In t_3, clause $(\neg a \vee b)$ is added to these BDDs via conjunction, giving $\{\{\{\neg a \wedge \neg b, (\neg a \vee b)\}\}\}$. In t_4, b is removed and c, e are introduced. The NSF is split at $block(b) = 1$. Additionally, induced clause $(a \vee c \vee e)$ is conjoined with the BDDs. The algorithm proceeds similarly for nodes t_5 and t_6. In t_7, the NSFs are joined. For instance, the leftmost branches in t_4 and t_6 are joined by conjunction of $\neg a \wedge (c \vee e)$ and $\neg e \wedge (a \vee c)$, yielding $\neg a \wedge \neg e \wedge c$. The second branch in t_7 stems from the leftmost branch in t_4 joined with the right branch in t_6. Joining proceeds for all branches in the NSFs, thereby taking into account the nestings in the NSFs. Figure 4 shows the NSF N in root node t_7 together with the BDDs obtained recursively when applying $evaluateQ(t_7, N)$. The procedure returns $\top$, as the QBF from Example 3.2 is valid.

4.3. Dependency Schemes

Quantifiers 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 [36] (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., [6]). Let $Q.\psi$ be a PCNF QBF with k quantifier blocks 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: $block(x) < k$;
2. simple: $block(x) < block(y)$; and
3. standard: $block(x) < block(y)$, $quantifier(x) \neq quantifier(y)$ and there is an X-path from x to y for some $X \subseteq \{z \mid z \in X_i, block(x) < i \leq k,\ quantifier(z) = \exists\}$. An X-path is a

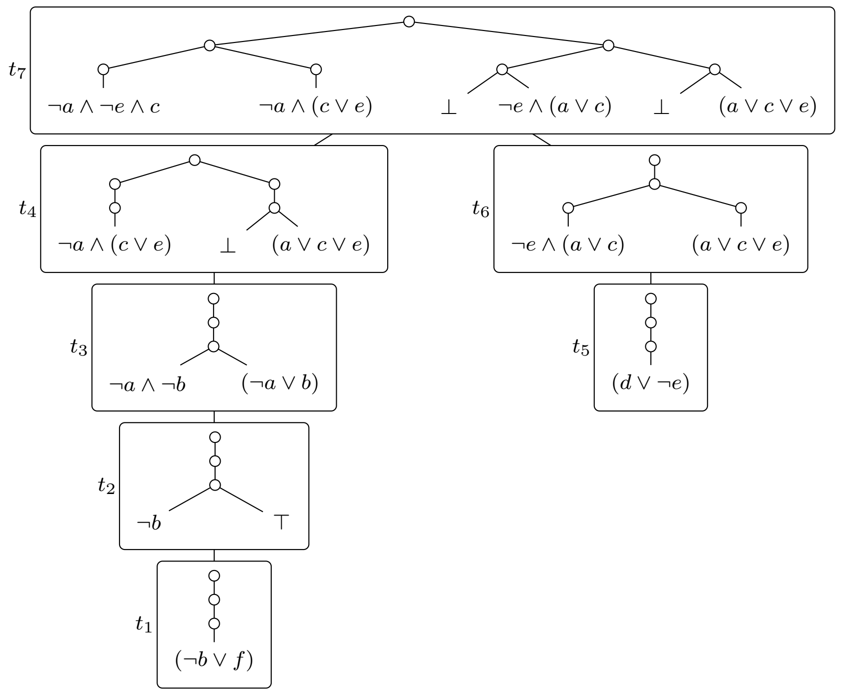

Figure 3. NSFs at the decomposition nodes of the running example.

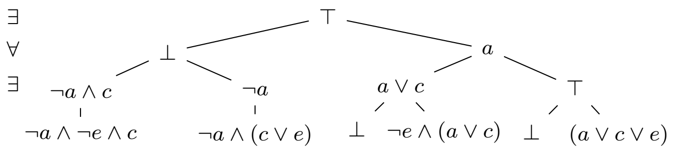

Figure 4. (Intermediate) results for $\mathit{evaluateQ}(t_7, N)$ executed on the NSF of root node t_7.

sequence $c_1, \dots, c_l$ 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 [37] for details).

We denote by $\mathit{dependent}^S_{Q.\psi}(x) = \{y \mid (x,y) \in D^S_{Q.\psi}\}$ the set of variables that are dependent on x in $Q.\psi$ w.r.t. S. Towards our adapted algorithm, for a tree decomposition node t of $\mathcal{T}$, we recursively define by $\mathit{removedSub}_{\mathcal{T}}(t) = \mathit{removed}_{\mathcal{T}}(t) \cup \bigcup_{t' \in \mathit{children}_{\mathcal{T}}(t)} \mathit{removedSub}_{\mathcal{T}}(t')$ the set of removed variables in the subtree of $\mathcal{T}$ rooted at t. Let $\mathit{removedBelow}_{\mathcal{T}}(t) = \mathit{removedSub}_{\mathcal{T}}(t) \setminus \mathit{removed}_{\mathcal{T}}(t)$ be the variables removed below t in $\mathcal{T}$. In Algorithm 1, $\mathit{split}(M, x)$ is replaced with S-$\mathit{dependentSplit}(t, N, x)$ (see Algorithm 5). If 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 4.5. The NSF at node t_2 of Figure 3 reduces to $\{\{\{\top\}\}\}$ (for all considered dependency schemes). Furthermore, we have $D^{\mathit{standard}}_{Q.\psi} = \{(a,c),(a,d),(c,e),(d,e)\}$. Since $\mathit{dependent}(b) =$

Algorithm 5: $S\text{-}dependentSplit(n, N, x)$

```
Input  : Tree decomposition node $n$, NSF $N$, variable $x$
Output: An NSF with abstracted or split $x$
if $dependent^S(x) \subseteq removedBelow(n)$ then
    if $quantifier(x) = \exists$ then return $N[B/\exists x B]$
    if $quantifier(x) = \forall$ then return $N[B/\forall x B]$
else return $split(N, x)$
```

$\{\}$, b can be existentially abstracted in t_4. However, in t_6, d must be split, since $dependent(d) = \{e\} \not\subseteq removedBelow(t_6) = \{\}$.

We remark that for all considered dependency schemes variables at the innermost quantifier block 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 6) variables in the second quantifier block are dependent on those in the first block. In practice, for 2-QBFs we thus always apply the easily computable *naive* dependency scheme. For other instances *standard* turned out to be superior to *simple* and *naive*.

5. Towards Efficiency in Practice

Here we discuss several refinements for our algorithm that are necessary in order to make it useful in practice.

Clause splitting. Given a QBF $Q.\psi$, we construct a tree decomposition of width w for the primal graph of ψ. Due to Conditions 2 and 3 of Definition 3.3, $w \geq \max_{c \in \psi}|c| - 1$ 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.

Tree decomposition selection. It was shown that tree decomposition characteristics besides width play a crucial role in practice [38]. In 2-QBF instances usually most computational effort is required for joining the NSFs. We consider the number of children in join nodes $jNodes(\mathcal{T})$ which is given as

$$joinChildCount(\mathcal{T}) = \sum_{j \in jNodes(\mathcal{T})} |children_{\mathcal{T}}(j)|$$

Algorithm 6: $removeRedundant(N)$

```
Input  : An NSF $N$
Output: An NSF without supersets

if $depth(N) > 1$ then
    for $M \in N$ do $M := removeRedundant(M)$
    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
    for $M_1, M_2 \in N$ and $M_1 \neq M_2$ do
        if $quantifier(N) = \exists$ and $M_1 \vee M_2 = M_1$ then $N := N \setminus \{M_2\}$
        if $quantifier(N) = \forall$ and $M_1 \wedge M_2 = M_1$ then $N := N \setminus \{M_2\}$
    end
return $N$
```

Additionally, we consider the following tree decomposition characteristic. Variable dependencies can be exploited more efficiently if the variables are removed in the TD from the innermost to the outermost quantifier block. For a tree decomposition node t and block b, let $removedBelowBlock_{\mathcal{T},Q}(t, b) = \{r \mid r \in removedBelow_{\mathcal{T}}(t)$ and $block_{\mathsf{Q}}(r) < b\}$. Then,

$$removalBelow(\mathcal{T}, \mathsf{Q}) = \sum_{t \in T} \sum_{r \in removed_{\mathcal{T}}(t)} |removedBelowBlock_{\mathcal{T},\mathsf{Q}}(t, block(r))|$$

We construct several tree decompositions (using the min-fill heuristics [33]) and then select the one minimizing $joinChildCount(\mathcal{T})$ (for 2-QBFs) or $removalBelow(\mathcal{T}, \mathsf{Q})$ (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 [11]), or if two NSFs in the same nesting are in a subset relation. Algorithm 6 gives the pseudo-code for removing unnecessary elements[3]. 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 5.1. Figure 5 shows an NSF N before and after $removeRedundant(N)$. For instance, consider the leftmost branch of the NSF at depth 1, i.e. $N_1 = \{\bot, \neg a\}$. Since $quantifier(N_1) = \exists$ 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.

[3]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 block (nesting).

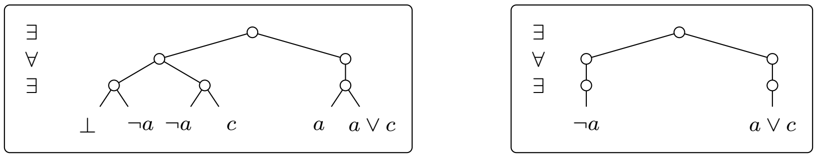

Figure 5. Example NSF before (left) and after (right) compression.

Intermediate unsatisfiability checks. Procedure $evaluateQ(t, N)$ can be applied to any NSF during the tree decomposition traversal. If it returns $\bot$, the QBF is invalid. However, if it returns $\top$, the QBF might still be invalid due to clauses that are encountered later in the traversal. In our setting, the overhead for these checks is negligible. Additionally, in order to remove unnecessary nestings, $evaluateQ(t, N)$ can be adapted as follows: if the evaluation of quantifiers yields a BDD that represents $\bot$ anywhere during the recursive NSF traversal, the respective NSF substructure is replaced by an NSF of the same depth that only contains a single BDD $\bot$.

Estimated NSF size (removal caching). For a node t of decomposition $\mathcal{T}$, let $sizeNSF(t)$ be the number of BDDs in the NSF N computed at node t, and $maxSizeBDD(t)$ 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. $sizeNSF(t)$ 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 $maxSizeBDD(t)$ (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 worst-case 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 [39]. Since finding an optimal variable ordering is in general NP-hard [5], BDD-internal heuristics for finding such a good ordering can be used. For our purposes, we initialize the ordering with the occurrence of variables in the instance (which usually implies that the ordering corresponds to that of the QBF prefix), and apply dynamic reordering during the computation via lazy sifting [40].

Example 5.2. Solving-related optimizations in combination with dependency schemes are illustrated in Figure 6. It shows the NSFs computed at the tree decomposition nodes with the following optimizations enabled: the naive dependency scheme is used (its application is marked in the figure with $D^{naive}_{Q.\psi}$), compression is applied by removing subset-related redundancies (marked with $\subset$) and unsatisfiable NSFs (identified during quantifier evaluation) are replaced with an NSF containing $\bot$ (marked with $evalQ$). The elements removed due to our optimizations are denoted in gray and with dashed lines. Naturally, these simplifications propagate to NSFs upwards in the tree decomposition and thus their size is also reduced. Compared to Figure 3, observe that due to the naive dependency scheme, the sets at depth 1 of the nestings only contain a single BDD. For instance, in t_2 BDD $\neg b$ is not explicitly computed due to $D^{naive}_{Q.\psi}$. Furthermore, by considering dependency scheme integra-

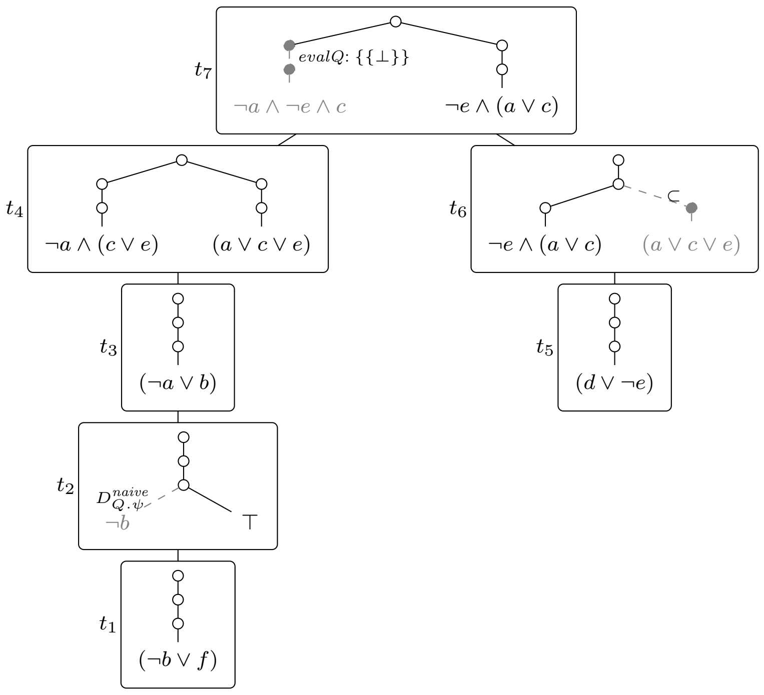

Figure 6. NSFs at the decomposition nodes of the running example, including optimizations (marked in gray, with dashed lines).

tion, comparison of models in BDDs is shifted to NSFs of depth 2. Consider the NSF in t_6. Originally, we have $\{\{\{\neg e \wedge (a \vee c)\}, \{(a \vee c \vee e)\}\}\}$. Since the quantifier at depth 2 is universal, and $(\neg e \wedge (a \vee c)) \wedge (a \vee c \vee e)$ corresponds to $(\neg e \wedge (a \vee c))$, the right branch in this NSF can be removed, yielding $\{\{\{\neg e \wedge (a \vee c)\}\}\}$. Now, consider node t_7, and the NSF at level 2 on the left branch, which is $\{\{\neg a \wedge \neg e \wedge c\}\}$. $evaluateQ(t_7, \{\{\neg a \wedge \neg e \wedge c\}\})$ returns $\bot$, hence this NSF is replaced by $\{\{\bot\}\}$.

6. Experimental Evaluation

The presented algorithms are implemented in the *dynQBF* system, which is freely available (including source code and binaries) at `https://github.com/gcharwat/dynqbf`. dynQBF relies on HTD [41] for tree decomposition construction, CUDD [40] for BDD management, and optionally DepQBF [19] for computing the standard dependency scheme.

We first study the impact of dependency schemes and optimizations as described in Sections 4.3 and 5. Then we compare dynQBF to state-of-the-art systems that successfully participated in the official 2016 and 2017 QBF competitions. All experiments were run once (per solver or configuration) on each instance using a fixed seed. Tests were performed on a single core of an Intel Xeon E5-2637 (3.5GHz) running Debian 8.3, with a memory limit of 16 GB. For our analysis of optimizations we set the time limit to 6 minutes. In the system comparison each run was limited to 10 minutes.

	All instances			Solved			
Configuration	atoms	clauses	width	#	atoms	clauses	width
default	2552	8044	56	142	1984	6283	45
w/o TD selection	2552	8044	56	135	1902	7101	46
w/o cl. splitting	2535	8028	172	124	1818	5943	48
w/o var. reordering	2552	8044	56	84	1656	5717	28
w/o rem. cache	2552	8044	56	70	1109	3525	24
single TD node	2552	8044	2551	40	390	1236	389

Table 1. 2-QBF'10: dynQBF variants comparison details. For each configuration, the table shows the average number of atoms, clauses, and width for all 200 instances, and for the solved instances. # gives the number of solved instances.

	All instances				Solved				
Configuration	atoms	clauses	blocks	width	#	atoms	clauses	blocks	width
default	21457	78019	14.7	167	92	3819	11062	12.8	45
w/o cl. splitting	21147	77710	14.7	312	92	3855	10913	12.8	48
w/o TD selection	21457	78019	14.7	194	90	3910	11345	12.9	47
naive dep. scheme	21457	78019	14.7	167	67	5476	14728	5.7	51
w/o var. reordering	21457	78019	14.7	167	61	3937	12501	18.9	40
w/o rem. cache	21457	78019	14.7	167	24	6107	17918	8.5	34
single TD node	21457	78019	14.7	21456	15	635	2124	24.1	634

Table 2. PCNF'14: dynQBF variants comparison details. For each configuration, the table shows the average number of atoms, clauses, quantifier blocks, and width for all 305 instances, and for the solved instances. # gives the number of solved instances.

6.1. Impact of Optimizations

In this section we analyze the dynQBF-internal optimization strategies with respect to their effect on the number of solved instances and overall runtime. The goal is to illustrate the impact of optimizations and to identify approaches that may also be beneficial for tree decomposition-based algorithms in other problem domains. We considered the following two data sets: *2-QBF'10*: This is the 2-QBF data set[4] used in the QBF Evaluation 2010, which consists of 200 instances. *PCNF'14*: This data set contains the PCNF instances[5] of the QBF Gallery 2014 competition, comprising of 345 instances. We compare dynQBF (version v1.0.0-final) with variants where in each variant one particular optimization is disabled. The following configurations are considered.

- *default*: All optimizations are enabled, and configured as released in dynQBF 1.0.0-final.
- *w/o TD selection*: Tree decomposition selection is disabled (i.e., only one TD is constructed).

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

[5]Available at `http://www.kr.tuwien.ac.at/events/qbfgallery2013/benchmarks/eval2012r2.tar.7z`.

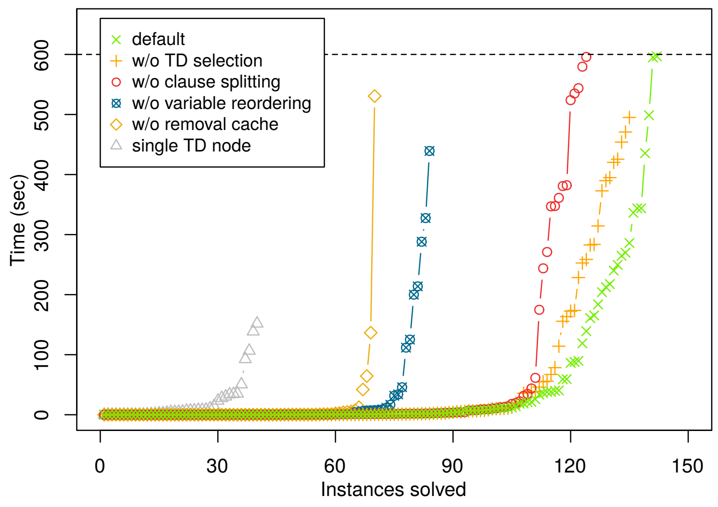

Figure 7. 2-QBF'10: dynQBF variants comparison. The cactus plot shows dynQBF with all optimizations enabled (*default*), and single features disabled.

- *w/o clause splitting*: Splitting of clauses with more than 30 literals is disabled.
- *naive dependency scheme* (applies only to the PCNF'14 instances): the naive dependency scheme is applied instead of the standard dependency scheme.
- *w/o variable reordering*: Dynamic BDD variable reordering is disabled.
- *w/o removal cache*: The removal cache is disabled. Due to the concepts underlying dynQBF, the naive dependency scheme is used for all instances.
- *single TD node*: A trivial tree decomposition with a single node containing all variables is used.

Tables 1 and 2 summarize details on the instances under consideration. We report the average number of atoms, clauses, and the measured width for all instances, as well as for solved instances. For the latter also the number of instances solved is given. Furthermore, for the PCNF'14 instances (Table 2) we report the average number of quantifier blocks. "All instances" is measured after clause splitting (if applied). Furthermore, we remark that the average width is aggregated only over instances that could be decomposed within the time limit.

Figures 7 and 8 contain cactus plots of the obtained results: for each configuration, solved instances are sorted by the required runtime.

Analysis. We highlight here the most important observations from the obtained data.

w/o TD selection: Despite optimizing *removedBelow* for instances with more than one quantifier alternation (and *joinChildCount* for the other instances), in the PCNF'14 data set we also observed an overall reduction in width when tree decomposition selection is enabled.

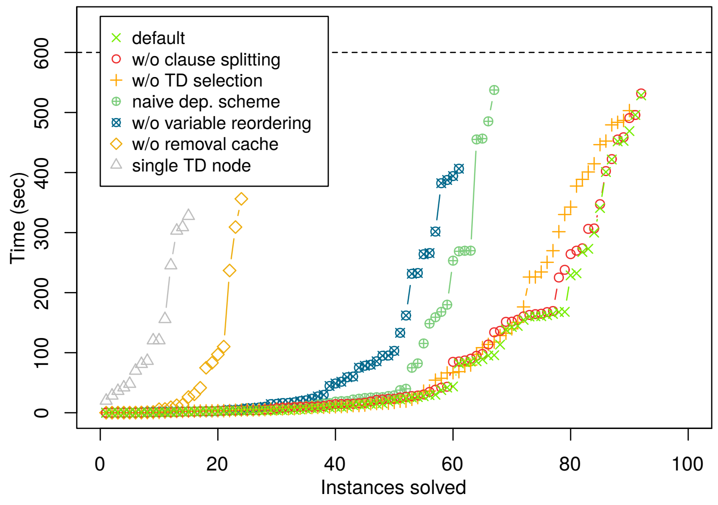

Figure 8. PCNF'14: dynQBF variants comparison. The cactus plot shows dynQBF with all optimizations enabled (*default*), and single features disabled.

w/o clause splitting: In both data sets, clause splitting only marginally increases the average number of atoms and clauses. However, the average width is drastically reduced in both cases. For the 2-QBF'10 data set we observe that clause splitting has a larger impact on the overall performance, while in the PCNF'14 data set the performance is almost the same.

naive dependency scheme: In the PCNF'14 data set, 65 out of 67 instances were also solved in the *default* configuration. One can see that integration of the standard dependency scheme improves the performance on instances with a larger number of quantifier blocks.

w/o variable reordering: From the results it becomes apparent that dynamic variable reordering is an important ingredient for keeping the BDDs compact, despite the additional computational effort for reordering. While we observe again a high average number of quantifier blocks for solved instances in the PCNF'14 data set, 54 out of 61 instances where also solved in the *default* configuration.

w/o removal cache: Without caching variables have to be removed immediately from the BDDs, potentially yielding larger NSFs (despite smaller BDDs). Additionally, for the PCNF'14 instances, the naive dependency scheme is applied (due to implementation restrictions), which could, as a side effect, explain the lower average number of quantifier blocks in solved instances. Overall, from results it becomes immanent that removal caching is beneficial for performance in our implementation.

single TD node: The large number of variables to be handled at once (which is also reflected by the reported width) explains the low number of solved instances in this setting. The BDD of the NSF computed at the single tree decomposition node stores the whole QBF matrix before the variables are removed by quantified abstraction. Results show that our decomposition-based approach is clearly beneficial to simple expansion (as applied when only using a single tree decomposition node). Besides that, interestingly, in the PCNF'14 data set we measure an average of 24.1 quantifier blocks for solved instances. However, all of these instances were also solved in the *default* configuration.

6.2. System Comparison

We conducted experiments along the lines of the official 2016 and 2017 QBF competitions (QBFEval'16 [7], QBFEval'17 (`http://www.qbflib.org/qbfeval17.php`)). We considered systems that participated in the respective competition based on public availability and good ranking in the competition. Instances under consideration are from the 2-QBF and PCNF competition tracks. Since preprocessing oftentimes influences performance, we evaluated the plain systems (i.e., without explicit, pipelined preprocessing) both on the original instances as well as on the instances resulting from preprocessing with Bloqqer 37 [42]. Additionally, we studied the impact of width on the performance of the solvers.

In the following, we report on the number of solved, solved valid ($\top$) and invalid ($\perp$) instances. The stated time is the accumulated user time in thousands of seconds (K), including a penalty of 10 minutes per instance that is not solved. Additionally, we give the number of instances uniquely solved by a single system (U). dynQBF was run with one random, fixed seed. However, the performance is influenced by the heuristically constructed tree decomposition. 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.

6.2.1. QBFEval'16 Benchmark Setting

In this setting we compare the following systems: the 2-QBF solver AReQS (20160702) [12]; the search-based solvers DepQBF 5.0.1 [19] and GhostQ (CEGAR 2016) [13]; the expansion-based system RAReQS 1.1 [13]; CAQE 2 [43] that relies on variable level-based decomposition; as well as Questo 1.0 [44] and QSTS (2016) [45] that use SAT solvers. dynQBF is tested in version 1.0.0. We consider the 305 2-QBF'16[6] and 825 PCNF'16[7] competition instances.

2-QBF'16. Table 3 shows that in this setting our system is competitive on 2-QBF instances. Regarding the original instances, only the 2-QBF solver AReQS performed better. When considering 10 different seeds, *Bo10* indicates that there is still potential for our feature-based tree decomposition selection. With preprocessing, 130 out of 305 instances were directly solved by Bloqqer. Qesto and RAReQS benefited the most from preprocessing. Overall, dynQBF is particularly strong on valid instances. Additionally, we report on a large number of uniquely solved instances. For the original data set, they mostly stem from QBF encodings for ranking functions ("`rankfunc*`"). Interestingly, after preprocessing we observed that 43 instances from the area of formal verification ("`stmt*`") were uniquely solved.

To study the influence of treewidth on solving, we considered the 175 preprocessed instances that were not solved directly by Bloqqer. Since computing the exact treewidth is infeasible, we used HTD [41] to heuristically obtain an over-approximation. In Table 4, the data set is partitioned based on the computed width w. Here, the influence of the width on the performance of dynQBF becomes apparent.

[6]Available at `http://www.qbflib.org/TS2016/Dataset_3.tar.gz`.
[7]Available at `http://www.qbflib.org/TS2016/Dataset_1.tar.gz`.

Table 3. 2-QBF'16: System comparison for the original (left) and preprocessed (right) instances.

2-QBF'16 (original)						2-QBF'16 (preprocessed)					
System	Solved	Time	$\top$	$\perp$	U	System	Solved	Time	$\top$	$\perp$	U
AReQS	181	79K	126	55	0	Qesto	236	50K	160	76	0
dynQBF	170	86K	140	30	13	RAReQS	232	51K	161	71	1
GhostQ	156	98K	108	48	0	**dynQBF**	221	53K	172	49	43
DepQBF	120	116K	55	65	11	DepQBF	221	56K	143	78	1
QSTS	97	132K	60	37	8	QSTS	220	58K	162	58	2
Qesto	78	140K	47	31	2	CAQE	204	65K	153	51	0
RAReQS	70	142K	44	26	0	AReQS	202	66K	141	61	0
CAQE	57	151K	35	22	1	GhostQ	151	95K	123	28	0
dynQBF Bo10	203	68K	154	49		dynQBF Bo10	225	49K	172	53	
dynQBF Ao10	169.9	86K	141.5	28.4		dynQBF Ao10	221.2	53K	171.0	50.2	

Table 4. 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

PCNF'16. Results for the PCNF'16 data set are summarized in Table 5. The obtained data confirms that dynQBF is indeed sensitive to the number of quantifier blocks (k). For the original instances we measured an average k of 17, and 14.8 for instances solved by dynQBF. 75 instances have 2 (or less) quantifier blocks, of which dynQBF solved the most instances (55). Of the 391 instances with $k = 3$, dynQBF solved 142 instances, while the best solver here is GhostQ with 299 instances. Of the 359 instances with $k > 3$, dynQBF solved 168 instances, but GhostQ solved 256 instances. With preprocessing, 341 instances were solved by Bloqqer. Interestingly, all solvers except GhostQ benefited from preprocessing. Regarding the impact of quantifiers on the performance of dynQBF we obtained a similar picture as for the original instances. Overall, we again observed several instances uniquely solved by dynQBF. As in the 2-QBF setting, we considered the width w of the preprocessed, non-trivial instances. Table 6 again shows that dynQBF performed 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.

Table 5. PCNF'16: System comparison for the original (left) and preprocessed (right) instances.

Original						Preprocessed					
System	Solved	Time	$\top$	$\bot$	U	System	Solved	Time	$\top$	$\bot$	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 6. 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

6.2.2. QBFEval'17 Benchmark Setting

To evaluate whether dynQBF performs similarly in further settings, and to compare how it competes with most recent QBF solvers, we additionally conducted experiments along the lines of the 2017 QBF competition. In this evaluation we considered the following systems, with potentially updated binaries compared to the 2016 competition. In case no version information is available, the download date is given in brackets: For QSTS 2016 [45] and RAReQS 1.1 [13] the same binaries were used as in the QBFEval'16 comparison. For GhostQ (CEGAR, 2017-07-26) [13], CAQE 2017 [43] and DepQBF 6.03 [19] updated binaries were available. Additionally, we considered AIGSolve (2017-09-27) [46] which is based on And-Inverter-Graphs; QELL (2017-09-28) [47] that implements clause learning and relies on SAT solving techniques; and Qute (2017-09-26) [48] that is based on dependency learning. dynQBF is tested in version 1.0.2. Compared to version 1.0.0 as used in the 2016 comparison, it includes minor improvements and optimized default parameters. The 2-QBF'17 dataset contains 384 instances, and the PCNF'17 dataset comprises of 523 instances[8].

[8]Both datasets are available at `http://www.qbflib.org/eval17.zip`.

Table 7. 2-QBF’17: System comparison for the original (left) and preprocessed (right) instances.

2-QBF’17 (original)

System	Solved	Time	$\top$	$\bot$	U
GhostQ	243	93K	138	105	25
AIGSolve	204	116K	141	63	22
dynQBF	115	169K	80	35	2
QELL	102	175K	40	62	1
QSTS	79	192K	37	42	2
DepQBF	74	191K	21	53	3
CAQE	56	200K	15	41	4
RAReQS	56	200K	26	30	0
Qute	20	219K	4	16	0
dynQBF Bo10	159	147K	104	55	
dynQBF Ao10	114.8	169K	82.2	32.6	

2-QBF’17 (preprocessed)

System	Solved	Time	$\top$	$\bot$	U
CAQE	231	100K	140	91	1
QELL	231	104K	136	95	3
QSTS	231	105K	151	80	8
RAReQS	224	105K	135	89	2
dynQBF	199	115K	136	63	39
DepQBF	188	123K	108	80	0
AIGSolve	158	139K	106	52	2
Qute	157	137K	91	66	1
GhostQ	143	149K	93	50	0
dynQBF Bo10	209	107K	139	70	
dynQBF Ao10	198.9	115K	134	64.9	

Table 8. 2-QBF’17 (preprocessed, non-trivial): Influence of width w on the system performance.

$w \leq 80$ (82 instances)			$w > 80$ (182 instances)		
System	Solved	Time	System	Solved	Time
dynQBF	65	13K	QSTS	100	62K
CAQE	28	34K	QELL	85	66K
QELL	26	37K	RAReQS	84	65K
DepQBF	23	37K	CAQE	83	66K
RAReQS	20	40K	DepQBF	45	86K
Qute	17	40K	AIGSolve	26	95K
AIGSolve	12	43K	GhostQ	20	101K
QSTS	11	43K	Qute	20	98K
GhostQ	3	48K	**dynQBF**	14	102K

2-QBF’17. Table 7 summarizes our results for the QBFEval’17 instances. The updated version of GhostQ performed particularly well on the original dataset. One reason may be that GhostQ implements internal preprocessing: the 2017 version features reverse engineering of the Plaisted-Greenbaum transformation in order to reconstruct structure. The latter is often applied in order to transform instances to PCNF [49]. The second-ranked solver AIGSolve internally implements standard techniques for QBF preprocessing, such as unit propagation, subsumption checking, for-all reduction and equivalence reduction [46], which is obviously beneficial for solving. Regarding the preprocessed dataset, Bloqqer directly solved 120 out of 384 instances. Except for GhostQ and AIGSolve, all compared solvers benefited from preprocessing by Bloqqer. It may be the case that those two systems are tailored towards their own preprocessing, and that required structural information is disguised by Bloqqer. Similar to our results for the QBFEval’16 benchmark setting, dynQBF per-

Table 9. PCNF'17: System comparison for the original (left) and preprocessed (right) instances.

PCNF'17 (original)						PCNF'17 (preprocessed)					
System	Solved	Time	$\top$	$\bot$	U	System	Solved	Time	$\top$	$\bot$	U
AIGSolve	245	178K	75	170	39	RAReQS	241	180K	69	172	5
QELL	188	210K	53	135	1	CAQE	231	185K	76	155	4
GhostQ	183	214K	66	117	11	QELL	230	184K	67	163	1
CAQE	160	227K	38	122	3	AIGSolve	208	197K	64	144	21
RAReQS	156	228K	28	128	3	QSTS	195	204K	56	139	5
QSTS	140	235K	33	107	1	DepQBF	173	218K	61	112	4
DepQBF	135	240K	40	95	13	GhostQ	153	228K	58	95	0
Qute	91	263K	12	79	0	Qute	140	233K	40	100	1
dynQBF	62	281K	31	31	2	**dynQBF**	137	236K	51	86	20
dynQBF Bo10	88	270K	43	45		dynQBF Bo10	154	226K	58	96	
dynQBF Ao10	61.7	281K	30.8	30.9		dynQBF Ao10	135.3	236K	50.4	84.9	

Table 10. PCNF'17 (preprocessed, non-trivial): Influence of width w on the system performance.

$w \leq 80$ (110 instances)			$w > 80$ (339 instances)		
System	Solved	Time	System	Solved	Time
CAQE	49	38K	RAReQS	122	139K
dynQBF	47	41K	QELL	110	144K
QELL	46	41K	CAQE	108	147K
RAReQS	45	41K	QSTS	93	154K
AIGSolve	45	42K	AIGSolve	89	155K
GhostQ	29	50K	DepQBF	71	168K
DepQBF	28	50K	GhostQ	50	177K
QSTS	28	50K	Qute	43	179K
Qute	23	53K	**dynQBF**	16	195K

formed very well on 2-QBF instances. Additionally, for the preprocessed instances we again report a very large number of uniquely solved instances. Finally, Table 8 confirms the impact of width on the solving performance of dynQBF.

PCNF'17. Results for the PCNF'17 dataset are given in Table 7. Overall dynQBF solved less instances than the best-performing systems that participated in QBFEval'17. On the original instances, the average number of quantifier blocks (k) is 27.2 (showing an increase compared to PCNF'16, where $avg(k) = 17$). The 62 instances solved by dynQBF have an average k of 7.2. Preprocessing by Bloqqer directly solved 74 instances. The remaining instances have 9.8 blocks on average. Here, instances solved by dynQBF comprise of 8.2 quantifier blocks on average. For the preprocessed dataset, similar to the PCNF'16 setting, we observed that several instances are uniquely solved by dynQBF. Our analysis with respect to width (see Table 10) confirms that dynQBF again performs well on instances of low-to-medium width.

Official QBFEval'17 results. Besides the benchmark results reported in this section, we remark that dynQBF additionally participated in the official QBFEval'17 competition. There it was submitted in combination with preprocessors Bloqqer [42] and HQSPre [50]. In the official 2-QBF'17 track, dynQBF (plus preprocessing) was ranked 8 out of 29 participants, and in the PCNF'17 track it achieved the 13th place out of 30 participants (see `http://www.qbflib.org/event_page.php?year=2017`).

7. Conclusion

In this work we presented 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 tree decomposition and by using BDDs. We showed how dependency schemes can be used within our algorithm, and discussed entry points for heuristic optimizations of our technique. We studied the impact of optimizations on our core algorithm and conducted a thorough experimental analysis along the lines of QBFEval'16 and QBFEval'17. The latter shows that our approach is competitive for 2-QBF instances, an observation that is underpinned by a recent study on the impact of quantifier alternations [51]. Furthermore, dynQBF is competitive 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.

One important aspect of future work is the development of dedicated preprocessing techniques. While Bloqqer already improves the performance of dynQBF, we believe that our approach could greatly benefit from treewidth-tailored preprocessing, as well as from techniques that take structural aspects of the tree decomposition into account. Since our approach is very sensitive to the tree decomposition used, machine learning techniques (see, for instance, [38]) could be applied to automatically choose a suitable tree decomposition based on the given QBF instance. Regarding solving, more restrictive dependency schemes could improve performance on instances with more than two quantifier blocks. Additionally, we need a better understanding of the relation between different variable orderings in the BDDs and the shape of the used tree decomposition.

Finally, we believe that dynQBF would be a valuable addition to QBF portfolio solving. There, the width of the heuristically computed TD or the number of quantifier alternations could be used to decide which solver shall be used. Towards this, in the recent QBFEval'18 competition dynQBF participated as part of *Pre-DynDep*. On instances that exhibit low-to-medium treewidth, dynQBF is called for solving. Otherwise, DepQBF is used. In the 2-QBF'18 track the system achieved the 3rd place out of 23 solver configurations[9].

[9]See `http://www.qbflib.org/eval18.html` for preliminary results.

References

[1] Stockmeyer LJ, Meyer AR. Word problems requiring exponential time (preliminary report). In: Proc. TOC. ACM, 1973 pp. 1–9.

[2] Downey RG, Fellows MR. Parameterized Complexity. Monographs in Computer Science. Springer, 1999.

[3] Chen H. Quantified constraint satisfaction and bounded treewidth. In: Proc. ECAI. IOS Press, 2004 pp. 161–165.

[4] Atserias A, Oliva S. Bounded-width QBF is PSPACE-complete. *J. Comput. Syst. Sci.*, 2014. **80**(7):1415–1429.

[5] Bryant RE. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 1986. **100**(8):677–691.

[6] Slivovsky F, Szeider S. Computing resolution-path dependencies in linear time. In: Proc. SAT, volume 7317 of *LNCS*. Springer, 2012 pp. 58–71.

[7] Pulina L. The ninth QBF solvers evaluation - preliminary report. In: Proc. QBF, volume 1719 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016 pp. 1–13.

[8] Charwat G, Woltran S. Expansion-based QBF solving on tree decompositions. In: Proc. RCRA, volume 2011 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017 pp. 16–26.

[9] Marin P, Narizzano M, Pulina L, Tacchella A, Giunchiglia E. An empirical perspective on ten years of QBF solving. In: Proc. RCRA, volume 1451 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015 pp. 62–75.

[10] Ayari A, Basin DA. QUBOS: Deciding quantified Boolean logic using propositional satisfiability solvers. In: Proc. FMCAD, volume 2517 of *LNCS*. Springer, 2002 pp. 187–201.

[11] Biere A. Resolve and expand. In: Proc. SAT, volume 3542 of *LNCS*. Springer, 2004 pp. 59–70.

[12] Janota M, Marques-Silva J. Abstraction-based algorithm for 2QBF. In: Proc. SAT, volume 6695 of *LNCS*. Springer, 2011 pp. 230–244.

[13] Janota M, Klieber W, Marques-Silva J, Clarke EM. Solving QBF with counterexample guided refinement. In: Proc. SAT, volume 7317 of *LNCS*. Springer, 2012 pp. 114–128.

[14] Clarke EM, Grumberg O, Jha S, Lu Y, Veith H. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 2003. **50**(5):752–794.

[15] Cadoli M, Giovanardi A, Schaerf M. An algorithm to evaluate quantified Boolean formulae. In: Proc. IAAI. AAAI Press / The MIT Press, 1998 pp. 262–267.

[16] Zhang L, Malik S. Towards a symmetric treatment of satisfaction and conflicts in quantified Boolean formula evaluation. In: Proc. CP, volume 2470 of *LNCS*. Springer, 2002 pp. 200–215.

[17] Giunchiglia E, Narizzano M, Tacchella A. Clause/term resolution and learning in the evaluation of quantified Boolean formulas. *J. Artif. Intell. Res.*, 2006. **26**:371–416.

[18] Lonsing F, Biere A. DepQBF: A dependency-aware QBF solver. *J. SAT*, 2010. **7**(2-3):71–76.

[19] Lonsing F, Bacchus F, Biere A, Egly U, Seidl M. Enhancing search-based QBF solving by dynamic blocked clause elimination. In: Proc. LPAR, volume 9450 of *LNCS*. Springer, 2015 pp. 418–433.

[20] Klieber W, Sapra S, Gao S, Clarke EM. A Non-prenex, non-clausal QBF solver with game-state learning. In: Proc. SAT, volume 6175 of *LNCS*. Springer, 2010 pp. 128–142.

[21] Bjørner N, Janota M, Klieber W. On conflicts and strategies in QBF. In: Proc. LPAR. EasyChair, 2015 pp. 28–41.

[22] Pan G, Vardi MY. Symbolic decision procedures for QBF. In: Proc. CP, volume 3258 of *LNCS*. Springer, 2004 pp. 453–467.

[23] Jussila T, Sinz C, Biere A. Extended resolution proofs for symbolic SAT solving with quantification. In: Proc. SAT, volume 4121 of *LNCS*. Springer, 2006 pp. 54–60.

[24] Pulina L, Tacchella A. Hard QBF encodings made easy: Dream or reality? In: Proc. AI*IA, volume 5883 of *LNCS*. Springer, 2009 pp. 31–41.

[25] Chen H, Dalmau V. From pebble games to tractability: An ambidextrous consistency algorithm for quantified constraint satisfaction. In: Proc. CSL, volume 3634 of *LNCS*. Springer, 2005 pp. 232–247.

[26] Pulina L, Tacchella A. An empirical study of QBF encodings: From treewidth estimation to useful preprocessing. *Fundam. Inform.*, 2010. **102**(3-4):391–427.

[27] Pulina L, Tacchella A. QuBIS: An (in)complete solver for quantified Boolean formulas. In: Proc. MICAI, volume 5317 of *LNCS*. Springer, 2008 pp. 34–43.

[28] Pulina L, Tacchella A. A structural approach to reasoning with quantified Boolean formulas. In: Proc. IJCAI. AAAI Press, 2009 pp. 596–602.

[29] Robertson N, Seymour PD. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 1984. **36**(1):49–64.

[30] Arnborg S, Corneil DG, Proskurowski A. Complexity of finding embeddings in a k-tree. *SIAM J. Algebra. Discr. Meth.*, 1987. **8**:277–284.

[31] Bodlaender HL. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 1996. **25**(6):1305–1317.

[32] Bodlaender HL, Koster AMCA. Treewidth computations I. Upper bounds. *Inf. Comput.*, 2010. **208**(3):259–275.

[33] Dechter R. Constraint Processing. Morgan Kaufmann, 2003.

[34] Benedetti M. Quantifier trees for QBFs. In: Proc. SAT, volume 3569 of *LNCS*. Springer, 2005 pp. 378–385.

[35] Ferrara A, Pan G, Vardi MY. Treewidth in verification: Local vs. global. In: Proc. LPAR, volume 3835 of *LNCS*. Springer, 2005 pp. 489–503.

[36] Samer M, Szeider S. Backdoor sets of quantified Boolean formulas. *J. Autom. Reasoning*, 2009. **42**(1):77–97.

[37] Lonsing F, Biere A. A compact representation for syntactic dependencies in QBFs. In: Proc. SAT, volume 5584 of *LNCS*. Springer, 2009 pp. 398–411.

[38] Abseher M, Musliu N, Woltran S. Improving the efficiency of dynamic programming on tree decompositions via machine learning. *J. Artif. Intell. Res.*, 2017. **58**:829–858.

[39] Friedman SJ, Supowit KJ. Finding the optimal variable ordering for binary decision diagrams. In: Proc. DAC. ACM, 1987 pp. 348–356.

[40] Somenzi F. CU Decision Diagram package release 3.0.0. Department of Electrical and Computer Engineering, University of Colorado at Boulder, 2015.

[41] Abseher M, Musliu N, Woltran S. htd – A free, open-source framework for tree decompositions and beyond. Technical Report DBAI-TR-2016-96, TU Wien, 2016.

[42] Biere A, Lonsing F, Seidl M. Blocked clause elimination for QBF. In: Proc. CADE, volume 6803 of *LNCS*. Springer, 2011 pp. 101–115.

[43] Rabe MN, Tentrup L. CAQE: A certifying QBF solver. In: Proc. FMCAD. IEEE, 2015 pp. 136–143.

[44] Janota M, Marques-Silva J. Solving QBF by clause selection. In: Proc. IJCAI. AAAI Press, 2015 pp. 325–331.

[45] Bogaerts B, Janhunen T, Tasharrofi S. Solving QBF instances with nested SAT solvers. In: Proc. Beyond NP, volume WS-16-05 of *AAAI Workshops*. AAAI Press, 2016 .

[46] Scholl C, Pigorsch F. The QBF solver AIGSolve. In: Proc. QBF, volume 1719 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016 pp. 55–62.

[47] Tu K, Hsu T, Jiang JR. QELL: QBF reasoning with extended clause learning and levelized SAT solving. In: Proc. SAT, volume 9340 of *LNCS*. Springer, 2015 pp. 343–359.

[48] Peitl T, Slivovsky F, Szeider S. Dependency learning for QBF. In: Proc. SAT, volume 10491 of *LNCS*. Springer, 2017 pp. 298–313.

[49] Goultiaeva A, Bacchus F. Recovering and utilizing partial duality in QBF. In: Proc. SAT, volume 7962 of *LNCS*. Springer, 2013 pp. 83–99.

[50] Wimmer R, Reimer S, Marin P, Becker B. HQSpre - An effective preprocessor for QBF and DQBF. In: Proc. TACAS, volume 10205 of *LNCS*. 2017 pp. 373–390.

[51] Lonsing F, Egly U. Evaluating QBF Solvers: Quantifier Alternations Matter. In: Proc. CP, volume 11008 of *LNCS*. Springer, 2018 pp. 276–294.

A. Proof

Here we show the correctness of Algorithm 1 for deciding QSAT by dynamic programming over tree decompositions. At its core, the proof is based on induction over the tree decomposition by showing the correspondence of NSFs to *expansion trees*. First, in addition to Definition 3.1, we equivalently define closed PCNF QBFs without quantifier blocks as follows:

Definition A.1. In a *PCNF QBF* $\mathsf{Q}.\psi$, Q is the *quantifier prefix* and ψ is a CNF formula, also called the *matrix*. Q is of the form $Q_1 x_1 Q_2 x_2 \dots Q_m x_m$ where $Q_i \in \{\exists, \forall\}$ for $1 \leq i \leq m$, $Q_m = \exists$ and ψ is a formula over variables $X = \{x_1, \dots, x_m\}$. We say that a variable x_i is at *position* i in the QBF.

We remark that such a QBF can be easily transformed into one with k quantifier blocks, where subsequent quantifiers $Q_j x_j \dots Q_k x_k$ ($1 \leq j < k \leq m$) with $Q_j = \dots = Q_k$ are merged into a single block QX where $Q = Q_j$ and $X = \{x_j, \dots, x_k\}$. In the following we will use both syntactic notations interchangeably.

Expansion of a PCNF QBF can be represented in form of an expansion tree.

Definition A.2. Let $\mathsf{Q}.\psi$ with $\mathsf{Q} = Q_1 x_1 Q_2 x_2 \dots Q_m x_m$ be a closed PCNF QBF over variables $X = \{x_1, \dots, x_m\}$ and $Y = \{y_1, \dots, y_o\} \subseteq X$ be the set of variables to be expanded on. An *expansion tree* $E(\mathsf{Q}.\psi, Y) = (V, C, \mathit{ass})$ is a rooted tree with nodes V, arcs C and assignments $\mathit{ass} : C \rightarrow \{\top, \bot, *\}$ (here, $*$ denotes that a variable is not assigned a truth value). Each internal node $v \in V$ at a level i in the tree (for $1 \leq i \leq m$) represents the variable at the same position in Q, denoted by $\mathit{var}(v) = x_i$. Node v is associated with a connective, where $\mathit{conn}(v) = \wedge$ if $Q_i = \forall$ and $\mathit{conn}(v) = \vee$ if $Q_i = \exists$. The respective block in the quantifier prefix is denoted by $\mathit{block}(v)$. Each leaf node is associated with the propositional formula ψ.

The tree contains $m + 1$ levels and has the following structure. For an internal node $v \in V$:

- If $\mathit{var}(v) \notin Y$, v has a single child v' (i.e., $(v, v') \in C$) and $\mathit{ass}((v, v')) = *$.
- If $\mathit{var}(v) \in Y$, v has two children v' and v'' (i.e., $(v, v'), (v, v'') \in C$), and $\mathit{ass}((v, v')) = \top$ and $\mathit{ass}((v, v'')) = \bot$.

Let v_{m+1} be a leaf node and $v_1, \dots, v_m, v_{m+1}$ be the unique path from root node v_1 to v_{m+1} in the tree. For $1 \leq i \leq m$, if $\mathit{ass}((v_i, v_{i+1})) = \top$ ($\bot$) this represents the assignment $[\mathit{var}(v_i)/\top]$ ($[\mathit{var}(v_i)/\bot]$). We denote by $[Y/v_{m+1}]$ the assignment of all variables in Y with respect to v_{m+1}.

An expansion tree is evaluated as follows.

Definition A.3. Let $\mathsf{Q}.\psi$ be a closed PCNF QBF over variables X and $E(\mathsf{Q}.\psi, X) = (V, C, \mathit{ass})$ be the expansion tree of the QBF where all variables are expanded. The *evaluation* $\mathit{eval}(E(\mathsf{Q}.\psi, X))$ is defined as the result of recursively applying evaluation function $e(v)$ on nodes $v \in V$:

- If v is a leaf node, $e(v) = \psi[X/v]$;
- Otherwise, v has two children v' and v'', and $e(v) = e(v') \cdot e(v'')$ where $\cdot = \mathit{conn}(v)$.

The following proposition follows directly from the semantics of QBFs.

Proposition A.4. A closed PCNF QBF formula $\mathsf{Q}.\psi$ over variables X can be decided by constructing its corresponding expansion tree $E(\mathsf{Q}.\psi, X)$ and applying $\mathit{eval}(E(\mathsf{Q}.\psi, X))$.

For reasonably-sized QBFs, constructing and evaluating $E(\mathsf{Q}.\psi, X)$ at once is infeasible in practice. Thus, we first define expansion trees in the context of tree decompositions. Then, we show how expansion trees can be represented equivalently by NSFs, whose size is exponential only in the number of quantifier blocks and the treewidth of the QBF matrix. Recall Definition 3.3 of tree decompositions and Definition 3.4 of weakly normalized tree decompositions. Towards readability, we prove the correctness of our algorithm on *normalized tree decompositions*.

Definition A.5. A tree decomposition $\mathcal{T} = ((T, F), \mathit{bag}_{\mathcal{T}})$ is *normalized*, if $\mathcal{T}$ is weakly normalized and each exchange node $t \in T$ with child node t_1 is either

- an *introduction* node where $\mathit{bag}_{\mathcal{T}}(t_1) \subset \mathit{bag}_{\mathcal{T}}(t)$ and $|\mathit{bag}_{\mathcal{T}}(t) \setminus \mathit{bag}_{\mathcal{T}}(t_1)| = 1$; or
- a *removal* node where $\mathit{bag}_{\mathcal{T}}(t) \subset \mathit{bag}_{\mathcal{T}}(t_1)$ and $|\mathit{bag}_{\mathcal{T}}(t_1) \setminus \mathit{bag}_{\mathcal{T}}(t)| = 1$.

Furthermore, for root node r in $\mathcal{T}$, $bag_{\mathcal{T}}(r) = \{\}$.

A weakly-normalized tree decomposition can easily be transformed into a normalized one by replacing exchange nodes with a sequence of removal and introduction nodes. For a QBF and tree decomposition $\mathcal{T}$, we associate with each node in $\mathcal{T}$ a corresponding *local expansion tree*.

Definition A.6. Let $\mathsf{Q}.\psi$ be a closed PCNF QBF and $\mathcal{T}$ be a tree decomposition of ψ. For a tree decomposition node t, its associated *local expansion tree* $E_{\leq t}$ is defined as $E(\mathsf{Q}.\psi_{\leq t}, Y_{\leq t})$, where

- $\psi_{\leq t}$ are the clauses in ψ induced by the bag contents of the subtree rooted at t.
- $Y_{\leq t}$ are all variables removed in the subtree rooted at t.

Obviously, at the root r of the tree decomposition, the local expansion tree $E_{\leq r}$ corresponds to $E(\mathsf{Q}.\psi, X)$, and the original QBF instance can be evaluated. Here we use NSFs to only represent information that is necessary for evaluating the QBF (recall also Definition 4.3). The relation of an expansion tree to an NSF that contains all necessary information is captured as follows.

Definition A.7. Let $\mathsf{Q}.\psi$ be a closed PCNF QBF, $\mathcal{T}$ be a tree decomposition of ψ and $t \in \mathcal{T}$ be a tree decomposition node. Furthermore, let $E_{\leq t} = E(\psi_{\leq t}, Y_{\leq t})$ be the local expansion tree for t. A nested set of formulae N_t is called a *valid NSF* (for $\mathsf{Q}.\psi$ in t w.r.t. $\mathcal{T}$) if there exists a mapping α from the nodes in $E_{\leq t}$ to the nestings in N_t that satisfies the following properties:

1. α maps every internal node v in $E_{\leq t}$ to a nesting n in N_t such that $block(v) = block(n)$. Suppose there is some internal node v in $E_{\leq t}$ that is mapped to $\alpha(v) = n$ in N_t. Let v' be a child node of v.
   - If $block(v) = block(v')$, $\alpha(v') = n$ (i.e., v and v' map to the same nesting in N_t).
   - Otherwise, $\alpha(v') = n'$ where $n' \in n$.
2. If v in $E_{\leq t}$ is a leaf node, $\psi_{\leq t}[Y_{\leq t}/v] \equiv \alpha(v)$ (the formula resulting from assigning the truth values is equivalent to the BDD mapped to by α).
3. Each element in N_t is mapped to by α (the mapping is surjective).

Towards readability we overload the notation and sometimes simply write $\alpha : E \to N$ to denote a mapping of *nodes* in an expansion tree E to *nestings* in an NSF N.

The idea is now to compute valid NSFs at each tree decomposition node by dynamic programming. Then, at the root node r of the tree decomposition, this yields NSF N_r. Provided that if N_r is valid, instead of computing and evaluating $E_{\leq r}$, N_r can be evaluated by recursively applying the connectives associated with the nestings on the elements of the nestings to decide satisfiability of the instance:

Proposition A.8. Let $\mathsf{Q}.\psi$ be a closed PCNF QBF over variables X and $\mathcal{T}$ be a tree decomposition of ψ with root node r. Furthermore, let N_r be a valid NSF in r. Then, $eval(E(\mathsf{Q}.\psi, X)) \equiv evaluateQ(r, N_r)$ (recall the definition of $evaluateQ(t, N)$ in Algorithm 4).

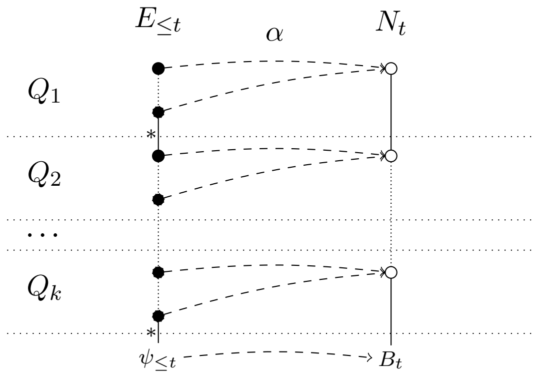

Figure 9. Scematic proof concept for leaf nodes.

Proof:
Since N_r is valid, there exists a mapping $\alpha : E(\mathsf{Q}.\psi, X) \to N_r$ satisfying the properties of Definition A.7. We now show correctness by induction over the structure of the expansion tree.

- Let v be a leaf node in $E(\mathsf{Q}.\psi, X)$. Then $e(v) = \psi[X/v]$ (recall Definition A.3). By Condition 2 of Definition A.7, $\psi[X/v] \equiv \alpha(v)$. Since $\mathit{evaluateQ}(r, \alpha(v)) = \alpha(v)$, $e(v) \equiv \mathit{evaluateQ}(r, \alpha(v))$.

- Let v_s be an internal node in $E(\mathsf{Q}.\psi, X)$ such that $\mathit{block}(v_s) \neq \mathit{block}(v_p)$ where v_p is the parent of v_s in $E(\mathsf{Q}.\psi, X)$ (if any). Let $V = \{v_s\} \cup \{v_1, \dots, v_i\}$ be the subset-maximal set such that $v_1, \dots, v_i$ are the descendants of v_s in $E(\mathsf{Q}.\psi, X)$ with $\mathit{block}(v_s) = \mathit{block}(v_1) = \cdots = \mathit{block}(v_i)$. By Condition 1 of Definition A.7, $\alpha(v_s) = \alpha(v_1) = \cdots = \alpha(v_i)$.

  We now show that $e(v_s) \equiv \mathit{evaluateQ}(r, \alpha(v_s))$. Let $V' = \{v'_1, \dots, v'_j\}$ be the adjacent children of nodes V in $E(\mathsf{Q}.\psi, X)$ where $\mathit{block}(v') \neq \mathit{block}(v_s)$ for all $v' \in V'$.

  We have $\mathit{conn}(v_s) = \mathit{conn}(v_1) = \cdots = \mathit{conn}(v_i)$. Let us assume that $\mathit{conn}(v_s) = \wedge$ (the following argument holds analogously for $\mathit{conn}(v_s) = \vee$). Since $\wedge$ is associative and commutative, we have $e(v_s) \equiv \bigwedge_{v' \in V'}(e(v'))$, which, by induction hypothesis, is equivalent to $\bigwedge_{v' \in V'}(\mathit{evaluateQ}(\alpha(v')))$. By Condition 1 (Item 2) of Definition A.7, for each $v' \in V'$ we have $\alpha(v') \in \alpha(v_s)$, and by Condition 3 there is no $n' \in \alpha(v_s)$ s.t. $\alpha^{-1}(n') \cap V' = \emptyset$. Thus, $\bigwedge_{v' \in V'}(\mathit{evaluateQ}(\alpha(v'))) \equiv \bigwedge_{n' \in \alpha(v_s)}(\mathit{evaluateQ}(n'))$. Since the bag of root node r is empty, this corresponds to $\mathit{evaluateQ}(r, \alpha(v_s))$ and we have $e(v_s) \equiv \mathit{evaluateQ}(r, \alpha(v_s))$.

Overall, for root node r it therefore holds that $\mathit{eval}(E(\mathsf{Q}.\psi, X)) \equiv \mathit{evaluateQ}(r, N_r)$. □

It remains to prove that the NSF at the root node is indeed a valid NSF. We show this by induction over the node types of the tree decomposition.

Lemma A.9. Let $\mathsf{Q}.\psi$ be a closed PCNF QBF with k quantifier blocks, $\mathcal{T}$ be a tree decomposition of ψ and $t \in \mathcal{T}$ be a leaf node. Then, N_t as constructed in Algorithm 1 (Line 1) is a valid NSF.

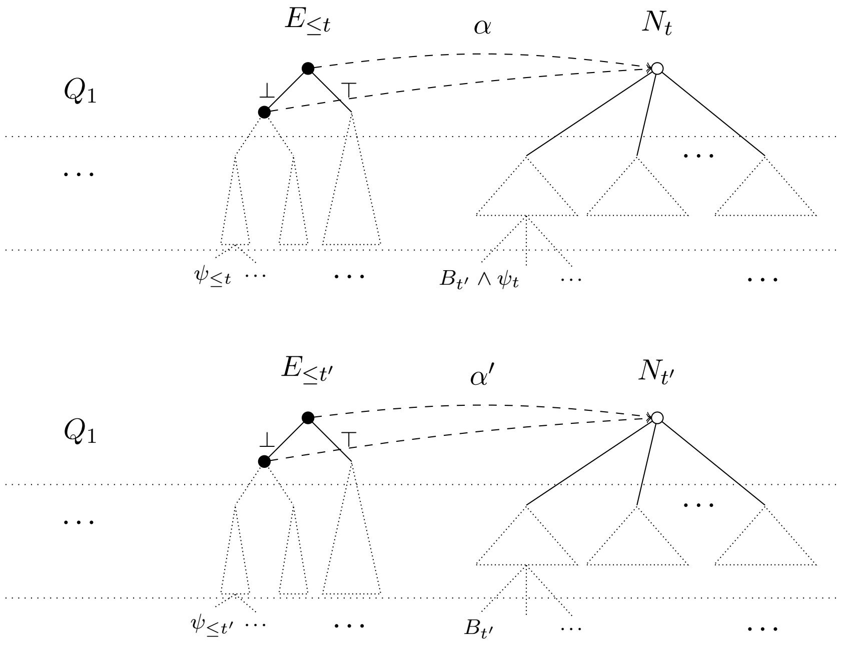

Figure 10. Scematic proof concept for introduction nodes.

Proof:
Figure 9 illustrates the proof strategy for leaf nodes. The local expansion tree $E_{\leq t}$ is specified as $E(\psi_t, \emptyset)$ (i.e., $\psi_{\leq t} = \psi_t$ and no variables are yet removed), which yields a single path $v_1, \dots, v_m, v_{m+1}$. By Algorithm 1, N_t is a nesting of depth k, where each nesting contains exactly one element, and the innermost element is a BDD $B_t = \psi_t$.

We can construct a mapping $\alpha : E_{\leq t} \to N_t$ that satisfies the properties of Definition A.7: let j (with $1 \leq j \leq k$) be a quantifier block. Each node v in $E_{\leq t}$ where $block(v) = j$ maps to the single nesting n in N_t where $block(n) = j$. Additionally, we can specify α to map the single leaf v_{m+1} in $E_{\leq t}$ to the innermost nesting in N_t, since $\psi_{\leq t}[\emptyset / v_{m+1}] \equiv \psi_t$. □

Lemma A.10. Let Q.ψ be a closed PCNF QBF with k quantifier blocks, $\mathcal{T}$ be a tree decomposition of ψ, $t \in \mathcal{T}$ be a introduction node with child node t' and let N_t be obtained from a valid NSF $N_{t'}$ in the child node by $N_t = N_{t'}[B/B \wedge \psi_t]$ (see also Algorithm 1, Line 5). Then, also N_t is a valid NSF.

Proof:
Our proof strategy is illustrated in Figure 10. Let $E_{\leq t} = E(\psi_{\leq t}, Y_{\leq t})$ and $E_{\leq t'} = E(\psi_{\leq t'}, Y_{\leq t'})$ be the local expansion trees at t and t'. By assumption, $N_{t'}$ is a valid NSF in t'. We have to show that N_t, obtained by $N_t = N_{t'}[B/B \wedge \psi_t]$ is a valid NSF in t. By induction hypothesis, we know that there exists a mapping $\alpha' : E_{\leq t'} \to N_{t'}$ that satisfies the conditions of Definition A.7.

For now, let N_t^* be the NSF resulting from $N_{t'}$ by applying the introduction procedure, but without set semantics (i.e., duplicates are not removed in N_t^*). First, we show that we can construct a mapping

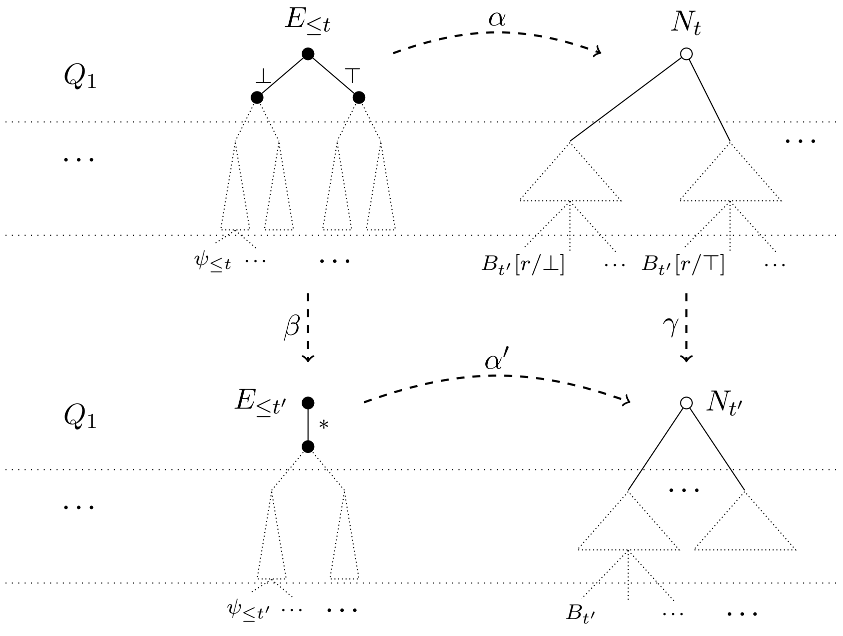

Figure 11. Scematic proof concept for removal nodes.

$\alpha^* : E_{\leq t} \to N_t^*$ that satisfies the conditions of Definition A.7. Observe that the tree structure in $E_{\leq t}$ and $E_{\leq t'}$ is the same since $Y_{\leq t} = Y_{\leq t'}$ (no variable is removed in t). Also, the structure of nestings in $N_{t'}$ and N_t^* is preserved when applying $N_t^* = N_{t'}[B/B \wedge \psi_t]$. Hence, for α^* we can directly reuse mapping α'. It remains to show that for all leaf nodes v in $E_{\leq t}$ it holds that $\psi_{\leq t}[Y_{\leq t}/v] \equiv \alpha^*(v)$. We have that $\psi_{\leq t}[Y_{\leq t}/v]$ is equivalent to $(\psi_{\leq t'} \wedge \psi_t)[Y_{\leq t'}/v']$, which corresponds to $\psi_{\leq t'}[Y'_{\leq t}/v'] \wedge \psi_t$ since $[Y_{\leq t'}/v']$ only assigns variables that are already removed and can thus not occur in ψ_t (recall the connectedness condition of tree decompositions as stated in Definition 3.3). Now, $\psi_{\leq t'}[Y_{\leq t'}/v'] \wedge \psi_t = \alpha'(v') \wedge \psi_t$ which exactly corresponds to the construction of innermost elements in N_t by our introduction procedure. Overall, α^* satisfies the required properties.

Finally, from α^* we can construct a mapping $\alpha : E_{\leq t} \to N_t$ where duplicate elements in N_t^* are removed due to set semantics in N_t, and for such elements α maps to the single remaining nesting. Then, α satisfies all conditions of Definition A.7 and N_t is indeed a valid NSF. □

Lemma A.11. Let $Q.\psi$ be a closed PCNF QBF with k quantifier blocks, $\mathcal{T}$ be a tree decomposition of ψ, $t \in \mathcal{T}$ be a removal node with child node t' and let N_t be obtained from a valid NSF $N_{t'}$ as specified in Algorithm 1 (Line 4). Then, also N_t is a valid NSF.

Proof:
Figure 11 illustrates our proof strategy for removal nodes. Let r be the removed variable. Furthermore, $E_{\leq t} = E(\psi_{\leq t}, Y_{\leq t}) = (V_{\leq t}, C_{\leq t}, ass_{\leq t})$ and $E_{\leq t'} = E(\psi_{\leq t'}, Y_{\leq t'}) = (V_{\leq t'}, C_{\leq t'}, ass_{\leq t'})$ are the

local expansion trees at t and t'. We have $\psi_{\leq t} = \psi_{\leq t'}$ and $Y_{\leq t} = Y_{\leq t'} \cup \{r\}$. We first define a mapping β from nodes in $E_{\leq t}$ to nodes in $E_{\leq t'}$ (denoted by $\beta : V_{\leq t} \rightarrow V_{\leq t'}$):

- Let $v \in V_{\leq t}$ and $v' \in V_{\leq t'}$ be the root nodes in $E_{\leq t}$ and $E_{\leq t'}$. Then, $\beta(v) = v'$.
- For $(v, v_c) \in C_{\leq t}$ where $var(v) \neq r$, we have $\beta(v_c) = v'_c$ s.t. $\beta(v) = v'$ with $(v', v'_c) \in C_{\leq t'}$ and $ass_{\leq t}((v, v_c)) = ass_{\leq t'}((v', v'_c))$.
- For $(v, v_c) \in C_{\leq t}$ where $var(v) = r$, we have $\beta(v_c) = v'_c$ s.t. $\beta(v) = v'$ with $(v', v'_c) \in C_{\leq t'}$.

For the valid NSF $N_{t'}$, we know that there exists a mapping $\alpha' : E_{\leq t'} \rightarrow N_{t'}$ (satisfying the properties of Definition A.7). We now show that there exists a mapping $\alpha : E_{\leq t} \rightarrow N_t$, such that N_t, obtained from $N_{t'}$ via the removal procedure as specified in Algorithm 4, is a valid NSF.

Towards this, let N_t^* be the NSF resulting from $N_{t'}$ when applying the removal procedure, but without set semantics (i.e., duplicates are not removed). We store a mapping γ^* that gives for each nesting in N_t^* the nesting in $N_{t'}$ it originated from. Recall Algorithm 4 for the removal of variables. Let n^* be a nesting in N_t^* and n' be a nesting in $N_{t'}$.

- If n^* is obtained from n' by $n^* = split(n', r)$, $n^* = n'[B/B[r/\bot]]$, or $n^* = n'[B/B[r/\top]]$, then $\gamma^*(n^*) = n'$.

Now we specify a mapping $\alpha^* : E_{\leq t} \rightarrow N_t^*$ as follows.

- Let v and n^* be the root nodes in $E_{\leq t}$ and N_t^*. Then $\alpha^*(v) = n^*$.
- Let v_c be a child node of some node v in $E_{\leq t}$.
  - If v_c is an internal node and $block(v_c) = block(v)$ then $\alpha^*(v_c) = \alpha^*(v)$
  - otherwise $\alpha^*(v_c) = m^*$ s.t. $m^* \in \alpha^*(v)$ and $\alpha'(\beta(v_c)) = \gamma^*(m^*)$
- If v in $E_{\leq t}$ is a leaf node, if r is assigned $\top$ ($\bot$) on the path from the root to v, then $\alpha^*(v)$ is the BDD resulting from assigning r to $\top$ ($\bot$) in the removal procedure.

We now show that α^* satisfies the properties as specified in Definition A.7. Obviously, nesting of blocks as required by Condition 1 is preserved due to the recursive definition of α^* with respect to blocks. Mapping α^* preserves the internal structure of nestings w.r.t. blocks and levels. In particular, the mapping is defined for all nodes in $E_{\leq t}$, and all nestings in N_t^* are mapped to by α^*.

We now show that for all leaves v in $E_{\leq t}$ it holds that $\psi_{\leq t}[Y_{\leq t}/v] \equiv \alpha^*(v)$. Let $\cdot \in \{\top, \bot\}$ be the assignment of r in the context of v. We have $\psi_{\leq t}[Y_{\leq t}/v] \equiv \psi_{\leq t'}[Y_{\leq t'}/\beta(v)][r/\cdot]$. By induction hypothesis, we know that $\psi_{\leq t'}[Y_{\leq t'}/\beta(v)] \equiv \alpha'(\beta(v))$. Thus, $\psi_{\leq t}[Y_{\leq t}/v] \equiv \alpha'(\beta(v))[r/\cdot]$. Since $\alpha'(\beta(v)) = \gamma^*(\alpha^*(v))$ (by specification of α^*), and $\gamma^*(\alpha^*(v))[r/\cdot] = \alpha^*(v)$ (by our algorithm) we have $\psi_{\leq t}[Y_{\leq t}/v] \equiv \alpha^*(v)$. Finally, from α^* we can construct a mapping $\alpha : E_{\leq t} \rightarrow N_t$ such that elements in N_t^* that coincide due to set semantics in N_t, α maps to the single remaining nesting. α then satisfies all conditions of Definition A.7 and N_t is a valid NSF. □

Lemma A.12. Let $Q.\psi$ be a closed PCNF QBF with k quantifier blocks, $\mathcal{T}$ be a tree decomposition of ψ, $t \in \mathcal{T}$ be a join node with child nodes t' and t'' and let N_t be the NSF obtained from valid NSFs $N_{t'}$ and $N_{t''}$ as specified in Algorithm 1, Line 10. Then, also N_t is a valid NSF.

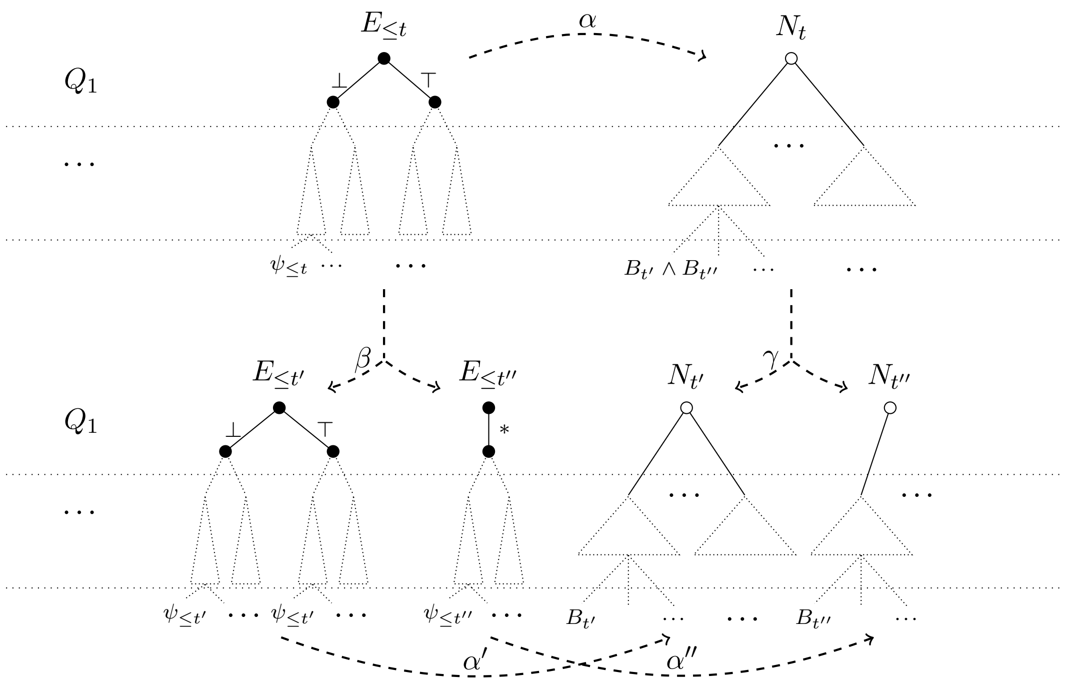

Figure 12. Scematic proof concept for join nodes.

Proof:
Figure 12 illustrates our proof strategy for join nodes. $E_{\leq t} = E(\psi_{\leq t}, Y_{\leq t}) = (V_{\leq t}, C_{\leq t}, ass_{\leq t})$, $E_{\leq t'} = E(\psi_{\leq t'}, Y_{\leq t'}) = (V_{\leq t'}, C_{\leq t'}, ass_{\leq t'})$ and $E_{\leq t''} = E(\psi_{\leq t''}, Y_{\leq t''}) = (V_{\leq t''}, C_{\leq t''}, ass_{\leq t''})$ are the local expansion trees in t, t' and t'' respectively. We have $Y_{\leq t} = Y_{\leq t'} \cup Y_{\leq t''}$. Due to the connectedness property of tree decompositions we know that $Y_{\leq t'} \cap Y_{\leq t''} = \{\}$. In other words, a variable associated with a node contained in $Y_{\leq t}$ is expanded in $E_{\leq t'}$ or $E_{\leq t''}$, but not in both. Hence we can define a mapping $\beta : V_{\leq t} \to V_{\leq t'} \times V_{\leq t''}$ that relates nodes in $E_{\leq t}$ to pairs of nodes in $E_{\leq t'}$ and $E_{\leq t''}$ as follows:

- Let $v \in V_{\leq t}$, $v' \in V_{\leq t'}$ and $v'' \in V_{\leq t''}$ be the root nodes in $E_{\leq t}$, $E_{\leq t'}$ and $E_{\leq t''}$. Then, $\beta(v) = (v', v'')$.

- For $(v, v_c) \in C_{\leq t}$, we have $\beta(v_c) = (v'_c, v''_c)$ such that $\beta(v) = (v', v'')$ with $(v', v'_c) \in C_{\leq t'}$ and $(v'', v''_c) \in C_{\leq t''}$, and:
  - If $ass_{\leq t}((v, v_c)) = \top$ then $ass_{\leq t'}((v', v'_c)) = \top$ or $ass_{\leq t''}((v'', v''_c)) = \top$.
  - If $ass_{\leq t}((v, v_c)) = \perp$ then $ass_{\leq t'}((v', v'_c)) = \perp$ or $ass_{\leq t''}((v'', v''_c)) = \perp$.

Since $N_{t'}$ and $N_{t''}$ are valid NSFs (by induction hypothesis), we know that there exist mappings $\alpha' : E_{\leq t'} \to N_{t'}$ and $\alpha'' : E_{\leq t''} \to N_{t''}$ satisfying the conditions in Definition A.7. We now show that there exists a mapping $\alpha : E_{\leq t} \rightarrow N_t$ s.t. N_t, obtained by the joining procedure as given in Algorithm 10, is a valid NSF.

Similar to before, first let N_t^* be the NSF resulting from joining $N_{t'}$ and $N_{t''}$, but without set semantics (i.e., duplicates are not removed). We store a mapping γ^* that gives for each nesting in N_t^* the pair of nestings in $N_{t'}$ and $N_{t''}$ it originated from. Recall Algorithm 10 for joining NSFs. Let n^* be a nesting in N_t^*, n' a nesting in $N_{t'}$ and n'' a nesting in $N_{t''}$.

- If n^* is obtained from n' and n'' by $n^* = \mathit{join}(n', n'')$, then $\gamma^*(n^*) = (n', n'')$.

Now we specify a mapping $\alpha^* : E_{\leq t} \rightarrow N_t^*$ as follows.

- Let v and n^* be the root nodes in $E_{\leq t}$ and N_t^*. Then $\alpha^*(v) = n^*$.
- Let v_c be a child node of some node v in $E_{\leq t}$.
  - If v_c is an internal node and $\mathit{block}(v_c) = \mathit{block}(v)$ then $\alpha^*(v_c) = \alpha^*(v)$
  - otherwise $\alpha^*(v_c) = m^*$ s.t. $m^* \in \alpha^*(v)$. Furthermore, let $\beta(v_c) = (v_c', v_c'')$ and $\gamma^*(m^*) = (m', m'')$. Then $\alpha'(v_c') = m'$ and $\alpha''(v_c'') = m''$ must hold.

We now show that α^* satisfies the properties as specified in Definition A.7.

Obviously, the specification of α^* retains nesting of blocks as required by Condition 1. Furthermore, the mapping is defined for all nodes in $E_{\leq t}$, and all nestings in N_t^* are mapped to by α^* (Condition 3). Now, let v be a leaf node in $E_{\leq t}$. We show that $\psi_{\leq t}[Y_{\leq t}/v] \equiv \alpha^*(v)$ (Condition 2). Let $\beta(v) = (v', v'')$. We have $\psi_{\leq t}[Y_{\leq t}/v] \equiv \psi_{\leq t'}[Y_{\leq t'}/\beta(v')] \wedge \psi_{\leq t''}[Y_{\leq t''}/\beta(v'')]$ (since $\psi_{\leq t} = \psi_{\leq t'} \wedge \psi_{\leq t''}$ and $Y_{\leq t'} \cap Y_{\leq t''} = \{\}$ due to the connectedness condition). By induction hypothesis, $\psi_{\leq t'}[Y_{\leq t'}/\beta(v')] \equiv \alpha'(v')$ and $\psi_{\leq t''}[Y_{\leq t''}/\beta(v'')] \equiv \alpha'(v'')$. Let $\gamma^*(\alpha^*(v)) = (m', m'')$. By definition of our algorithm, $\alpha^*(v) = m' \wedge m''$. By construction of α^*, $m' = \alpha'(v')$ and $m' = \alpha'(v')$. Overall, $\psi_{\leq t}[Y_{\leq t}/v] \equiv \alpha^*(v)$.

Similar to before, from α^* we can construct a mapping $\alpha : E_{\leq t} \rightarrow N_t$ such that for elements in N_t^* that coincide due to set semantics, α maps to the single remaining nesting in N_t. Mapping α then satisfies all conditions of Definition A.7 and N_t is a valid NSF. $\square$