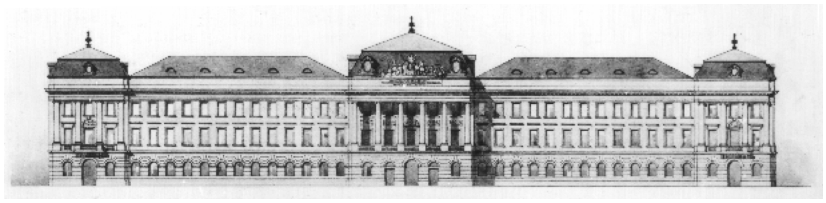
dbai

Technical Report



Institut für Informationssysteme

Abteilung Datenbanken und Artificial Intelligence

BDD-based Dynamic Programming on Tree Decompositions

DBAI-TR-2016-95

Günther Charwat **Stefan Woltran**

Institut für Informationssysteme
Abteilung Datenbanken und Artificial Intelligence
Technische Universität Wien
Favoritenstr. 9
A-1040 Vienna, Austria
Tel: +43-1-58801-18403
Fax: +43-1-58801-18493
sekret@dbai.tuwien.ac.at
www.dbai.tuwien.ac.at

DBAI Technical Report
2016

TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

DBAI Technical Report
DBAI Technical Report DBAI-TR-2016-95, 2016

BDD-based Dynamic Programming on Tree Decompositions

Günther Charwat [1] **Stefan Woltran** [1]

Abstract. Dynamic programming on tree decompositions is a well-studied approach for solving computationally hard problems efficiently – given that the input exhibits small treewidth. Usually, implementations rely on tables for storing information, and algorithms specify how tuples are manipulated during traversal of the decomposition. However, a bottleneck of such table-based algorithms is relatively high memory consumption. Binary Decision Diagrams (BDDs) and related concepts have been shown to be very well suited to store information efficiently.
In this report we illustrate BDD-based dynamic programming on tree decompositions. We first show algorithms for several well-known NP-complete problems that are fixed-parameter tractable w.r.t treewidth. These algorithms are specified on a logical level in form of set-based formula manipulation operations that are executed directly on the underlying BDD data structure.
We then extend our approach to Quantified Boolean Formula (QBF) solving. Since the corresponding decision problem (QSAT) is PSPACE-complete, we require additional machinery. The data structure is extended to nestings of BDDs, which account for quantifier alternations of the QBF instance. Additionally, several algorithm optimizations, including heuristic data structure compression, delayed variable removal and intermediate unsatisfiability checks, are introduced. We develop the prototypical QBF solver *dynQBF*, that shows to be useful in practice for instances with few quantifier alternations and where the treewidth of the propositional formula does not exceed 50. Compared to state-of-the-art solvers, our system performs well on 2-QBF instances, and we even identify classes of instances that are uniquely solved by dynQBF.

[1]Institute for Information Systems 184/2, Technische Universität Wien, Favoritenstrasse 9-11, 1040 Vienna, Austria. E-mail: {gcharwat,woltran}@dbai.tuwien.ac.at

Acknowledgements: This work has been supported by the Austrian Science Fund (FWF): Y698, P25607, P25518. The authors would like to thank Michael Abseher for his effort on the very efficient htd tree decomposition library.

Contents

1 Introduction

An important task in computer science is to develop efficient procedures for processing large amounts of data. This imposes a great challenge, in particular for problems that are known to be intractable. One approach, that emerged from the field of parameterized complexity [43], is to analyze the problem at hand, and to identify parameters that are responsible for its hardness. Then, a dedicated algorithm is designed, that aims at exploiting those parameters. Given such a parameter p, many intractable problems can be solved in polynomial time, assuming that p is a fixed constant. In particular, the complexity class FPT contains problems that are solvable in time $f(p) \cdot n^{\mathcal{O}(1)}$, where f is a computable function, p is the parameter, and n is the input size. Here, the runtime might be exponential in p, but it is only polynomial in n. Hence, even large amounts of data can be handled. In practice, the goal is to design algorithms that solve the problem efficiently, given that the parameter is rather small. Parameters can, for instance, be related to restrictions of (parts of) the input, the solution size, or the structure of the input. Here, we consider the parameter *treewidth*, which is defined on the tree decomposition [94] of the graph representation of the input. A tree decomposition is a mapping from a graph to a tree that splits the instance into smaller parts, thereby taking into account its structure. Roughly speaking, treewidth measures the "tree-likeness" of the input. Using treewidth as a parameter emerged from the observation that many problems are easier to be solved on trees than they are on arbitrary graphs. Additionally, real-world data usually is not random but exhibits certain structure (consider, for instance, railway networks or friendship relations in social networks).

Courcelle showed that every problem that is definable in monadic second-order logic (MSO) is fixed-parameter tractable with respect to treewidth [37]. There, the problem is solved via translation to a finite tree automaton (FTA). However, the algorithms resulting from such "MSO-to-FTA" translations are oftentimes impractical due to large constants [87]. One approach to overcome this problem is to develop dedicated algorithms for the problems at hand. These algorithms usually apply dynamic programming (DP) over the tree decomposition (see, e.g., [40] for a comprehensive introduction and examples). Several general systems are readily available, including Sequoia [68] and D-FLAT [1]. Additionally, there exist problem-tailored implementations, e.g. for the area of abstract argumentation [30], Answer Set Programming [49,84] or bio-informatics [98]. In this work, we extend this promising research area, and push it towards Quantified Boolean Formula (QBF) solving.

Despite continuous progress, both from a conceptual perspective as well as practical realizations, state-of-the-art systems still exhibit certain shortcomings. One major advantage of Sequoia is its capability to *directly* evaluate a given MSO formula by internal execution of a DP algorithm. D-FLAT allows one to declaratively specify the DP algorithm, and is hence particularly useful for the prototypical implementation of novel algorithms. However, their execution is still quite resource-intense, a problem that has been addressed for instance in D-FLAT by a novel approach for anytime computations [20] or by providing optimized support for problems that require subset minimization [19]. Other proposed solutions include special heuristics [15] or reducing the number of simultaneously stored information [7].

Our approach, however, is different, as we want to explicitly tackle the problem of high memory demand by optimizing the data structure that is used for storing intermediate results during the dynamic programming. Additionally, the data structure should give significant advantages in runtime performance. To this end, Binary Decision Diagrams (BDDs) [28] are explored as a suitable substitution of simple tables (see e.g. [87]). A BDD is a data structure that represents models of a Boolean formula in form of a rooted directed acyclic graph (DAG). By omitting redundancies in the DAG, the models can usually be stored in a compact way. BDDs have undergone decades of research and are a well-established concept used, e.g., in model-checking [80], planning [65] and software verification [16].

Overall, our goal is to first elaborate on a novel concept of BDD-based dynamic programming on tree decompositions. Ultimately, we consider the problem of QBF satisfiability checking (QSAT), which is PSPACE-complete [99]. QBFs are a powerful tool to compactly encode many computationally hard problems, which makes them amenable to several application fields where highly complex tasks emerge, e.g. planning, verification, and many more. Most of today's QBF solvers rely on extending the DPLL/CDCL procedures (see e.g. [77]), but also alternative methods based on Binary Decision Diagrams (BDDs) [88] or abstraction-refinement [61] proved successful. Here we want to develop a QBF solver that uses our novel techniques for combined dynamic programming on tree decompositions and BDD-based solving. In our approach nestings of BDDs are to be used to account for quantifier alternations in the QBF instance. We deem BDDs suitable since (1) they allow for a compact representation of (partial) assignments to the QBF; and (2) BDDs are canonical in the sense that equivalent formulas are represented by identical BDDs; thus in order to keep our data structure compact, explicit tests for duplicates are not needed. Our motivation is to develop a novel method for QBF solving that has not been considered yet.

In summary, we tackle the imminent problem of performance shortcomings in state-of-the-art systems that implement dynamic programming on tree decompositions. To this end, the question is how (classically used) tables can be replaced by BDDs. Since BDDs compactly represent Boolean formulae, also the algorithms are to be specified on a logical level, and implications (advantages as well as disadvantages) are to be studied. A particular focus is on QBF solving, following the aforementioned paradigm. For several logical problems, e.g. SAT [96] and CSP [95], such treewidth-based algorithms have already been presented in the literature. For QBF solving, approaches based on treewidth have been developed in [90]. There, the algorithms use BDDs or Zero-suppressed Decision Diagrams (ZDDs). However, in contrast to our approach, they proceed by eliminating the variables from inside out (w.r.t. to the quantification level). Additionally, in [92] a QBF solver that combines search and resolution-based approaches is presented. There, structural parameters of the input are taken into account to decide how to proceed in the solving process. Here, our goal is to develop an alternative QBF solving procedure, that combines dynamic programming on tree decompositions with BDDs. Furthermore, current QBF solvers implement sophisticated optimization strategies. We want to identify existing strategies that are suitable for our approach. Additionally, our approach paves the way for novel optimization strategies during the tree decomposition traversal, which are to be investigated. Our task is to develop a system that is competitive with state-of-the-art QBF solvers, thus showing that our approach can be indeed superior to classical approaches on instances that exhibit low treewidth. Thereby, new insights into QBF solving shall be gained.

In Section 3 we introduce BDD-based dynamic programming on tree decompositions. In particular, we provide logic-based algorithm specifications for several NP-complete problems that are fixed-parameter tractable w.r.t. treewidth. These problems impose different challenges to the algorithm designer:

- 3-Colorability: Once set, the truth value of all variables in the BDDs remains *fixed*.
- Directed Independent Dominating Set: We additionally have to handle variables with *changing* truth value (during the dynamic programming, vertices become dominated).
- Hamiltonian Cycle: Connectedness has to be handled in the DP algorithm.

In Section 4 a novel algorithm for QBF solving of instances in prenex CNF (PCNF) form is presented. The data structure to be used contains (nested) sets of BDDs. The size of each BDD is bounded by the width of the used decomposition, and the overall number of BDDs required in each decomposition node is bounded by the width and by the number of quantifiers in the instance. Thus, the runtime depends exponentially on the structural parameters instead of the size of the formula. In Section 4.4 we introduce several optimizations for the developed algorithm that are crucial for its performance. The approach is implemented in the *dynQBF* system. In Section 5 we provide an experimental evaluation which indicates that our method already performs well on QBFs with one quantifier alternation, while for QBFs with a higher number of alternations our system does not reach the performance of state-of-the-art tools yet. However, we encountered several instances that our solver was able to solve, but where others (we compared our system with DepQBF, RAReQS, and EBDDRES) ran into a timeout. Our method gives rise to several directions of advancements, such as QBF-tailored tree decomposition heuristics and width-reducing preprocessing.

Substantial parts of this report are based on papers published at LPNMR'15 [32] and QBF'16 [33].

2 Preliminaries

In this section, we introduce tree decompositions and give account to the structural parameter *treewidth*. Furthermore, we define Binary Decision Diagrams (BDDs), that serve as the key ingredient for our dynamic-programming based algorithms over the tree decompositions. Additionally, we review state-of-the-art tree decomposition-based systems and study their specifics.

2.1 Overview

In the area of *parameterized complexity* [43] one considers the complexity of the problem at hand with respect to certain parameters of the input or output. The idea is that many hard problems become tractable in case the parameters are bounded by a fixed constant. The complexity class FPT comprises of problems that are solvable in time $f(p) \cdot n^{\mathcal{O}(1)}$, where p is a (fixed) parameter, f is a computable function with a runtime that is only dependent on p, and n is the input size of the problem. Observe that here the explosion in runtime is confined to p instead of the input size. In case p is small, the problem can be solved efficiently, despite a potentially large input size. For problems that are represented as graphs, "obvious" parameters include the number of vertices or edges in the input, or the solution size. Furthermore, there exist several parameters that describe the structure of the graph, such as minimum and maximum degree, connectivity, cliquewidth, branchwidth, pathwidth or treewidth (for a survey on width parameters, see [59]). The latter will be exploited in this work.

2.2 Tree Decompositions and Treewidth

Many computationally hard problems, where the input is represented in form of a graph, become tractable when the input graph forms a tree. One approach to exploit this is the notion of *tree decompositions* (TDs), which were originally introduced in 1984 in [94]. Similar notations appeared already earlier, see, e.g., [14,58]. A tree decomposition is a mapping from an (arbitrary) graph to a tree, where each node in the tree decomposition can contain several vertices of the original graph (called the node's *bag*).

Definition 1. *A* tree decomposition *of an arbitrary graph $G = (V, E)$ is defined as a pair $\mathcal{T} = (T, bag_{\mathcal{T}})$ where $T = (N, F)$ is a (rooted) tree with nodes N and edges F, and $bag_{\mathcal{T}} : N \to 2^V$ assigns to each node a set of vertices, such that the following conditions are met:*

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

Intuitively, Condition 1 and 2 guarantee that the whole input 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. to the bag contents) of the decomposition.

The *width* of $\mathcal{T}$ is defined as $\max_{n \in N} |bag_{\mathcal{T}}(n)| - 1$. The *treewidth* of a graph is the minimum width over all its tree decompositions. Treewidth is a measurement of the "tree-likeness" of a graph: the smaller the treewidth, the easier the problem becomes to solve, with trees having a treewidth of one. Although, given a graph and an integer k, deciding whether the graph has at most treewidth k is NP-complete [4], the problem itself is in FPT

when k is considered as parameter. Furthermore, there are heuristics that give "good" tree decompositions in polynomial time [25, 41, 42].

In this work we will consider several normalization types of tree decompositions. A tree decomposition can be transformed into a (weakly) normalized one in linear time without increasing the width [67].

Definition 2. *A tree decomposition $\mathcal{T} = (T, bag_{\mathcal{T}})$ with $T = (N, F)$ is* weakly normalized *if each node $n \in N$ is of one of the following types:*

- join node*: n has children $n_1, \ldots, n_m$ such that $m \geq 2$, and $bag_{\mathcal{T}}(n) = bag_{\mathcal{T}}(n_1) = \cdots = bag_{\mathcal{T}}(n_m)$ holds.*
- exchange node*: n has exactly one child n_1, such that $bag_{\mathcal{T}}(n) \neq bag_{\mathcal{T}}(n_1)$.*
- leaf node*: n has no children.*

Definition 3. *A tree decomposition $\mathcal{T} = (T, bag_{\mathcal{T}})$ with $T = (N, F)$ is* normalized *(or* nice*) if each node $n \in N$ is of one of the following types:*

- join node*: n has exactly two children n_1 and n_2, and $bag_{\mathcal{T}}(n) = bag_{\mathcal{T}}(n_1) = bag_{\mathcal{T}}(n_2)$.*
- introduction node*: n has exactly one child n_1, such that $bag_{\mathcal{T}}(n) = bag_{\mathcal{T}}(n_1) \cup \{x\}$ with $x \notin bag_{\mathcal{T}}(n_1)$.*
- removal node*: n has exactly one child n_1, such that $bag_{\mathcal{T}}(n) = bag_{\mathcal{T}}(n_1) \setminus \{x\}$ with $x \in bag_{\mathcal{T}}(n_1)$.*
- leaf node*: n has no children.*

In the root node r of $\mathcal{T}$ we have $bag_{\mathcal{T}}(r) = \{\}$.

In this work, normalization types are merely used to provide a more succinct and readable representation of our algorithms, but could be easily transformed to algorithms for non-normalized tree decompositions. Furthermore, note that join nodes are defined differently in weakly normalized and normalized tree decompositions. The node type to be considered will be made clear from the context.

Example 1. *Figure 1 shows an example graph G and a possible tree decompositions $\mathcal{T}$ of G. The different normalization types are illustrated by means of $\mathcal{T}_w$ (weakly normalized) and $\mathcal{T}_n$ (normalized). The tree decompositions have a width of 2, which corresponds to the optimal width (i.e., the treewidth) of G.*

Given a tree decomposition $\mathcal{T} = (T, bag_{\mathcal{T}})$ with $T = (N, F)$, for a tree decomposition node $n \in N$ we denote its set of children in T by $children_{\mathcal{T}}(n)$. In order to iterate over the children, we specify $firstChild_{\mathcal{T}}(n)$ and $nextChild_{\mathcal{T}}(n)$ as procedures to access the children, and $hasNextChild_{\mathcal{T}}(n)$ to check whether further children exist. $isLeaf_{\mathcal{T}}(n)$ returns true if n has no children, introduction (removal) nodes are tested by

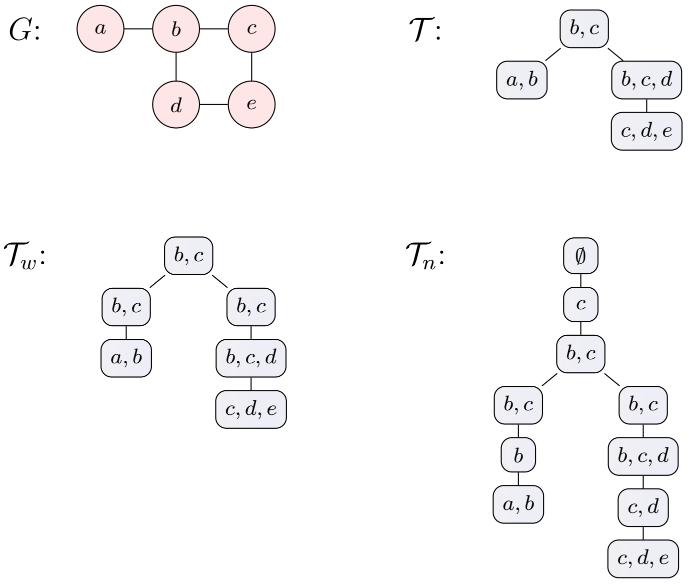


Figure 1: Example graph G, tree decomposition $\mathcal{T}$, weakly-normalized tree decomposition $\mathcal{T}_w$ and normalized tree decomposition $\mathcal{T}_n$.

$\mathit{isIntroduction}_{\mathcal{T}}(n)$ ($\mathit{isRemoval}_{\mathcal{T}}(n)$). Exchange nodes are tested with $\mathit{isExchange}_{\mathcal{T}}(n)$. For nodes with more than one child, $\mathit{isJoin}_{\mathcal{T}}(n)$ returns true. For a node n with single child node n_1, we denote by $\mathit{introduced}_{\mathcal{T}}(n) = \mathit{bag}_{\mathcal{T}}(n) \setminus \mathit{bag}_{\mathcal{T}}(n_1)$ the variables introduced in n; and $\mathit{removed}_{\mathcal{T}}(n) = \mathit{bag}_{\mathcal{T}}(n_1) \setminus \mathit{bag}_{\mathcal{T}}(n)$ gives the variables removed in n. Furthermore, edges of a graph instance $G = (V, E)$ are related to nodes in the tree decomposition by $\mathit{edges}_{\mathcal{T},G}(n) = \{e \mid e \in E, e \subseteq \mathit{bag}_{\mathcal{T}}(n)\}$. For improved readability, we will usually omit subscript $\mathcal{T}$ and G.

2.3 Dynamic Programming on Tree Decompositions: State of the Art

The famous theorem by Courcelle in 1990 (see [37]), stating that any problem definable in Monadic Second-Order logic (MSO) can be solved in linear time on graphs with bounded treewidth, provides an important theoretical basis for today's tree decomposition-based approaches. MSO is an extension to First-Order logic (FO), where not only quantification over objects, but also over sets of objects is allowed. In [5], the theorem is extended to a large class of decision, counting and optimization problems. There, an additional function can be provided along with the MSO formula to evaluate the solutions to the formula. The proof of Courcelle's theorem is based on the translation of an MSO formula to a suitable finite-state tree automaton (FTA). A software that follows this approach is called MONA [66]. It implements particular algorithms for minimizing the finite-state automata that use BDDs. However, in general it was observed that a direct implementation following the proof of Courcelle's theorem is oftentimes impractical, since there are large constants involved resulting in space explosion, even for instances of small treewidth, see [87]. This calls for a second step, in which problem-specific algorithms with improved efficiency are to be developed [87]. Other authors provide similar arguments, see, e.g., [56, 57].

A vast amount of dedicated algorithms was developed in recent years. Most of today's algorithms are based on dynamic programming (DP) and follow the concepts presented in [6]. A prominent representative is the system SEQUOIA [68,71], that solves a given MSO formula via dynamic programming over an internally constructed tree decomposition. It follows a game-theoretic approach, and avoids the space-explosion that usually emerges during the expensive power-set construction in automata-based approaches. Since the tool can evaluate MSO formulae directly, it is quite expressive and easy to use. However, it does not give control over the applied dynamic programming algorithm. The D-FLAT system [1] allows one to specify the dynamic programming algorithm in the declarative language of answer-set programming [27, 53]. Hence, the algorithm developer can take problem-specific shortcuts into account, despite, of course, a new dynamic programming algorithm has to be designed for every problem separately. The algorithm is then executed on the internally constructed tree decomposition. It was shown that the tool can solve any MSO-definable problem in fpt time as stated in Courcelle's theorem [21].

Additionally, systems for particular problem domains were developed. For instance, in the logic domain, there exists the ASP solver dynASP [84, 85]. Additionally, algorithms for the Boolean Satisfiability problem [96] and the Constraint Satisfaction problem [95] were designed. In the area of abstract argumentation [45], the dynPARTIX system was developed [30]. For counting problems, this system can outperform state-of-the-art systems on instances of small treewidth.

Excellent surveys on algorithms for graphs of bounded treewidth include [22] and [24]. An intuitive introduction to dynamic programming on tree decompositions is given in [40]. Additionally, in [72] a comprehensive overview on MSO and tree decompositions is given.

2.4 Binary Decision Diagrams

A BDD is a well-studied and widely-used data structure that represents Boolean formulae in form of a rooted directed acyclic graph [3, 73]. In this work, we use a special type of BDDs, so-called Reduced Ordered Binary Decision Diagrams (ROBDDs) [28] as one key ingredient for efficiently storing information. Furthermore, for a fixed ordering over variables occurring in the formula, they are canonical, i.e., equivalent formulae are represented by the same ROBDD.

Definition 4. *An* Ordered Binary Decision Diagram $B = (V_B, A_B)$ *is a rooted, connected, directed acyclic graph where* $V_B = V_T \cup V_N$ *and* $A_B = A_\top \cup A_\bot$*. The following conditions have to be satisfied:*

1. V_T *may contain the terminal nodes* $\top$ *and* $\bot$*.*

2. V_N *contains the internal nodes, where each* $v \in V_N$ *represents a variable* v*.*

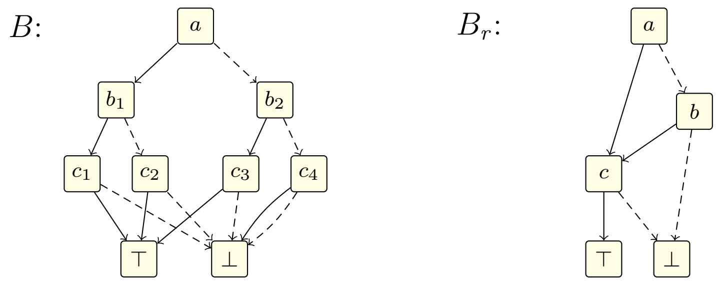


Figure 2: OBBD B and ROBBD B_r of formula $(a \wedge b \wedge c) \vee (a \wedge \neg b \wedge c) \vee (\neg a \wedge b \wedge c)$.

3. *Each $v \in V_N$ has exactly one outgoing arc in $A_\top$ and one in $A_\bot$.*
4. *For every path from the root to a terminal node, each variable occurs at most once and in the same order (i.e., we have a strict total order over the variables).*

Given an OBDD B, propositional variables V_N and an assignment A to V_N, the *corresponding path* in B is the unique path from the root node to a terminal node, such that for every $v \in V_N$ it includes the outgoing arc in $A_\top$ ($A_\bot$) iff A gets assigned true (false) for v. A is a satisfying assignment of the formula represented by B iff the path ends in $\top$.

Definition 5. *A* Reduced OBDD *(ROBBD), is an OBDD where isomorphic nodes are merged into a single node with several incoming edges. Furthermore, nodes $v \in V_N$ where both outgoing arcs reach the same node $v' \in V_B$, are removed.*

Example 2. *Figure 2 shows an OBDD B and the corresponding ROBBD B_r for formula $(a \wedge b \wedge c) \vee (a \wedge \neg b \wedge c) \vee (\neg a \wedge b \wedge c)$. Arcs in $A_\top$ are represented by a solid arrow, a dashed arrow marks arcs in $A_\bot$. Nodes c_1, c_2 and c_3 represent the same variable c and have arcs to the same terminal nodes. Hence, these isomorphic nodes are merged to a single node c. Then, both outgoing arcs of b_1 reach c, and b_1 is removed. Furthermore, c_4 is removed.*

BDDs support standard logical operators *conjunction* ($\wedge$), *disjunction* ($\vee$), *negation* ($\neg$) and *equivalence* ($\leftrightarrow$). Furthermore, for a BDD B, *existential quantification* over a set of variables V, $V \subseteq V_N$ is denoted by $\exists V B$. *Restriction* of a variable $v \in V_N$ to true ($\top$) or false ($\bot$) and *renaming* to a variable v' is denoted by $B[v/\cdot]$ where $\cdot \in \{\top, \bot, v'\}$. For sets of variables $V \subseteq V_N$, $B[V/\cdot]$ with $\cdot \in \{\top, \bot, V'\}$ and $V' = \{v' \mid v \in V\}$, denotes restriction or renaming of each $v \in V$ by applying $B[v/\cdot]$. In the following we will not explicitly give the BDDs, but state Boolean formulae instead.

2.5 Related Work on BDDs

As mentioned before, Binary Decision Diagrams were originally proposed in [73] and later refined in [3]. In 1984, Bryant's work on Reduced Ordered BDDs [28] paved the way to obtain efficient algorithms and implementations. Although today's focus shifts to efficient SAT solving techniques (usually based on the Davis–Putnam– Logemann–Loveland (DPLL) procedure), BDD technology is still popular in applications to software verification and model checking (see [44] for an overview of the topic).

Besides BDDs, there exists a vast amount of decision diagrams, each providing its distinct set of features. For sparse graphs, Zero-suppressed DDs (ZDD) [82] showed to be suitable (see also the superseding "review article" [83]). Multi-Terminal BDDs (MTBDDs) [51] extend the binary terminal nodes to terminal nodes with real numbers, which are also called Algebraic DDs (ADDs) [11]. Edge-valued BDDs (EVBDDs) [70] allow for arcs to be labelled with Integers. They are particularly useful for solving Integer linear programming (ILP) problems [69]. In Multi-valued DDs (MDDs) [64], variables can have multiple values (over a fixed domain), and were extended to Edge-valued MDDs (EV+MDDs) [52], where arcs additionally get a weight, applicable for instance to constraint programming. Data DDs (DDDs) [38] also allow for multiple values, but with an a priori unbounded domain. Additionally, the variable order is not necessarily fixed. If the domain is fixed, they correspond to MDDs, if the domain is Boolean, they are just BDDs. Set DDs (SDDs) [39, 101] have arcs labelled with sets that are themselves stored as SDDs. Sentential DDs (also called SDDs) are decision diagrams that contain decision nodes (disjunction) and elements (conjunction). Sigma DDs ($\sum$DDs) [29] handle sets of terms efficiently. In this work we stick to (Reduced Ordered) BDDs, since we want to ultimately tackle the problem of efficient QBF solving. Since BDDs can efficiently represent Boolean formulae (and are designed for this purpose), they provide all features we require.

Examples for successful implementations of (Binary) Decision Diagrams include CUDD [97], BuDDy [74] and Biddy [81]. All systems provide similar features, such as dynamic variable reordering heuristics and garbage collection, and are implemented in C. Both CUDD and BuDDy additionally provide a C++ interface. Furthermore, CUDD supports BDDs, ADDs and ZDDs. Besides that, there are also Java implementations available, such as JavaBDD [103] and JDD [102]. JINC [89] implements Shared OBDDs, and is tailored towards multi-threading. A (rather old) survey on available systems is given in [62]. In this work we use CUDD due to its continuous maintenance and good performance.

Regarding BDDs and decomposition-based approaches, in the area of knowledge compilation, so-called "Tree-of-BDDs" [48, 100] are constructed in an offline phase from a given CNF, and queried in the online phase to answer questions on this data structure in linear time. Furthermore, ADDs are used for compiling Bayesian networks in such a way that the structure of the network can be exploited in order to compute inference efficiently [34]. Combining DP and decision diagrams has been proven well-suited also for Constraint Optimization Problems (COPs) [95]. The key idea is to employ ADDs to store the set of possible solutions, and the branch-and-bound algorithm is executed on a decomposition of the COP instance. This was shown to be superior to earlier approaches in [26], where additionally (no)good recording is applied during computation.

Furthermore, BDDs were already successfully applied in the area of QBF solving, and we review different applications in Section 5.2.

3 BDD-based Dynamic Programming

In this section we introduce the general concept of how dynamic programming proceeds over the tree decomposition. We introduce different algorithm design principles, called *early decision method* (EDM) and *late decision method* (LDM). Several NP-complete problems, that are fixed-parameter tractable with respect to treewidth, are introduced, and their dynamic-programming algorithms are presented. In Section 4 we extend our approach to a problem beyond NP, namely QBF solving. An experimental comparison to state-of-the-art tree decomposition-based tools is given in Section 5.

3.1 Concept

Given some problem $\mathcal{P}$ and input instance I, we can decide $\mathcal{P}$ on I as follows.

1. Transform I into an appropriate graph representation I^G. For some problems, the input is already specified in form of a graph (e.g. for 3-Colorability or Hamiltonian Cycle). Instances of other problems, such as Boolean Satisfiability, can be transformed into a graph, e.g., by constructing the *incidence* or *primal* graph of I (see [96]).

2. Construct a tree decomposition $\mathcal{T}$ of I^G in fpt time, or use heuristics to obtain a "good" one in polynomial time (see Section 2.2). Depending on the dynamic programming algorithm specification, normalize the tree decomposition. In this section we will assume $\mathcal{T}$ to be *normalized.*

3. Traverse $\mathcal{T} = (T, \mathit{bag})$ with $T = (N, F)$ in post-order (i.e., start at the leaves, and continue upwards until the root node of the tree decomposition). At each decomposition node $n \in N$, compute a BDD B_n that represents partial solution candidates for $\mathcal{P}$ on I (details are given in Section 3.1.1).

4. At the root node r of $\mathcal{T}$, either $B_r = \top$ or $B_r = \bot$ holds, giving the solution to the problem.

3.1.1 Recursive Tree Decomposition Traversal

Algorithm 1 illustrates our generic procedure for dynamic programming over a tree decomposition to solve some problem $\mathcal{P}$. Here, it is assumed that the tree decomposition $\mathcal{T}$ is globally available to the algorithm. $\mathit{computeBDD}(n)$ is called recursively (in post-order) on the children of n. In each recursion step, the algorithm distinguishes between the four node types of a normalized tree decomposition. Depending on the node type of the current n, it calls the problem-specific implementations of $\mathit{leaf}^{\mathcal{P}}$,$\mathit{remove}^{\mathcal{P}}$, $\mathit{intro}^{\mathcal{P}}$ or $\mathit{join}^{\mathcal{P}}$. These procedures return a BDD B for the current node n, which is constructed based on information related to n as well as BDDs B_1 and B_2 of the child node(s) (if any). Each model in B represents a partial solution candidate with respect to the subtree of $\mathcal{T}$ that is rooted in

Algorithm 1: Generic recursive procedure *computeBDD*(n) for problem $\mathcal{P}$

```
Input  : A tree decomposition node $n$
Output: A BDD $B$ with partial solution candidates for $n$
1 if $isLeaf(n)$ then $B := leaf^{\mathcal{P}}(n)$
2 else
3     $B_1 := computeBDD(firstChild(n))$
4     if $isRemoval(n)$ then $B := remove^{\mathcal{P}}(n, B_1)$
5     else if $isIntroduction(n)$ then $B := intro^{\mathcal{P}}(n, B_1)$
6     else if $isJoin(n)$ then
7         $B_2 := computeBDD(nextChild(n))$
8         $B := join^{\mathcal{P}}(n, B_1, B_2)$
9 return $B$
```

n (which are restricted to elements in $bag(n)$; thus named "partial"), which might turn out to not represent a solution, later during the tree traversal (thus named "candidate"). If the procedure is called with the root r of $\mathcal{T}$, it returns the overall solution.

For a BDD B_n computed at some node n, all variables in B_n are related to vertices contained in $bag(n)$. In other words, the size of B_n (i.e., the number of nodes in B_n) is restricted by $f(|bag(n)|)$, where f is some computable function. For the algorithms presented in the remainder of the section, the size is bounded by $\mathcal{O}(2^{wl})$ where w is the width of $\mathcal{T}$ and l the number of variables stored per bag element. However, in practice the size may be exponentially smaller, in particular in case a "good" variable ordering is applied [50]. Since finding an optimal variable ordering is in general NP-hard [28], we rely on BDD-internal heuristics for finding such a good ordering [97]. With this, the BDDs require much less space than an equivalent table representation.

3.1.2 Decision Methods

We present two algorithm design choices that affect how $leaf^{\mathcal{P}}$, $remove^{\mathcal{P}}$, $intro^{\mathcal{P}}$ and $join^{\mathcal{P}}$ are specified.

Early decision method (EDM). Here, bag information is incorporated within $intro^{\mathcal{P}}$. This approach is comparable to "classical" table-based implementations (such as usually done in D-FLAT [1]). For unsatisfiable instances, conflicts can be detected early (i.e., immediately when the conflict is introduced).

Late decision method (LDM). BDD manipulation (w.r.t. bag elements) is delayed until removal of vertices. Typically, this approach yields smaller BDDs and less computational effort, and is hence particularly useful for "complicated" algorithms.

3.2 Case Studies

We now illustrate how dynamic programming on tree decompositions with BDDs can be applied to well-known NP-complete problems. We introduce EDM- and LDM-based algorithms for 3-Colorability, Directed Independent Dominating Set and Hamiltonian Cycle. The problems impose different challenges on the algorithm design, which oftentimes reoccur in other problem domains. While the 3-Colorability problem only requires to guess the color assignment of vertices, in the Directed Independent Dominating Set problem it is necessary to keep track of dominated vertices during the dynamic programming. The Hamiltonian Cycle problem additionally requires to handle connectedness of vertices in the cycle.

3.2.1 3-Colorability

Input:	A simple graph $G = (V, E)$.
Question:	Is G 3-colorable?

The 3-Colorability problem is well-suited to illustrate how DP algorithms for problems that are fixed-parameter tractable with respect to tree-width can be specified following our approach. First, we define the set of colors $C = \{r, g, b\}$. Then, the following variables are to be used in the BDDs. For all $c \in C$ and $x \in V$, the truth value of variable c_x denotes whether vertex x gets assigned color c.

EDM. Algorithm 2 shows the BDD manipulation operations for the respective decomposition node types. In order to solve the problem, the operations have to encode that every vertex gets assigned exactly one color, and adjacent vertices do not have the same color. Intuitively, $\mathit{leaf}^{\mathit{3col}_e}$ and $\mathit{intro}^{\mathit{3col}_e}$ add the respective constraints for introduced vertices to the returned BDD. In $\mathit{remove}^{\mathit{3col}_e}$, due to the definition of tree decompositions, we know that all constraints related to removed vertex u were already taken into account. Hence, we can abstract away the variables associated with u, thereby keeping the size of the BDD bound by the width of the tree decomposition. In join nodes, $\mathit{join}^{\mathit{3col}_e}$ combines the intermediate results obtained in the child nodes of the decomposition by a simple conjunction of the BDDs.

LDM. Another possibility for specifying the algorithm is to incorporate information as late as possible, that is, when a vertex is removed from the decomposition, see Algorithm 3. In leaf nodes the BDD $\mathit{leaf}^{\mathit{3col}_l}$ is initialized with $\top$ (i.e., there is no constraint to be considered yet), and in introduction nodes the BDD $\mathit{intro}^{\mathit{3col}_l}$ corresponds to that of the child nodes. When a vertex u is removed, one variable out of r_u, g_u, b_u is set to true, thereby assigning to the vertex exactly one color $c \in C$. Furthermore, adjacent vertices x with $\{x, u\} \in \mathit{edges}(n_1)$

Algorithm 2: Operations for 3-Colorability (EDM)

```
$leaf^{3col_e}(n)$:            $B_{color} = \bigwedge_{c \in C} \bigwedge_{\{x,y\} \in edges(n)} \neg(c_x \wedge c_y) \bigwedge_{x \in current(n)} (r_x \vee g_x \vee b_x)$
                               $B_{neighbor} = \bigwedge_{x \in current(n)} \Big(\neg(r_x \wedge g_x) \wedge \neg(r_x \wedge b_x) \wedge \neg(g_x \wedge b_x)\Big)$
                               **return** $B_{color} \wedge B_{neighbor}$
$remove^{3col_e}(n, B_1)$:     $\{u\} = removed(n)$
                               **return** $\exists r_u g_u b_u [B_1]$
$intro^{3col_e}(n, B_1)$:      $\{u\} = introduced(n)$
                               $B_{color} = \bigwedge_{c \in C} \bigwedge_{\{x,u\} \in edges(n)} \neg(c_x \wedge c_u) \wedge (r_u \vee g_u \vee b_u)$
                               $B_{neighbor} = \neg(r_u \wedge g_u) \wedge \neg(r_u \wedge b_u) \wedge \neg(g_u \wedge b_u)$
                               **return** $B_1 \wedge B_{color} \wedge B_{neighbor}$
$join^{3col_e}(n, B_1, B_2)$:  **return** $B_1 \wedge B_2$
```

Algorithm 3: Operations for 3-Colorability (LDM)

```
$leaf^{3col_l}(n)$:            **return** $\top$
$remove^{3col_l}(n, B_1)$:     $\{u\} = removed(n)$
                               $n_1 = firstChild(n)$
                               $B = B_1[r_u/\top, g_u/\bot, b_u/\bot] \wedge \bigwedge_{\{x,u\} \in edges(n_1)} \neg r_x) \vee$
                                     $B_1[r_u/\bot, g_u/\top, b_u/\bot] \wedge \bigwedge_{\{x,u\} \in edges(n_1)} \neg g_x) \vee$
                                     $B_1[r_u/\bot, g_u/\bot, b_u/\top] \wedge \bigwedge_{\{x,u\} \in edges(n_1)} \neg b_x)$
                               **return** $B$
$intro^{3col_l}(n, B_1)$:      **return** $B_1$
$join^{3col_l}(n, B_1, B_2)$:  **return** $B_1 \wedge B_2$
```

for child n_1 of n must not get assigned the same color, which is achieved by adding $\neg c_x$ to the formula. $remove^{3col_l}$ simply combines the three BDDs resulting from the choice of the color via disjunction. As in EDM, it is sufficient to compute $join^{3col_l}$ via conjunction of the child BDDs.

3.2.2 Directed Independent Dominating Set

The Directed Independent Dominating Set problem corresponds to the (decision variant of the) Stable Extension problem from the area of abstract argumentation [45]. In this area, the parameterized complexity of various semantics and reasoning types with respect to several parameters has been studied, including a table-based DP approach for treewidth [31, 46].

Algorithm 4: Operations for Directed Independent Dominating Set (EDM)

```
$leaf^{dom_e}(n)$:            $B_{ind} = \bigwedge_{(x,y)\in edges(n)} (\neg i_x \vee \neg i_y)$
                              $B_{dom} = \bigwedge_{y\in bag(n)} \left(d_y \leftrightarrow \bigvee_{(x,y)\in edges(n)} i_x\right)$
                              **return** $B_{ind} \wedge B_{dom}$
$remove^{dom_e}(n, B_1)$:     $\{u\} = removed(n)$
                              **return** $B_1[i_u/\top, d_u/\bot] \vee B_1[i_u/\bot, d_u/\top]$
$intro^{dom_e}(n, B_1)$:      $\{u\} = introduced(n)$
                              $n_1 = firstChild(n)$
                              $B_{ind} = \bigwedge_{\{u,y\}\in edges(n)} (\neg i_u \vee \neg i_y)$
                              $B_{dom} = \left(d_u \leftrightarrow \bigvee_{(x,u)\in edges(n)} i_x\right) \wedge \bigwedge_{(u,y)\in edges(n)\wedge u\neq y} \left(d_y \leftrightarrow d'_y \vee i_u\right) \wedge$
                                  $\bigwedge_{y\in bag(n)\wedge(u,y)\notin edges(n)} \left(d_y \leftrightarrow d'_y\right)$
                              **return** $\exists D(n_1)'(B_1[D(n_1)/D(n_1)'] \wedge B_{ind} \wedge B_{dom})$
$join^{dom_e}(n, B_1, B_2)$:  $B_{dom} = \bigwedge_{x\in bag(n)} (d_x \leftrightarrow d'_x \vee d''_x)$
                              $B = B_1[D(n)/D(n)'] \wedge B_2[D(n)/D(n)''] \wedge B_{dom}$
                              **return** $\exists D(n)'\exists D(n)'' B$
```

Input:	A directed graph $G = (V, E)$ (with self-loops).
Question:	Is there a independent dominating set $X \subseteq V$ in G, i.e.: (1) for all $x, y \in X : \{x, y\} \notin E$ (X is independent); and (2) for all $y \in V \backslash X : \exists x \in X$ such that $(x, y) \in E$ (y is dominated)?

For a directed graph $G = (V, E)$, an arc from x to y in G is given by a pair $(x, y) \in E$, while we denote by $\{x, y\} \in E$ that $(x, y) \in E$ or $(y, x) \in E$. We encode the problem using variables $I \cup D \cup D' \cup D''$ with $I = \{i_x \mid x \in V\}$ and $D = \{d_x \mid x \in V\}$. The truth value of some $i_x \in I$ denotes whether x is in some set X. Furthermore, the assignment of true to some variable $d_x \in D$ represents that x is dominated. The sets $D' = \{d'_x \mid x \in V\}$ and $D'' = \{d''_x \mid x \in V\}$ will serve as temporary variables that represent "dominated" information in child nodes during the bottom-up traversal. For a node n of a tree decomposition $\mathcal{T}$, we denote by $D_{\mathcal{T}}(n) = \{d_x \mid x \in bag_{\mathcal{T}}(n)\}$ ($\mathcal{T}$ will be omitted in the following).

EDM. Algorithm 4 shows the operations for solving the Directed Independent Dominating Set problem. In $leaf^{dom_e}$ the BDD encodes that adjacent vertices can not both be contained in a dominating set, and a vertex y (in the node's bag) is dominated (d_y is true) iff there exists an arc from some $x \in bag(n)$ to y, such that i_x is true. For removal nodes, $remove^{dom_e}$ guarantees that the removed vertex u is either contained in the dominating set (by $i_u/\top$) or it is dominated ($d_u/\top$). Note that the "independent property" would

Algorithm 5: Operations for Directed Independent Dominating Set (LDM)

```
$leaf^{dom_l}(n)$:              return $\bigwedge_{x \in bag(n)} \neg d_x$
$remove^{dom_l}(n, B_1)$:       $\{u\} = removed(n)$
                                $n_1 = firstChild(n)$
                                $B_{ind} = \bigwedge_{\{u,y\} \in edges(n_1)} (\neg i_u \vee \neg i_y)$
                                $B_{dom} = \bigwedge_{y \in bag(n)} (d_y \leftrightarrow d'_y \vee_{(u,y) \in edges(n_1)} i_u) \wedge$
                                        $(d_u \leftrightarrow d'_u \vee \bigvee_{(x,u) \in edges(n_1)} i_x)$
                                $B = \exists D(n_1)'(B_1[D(n_1)/D(n_1)'] \wedge B_{ind} \wedge B_{dom})$
                                return $B[i_u/\top, d_u/\bot] \vee B[i_u/\bot, d_u/\top]$
$intro^{dom_l}(n, B_1)$:        $\{u\} = introduced(n)$
                                return $B_1 \wedge \neg d_u$
$join^{dom_l}(n, B_1, B_2)$:    $B_{dom} = \bigwedge_{x \in bag(n)} (d_x \leftrightarrow d'_x \vee d''_x)$
                                $B = B_1[D(n)/D(n)'] \wedge B_2[D(n)/D(n)''] \wedge B_{dom}$
                                return $\exists D(n)' \exists D(n)'' B$
```

be violated in case u is both in the extension and defeated. In $intro^{dom_e}$, for introduced vertex u the formula is constructed as in leaf nodes. In order to update the truth value of variables representing the domination of vertices, for any vertex y we apply a general pattern of $\exists y'[B_1[y/y'] \wedge (y \leftrightarrow (y' \vee cond))]$, that is, renaming, potentially adding conditions ($cond$), and removing the renamed variable y' by existential quantification. Here, $cond$ contains i_u in case u is an incoming neighbor of y. With this, the size of the BDDs remains bounded by the width of the decomposition. Finally, in $join^{dom_e}$, a vertex is "dominated" if it was in one of the child nodes. This information is again propagated via renaming, equivalence, and existential quantification.

LDM. A very compact algorithm for Directed Independent Dominating Set can be specified following the LDM paradigm (see Algorithm 5). Introduced vertices cannot become dominated in leaf or introduction nodes, and the corresponding variables are initialized with $\bot$. In $remove^{dom_l}$, we guess whether the removed vertex u is in the dominating set or dominated. Furthermore, we guarantee independence with vertices adjacent to u. A vertex y becomes dominated if there is some incoming edge from u and u is in the dominating set, and u is dominated if it was already dominated by an already-removed vertex, or by a vertex on an arc in $edges(n_1)$. Note that we use a small disjunction symbol with condition whenever there is at most one disjunction in the instantiated formula, and a large symbol otherwise. Finally, $join^{dom_l}$ is specified as in the EDM variant.

3.2.3 Hamiltonian Cycle

Input:	A simple graph $G = (V, E)$.
Question:	Is there a Hamiltonian cycle $F \subseteq E$ in G?

Here, a more involved algorithm specification is required. Monolithic propositional encodings (where the whole instance is available at once) allow one to assign a global order over the variables that specifies the ordering over the vertices in the cycle. However, in our DP-based approach, we are restricted to information that is available in the current decomposition node. Hence, we consider a *relative* ordering as follows. We use variables $S = I \cup O \cup T \cup A \cup I' \cup O' \cup A' \cup I'' \cup O'' \cup A''$, where an assignment of true to variables in $I = \{i_x \mid x \in V\}$ ($O = \{o_x \mid x \in V\}$) specifies that a variable x has exactly one incoming (outgoing) edge. Variables in $T = \{t_{xy} \mid \{x, y\} \in E\}$ represent edges that are selected on a cycle. Additionally, we have to guarantee that we have a *single* cycle that covers all vertices. Therefore we select a fixed vertex $f \in V$ that denotes where the cycle starts and ends. Variables in $A = \{a_{xy} \mid x, y \in V\}$ denote that x lies after y on the path from f to f. Primed variables are again used to temporarily hold changing information from the child node(s). For a tree decomposition node n we have $C_{\mathcal{T}}(n) = \{i_x, o_x, a_{xy} \mid x, y \in X_t\}$. Furthermore, for a vertex $u \in bag_{\mathcal{T}}(n)$, let $T_{\mathcal{T}}(n, u) = \{t_{xu}, t_{ux} \mid \{x, u\} \in edges_{\mathcal{T}}(n)\}$. In the following we only present the LDM version, but the concepts can be directly carried over to an EDM-based algorithm.

LDM. Algorithm 6 illustrates the operations for Hamiltonian Cycle. In leaf and introduction nodes all changing variables are initialized with $\perp$. In removal nodes, at least one incoming edge for removed vertex u is selected. Here, i'_u is true iff the incoming neighbor of u was already removed from the bag. Furthermore, at most one incoming edge from $edges(n_1)$ is selected. Finally, if i'_u is true, we cannot select an additional incoming edge, and the incoming and outgoing edges for u have to be different. The same construction is used to guarantee exactly one outgoing edge for u. For vertices $x \in bag(n)$, i_x and o_x are updated in case u was a neighbor of x. Again, at most one incoming (outgoing) edge must be selected. For $x, z \in bag(n)$, a_{xz} becomes true if $u \neq f$ and u lies on the path between x and z. With this, we keep information on the path (from f to f), restricted to $bag(n)$, where the truth value of t_{xy}-variables represents selected edges in $edges(n)$ and a_{xy}-variables denote that x is before y on the path where intermediate vertices were already removed. Finally, in case a_{xx} for $x \neq f$ is true, we know that there is a cycle that does not cover f, and is therefore no Hamiltonian cycle. In join nodes, i_x, o_x and a_{xy} variables are propagated as usual. Here, whenever both i'_x and i''_x are true, due to the connectedness condition of tree decompositions and the fact that these variables are updated when a vertex is removed, x has two different incoming edges, and is hence not a solution. The same holds for outgoing edges.

Algorithm 6: Operations for Hamiltonian Cycle (LDM)

```
$leaf^{ham_l}(n)$:          return $\bigwedge_{x \in bag(n)}(\neg i_x \wedge \neg o_x) \wedge \bigwedge_{x,y \in bag(n)} \neg a_{xy}$
$remove^{ham_l}(n, B_1)$:   $\{u\} = removed(n)$
                            $n_1 = firstChild(n)$
                            $B_{i_u} = i'_u \vee \bigvee_{\{x,u\} \in edges(n_1)} t_{xu}$
                            $B_{o_u} = o'_u \vee \bigvee_{\{u,y\} \in edges(n_1)} t_{uy}$
                            $B_{s^1_u} = \bigwedge_{\substack{\{x',u\} \in edges(n_1) \wedge \\ \{x'',u\} \in edges(n_1) \wedge x' \neq x''}} \big(\neg(t_{x'u} \wedge t_{x''u}) \wedge \neg(t_{ux'} \wedge t_{ux''})\big)$
                            $B_{s^2_u} = \bigwedge_{\{x,u\} \in edges(n_1)} \big(\neg(i'_u \wedge t_{xu}) \wedge \neg(o'_u \wedge t_{ux}) \wedge \neg(t_{xu} \wedge t_{ux})\big)$
                            $B_{i_x} = \bigwedge_{x \in bag(n)} \big(i_x \leftrightarrow (i'_x \vee_{\{u,x\} \in edges(n_1)} t_{ux})\big)$
                            $B_{o_x} = \bigwedge_{x \in bag(n)} \big(o_x \leftrightarrow (o'_x \vee_{\{x,u\} \in edges(n_1)} t_{xu})\big)$
                            $B_{s_x} = \bigwedge_{\{x,u\} \in edges(n_1)} \big(\neg(i'_x \wedge t_{ux}) \wedge \neg(o'_x \wedge t_{xu})\big)$
                            $B_a = \bigwedge_{x,z \in bag(n)} \Big(a_{xz} \leftrightarrow a'_{xz} \vee_{u \neq f} \big((a'_{xu} \vee_{\{x,u\} \in edges(n_1)} t_{xu}) \wedge$
                                    $(a'_{uz} \vee_{\{u,z\} \in edges(n_1)} t_{uz})\big)\Big)$
                            $B_{noc} = \bigwedge_{x \in bag(n) \wedge x \neq f} \neg a_{xx}$
                            $B = B_{i_u} \wedge B_{o_u} \wedge B_{s^1_u} \wedge B_{s^2_u} \wedge B_{i_x} \wedge B_{o_x} \wedge B_{s_x} \wedge B_a \wedge B_{noc}$
                            return $\exists T(n_1, u) \exists C(n_1)'(B_1[C(n_1)/C(n_1)'] \wedge B)$
$intro^{ham_l}(n, B_1)$:    $\{u\} = introduced(n)$
                            return $B_1 \wedge \neg i_u \wedge \neg o_u \wedge \bigwedge_{x \in bag(n)}(\neg a_{xu} \wedge \neg a_{ux})$
$join^{ham_l}(n, B_1, B_2)$: $B_{io} = \bigwedge_{x \in bag(n)} \Big(\big(i_x \leftrightarrow (i'_x \vee i''_x)\big) \wedge \big(o_x \leftrightarrow (o'_x \vee o''_x)\big) \wedge$
                                    $\neg(i'_x \wedge i''_x) \wedge \neg(o'_x \wedge o''_x)\Big)$
                            $B_a = \bigwedge_{x,y \in bag(n)} \big(a_{xy} \leftrightarrow (a'_{xy} \vee a''_{xy})\big)$
                            $B = B_1[C(n)/C(n)'] \wedge B_2[C(n)/C(n)''] \wedge B_{io} \wedge B_a$
                            return $\exists C(n)' \exists C(n)'' B$
```

4 QBF Solving

Here, we elaborate on a novel approach for QBF solving. We again exploit the parameter treewidth, which is obtained from the matrix of a given QBF in prenex CNF. The resulting algorithm is designed to be particularly efficient on tree-like QBF instances with a low number of quantifier alternations. For other logical problems, e.g. algorithms for SAT [96] and CSP [95], such treewidth-based algorithms have already been presented in the literature. Similar to the algorithms presented in Section 3, we make use of BDDs for efficient storage of intermediate results. However, we now store so-called *nested sets of formulae* (NSF) where the formulae are represented by BDDs, and the nestings of the BDDs handle quantifier alternations in the QBF (see Section 4.2).

The main procedure for tree decomposition-based QBF solving is given in Section 4.3.

Overall, in our approach the size of each BDD is bounded by the width of the used decomposition, and the overall number of BDDs required in each node is bounded by the width and by the number of quantifier alternations in the instance. Thus, the runtime depends exponentially on the structural parameter instead of the size of the formula. To be competitive with state-of-the-art systems on instances of low treewidth, we develop further techniques to reduce the size of the data structure (see Section 4.4). It turns out that our method already performs well on QBFs with one quantifier alternation, while for QBFs with a higher number of alternations our system does not reach the performance of state-of-the-art tools yet (see Section 5.3).

4.1 Basics

We briefly introduce QBFs and present the running example used in the subsequent sections. As usual, a *literal* is a variable or its negation. A *clause* is a disjunction of literals. A Boolean formula in conjunctive normal form (CNF) is a conjunction of clauses. Depending on the context, we will sometimes denote clauses as sets of literals, and a formula in CNF as a set of clauses. Herein, we consider Quantified Boolean Formulae (QBFs) in closed prenex CNF (PCNF) form.

Definition 6. *A PCNF QBF instance is of the form $Q.\psi$ where Q is the quantifier prefix and ψ is a CNF formula. The* quantifier prefix *Q is of the form $Q_1X_1Q_2X_2 \dots Q_kX_k$ where $Q_i \neq Q_{i+1}$ for $1 \leq i < k$. Furthermore, every variable in ψ occurs in exactly one set X_j for $1 \leq j \leq k$.*

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

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

The problem we are interested in states as follows.

Input:	A QBF instance $Q.\psi$ in prenex CNF form.
Question:	Is $Q.\psi$ satisfiable?

Example 3. *As our running example, we will consider QBF $Q.\psi$ with $Q = \exists ab\ \forall cd\ \exists ef$ and $\psi = (a \vee c \vee e) \wedge (\neg b \vee d) \wedge (e \vee f) \wedge (c \vee \neg e) \wedge (\neg d \vee f)$, which is satisfiable (for $a = \top$, $b = \bot$). Note that this example is designed to illustrate our approach. Hence, simplifications (e.g. pure literal elimination) are not considered.*

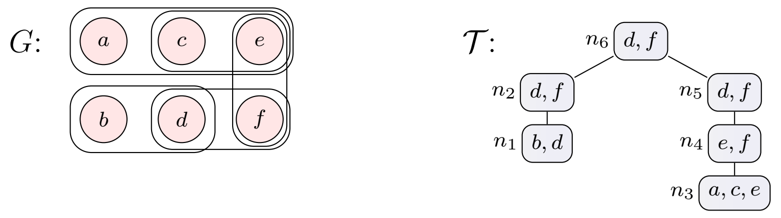


Figure 3: Graph G and possible (weakly-normalized) tree decomposition $\mathcal{T}$ of G.

In order to employ dynamic programming on tree decompositions for QBF solving, we have to construct a tree decomposition from the given QBF instance. Herein, we consider hypergraphs, i.e. graphs where the edges may have multiple endpoints. A QBF instance $Q.\psi$ can naturally be represented as a hypergraph $G = (V, E)$ where V are the variables occurring in $Q.\psi$ and for each clause $c \in \psi$, $\mathit{variables}(c)$ forms a hyperedge in G. Furthermore, clauses in $Q.\psi$ are related to nodes in the tree decomposition by $\mathit{clauses}_{\mathcal{T},\psi}(n) = \{c \mid c \in \psi, \mathit{variables}_\psi(c) \subseteq \mathit{bag}_\mathcal{T}(n)\}$. As usual, we will omit subscript $\mathcal{T}$ and ψ in the following. Note that this representation of CNF formulae is commonly used, e.g. for tree decomposition-based SAT solving [96]. The number of nodes in the tree decomposition is linear in the size of the QBF (i.e., its variables).

Example 4. *Given formula $\psi = (a \vee c \vee e) \wedge (\neg b \vee d) \wedge (e \vee f) \wedge (c \vee \neg e) \wedge (\neg d \vee f)$ of our running example, Figure 3 illustrates its hypergraph representation G, and $\mathcal{T}$ represents a weakly-normalized tree decomposition for ψ of width 2.*

4.2 Data Structure

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

Definition 7. *Given a QBF instance $Q.\psi$ with k quantifiers, we have a* nested set of formulae (NSF) *of depth k whose elements are inductively defined over the depth of nestings d with $0 \leq d \leq k$: for $d = 0$, the NSF is a BDD; for $1 \leq d \leq k$, the NSF is a set of NSFs of depth $d - 1$.*

For a QBF $Q.\psi$ with $Q = Q_1X_1 \dots Q_kX_k$ and an NSF N of depth k, for any NSF M appearing somewhere in N we denote by $\mathit{depth}(M)$ the depth of the nesting of M, $\mathit{level}_Q(M) = k - \mathit{depth}(M) + 1$ is the level of M, and $\mathit{quantifier}_Q(M) = Q_{\mathit{level}_Q(M)}$ (for $\mathit{level}_Q(M) \leq k$). Subscripts will be again omitted in the following.

Additionally, we define the procedure $\mathit{init}(k, \phi)$ that initializes an NSF of depth k, such that each set contains exactly one NSF, and the innermost NSF represents ϕ. For instance,

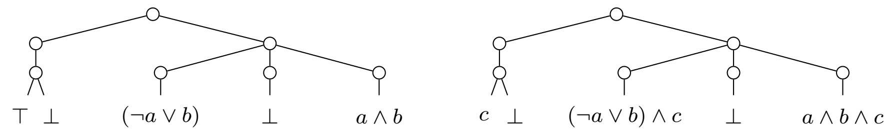


Figure 4: Example NSF N, represented as tree, and $N[B/B \wedge c]$ applied to N.

init$(3, \top)$ returns $\{\{\{\top\}\}\}$. Furthermore, for an NSF N we denote by $N[B/B']$ the replacement of each BDD B in N by some B'.

Example 5. *Suppose we have given an NSF* $N = \{\{\{\top, \bot\}\}, \{\{\neg a \vee b\}, \{\bot\}, \{a \wedge b\}\}\}$. *For the ease of readability, we will illustrate nested sets in form of a tree where the leaves contain the Boolean formulae represented by the BDDs, and each circle denotes a non-leaf NSF in the nestings with its contents being the children in the tree. Figure 4 shows the tree representing NSF* N *together with the one resulting from* $N[B/B \wedge c]$.

NSFs are tailored towards efficient representation of partial solution candidates. Opposed to the similar concept of quantifier trees [12], NSFs follow set semantics in order to automatically remove (trivial) redundancies. Furthermore, the depth of nestings is specified by the number of quantifiers, not by the number of variables in the instance. As we will see, NSFs can be directly used to keep track of parts of the solution space, instead of representing the whole QBF instance at once.

4.3 Main Procedure

Algorithm 7 illustrates the recursive procedure for the bottom-up traversal of the tree decomposition and computing the partial solution candidates. It is similar to the general procedure for problems in NP, see Algorithm 1. However, the algorithm supports weakly-normalized tree decompositions, and uses procedures particularly designed for QBF solving. When called with the root node of the tree decomposition, it returns an NSF that represents the overall solution to the problem.

Procedure *compute*(n) calls itself based on the child nodes of n. At each node, we distinguish between leaf, exchange and join nodes. In leaf nodes, an NSF of depth k (i.e., the number of quantifiers in the QBF instance) is initialized with the innermost set containing a BDD that represents the clauses associated with the current decomposition node. In an exchange node, we have to deal with removed as well as introduced variables (w.r.t. the bag's contents). First the NSF of the child node is computed. Then, removed variables are handled by "splitting" the NSF. Procedure *split*(N, x) (see Algorithm 8) handles this removal of a variable x. It is called recursively for the contents of the NSF until the level of x is reached. Then, for each NSF at this level, the NSF is updated once by replacing all occurrences of x in the BDDs with $\top$, and once with $\bot$. Due to the connectedness property of the tree decomposition, we know that a removed variable will never reappear somewhere upwards the tree decomposition, and therefore all clauses related to the removed variable were

Algorithm 7: Recursive procedure *compute*(n) for QBF solving

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

Algorithm 8: Recursive procedure *split*(N, x)

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

Algorithm 9: Recursive procedure *join*(N_1, N_2)

```
Input  : NSFs $N_1$ and $N_2$ of same depth
Output: A joined NSF
if $depth(N_1) = 0$ then
    return $N_1 \wedge N_2$
else
    return $\{join(M_1, M_2) \mid M_1 \in N_1, M_2 \in N_2\}$
```

already considered. Thereby we are also guaranteed that the size of each BDD is bounded by the bag's size. After splitting, the BDDs in the NSF are updated by adding the clauses associated with the current node via conjunction to the BDDs in the NSF. In join nodes,

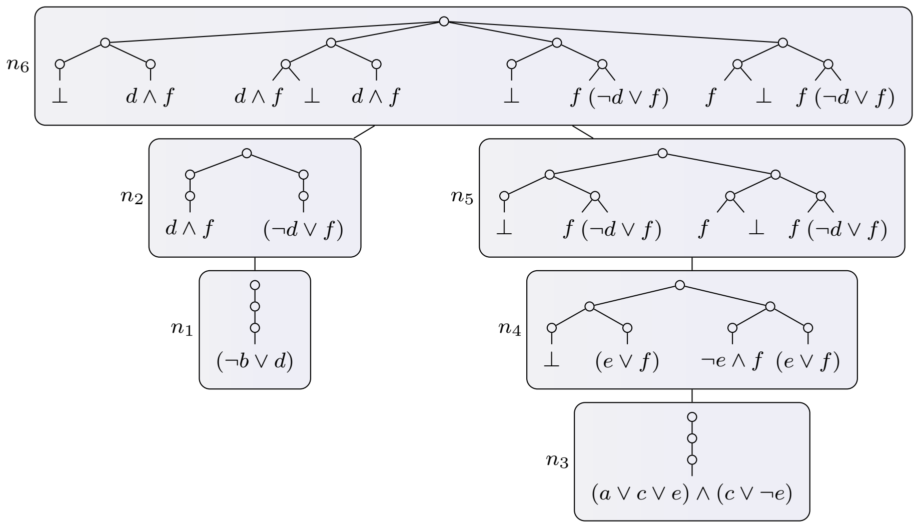


Figure 5: Computed NSFs for our running example.

NSFs computed in the child nodes are successively combined by procedure *join*(N_1, N_2) (see Algorithm 9). Observe that the procedure guarantees that the structure (nesting) of the NSFs to be joined is preserved. BDDs in the NSFs are then combined via conjunction, thus already considered information (i.e., clauses of the sub-hypergraph induced by the subtree's bag) of both child tree decomposition nodes is combined. Note that this procedure does not take the quantifiers of the QBF instance into account. They will be evaluated on the NSF of the root node.

Example 6. *Figure 5 shows the NSFs computed at the tree decomposition nodes of our running example. In n_1, an NSF of depth 3 is initialized with $(\neg b \vee d)$, i.e., the clause associated with this tree decomposition node. In n_2 variable b is removed. Hence the NSF is split at level$(b) = 1$, once by setting b to true (left NSF branch), yielding formula d and once by false (right branch), yielding $\top$. Furthermore, the current clause $(\neg d \vee f)$ is added to these BDDs via conjunction. Similarly, the right branch of the tree decomposition (nodes n_3–n_5) is computed. In n_6, the NSFs are joined. For instance, the leftmost branches in n_2 and n_5 are joined by conjunction of $d \wedge f$ and $\bot$, yielding$\bot$.*

Obtaining the solution. At the root node r of the tree decomposition, we can decide the problem since the whole input instance was taken into account. We apply quantifier elimination by evaluating the NSF as shown in Algorithm 10, which is similar to the approach described in [90]. Procedure *evaluateQ*(r, N) recursively combines the elements of the NSF by disjunction (for existential quantifiers) or conjunction (for universal quantifiers), starting at the innermost NSFs. Furthermore, variables contained in the current bag are abstracted away from the merged BDD according to the quantifier. Thus, this procedure finally returns a single BDD B without variables. There, if $B \equiv \bot$, the QBF instance is unsatisfiable, otherwise it is satisfiable.

```
Algorithm 10: Recursive procedure evaluateQ(n, N)
Input  : A tree decomposition node $n$ and an NSF $N$
Output: A BDD $B$ of $N$, obtained by evaluating the quantifiers
if $depth(N) = 0$ then
    $B := N$
else
    $X := \{x \mid x \in bag(n)$ and $level(x) = level(N)\}$
    if $quantifier(N) = \exists$ then
        $B := \exists X \bigvee_{M \in N} evaluateQ(n, M)$
    else if $quantifier(N) = \forall$ then
        $B := \forall X \bigwedge_{M \in N} evaluateQ(n, M)$
return $B$
```

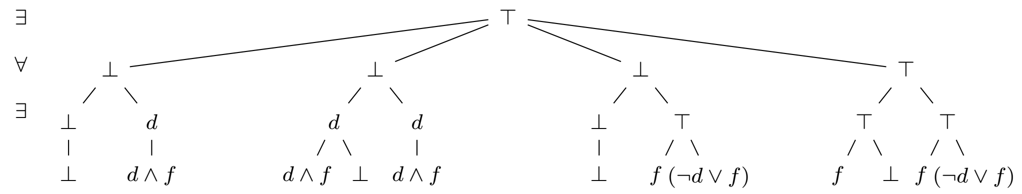


Figure 6: Results for *evaluateQ*(n_6, N) executed on the NSF of root node n_6.

Example 7. *Figure 6 shows the NSF N in root node n_6 of our running example, and the BDDs obtained recursively (bottom-up) when applying evaluateQ(r, N). Note that bag$(n_6) = \{d, f\}$ with level$(d) = 2$ and level$(f) = 3$, which are additionally taken into account when evaluating the quantifiers. The procedure returns $\top$ for our running example, hence the QBF is satisfiable.*

We omit a formal proof of the correctness of the proposed algorithm; instead we give an informal discussion about its runtime. Given a QBF $Q.\psi$ with $Q = Q_1X_1 \dots Q_kX_k$ and a tree decomposition for ψ of width w, the algorithm determines the truth of $Q.\psi$ in time $\mathcal{O}(2^{2^{\cdot^{\cdot^{2^{w+1}}}}} \cdot |\psi|)$, where the height of the tower of exponents in $2^{2^{\cdot^{\cdot^{2^{w+1}}}}}$ is $k+1$, since the size of each BDD is at most 2^{w+1} and we have k quantifiers[1]. Furthermore, $|\psi|$ denotes the size of ψ. We recall that the number of nodes of a tree decomposition is linear in the size of ψ. Moreover, any BDD involved in the algorithm is given over at most w variables and NSFs are just built upon such BDDs. Thus all operations on NSFs are also bound by w. Recall that due to the canonical form of BDDs, there are no duplicates at any level of an NSF, thus yielding this runtime.

[1] It is known that QSAT can be solved in FPT time when treewidth and number of quantifiers are bounded, which follows, for instance, from [35]. However, the QSAT problem is *not* fixed-parameter tractable w.r.t. parameter treewidth [8], unless the number of quantifiers is also bound or, more generally, the dependencies between variables are restricted (see [47]).

4.4 Optimizations

Algorithm 11: Recursive procedure *removeInnermostQuantifier*(N)

```
Input  : An NSF $N$
Output: An NSF with combined BDDs for innermost quantifier
if $depth(N) = 1$ then
    if $quantifier(N) = \exists$ then
        $B := \bigvee_{M \in N} M$
    if $quantifier(N) = \forall$ then
        $B := \bigwedge_{M \in N} M$
    return $\{B\}$
else
    return $\{removeInnermostQuantifier(M) \mid M \in N\}$
```

Algorithm 12: Recursive procedure *removeRedundant*(N)

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

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

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

Evaluate innermost quantifier. For any NSF N at depth one, the quantifier can be evaluated immediately: Procedure *removeInnermostQuantifier*(N) (see Algorithm 11) specifies that for an NSF N the NSFs at depth one are sets containing exactly one BDD. This BDD is constructed by disjunction (for existential quantification) or conjunction (for universal quantification) of the original BDDs. Thereby, the overall size can be reduced, since usually the single BDD stores models more efficiently than several BDDs. Furthermore, redundant models (i.e., models that are stored in several BDDs) are now only kept once, and for universal quantification additionally only models appearing in all BDDs are stored in the newly created BDD.

Remove redundant NSFs. Redundant NSFs can be removed by checking for subsets w.r.t. models represented by the BDDs (similar to subsumption checking [17]), and subsets w.r.t. nested sets. Procedure *removeRedundant*(N) (see Algorithm 12) gives the pseudo-code for removing unnecessary elements.

Example 8. *Figure 7 shows some NSF before and after the application of removeRedundant*(N)*. For instance, consider the leftmost branch of the NSF at depth 1, which is existentially quantified. It contains two NSFs, i.e. the BDDs* $\perp$ *and* $\neg a$*: since* $\perp \vee \neg a = \neg a$*,* $\perp$ *is removed. Next, at depth 2, we subsequently have NSFs* $\{\neg a\}$ *and* $\{\neg a, c\}$*. Since* $\{\neg a\} \subseteq \{\neg a, c\}$*,* $\{\neg a, c\}$ *is removed. Similarly, in the right branch at depth 1,* $a \vee (a \vee c) = a$*, and* $(a \vee c)$ *is removed.*

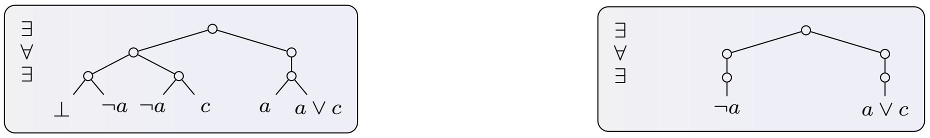


Figure 7: Example NSF before (left) and after (right) compression with procedure *removeRedundant*(N).

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

Example 9. *Figure 8 shows the NSFs computed at the tree decomposition nodes of our running example. Compared to Figure 5, here our algorithm optimizations were taken into account. For instance, due to immediate evaluation of the innermost quantifier, the sets at level three only contain one BDD. Another example would be the NSF in n_4, where in the left branch the NSF containing BDD $(e \vee f)$ (cf. Figure 5) is removed since its models are a superset of $\bot$.*

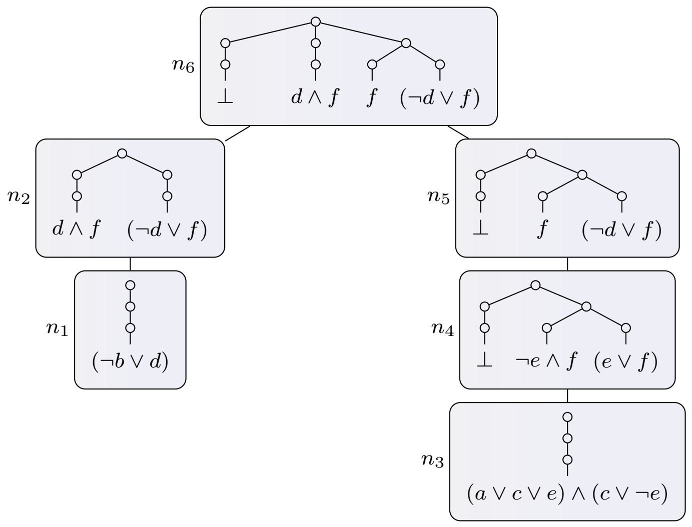


Figure 8: Computed NSFs (including optimizations) for our running example.

5 Implementation and Experimental Evaluation

The algorithms for problems in NP (see Section 3) were designed as case studies that illustrate how BDD-based dynamic programming algorithms proceed. We additionally developed a proof-of-concept system, called *dynBDD*, that puts these algorithms into practice. In [33] we showed that our BDD-based approach can indeed outperform current tree decomposition-based approaches, as implemented in D-FLAT [1], dynPARTIX [30] and Sequoia [68]. In particular, the reduction in memory requirements was demonstrated. It thus provided the basis for our QBF solver *dynQBF*, which we analyze in this section. The solver incorporates lessons learned from dynBDD, and implements the concepts and optimizations presented in Section 4. We compare dynQBF to state-of-the-art QBF solvers. Several of these solvers successfully participated in the 2016 QBF competition [91]. Additionally, we identify particular instances where our approach is superior to standard QBF solving techniques.

5.1 The dynQBF System

The dynQBF system supports PCNF QBF instances in the QDIMACS [86] format. Besides deciding whether QBFs are satisfiable, it can enumerate the models in case the outermost quantifier is existential and the instance is satisfiable. In this case, variables bounded by the outermost quantifier are not abstracted away but kept during computation – thus the NSF size is no longer bounded by the width of the decomposition. Furthermore, the system provides various debug output options (monitoring the computation progress, giving detailed NSF information, or printing performance information). Besides the standard EDM algorithm (as presented in the previous section), also an LDM version of the algorithm is available. Additionally, we implemented a variant called *BDD (naive)*, where the formula is not decomposed, but instead given to a single BDD at once. This gives a hint of whether the overhead for constructing and traversing the decomposition pay off. Algorithm optimizations can be configured by enabling intermediate unsatisfiability checks, the interval of when the NSFs are checked for redundancies, the maximal NSF size (when it is reached, NSFs are no longer split and the removed variable is kept in a cache of the data structure for later removal), as well as the maximum BDD size (if it is reached, the NSF is always split, regardless of the maximum NSF size). These options allow the system user to individually tune the system depending on the QBF instance at hand, if necessary.

The system relies on external libraries for tree decomposition computation and BDD handling.

- htd [2]: This library provides a rich feature set, including various input graph representations (such as simple, directed, multi-, or hypergraphs, trees and paths). In particular, it supports labelled hypergraphs that are used in the dynQBF system to represent clauses (with the labels storing negation of atoms). As output, it provides a graph, path, tree or hypertree decomposition that can additionally be adapted to current needs (e.g., by normalizing the decomposition). The dynQBF system supports path and tree decompositions (a path decomposition is a tree decomposition without join nodes). Furthermore, the implemented bucket elimination algorithm for tree decomposition generation supports various heuristics, such as minimum fill, minimum degree and maximum cardinality search [23, 41]. All these features are utilized by dynQBF, and configurable via the command line interface.

- CUDD [97]: BDDs, ADDs and ZDDs are supported by this library. It is implemented in C, but also includes a C++ interface. CUDD is continuously developed for over 20 years, and one of the most widely used decision diagram packages. The library supports many variable reordering heuristics (such as (lazy) sifting, random, simulated annealing or genetic algorithms).

The C++ code of dynQBF is publicly available at `http://dbai.tuwien.ac.at/proj/decodyn/dynqbf`.

5.2 QBF Solvers: An Overview

Advances in QBF solving are reported and benchmarked in the regular QBF competition events. The latest 2016 event [91] lists more than 20 solvers (plus variants, obtained from different configurations or solvers in combination with preprocessing tools). A comprehensive overview on tools is given for example in the QBF Gallery 2013 and 2014 reports [60, 78], and in a survey article [79]. Here, we discuss several systems that participated in the 2016 competition, as well as some (earlier) BDD-based approaches.

DepQBF [75, 77] is a search-based solver that implements conflict-driven clause learning and solution-driven cube learning [55, 104]. The solver analyzes (in)dependencies between variables, that are to be exploited in the DPLL-based procedure for QBF solving. RAReQS [61] is based on a technique called counterexample abstraction refinement (CEGAR) [36]. The idea is to gradually expand the given formula into a propositional one, but contrary to other expansion-based approaches (e.g. QUBOS [10], that utilizes propositional SAT solvers; quantor [17]; or Nenofex [76], that expands QBFs in negation normal form (NNF)), the CEGAR approach mitigates the problem of space explosion by terminating the expansion process in time.

Regarding systems that are based on BDDs, EBDDRES [63] is a tool that was originally designed as a SAT solver that provides resolution proofs. The latest version also supports QBFs. Unfortunately, EBDDRES is only available as a 32 bit binary. Further systems include QBFBDD [9] (based on a DPLL procedure combined with BDDs to provide flexibility on the order of quantifiers in the QBF) and eBDD-QBF [88] (improving early quantification), as well as the quantifier-tree based tool sKizzo [13]. However, to the best of our knowledge, they are not publicly available.

Oftentimes, systems are combined with preprocessors. One widely-used tool is Bloqqer [18]. It implements many elimination techniques, such as literal, blocked clause or tautology elimination, and supports subsumption checking, variable expansion and equivalence detection. Other preprocessors include Hiqqer (a variant of Bloqqer) and sQueezeBF [54] (that tries to recover structure that got lost during the QBF translation to CNF).

5.3 dynQBF: Experimental Evaluation

The dynQBF system (version as of 2016-08-23) is tested in the following configuration: Tree decompositions are generated following the minimum fill heuristic, and are non-normalized. BDD variables are dynamically reordered using the lazy-sift heuristic. The computation checks for subsets after every fourth computation step (in particular, after joining NSFs), the maximum NSF size is set to 1000 BDDs, and each BDD is configured to contain less than 3000 nodes. Those parameters were chosen based on an evaluation preceding this report.

We compare the total runtime (i.e., including the time from program invocation, input parsing to decomposition generation and solving, until program termination) to DepQBF version 5.0 and RAReQS version 1.1. Additionally, we consider the BDD-based system EBDDRES version 1.2. Finally, we compare it to the naive self-implemented approach BDD (naive).

Tests were performed on a single core of an Intel Xeon E5-2637 processor with 3.5GHz running Debian 8.3 (kernel 3.16.0-4-amd64). Each run was limited to a runtime of 10 minutes (TO) and 16 GB of memory (MO).

Table 1: 2-QBF instances: System comparison. The table gives the overall number of solved, solved satisfiable and solved unsatisfiable instances, as well as the number of observed timeouts (TO) and memouts (MO), the number of uniquely solved instances, and the total user time required for solving.

System	Solved	SAT	UNSAT	TO	MO	Unique	Time (solved)
dynQBF	**143**	**125**	18	162	0	**57**	3577.8s
DepQBF	122	57	**65**	183	0	40	5670.5s
RAReQS	70	44	26	235	0	17	1080.2s
BDD (naive)	48	47	1	257	0	0	1722.4s
EBDDRES	32	31	1	0	273	1	165.3s

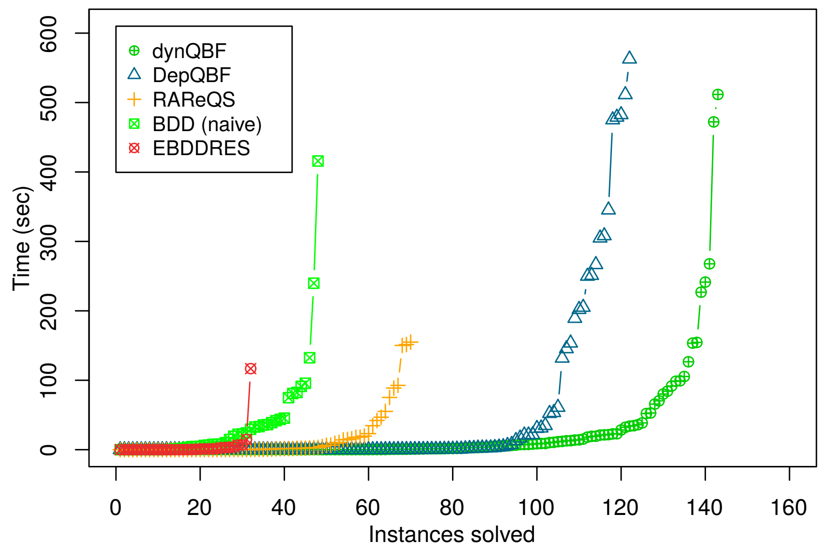


Figure 9: 2-QBF instances: Cactus plot of user time for compared systems. For each system, the solved instances are sorted by the required user time (in seconds).

5.3.1 2-QBF Instances

We used 305 publicly available 2-QBF (i.e., QBFs with a ∀∃ quantifier prefix) instances [91] (Dataset 3) of the QBFEval'16 competition. Table 1 reports on the total number of solved instances per system, and Figures 9 and 10 show the cactus plots for our benchmark runs. Here, dynQBF solved the most instances, followed by DepQBF. Observe that our BDD-based implementations are particularly successful on satisfiable instances, while DepQBF and RAReQS solved more unsatisfiable instances. EBDDRES is limited to roughly 4GB of memory (since it is only available as 32 bit binary), which explains the high number of memouts. However, Figure 10 shows that this system also requires more time for solved instances compared to the better-performing systems. Our baseline implementation BDD (naive) solves far less instances than dynQBF, indicating that the overhead for computing and traversing the tree decomposition pays off.

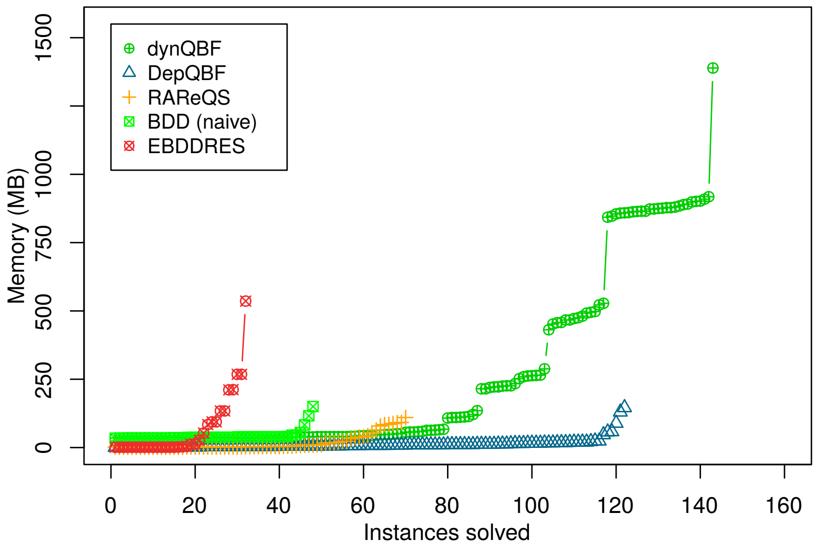


Figure 10: 2-QBF instances: Cactus plot of memory for compared systems. For each system, the solved instances are sorted by the required memory (in MB).

Interestingly, there are several instances that are uniquely solved by one of the compared systems (see Table 1, column "Unique"). This indicates that the solvers work particularly well for different types of instances. In order to give strengths and weaknesses of the systems, we analyzed the different classes of instances contained the 2-QBF dataset (identifiable by the instance file names). Table 2 shows an overview on the groups, the number of instances per group and average information on the number of atoms and clauses, as well as the average (heuristically computed) width of the tree decompositions. Group "other" collects instances that we could not assign to a particular group. Table 3 shows our results for each group individually. It appears that dynQBF was quite successful in groups "mutex*", "qshifter*", "stmt*" and "tree*", and solved approximately one third of the instances in "rankfunc*". These are exactly the groups of instances with lower width, but their average number of atoms and clauses is higher than in most of the other groups where dynQBF was less successful. This result is in line with design of our algorithm, which directly tries to exploit this structural parameter.

5.3.2 Prenex CNF Instances

Here, we tested the 825 available Prenex CNF instances [91] (Dataset 1) of the QBFEval'16. In Table 4 we summarize again the number of solved instances per system, solved satisfiable and unsatisfiable instances, as well as the measured timeouts and memouts. In Figure 11 a cactus plot of the required time for solving the instances is given, and Figure 12 shows the required memory. Here, dynQBF is not competitive with the other systems. In the following we analyze the instances to identify characteristics that could be responsible for the system's performance.

Table 2: 2-QBF Instances: Groups of instances in the QBFEval'16 dataset. The table groups the instances based on their file names. Furthermore, the total number of instances per group as well as the average number of atoms, clauses and the average width are given.

Group	#Instances	#Atoms (avg)	#Clauses (avg)	Width (avg)
mutex*	7	3224.00	4172.71	15.00
other	9	1769.56	10763.00	294.89
qshifter*	6	173.50	29120.00	89.50
query*	40	2423.45	10409.80	349.20
rankfunc*	50	3364.52	8795.12	152.40
sortnet*	42	4500.31	7531.60	2173.33
stmt*	146	4457.67	16509.19	53.51
tree*	5	40.00	38.00	2.00

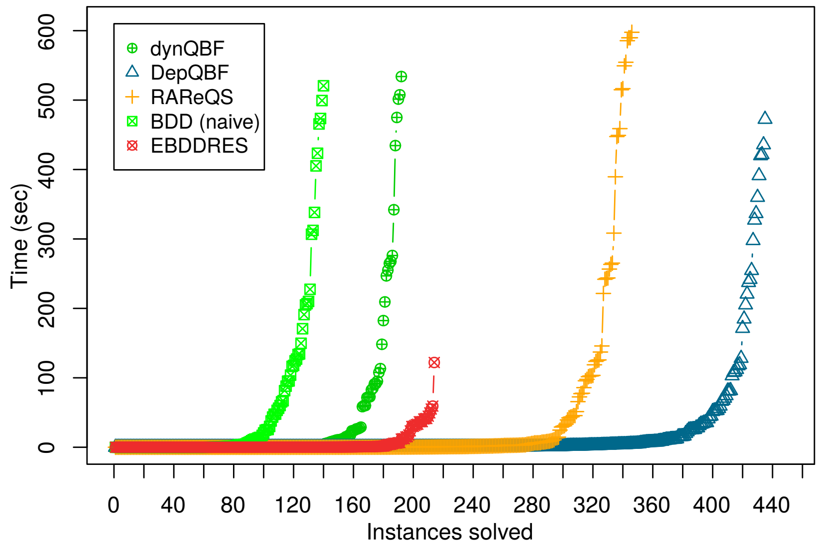


Figure 11: Prenex CNF instances: Cactus plot of user time for compared systems. For each system, the solved instances are sorted by the required user time (in seconds).

Table 5 gives details on the benchmark tests for dynQBF. 84 instances could not even be decomposed within the time limit. These instances are very large, with an average number of more than 160000 atoms and 500000 clauses. Here, a faster tree decomposition heuristic (e.g. minimum degree instead of minimum fill) could be used. However, the resulting decompositions then typically exhibit a higher width. The decomposed instances are (on average) much smaller, but have a very high width (448.58) and a high number of quantifier alternations (20.16). It is not surprising that many of these instances could not be solved with our approach, since our algorithm runs (in the worst case) exponentially both in the width and the number of quantifier alternations. Regarding the 192 solved instances (with an average width of 33.98 and 8.53 quantifier alternations) it becomes evident that these

Table 3: 2-QBF instances: System comparison, grouped by instance types.

Group	System	Solved	SAT	UNSAT	TO	MO	Unique
mutex*	DepQBF	3	3	0	4	0	0
mutex*	**dynQBF**	**7**	**7**	0	0	0	0
mutex*	BDD (naive)	1	1	0	6	0	0
mutex*	**EBDDRES**	**7**	**7**	0	0	0	0
mutex*	**RAReQS**	**7**	**7**	0	0	0	0
other	**DepQBF**	**6**	**3**	**3**	3	0	0
other	dynQBF	3	2	1	6	0	0
other	BDD (naive)	2	2	0	7	0	0
other	EBDDRES	1	1	0	0	8	0
other	**RAReQS**	**6**	**3**	**3**	3	0	0
qshifter*	DepQBF	3	3	0	3	0	0
qshifter*	dynQBF	5	5	0	1	0	0
qshifter*	BDD (naive)	2	2	0	4	0	0
qshifter*	**EBDDRES**	**6**	**6**	0	0	0	**1**
qshifter*	RAReQS	1	1	0	5	0	0
query*	DepQBF	9	3	6	31	0	0
query*	dynQBF	1	1	0	39	0	0
query*	BDD (naive)	0	0	0	40	0	0
query*	EBDDRES	0	0	0	0	40	0
query*	**RAReQS**	**17**	**10**	**7**	23	0	**8**
rankfunc*	**DepQBF**	**22**	**21**	1	28	0	**10**
rankfunc*	dynQBF	18	17	1	32	0	6
rankfunc*	BDD (naive)	1	0	1	49	0	0
rankfunc*	EBDDRES	1	0	1	0	49	0
rankfunc*	RAReQS	0	0	0	50	0	0
sortnet*	DepQBF	18	9	9	24	0	0
sortnet*	dynQBF	0	0	0	42	0	0
sortnet*	BDD (naive)	0	0	0	42	0	0
sortnet*	EBDDRES	0	0	0	0	42	0
sortnet*	**RAReQS**	**27**	**11**	**16**	15	0	**9**
stmt*	DepQBF	56	10	**46**	90	0	30
stmt*	**dynQBF**	**104**	**88**	16	42	0	**51**
stmt*	BDD (naive)	37	37	0	109	0	0
stmt*	EBDDRES	12	12	0	0	134	0
stmt*	RAReQS	7	7	0	139	0	0
tree*	**DepQBF**	**5**	**5**	0	0	0	0
tree*	**dynQBF**	**5**	**5**	0	0	0	0
tree*	**BDD (naive)**	**5**	**5**	0	0	0	0
tree*	**EBDDRES**	**5**	**5**	0	0	0	0
tree*	**RAReQS**	**5**	**5**	0	0	0	0

Table 4: Prenex CNF instances: System comparison. The table gives the overall number of solved, solved satisfiable and solved unsatisfiable instances, as well as the number of observed timeouts (TO) and memouts (MO), the number of uniquely solved instances, and the total user time required for solving.

System	Solved	SAT	UNSAT	TO	MO	Unique	Time (solved)
DepQBF	**435**	**188**	**247**	386	4	86	7655.67
RAReQS	346	136	210	479	0	38	10799.33
EBDDRES	214	116	98	7	604	41	913.42
dynQBF	192	94	98	619	14	11	6155.06
BDD (naive)	140	75	65	682	3	1	7092.65

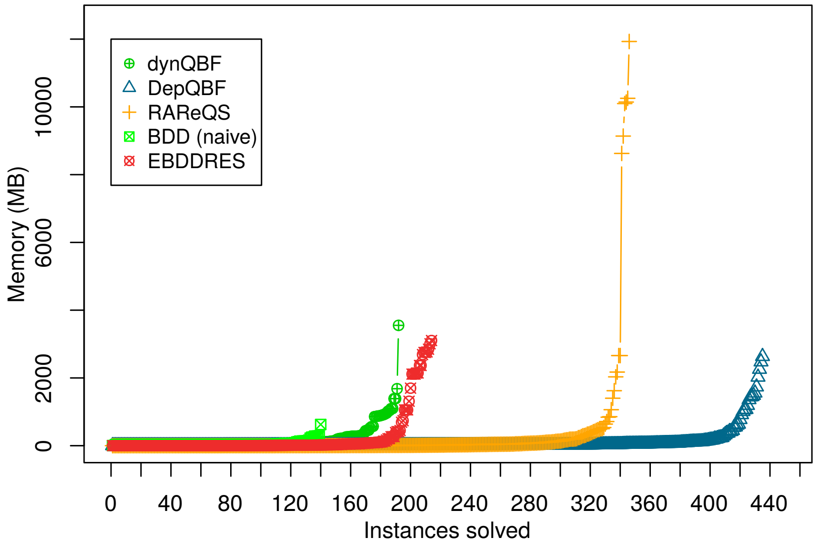


Figure 12: Prenex CNF instances: Cactus plot of memory for compared systems. For each system, the solved instances are sorted by the required memory (in MB).

instance (and decomposition) characteristics are important for our approach.

6 Conclusion

6.1 Summary

In this report, we developed a novel approach for dynamic programming on tree decompositions, where BDDs are used as a compact representation of intermediate results during the computation process. We first provided several case studies for NP-complete problems that are fixed-parameter tractable with respect to treewidth. There, a single BDD is sufficient to store results obtained in the decomposition nodes. We illustrated how various properties, that reoccur in many problem domains, can be implemented in our approach (such as con-

Table 5: Prenex CNF instances: Details for dynQBF. The table categorizes the instances based on the solving status, and reports on the number of instances, the average number of atoms, clauses, with and quantifiers per category, as well as the average decomposition (TD) and average overall (O) solving time.

Category	#	Atoms	Clauses	Width	Quant.	Time TD	Time O
not decomposed	84	162866.72	526224.11	-	8.63	-	-
decomposed	741	8363.42	34478.75	448.58	20.16	18.49	-
- not solved	549	9907.89	43548.67	592.06	24.22	23.94	-
- solved	192	3900.71	8271.45	33.98	8.53	2.77	32.06
- SAT	94	1408.80	6521.31	30.85	6.40	0.05	26.77
- UNSAT	98	6340.71	9985.14	37.04	10.56	5.44	37.13

nectedness for Hamiltonian Cycle or changing information (in Directed Independent Dominating Set)).

These case studies paved the way to our technique of solving the QSAT problem, which is PSPACE-complete, but known to be fixed-parameter tractable for a fixed number of quantifier alternations plus treewidth. We developed a dedicated data structure, called *nested set of formulae* (NSF), where the nestings account for quantifier alternations in the QBF instance, and BDDs compactly store partial assignments to variables in the QBF instance. Despite providing an fpt algorithm, a direct implementation is still infeasible in practice. Hence, we developed various optimizations for our approach. Most importantly, the NSF data structure can be reduced in size by removing redundant nestings or BDDs. The number of elements in an NSF and the maximum size of the BDDs can be balanced by delaying removal of variables from the BDDs. Additionally, checking for unsatisfiability during the tree decomposition traversal allows us to abort the computation earlier.

In our experimental evaluation we compared the developed system *dynQBF* to state-of-the-art QBF solvers (DepQBF, RAReQS), as well as an earlier BDD-based approach for QBF solving (EBDDRES). Additionally, we analyzed whether the overhead of computing and traversing the tree decomposition pays off (BDD (naive)).

6.2 Discussion

Our approach provides a novel method of specifying dynamic programming algorithms on tree decompositions. They are given as Boolean formula manipulation operations, and define how *sets* of partial solution candidates are computed (contrary to table-based approaches). Additionally, we shift the challenging task of maintaining an efficient data structure from the algorithm developer to the BDDs, which is natively supported by current ROBDD libraries.

Regarding QBF solving, we also rely on BDDs, but additionally provide algorithms that are required to keep our NSFs compact. Experiments showed the potential of this method for $\forall\exists$-QBFs, revealing that this method appears to be well suited for particular classes of QBFs that are hard to solve for other systems. This is in line with observations for SAT solving, where resolution-based solving can outperform search (DPLL-like) solving for instances of small width [93]. However, for formulas with more alternations and higher width, further improvements are necessary to be competitive with state-of-the-art solvers. One explanation for the rather poor performance of our method on more involved formulas is that we have not considered any analysis of quantifier dependencies in our algorithm yet.

Different QBF solving techniques have their respective merits on different types of instances. This could be important in the design of portfolio QBF solvers. Additionally, our ideas could help to improve other systems by also taking the structural property treewidth explicitly into account.

6.3 Future Work and Open Issues

The current implementation of dynQBF does not include any preprocessing. For instance, simply by splitting large clauses into smaller ones, the width of the decomposition could be reduced. Additionally, the aforementioned analysis of quantifier dependencies could be implemented in a preprocessing step. Besides that, standard techniques (such as blocked clause elimination or equivalence detection) could be added to the system.

Furthermore, we still need a better understanding of the interplay between different variable orderings in the BDDs and the shape of the used decomposition. Since good variable orderings are crucial to keep the size of the BDDs low, we expect that such insights can also lead to significant improvements on certain instances. Furthermore, we could study how the shape of the tree decomposition influences the solving process. Currently, we aim at decompositions that exhibit a low width, but other properties could be important as well. For instance, it could be beneficial to have variables with a high quantification level near the leaf nodes of the tree decomposition.

In principle, techniques from other dynamic programming systems, such as lazy solving [20], could be adapted and integrated into our approach. Also incremental solving (as e.g. available for DepQBF) could be useful, and be applied, for instance, by dynamically extending the tree decomposition. Furthermore, our ideas of BDD-based dynamic programming could be integrated into other systems.

References

[1] Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, Markus Hecher, and Stefan Woltran. The D-FLAT system for dynamic programming on tree decompositions. In *Proc. of the 14th European Conference on Logics in Artificial Intelligence (JELIA 2014)*, volume 8761 of *LNCS*, pages 558–572. Springer, 2014.

[2] Michael Abseher, Nysret Musliu, and Stefan Woltran. htd – a free, open-source framework for tree decompositions and beyond. Technical Report DBAI-TR-2016-96, DBAI, Fakultät für Informatik an der Technischen Universität Wien, 2016.

[3] Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 100(6):509–516, 1978.

[4] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8:277–284, 1987.

[5] Stefan Arnborg, Jens Lagergren, and Detlef Seese. Easy problems for tree-decomposable graphs. *J. Algorithms*, 12(2):308–340, 1991.

[6] Stefan Arnborg and Andrzej Proskurowski. Linear time algorithms for np-hard problems restricted to partial k-trees. *Discrete Applied Mathematics*, 23(1):11–24, 1989.

[7] Bengt Aspvall, Jan Arne Telle, and Andrzej Proskurowski. Memory requirements for table computations in partial k-tree algorithms. *Algorithmica*, 27(3):382–394, 2000.

[8] Albert Atserias and Sergi Oliva. Bounded-width QBF is PSPACE-complete. *J. Comput. Syst. Sci.*, 80(7):1415–1429, 2014.

[9] Gilles Audemard and Lakhdar Sais. SAT based BDD solver for quantified Boolean formulas. In *Proc. of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004)*, pages 82–89. IEEE Computer Society, 2004.

[10] Abdelwaheb Ayari and David A. Basin. QUBOS: deciding quantified boolean logic using propositional satisfiability solvers. In *Proc. of the 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2002)*, volume 2517 of *LNCS*, pages 187–201. Springer, 2002.

[11] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206, 1997.

[12] Marco Benedetti. Quantifier trees for QBFs. In *Proc. of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, volume 3569 of *LNCS*, pages 378–385. Springer, 2005.

[13] Marco Benedetti. sKizzo: A suite to evaluate and certify QBFs. In *Proc. of the 20th International Conference on Automated Deduction (CADE 2005)*, volume 3632 of *LNCS*, pages 369–376. Springer, 2005.

[14] Umberto Bertelè and Francesco Brioschi. On non-serial dynamic programming. *J. Comb. Theory, Ser. A*, 14(2):137–148, 1973.

[15] Nadja Betzler, Rolf Niedermeier, and Johannes Uhlmann. Tree decompositions of graphs: Saving memory in dynamic programming. *Discrete Optimization*, 3(3):220–229, 2006.

[16] Dirk Beyer and Andreas Stahlbauer. BDD-based software verification – applications to event-condition-action systems. *STTT*, 16(5):507–518, 2014.

[17] Armin Biere. Resolve and expand. In *Proc. of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004), Revised Selected Papers*, volume 3542 of *LNCS*, pages 59–70, 2004.

[18] Armin Biere, Florian Lonsing, and Martina Seidl. Blocked clause elimination for QBF. In *Proc. of the 23rd International Conference on Automated Deduction (CADE 2011)*, volume 6803 of *LNCS*, pages 101–115. Springer, 2011.

[19] Bernhard Bliem, Günther Charwat, Markus Hecher, and Stefan Woltran. D-FLAT^2: Subset minimization in dynamic programming on tree decompositions made easy. *Fundamenta Informaticae*, 147:27–61, 2016. To appear.

[20] Bernhard Bliem, Benjamin Kaufmann, Torsten Schaub, and Stefan Woltran. ASP for anytime dynamic programming on tree decompositions. In *Proc. of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*, pages 979–986. AAAI Press, 2016.

[21] Bernhard Bliem, Reinhard Pichler, and Stefan Woltran. Implementing Courcelle's Theorem in a declarative framework for dynamic programming. *Journal of Logic and Computation*, 2016.

[22] Hans L. Bodlaender. Treewidth: Algorithmic techniques and results. In *International Symposium on Mathematical Foundations of Computer Science*, pages 19–36. Springer, 1997.

[23] Hans L. Bodlaender. Discovering treewidth. In *Proc. of the 31st Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2005)*, volume 3381 of *LNCS*, pages 1–16. Springer, 2005.

[24] Hans L. Bodlaender and Arie M. C. A. Koster. Combinatorial optimization on graphs of bounded treewidth. *Comput. J.*, 51(3):255–269, 2008.

[25] Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations I. Upper bounds. *Inf. Comput.*, 208(3):259–275, 2010.

[26] Karim Boutaleb, Philippe Jégou, and Cyril Terrioux. (No)good recording and ROBDDs for solving structured (V)CSPs. In *Proc. of the 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2006)*, pages 297–304. IEEE Computer Society, 2006.

[27] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.

[28] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986.

[29] Didier Buchs and Steve P. Hostettler. Sigma decision diagrams. In *Prelim. procs. of the 5th International Workshop on Computing with Terms and Graphs (TERMGRAPH 2009)*, 2009. ID: unige:12331.

[30] Günther Charwat and Wolfgang Dvořák. dynPARTIX 2.0 - dynamic programming argumentation reasoning tool. In *Proc. of the 4th International Conference on Computational Models of Argument (COMMA 2012)*, volume 245 of *FAIA*, pages 507–508. IOS Press, 2012.

[31] Günther Charwat, Wolfgang Dvořák, Sarah Alice Gaggl, Johannes Peter Wallner, and Stefan Woltran. Methods for solving reasoning problems in abstract argumentation - a survey. *Artif. Intell.*, 220:28–63, 2015.

[32] Günther Charwat and Stefan Woltran. Efficient problem solving on tree decompositions using binary decision diagrams. In *Proc. of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2015)*, volume 9345 of *LNCS*, pages 213–227. Springer, 2015.

[33] Günther Charwat and Stefan Woltran. Dynamic programming-based QBF solving. In *International Workshop on Quantified Boolean Formulas (and Beyond) (QBF 2016)*, 2016. To appear.

[34] Mark Chavira and Adnan Darwiche. Compiling Bayesian networks using variable elimination. In *Proc. of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 2443–2449, 2007.

[35] Hubie Chen. Quantified constraint satisfaction and bounded treewidth. In *Proc. of the 16th Eureopean Conference on Artificial Intelligence (ECAI 2004)*, pages 161–165. IOS Press, 2004.

[36] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[37] Bruno Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990.

[38] Jean-Michel Couvreur, Emmanuelle Encrenaz, Emmanuel Paviot-Adet, Denis Poitrenaud, and Pierre-André Wacrenier. Data decision diagrams for petri net analysis. In *Proc. of the 23rd International Conference on Applications and Theory of Petri Nets (ICATPN 2002)*, volume 2360 of *LNCS*, pages 101–120. Springer, 2002.

[39] Jean-Michel Couvreur and Yann Thierry-Mieg. Hierarchical decision diagrams to exploit model structure. In *Proc. of the 25th International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2005)*, volume 3731 of *LNCS*, pages 443–457. Springer, 2005.

[40] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms.* Springer, 2015.

[41] Rina Dechter. *Constraint Processing.* Morgan Kaufmann, 2003.

[42] Artan Dermaku, Tobias Ganzow, Georg Gottlob, Benjamin J. McMahan, Nysret Musliu, and Marko Samer. Heuristic methods for hypertree decomposition. In *Proc. of the 7th Mexican International Conference on Artificial Intelligence (MICAI 2008)*, volume 5317 of *LNCS*, pages 1–11. Springer, 2008.

[43] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity.* Monographs in Computer Science. Springer, 1999.

[44] Rolf Drechsler and Detlef Sieling. Binary decision diagrams in theory and practice. *STTT*, 3(2):112–136, 2001.

[45] Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321 – 357, 1995.

[46] Wolfgang Dvořák, Reinhard Pichler, and Stefan Woltran. Towards fixed-parameter tractable algorithms for abstract argumentation. *Artif. Intell.*, 186:1–37, 2012.

[47] Eduard Eiben, Robert Ganian, and Sebastian Ordyniak. Using decomposition-parameters for QBF: Mind the prefix! In *Proc. of the 13th AAAI Conference on Artificial Intelligence (AAAI 2016)*, pages 964–970. AAAI Press, 2016.

[48] Hélène Fargier and Pierre Marquis. Knowledge compilation properties of Trees-of-BDDs, revisited. In *Proc. of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 772–777, 2009.

[49] Johannes Klaus Fichte, Markus Hecher, Michael Morak, and Stefan Woltran. Counting answer sets via dynamic programming. In *Workshop on Trends and Applications of Answer Set Programming (TAASP 2016)*, 2016. To appear.

[50] Steven J. Friedman and Kenneth J. Supowit. Finding the optimal variable ordering for binary decision diagrams. In *Proc. of the IEEE Design Automation Conference (DAC 1987)*, pages 348–356. ACM, 1987.

[51] Masahiro Fujita, Patrick C. McGeer, and Jerry Chih-Yuan Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2-3):149–169, 1997.

[52] Graeme Gange, Peter J. Stuckey, and Pascal Van Hentenryck. Explaining propagators for edge-valued decision diagrams. In *Proc. of the 19th International Conference on Principles and Practice of Constraint Programming (CP 2013)*, volume 8124 of *LNCS*, pages 340–355. Springer, 2013.

[53] Michael Gelfond and Nicola Leone. Logic programming and knowledge representation – the A-Prolog perspective. *Artif. Intell.*, 138(1-2):3–38, 2002.

[54] Enrico Giunchiglia, Paolo Marin, and Massimo Narizzano. squeezebf: An effective preprocessor for qbfs based on equivalence reasoning. In *Proc. of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT 2010)*, volume 6175 of *LNCS*, pages 85–98. Springer, 2010.

[55] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Clause/term resolution and learning in the evaluation of quantified boolean formulas. *J. Artif. Intell. Res.*, 26:371–416, 2006.

[56] Georg Gottlob, Reinhard Pichler, and Fang Wei. Monadic datalog over finite structures of bounded treewidth. *ACM Trans. Comput. Log.*, 12(1):3, 2010.

[57] Martin Grohe. Descriptive and parameterized complexity. In *Proc. of the 13th International Workshop on Computer Science Logic (CSL 1999)*, volume 1683 of *LNCS*, pages 14–31. Springer, 1999.

[58] Rudolf Halin. S-functions for graphs. *Journal of Geometry*, 8(1-2):171–186, 1976.

[59] Petr Hlinený, Sang-il Oum, Detlef Seese, and Georg Gottlob. Width parameters beyond tree-width and their applications. *Comput. J.*, 51(3):326–362, 2008.

[60] Mikolas Janota, Charles Jordan, Will Klieber, Florian Lonsing, Martina Seidl, and Allen Van Gelder. The QBFGallery 2014: The QBF competition at the FLoC olympic games. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:187–206, 2016.

[61] Mikolás Janota, William Klieber, João Marques-Silva, and Edmund M. Clarke. Solving QBF with counterexample guided refinement. In *Proc. of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT 2012)*, volume 7317 of *LNCS*, pages 114–128. Springer, 2012.

[62] Geert Janssen. A consumer report on BDD packages. In *Proc. of the 16th Annual Symposium on Integrated Circuits and Systems Design (SBCCI 2003)*, page 217. IEEE Computer Society, 2003.

[63] Toni Jussila, Carsten Sinz, and Armin Biere. Extended resolution proofs for symbolic SAT solving with quantification. In *Proc. of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT 2006)*, volume 4121 of *LNCS*, pages 54–60. Springer, 2006.

[64] Timothy Kam, Tiziano Villa, Robert Brayton, and Alberto Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1-2):9–62, 1998.

[65] Peter Kissmann and Jörg Hoffmann. BDD ordering heuristics for classical planning. *J. Artif. Intell. Res.*, 51:779–804, 2014.

[66] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, Aarhus University, January 2001. Notes Series NS-01-1. Available from `http://www.brics.dk/mona/`. Revision of BRICS NS-98-3.

[67] Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *LNCS*. Springer, 1994.

[68] Joachim Kneis, Alexander Langer, and Peter Rossmanith. Courcelle's theorem - A game-theoretic approach. *Discrete Optimization*, 8(4):568–594, 2011.

[69] Yung-Te Lai, Massoud Pedram, and Sarma B. K. Vrudhula. EVBDD-based algorithms for integer linear programming, spectral transformation, and function decomposition. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 13(8):959–975, 1994.

[70] Yung-Te Lai and Sarma Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proc. of the 29th ACM/IEEE Design Automation Conference (DAC 1992)*, pages 608–613, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[71] Alexander Langer, Felix Reidl, Peter Rossmanith, and Somnath Sikdar. Evaluation of an MSO-solver. In *Proc. of the 14th Meeting on Algorithm Engineering & Experiments (ALENEX 2012)*, pages 55–63. SIAM / Omnipress, 2012.

[72] Alexander Langer, Felix Reidl, Peter Rossmanith, and Somnath Sikdar. Practical algorithms for MSO model-checking on tree-decomposable graphs. *Computer Science Review*, 13-14:39–74, 2014.

[73] C.Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, 1959.

[74] Jorn Lind-Nielsen. BuDDy – a BDD package. `https://sourceforge.net/projects/buddy/`. Accessed: 2016-09-14.

[75] Florian Lonsing, Fahiem Bacchus, Armin Biere, Uwe Egly, and Martina Seidl. Enhancing search-based QBF solving by dynamic blocked clause elimination. In *Proc. of the 20th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2015)*, volume 9450 of *LNCS*, pages 418–433. Springer, 2015.

[76] Florian Lonsing and Armin Biere. Nenofex: Expanding NNF for QBF solving. In *Proc. of the 11th International Conference on Theory and Applications of Satisfiability Testing (SAT 2008)*, volume 4996 of *LNCS*, pages 196–210. Springer, 2008.

[77] Florian Lonsing and Armin Biere. DepQBF: A dependency-aware QBF solver. *JSAT*, 7(2-3):71–76, 2010.

[78] Florian Lonsing, Martina Seidl, and Allen Van Gelder. The QBF gallery: Behind the scenes. *Artif. Intell.*, 237:92–114, 2016.

[79] Paolo Marin, Massimo Narizzano, Luca Pulina, Armando Tacchella, and Enrico Giunchiglia. An empirical perspective on ten years of QBF solving. In *Proc. of the 22nd RCRA International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion 2015 (RCRA 2015)*, volume 1451 of *CEUR Workshop Proceedings*, pages 62–75. CEUR-WS.org, 2015.

[80] Artur Męski, Wojciech Penczek, Maciej Szreter, Bożena Woiźna-Szcześniak, and Andrzej Zbrzezny. BDD-versus SAT-based bounded model checking for the existential fragment of linear temporal logic with knowledge: algorithms and their performance. *Autonomous Agents and Multi-Agent Systems*, 28(4):558–604, 2014.

[81] Robert Meolic. Biddy BDD package. `http://savannah.nongnu.org/projects/biddy/`. Accessed: 2016-09-14.

[82] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. of the 30th ACM/IEEE Design Automation Conference (DAC 1993)*, pages 272–277, 1993.

[83] Shin-ichi Minato. Zero-suppressed BDDs and their applications. *International Journal on Software Tools for Technology Transfer*, 3(2):156–170, 2001.

[84] Michael Morak, Nysret Musliu, Reinhard Pichler, Stefan Rümmele, and Stefan Woltran. A new tree-decomposition based algorithm for answer set programming. In *Proc. of the IEEE 23rd International Conference on Tools with Artificial Intelligence (ICTAI 2011)*, pages 916–918. IEEE Computer Society, 2011.

[85] Michael Morak, Reinhard Pichler, Stefan Rümmele, and Stefan Woltran. A dynamic-programming based ASP-solver. In *Proc. of the 12th European Conference on Logics in Artificial Intelligence (JELIA 2010)*, volume 6341 of *LNCS*, pages 369–372. Springer, 2010.

[86] Massimo Narizzano, Luca Pulina, and Armando Tacchella. QDIMACS standard, version 1.1, realeased on December 21, 2005. `http://www.qbflib.org/qdimacs.html`, 2005. Accessed: 2016-09-12.

[87] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms.* Oxford Lecture Series in Mathematics and its Applications. OUP, 2006.

[88] Oswaldo Olivo and E. Allen Emerson. A more efficient BDD-based QBF solver. In *Proc. of the 17th International Conference on Principles and Practice of Constraint Programming (CP 2011)*, volume 6876 of *LNCS*, pages 675–690. Springer, 2011.

[89] Jörn Ossowski. *JINC: a multi-threaded library for higher-order weighted decision diagram manipulation.* PhD thesis, University of Bonn, 2010.

[90] Guoqiang Pan and Moshe Y. Vardi. Symbolic decision procedures for QBF. In *Proc. of the 10th International Conference on Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *LNCS*, pages 453–467. Springer, 2004.

[91] Luca Pulina. QBFEval'16 – competitive evaluation of QBF solvers. `http://www.qbflib.org/qbfeval16.php`. Accessed: 2016-09-12.

[92] Luca Pulina and Armando Tacchella. A structural approach to reasoning with quantified boolean formulas. In *Proc. of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 596–602, 2009.

[93] Irina Rish and Rina Dechter. Resolution versus search: Two strategies for SAT. *J. Autom. Reasoning*, 24(1/2):225–275, 2000.

[94] Neil Robertson and Paul D. Seymour. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.

[95] Martin Sachenbacher and Brian C. Williams. Bounded search and symbolic inference for constraint optimization. In *Proc. of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 286–291. PBC, 2005.

[96] Marko Samer and Stefan Szeider. Algorithms for propositional model counting. *J. Discrete Algorithms*, 8(1):50–64, 2010.

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

[98] Yinglei Song, Chunmei Liu, Russell Malmberg, Fangfang Pan, and Liming Cai. Tree decomposition based fast search of rna structures including pseudoknots in genomes. In *Proc. of the Computational Systems Bioinformatics Conference (CSB 2005)*, pages 223–234. IEEE, 2005.

[99] Larry J. Stockmeyer and Albert R. Meyer. Word problems requiring exponential time (preliminary report). In *Proc. of the 5th annual ACM symposium on Theory of Computing (TOC 1973)*, pages 1–9. ACM, 1973.

[100] Sathiamoorthy Subbarayan. Integrating CSP decomposition techniques and BDDs for compiling configuration problems. In *Proc. of the 2nd International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2005)*, volume 3524 of *LNCS*, pages 351–365. Springer, 2005.

[101] Yann Thierry-Mieg, Denis Poitrenaud, Alexandre Hamez, and Fabrice Kordon. Hierarchical set decision diagrams and regular models. In *Proc. of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 5505 of *LNCS*, pages 1–15. Springer, 2009.

[102] Arash Vahidi. JDD – a pure Java BDD and Z-BDD library. `https://bitbucket.org/vahidi/jdd/wiki/Home`. Accessed: 2016-09-14.

[103] John Whaley. JavaBDD – a Java binary decision diagram library. `http://javabdd.sourceforge.net/`. Accessed: 2016-09-14.

[104] Lintao Zhang and Sharad Malik. Towards a symmetric treatment of satisfaction and conflicts in quantified boolean formula evaluation. In *Proc. of the 8th International Conference on Principles and Practice of Constraint Programming (CP 2002)*, volume 2470 of *LNCS*, pages 200–215. Springer, 2002.