TPLP **22** (1): 128–157, 2022. © The Author(s), 2021. Published by Cambridge University Press
doi:10.1017/S147106842100003X First published online 12 March 2021

Exploiting Database Management Systems and Treewidth for Counting[*]

JOHANNES K. FICHTE
UC Berkeley, Berkeley, CA, USA
(*e-mail:* johannes.fichte@berkeley.edu)

MARKUS HECHER, PATRICK THIER and STEFAN WOLTRAN
TU Wien, Vienna, Austria
(*e-mails:* hecher@dbai.tuwien.ac.at, thier@dbai.tuwien.ac.at, woltran@dbai.tuwien.ac.at)

submitted 20 May 2020; revised 7 February 2021; accepted 8 February 2021

Abstract

Bounded treewidth is one of the most cited combinatorial invariants in the literature. It was also applied for solving several counting problems efficiently. A canonical counting problem is #Sat, which asks to count the satisfying assignments of a Boolean formula. Recent work shows that benchmarking instances for #Sat often have reasonably small treewidth. This paper deals with counting problems for instances of small treewidth. We introduce a general framework to solve counting questions based on state-of-the-art database management systems (DBMSs). Our framework takes explicitly advantage of small treewidth by solving instances using dynamic programming (DP) on tree decompositions (TD). Therefore, we implement the concept of DP into a DBMS (PostgreSQL), since DP algorithms are already often given in terms of table manipulations in theory. This allows for elegant specifications of DP algorithms and the use of SQL to manipulate records and tables, which gives us a natural approach to bring DP algorithms into practice. To the best of our knowledge, we present the first approach to employ a DBMS for algorithms on TDs. A key advantage of our approach is that DBMSs naturally allow for dealing with huge tables with a limited amount of main memory (RAM).

KEYWORDS: dynamic programming, parameterized algorithmics, bounded treewidth, database systems, SQL, relational algebra, counting

1 Introduction

Counting solutions is a well-known task in mathematics, computer science, and other areas (Chakraborty *et al.* 2016; Domshlak and Hoffmann 2007; Gomes *et al.* 2009; Sang *et al.* 2005). In combinatorics, for instance, one characterizes the number of solutions to problems by means of mathematical expressions, for example, generating functions (Doubilet *et al.* 1972). One particular counting problem, namely, *model counting* (#Sat) asks to output the number of solutions of a given Boolean formula. While we stay in the realm of model counting, the findings of this work are also relevant for answer set programming. This is particularly true for tight programs (using, e.g., Clark's

[*] This is an extended version of a paper (Fichte *et al.* 2020) that appeared in the Proceedings of the 22nd International Symposium on Practical Aspects of Declarative Languages (PADL'20).

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

completion Clark 1977), but also interesting for applications of quantitative reasoning, solved by programs that are compiled to SAT with the help of tools like lp2sat (Janhunen 2006) or lp2acyc (Bomanson *et al.* 2016).

Model counting and variants thereof have already been applied for solving a variety of real-world applications and questions in modern society related to reasoning (Chakraborty *et al.* 2016; Choi *et al.* 2015; Dueñas-Osorio *et al.* 2017; Xue *et al.* 2012). Such problems are typically considered very hard, since #SAT is complete for the class #P (Bacchus *et al.* 2003; Roth 1996), that is, one can simulate any problem of the polynomial hierarchy with polynomially many calls (Toda 1991) to a #SAT solver. Taming this high complexity is possible with techniques from parameterized complexity (Cygan *et al.* 2015). In fact, many of the publicly available #SAT instances show good structural properties after using regular preprocessors like pmc (Lagniez and Marquis 2014), see Fichte *et al.* (2018b; 2019). By good structural properties, we mean that graph representations of these instances have reasonably small *treewidth*. The measure treewidth is a structural parameter of graphs which models the closeness of the graph of being a tree. Treewidth is one of the most cited combinatorial invariants studied in parameterized complexity (Cygan *et al.* 2015) and was subject to algorithmics competitions (Dell *et al.* 2018).

The observation, stated above, that various recent problem instances for #SAT have small treewidth, leads to the question whether a general framework that leverages treewidth is possible for counting problems. The general idea to develop such frameworks is indeed not new, since there are (a) specialized solvers such as dynQBF, gpuSAT, and fvs-pace (Charwat and Woltran 2019; Fichte *et al.* 2019; Kiljan and Pilipczuk 2018) as well as (b) general systems that exploit treewidth like D-FLAT (Bliem *et al.* 2016), Jatatosk (Bannach and Berndt 2019), and sequoia (Langer *et al.* 2012). Some of these systems explicitly use *dynamic programming (DP)* to directly exploit treewidth by means of so-called *tree decompositions (TDs)*, whereas others provide some kind of declarative layer to model the problem (and perform decomposition and DP internally). In this work, we solve (counting) problems by means of explicitly specified DP algorithms, where essential parts of the DP algorithm are specified in the form of SQL `SELECT` queries. The actual run of the DP algorithm is then delegated to our system `dpdb`, which employs *database management systems (DBMSs)* (Ullman 1989). This has not only the advantage of naturally describing and manipulating the tables that are obtained during DP but also allows `dpdb` to benefit from decades of database technology in form of the capability to deal with huge tables using a limited amount of main memory (RAM), dedicated database joins, query optimization, and data-dependent execution plans. Compared to other generic DP systems like D-FLAT (Bliem *et al.* 2016), our system `dpdb` uses relational algebra (SQL) for specifying DP algorithms, which is even competitive with specialized systems for model counting, and therefore applicable beyond rapid prototyping.

Contribution. We implement a system `dpdb` for solving counting problems based on dynamic programming on tree decompositions and present the following contributions. (i) Our system `dpdb` uses DBMSs to handle table operations needed for performing DP efficiently. The system `dpdb` is written in Python and employs PostgreSQL as DBMS, but can work with other DBMSs easily. (ii) The architecture of `dpdb` allows to solve general problems of bounded treewidth that can be solved by means of table operations (in form of relational algebra and SQL) on TDs. As a result, `dpdb` is a generalized framework for

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

DP on TDs, where one only needs to specify the essential and problem-specific parts of DP in order to solve (counting) problems. (iii) Finally, we show how to solve the canonical problem #Sat with the help of dpdb, where it seems that the architecture of dpdb is particularly well-suited. In more detail, we compare the runtime of our system with state-of-the-art model counters. We observe a competitive behavior and promising indications for future work.

Prior Work. This is an extended version of a paper (Fichte *et al.* 2020) that appeared at the 22nd International Symposium on Practical Aspects of Declarative Languages. The new material includes improved and extended examples as well as a detailed description of our DP algorithms and how these algorithms can be implemented for the system dpdb. Further, we added two new DP algorithms for the additional problems MaxSat and MinIDS, to demonstrate how to use dpdb: The problem MaxSat is similar to Sat, but consists of hard clauses that need to be satisfied as well as soft clauses. The goal of MaxSat is to compute the maximum number of soft clauses that can be satisfied using only assignments that also satisfy all the hard clauses. Further, problem MinIDS is a popular graph problem that aims for computing for a given graph, a set of vertices (called independent dominating set) such that there is no edge between these vertices and all the other vertices of the graph have an edge to at least one of the vertices in this set. Both problems can be easily extended to counting, where we require to compute the number of witnessing solutions. Finally, we added new experimental results, where we used the most recent version 12 of PostgreSQL as the underlying database management system, which operated on a ramdisk drive.

2 Preliminaries

We assume familiarity with the terminology of graphs and trees. For details, we refer to the literature and standard textbooks (Diestel 2012).

2.1 Boolean satisfiability

We define Boolean formulas and their evaluation in the usual way, cf., Kleine Büning and Lettman (1999). A literal is a Boolean variable x or its negation $\neg x$. A *CNF formula* φ is a set of *clauses* interpreted as conjunction. A clause is a set of literals interpreted as disjunction. For a formula or clause X, we abbreviate by $\mathrm{var}(X)$ the variables that occur in X. An *assignment* of φ is a mapping $I : \mathrm{var}(\varphi) \to \{0, 1\}$. The formula $\varphi(I)$ *under assignment* I is obtained by removing every clause c from φ that contains a literal set to 1 by I and removing from every remaining clause of φ all literals set to 0 by I. An assignment I is *satisfying* if $\varphi(I) = \emptyset$. *Problem* #Sat asks to output the number of satisfying assignments of a formula.

We also allow *equality formulas*, which are Boolean formulas, where variables are expressions using equality. In more detail, let d be a fixed constant over domain $\mathrm{dom}(v)$, where we call d *term constant*. Let v and v' be variables over some domain $\mathrm{dom}(v)$ and $\mathrm{dom}(v')$, where we call v and v' *term variables*. Then, an *equality formula* β is an expression of the form $v = d$ or $v = v'$. A *term assignment* J of equality formula β over term variables $\mathrm{tvar}(\beta)$ assigns each domain variable $v \in \mathrm{tvar}(\beta)$ a value over domain $\mathrm{dom}(v)$. The Boolean formula $\beta(J)$ *under term assignment* J is obtained as follows. First,

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

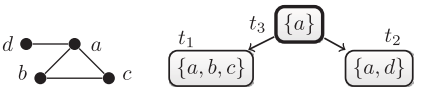


Fig. 1. Graph G (left) with a TD $\mathscr{T}$ of graph G (right).

we replace all expressions $v = d$ in β by 1 if $J(v) = d$, all expressions $v = v'$ by 1 if $J(v) = J(v')$, and by 0 otherwise. Second, we remove from the resulting clauses in $\beta(J)$ each clause c that contains a literal set to 1. Finally, we remove from every remaining clause in $\beta(J)$ every literal that is set to 0. We say a term assignment J is *satisfying* if $\beta(J) = \emptyset$.

2.2 Tree decomposition and treewidth

Treewidth is widely used for fine-grained complexity analyzes and to establish algorithms that provide tractability when bounding the treewidth. While it is only defined for graphs and hence widely used in graph theory (Bodlaender and Koster 2008), one can define graph representations of input instances for a variety of problems. Dedicated techniques then allow to solve problems from many domains such as propositional satisfiability (Samer and Szeider 2010), artificial intelligence (Gottlob and Szeider 2007), knowledge representation (Gottlob *et al.* 2006), argumentation (Fichte *et al.* 2019), nonmonotonic reasoning (Fichte *et al.* 2018), abduction in Datalog (Gottlob *et al.* 2007), and databases (Grohe 2007), probabilistic inference (Dechter 1999) (under the name bucket elimination) including constraint satisfaction, Fourier and Gaussian elimination for solving linear equalities and inequalities, and combinatorial optimization. While theoretical conditional lower bound results seem to discourage using algorithms that exploit bounded treewidth (Pan and Vardi 2006; Fichte *et al.* 2020), DP along TDs or related decompositions has recently been used to establish practical algorithms (Fichte *et al.* 2019; Dudek *et al.* 2020). An algorithm that exploits small treewidth takes a TD, which is an arrangement of a graph into a tree, and evaluates the problem in parts, via DP on the tree decomposition. Informally speaking, the tree decomposition provides an evaluation ordering, which one employs by a problem specific algorithm where the runtime of combinatorial hard part is bounded by a (potentially exponential) function of the treewidth w. Then, the underlying idea of treewidth is that it provides a measure for the closeness of a potential evaluation ordering which is simply a tree. Below, we provide the formal definitions of the notions tree decomposition and treewidth.

A *tree decomposition (TD)* (Kloks 1994; Cygan *et al.* 2015) of a given graph G is a pair $\mathscr{T} = (T, \chi)$ where T is a rooted tree and χ is a mapping which assigns to each node $t \in V(T)$ a set $\chi(t) \subseteq V(G)$, called *bag*, such that (i) $V(G) = \bigcup_{t \in V(T)} \chi(t)$ and $E(G) \subseteq \{\{u, v\} \mid t \in V(T), \{u, v\} \subseteq \chi(t)\}$; and (ii) for each $r, s, t \in V(T)$, such that s lies on the path from r to t, we have $\chi(r) \cap \chi(t) \subseteq \chi(s)$. We let $\mathrm{width}(\mathscr{T}) := \max_{t \in V(T)} |\chi(t)| - 1$. The *treewidth* $\mathrm{tw}(G)$ of G is the minimum $\mathrm{width}(\mathscr{T})$ over all TDs $\mathscr{T}$ of G. For $k \in \mathscr{N}$, we can compute a tree decomposition of width k or output that none exists in time $2^{\mathscr{O}k^3} \cdot |V|$ (Bodlaender and Koster 2008).

Example 1
Figure 1 depicts a graph G (left) and a TD $\mathscr{T}$ of G (right) of width 2. The treewidth of G is also 2 since G contains a complete graph with three vertices (Kloks 1994). ■

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

Next, we give the definition of a standard restriction for TDs, so called nice TDs. A nice TD ensures that one needs to consider only a small number of case distinctions in a DP algorithm later. For a node $t \in V(T)$, we say that type(t) is *leaf* if t has no children and $\chi(t) = \emptyset$; *join* if t has children t' and t'' with $t' \neq t''$ and $\chi(t) = \chi(t') = \chi(t'')$; *intr* ("introduce") if t has a single child t', $\chi(t') \subseteq \chi(t)$ and $|\chi(t)| = |\chi(t')| + 1$; *rem* ("removal") if t has a single child t', $\chi(t') \supseteq \chi(t)$ and $|\chi(t')| = |\chi(t)| + 1$. If for every node $t \in V(T)$, type(t) $\in \{$*leaf*, *join*, *intr*, *rem*$\}$, then the TD is called *nice*. The conditions allow us to focus on each of the cases of our algorithms individually.

3 Dynamic programming on tree decompositions

In the preliminaries, we gave definitions for TDs and treewidth and stated a variety of application areas. We mentioned that treewidth is widely used as a structural measure to establish tractability results when we consider in addition to the input size of the instance also the treewidth. If we want to exploit small treewidth of an input instance by an algorithm in practice, one can design a so-called DP algorithm, which works as follows: Subproblems are evaluated along the tree decomposition. At each node of the tree, information is gathered in tables. A table contains tuples of a fixed form that are designed to ensure certain properties. Then, a table algorithm maintains these tables during a post-order traversal. Thereby, it handles different cases according to the node contents of the TD and it ensures that properties required to solve the problem in parts sustain. The size of a table depends on the number of items in the bag, but is allowed to be exponential in the size of a bag. Hence, the overall technique works in linear time in the size of the problem and exponential in the bag size. Intuitively, the tree decomposition fixes an order in which we evaluate our problem. As a result, evaluating a problem along a tree decomposition allows for solving the problem at interest in parts, where the tree decomposition provides these parts and directs how solutions to the parts are supposed to be merged.

More formally, a solver based on *DP* evaluates the input $\mathscr{I}$ in parts along a given TD of a graph representation G of the input. Thereby, for each node t of the TD, intermediate results are stored in a *table* τ_t. This is achieved by running a so-called *table algorithm* A, which is designed for a certain graph representation and stores in τ_t results of problem parts of $\mathscr{I}$, thereby considering tables $\tau_{t'}$ for child nodes t' of t. DP works for many problems $\mathscr{P}$ as follows.

1. Construct a graph representation G of the given input instance $\mathscr{I}$.
2. Heuristically compute a tree decomposition $\mathscr{T} = (T, \chi)$ of G.
3. Traverse the nodes in $V(T)$ in post-order, that is, perform a bottom-up traversal of T. At every node t during post-order traversal, execute a table algorithm A that takes as input t, bag $\chi(t)$, a *local instance* $\mathscr{P}(t, \mathscr{I}) = \mathscr{I}_t$ depending on $\mathscr{P}$, as well as previously computed child tables of t and stores the result in τ_t.
4. Interpret table τ_n for the root n of T in order to output the solution of $\mathscr{I}$.

When specifying a DP algorithm for a specific problem such as #SAT, it is often sufficient to provide the data structures and the table algorithm for the specific problem as the general outline of the DP works the same for most problems. Hence, we focus on table algorithms and their description in the following.

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

Algorithm 1: Table algorithm $\mathsf{Sat}(t, \chi(t), \varphi_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for solving #Sat (Samer and Szeider 2010).

In: Node t, bag $\chi(t)$, clauses φ_t, and a sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child tables.

Out: Table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t := \{\langle \emptyset, 1 \rangle\}$

2 **else if** type$(t) =$ *intr, and* $a \in \chi(t)$ *is introduced* **then**

3 $\quad \tau_t := \{\langle I \cup \{a \mapsto 0\}, c \rangle \quad | \ \langle I, c \rangle \in \tau_1, \varphi_t(I \cup \{a \mapsto 0\}) = \emptyset\} \cup$

4 $\quad\quad \langle I \cup \{a \mapsto 1\}, c \rangle \quad | \ \langle I, c \rangle \in \tau_1, \varphi_t(I \cup \{a \mapsto 1\}) = \emptyset\}$

5 **else if** type$(t) =$ *rem, and* $a \notin \chi(t)$ *is removed* **then**

6 $\quad \tau_t := \{\langle I \setminus \{a \mapsto 0, a \mapsto 1\}, \sum_{c \in \mathrm{C}(I)} c \rangle\} \quad | \langle I, \cdot \rangle \in \tau_1\}$

`/* C(I) is the set that contains all counters for which assignments J are the same as I after we remove a from the assignment I */`

7 $\quad \mathrm{C}(I) := \{c \mid \langle J, c \rangle \in \tau_1, J \setminus \{a \mapsto 0, a \mapsto 1\} = I \setminus \{a \mapsto 0, a \mapsto 1\}\}$

8 **else if** type$(t) =$ *join* **then**

9 $\quad \tau_t := \{\langle I, c_1 \cdot c_2 \rangle \quad | \ \langle I, c_1 \rangle \in \tau_1, \langle I, c_2 \rangle \in \tau_2\}$

10 **return** τ_t

Next, we state the graph representation and table algorithm that we need to solve the problem $\mathscr{P} =$ #Sat (Samer and Szeider 2010). First, we need the following graph representation. The *primal graph* G_φ of a formula φ has as vertices its variables, where two variables are joined by an edge if they occur together in a clause of φ. Given formula φ, a TD $\mathscr{T} = (T, \chi)$ of G_φ and a node t of T. Sometimes, we refer to the treewidth of the primal graph of a given formula by the *treewidth of the formula*. Then, we let local instance #Sat$(t, \varphi) = \varphi_t$ be $\varphi_t := \{c \mid c \in \varphi, \mathrm{var}(c) \subseteq \chi(t)\}$, which are the clauses entirely covered by $\chi(t)$.

Table algorithm Sat as presented in Algorithm 1 shows all the cases that are needed to solve #Sat by means of DP over nice TDs. Each table τ_t consist of rows of the form $\langle I, c \rangle$, where I is an assignment of φ_t and c is a counter. Nodes t with type$(t) =$ *leaf* consist of the empty assignment and counter 1, cf., Line 1. For a node t with introduced variable $a \in \chi(t)$, we guess in Line 3 for each assignment β of the child table, whether a is set to true or to false, and ensure that φ_t is satisfied. When an atom a is removed in node t, we project assignments of child tables to $\chi(t)$, cf., Line 5, and sum up counters of the same assignments. For join nodes, counters of common assignments are multiplied, cf., Line 7. In Example 2 below, we explain the algorithm for a selected formula.

Example 2
Consider formula $\varphi := \{\overbrace{\{\neg a, b, c\}}^{c_1}, \overbrace{\{a, \neg b, \neg c\}}^{c_2}, \overbrace{\{a, d\}}^{c_3}, \overbrace{\{a, \neg d\}}^{c_4}\}$. Satisfying assignments of formula φ are, for example, $\{a \mapsto 1, b \mapsto 1, c \mapsto 0, d \mapsto 0\}$, $\{a \mapsto 1, b \mapsto 0, c \mapsto 1, d \mapsto 0\}$ or $\{a \mapsto 1, b \mapsto 1, c \mapsto 1, d \mapsto 1\}$. In total, there are six satisfying assignments of φ. Observe that graph G of Figure 1 depicts the primal graph G_φ of φ. Intuitively, $\mathscr{T}$ of Figure 1 allows to evaluate formula φ in parts. Figure 2 illustrates a nice TD $\mathscr{T}' = (\cdot, \chi)$ of the primal graph G_φ and tables τ_1, …, τ_{12} that are obtained during the execution

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

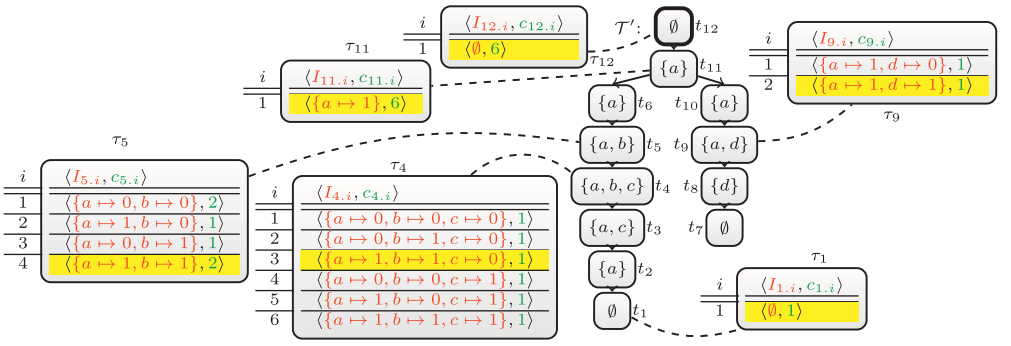


Fig. 2. Selected tables obtained by DP on $\mathscr{T}'$ for φ of Example 2 using algorithm Sat.

of Sat on nodes $t_1, \ldots, t_{12}$. We assume that each row in a table τ_t is identified by a number, that is, row i corresponds to $u_{t.i} = \langle I_{t.i}, c_{t.i}\rangle$.

Table $\tau_1 = \{\langle \emptyset, 1\rangle\}$ has type$(t_1) = \mathit{leaf}$. Since type$(t_2) = \mathit{intr}$, we construct table τ_2 from τ_1 by taking $I_{1.i} \cup \{a \mapsto 0\}$ and $I_{1.i} \cup \{a \mapsto 1\}$ for each $\langle I_{1.i}, c_{1.i}\rangle \in \tau_1$. Then, t_3 introduces c and t_4 introduces b. $\varphi_{t_1} = \varphi_{t_2} = \varphi_{t_3} = \emptyset$, but since $\chi(t_4) \subseteq \mathrm{var}(c_1)$ we have $\varphi_{t_4} = \{c_1, c_2\}$ for t_4. In consequence, for each $I_{4.i}$ of table τ_4, we have $\{c_1, c_2\}(I_{4.i}) = \emptyset$ since Sat enforces satisfiability of φ_t in node t. Since type$(t_5) = \mathit{rem}$, we remove variable c from all elements in τ_4 and sum up counters accordingly to construct τ_5. Note that we have already seen all rules where c occurs and hence c can no longer affect interpretations during the remaining traversal. We similarly create $\tau_6 = \{\langle \{a \mapsto 0\}, 3\rangle, \langle \{a \mapsto 1\}, 3\rangle\}$ and $\tau_{10} = \{\langle \{a \mapsto 1\}, 2\rangle\}$. Since type$(t_{11}) = \mathit{join}$, we build table τ_{11} by taking the intersection of τ_6 and τ_{10}. Intuitively, this combines assignments agreeing on a, where counters are multiplied accordingly. By definition (primal graph and TDs), for every $c \in \varphi$, variables var(c) occur together in at least one common bag. Hence, since $\tau_{12} = \{\langle \emptyset, 6\rangle\}$, we can reconstruct for example model $\{a \mapsto 1, b \mapsto 1, c \mapsto 0, d \mapsto 1\} = I_{11.1} \cup I_{5.4} \cup I_{9.2}$ of φ using highlighted (yellow) rows in Figure 2. On the other hand, if φ was unsatisfiable, τ_{12} would contain no values, that is, $\tau_{12} = \emptyset$. ■

4 Dynamic programming on tree decompositions expressed in relational algebra

While algorithms that run DP on bounded treewidth can be quite useful for efficient problem solving in practice, implementations turn out to be tedious already for problems such as the propositional satisfiability problem. In the following of the paper, we aim for rapid prototyping with DP by a declarative approach that ideally uses existing systems, gets parallel execution for free, and remains fairly efficient.

In the previous section, we explained that the traversal of the tree decomposition and the overall methodology of the procedure stays the same. But the core of DP on TDs for various problems is mostly the specification of the table algorithm that modifies a table based on previously computed tables. Hence, one can often focus on the table algorithms and their descriptions. When recalling basics from databases 101 from undergraduate studies (Elmasri and Navathe 2016) and taking a closer look on Algorithm 1 above,

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

we can immediately spot that we are effectively describing a query on existing data that produces a new table by Algorithm 1. This motivates our idea to use a database management system to execute the query and specify the query in SQL. Before we can proceed with our idea to use databases for executing DP algorithms, we take a step back and recall that the theory of SQL queries is based on relational algebra.

Relational algebra allows us to describe our algorithms and later use SQL encodings for specifying the table algorithm. The intermediate step of stating the algorithm in a relation algebra description is twofold. First, we can immediately see the connection between the algorithms given in the literature, which allows us to use the existing algorithms without reproving all properties. Second, we obtain a compact mathematical description, which is not just a lengthy and technical SQL query that might be hard to understand to researchers from the community who are usually not very familiar with practical databases and the usage of query languages.

4.1 Relational algebra

Before we start with details on our approach, we briefly recall basics in relational algebra. The classical relational algebra was introduced by Codd (1970) as a mathematical framework for manipulating relations (tables). Since then, relational algebra serves as the formal background and theoretical basis in relational databases and their standard language *SQL (Structured Query Language)* for querying tables (Ullman 1989). In fact, in the following, we need extended constructs, which have not been defined in the original framework by Codd, but are standard notions in databases nowadays (Elmasri and Navathe 2016). For the understanding later, we would like to mention that the SQL table model and relational algebra model slightly differ. The SQL table model is a bag (multiset) model, rather than a set (Garcia-Molina *et al.* 2009, Chapter 5). Below we also use extended projection and aggregation by grouping. Sometimes these are defined on bags. We avoid this in the definitions in order to keep the algorithms close to the formal set-based notation. Finally, we would like to emphasize that we are not using relation algebra here as developed by Alfred Tarski for the field of abstract algebra, but really relational algebra as used in database applications and theory.

A *column* a is of a certain finite *domain* $\mathrm{dom}(a)$. Then, a *row* r over set $\mathrm{cols}(r)$ of columns is a set of pairs of the form (a, v) with $a \in \mathrm{cols}(r), v \in \mathrm{dom}(a)$ such that for each $a \in \mathrm{cols}(r)$, there is exactly one $v \in \mathrm{dom}(a)$ with $(a, v) \in r$. In order to *access* the value v of an attribute a in a row r, we sometimes write $r.a$, which returns the unique value v with $(a, v) \in r$. A *table* τ is a finite set of rows r over set $\mathrm{cols}(\tau) := \mathrm{cols}(r)$ of columns, using domain $\mathrm{dom}(\tau) := \bigcup_{a \in \mathrm{cols}(\tau)} \mathrm{dom}(a)$. We define *renaming* of τ, given a set A of columns and a bijective mapping $m : \mathrm{cols}(\tau) \to A$ with $\mathrm{dom}(a) = \mathrm{dom}(m(a))$ for $a \in \mathrm{cols}(\tau)$, by $\rho_m(\tau) := \{(m(a), v) \mid (a, v) \in \tau\}$. In SQL, renaming can be achieved by means of the `AS` keyword.

Selection of rows in τ according to a given equality formula φ over term variables $\mathrm{cols}(\tau)$ is defined[1] by $\sigma_\varphi(\tau) := \{r \mid r \in \tau, \varphi(\mathrm{ass}(r)) = \emptyset\}$, where function ass provides the *corresponding term assignment* of a given row $r \in \tau$. Selection in SQL is specified using keyword `WHERE`. Given a relation τ' with $\mathrm{cols}(\tau') \cap \mathrm{cols}(\tau) = \emptyset$. Then, we refer to the

[1] We abbreviate for binary $v \in \mathrm{cols}(\tau)$ with $\mathrm{dom}(v) = \{0, 1\}$, $v = 1$ by v and $v = 0$ by $\neg v$.

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

cross-join by $\tau \times \tau' := \{r \cup r' \mid r \in \tau, r' \in \tau'\}$. Further, a *θ-join* (according to φ) corresponds to $\tau \bowtie_\varphi \tau' := \sigma_\varphi(\tau \times \tau')$. Interestingly, in SQL a θ-join can be achieved by specifying the two tables (cross-join) and adding the selection according to φ by means of `WHERE`.

Assume in the following a set $A \subseteq \text{cols}(\tau)$ of columns. Then, we let table τ *projected to* A be given by $\Pi_A(\tau) := \{r_A \mid r \in \tau\}$, where $r_A := \{(a, v) \mid (a, v) \in r, a \in A\}$. This concept of projection can be lifted to *extended projection* $\dot{\Pi}_{A,S}$, where we assume in addition to A, a set S of expressions of the form $a \leftarrow f$, such that $a \in \text{cols}(\tau) \setminus A$, f is an arithmetic function that takes a row $r \in \tau$, and there is at most one such expression for each $a \in \text{cols}(\tau) \setminus A$ in S. Formally, we define $\dot{\Pi}_{A,S}(\tau) := \{r_A \cup r^S \mid r \in \tau\}$ with $r^S := \{(a, f(r)) \mid a \in \text{cols}(r), (a \leftarrow f) \in S\}$. SQL allows to specify (extended) projection directly after initiating an SQL query with the keyword `SELECT`.

Later, we use *aggregation by grouping* ${}_AG_{(a \leftarrow g)}$, where we assume $a \in \text{cols}(\tau) \setminus A$ and a so-called *aggregate function* $g : 2^\tau \to \text{dom}(a)$, which intuitively takes a table of (grouped) rows. Therefore, we let ${}_AG_{(a \leftarrow g)}(\tau) := \{r \cup \{(a, g(\tau[r]))\} \mid r \in \Pi_A(\tau)\}$, where $\tau[r] := \{r' \mid r' \in \tau, r \subseteq r'\}$. For this purpose, we use for a set $S \subseteq \mathbb{S}$ of integers, the functions `SUM` for summing up values in S, `MIN` for providing the smallest integer in S, as well as `MAX` for obtaining the largest integer in S, which are often used for aggregation in this context. The SQL standard uses projection (`SELECT`) to specify A as well as the aggregate function g, such that these two parts are distinguished by means of the keyword `GROUP BY`.

Example 3
Assume a table $\tau_1 := \{r_1, r_2, r_3\}$ of 2 columns a, b over Boolean domain $\text{dom}(a) = \text{dom}(b) = \{0, 1\}$, where $r_1 := \{(a, 1), (b, 1)\}$, $r_2 := \{(a, 0), (b, 0)\}$, $r_3 := \{(a, 0), (b, 1)\}$.

τ_1	a	b
r_1	1	1
r_2	0	0
r_3	0	1

τ_2	b	a
r_1	1	1
r_2	0	0
r_3	1	0

Then, $r_3.a = 0$ and $r_3.b = 1$. Rows can be swapped by renaming and we let $\tau_2 := \rho_{\{a \mapsto b, b \mapsto a\}}\tau_1$. Observe that, for example, $\rho_{\{a \mapsto b, b \mapsto a\}}(\{r_3\})$ corresponds to $\{\{(a, 1), (b, 0)\}\}$, that is, considering r_3 and swapping a and b. We select rows by using the selection σ. For example, if we want to select rows where $b = 1$ (colored in blue) we can use $\sigma_{b=1}(\tau_1)$.

τ_1	a	b
r_1	1	1
r_2	0	0
r_3	0	1

Hence, applying $\sigma_{b=1}(\tau_1)$ results in $\{r_1, r_3\}$. Table τ_1 can be θ-joined with τ_2, but before, we need to have disjoint columns, which we obtain by renaming each column c to a fresh column c' as below by $\rho_{a \mapsto a', b \mapsto b'}\tau_2$. Then, $\tau_3 := \tau_1 \bowtie_{a=a' \wedge b=b'} (\rho_{a \mapsto a', b \mapsto b'}\tau_2)$.

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

τ_1	a	b
r_1	1	1
r_2	0	0
r_3	0	1

$\rho_{a\mapsto a',b\mapsto b'}(\tau_2)$	b'	a'
r_1	1	1
r_2	0	0
r_3	1	0

τ_3	a	b	b'	a'
r_1	1	1	1	1
r_2	0	0	0	0

Consequently, we have $\tau_3 = \{\{(a,0),(a',0),(b,0),(b',0)\},\{(a,1),(a',1),(b,1),(b',1)\}\}$. Extended projection allows not only to filter certain columns but also to add additional columns. As a result, if we only select column a of each row of τ_1, but add a fresh column c holding the sum of the values for a and b, then $\dot{\Pi}_{\{a\},\{c\leftarrow a+b\}}\tau_1$ corresponds to $\{\{(a,1),(c,2)\},\{(a,0),(c,0)\},\ \{(a,0),(c,1)\}\}$.

$\dot{\Pi}_{\{a\},\{c\leftarrow a+b\}}\tau_1$	a	c
r_1	1	2
r_2	0	0
r_3	0	1

Grouping τ_1 according to the value of column a, where we aggregate each group by summing up the values of columns b in a fresh column d, results in ${}_{\{a\}}G_{d\leftarrow\tau\mapsto \mathtt{SUM}(\{r.b|r\in\tau\})}(\tau_1)$, which simplifies to $\{\{(a,1),(d,1)\},\{(a,0),\{d,1)\}\}$ as illustrated below.

τ_1	a	b
r_1	1	1
r_2	0	0
r_3	0	1

${}_{\{a\}}G_{d\leftarrow\tau\mapsto \mathtt{SUM}(\{r.b|r\in\tau\})}(\tau_1)$	a	d
r_1	1	1
r_3	0	1

■

4.2 Table algorithms in relational algebra

Now, we are in the position to use relational algebra instead of set theory-based notions to describe how tables are obtained during DP. The step from set notation to relational algebra is driven by the observation that in these table algorithms one can identify recurring patterns and one mainly has to adjust problem-specific parts of it. We continue the description with our problem #Sat. We picture tables τ_t for each TD node t as relations, where τ_t distinguishes a unique column x for each $x \in \chi(t)$. In addition, we require a column *cnt* for counting in #Sat, or a column for modeling costs or weights in case of optimization problems.

Algorithm 2 presents a table algorithm for problem #Sat that is equivalent to Algorithm 1, but relies on relational algebra for computing tables. Since our description in relation algebra yields the same results as the set based-notation above, we omit formal correctness proofs. Nonetheless, we briefly explain below why both notations are identical. We highlight the crucial parts by coloring Algorithm 1. In particular, one typically derives for nodes t with type$(t) =$ *leaf*, a fresh initial table τ_t, cf., Line 1 of Algorithm 2. Then, whenever a variable a is introduced, such algorithms often use θ-joins with a fresh initial table for the introduced variable a. Hence, the new column represents

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

Algorithm 2: Table algorithm $\mathsf{Sat}'(t, \chi(t), \varphi_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for solving #SAT.

In: Node t, bag $\chi(t)$, local formula φ_t, sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child tables.

Out: Table τ_t.

1 **if** $\mathrm{type}(t) = \mathit{leaf}$ **then** $\tau_t := \{\{(\mathrm{cnt}, 1)\}\}$

2 **else if** $\mathrm{type}(t) = \mathit{intr}$, *and* $a \in \chi(t)$ *is introduced* **then**

3 $\quad \tau_t := \tau_1 \bowtie_{\varphi_t} \{\{(a,0)\}, \{(a,1)\}\}$

4 **else if** $\mathrm{type}(t) = \mathit{rem}$, *and* $a \notin \chi(t)$ *is removed* **then**

5 $\quad \tau_t := {}_{\chi(t)}G_{\mathrm{cnt} \leftarrow \tau \mapsto \mathtt{SUM}(\{r.\mathrm{cnt} \mid r \in \tau\})}(\Pi_{\mathrm{cols}(\tau_1) \setminus \{a\}} \tau_1)$

6 **else if** $\mathrm{type}(t) = \mathit{join}$ **then**

7 $\quad \tau_t := \dot{\Pi}_{\chi(t), \{\mathrm{cnt} \leftarrow \mathrm{cnt} \cdot \mathrm{cnt}'\}}(\tau_1 \bowtie_{\bigwedge_{u \in \chi(t)} u = u'} \rho_{\bigcup_{u \in \mathrm{cols}(\tau_2)} \{u \mapsto u'\}} \tau_2)$

8 **return** τ_t

the potential values for variable a. In Line 3, the selection of the θ-join is performed according to φ_t, that is, corresponding to the local instance of a formula. Further, for nodes t with $\mathrm{type}(t) = \mathit{rem}$, these table algorithms typically need projection. In case of Algorithm 2, Line 5 also needs grouping to sum up the counters for those rows of τ_1 that concur in τ_t. Thereby, rows are grouped according to values of columns $\chi(t)$ and we keep only one row per group in table τ, where the fresh counter *cnt* is the sum among all counters in τ. Finally, in Line 7 for a node t with $\mathrm{type}(t) = \mathit{join}$, we use extended projection and θ-joins, where we join on the same truth assignments. This allows us later to leverage database technology for a usually expensive operation. Extended projection is needed for multiplying the counters of the two rows containing the same assignment.

4.3 Table algorithms for selected problems

DP algorithms are known for a variety of problems. Standard texts in the area of parameterized algorithms and introductory lectures provide various specifications. For formal properties and detailed algorithm descriptions, we refer to other works (Bodlaender 1988; Cygan *et al.* 2015, Chapter 7; Dechter 1999; Bannach and Berndt 2019). Below, we present the table algorithms for a selection of combinatorial problems in relational algebra notation. In order to simplify the presentation, we assume that the instance is given by $\mathscr{I}$ and that the used tree decomposition is nice and given by $\mathscr{T} = (T, \chi)$. If the problem is a graph problem, $\mathscr{T}$ is a TD of $\mathscr{I}$, otherwise we implicitly assume that $\mathscr{T}$ is a TD of the primal graph of instance $\mathscr{I}$. For graph problems $\mathscr{I}$ and a given node t of T, we refer to the local instance of $\mathscr{I} = G = (V, E)$ by *local graph* G_t and define it by $G_t := (V \cap \chi(t), E \cap [\chi(t) \times \chi(t)])$.

*4.3.1 Problem #o-*COL

For a given graph instance $\mathscr{I} = G = (V, E)$, an *o-coloring* is a mapping $\iota : V \to \{1, \ldots, o\}$ such that for each edge $\{u, v\} \in E$, we have $\iota(u) \neq \iota(v)$. Then, the problem #o-COL asks to count the number of o-colorings of G, whose local instance #o-COL(t, G) is the local graph G_t. The table algorithm for this problem #o-COL is given in Algorithm 3. Similarly

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

Algorithm 3: Table algorithm $\mathsf{Col}(t, \chi(t), G_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for solving $\#o$-Col.

In: Node t, bag $\chi(t)$, local graph G_t, and a sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child tables.

Out: Table τ_t.

1 **if** $\mathrm{type}(t) = \mathit{leaf}$ **then** $\tau_t := \{(\mathrm{cnt}, 1)\}$

2 **else if** $\mathrm{type}(t) = \mathit{intr}$, *and* $a \in \chi(t)$ *is introduced* **then**

3 $\quad \tau_t := \tau_1 \bowtie_{\bigwedge_{\{u,v\} \in E(G_t)} u \neq v} \{\{(a,0)\}, \{(a,1)\}, \ldots, \{(a,o)\}\}$

4 **else if** $\mathrm{type}(t) = \mathit{rem}$, *and* $a \notin \chi(t)$ *is removed* **then**

5 $\quad \tau_t := {}_{\chi(t)}G_{\mathrm{cnt} \leftarrow \tau \mapsto \mathtt{SUM}(\{r.\mathrm{cnt} \mid r \in \tau\})}(\Pi_{\mathrm{cols}(\tau_1) \setminus \{a\}} \tau_1)$

6 **else if** $\mathrm{type}(t) = \mathit{join}$ **then**

7 $\quad \tau_t := \dot{\Pi}_{\chi(t), \{\mathrm{cnt} \leftarrow \mathrm{cnt} \cdot \mathrm{cnt}'\}}(\tau_1 \bowtie_{\bigwedge_{u \in \chi(t)} u = u'} \rho_{\bigcup_{u \in \mathrm{cols}(\tau_2)} \{u \mapsto u'\}} \tau_2)$

8 **return** τ_t

to Algorithm 2, for (empty) leaf nodes, the counter *cnt* is set to 1 in Line 1. Whenever a vertex a is introduced, in Line 3, one of the o many color values for a are guessed and θ-joined with the table τ_1 for the child node of t such that only colorings with different values for two *adjacent* vertices are kept. Similarly to Algorithm 2, whenever a vertex a is removed, Line 5 ensures that the column for a is removed and that counters *cnt* are summed up for rows that concur due to the removal of column a. Then, the case for join nodes in Line 7 is again analogous to Algorithm 2, where only rows with the same colorings in both child tables are kept and counters *cnt* are multiplied accordingly.

4.3.2 Problem MinVC

Given a graph instance $\mathscr{I} = G = (V, E)$, a *vertex cover* is a set of vertices $C \subseteq V$ of G such that for each edge $\{u, v\} \in E$, we have $\{u, v\} \cap C \neq \emptyset$. Then, MinVC asks to find the minimum cardinality $|C|$ among all vertex covers C, that is, C is such that there is no vertex cover C' with $|C'| < |C|$. Local instance $\mathrm{MinVC}(t, G) := G_t$, where the local graph G_t is defined above. We use an additional column *card* for storing cardinalities. The table algorithm for solving MinVC is provided in Algorithm 4, where, for leaf nodes the cardinality is 0, cf., Line 1. Then, when introducing vertex a, we guess in Line 3 whether a shall be in the vertex cover or not, and enforce that for each edge of the local instance at least one of the two endpoint vertices has to be in the vertex cover. Note that the additional cardinality column only takes removed vertices into account. More precisely, when a vertex a is removed, we group in Line 5 according to the bag columns $\chi(t)$, where the fresh cardinality value is the minimum cardinality (plus 1 for a if a shall be in the vertex cover), among those rows that concur due to the removal of a. The join node is similar to before, but in Line 7 we additionally need to sum up the cardinalities of two adjoining child table rows.

4.3.3 Problem MaxSat

Given Boolean formulas φ and ψ, an instance of problem MaxSat is of the form $\mathscr{I} = (\varphi, \psi)$ and we assume that $\mathscr{T}$ is a TD of primal graph $G_{\varphi \cup \psi}$. A solution to MaxSat

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

Algorithm 4: Table algorithm $\mathsf{VC}(t, \chi(t), G_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for solving MinVC.

In: Node t, bag $\chi(t)$, local graph G_t, and a sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child tables.

Out: Table τ_t.

1 **if** $\mathrm{type}(t) = \mathit{leaf}$ **then** $\tau_t := \{(\mathrm{card}, 0)\}$

2 **else if** $\mathrm{type}(t) = \mathit{intr}$, *and* $a \in \chi(t)$ *is introduced* **then**

3 $\quad \tau_t := \tau_1 \bowtie_{\bigwedge_{\{u,v\} \in E(G_t)} u \vee v} \{\{(a,0)\},\{(a,1)\}\}$

4 **else if** $\mathrm{type}(t) = \mathit{rem}$, *and* $a \notin \chi(t)$ *is removed* **then**

5 $\quad \tau_t := {}_{\chi(t)}G_{\mathrm{card} \leftarrow \tau \mapsto \mathtt{MIN}(\{r.\mathrm{card} + r.a \mid r \in \tau\})}(\Pi_{\mathrm{cols}(\tau_1) \setminus \{a\}} \tau_1)$

6 **else if** $\mathrm{type}(t) = \mathit{join}$ **then**

7 $\quad \tau_t := \dot{\Pi}_{\chi(t), \{\mathrm{card} \leftarrow \mathrm{card} + \mathrm{card}'\}}(\tau_1 \bowtie_{\bigwedge_{u \in \chi(t)} u = u'} \rho \bigcup_{u \in \mathrm{cols}(\tau_2)} \{u \mapsto u'\} \tau_2)$

8 **return** τ_t

Algorithm 5: Table algorithm $\mathsf{MSat}(t, \chi(t), \mathscr{I}_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for solving MaxSat.

In: Node t, bag $\chi(t)$, local instance $\mathscr{I}_t = (\varphi_t, \psi_t)$, and a sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child tables.

Out: Table τ_t.

1 **if** $\mathrm{type}(t) = \mathit{leaf}$ **then** $\tau_t := \{(\mathrm{card}, 0)\}$

2 **else if** $\mathrm{type}(t) = \mathit{intr}$, *and* $a \in \chi(t)$ *is introduced* **then**

3 $\quad \tau_t := \tau_1 \bowtie_{\varphi_t} \{\{(a,0)\},\{(a,1)\}\}$

4 **else if** $\mathrm{type}(t) = \mathit{rem}$, $a \notin \chi(t)$ *is removed, and* ψ' *are removed local soft-clauses* **then**

5 $\quad \tau_t := {}_{\chi(t)}G_{\mathrm{card} \leftarrow \tau \mapsto \mathtt{MAX}(\{r.\mathrm{card} + \Sigma_{c \in \psi',\, c(\mathrm{ass}(r)) = \emptyset} 1 \mid r \in \tau\})}(\Pi_{\mathrm{cols}(\tau_1) \setminus \{a\}} \tau_1)$

6 **else if** $\mathrm{type}(t) = \mathit{join}$ **then**

7 $\quad \tau_t := \dot{\Pi}_{\chi(t), \{\mathrm{card} \leftarrow \mathrm{card} + \mathrm{card}'\}}(\tau_1 \bowtie_{\bigwedge_{u \in \chi(t)} u = u'} \rho \bigcup_{u \in \mathrm{cols}(\tau_2)} \{u \mapsto u'\} \tau_2)$

8 **return** τ_t

is a satisfying assignment I of *hard-clauses* φ such that $|\{c \mid c \in \psi, c(I) = \emptyset\}|$ is maximized, that is, I is a satisfying assignment of φ that satisfies the maximum number of *soft-clauses* ψ among all satisfying assignments of φ. We define the local instance $\mathscr{I}_t := (\varphi_t, \psi_t)$ consisting of local formula φ_t, referred to by *local hard-clauses* and local formula ψ_t, called *local soft-clauses*.

The table algorithm for problem MaxSat is given in Algorithm 5, where we use column *card* for holding satisfied soft-clauses. Leaf tables only hold a cardinality value of 0 as in Line 1. Then, similar to the table algorithm Sat' (cf., Algorithm 2), when introducing a variable a, we guess the truth value and keep those rows, where local formula φ_t is satisfied. Whenever a variable a is removed in a node t, we remove column a and group rows that have common values over columns $\chi(t)$. Thereby, the new cardinality *card* for each group is the maximum among the values of *card* including the number of satisfied local soft-clauses ψ' of the child node of t that are removed in ψ_t (due to removal of a).

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

Finally, similar to Algorithm 4, a join node sums up cardinalities of two child rows containing the same assignment.

4.3.4 Problem MinIDS

Given a graph instance $\mathscr{I} = G = (V, E)$, a *dominating set* of G is a set of vertices $D \subseteq V$ of G, where each vertex $v \in V$ is either in D or is adjacent some vertex in D, that is, there is a vertex $d \in D$ with $\{d, v\} \in E$. A dominating set D is an *independent dominating set* of G, if there is no edge in E between vertices of D. Then, the problem MinIDS asks to find the minimum cardinality $|D|$ among all independent dominating sets D of G (if exists). We define local instance by $\text{MinIDS}(t, G) := G_t$.

The table algorithm for solving MinIDS is given in Algorithm 6, where a table τ_t of a node t uses column *card* for cardinalities of potential dominating sets, and an additional Boolean column d_u per bag vertex $u \in \chi(t)$. Intuitively, d_u indicates whether vertex u is already "dominated", that is, either u is in the dominating set or u has an adjacent vertex to the dominating set. Similar to before, leaf nodes set cardinality *card* to 0, cf., Line 1. For a node t with an introduced vertex a, we guess in Line 3 whether a shall be in the dominating set or not (and set d_a to 0 or 1, respectively). Then, we only keep rows that are independent, that is, a cannot be in the dominating set and adjacent to u in edges $E(G_t)$ of local graph G_t at the same time. Finally, values d_u (dominance status) for a and for neighbors of a are updated accordingly. When a vertex a is removed in a node t, Line 5 only keeps rows, where d_a is true, that is, a is indeed dominated, and removes columns a, d_a. Further, we group rows according to their values to $\chi(t) \cup \{d_u \mid u \in \chi(t)\}$ and for each group we set the cardinality to the minimum among the cardinalities of the group rows (including a if a is in the set). For join nodes t, Line 7 sums up cardinalities of rows holding the same dominating set and treats a vertex $u \in \chi(t)$ as dominated if it is dominated in at least one of the two rows.

Algorithm 6: Table algorithm $\mathsf{IDS}(t, \chi(t), G_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for solving MinIDS.

In: Node t, bag $\chi(t)$, local graph G_t, and a sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child tables.

Out: Table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t := \{(\text{card}, 0)\}$

2 **else if** type$(t) =$ *intr, and* $a \in \chi(t)$ *is introduced* **then**

3 $\quad \tau_t := \dot{\Pi}_{\chi(t), \bigcup_{u \in \chi(t)} \{d_u \leftarrow \bigvee_{\{u,v\} \in E(G_t)} d_u \vee v, \text{card} \leftarrow \text{card}\}} (\tau_1 \bowtie_{\bigwedge_{\{u,v\} \in E(G_t)} \neg u \vee \neg v} \{\{(a, 0), (d_a, 0)\}, \{(a, 1), (d_a, 1)\}\})$

4 **else if** type$(t) =$ *rem, and* $a \notin \chi(t)$ *is removed* **then**

5 $\quad \tau_t := {}_{\chi(t) \cup \{d_u \mid u \in \chi(t)\}} G_{\text{card} \leftarrow \tau \mapsto \text{MIN}(\{r.\text{card} + r.a \mid r \in \tau\})} (\Pi_{\text{cols}(\tau_1) \setminus \{a, d_a\}} \sigma_{d_a} \tau_1)$

6 **else if** type$(t) =$ *join* **then**

7 $\quad \tau_t := \dot{\Pi}_{\chi(t), \bigcup_{u \in \chi(t)} \{d_u \leftarrow d_u \vee d'_u, \text{card} \leftarrow \text{card} + \text{card}'\}} (\tau_1 \bowtie_{\bigwedge_{u \in \chi(t)} u = u'} \rho_{\bigcup_{u \in \text{cols}(\tau_2)} \{u \mapsto u'\}} \tau_2)$

8 **return** τ_t

Similar to MinVC and #*o*-Col one can model several other (graph) problems. One could also think of counting the number of solutions to problem MinVC, where both a

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

column for cardinalities and one for counting is used. There, in addition to grouping, we additionally could use conditions over groups where only rows are kept whose column values for *card* form the minimum within the group.

4.4 Generalizing the patterns of table algorithms

In the previous sections, we presented the table algorithms for solving a selection of combinatorial problems, namely, #Sat, #*o*-Col, MinVC, MaxSat, and MinIDS, by DP. As mentioned in Section 2.2, there are a variety of application areas where such algorithms allow for solving problems efficiently. When specifying most algorithms, we focus on the table algorithm A, which is executed for each node t of T of the considered tree decomposition $\mathscr{T} = (T, \chi)$ and computes a new table depending on the previously computed tables at the children of t. From the descriptions above, it is easy to see that the algorithms effectively follow standard patterns. Therefore, we present a general template in Algorithm 7, where parts of table algorithms for problems that are typically problem specific are replaced by colored placeholders of the form #placeHolder#, cf., Algorithm 2. The general template of table algorithms works for many problems, including decision problems, counting problems as well as optimization problems.

Algorithm 7: Template table algorithm $\mathsf{A}(t, \chi(t), \mathscr{I}_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for solving problem $\mathscr{P}$.

In: Node t, bag $\chi(t)$, local instance $\mathscr{I}_t$, and a sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child tables.

Out: Table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t :=$ #leafTab#

2 **else if** type$(t) =$ *intr, and* $a \in \chi(t)$ *is introduced* **then**

3 $\quad \tau_t := \dot{\Pi}_{\chi(t),\#\mathsf{intrAddCols}\#}(\tau_1 \bowtie_{\#\mathsf{intrFilter}\#} \#\mathsf{intrTab}\#)$

4 **else if** type$(t) =$ *rem, and* $a \notin \chi(t)$ *is removed* **then**

5 $\quad \tau_t := {}_{\chi(t)\cup\#\mathsf{remGroupCols}\#}G_{\#\mathsf{remAggr}\#}(\Pi_{\mathrm{cols}(\tau_1)\setminus\{a,\#\mathsf{remCols}\#\}}\sigma_{\#\mathsf{remFilter}\#}\tau_1)$

6 **else if** type$(t) =$ *join* **then**

7 $\quad \tau_t := \dot{\Pi}_{\chi(t),\#\mathsf{joinAddCols}\#}(\tau_1 \bowtie_{\bigwedge_{u\in\chi(t)} u=u' \wedge \#\mathsf{joinAddFilter}\#} \rho_{\bigcup_{u\in\mathrm{cols}(\tau_2)}\{u\mapsto u'\}}\tau_2)$

8 **return** τ_t

The intuition behind these placeholders is as follows: For leaf nodes, the initial table (typically empty) can be specified using #leafTab#. For introduce nodes, the potential cases for the introduced vertex a are given with the help of #intrTab#. Then, according to the local instance, we only keep those rows that satisfy #intrFilter#. The placeholder #intrAddCols# allows to add additional columns, which we often need when solving problems that involve counting or optimizing a value. In other words, placeholder #intrAddCols# in Line 3 of Algorithm 7 uses extended projection, which is needed for problems requiring changes on vertex introduction. Nodes, where an atom a is removed sometimes require to filter rows, which do not lead to a solution using #remFilter#, and to remove columns concerning a by #remCols#. Further, one oftentimes needs to aggregate rows according to the values of the columns of the bag and additional columns (given by #remGroupCols#), where the aggregation is specified by #remAggr#. Finally,

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

for join nodes, one can specify an additional filter #joinAddFilter# that goes beyond checking equivalence of row values in the θ-join operation. Further, depending on the problem one might need to add and update the values of additional columns by using extended projection in the form of placeholder #joinAddCols#.

Note that while the algorithms presented here assume for simplicity nice TDs, the whole architecture does not depend on certain restrictions of TDs, or whether it is nice or not. Instead, a table algorithm of any TD is simply specified by handling *problem-specific* implementations of the placeholders of Algorithm 7, where the system following this architecture is responsible for interleaving and overlapping these cases within a node t. In fact, we discuss an implementation of a system according to this architecture next, where it is crucial to implement non-nice TDs to obtain higher efficiency.

5 System dpdb: Dynamic programming with databases and SQL

In this section, we present a general architecture to model table algorithms by means of DBMs. We move from relational algebra definitions to specifications of the table algorithms in terms of SQL queries. The overall architecture follows the DP approach as presented in Section 3. It works as depicted in Figure 3 where the steps highlighted in yellow and blue need to be specified depending on the problem $\mathscr{P}$. Steps outside Step 3 are mainly setup tasks, the yellow "E"s indicate *events* that might be needed to solve more complex problems on the polynomial hierarchy. For example, one could create and drop auxiliary subtables for each node during Step 3 within such events. Observe that after the generation of a TD $\mathscr{T} = (T, \chi)$, Step 2b automatically creates tables τ_t for each node t of T, where the corresponding table columns of τ_t are specified in the blue part, that is, within A. The *default columns* of such a table τ_t that are assumed in this section foresee one column for each element of the bag $\chi(t)$, where additional columns that are needed for solving the problem can be added. This includes additional auxiliary columns, which can be also counters or costs for counting or optimization, respectively. Besides the definition of table schemes, the blue part concerns the specification of the table algorithm by means of a procedural *generator template* that describes how to obtain SQL code for each node t, thereby depending on $\chi(t)$ and on the tables for child nodes of t. This generated SQL code is then used internally for manipulation of tables τ_t during the tree decomposition traversal in Step 3 of DP.

We implemented the proposed architecture of the previous section in the prototypical system dpdb. The system is open-source,[2] written in Python 3, and uses PostgreSQL as DBMS. We are certain that one can easily replace PostgreSQL by any other state-of-the-art relational database that uses SQL. In the following, we discuss implementation specifics that are crucial for a performant system that is still extendable and flexible.

Computing TDs. TDs are computed mainly with the library *htd* version 1.2 with default settings (Abseher *et al.* 2017), which finds TDs extremely quick also for interesting instances (Fichte *et al.* 2019) due to heuristics. Note that dpdb directly supports the TD format of recent competitions (Dell *et al.* 2018), that is, one could easily replace the TD library. It is important not to enforce htd to compute nice TDs, as this would cause a

[2] Our system dpdb is available under GPL3 license at github.com/hmarkus/dp_on_dbs.

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

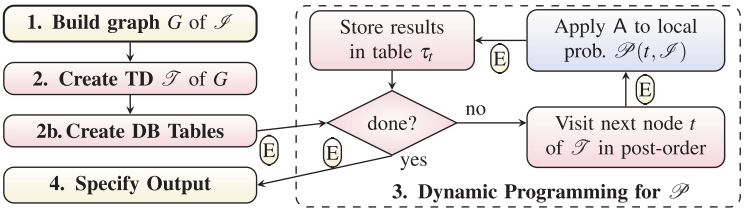


Fig. 3. Architecture of Dynamic Programming with Databases. Steps highlighted in red are provided by the system depending on the specification of yellow and blue parts, which is given by the user for specific problems $\mathscr{P}$. The yellow "E"s represent events that can be intercepted and handled by the user. The blue part concentrates on table algorithm A, where the user specifies how SQL code is generated in a modular way.

lot of overhead later in dpdb for copying tables. However, in order to benefit from the implementation of θ-joins, query optimization, and state-of-the-art database technology in general, we observed that it is crucial to limit the number of child nodes of every TD node. In result, when huge tables are involved, θ-joins among child node tables cover at most a limited number of child node tables. Hence, the query optimizer of the database system can still come up with meaningful execution plans depending on the contents of the table. Nonetheless, we prefer θ-joins with more than just two tables, since binary θ-joins already fix in which order these tables shall be combined, which already limits the query optimizer. Apart from this trade-off, we tried to outsource the task of joining tables to the DBMS, since the performance of database systems highly depends on query optimization. The actual limit, which is a restriction from experience and practice only, highly depends on the DBMS that is used. For PostgreSQL, we set a limit of at most 5 child nodes for each node of the TD, that is, each θ-join covers at most 5 child tables.

Toward non-nice TDs. Although this paper presents the algorithms for nice TDs (mainly due to simplicity), the system dpdb interleaves these cases as presented in Algorithm 7. More precisely, the system executes one query per table τ_t for each node t during the traversal of TD $\mathscr{T}$. This query consists of several parts and we briefly explain its parts from outside to inside in accordance with Algorithm 7. First of all, the innermost part concerns the *row candidates* for τ_t consisting of the θ-join among all child tables of τ_t as in Line 7 of Algorithm 7. If there is no child node of t, table #leafTab# of Line 1 is used instead. Next, the result is cross-joined with #intrTab# for each introduced variable as in Line 3, but without using the filter #intrFilter# yet. Then, the result is projected by using extended projection involving $\chi(t)$ as well as both #joinAddCols# and #intrAddCols#. Actually, there are different configurations of how dpdb can deal with the resulting row candidates. For debugging (see below) one could (1) actually materialize the result in a table, whereas for performance reasons, one should use (2) *common table expressions (CTEs or `WITH`-queries)* or (3) *sub-queries (nested queries)*, which both result in one nested SQL query per table τ_t. On top of these row candidates, selection according to #intrFilter#, cf., Line 3, is executed. Finally, the resulting table is plugged as table τ_1 into Line 5, where in particular the result is grouped by using both $\chi(t)$[3] and #remGroupCols# and each group is aggregated by #remAggr# accordingly. It turns

[3] Actually, dpdb keeps only columns relevant for the table of the parent node of t.

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

out that PostgreSQL can do better with subqueries than CTEs, since we observed that the query optimizer oftentimes pushes (parts of) outer selections and projections into the subquery if needed, which is not the case for CTEs, as discussed in the PostgreSQL manual (PostgreSQL Global Development Group 2020, Section 7.8.1). On different DBMSs or other vendors, for example, Oracle, it might be better to use CTEs instead.

Example 4
Consider again Example 2 and Figure 1 and let us use table algorithm Sat′ with dpdb on formula φ of TD $\mathscr{T}$ and Option (3): subqueries, where the row candidates are expressed via a subqueries. Then, for each node t_i of $\mathscr{T}$, dpdb generates a view vi as well as a table τ_i containing in the end the content of vi. Observe that each view only has one column a for each variable a of φ since the truth assignments of the other variables are not needed later. This keeps the tables compact, only τ_1 has two rows, τ_2, and τ_3 have only one row. We obtain the following views.

```
CREATE VIEW v1 AS SELECT a, sum(cnt) AS cnt FROM
 (WITH intrTab AS (SELECT 0 AS val UNION ALL SELECT 1)
   SELECT i1.val AS a, i2.val AS b, i3.val AS c, 1 AS cnt
          FROM intrTab i1, intrTab i2, intrTab i3)
WHERE (NOT a OR b OR c) AND (a OR NOT b OR NOT c) GROUP BY a

CREATE VIEW v2 AS SELECT a, sum(cnt) AS cnt FROM
 (WITH intrTab AS (SELECT 0 AS val UNION ALL SELECT 1)
   SELECT i1.val AS a, i2.val AS d, 1 AS cnt FROM intrTab i1, intrTab i2)
WHERE (a OR d) AND (a OR NOT d) GROUP BY a

CREATE VIEW v3 AS SELECT a, sum(cnt) AS cnt FROM
 (SELECT τ1.a, τ1.cnt * τ2.cnt AS cnt FROM τ1, τ2 WHERE τ1.a = τ2.a)
GROUP BY a
```

■

Parallelization. A further reason to not overrestrict the number of child nodes within the TD, lies in parallelization. In dpdb, we compute tables in parallel along the TD, where multiple tables can be computed at the same time, as long as the child tables are computed. Therefore, we tried to keep the number of child nodes in the TD as high as possible. In our system dpdb, we currently allow for at most 24 worker threads for table computations and 24 database connections at the same time (both pooled and configurable). On top of that we have two additional threads and database connections for job assignments to workers, as well as one dedicated watcher thread for clean-up and connection termination, respectively.

Logging, Debugging and Extensions. Currently, we have two versions of the dpdb system implemented. One version aims for performance and the other one tries to achieve comprehensive logging and easy debugging of problem (instances), thereby increasing explainability. The former does neither keep intermediate results nor create database tables in advance (Step 2b), as depicted in Figure 3, but creates tables according to an SQL SELECT statement. In the latter, we keep all intermediate results, we record database timestamps before and after certain nodes, provide statistics as, for example, width and

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

number of rows. Further, since for each table τ_t, exactly one SQL statement is executed for filling this table, we also have a dedicated view of the SQL `SELECT` statement, whose result is then inserted in τ_t. Together with the power and flexibility of SQL queries, we observed that this helps in finding errors in the table algorithm specifications.

Besides convient debugging, system dpdb immediately contains an extension for *approximation*. There, we restrict the table contents to a maximum number of rows. This allows for certain approximations on counting problems or optimization problems, where it is infeasible to compute the full tables. Further, dpdb foresees a dedicated *randomization* on these restricted number of rows such that we obtain different approximate results on different random seeds.

Note that dpdb can be easily extended. Each problem can overwrite existing default behavior and dpdb also supports problem-specific argument parsers for each problem individually. Out-of-the-box, we support the formats DIMACS SAT and DIMACS graph (Liu *et al.* 2006) as well as the common format for TDs (Dell *et al.* 2018).

5.1 *Implementing table algorithms with* dpdb *for selected problems*

The system dpdb allows for *easy prototyping* of DP algorithms on TDs. In the following, we present the relevant parts of table algorithm implementations according to the template in Algorithm 7 for our selection of problems below.[4] More precisely, we give the SQL implementations of the table algorithms of the previous section in the form of specifying the corresponding placeholders as given by the template algorithm A. Thereby, we only specify the placeholders needed for solving the problems, that is, placeholders of template algorithm A that are not used (empty) are left out. To this end, we assume in this section for each problem a not necessarily nice TD $\mathscr{T} = (T, \chi)$ of the corresponding graph representation of our given instance $\mathscr{I}$, as well as any node t of T and its child nodes $t_1, \ldots, t_\ell$.

Problem #Sat. Given instance formula $\mathscr{I} = \varphi$. Then, the specific placeholders of the template for #Sat for a node t with $\varphi_t = \{\{l_{1,1}, \ldots, l_{1,k_1}\}, \ldots, \{l_{n,1}, \ldots, l_{n,k_n}\}\}$ that are required for dpdb to solve the problem are as follows.

- #leafTab#: `SELECT 1 AS cnt`
- #intrTab#: `SELECT 0 AS val UNION ALL SELECT 1`
- #intrFilter#: $(l_{1,1}$ `OR ...` $l_{1,k_1})$ `AND ...` $(l_{n,1}$ `OR ...` $l_{n,k_n})$
- #remAggr#: `SUM(cnt) AS cnt`
- #joinAddCols#: τ_1`.cnt * ...` τ_ℓ`.cnt AS cnt`

If one compares this specification to Algorithm 2, one sees that conceptually the same idea is given above. However, for efficiency dpdb does not rely on nice TDs. Observe that for the plain decision problem Sat, where the goal is to decide only the existence of a satisfying assignment for given formula φ, placeholder #leafTab# would need to return the empty table and parts #remAggr# and #joinAddCols# are just empty since no counter *cnt* is needed.

[4] Prototypical implementations for problems #Sat as well as MinVC are readily available in dpdb.

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

Problem #*o*-Col. Recall the problem #*o*-Col and Algorithm 3. Let $\mathscr{I} = G = (V, E)$ be a given input graph. Then, specific implementation parts for #*o*-Col for a node t with $E(G_t) = \{\{u_1, v_1\}, \ldots, \{u_n, v_n\}\}$ is given as follows.

- #leafTab#: `SELECT 1 AS cnt`
- #intrTab#: `SELECT 0 AS val UNION ALL ... UNION ALL SELECT` o
- #intrFilter#: `NOT` $(u_1 = v_1)$ `AND ... AND NOT` $(u_n = v_n)$
- #remAggr#: `SUM(cnt) AS cnt`
- #joinAddCols#: τ_1`.cnt * ... *` τ_ℓ`.cnt AS cnt`

Problem MinVC. Given any input graph $\mathscr{I} = G = (V, E)$ of MinVC. Then, problem MinVC for a node t with $E(G_t) = \{\{u_1, v_1\}, \ldots, \{u_n, v_n\}\}$ and removed vertices $\chi(t) \setminus (\chi(t_1) \cup \ldots \cup \chi(t_\ell)) = \{r_1, \ldots, r_m\}$ is specified by the following placeholders (cf., Algorithm 4).

- #leafTab#: `SELECT 0 AS card`
- #intrTab#: `SELECT 0 AS val UNION ALL SELECT 1`
- #intrFilter#: $(u_1$ `OR` $v_1)$ `AND ...` `AND` $(u_n$ `OR` $v_n)$
- #remAggr#: `MIN(card +` r_1 `+ ... +` r_m`) AS card`
- #joinAddCols#: τ_1`.card + ... +` τ_ℓ`.card AS card`

Problem MaxSat. Given an instance $\mathscr{I} = (\varphi, \psi)$ of problem MaxSat. Then, the problem for a node t with local hard clauses $\varphi_t = \{\{l_{1,1}, \ldots, l_{1,k_1}\}, \ldots, \{l_{n,1}, \ldots, l_{n,k_n}\}\}$ and local soft clauses $\psi_t = \{\{l'_{1,1}, \ldots, l'_{1,k'_1}\}, \ldots, \{l'_{p,1}, \ldots, l_{p,k'_p}\}\}$ is specified by the following placeholders (cf., Algorithm 5).

- #leafTab#: `SELECT 0 AS card`
- #intrTab#: `SELECT 0 AS val UNION ALL SELECT 1`
- #intrFilter#: $(l_{1,1}$ `OR ...` `OR` $l_{1,k_1})$ `AND ...` `AND` $(l_{n,1}$ `OR ...` `OR` $l_{n,k_n})$
- #remAggr#: `MIN(card +` $(l'_{1,1}$ `OR ...` `OR` $l'_{1,k'_1})$ `+ ... +` $(l'_{p,1}$ `OR ...` `OR` $l'_{p,k'_p}))$ `AS card`
- #joinAddCols#: τ_1`.card + ... +` τ_ℓ`.card AS card`

Problem MinIDS. Recall an instance $\mathscr{I} = G = (V, E)$ of problem MinIDS and table algorithm IDS as given in Algorithm 6. The implementation of table algorithm IDS for MinIDS for a node t assumes that $E(G_t) = \{\{u_1, v_1\}, \ldots, \{u_n, v_n\}\}$. Further, we let bag $\chi(t) = \{a_1, \ldots, a_k\}$, removed vertices $\chi(t) \setminus (\chi(t_1) \cup \ldots \cup \chi(t_\ell)) = \{r_1, \ldots, r_m\}$, and we let the w_i many neighbors N_i of each vertex a_i in G (with $1 \leq i \leq k$) be given by $N_i = \{e_{a_i,1}, \ldots, e_{a_i,w_i}\}$. Then, the SQL implementation can be specified as follows.

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

- #leafTab#: `SELECT 0 AS card`
- #intrTab#: `SELECT 0 AS val, 0 AS d UNION ALL SELECT 1, 1`
- #intrFilter#: (NOT u_1 OR NOT v_1) AND ... AND (NOT u_n OR NOT v_n)
- #intrAddCols#: card, d_{a_1} OR $e_{a_1,1}$ OR ... OR e_{a_1,w_1} AS d_{a_1}, ... d_{a_k} OR $e_{a_k,1}$ OR ... OR e_{a_k,w_k} AS d_{a_k}
- #remFilter#: d_{a_1} AND ... AND d_{a_k}
- #remAggr#: MIN(card + r_1 + ... + r_m) AS card
- #remGroupCols#: d_{a_1}, ..., d_{a_k}
- #joinAddCols#: τ_1.card + ... + τ_ℓ.card AS card, $\tau_1.d_{a_1}$ OR ... OR $\tau_\ell.d_{a_1}$ AS d_{a_1}, ... $\tau_1.d_{a_k}$ OR ... OR $\tau_\ell.d_{a_k}$ AS d_{a_k}

Note that implementations could generate and apply parts of this specification, as for example in #intrFilter# only edges that have not been checked so far in any descending node, need to be checked.

Similar to MinVC, #o-Col, and MinIDS one can model several related (graph) problems. One could also think of counting the number of solutions to problem MaxSat, where both, a column for cardinalities and one for counting is used. There, in addition to grouping with `GROUP BY` in `dpdb`, we additionally use the `HAVING` construct of SQL, where only rows are kept, whose column `card` is minimal.

6 Experiments

We conducted a series of experiments using publicly available benchmark sets for #Sat. Our tested benchmarks (Fichte *et al.* 2018a) are publicly available and our results are also on github at github.com/hmarkus/dp_on_dbs/tplp.

6.1 Setup

Measure & Resources. We mainly compare wall clock time and number of timeouts. In the time we include *preprocessing time* as well as *decomposition time* for computing a TD with a fixed random seed. For parallel solvers we allowed access to 24 physical cores on machines. We set a timeout of 900 s and limited available RAM to 14 GB per instance and solver. However, since our solver `dpdb` is a solver using multiple threads, we restricted the results of `dpdb` to a total of 900 s of CPU time. While allowing for all the other (parallel) solvers more than 900 s of CPU time. For `dpdb`, we only allowed 900 s of CPU time, but at the same time restricted to 900 s wall clock time.

Benchmark Instances. We considered a selection of overall 1494 instances from various publicly available benchmark sets #Sat consisting of `fre/meel` benchmarks[5](1480

[5] See: tinyurl.com/countingbenchmarks.

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

instances), and `c2d` benchmarks[6] (14 instances). We preprocessed the instances by the #SAT preprocessor *pmc* (Lagniez and Marquis 2014), similar to results of recent work on #SAT (Fichte *et al.* 2019), where it was also shown that more than 80% of the #SAT instances have primal treewidth below 19 after preprocessing. For preprocessing with *pmc* we used the recommended options `-vivification -eliminateLit -litImplied -iterate=10 -equiv -orGate -affine`, which ensures that model counts are preserved.

Benchmarked system dpdb. We used PostgreSQL 12 for our system `dpdb` on a tmpfs-ramdisk (/tmp) that could grow up to at most 1 GB per run. To ensure comparability with previous results (Fichte *et al.* 2020), where we had employed PostgreSQL 9.5 for our system `dpdb`, we also considered the configuration `dpdb pg9` that uses the preinstalled database system PostgreSQL 9.5 without any ramdisk at all (plain hard disk). However, we observed major performance increases of `dpdb` compared to `dpdb pg9`. We allow parallel execution for the database management system PostgreSQL with access to all cores of the benchmarking system. However, we restrict the total CPU time to ensure that we do not bias system resources toward dpdb.

Other benchmarked systems. In our experimental work, we present results for the most recent versions of publicly available #SAT solvers, namely, *c2d* 2.20 (Darwiche 2004), *d4* 1.0 (Lagniez and Marquis 2017), *DSHARP* 1.0 (Muise *et al.* 2012), *miniC2D* 1.0.0 (Oztok and Darwiche 2015), *cnf2eadt* 1.0 (Koriche *et al.* 2013), *bdd_minisat* 1.0.2 (Toda and Soh 2015), and *sdd* 2.0 (Darwiche 2011), which are all based on knowledge compilation techniques. We also considered rather recent approximate solvers *ApproxMC2*, *ApproxMC3* (Chakraborty *et al.* 2014), *ganak* (Sharma *et al.* 2019), and *sts* 1.0 (Ermon *et al.* 2012), as well as CDCL-based solvers *Cachet* 1.21 (Sang *et al.* 2004), *sharpCDCL*,[7] and *sharpSAT* 13.02 (Thurley 2006). Finally, we also included multi-core solvers *gpusat* 1.0 and *gpusat* 2.0 (Fichte *et al.* 2018b; 2019), which both are based on DA, as well as *countAntom* 1.0 (Burchard *et al.* 2015) on 12 physical CPU cores, which performed better than on 24 cores. Experiments were conducted with default solver options. Note that we excluded distributed solvers such as dCountAntom (Burchard *et al.* 2016) and DMC (Lagniez *et al.* 2018) from our experimental setup. Both solvers require a cluster with access to the Open-MPI framework (Gabriel *et al.* 2004) and fast physical interconnections. Unfortunately, we do not have access to OpenMPI on our cluster. Nonetheless, our focus are shared-memory systems and since dpdb might well be used in a distributed setting, it leaves an experimental comparison between distributed solvers that also include dpdb as subsolver to future work.

Benchmark Hardware. Almost all solvers were executed on a cluster of 12 nodes. Each node is equipped with two Intel Xeon E5-2650 CPUs consisting of 12 physical cores each at 2.2 GHz clock speed, 256 GB RAM and 1 TB hard disc drives (*not* an SSD)

[6] See: reasoning.cs.ucla.edu/c2d.
[7] See: tools.computational-logic.org.

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

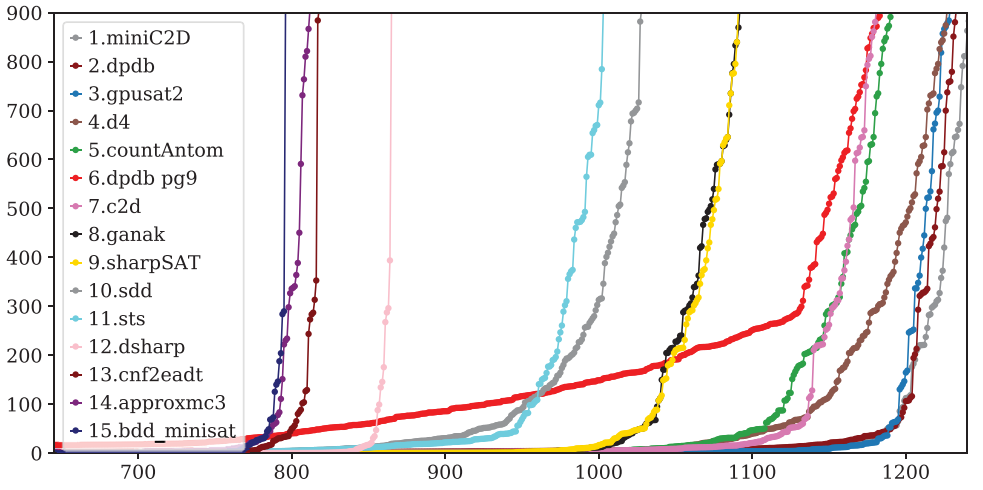


Fig. 4. Runtime for the top 15 solvers over all #SAT instances. The x-axis refers to the number of instances, and the y-axis depicts the runtime sorted in ascending order for each solver individually.

Seagate ST1000NM0033. The results were gathered on Ubuntu 16.04.1 LTS machines with disabled hyperthreading on kernel 4.4.0-139. As we also took into account solvers using a GPU, for gpusat1 and gpusat2 we used a machine equipped with a consumer GPU: Intel Core i3-3245 CPU operating at 3.4 GHz, 16 GB RAM, and one Sapphire Pulse ITX Radeon RX 570 GPU running at 1.24 GHz with 32 compute units, 2048 shader units, and 4GB VRAM using driver amdgpu-pro-18.30-641594 and OpenCL 1.2. The system operated on Ubuntu 18.04.1 LTS with kernel 4.15.0-34.

6.2 Results

Figure 4 illustrates the top 15 solvers, where instances are preprocessed by pmc, in a cactus-like plot, which provides an overview over all the benchmarked #SAT instances. The x-axis of these plots refers to the number of instances, and the y-axis depicts the runtime sorted in ascending order for each solver individually. Overall, `dpdb` seems to be quite competitive and beats most of the solvers, as for example d4, countAntom, c2d, ganak, sharpSAT, dsharp, and approxmc. Interestingly, `dpdb` solves also instances, whose treewidth upper bounds are larger than 41. Surprisingly, `dpdb pg9` shows a different runtime behavior than the other solvers. We believe that the reason lies in an initial overhead caused by the creation of the tables that seems to depend on the number of nodes of the used TD. There, *I/O operations* of writing from main memory to hard disk seem to kick in. This disadvantage is resolved if benchmarking `dpdb` on recent versions of PostgreSQL (version 12) and using tmpfs-ramdisks instead of plain hard disks. Table 1 presents more detailed runtime results, showing a solid second place for `dpdb` as our system solves the vast majority of the instances. Notably, it seems that the ganak solver is among the fastest on a lot of instances. We observed that ganak has the fastest runtime on 633 instances, when considering results of all 15 presented solvers. Assume we only

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

Table 1. *Number of solved #SAT instances, preprocessed by pmc and grouped by intervals of upper bounds of the treewidth. time[h] is the cumulated wall clock time in hours, where unsolved instances are counted as 900 s.*

	solver	0–20	21–30	31–40	41–50	51–60	>60	best	unique	$\sum$	time[h]
preprocessed by pmc	1. miniC2D	1193	29	**10**	2	1	7	11	0	**1242**	**68.77**
	2. dpdb	1193	31	7	2	0	0	2	1	1233	70.44
	3. gpusat2	**1196**	**32**	1	0	0	0	154	1	1229	71.27
	4. d4	1163	20	**10**	2	**4**	28	33	1	1227	76.86
	5. countAntom	1141	18	**10**	5	**4**	13	102	0	1191	84.39
	6. dpdb pg9	1159	19	5	2	0	0	0	0	1184	100.99
	7. c2d	1124	31	**10**	3	3	10	24	0	1181	84.41
	8. ganak	1031	16	10	2	**4**	29	**633**	0	1092	107.25
	9. sharpSAT	1029	16	**10**	2	**4**	30	80	0	1091	106.88
	10. sdd	1014	4	7	1	0	2	0	0	1028	124.23
	11. sts	927	4	8	**7**	5	**52**	35	**21**	1003	128.43
	12. dsharp	853	3	7	2	0	0	29	0	865	157.87
	13. cnf2eadt	799	3	7	2	0	7	157	0	818	170.17
	14. approxmc 3	794	3	7	2	0	6	1	0	812	173.35
	15. bdd_minisat	791	4	1	0	0	0	46	0	796	175.09

have instances up to an upper bound[8] of treewidth 35. Then, if only instances with TDs up to width 50 are considered, dpdb solves about the same number of instances than miniC2D solves.

7 Final discussion and conclusions

We presented a generic system dpdb for explicitly exploiting treewidth by means of DP on databases. The idea of dpdb is to use DBMSs for table manipulation, which makes it (i) easy and elegant to perform *rapid prototyping* for problems with DP algorithms and (ii) allows to leverage decades of database theory and database system tuning. It turned out that all the cases that occur in DP can be handled quite elegantly with plain SQL queries. Our system dpdb can be used for both decision and counting problems, thereby also considering optimization. We see our system particularly well suited for counting problems, especially, since it was shown that for model counting (#SAT) instances of practical relevance typically have small treewidth (Fichte *et al.* 2019). In consequence, we carried out preliminary experiments on publicly available instances for #SAT, where we see competitive behavior compared to most recent solvers.

7.1 Future work

Our results give rise to several research questions. We want to push toward other database systems and vendors. For example, we expect major improvements in commercial database management systems due to the availability of efficient enterprise features. In particular, we expect in the DBMS Oracle that the behavior when we use different strategies on how to write and evaluate our SQL queries, for example, subqueries

[8] These upper bounds were obtained via decomposer htd in at most two seconds.

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

compared to common table expressions. Currently, we do not create or use any indices, as preliminary tests showed that *meaningful B*tree indices* are hard to create and creation is oftentimes too expensive. Further, the exploration of bitmap indices, as available in Oracle *enterprise DBMS* would be worth trying in our case (and for #Sat), since one can efficiently combine database columns by using extremely *efficient bit operations*. It would also be interesting to investigate whether operating system features to handle memory access can be helpful (Fichte *et al.* 2020). In addition, one might consider dpdb in the setting of distributed algorithms such as dCountAntom (Burchard *et al.* 2016) and DMC (Lagniez *et al.* 2018).

It might be worth to rigorously test and explore our extensions on limiting the number of rows per table for *approximating* #Sat or other counting problems, cf., Chakraborty *et al.* (2016), Dueñas-Osorio *et al.* (2017), Sharma *et al.* (2019) and compare to the recent winners of the newly established model counting competition (Fichte *et al.* 2020). Recent results (Hecher *et al.* 2020) indicate that by using hybrid solving and abstractions our results can also be extended to projected model counting (Fichte *et al.* 2018).

Another interesting research direction is to study whether efficient data representation techniques on DBMSs can be combined with DP in order to lift our solver to quantified Boolean formulas.

It would also be interested to consider other measures such as (fractional) hypertree width (Fichte *et al.* 2020; Dzulfikar *et al.* 2019) and investigate whether TDs with additional properties (Jégou and Terrioux 2014) or other heuristics to compute TDs improve solving (Strasser 2017). Furthermore, interesting directions for future research would be to implement counting various problems in our framework, such as in constraint satisfaction (Durand and Mengel 2015; Khamis *et al.* 2016), constraint networks (Jégou and Terrioux 2014), argumentation (Fichte *et al.* 2019), description logics (Fichte *et al.* in press), or epistemic logic programming (Hecher *et al.* 2020).

System and License

Our system `dpdb` is available under GPL3 license at github.com/hmarkus/dp_on_dbs.

Acknowledgments

Part of this work was done while Johannes K. Fichte was visiting the Simons Institute for the Theory of Computing. Main work was carried out while he was a PostDoc at TU Dresden. The work has been supported by the Austrian Science Fund (FWF), Grants Y698 and P32830, as well as the Vienna Science and Technology Fund, Grant WWTF ICT19-065. Markus Hecher is also affiliated with the University of Potsdam, Germany.

References

Abseher, M., Musliu, N. and Woltran, S. 2017. htd – a free, open-source framework for (customized) TDPs and beyond. In *Proceedings of the 14th International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR'17)*, D. S. Lombardi, Ed. Lecture Notes in Computer Science, vol. 10335. Springer Verlag, Padua, Italy, 376–386.

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

Bacchus, F., Dalmao, S. and Pitassi, T. 2003. Algorithms and complexity results for #SAT and Bayesian inference. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS'03)*, C. S. Chekuri and D. Micciancio, Eds. IEEE Computer Society, Cambridge, MA, USA, 340–351.

Bannach, M. and Berndt, S. 2019. Practical access to dynamic programming on tree decompositions. *Algorithms 12,* 8, 172.

Bliem, B., Charwat, G., Hecher, M. and Woltran, S. 2016. D-flat^2: Subset minimization in dynamic programming on tree decompositions made easy. *Fundamentals of Informatics 147,* 1, 27–61.

Bodlaender, H. and Koster, A. 2008. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal 51,* 3, 255–269.

Bodlaender, H. L. 1988. Dynamic programming on graphs with bounded treewidth. In *Automata, Languages and Programming*, T. Lepistö and A. Salomaa, Eds. Springer, Berlin, Heidelberg, 105–118.

Bomanson, J., Gebser, M., Janhunen, T., Kaufmann, B. and Schaub, T. 2016. Answer set programming modulo acyclicity. *Fundamentals of Informatics 147,* 1, 63–91.

Burchard, J., Schubert, T. and Becker, B. 2015. Laissez-faire caching for parallel #SAT solving. In *SAT'15*, M. Heule and S. Weaver, Eds. Lecture Notes in Computer Science, vol. 9340. Springer Verlag, Austin, TX, USA, 46–61.

Burchard, J., Schubert, T. and Becker, B. 2016. Distributed parallel #SAT solving. In *Proceedings of the 2016 IEEE International Conference on Cluster Computing (CLUSTER'16)*, B. R. de Supinski, Ed. IEEE Computer Society, Taipei, Taiwan, 326–335.

Chakraborty, S., Fremont, D. J., Meel, K. S., Seshia, S. A. and Vardi, M. Y. 2014. Distribution-aware sampling and weighted model counting for SAT. In *Proceedings of the 28th Conference on Artificial Intelligence (AAAI'14)*, C. E. Brodley and P. Stone, Eds. The AAAI Press, Québec City, QC, Canada, 1722–1730.

Chakraborty, S., Meel, K. S. and Vardi, M. Y. 2016. Improving approximate counting for probabilistic inference: From linear to logarithmic SAT solver calls. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI'16)*, S. Kambhampati, Ed. The AAAI Press, New York City, NY, USA, 3569–3576.

Charwat, G. and Woltran, S. 2019. Expansion-based QBF solving on tree decompositions. *Fundamentals of Informatics 167,* 1–2, 59–92.

Choi, A., Van den Broeck, G. and Darwiche, A. 2015. Tractable learning for structured probability spaces: A case study in learning preference distributions. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*, Q. Yang, Ed. The AAAI Press, Buenos Aires, Argentina, 2861–2868.

Clark, K. L. 1977. Negation as failure. In *Logic and Data Bases*. Advances in Data Base Theory. Plemum Press, New York, 293–322.

Codd, E. F. 1970. A relational model of data for large shared data banks. *Communications of the ACM 13,* 6, 377–387.

Cygan, M., Fomin, F. V., Kowalik, Ł., Lokshtanov, D., Dániel Marx, M. P., Pilipczuk, M. and Saurabh, S. 2015. *Parameterized Algorithms*. Springer Verlag, Cham, Heidelberg, New York, Dordrecht, London.

Darwiche, A. 2004. New advances in compiling CNF to decomposable negation normal form. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04)*, R. López De Mántaras and L. Saitta, Eds. IOS Press, Valencia, Spain, 318–322.

Darwiche, A. 2011. SDD: A new canonical representation of propositional knowledge bases. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, T. Walsh, Ed. AAAI Press/IJCAI, Barcelona, Catalonia, Spain, 819–826.

Dechter, R. 1999. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence 113,* 1, 41–85.

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

Dell, H., Komusiewicz, C., Talmon, N. and Weller, M. 2018. The PACE 2017 parameterized algorithms and computational experiments challenge: The second iteration. In *IPEC 2017*, D. Lokshtanov and N. Nishimura, Eds. Leibniz International Proceedings in Informatics (LIPIcs), vol. 89. Dagstuhl Publishing, Vienna, Austria, 30:1–30:12.

Diestel, R. 2012. *Graph Theory*, 4th ed. Graduate Texts in Mathematics, vol. 173. Springer Verlag, Heidelberg.

Domshlak, C. and Hoffmann, J. 2007. Probabilistic planning via heuristic forward search and weighted model counting. *Journal of Artificial Intelligence Research 30*, 1–56.

Doubilet, P., Rota, G.-C. and Stanley, R. 1972. On the foundations of combinatorial theory. vi. the idea of generating function. In *Berkeley Symposium on Mathematical Statistics and Probability*, L. M. L. Cam, J. Neyman and E. L. Scott, Eds. University of California Press, Berkeley, CA, USA, 2: 267–318.

Dudek, J. M., Phan, V. H. N. and Vardi, M. Y. 2020. DPMC: Weighted model counting by dynamic programming on project-join trees. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP'20)*, H. Simonis, Ed. Lecture Notes in Computer Science, vol. 12333. Springer Verlag, Cham, 211–230.

Dueñas-Osorio, L., Meel, K. S., Paredes, R. and Vardi, M. Y. 2017. Counting-based reliability estimation for power-transmission grids. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI'17)*, S. P. Singh and S. Markovitch, Eds. The AAAI Press, San Francisco, CA, USA, 4488–4494.

Durand, A. and Mengel, S. 2015. Structural tractability of counting of solutions to conjunctive queries. *Theoretical Computer Science 57,* 4, 1202–1249.

Dzulfikar, M. A., Fichte, J. K. and Hecher, M. 2019. The PACE 2019 parameterized algorithms and computational experiments challenge: The fourth iteration (invited paper). In *Proceedings of the 14th International Symposium on Parameterized and Exact Computation (IPEC'19)*, B. M. P. Jansen and J. A. Telle, Eds. Leibniz International Proceedings in Informatics (LIPIcs), vol. 148. Dagstuhl Publishing, Munich, Germany, 25:1–25:23.

Elmasri, R. and Navathe, S. B. 2016. *Fundamentals of Database Systems*, 7th ed. Pearson.

Ermon, S., Gomes, C. P. and Selman, B. 2012. Uniform solution sampling using a constraint solver as an oracle. In *Proceedings of the 28th Conference on Uncertainty in Artificial Intelligence (UAI'12)*, N. de Freitas and K. Murphy, Eds. AUAI Press, Catalina Island, CA, USA, 255–264.

Fichte, J. K., Hecher, M. and Hamiti, F. 2020. The model counting competition 2020. CoRR abs/2012.01323, 1–25.

Fichte, J. K., Hecher, M. and Meier, A. 2019. Counting complexity for reasoning in abstract argumentation. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI'19)*, P. V. Hentenryck and Z.-H. Zhou, Eds. The AAAI Press, Honolulu, Hawaii, USA, 2827–2834.

Fichte, J. K., Hecher, M. and Meier, A. in press. Counting complexity for reasoning in abstract argumentation. In *Proceedings of the 35rd AAAI Conference on Artificial Intelligence (AAAI'21)*, K. Leyton-Brown and Mausam, Eds. The AAAI Press, Vancouver, Canada.

Fichte, J. K., Hecher, M., Morak, M. and Woltran, S. 2018. Exploiting treewidth for projected model counting and its limits. In *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT'18)*, O. Beyersdorff and C. M. Wintersteiger, Eds. Lecture Notes in Computer Science, vol. 10929. Springer Verlag, Oxford, UK, 165–184. Held as Part of the Federated Logic Conference, FloC'18.

Fichte, J. K., Hecher, M. and Pfandler, A. 2020. Lower bounds for QBFs of bounded treewidth. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'20)*, N. Kobayashi, Ed. Association for Computing Machinery, New York, Saarbrücken, Germany, 410–424.

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

Fichte, J. K., Hecher, M. and Schindler, I. 2018. Default logic and bounded treewidth. In *Proceedings of the 12th International Conference on Language and Automata Theory and Applications (LATA'18)*, S. T. Klein and C. Martín-Vide, Eds. Lecture Notes in Computer Science. Springer Verlag, Ramat Gan, Israel.

Fichte, J. K., Hecher, M. and Szeider, S. 2020. Breaking symmetries with rootclique and lextopsort. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP'20)*, H. Simonis, Ed. Lecture Notes in Computer Science, vol. 12333. Springer Verlag, Louvain-la-Neuve, Belgium, 286–303.

Fichte, J. K., Hecher, M., Thier, P. and Woltran, S. 2020. Exploiting database management systems and treewidth for counting. In *Proceedings of the 22nd International Symposium on Practical Aspects of Declarative Languages (PADL'20)*, E. Komendantskaya and Y. A. Liu, Eds. Lecture Notes in Computer Science, vol. 12007. Springer Verlag, New Orleans, LA, USA, 151–167.

Fichte, J. K., Hecher, M., Woltran, S. and Zisser, M. 2018a. A Benchmark Collection of #SAT Instances and Tree Decompositions (Benchmark Set). URL: https://doi.org/10.5281/zenodo.1299752.

Fichte, J. K., Hecher, M., Woltran, S. and Zisser, M. 2018b. Weighted model counting on the GPU by exploiting small treewidth. In *Proceedings of the 26th Annual European Symposium on Algorithms (ESA'18)*, Y. Azar, H. Bast and G. Herman, Eds. Leibniz International Proceedings in Informatics (LIPIcs), vol. 112. Dagstuhl Publishing, Helsinki, Finland, 28:1–28:16.

Fichte, J. K., Hecher, M. and Zisser, M. 2019. An improved GPU-based SAT model counter. In *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP'19)*, T. Schiex and S. de Givry, Eds. Springer Verlag, Stamford, CT, USA, 491–509.

Fichte, J. K., Manthey, N., Stecklina, J. and Schidler, A. 2020. Towards faster reasoners by using transparent huge pages. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP'20)*, H. Simonis, Ed. Lecture Notes in Computer Science, vol. 12333. Springer Verlag, Louvain-la-Neuve, Belgium, 304–322.

Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L. and Woodall, T. S. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, D. Kranzlmüller, P. Kacsuk and J. J. Dongarra, Eds. Lecture Notes in Computer Science, vol. 3241. Springer Verlag, Budapest, Hungary, 97–104.

Garcia-Molina, H., Ullman, J. D. and Widom, J. 2009. *Database Systems: the Complete Book*, 2nd ed. Pearson Prentice Hall, Upper Saddle River, New Jersey.

Gomes, C. P., Sabharwal, A. and Selman, B. 2009. Chapter 20: Model counting. In *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren and T. Walsh, Eds. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, Amsterdam, Netherlands, 633–654.

Gottlob, G., Pichler, R. and Wei, F. 2006. Bounded treewidth as a key to tractability of knowledge representation and reasoning. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI'06)*, J. A. Hendler, Ed. The AAAI Press, Boston, MA, USA, 250–256.

Gottlob, G., Pichler, R. and Wei, F. 2007. Efficient datalog abduction through bounded treewidth. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI'07)*, R. C. Holte and A. Howe, Eds. The AAAI Press, Vancouver, BC, Canada, 1626–1631.

Gottlob, G. and Szeider, S. 2007. Fixed-parameter algorithms for artificial intelligence, constraint satisfaction and database problems. *The Computer Journal 51,* 3 (09), 303–325.

Grohe, M. 2007. The complexity of homomorphism and constraint satisfaction problems seen from the other side. *Journal of the ACM 54,* 1, 1:1–1:24.

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

Hecher, M., Morak, M. and Woltran, S. 2020. Structural decompositions of epistemic logic programs. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI'20)*, V. Conitzer and F. Sha, Eds. The AAAI Press, New York City, NY, USA, 2830–2837.

Hecher, M., Thier, P. and Woltran, S. 2020. Taming high treewidth with abstraction, nested dynamic programming, and database technology. In *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT'20)*, L. Pulina and M. Seidl, Eds. Lecture Notes in Computer Science, vol. 12178. Springer Verlag, Alghero, Italy, 343–360.

Janhunen, T. 2006. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics 16,* 1–2, 35–86.

Jégou, P. and Terrioux, C. 2014. Tree-decompositions with connected clusters for solving constraint networks. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP'14)*, B. O'Sullivan, Ed. Lecture Notes in Computer Science, vol. 8656. Springer Verlag, Lyon, France, 407–423.

Khamis, M. A., Ngo, H. Q. and Rudra, A. 2016. FAQ: Questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS'16)*, T. Milo and W. Tan, Eds. Association for Computing Machinery, New York, San Francisco, CA, USA, 13–28.

Kiljan, K. and Pilipczuk, M. 2018. Experimental evaluation of parameterized algorithms for feedback vertex set. In *Proceedings of the 17th International Symposium on Experimental Algorithms (SEA'18)*, G. D'Angelo, Ed. Leibniz International Proceedings in Informatics (LIPIcs), vol. 103. Dagstuhl Publishing, Dagstuhl, Germany, 12:1–12:12.

Kleine Büning, H. and Lettman, T. 1999. *Propositional Logic: Deduction and Algorithms*. Cambridge University Press, Cambridge, New York, NY, USA.

Kloks, T. 1994. *Treewidth. Computations and Approximations*. Lecture Notes in Computer Science, vol. 842. Springer Verlag, Berlin, Heidelber.

Koriche, F., Lagniez, J.-M., Marquis, P. and Thomas, S. 2013. Knowledge compilation for model counting: Affine decision trees. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI'13)*, F. Rossi and S. Thrun, Eds. The AAAI Press, Beijing, China.

Lagniez, J. and Marquis, P. 2014. Preprocessing for propositional model counting. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI'14)*, C. E. Brodley and P. Stone, Eds. The AAAI Press, Québec City, QC, Canada, 2688–2694.

Lagniez, J.-M. and Marquis, P. 2017. An improved decision-DDNF compiler. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, C. Sierra, Ed. The AAAI Press, Melbourne, VIC, Australia, 667–673.

Lagniez, J.-M., Marquis, P. and Szczepanski, N. 2018. DMC: A distributed model counter. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI'18)*, J. Lang, Ed. The AAAI Press, Stockholm, Sweden, 1331–1338.

Langer, A., Reidl, F., Rossmanith, P. and Sikdar, S. 2012. Evaluation of an MSO-solver. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments (ALENEX'12)*, D. A. Bader and P. Mutzel, Eds. Society for Industrial and Applied Mathematics (SIAM), Miyako, Kyoto, Japan, 55–63.

Liu, J., Zhong, W. and Jiao, L. 2006. Comments on "the 1993 DIMACS graph coloring challenge" and "energy function-based approaches to graph coloring". *IEEE Transactions on Neural Networks 17,* 2, 533.

Muise, C. J., McIlraith, S. A., Beck, J. C. and Hsu, E. I. 2012. Dsharp: Fast d-DNNF compilation with sharpSAT. In *Proceedings of the 25th Canadian Conference on Artificial Intelligence (AI'17)*, L. Kosseim and D. Inkpen, Eds. Lecture Notes in Computer Science, vol. 7310. Springer Verlag, Toronto, ON, Canada, 356–361.

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press

Oztok, U. and Darwiche, A. 2015. A top-down compiler for sentential decision diagrams. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*, Q. Yang and M. Wooldridge, Eds. The AAAI Press, Buenos Aires, Argentina, 3141–3148.

Pan, G. and Vardi, M. Y. 2006. Fixed-parameter hierarchies inside PSPACE. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, R. Alur, Ed. IEEE Computer Society, Seattle, WA, USA, 27–36.

PostgreSQL Global Development Group. 2020. PostgreSQL documentation 12. URL: https://www.postgresql.org/docs/12/queries-with.html.

Roth, D. 1996. On the hardness of approximate reasoning. *Artificial Intelligence 82,* 1–2, 273–302.

Samer, M. and Szeider, S. 2010. Algorithms for propositional model counting. *Journal of Discrete Algorithms 8,* 1, 50—64.

Sang, T., Bacchus, F., Beame, P., Kautz, H. and Pitassi, T. 2004. Combining component caching and clause learning for effective model counting. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, H. H. Hoos and D. G. Mitchell, Eds. Springer Verlag, Vancouver, BC, Canada, 1–9.

Sang, T., Beame, P. and Kautz, H. 2005. Performing Bayesian inference by weighted model counting. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI'05)*, M. M. Veloso and S. Kambhampati, Eds. The AAAI Press, Pittsburgh, Pennsylvania, USA, 475–481.

Sharma, S., Roy, S., Soos, M. and Meel, K. S. 2019. GANAK: A scalable probabilistic exact model counter. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI'19)*, S. Kraus, Ed. ijcai.org, Macao, China, 1169–1176.

Strasser, B. 2017. Computing tree decompositions with flowcutter: PACE 2017 submission. CoRR abs/1709.08949, 1–9.

Thurley, M. 2006. sharpSAT – counting models with advanced component caching and implicit BCP. In *Proceedings of the 9th International Conference on International Conference on Theory and Applications of Satisfiability Testing (SAT'06)*, A. Biere and C. P. Gomes, Eds. Lecture Notes in Computer Science, vol. 4121. Springer Verlag, Seattle, WA, USA, 424–429.

Toda, S. 1991. PP is as hard as the polynomial-time hierarchy. *SIAM Journal on Computing 20,* 5, 865–877.

Toda, T. and Soh, T. 2015. Implementing efficient all solutions SAT solvers. *ACM Journal of Experimental Algorithmics 21*, 1.12. Special Issue SEA 2014.

Ullman, J. D. 1989. *Principles of Database and Knowledge-Base Systems, Volume II.* Computer Science Press, New York, NY, USA.

Xue, Y., Choi, A. and Darwiche, A. 2012. Basing decisions on sentences in decision diagrams. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI'12)*, J. Hoffmann and B. Selman, Eds. The AAAI Press, Toronto, ON, Canada, 842–849.

https://doi.org/10.1017/S147106842100003X Published online by Cambridge University Press