Exploiting Database Management Systems and Treewidth for Counting[⋆]

Johannes K. Fichte[1][0000−0002−8681−7470], Markus Hecher[2,3][0000−0003−0131−6771], Patrick Thier[2], and Stefan Woltran[2][0000−0003−1594−8972]

[1] TU Dresden, Germany johannes.fichte@tu-dresden.de
[2] TU Wien, Austria {hecher,woltran,thier}@dbai.tuwien.ac.at
[3] University of Potsdam, Germany hecher@uni-potsdam.de

Abstract Bounded treewidth is one of the most cited combinatorial invariants, which was applied in the literature 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 (DBMS). 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 DBMS naturally allow to deal with huge tables with a limited amount of main memory (RAM), parallelization, as well as suspending computation.

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 [8,17,24,38]. In combinatorics, for instance, one characterizes the number of solutions to problems by means of mathematical expressions, e.g., generating functions [18]. One particular counting problem, namely *model counting* (#SAT) asks to output the number of solutions of a given Boolean formula. Model counting and variants thereof have already been applied for solving a variety of real-world applications [8,10,19,43]. Such problems are typically considered rather hard, since #SAT is complete for the class #P [3,35], i.e., one can simulate

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

any problem of the polynomial hierarchy with polynomially many calls [40] to a #Sat solver. Taming this high complexity is possible with techniques from parameterized complexity [12]. In fact, many of the publicly available #Sat instances show good structural properties after using regular preprocessors like pmc [29], see [23]. By good structural properties, we mean that graph representations of these instance have reasonably small *treewidth*. The measure treewidth is a structural parameter of graphs which models the closeness of the graph to being a tree and is one of the most cited combinatorial invariants studied in parameterized complexity [12], and subject of recent competitions [15].

This observation gives rise to a general framework for counting problems that leverages treewidth. The general idea to develop such frameworks is indeed not new, since there are both, specialized solvers [9,23,25], as well as general systems like D-FLAT [5], Jatatosk [4], and sequoia [31], that exploit treewidth. 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 form of SQL `SELECT` queries. The actual run of the DP algorithm is then delegated to our system `dpdb`, which employs *database management systems (DBMS)* [42]. 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 limited amount of main memory (RAM), dedicated database joins, as well as query optimization and data-dependent execution plans.

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 database management systems to handle table operations needed for performing dynamic programming 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 by means of table operations (in form of relational algebra and SQL) on tree decompositions. As a result, `dpdb` is a generalized framework for dynamic programming on tree decompositions, where one only needs to specify the essential and problem-specific parts of dynamic programming 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. Concretely, we compare the runtime of our system with state-of-the-art model counters, where we observe competitive behavior and promising indications for future work.

2 Preliminaries

We assume familiarity with terminology of graphs and trees. For details, we refer to the literature and standard textbooks [16].

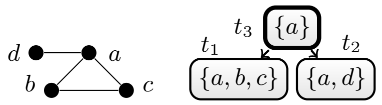


Figure 1: Graph G (left) with a TD $\mathcal{T}$ of graph G (right).

Boolean Satisfiability. We define Boolean formulas and their evaluation in the usual way, cf., [26]. 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 $\text{var}(X)$ the variables that occur in X. An *assignment* of φ is a mapping $I : \text{var}(\varphi) \to \{0,1\}$. 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.

Tree Decomposition and Treewidth. A *tree decomposition (TD)* [27,12] of a given graph G is a pair $\mathcal{T} = (T, \chi)$ 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 $\text{width}(\mathcal{T}) := \max_{t\in V(T)} |\chi(t)| - 1$. The *treewidth* $\text{tw}(G)$ of G is the minimum $\text{width}(\mathcal{T})$ over all TDs $\mathcal{T}$ of G. For a node $t \in V(T)$, we say that $\text{type}(t)$ 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 $t \in V(T)$, $\text{type}(t) \in \{$*leaf*, *join*, *intr*, *rem*$\}$, then the TD is called *nice*.

Example 1. Figure 1 depicts a graph G and a TD $\mathcal{T}$ of G of width 2. The treewidth of G is also 2 since G contains a complete graph with 3 vertices [27]. ■

Relational Algebra. We use relational algebra [11] for manipulation of relations, which forms the theoretical basis of its the well-known implementation database standard *Structured Query Language (SQL)* [42] on tables. An *attribute* a is of a certain finite *domain* $\text{dom}(a)$. Then, a *tuple* r over set $\text{att}(r)$ of attributes is a set of pairs of the form (a, v) with $a \in \text{att}(r), v \in \text{dom}(a)$ s.t. for each $a \in \text{att}(r)$, there is exactly one $v \in \text{dom}(a)$ with $(a, v) \in r$. A *relation* R is a finite set of tuples r over set $\text{att}(R) := \text{att}(r)$ of attributes. Given a relation R over $\text{att}(R)$. Then, we let $\text{dom}(R) := \bigcup_{a\in\text{att}(R)} \text{dom}(a)$, and let relation R *projected to* $A \subseteq \text{att}(R)$ be given by $\Pi_A(R) := \{r_A \mid r \in R\}$, where $r_A := \{(a, v) \mid (a, v) \in r, a \in A\}$. This concept can be lifted to *extended projection* $\dot{\Pi}_{A,S}$, where we assume in addition to $A \subseteq \text{att}(R)$, a set S of expressions of the form $a \leftarrow f$, such that $a \in \text{att}(R) \setminus A$, and f is an arithmetic function that takes a tuple $r \in R$, such that there is at most one expression in S for each $a \in \text{att}(R) \setminus A$. Formally, we define $\dot{\Pi}_{A,S}(R) := \{r_A \cup r^S \mid r \in R\}$ with $r^S := \{(a, f(r)) \mid a \in \text{att}(r), (a \leftarrow f) \in S\}$. Later, we use *aggregation by grouping* ${}_AG_{(a\leftarrow g)}$, where we assume $A \subseteq \text{att}(R), a \in \text{att}(R) \setminus A$ and a so-called *aggregate function* g, which takes a relation $R' \subseteq R$ and returns a

Listing 1: Table algorithm $\mathsf{S}(t, \chi(t), \varphi_t, \langle\tau_1, \dots, \tau_\ell\rangle)$ for #Sat [36] using nice TD.

```
In: Node $t$, bag $\chi(t)$, clauses $\varphi_t$, sequence $\langle\tau_1, \dots \tau_\ell\rangle$ of child tables. Out: Table $\tau_t$.
if type($t$) = leaf then $\tau_t := \{\langle\emptyset, 1\rangle\}$
else if type($t$) = intr, and $a \in \chi(t)$ is introduced then
    $\tau_t := \{\langle J, c\rangle \mid \langle I, c\rangle \in \tau_1, J \in \{I^+_{a\mapsto 0}, I^+_{a\mapsto 1}\}, \varphi_t(J) = \emptyset\}$
else if type($t$) = rem, and $a \notin \chi(t)$ is removed then
    $\tau_t := \{\langle I^-_a, \Sigma_{\langle J,c\rangle\in\tau_1 : I^-_a = J^-_a} c\rangle \mid \langle I, \cdot\rangle \in \tau_1\}$
else if type($t$) = join then
    $\tau_t := \{\langle I, c_1 \cdot c_2\rangle \mid \langle I, c_1\rangle \in \tau_1, \langle I, c_2\rangle \in \tau_2\}$
```

$S^-_e := S \setminus \{e \mapsto 0, e \mapsto 1\}$, $S^+_s := S \cup \{s\}$.

value of domain $\mathrm{dom}(a)$. Therefore, we let ${}_AG_{(a\leftarrow g)}(R) := \{r \cup \{(a, g(R[r]))\} \mid r \in \Pi_A(R)\}$, where $R[r] := \{r' \mid r' \in R, r \subseteq r'\}$. We define *renaming* of R given set A of attributes, and a bijective mapping $m : \mathrm{att}(R) \to A$ s.t. $\mathrm{dom}(a) = \mathrm{dom}(m(a))$ for $a \in \mathrm{att}(R)$, by $\rho_m(R) := \{(m(a), v) \mid (a, v) \in R\}$. *Selection* of rows in R according to a given Boolean formula φ with equality[4] is defined by $\sigma_\varphi(R) := \{r \mid r \in R, \varphi(r^E) = \emptyset\}$, where r^E is a truth assignment over $\mathrm{var}(\varphi)$ such that for each $v, v', v'' \in \mathrm{dom}(R) \cup \mathrm{att}(R)$ (1) $r^E(v{\approx}v') = 1$ if $(v, v') \in r$, (2) $r^E(v{\approx}v) = 1$, (3) $r^E(v{\approx}v') = r^E(v'{\approx}v)$, and (4) if $r^E(v{\approx}v') = 1$, and $r^E(v'{\approx}v'') = 1$, then $r^E(v{\approx}v'') = 1$. Given a relation R' with $\mathrm{att}(R') \cap \mathrm{att}(R) = \emptyset$. Then, we refer to the *cross-join* by $R \times R' := \{r \cup r' \mid r \in R, r' \in R'\}$. Further, a *$\theta$-join* (according to φ) corresponds to $R \bowtie_\varphi R' := \sigma_\varphi(R \times R')$.

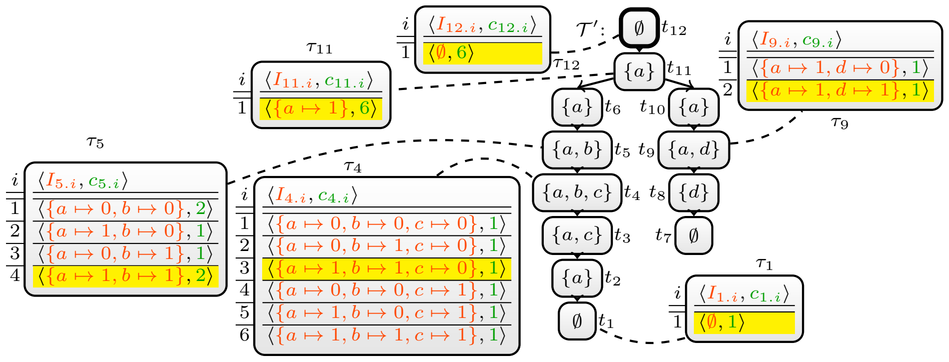
3 Towards Relational Algebra for Dynamic Programming

A solver based on *dynamic programming (DP)* evaluates the input $\mathcal{I}$ 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 $\mathcal{I}$, thereby considering tables $\tau_{t'}$ for child nodes t' of t. DP works for many problems $\mathcal{P}$ as follows.

1. Construct a graph representation G of the given input instance $\mathcal{I}$.
2. Heuristically compute a tree decomposition $\mathcal{T} = (T, \chi)$ of G.
3. Traverse the nodes in $V(T)$ in post-order, i.e., 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 problem* $\mathcal{P}(t, \mathcal{I}) = \mathcal{I}_t$ depending on $\mathcal{P}$, 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 $\mathcal{I}$.

For solving problem $\mathcal{P}$ = #Sat, we need the following graph representation. The *primal graph* G_φ [36] 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 $\mathcal{T} = (T, \chi)$ of G_φ and a node t of T. Sometimes, we refer to the

[4] We allow for φ to contain expressions $v{\approx}v'$ as variables for $v, v' \in \mathrm{dom}(R) \cup \mathrm{att}(R)$, and we abbreviate for $v \in \mathrm{att}(R)$ with $\mathrm{dom}(v) = \{0, 1\}$, $v{\approx}1$ by v and $v{\approx}0$ by $\neg v$.



i	$\langle I_{12.i}, c_{12.i}\rangle$
1	$\langle \emptyset, 6\rangle$

i	$\langle I_{11.i}, c_{11.i}\rangle$
1	$\langle \{a \mapsto 1\}, 6\rangle$

i	$\langle I_{9.i}, c_{9.i}\rangle$
1	$\langle \{a \mapsto 1, d \mapsto 0\}, 1\rangle$
2	$\langle \{a \mapsto 1, d \mapsto 1\}, 1\rangle$

i	$\langle I_{5.i}, c_{5.i}\rangle$
1	$\langle \{a \mapsto 0, b \mapsto 0\}, 2\rangle$
2	$\langle \{a \mapsto 1, b \mapsto 0\}, 1\rangle$
3	$\langle \{a \mapsto 0, b \mapsto 1\}, 1\rangle$
4	$\langle \{a \mapsto 1, b \mapsto 1\}, 2\rangle$

i	$\langle I_{4.i}, c_{4.i}\rangle$
1	$\langle \{a \mapsto 0, b \mapsto 0, c \mapsto 0\}, 1\rangle$
2	$\langle \{a \mapsto 0, b \mapsto 1, c \mapsto 0\}, 1\rangle$
3	$\langle \{a \mapsto 1, b \mapsto 1, c \mapsto 0\}, 1\rangle$
4	$\langle \{a \mapsto 0, b \mapsto 0, c \mapsto 1\}, 1\rangle$
5	$\langle \{a \mapsto 1, b \mapsto 0, c \mapsto 1\}, 1\rangle$
6	$\langle \{a \mapsto 1, b \mapsto 1, c \mapsto 1\}, 1\rangle$

i	$\langle I_{1.i}, c_{1.i}\rangle$
1	$\langle \emptyset, 1\rangle$

Figure 2: Selected tables obtained by DP on $\mathcal{T}'$ for φ of Example 2 using algorithm S.

treewidth of the primal graph of a given formula by the *treewidth of the formula*. Then, we let local problem $\#\textsc{Sat}(t, \varphi) = \varphi_t$ 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 S as presented in Listing 1 shows all the cases that are needed to solve $\#\textsc{Sat}$ 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 $\mathrm{type}(t) = \textit{leaf}$ 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.

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, e.g., $\{a \mapsto 1, b \mapsto 1, c \mapsto 0, d \mapsto 0\}$, $\{a \mapsto 1, b \mapsto 0, c \mapsto 1, d \mapsto 0\}$ or $\{a \mapsto 1, b \mapsto 1, c \mapsto 1, d \mapsto 1\}$. In total, there are 6 satisfying assignments of φ. Observe that graph G of Figure 1 actually depicts the primal graph G_φ of φ. Intuitively, $\mathcal{T}$ of Figure 1 allows to evaluate formula φ in parts. Figure 2 illustrates a nice TD $\mathcal{T}' = (\cdot, \chi)$ of the primal graph G_φ and tables τ_1, …, τ_{12} that are obtained during the execution of S on nodes $t_1, \ldots, t_{12}$. We assume that each row in a table τ_t is identified by a number, i.e., row i corresponds to $\boldsymbol{u_{t.i}} = \langle I_{t.i}, c_{t.i}\rangle$.

Table $\tau_1 = \{\, \langle \emptyset, 1\rangle \,\}$ has $\mathrm{type}(t_1) = \textit{leaf}$. Since $\mathrm{type}(t_2) = \textit{intr}$, 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 S enforces satisfiability of φ_t in node t. Since $\mathrm{type}(t_5) = \textit{rem}$, 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 $\mathrm{type}(t_{11}) = \textit{join}$, we build table τ_{11} by taking

Listing 2: Alternative table algorithm $\mathsf{S}_{\mathsf{RAlg}}(t, \chi(t), \varphi_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for #SAT.

```
   In: Node $t$, bag $\chi(t)$, clauses $\varphi_t$, sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child tables. Out: Table $\tau_t$.
1  if type($t$) = leaf then $\tau_t := \{\{(\text{cnt}, 1)\}\}$
2  else if type($t$) = intr, and $a \in \chi(t)$ is introduced then
3      $\tau_t := \tau_1 \bowtie_{\varphi_t} \{\{([\![a]\!], 0)\}, \{([\![a]\!], 1)\}\}$
4  else if type($t$) = rem, and $a \notin \chi(t)$ is removed then
5      $\tau_t := {}_{\chi(t)}G_{\text{cnt} \leftarrow \texttt{SUM}(\text{cnt})}(\Pi_{\text{att}(\tau_1) \setminus \{[\![a]\!]\}} \tau_1)$
6  else if type($t$) = join then
7      $\tau_t := \dot{\Pi}_{\chi(t), \{\text{cnt} \leftarrow \text{cnt} \cdot \text{cnt}'\}}(\tau_1 \bowtie_{\bigwedge_{a \in \chi(t)} [\![a]\!] \approx [\![a]\!]'} \rho_{\bigcup_{a \in \text{att}(\tau_2)} \{[\![a]\!] \mapsto [\![a]\!]'\}} \tau_2)$
```

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 be empty ($\emptyset$). ■

Alternative: Relational Algebra. Instead of using set theory to describe how tables are obtained during dynamic programming, one could alternatively use relational algebra. There, tables τ_t for each TD node t are pictured as relations, where τ_t distinguishes a unique column (attribute) $[\![x]\!]$ for each $x \in \chi(t)$. Further, there might be additional attributes required depending on the problem at hand, e.g., we need an attribute cnt for counting in #SAT, or an attribute for modeling costs or weights in case of optimization problems. Listing 2 presents a table algorithm for problem #SAT that is equivalent to Listing 1, but relies on relational algebra only for computing tables. This 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 (highlighted by coloring in Listing 1). In particular, one typically derives for nodes t with type(t) = *leaf*, a fresh initial table τ_t, cf., Line 1 of Listing 2. Then, whenever an atom a is introduced, such algorithms often use θ-joins with a fresh initial table for the introduced variable a representing potential values for a. In Line 3 the selection of the θ-join is performed according to φ_t, i.e. corresponding to the local problem of #SAT. Further, for nodes t with type(t) = *rem*, these table algorithms typically need projection. In case of Listing 2, Line 5 also needs grouping in order to maintain the counter, as several rows of τ_1 might collapse in τ_t. Finally, for a node t with type(t) = *join*, in Line 7 we use extended projection and θ-joins, where we join on the same truth assignments, which allows us later to leverage advanced database technology. Extended projection is needed for multiplying the counters of the two rows containing the same assignment.

4 Dynamic Programming on TDs using Databases & SQL

In this section, we present a general architecture to model table algorithms by means of database management systems. The architecture is influenced by the DP approach of the previous section and works as depicted in Figure 3, where

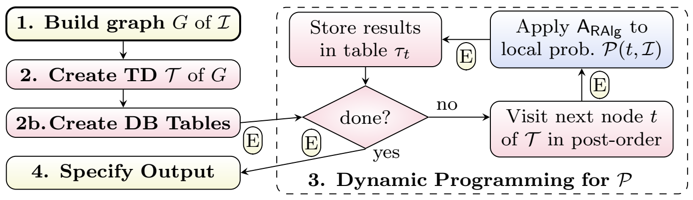

1. Build graph G of $\mathcal{I}$
2. Create TD $\mathcal{T}$ of G
2b. Create DB Tables
4. Specify Output
Store results in table τ_t
Apply $\mathsf{A}_{\mathsf{RAlg}}$ to local prob. $\mathcal{P}(t,\mathcal{I})$
done?
no
yes
Visit next node t of $\mathcal{T}$ in post-order
E
3. Dynamic Programming for $\mathcal{P}$

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

the steps highlighted in yellow and blue need to be specified depending on the problem $\mathcal{P}$. 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 sub-tables for each node during Step 3 within such events. Observe that after the generation of a TD $\mathcal{T} = (T, \chi)$, Step 2b automatically creates tables τ_t for each node t of T, where the corresponding table schema of τ_t is specified in the blue part, i.e., within $\mathsf{A}_{\mathsf{RAlg}}$. The *default schema* of such a table τ_t that is assumed in this section foresees one column for each element of the bag $\chi(t)$, where additional columns such as counters or costs can be added.

Actually, the core of this architecture is focused on the table algorithm $\mathsf{A}_{\mathsf{RAlg}}$ executed for each node t of T of TD $\mathcal{T} = (T, \chi)$. Besides the definition of table schemes, the blue part concerns specification of the table algorithm by means of a procedural *generator template* that describes how to dynamically obtain SQL codefor each node t thereby oftentimes depending on $\chi(t)$. This generated SQL code is then used internally for manipulation of tables τ_t during the tree decomposition traversal in Step 3 of dynamic programming. Listing 3 presents a general template, where parts of table algorithms for problems that are typically problem-specific are replaced by colored placeholders of the form #placeHolder#, cf., Listing 2. Observe that Line 3 of Listing 3 uses extended projection as in Line 7. This is needed for some problems requiring changes on vertex introduction.

Note, however, that the whole architecture does not depend on certain normalization or forms of TDs, e.g., 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 Listing 3, 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.

4.1 System dpdb: Dynamic Programming with Databases

We implemented the proposed architecture of the previous section in the prototypical dpdb system. The system is open-source[5], written in Python 3 and uses

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

Listing 3: Template of $\mathsf{A}_{\mathsf{RAlg}}(t, \chi(t), \mathcal{I}_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ of Figure 3 for problem $\mathcal{P}$.

```
  In: Node $t$, bag $\chi(t)$, instance $\mathcal{I}_t$, sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child tables. Out: Table $\tau_t$.
1 if type($t$) = leaf then $\tau_t := \#\varepsilon\mathsf{Tab}\#$
2 else if type($t$) = intr, and $a \in \chi(t)$ is introduced then
3     $\tau_t := \dot{\Pi}_{\chi(t),\#\mathsf{extProj}\#}(\tau_1 \bowtie_{\#\mathsf{localProbFilter}\#} \#\mathsf{intrTab}\#)$
4 else if type($t$) = rem, and $a \notin \chi(t)$ is removed then
5     $\tau_t := {}_{\chi(t)}G_{\#\mathsf{aggrExp}\#}(\Pi_{\mathrm{att}(\tau_1)\setminus\{[\![a]\!]\}}\tau_1)$
6 else if type($t$) = join then
7     $\tau_t := \dot{\Pi}_{\chi(t),\#\mathsf{extProj}\#}(\tau_1 \bowtie_{\bigwedge_{a\in\chi(t)}[\![a]\!]\approx[\![a]\!]'} \rho_{\bigcup_{a\in\mathrm{att}(\tau_2)}\{[\![a]\!]\mapsto[\![a]\!]'\}}\tau_2)$
```

PostgreSQL as DBMS. We are convinced though 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 [2], which finds TDs extremely quick also for interesting instances [23] due to heuristics. Note that `dpdb` directly supports the TD format of recent competitions [15], i.e., one could easily replace the TD library. It is important though to not enforce htd to compute nice TDs, as this would cause a 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. Then, especially when there are huge tables involved, θ-joins among child node tables cover at most a limited number of child node tables. In consequence, the query optimizer of the database system still has a chance to come up with meaningful execution plans depending on the contents of the table. Note that though one should consider θ-joins with more than just two tables, since such binary θ-joins already fix in which order these tables shall be combined, thereby again limiting 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, i.e., each θ-join covers at most 5 child tables.

Towards 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 Listing 3. Concretely, the system executes one query per table τ_t for each node t during the traversal of TD $\mathcal{T}$. This query consists of serveral parts and we briefly explain its parts from outside to inside. First of all, the inner-most part concerns the *row candiates* for τ_t consisting of the θ-join as in Line 7 of Listing 3, including parts of Line 3, namely cross-joins for each introduced variable, involving #intrTab# without the filtering on #localProbFilter#. Then, there are different configurations of `dpdb` concerning these row candidates. For debugging (see below) one could (1) actually materialize the result in a table, whereas for performance runs, 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, projection[6] and grouping involving #aggrExp# as in Line 5 of Listing 3, as well as selection acording to #localProbFilter#, cf., Line 3, is specified. It turns out that PostgreSQL can do better with sub-queries, where the query optimizer oftentimes pushes selection and projection into the sub-query if needed, which is not the case for CTEs, as discussed in the PostgreSQL manual [1, Sec. 7.8.1]. On different DBMS or other vendors, e.g., Oracle, it might be better to use CTEs instead.

Example 3. Consider again Example 2 and Figure 1. If we use table algorithm $\mathsf{S}_{\mathsf{RAlg}}$ with dpdb on formula φ of TD $\mathcal{T}$ and Option (3): sub-queries, where the row candidates are expressed via a sub-queries. Then, for each node t_i of $\mathcal{T}$, 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 assignment 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 true AS val UNION ALL SELECT false)
   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 true AS val UNION ALL SELECT false)
   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 over-restrict 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 2 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

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

increasing explainability. The former for instance 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 the intermediate results, we record database timestamps before and after certain nodes, provide statistics as, e.g., width, number of rows, etc. 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 parser for each problem individually. Out-of-the-box, we support the formats DIMACS sat and DIMACS graph [32] as well as the common format for TDs [15].

4.2 Table algorithms with `dpdb` for selected problems

The system `dpdb` allows for *easy protyping* of DP algorithms on TDs. This covers decision problems, counting problems as well as optimization problems. As a proof of concept, we present the relevant parts of table algorithm specification according to the template in Listing 3 for a selection of problems below[7]. To this end, we assume in this section a not necessarily nice TD $\mathcal{T} = (T, \chi)$ of the corresponding graph representation of our given instance $\mathcal{I}$. Further, for the following specifications of the table algorithm using the template $\mathsf{A}_{\mathsf{RAlg}}$ in Listing 2, we assume any node t of T and its child nodes $t_1, \ldots, t_\ell$.

Problem #SAT. Given instance formula $\mathcal{I} = \varphi$. Then, specific parts for #SAT for node t with $\varphi_t = \{\{l_{1,1}, \ldots, l_{1,k_1}\}, \ldots, \{l_{n,1}, \ldots, l_{n,k_n}\}\}$.

- #εTab#: `SELECT 1 AS cnt`
- #intrTab#: `SELECT 1 AS val UNION ALL 0`
- #localProbFilter#: $(l_{1,1}$ `OR ... OR` $l_{1,k_1})$ `AND ... AND` $(l_{n,1}$ `OR ... OR` $l_{n,k_n})$
- #aggrExp#: `SUM(cnt) AS cnt`
- #extProj#: τ_1`.cnt * ... *` τ_ℓ`.cnt AS cnt`

Observe that for the corresponding decision problem SAT, where the goal is to decide only the existence of a satisfying assignment for given formula φ, #εTab# returns the empty table and parts #aggrExp#, #extProj# are just empty since there is no counter needed.

[7] Implementation for problems #SAT as well as MINVC is readily available in `dpdb`.

Problem #o-Col. For given input graph $\mathcal{I} = G = (V, E)$, a *o-coloring* is a mapping $\iota : V \rightarrow \{1, \ldots, o\}$ such that for each edge $\{u, v\} \in E$, we have $\iota(u) \neq \iota(v)$. Problem #o-Col asks to count the number of o-colorings of G. Local problem #o-Col(t, G) is defined by the graph $G_t := (V \cap \chi(t), E \cap [\chi(t) \times \chi(t)])$.

Specific parts for #o-Col for node t with $E(G_t) = \{\{u_1, v_1\}, \ldots, \{u_n, v_n\}\}$.

- #εTab#: `SELECT 1 AS cnt`
- #intrTab#: `SELECT 1 AS val UNION ALL ... UNION ALL` o
- #localProbFilter#: `NOT` ($\llbracket u_1 \rrbracket = \llbracket v_1 \rrbracket$) `AND ... AND NOT` ($\llbracket u_n \rrbracket = \llbracket v_n \rrbracket$)
- #aggrExp#: `SUM(cnt) AS cnt`
- #extProj#: τ_1`.cnt * ... *` τ_ℓ`.cnt AS cnt`

Problem MinVC. Given input graph $\mathcal{I} = G = (V, E)$, 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, i.e., C is such that there is no vertex cover C' with $|C'| < |C|$. Local problem MinVC$(t, G) := G_t$ is defined as above. We use an additional column `card` for storing cardinalities.

Problem MinVC for node t with $E(G_t) = \{\{u_1, v_1\}, \ldots, \{u_n, v_n\}\}$ and $\chi(t) = \{a_1, \ldots, a_k\}$ can be specified as follows.

- #εTab#: `SELECT 0 AS card`
- #intrTab#: `SELECT 1 AS val UNION ALL 0`
- #localProbFilter#: ($\llbracket u_1 \rrbracket$ `OR` $\llbracket v_1 \rrbracket$) `AND ... AND` ($\llbracket u_n \rrbracket$ `OR` $\llbracket v_n \rrbracket$)
- #aggrExp#: `MIN(card) AS card`
- #extProj#: τ_1`.card + ... +` τ_ℓ`.card - (`$\Sigma_{i=1}^{\ell} |\chi(t_i) \cap \{a_1\}|$ `- 1) *` τ_1`.`$\llbracket a_1 \rrbracket$ `- ... - (`$\Sigma_{i=1}^{\ell} |\chi(t_i) \cap \{a_k\}|$ `- 1) *` τ_1`.`$\llbracket a_k \rrbracket$

Observe that #ExtProj# is a bit more involved on non-nice TDs, as, whenever the column for a vertex a is set to 1, i.e., vertex a is in the vertex cover, we have to consider a only with cost 1, also if a appears in several child node bags.

Note that concrete implementations could generate and apply parts of this specification, as for example in #localProbFilter# only edges involving newly introduced vertices need to be checked.

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

5 Experiments

We conducted a series of experiments using publicly available benchmark sets for #Sat. Our tested benchmarks [22] are publicly available, and our results are also on github at github.com/hmarkus/dp_on_dbs/padl2020.

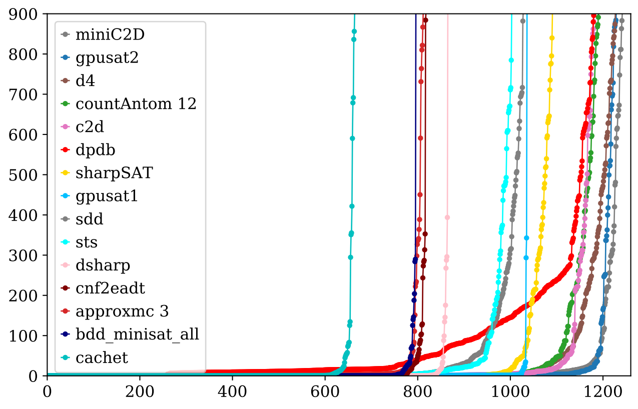


Figure 4: Runtime for the top 15 solvers over all #Sat instances. The x-axis refers to the number of instances and the y-axis depicts the runtime sorted in ascending order for each solver individually.

5.1 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 seconds and limited available RAM to 14 GB per instance and solver.

Benchmark Instances. We considered a selection of overall 1494 instances from various publicly available benchmark sets #Sat consisting of `fre/meel` benchmarks[8](1480 instances), and `c2d` benchmarks[9] (14 instances). However, we considered instances preprocessed by regular #Sat preprocessor *pmc* [29], similar to results of recent work on #Sat [23], where it was also shown that more than 80% of the #Sat instances have primal treewidth below 19 after preprocessing.

Benchmarked system `dpdb`*.* We used PostgreSQL 9.5 for our system `dpdb`, which was available on our benchmark described hardware below. However, we expect major performance increases if higher versions are used, which was not available on our benchmark machines. In particular, parallel queries, where a query is evaluated in parallel, were added and improved in every version greater than 9.6.

Other benchmarked systems. In our experimental work, we present results for the most recent versions of publicly available #Sat solvers, namely, *c2d* 2.20 [13], *d4* 1.0 [30], *DSHARP* 1.0 [33], *miniC2D* 1.0.0 [34], *cnf2eadt* 1.0 [28], *bdd_minisat_all* 1.0.2 [41], and *sdd* 2.0 [14], which are all based on knowledge compilation techniques. We also considered rather recent approximate solvers *ApproxMC2*, *ApproxMC3* [7] and *sts* 1.0 [20], as well as CDCL-based solvers *Cachet* 1.21 [37],

[8] See: tinyurl.com/countingbenchmarks
[9] See: reasoning.cs.ucla.edu/c2d

	solver	0-20	21-30	31-40	41-50	51-60	>60	best	unique	∑	time[h]
preprocessed by pmc [29]	miniC2D	1193	29	**10**	2	1	7	13	0	**1242**	**68.77**
	gpusat2	**1196**	**32**	1	0	0	0	250	2	1229	71.27
	d4	1163	20	**10**	2	**4**	28	52	1	1227	76.86
	countAntom 12	1141	18	**10**	5	**4**	13	101	0	1191	84.39
	c2d	1124	31	**10**	3	3	10	20	0	1181	84.41
	dpdb	1155	19	5	2	0	0	2	1	1181	100.40
	sharpSAT	1029	16	**10**	2	**4**	30	253	1	1091	106.88
	gpusat1	1020	16	0	0	0	0	106	1	1036	114.86
	sdd	1014	4	7	1	0	2	0	0	1028	124.23
	sts	927	4	8	**7**	5	**52**	73	**21**	1003	128.43
	dsharp	853	3	7	2	0	0	83	0	865	157.87
	cnf2eadt	799	3	7	2	0	7	**328**	0	818	170.17
	approxmc 3	794	3	7	2	0	6	10	0	812	173.35
	bdd_minisat_all	791	4	1	0	0	0	99	0	796	175.09
	cachet	624	3	8	2	3	24	3	0	664	209.26
	approxmc 2	447	3	0	0	0	0	1	0	450	265.31
	sharpCDCL	340	3	0	0	0	0	0	0	343	289.17

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 seconds.

sharpCDCL[10], and *sharpSAT* 13.02 [39]. Finally, we also included multi-core solvers *gpusat* 1.0 and *gpusat* 2.0 [23], which both are based on dynamic programming, as well as *countAntom* 1.0 [6] on 12 physical CPU cores, which performed better than on 24 cores. Experiments were conducted with default solver options.

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) 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.

5.2 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 gpusat1, sharpSAT, dsharp, approxmc as well as cachet. Surprisingly, our system shows a different runtime behavior than the other solvers. We believe

[10] See: tools.computational-logic.org

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. Table 1 presents more detailed runtime results, showing a solid sixth place for `dpdb` as our system solves the vast majority of the instances. Assume we only have instances up to an upper bound[11] of treewidth 35. Then, if instances with TDs up to width 35 are considered, `dpdb` solves even slightly more instances than c2d and countAntom.

6 Final Discussion & Conclusions

We presented a generic system `dpdb` for explicitly exploiting treewidth by means of dynamic programming on databases. The idea of `dpdb` is to use database management systems (DBMS) for table manipulation, which makes it (1) easy and elegant to perform *rapid prototyping* for problems, and (2) allows to leverage from decades of database theory and database system tuning. It turned out that all the cases that occur in dynamic programming 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 [23]. In consequence, we carried out preliminary experiments on publicly available instances for #SAT, where we see competitive behavior compared to most recent solvers.

Future Work. Our results give rise to several research questions. First of all, we want to push towards PostgreSQL 12, but at the same time also consider other vendors and systems, e.g., Oracle. In particular, the behavior of different systems might change, when we use different strategies on how to write and evaluate our SQL queries, e.g., sub-queries vs. 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 oftentimes cost too much time to create. 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 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., [8,19]. Another interesting research direction is to study whether efficient data representation techniques on DBMS can be combined with dynamic programming in order to lift our solver to quantified Boolean formulas. Finally, we are also interested in extending this work to projected model counting [21].

References

1. Postgresql documentation 12 (2019), available at: https://www.postgresql.org/docs/12/queries-with.html

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

2. Abseher, M., Musliu, N., Woltran, S.: htd – a free, open-source framework for (customized) tree decompositions and beyond. In: CPAIOR'17. LNCS, vol. 10335, pp. 376–386. Springer Verlag (2017)
3. Bacchus, F., Dalmao, S., Pitassi, T.: Algorithms and complexity results for #SAT and bayesian inference. In: FOCS'03. pp. 340–351. IEEE Computer Soc. (2003)
4. Bannach, M., Berndt, S.: Practical access to dynamic programming on tree decompositions. Algorithms **12**(8), 172 (2019)
5. Bliem, B., Charwat, G., Hecher, M., Woltran, S.: D-flat^2: Subset minimization in dynamic programming on tree decompositions made easy. Fundam. Inform. **147**(1), 27–61 (2016)
6. Burchard, J., Schubert, T., Becker, B.: Laissez-faire caching for parallel #SAT solving. In: SAT'15. LNCS, vol. 9340, pp. 46–61. Springer Verlag (2015)
7. Chakraborty, S., Fremont, D.J., Meel, K.S., Seshia, S.A., Vardi, M.Y.: Distribution-aware sampling and weighted model counting for SAT. In: AAAI'14. pp. 1722–1730. The AAAI Press (2014)
8. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Improving approximate counting for probabilistic inference: From linear to logarithmic sat solver calls. In: IJCAI'16. pp. 3569–3576. The AAAI Press (2016)
9. Charwat, G., Woltran, S.: Expansion-based QBF solving on tree decompositions. Fundam. Inform. **167**(1-2), 59–92 (2019)
10. Choi, A., Van den Broeck, G., Darwiche, A.: Tractable learning for structured probability spaces: A case study in learning preference distributions. In: IJCAI'15. The AAAI Press (2015)
11. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (1970)
12. Cygan, M., Fomin, F.V., Kowalik, Ł., Lokshtanov, D., Dániel Marx, M.P., Pilipczuk, M., Saurabh, S.: Parameterized Algorithms. Springer Verlag (2015)
13. Darwiche, A.: New advances in compiling CNF to decomposable negation normal form. In: ECAI'04. pp. 318–322. IOS Press (2004)
14. Darwiche, A.: SDD: A new canonical representation of propositional knowledge bases. In: IJCAI'11. pp. 819–826. AAAI Press/IJCAI (2011)
15. Dell, H., Komusiewicz, C., Talmon, N., Weller, M.: The PACE 2017 Parameterized Algorithms and Computational Experiments Challenge: The Second Iteration. In: IPEC 2017. Leibniz International Proceedings in Informatics (LIPIcs), vol. 89, pp. 30:1–30:12. Dagstuhl Publishing (2018)
16. Diestel, R.: Graph Theory, 4th Edition, Graduate Texts in Mathematics, vol. 173. Springer Verlag (2012)
17. Domshlak, C., Hoffmann, J.: Probabilistic planning via heuristic forward search and weighted model counting. Journal of Artificial Intelligence Research **30** (2007)
18. Doubilet, P., Rota, G.C., Stanley, R.: On the foundations of combinatorial theory. vi. the idea of generating function. In: Berkeley Symposium on Mathematical Statistics and Probability. pp. 2: 267–318 (1972)
19. Dueñas-Osorio, L., Meel, K.S., Paredes, R., Vardi, M.Y.: Counting-based reliability estimation for power-transmission grids. In: AAAI'17. pp. 4488–4494. The AAAI Press (2017)
20. Ermon, S., Gomes, C.P., Selman, B.: Uniform solution sampling using a constraint solver as an oracle. In: UAI'12. pp. 255–264. AUAI Press (2012)
21. Fichte, J.K., Hecher, M., Morak, M., Woltran, S.: Exploiting treewidth for projected model counting and its limits. In: SAT'18. LNCS, vol. 10929, pp. 165–184. Springer Verlag (2018)

22. Fichte, J.K., Hecher, M., Woltran, S., Zisser, M.: A Benchmark Collection of #SAT Instances and Tree Decompositions (Benchmark Set) (Jun 2018), https://doi.org/10.5281/zenodo.1299752
23. Fichte, J.K., Hecher, M., Zisser, M.: An improved gpu-based SAT model counter. In: CP. Lecture Notes in Computer Science, vol. 11802, pp. 491–509. Springer (2019)
24. Gomes, C.P., Sabharwal, A., Selman, B.: Chapter 20: Model counting. In: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 633–654. IOS Press (2009)
25. Kiljan, K., Pilipczuk, M.: Experimental evaluation of parameterized algorithms for feedback vertex set. In: SEA. LIPIcs, vol. 103, pp. 12:1–12:12. Schloss Dagstuhl (2018)
26. Kleine Büning, H., Lettman, T.: Propositional logic: deduction and algorithms. Cambridge University Press, Cambridge (1999)
27. Kloks, T.: Treewidth. Computations and Approximations, LNCS, vol. 842. Springer Verlag (1994)
28. Koriche, F., Lagniez, J.M., Marquis, P., Thomas, S.: Knowledge compilation for model counting: Affine decision trees. In: IJCAI'13. The AAAI Press (2013)
29. Lagniez, J., Marquis, P.: Preprocessing for propositional model counting. In: AAAI. pp. 2688–2694. AAAI Press (2014)
30. Lagniez, J.M., Marquis, P.: An improved decision-DDNF compiler. In: IJCAI'17. pp. 667–673. The AAAI Press (2017)
31. Langer, A., Reidl, F., Rossmanith, P., Sikdar, S.: Evaluation of an MSO-solver. In: Proc. ALENEX. pp. 55–63. SIAM / Omnipress (2012)
32. Liu, J., Zhong, W., Jiao, L.: Comments on "the 1993 DIMACS graph coloring challenge" and "energy function-based approaches to graph coloring". IEEE Trans. Neural Networks **17**(2), 533 (2006)
33. Muise, Christian J .and McIlraith, S.A., Beck, J.C., Hsu, E.I.: Dsharp: Fast d-DNNF compilation with sharpSAT. In: AI'17. LNCS, vol. 7310, pp. 356–361. Springer Verlag (2012)
34. Oztok, U., Darwiche, A.: A top-down compiler for sentential decision diagrams. In: IJCAI'15. pp. 3141–3148. The AAAI Press (2015)
35. Roth, D.: On the hardness of approximate reasoning. Artif. Intell. **82**(1-2), 273–302 (1996)
36. Samer, M., Szeider, S.: Algorithms for propositional model counting. Journal of Discrete Algorithms **8**(1), 50—64 (2010)
37. Sang, T., Bacchus, F., Beame, P., Kautz, H., Pitassi, T.: Combining component caching and clause learning for effective model counting. In: SAT'04 (2004)
38. Sang, T., Beame, P., Kautz, H.: Performing bayesian inference by weighted model counting. In: AAAI'05. The AAAI Press (2005)
39. Thurley, M.: sharpSAT – counting models with advanced component caching and implicit BCP. In: SAT'06. pp. 424–429. Springer Verlag (2006)
40. Toda, S.: PP is as hard as the polynomial-time hierarchy. SIAM J. Comput. **20**(5), 865–877 (1991)
41. Toda, T., Soh, T.: Implementing efficient all solutions SAT solvers. ACM Journal of Experimental Algorithmics **21**, 1.12 (2015), special Issue SEA 2014
42. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, Volume II. Computer Science Press (1989)
43. Xue, Y., Choi, A., Darwiche, A.: Basing decisions on sentences in decision diagrams. In: AAAI'12. The AAAI Press (2012)