# 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][TODO0000−0003−0131−6771], and Stefan Woltran[2][TODO0000−0003−0131−6771]

[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 Management Systems · SQL · Relational Algebra · Counting · Model Counting

## 1 Introduction

Counting solutions is a well-known task in mathematics, computer science, and other areas [9,17,29,43]. 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

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

real-world applications [9,11,20,48]. Such problems are typically considered rather hard, since #Sat is complete for the class #P [3,40], i.e., one can simulate any problem of the polynomial hierarchy with polynomially many calls [45] to a #Sat solver. However, recently it was shown that the better part [26] of the publicly available #Sat instances, techniques from parameterized complexity [28,19,38,13] might come to the rescue. In particular, after using regular preprocessors, e.g., pmc [34], in this area, more than 80% of these practically relevant instances (graph representations thereof) have reasonably small treewidth, where small treewidth is one of the most cited[4] combinatorial invariants. To be more concrete, the measure treewidth is a structural parameter of graphs, which models the closeness of the graph to being a tree. This, gives rise to a general framework for counting problems that leverage treewidth. The general idea to develop such frameworks is indeed not new, since there are both, specialized solvers [10,26,30], as well as general systems like D-FLAT [5], Jatatosk [4], and sequoia [36], that exploit treewidth (to name a few). 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. However, for solving problems, most of the general systems [4,36] require a descriptive model of the problem, where an abstract view or for example certain logics are used internally, but problems in these systems are not directly described by means of dynamic programming algorithms. In this work, we solve (counting) problems by means of DP, where the algorithm is specified by giving essential parts of the DP algorithm in form of SQL `SELECT` queries. The whole DP algorithm is done by our system `dpdb`, which employs *database management systems (DBMS)* [47] for solving. This has not only the advantage of naturally describing and manipulating the tables that are obtained during DP, but also allows us to leverage from decades of database technology in form of the capability to deal with huge tables using limited amount of main memory (RAM), dedicated implementations of 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 efficiently 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

[4] On October 17, 2019, Google Scholar shows 19,200 results on treewidth.

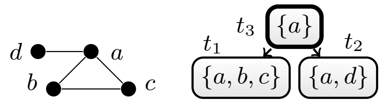


Figure 1: Graph $G$ (left) with a TD $\mathcal{T}$ of graph $G$ (right).

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 [6,16].

*Boolean Satisfiability.* We define Boolean formulas and their evaluation in the usual way, cf., [29,31]. A literal is a Boolean variable $x$ or its negation $\neg x$. A *CNF formula* $\varphi$ is a set of *clauses*, interpreted as conjunction, which are sets 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 $\varphi$ 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 $\varphi$ that contains a literal set to 1 by $I$, and removing from every remaining clause of $\varphi$ 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)* of a given graph $G$ is a pair $\mathcal{T} = (T, \chi)$ where $T$ is a rooted tree and $\chi$ 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 $\mathrm{width}(\mathcal{T}) := \max_{t\in V(T)} |\chi(t)| - 1$. The *treewidth* $\mathrm{tw}(G)$ of $G$ is the minimum $\mathrm{width}(\mathcal{T})$ over all TDs $\mathcal{T}$ of $G$. For a node $t \in V(T)$, we say that $\mathrm{type}(t)$ 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 $t \in N$, $\mathrm{type}(t) \in \{\mathit{leaf}, \mathit{join}, \mathit{intr}, \mathit{rem}\}$, 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 [32] a complete graph with 3 vertices. ■

*Relational Algebra.* We use relational algebra [12] for manipulation of relations, which forms the theoretical basis of its the well-known implementation database standard *Structured Query Language (SQL)* [47] on tables. An *attribute* $a$ is of a certain finite *domain* $\mathrm{dom}(a)$. Then, a *tuple* $r$ over set $\mathrm{att}(r)$ of attributes, is a set of pairs of the form $(a, v)$ with $a \in \mathrm{att}(r), v \in \mathrm{dom}(a)$ s.t. for each $a \in \mathrm{att}(r)$, there is exactly one $v \in \mathrm{dom}(a)$ 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 value of domain $\text{dom}(a)$. 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 $m : \text{att}(R) \to A$ s.t. $\text{dom}(a) = \text{dom}(m(a))$ for $a \in \text{att}(R)$, by $\rho_m(R) := \{(m(a), v) \mid (a, v) \in R\}$. *Selection* of rows in $R$ according to a given Boolean formula $\varphi$ with equality[5] is defined by $\sigma_\varphi(R) := \{r \mid r \in R, \varphi(r^E) = \emptyset\}$, where $r^E$ is a truth assignment over $\text{var}(\varphi)$ such that for each $v, v', v'' \in \text{dom}(R) \cup \text{att}(R)$ (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 $\text{att}(R') \cap \text{att}(R) = \emptyset$. Then, we refer to the *cross-join* by $R \times R' := \{r \cup r' \mid r \in R, r' \in R'\}$. Further, wet let *$\theta$-join* correspond 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* $\tau_t$. This is achieved by running a so-called *table algorithm* $\mathsf{A}$, which is designed for a certain graph representation, and stores in $\tau_t$ results of problem parts of $\mathcal{I}$, thereby considering tables $\tau_{t'}$ for child nodes $t'$ of $t$. The DP approach 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 $\mathsf{A}$ 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 $\tau_t$.
4. Interpret table $\tau_n$ for the root $n$ of $T$ in order to output the solution of $\mathcal{I}$.

For solving problem $\mathcal{P} = \#\textsc{Sat}$, we need the following graph representation. The *primal graph* $G_\varphi$ [41] of a formula $\varphi$ has as vertices its variables, where two variables are joined by an edge if they occur together in a clause of $\varphi$.

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

**Listing 2:** Table algorithm $\mathsf{S}(t, \chi(t), \varphi_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for #SAT [41] using nice TD.

**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** $\tau_t := \{\langle \emptyset, 1 \rangle\}$
2 **else if** type$(t) =$ *intr, and* $a \in \chi(t)$ *is introduced* **then**
3 | $\tau_t := \{\langle J, c \rangle \mid \langle I, c \rangle \in \tau_1, J \in \{I^+_{a \mapsto 0}, I^+_{a \mapsto 1}\}, \varphi_t(J) = \emptyset\}$
4 **else if** type$(t) =$ *rem, and* $a \notin \chi(t)$ *is removed* **then**
5 | $\tau_t := \{\langle I^-_a, \Sigma_{\langle J, c \rangle \in \tau_1 : I^-_a = J^-_a} c \rangle \mid \langle I, \cdot \rangle \in \tau_1\}$
6 **else if** type$(t) =$ *join* **then**
7 | $\tau_t := \{\langle I, c_1 \cdot c_2 \rangle \mid \langle I, c_1 \rangle \in \tau_1, \langle I, c_2 \rangle \in \tau_2\}$

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

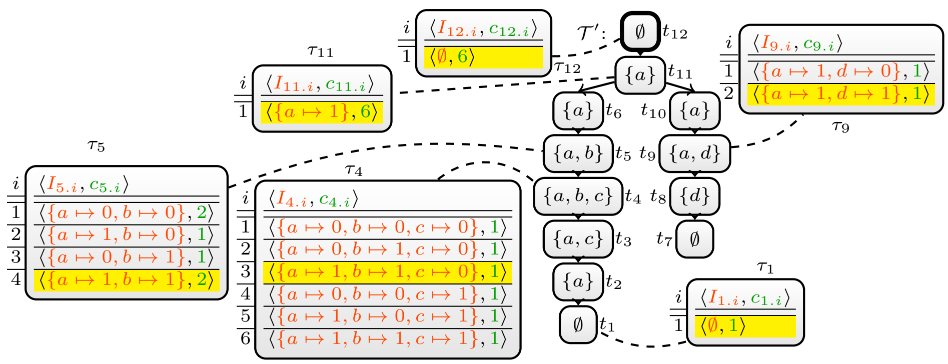


Figure 2: Selected tables obtained by DP on $\mathcal{T}'$ for $\varphi$ of Example 2 using algorithm $\mathsf{S}$.

Given formula $\varphi$, a TD $\mathcal{T} = (T, \chi)$ of $G_\varphi$ and a node $t$ of $T$. 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 $\mathsf{S}$ as presented in Listing 2 shows all the cases that are needed to solve #SAT by means of DP of nice TDs. Each table $\tau_t$ consist of rows of the form $\langle I, c \rangle$, where $I$ is an assignment of $\varphi_t$ 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 $\beta$ of the child table, whether $a$ is set to true or to false, and ensure that $\varphi_t$ is satisfied. When an atom $a$ is removed in node $t$, we project assignments of child tables to $\chi(t)$, cf., Line 5, and counters of the same assignments are summed up. For join nodes $t$, counters of common assignments in the child tables are multiplied as in 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 $\varphi$ 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 $\varphi$. Observe that graph $G$ of Figure 1 actually depicts the primal graph $G_\varphi$ of $\varphi$. Intuitively, $\mathcal{T}$ of Figure 1 allows to evaluate formula $\varphi$ in parts. Figure 2 illustrates a nice TD $\mathcal{T}' = (\cdot, \chi)$ of the primal graph $G_\varphi$ and tables $\tau_1, \ldots, \tau_{12}$ that are obtained during the execution of $\mathsf{S}$ on nodes $t_1, \ldots, t_{12}$.

**Listing 3:** Alternative table algorithm $\mathsf{S}_{\mathsf{RAlg}}(t, \chi(t), \varphi_t, \langle \tau_1, \dots, \tau_\ell \rangle)$ for #Sat.

**In:** Node $t$, bag $\chi(t)$, clauses $\varphi_t$, sequence $\langle \tau_1, \dots \tau_\ell \rangle$ of child tables. **Out:** Table $\tau_t$.

1 **if** type$(t)$ = *leaf* **then** $\tau_t := \{\{(\mathrm{cnt}, 1)\}\}$
2 **else if** type$(t)$ = *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** type$(t)$ = *rem, and* $a \notin \chi(t)$ *is removed* **then**
5 $\quad \tau_t := {}_{\chi(t)}G_{\mathrm{cnt} \leftarrow \mathtt{SUM(cnt)}}(\Pi_{\mathrm{att}(\tau_1) \setminus \{[\![a]\!]\}} \tau_1)$
6 **else if** type$(t)$ = *join* **then**
7 $\quad \tau_t := \dot{\Pi}_{\chi(t), \{\mathrm{cnt} \leftarrow \mathrm{cnt} \cdot \mathrm{cnt}'\}}(\tau_1 \bowtie_{\bigwedge_{a \in \chi(t)} [\![a]\!] \approx [\![a]\!]'} \rho_{\bigcup_{a \in \mathrm{att}(\tau_2)} \{[\![a]\!] \mapsto [\![a]\!]'\}} \tau_2)$

We assume that each row in a table $\tau_t$ is identified by a number, i.e., row $i$ corresponds to $\boldsymbol{u_{t.i}} = \langle I_{t.i}, c_{t.i} \rangle$.

Table $\tau_1 = \{\, \langle \emptyset, 1 \rangle \,\}$ as type$(t_1)$ = *leaf*. Since type$(t_2)$ = *intr*, we construct table $\tau_2$ from $\tau_1$ 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 $\tau_4$, we have $\{c_1, c_2\}(I_{4.i}) = \emptyset$ since $\mathsf{S}$ enforces satisfiability of $\varphi_t$ in node $t$. Since type$(t_5)$ = *rem*, we remove variable $c$ from all elements in $\tau_4$ and sum up counters accordingly to construct $\tau_5$. 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$(t_{11})$ = *join*, we build table $\tau_{11}$ by taking the intersection of $\tau_6$ and $\tau_{10}$. 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 $\varphi$ using highlighted (yellow) rows in Figure 2. On the other hand, if $\varphi$ was unsatisfiable, $\tau_{12}$ would be empty ($\emptyset$). ■

*Alternative: Relational Algebra.* Instead of using set theory to describe how tables are obtained during dynamic programming are performed, one could alternatively use relational algebra. There, tables $\tau_t$ for each TD node $t$ are pictured as relations, where $\tau_t$ 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 3 presents a table algorithm for problem #Sat that is equivalent to Listing 2, 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 2). In particular, one typically derives for nodes $t$ with type$(t)$ = *leaf*, a fresh initial table $\tau_t$, cf., Line 1 of Listing 3. Then, whenever an atom $a$ is introduced, such algorithms often use $\theta$-joins with a fresh initial table for the introduced variable $a$ that

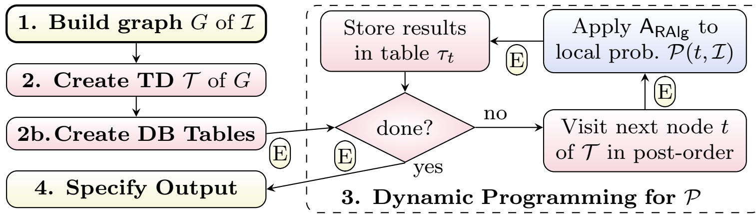


Figure 3: Architecture of Dynamic Programming with Databases. Steps highlighted in red are provided by the system depending on specification of yellow and blue parts, which is given by the user for specific problems $\mathcal{P}$. The yellow "E"s represent events that can be intercepted and handled by 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.

represent potential values $a$ can have. In Line 3 the selection of the $\theta$-join is performed by ensuring $\varphi_t$, corresponding to the local problem of #SAT. Further, for nodes $t$ with type$(t) = \mathit{rem}$, these table algorithms oftentimes need projection. In case of Listing 3, Line 5 also needs grouping in order to maintain the counter, as several rows of $\tau_1$ might collapse in $\tau_t$. Finally, for a node $t$ with type$(t) = \mathit{join}$, in Line 7 we use again $\theta$-joins (and extended projection for maintaining counters), which allows us later to leverage database technology of the last decades.

## 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 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 $\tau_t$ for each node $t$ of $T$, where the corresponding table schema of $\tau_t$ is specified in the blue part, i.e., within $\mathsf{A}_{\mathsf{RAlg}}$. The *default schema* of such a table $\tau_t$ 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 code[6] for each node $t$ thereby oftentimes depending on $\chi(t)$. This generated SQL code is then used internally for manipulation of tables $\tau_t$ during the tree decomposition traversal in Step 3 of dynamic programming. Listing 4 presents a general template, where parts of table algorithms for problems that are typically

[6] Recall that SQL is a specific implementation standard (set) of relational algebra.

**Listing 4:** 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 $\tau_t$.

**if** type$(t)$ = *leaf* **then** $\tau_t := \#\varepsilon\mathsf{Tab}\#$

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

  $\tau_t := \tau_1 \bowtie_{\#\mathsf{localProbFilter}\#} \#\mathsf{intrTab}\#$

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

  $\tau_t := {}_{\chi(t)}G_{\#\mathsf{aggrExp}\#}(\Pi_{\mathrm{att}(\tau_1)\setminus\{[\![a]\!]\}}\tau_1)$

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

  $\tau_t := \dot{\Pi}_{\chi(t),\#\mathsf{extProj}\#}(\tau_1 \bowtie_{\bigwedge_{a\in\chi(t)}[\![a]\!]\approx[\![a]\!]'} \rho_{\bigcup_{a\in\mathrm{att}(\tau_2)}\{[\![a]\!]\mapsto[\![a]\!]'\}}\tau_2)$

problem-specific are replaced by colored placeholders of the form $\#\mathsf{placeHolder}\#$, cf., Listing 3. 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 Listings 4, 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 for efficiency it is crucial to implement non-nice TDs.

### 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[7], written in Python 3 and uses 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 [26] due to heuristics. Note that dpdb directly supports the TD format of recent competitions, i.e., one could easily replace the TD library. It is crucial 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 $\theta$-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, $\theta$-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 it is not wise though to consider $\theta$-joins which do not take into account just two tables, since this already fixes in which order these tables shall be combined, thereby again limiting the query optimizer. Apart from

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

this trade-off, we tried to outsource the task of joining tables to the DBMS as much as possible, since the performance of database systems highly depend on query optimization. This 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 $\theta$-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 4 automatically. Concretely, the system executes one query per table $\tau_t$ for each node $t$ during the TD traversal. 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 $\tau_t$ consisting of the $\theta$-join as in Line 7 of Listing 4, 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) *subqueries (nested queries)*, which both result in one nested SQL query per table $\tau_t$. On top of these row candidates, projection[8] and grouping involving #aggrExp# as in Line 5 of Listing 4, as well as selection acording to #localProbFilter#, cf., Line 3, is specified. It turns out that PostgreSQL can do better with subqueries, where the query optimizer oftentimes pushes selection and projection into the subquery 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 $\varphi$ of TD $\mathcal{T}$ and Option (3): subqueries, where the row candidates are expressed via a subqueries. Then, for each node $t_i$ of $\mathcal{T}$, we dpdb generates a view $vi$ as well as a table $\tau_i$ containing in the end the content of $vi$. Observe that each view only has one column $[\![a]\!]$ for each variable $a$ of $\varphi$ since the truth assignment of the other variables are not needed later. Actually, in dpdb the additional columns are kept (empty, null) for readability. This keeps the tables compact, only $\tau_1$ has two rows, $\tau_2$, and $\tau_3$ 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)
```

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

```
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 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 termination of connections, respectively.

*Logging, Debugging and Extensions.* Currently, we currently 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 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 $\tau_t$, 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 $\tau_t$. Together with the power and flexibility of SQL queries, we observed that this really 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 if one repeats the computation with a different random seed, we obtain different approximate results.

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 CNF [], DIMACS GRAPH [], Edge [] as well as the format for TDs that has been used in recent competitions [].

mh: refs!

### 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 4, cf., Listing 3.

*Problem* #SAT. Specific parts for #SAT for node $t$ with child nodes $t_1, \dots, t_\ell$ and $\varphi_t = \{\{l_{1,1}, \dots, l_{1,k_1}\}, \dots, \{l_{n,1}, \dots, l_{n,k_n}\}\}$.

- #$\varepsilon$Tab#: `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#: $\tau_1$`.cnt` $\cdot \ldots \cdot$ $\tau_\ell$`.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 $\varphi$, #epsilonTab# returns the empty table and parts #aggrExp#, #extProj# are just empty since there is no counter needed.

*Problem* #$o$-COL. Given a graph $G = (V, E)$, a *$o$-coloring* is a mapping $\iota : V \rightarrow \{1, \dots, o\}$ 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 child nodes $t_1, \dots, t_\ell$ are as follows, where $E(G_t) = \{\{u_1, v_1\}, \dots, \{u_n, v_n\}\}$.

- #$\varepsilon$Tab#: `SELECT 1 AS cnt`
- #intrTab#: `SELECT 1 AS val UNION ALL ... UNION ALL` $o$
- #localProbFilter#: `NOT` ($[\![u_1]\!] = [\![v_1]\!]$) `AND ... AND NOT` ($[\![u_n]\!] = [\![v_n]\!]$)
- #aggrExp#: `SUM(cnt) AS cnt`
- #extProj#: $\tau_1$`.cnt` $\cdot \ldots \cdot$ $\tau_\ell$`.cnt AS cnt`

*Problem* MINVC. Given a graph $G = (V, E)$, 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 cover $C$, 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. To this end, we use an additional column `card` for storing cardinalities.

Problem MINVC for node $t$ with child nodes $t_1, \dots, t_\ell$, where $E(G_t) = \{\{u_1, v_1\}, \dots, \{u_n, v_n\}\}$ can be specified as follows.

- #$\varepsilon$Tab#: `SELECT 0 AS card`
- #intrTab#: `SELECT 1 AS val UNION ALL 0`
- #localProbFilter#: ($[\![u_1]\!]$ `OR` $[\![v_1]\!]$) `AND ... AND` ($[\![u_n]\!]$ `OR` $[\![v_n]\!]$)
- #aggrExp#: `MIN(card) AS card`
- #extProj#:

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 [23] are publicly available, and our results are also on github at github.com/hmarkus/dp_on_dbs/padl2020.

### 5.1 Setup

*Measure & Resources.* We mainly compare wall clock time and number of timeouts. In the time we include, if applicable, *preprocessing time* as well as *decomposition time* for computing a TD with a fixed random seed. For parallel CPU solvers we allow access to 24 physical cores on machines, where hyperthreading was disabled. 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[9](1480 instances), and `c2d` benchmarks[10] (14 instances). However, we used instances after being preprocessed by pmc [34], similar to a recent work on #SAT [26] There, more than 80% of the #SAT instances have primal treewidth below 19 after preprocessing. Before preprocessing

*Benchmarked Solvers.* In our experimental work, we present results for the most recent versions of publicly available #SAT solvers, namely, *c2d* 2.20 [14], *d4* 1.0 [35], *DSHARP* 1.0 [37], *miniC2D* 1.0.0 [39], *cnf2eadt* 1.0 [33], *bdd_minisat_all* 1.0.2 [46], and *sdd* 2.0 [15] (based on knowledge compilation techniques). We also considered rather recent approximate solvers *ApproxMC2*, *ApproxMC3* [8] and *sts* 1.0 [21], as well as CDCL-based solvers *Cachet* 1.21 [42], *sharpCDCL*[11], and *sharpSAT* 13.02 [44]. Finally, we also included multi-core solvers *gpusat* 1.0 and *gpusat* 2.0 [26], as well as *countAntom* 1.0 [7] on 12 physical CPU cores, which performed better than on 24 cores. We considered also additional solvers, e.g., *d-DNNF reasoner* 0.4.180625 on top of d4 as underlying knowledge compiler, where detailed results can be found online. All 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 and 256 GB RAM. The results were gathered on Ubuntu 16.04.1 LTS machines with disabled hyperthreading on kernel 4.4.0-139. 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

[9] See: tinyurl.com/countingbenchmarks
[10] See: reasoning.cs.ucla.edu/c2d
[11] See: tools.computational-logic.org

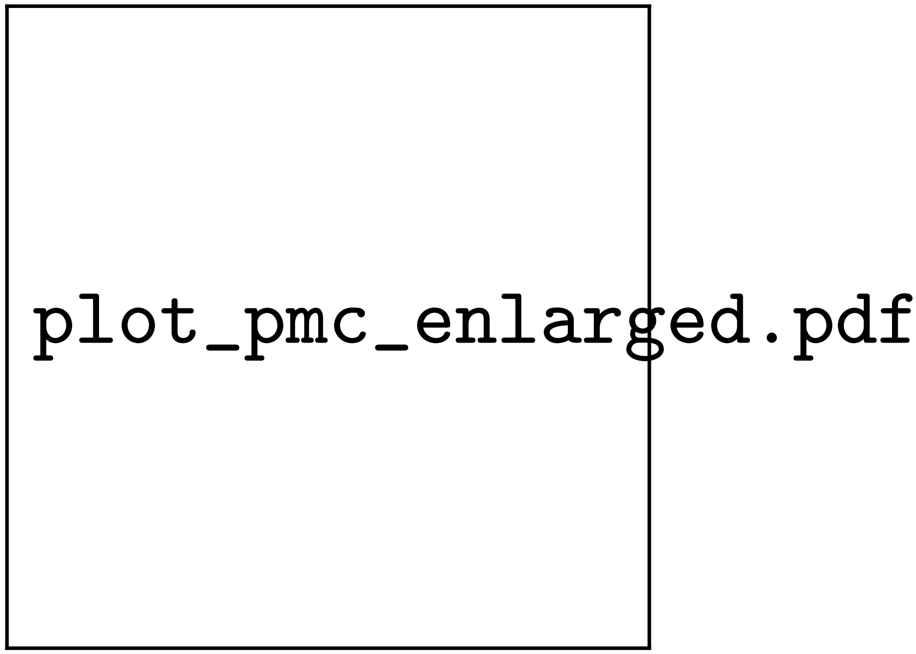


Figure 4: Runtime for the top 5 sequential and all parallel solvers over all the #SAT instances with pmc preprocessor. The x-axis refers to the number of instances and the y-axis depicts the runtime sorted in ascending order for each solver individually.

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 five sequential solvers, and all parallel counting solvers with preprocessor pmc in a cactus-like plot. Table 1 presents detailed runtime results for #SAT with preprocessors pmc, B+E, and without preprocessing, respectively. Since the solver sts produced results that varied from the correct result on average more than the value of the correct result, we excluded it from the presented results. If we disallow preprocessing, gpusat2 and gpusat1 perform only slightly better in the overall standing of the solvers. But gpusat2 solves 42 instances more and requires about 10 hours less of wallclock time. Further, we can observe, that the variant gpusat2($A$+$B$) performs particular well, mainly since for instances below width 30, the BST compression seems relatively expensive compared to the array data structure. Interestingly, when considering the results on preprocessing in Table 1 (top, mid) and Figure 4 we observe that the architectural improvements pay off quite well. gpusat2 can solve the vast majority of the instances and ranks second place. If one uses the B+E preprocessor shown in Table 1 (mid), gpusat2 solves even more instances as well as the other solvers. Still, it ranks fifth solving

| | solver | 0-20 | 21-30 | 31-40 | 41-50 | 51-60 | >60 | best | unique | ∑ | time[h] |
|---|---|---|---|---|---|---|---|---|---|---|---|
| pmc preprocessing | miniC2D | 1193 | 29 | **10** | 2 | 1 | 7 | 13 | 0 | **1242** | **68.77** |
| | gpusat2 | **1196** | 32 | 1 | 0 | 0 | 0 | 250 | **8** | 1229 | 71.27 |
| | d4 | 1163 | 20 | **10** | 2 | **4** | 28 | 52 | 1 | 1227 | 76.86 |
| | gpusat2($A$+$B$) | 1187 | 18 | 1 | 0 | 0 | 0 | 120 | 7 | 1206 | 74.56 |
| | 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 |
| | sharpSAT | 1029 | 16 | **10** | 2 | **4** | **30** | **253** | 1 | 1091 | 106.88 |
| | gpusat1 | 1020 | 16 | 0 | 0 | 0 | 0 | 106 | 7 | 1036 | 114.86 |
| | sdd | 1014 | 4 | 7 | 1 | 0 | 2 | 0 | 0 | 1028 | 124.23 |
| | solver | 0-20 | 21-30 | 31-40 | 41-50 | 51-60 | >60 | best | unique | ∑ | time[h] |
| B+E preprocessing | c2d | 1199 | 24 | **9** | 0 | 2 | 23 | 14 | 0 | **1257** | **63.46** |
| | miniC2D | 1203 | **27** | 8 | 0 | 2 | 12 | 8 | 0 | 1252 | 64.92 |
| | d4 | 1182 | 15 | **9** | **1** | **3** | 31 | 79 | 1 | 1241 | 69.32 |
| | countAntom 12 | 1177 | 14 | 8 | 0 | 2 | **34** | 100 | 0 | 1235 | 69.79 |
| | gpusat2 | **1204** | 26 | 1 | 0 | 0 | 0 | **150** | **3** | 1231 | 68.15 |
| | gpusat2($A$+$B$) | 1201 | 21 | 1 | 0 | 0 | 0 | 67 | **3** | 1223 | 70.39 |
| | sdd | 1106 | 11 | 4 | **1** | 1 | 4 | 0 | 0 | 1127 | 100.48 |
| | gpusat1 | 1037 | 16 | 0 | 0 | 0 | 0 | 87 | **3** | 1053 | 110.87 |
| | bdd_minisat_all | 926 | 6 | 3 | **1** | 1 | 0 | 101 | 0 | 937 | 140.59 |
| | solver | 0-20 | 21-30 | 31-40 | 41-50 | 51-60 | >60 | best | unique | ∑ | time[h] |
| without preprocessing | countAntom 12 | 118 | 511 | 139 | **175** | **21** | **181** | 318 | 15 | **1145** | **96.64** |
| | d4 | 124 | 514 | 148 | 162 | **21** | 168 | 69 | 15 | 1137 | 104.94 |
| | c2d | 119 | 525 | **165** | 161 | 18 | 120 | 48 | 15 | 1108 | 110.53 |
| | miniC2D | 122 | 514 | 128 | 149 | 9 | 62 | 0 | 0 | 984 | 141.22 |
| | sharpSAT | 100 | 467 | 124 | 156 | 12 | 123 | **390** | 4 | 982 | 135.41 |
| | gpusat2($A$+$B$) | **125** | **539** | 96 | 138 | 0 | 0 | 94 | **19** | 898 | 151.16 |
| | gpusat2 | **125** | 523 | 96 | 138 | 0 | 0 | 78 | 17 | 882 | 155.43 |
| | gpusat1 | **125** | 524 | 67 | 140 | 0 | 0 | 82 | 9 | 856 | 162.03 |
| | cachet | 99 | 430 | 71 | 152 | 8 | 57 | 3 | 0 | 817 | 176.26 |
| | solver | 0-20 | 21-30 | 31-40 | 41-50 | 51-60 | >60 | best | unique | ∑ | time[h] |

Table 1: Number of #SAT instances (grouped by treewidth upper bound intervals) solved by sum of the top five sequential and all parallel counting solvers with preprocessor pmc (top), B+E (mid), and without preprocessing (bottom). time[h] is the cumulated wall clock time in hours, where unsolved instances are counted as 900 seconds.

only 26 instances less than the best solver and 10 less than the third best solver and solves the most instances having width below 30.

### 5.3 Discussion

- approximate memory used by db for one of the largest problems solved
- Discussion section: we don't create/use any indices... Meaningful/useful B*Tree indices hard to create. Exploration of Bitmap indices (Oracle Enterprise feature) would be interesting.

## 6 Conclusion

We presented an improved OpenCL-based solver gpusat2 for solving #SAT and WMC. Compared to the weighted model counter gpusat1 that uses the

GPU, our solver gpusat2 implements adapted memory management, specialized data structures on the GPU, improved data type precision handling, and an initial approach to use customized TDs. We carried out rigorous experimental work, including establishing upper bounds for treewidth after preprocessing of commonly used benchmarks and comparing to most recent solvers.

*Future Work.* Our results give rise to several research questions. Since established preprocessors are mainly suited for #SAT, we are interested in additional preprocessing methods for weighted model counting (WMC) that reduce the treewidth or at least allow us to compute TDs of smaller width. It would also be interesting whether GPU-based techniques can successfully be used within knowledge compilation-based model counters. An interesting research direction is to study whether efficient data representation techniques can be combined with dynamic programming in order to lift our solver to counting in WCSP [27]. Further, we are also interested in extending this work to projected model counting [22,24,25].

## References


1. Postgresql documentation 9.5 (2019), available at: https://www.postgresql.org/docs/9.5/queries-with.html
2. Abseher, M., Musliu, N., Woltran, S.: htd – a free, open-source framework for (customized) tree decompositions and beyond. In: Salvagnin, D., Lombardi, M. (eds.) Proceedings of the 14th International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming (CPAIOR'17). Lecture Notes in Computer Science, vol. 10335, pp. 376–386. Springer Verlag, Padova, Italy (Jun 2017)
3. Bacchus, F., Dalmao, S., Pitassi, T.: Algorithms and complexity results for #SAT and bayesian inference. In: Chekuri, C.S., Micciancio, D. (eds.) Proceedings of the 44th IEEE Symposium on Foundations of Computer Science (FOCS'03). pp. 340–351. IEEE Computer Soc., Cambridge, MA, USA (Oct 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. Bondy, J.A., Murty, U.S.R.: Graph theory, Graduate Texts in Mathematics, vol. 244. Springer Verlag, New York, USA (2008)
7. Burchard, J., Schubert, T., Becker, B.: Laissez-faire caching for parallel #SAT solving. In: Heule, M., Weaver, S. (eds.) Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT'15). Lecture Notes in Computer Science, vol. 9340, pp. 46–61. Springer Verlag, Austin, TX, USA (2015)
8. Chakraborty, S., Fremont, D.J., Meel, K.S., Seshia, S.A., Vardi, M.Y.: Distribution-aware sampling and weighted model counting for SAT. In: Brodley, C.E., Stone, P. (eds.) Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI'14). pp. 1722–1730. The AAAI Press, Québec City, QC, Canada (2014)

9. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Improving approximate counting for probabilistic inference: From linear to logarithmic sat solver calls. In: Kambhampati, S. (ed.) Proceedings of 25th International Joint Conference on Artificial Intelligence (IJCAI'16). pp. 3569–3576. The AAAI Press, New York City, NY, USA (Jul 2016), https://bitbucket.org/kuldeepmeel/approxmc
10. Charwat, G., Woltran, S.: Expansion-based QBF solving on tree decompositions. In: RCRA@AI*IA. CEUR Workshop Proceedings, vol. 2011, pp. 16–26. CEUR-WS.org (2017)
11. Choi, A., Van den Broeck, G., Darwiche, A.: Tractable learning for structured probability spaces: A case study in learning preference distributions. In: Yang, Q. (ed.) Proceedings of 24th International Joint Conference on Artificial Intelligence (IJCAI'15). The AAAI Press (2015)
12. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (1970)
13. Cygan, M., Fomin, F.V., Kowalik, Ł., Lokshtanov, D., Dániel Marx, M.P., Pilipczuk, M., Saurabh, S.: Parameterized Algorithms. Springer Verlag (2015), 10.1007/978-3-319-21275-3
14. Darwiche, A.: New advances in compiling CNF to decomposable negation normal form. In: López De Mántaras, R., Saitta, L. (eds.) Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04). pp. 318–322. IOS Press, Valencia, Spain (2004)
15. Darwiche, A.: SDD: A new canonical representation of propositional knowledge bases. In: Walsh, T. (ed.) Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11). pp. 819–826. AAAI Press/IJCAI, Barcelona, Catalonia, Spain (Jul 2011)
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: Proceedings of the 6th Berkeley Symposium on Mathematical Statistics and Probability. pp. 2: 267–318 (1972)
19. Downey, R.G., Fellows, M.R.: Fundamentals of Parameterized Complexity. Texts in Computer Science, Springer Verlag, London, UK (2013), 10.1007/978-1-4471-5559-1
20. Dueñas-Osorio, L., Meel, K.S., Paredes, R., Vardi, M.Y.: Counting-based reliability estimation for power-transmission grids. In: Singh, S.P., Markovitch, S. (eds.) Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI'17). pp. 4488–4494. The AAAI Press, San Francisco, CA, USA (Feb 2017)
21. Ermon, S., Gomes, C.P., Selman, B.: Uniform solution sampling using a constraint solver as an oracle. In: de Freitas, N., Murphy, K. (eds.) Proceedings of the 28th Conference on Uncertainty in Artificial Intelligence (UAI'12). pp. 255–264. AUAI Press, Catalina Island, CA, USA (Aug 2012)
22. Fichte, J.K., Hecher, M., Morak, M., Woltran, S.: Exploiting treewidth for projected model counting and its limits. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) Proceedings on the 21th International Conference on Theory and Applications of Satisfiability Testing (SAT'18). Lecture Notes in Computer Science, vol. 10929, pp. 165–184. Springer Verlag, Oxford, UK (Jul 2018)
23. 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

24. Fichte, J.K., Hecher, M.: Treewidth and counting projected answer sets. In: Proceedings of the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'19). Lecture Notes in Computer Science, vol. 11481, pp. 105–119. Springer (2019)
25. Fichte, J.K., Hecher, M., Meier, A.: Counting complexity for reasoning in abstract argumentation. In: Hentenryck, P.V., Zhou, Z.H. (eds.) Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAA'19). Honolulu, Hawaii, USA (2018), to appear. Author's archived copy available at https://arxiv.org/abs/1811.11501
26. 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)
27. Fioretto, F., Pontelli, E., Yeoh, W., Dechter, R.: Accelerating exact and approximate inference for (distributed) discrete optimization with GPUs. Constraints **23**(1), 1–23 (Jan 2018)
28. Flum, J., Grohe, M.: Parameterized Complexity Theory, Theoretical Computer Science, vol. XIV. Springer Verlag, Berlin (2006), 10.1007/3-540-29953-X
29. Gomes, C.P., Sabharwal, A., Selman, B.: Chapter 20: Model counting. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 633–654. IOS Press, Amsterdam, Netherlands (Feb 2009)
30. 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 - Leibniz-Zentrum fuer Informatik (2018)
31. Kleine Büning, H., Lettman, T.: Propositional logic: deduction and algorithms. Cambridge University Press, Cambridge, New York, NY, USA (1999)
32. Kloks, T.: Treewidth. Computations and Approximations, Lecture Notes in Computer Science, vol. 842. Springer Verlag (1994)
33. Koriche, F., Lagniez, J.M., Marquis, P., Thomas, S.: Knowledge compilation for model counting: Affine decision trees. In: Rossi, F., Thrun, S. (eds.) Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI'13). The AAAI Press, Beijing, China (Aug 2013)
34. Lagniez, J., Marquis, P.: Preprocessing for propositional model counting. In: AAAI. pp. 2688–2694. AAAI Press (2014)
35. Lagniez, J.M., Marquis, P.: An improved decision-DDNF compiler. In: Sierra, C. (ed.) Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17). pp. 667–673. The AAAI Press, Melbourne, VIC, Australia (2017)
36. Langer, A., Reidl, F., Rossmanith, P., Sikdar, S.: Evaluation of an MSO-solver. In: Proc. ALENEX. pp. 55–63. SIAM / Omnipress (2012)
37. Muise, Christian J .and McIlraith, S.A., Beck, J.C., Hsu, E.I.: Dsharp: Fast d-DNNF compilation with sharpSAT. In: Kosseim, L., Inkpen, D. (eds.) Proceedings of the 25th Canadian Conference on Artificial Intelligence (AI'17). Lecture Notes in Computer Science, vol. 7310, pp. 356–361. Springer Verlag, Toronto, ON, Canada (2012)
38. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms, Oxford Lecture Series in Mathematics and its Applications, vol. 31. Oxford University Press, New York, NY, USA (2006)
39. Oztok, U., Darwiche, A.: A top-down compiler for sentential decision diagrams. In: Yang, Q., Wooldridge, M. (eds.) Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15). pp. 3141–3148. The AAAI Press (2015)
40. Roth, D.: On the hardness of approximate reasoning. Artif. Intell. **82**(1-2), 273–302 (1996)

41. Samer, M., Szeider, S.: Algorithms for propositional model counting. Journal of Discrete Algorithms **8**(1), 50—64 (2010)
42. Sang, T., Bacchus, F., Beame, P., Kautz, H., Pitassi, T.: Combining component caching and clause learning for effective model counting. In: Hoos, H.H., Mitchell, D.G. (eds.) Online Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04). Vancouver, BC, Canada (2004)
43. Sang, T., Beame, P., Kautz, H.: Performing bayesian inference by weighted model counting. In: Veloso, M.M., Kambhampati, S. (eds.) Proceedings of the 29th National Conference on Artificial Intelligence (AAAI'05). The AAAI Press (2005)
44. Thurley, M.: sharpSAT – counting models with advanced component caching and implicit BCP. In: Biere, A., Gomes, C.P. (eds.) Proceedings of the 9th International Conference Theory and Applications of Satisfiability Testing (SAT'06). pp. 424–429. Springer Verlag, Seattle, WA, USA (2006)
45. Toda, S.: PP is as hard as the polynomial-time hierarchy. SIAM J. Comput. **20**(5), 865–877 (1991)
46. Toda, T., Soh, T.: Implementing efficient all solutions SAT solvers. ACM Journal of Experimental Algorithmics **21**, 1.12 (2015), special Issue SEA 2014, Regular Papers and Special Issue ALENEX 2013
47. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, Volume II. Computer Science Press (1989)
48. Xue, Y., Choi, A., Darwiche, A.: Basing decisions on sentences in decision diagrams. In: Hoffmann, J., Selman, B. (eds.) Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI'12). The AAAI Press, Toronto, ON, Canada (2012)