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 · Counting · Database Management Systems

1 Introduction

The *model counting problem* (#Sat) asks to compute the number of solutions of a propositional formula. A natural generalization of #Sat is weighted model counting (WMC), where formulas are extended by weights. Both #Sat and WMC are special cases of the weighted constraint satisfaction problem [31,39]. Nonetheless, they can already be used to solve a variety of applications to real-world questions in modern society, reasoning, and combinatorics [6,11,12,38,42]. Both #Sat and WMC are known to be complete for the class #P [3,35].

write following part before "Contribution"

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

In this paper, we consider both problems from the practical perspective. We present and evaluate a new version of a parallel model counter, called `gpusat2`, which is based on *dynamic programming (DP)* on tree decompositions (TDs) [36]. The idea of solving #SAT decomposing graph representations of the formula and applying DP on them is in fact quite well-known [36] and has earlier already been introduced for the constraint satisfaction problem (CSP) by Kask et al. [23]. Its underlying ideas are as follows. A TD of a propositional formula F is defined on a graph representation of F and formalizes a certain static relationship of the variables of F among each other. The decomposition then gives rise to an evaluation order and to sets of variables, which define which variables have to be evaluated together when solving the given formula. Intuitively, the width of a TD indicates how many variables have to be considered exhaustively together during the computation. Our previous solver `gpusat1` already implements DP-based weighted model counting and model counting using uniform weights on a GPU [20]. Prior to this, Fioretto et al. [21] presented an approach and implementation to compute one solution in weighted CSP, which could also be extended to solve the sum-of-products problem[4]. Here, we focus on an efficient computation and implementation of #SAT solving by introducing a novel architecture in our solver `gpusat2`. We focus on the so-called primal graph as graph representation, even though the incidence graph [36] theoretically allows for smaller width (off by one), mainly because simplicity of algorithms on the primal graph often outweighs the benefits of potential smaller width [14,20]. Our solver implements the principle of parallel programming of single instructions on multiple threads (SIMT) on a GPU. Therefore, we parallelize by executing the computation of variables that have to be considered exhaustively together on multiple threads, since the computation of an assignment to these variables is independent of other assignments during DP.

Contribution. We implement a solver `dpdb` for solving counting problems based on dynamic programming on tree decompositions, and present the following contributions. (i) Our solver `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 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. In particular, we compare the runtime of our system with state-of-the-art model counters, where we observe competitive behavior.

lots of it

Related Work.

[4] The sum-of-product problem is often also referred to as weighted counting, partition function, or probability of evidence.

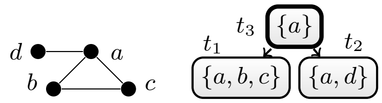

Figure 1: Graph G (left) with a TD $\mathcal{T}$ of graph G (right).

2 Preliminaries

Boolean Satisfiability. We define Boolean formulas and their evaluation in the usual way, cf., [22,24]. A literal is a Boolean variable x or its negation $\neg x$. A *CNF formula* φ 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 φ is a mapping $I : \mathrm{var}(\varphi) \to \{0, 1\}$. The formula $\varphi(I)$ *under assignment* I is obtained by removing every clause c from φ that contains a literal set to 1 by I, and removing from every remaining clause of φ all literals set to 0 by I. An assignment I is *satisfying* if $\varphi(I) = \emptyset$. *Problem* #SAT asks to output the number of satisfying assignments of a formula.

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 χ is a mapping which assigns to each node $t \in V(T)$ a set $\chi(t) \subseteq V(G)$, called *bag*, such that (i) $V(G) = \bigcup_{t \in V(T)} \chi(t)$ and $E(G) \subseteq \{\, \{u, v\} \mid t \in V(T), \{u, v\} \subseteq \chi(t) \,\}$; and (ii) for each $r, s, t \in V(T)$, such that s lies on the path from r to t, we have $\chi(r) \cap \chi(t) \subseteq \chi(s)$. We let $\mathrm{width}(\mathcal{T}) := \max_{t \in V(T)} |\chi(t)| - 1$. 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 [25] a complete graph with 3 vertices. ■

Relational Algebra. We use relational algebra [7] for manipulation of relations, which forms the theoretical basis of its the well-known implementation database standard *Structured Query Language (SQL)* [] 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 $\mathrm{att}(R) := \mathrm{att}(r)$ of attributes. Given a relation R over $\mathrm{att}(R)$. Then, we let $\mathrm{dom}(R) := \bigcup_{a \in \mathrm{att}(R)} \mathrm{dom}(a)$, and let relation R *projected to* $A \subseteq \mathrm{att}(R)$ be given by $\Pi_A(R) := \{r_A \mid r \in R\}$, where $r_A := \{(a, v) \mid (a, v) \in r, a \in A\}$. mh:extended projection
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

Listing 2: 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 φ_t, sequence $\langle \tau_1, \dots \tau_\ell \rangle$ of child tables. **Out:** Table τ_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**
 $\tau_t := \{\langle I, c_1 \cdot c_2 \rangle \mid \langle I, c_1 \rangle \in \tau_1, \langle I, c_2 \rangle \in \tau_2\}$

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

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 var(φ) 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, wet let *θ-join* correspond to $R \bowtie_\varphi R' := \sigma_\varphi(R \times R')$.

mh: group by

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. 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 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. Then, we let local problem #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 2 shows all the cases that are needed to solve #Sat by means of DP of nice TDs. Each table τ_t consist of rows

[5] 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 $\neg v$.

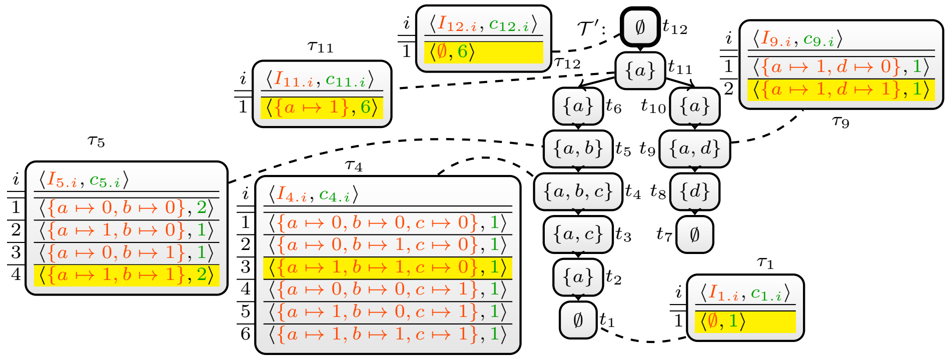

Figure 2: Selected tables obtained by DP on $\mathcal{T}'$ for φ of Example 2 using algorithm S.

of the form $\langle I, c\rangle$, where I is an assignment of φ_t and c is a counter. Nodes t with $\text{type}(t) = \textit{leaf}$ 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 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 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 $\tau_1 = \{\, \langle \emptyset, 1\rangle \,\}$ as $\text{type}(t_1) = \textit{leaf}$. Since $\text{type}(t_2) = \textit{intr}$, 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 $\chi(t_4) \subseteq \text{var}(c_1)$ 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 $\text{type}(t_5) = \textit{rem}$, 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 $\text{type}(t_{11}) = \textit{join}$, we build table τ_{11} by taking the intersection of τ_6 and τ_{10}. Intuitively, this combines assignments agreeing on a, where counters are multiplied accordingly. By definition (primal graph and TDs), for every $c \in \varphi$, variables $\text{var}(c)$ occur together in at least one common bag. Hence, since $\tau_{12} = \{\langle \emptyset, 6\rangle\}$, we can reconstruct for example model $\{a \mapsto$

Listing 3: 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 φ_t, sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child tables. **Out:** Table τ_t.

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

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

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

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

5 $\quad \tau_t := {}_{\chi(t)}G_{\mathrm{cnt} \leftarrow \mathtt{SUM(cnt)}}(\Pi_{\mathrm{att}(\tau_1) \setminus \{[\![a]\!]\}} \tau_1)$

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

7 $\quad \tau_t := \Pi_{\mathrm{att}(\tau_1)}[\dot{\Pi}_{\{\mathrm{cnt} \leftarrow \mathrm{cnt} \cdot \mathrm{cnt}'\}}(\tau_1 \bowtie_{\bigwedge_{a \in \chi(t)} [\![a]\!] \approx [\![a]\!]'} \rho_{\bigcup_{a \in \mathrm{att}(\tau_2)} \{[\![a]\!] \mapsto [\![a]\!]'\}} \tau_2)]$

$1, b \mapsto 1, c \mapsto 0, d \mapsto 1\} = I_{11.1} \cup I_{5.4} \cup I_{9.2}$ 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 are performed, 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 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 $\mathrm{type}(t) = \mathit{leaf}$, a fresh initial table τ_t, cf., Line 1 of Listing 3. Then, whenever an atom a is introduced, such algorithms often use θ-joins with a fresh initial table for the introduced variable a that represent potential values a can have. In Line 3 the selection of the θ-join is performed by ensuring φ_t, corresponding to the local problem of #SAT. Further, for nodes t with $\mathrm{type}(t) = \mathit{rem}$, 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 τ_1 might collapse in τ_t. Finally, for a node t with $\mathrm{type}(t) = \mathit{join}$, in Line 7 we use again θ-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

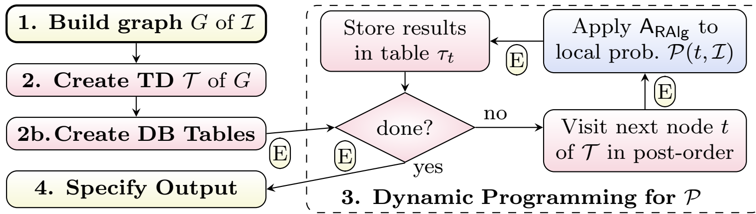

Figure 3: Architecture of Dynamic Programming with Databases. Steps highlighted in red are provided by the system depending on specification of yellow and blue parts, which is given by the user for specific problems $\mathcal{P}$. The yellow "E"s represent events that can be intercepted and handled by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

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 code[6] for 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 4 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 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

[6] Recall that SQL is a specific implementation standard (set) of relational algebra.
[7] Our system dpdb is available under GPL3 license at github.com/hmarkus/dp_on_dbs.

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

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

5 $\quad \tau_t := {}_{\chi(t)}G_{\#\mathsf{aggrExp}\#}(\Pi_{\mathrm{att}(\tau_1) \setminus \{[\![a]\!]\}} \tau_1)$

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

7 $\quad \tau_t := \Pi_{\mathrm{att}(\tau_1)}[\dot{\Pi}_{\#\mathsf{extProj}\#}(\tau_1 \bowtie_{\bigwedge_{a \in \chi(t)} [\![a]\!] \approx [\![a]\!]'} \rho_{\bigcup_{a \in \mathrm{att}(\tau_2)} \{[\![a]\!] \mapsto [\![a]\!]'\}} \tau_2)]$

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 works extremely quick also for interesting instances [**?**] 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 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 it is not wise though to consider θ-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 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 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 4 automatically. Concretely, the system executes one query per table τ_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 τ_t consisting of the θ-join as in Line 7 of Listing 4, including parts of Line 3, namely cross-joins for each introduced variable, involving $\#\mathsf{intrTab}\#$ without the filtering on $\#\mathsf{localProbFilter}\#$. 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 τ_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 φ 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 ti containing in the end the content of vi. Observe that each view only has one column a for variable a of φ 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 $t1$ has two rows, $t2$, and $t3$ have only one row. We obtain the following views.

```
CREATE VIEW v1 AS SELECT a, sum(cnt) FROM
 (WITH intrTab AS (SELECT true val UNION ALL SELECT false)
   SELECT i1.val a, i2.val b, i3.val c, 1 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) FROM
 (WITH intrTab AS (SELECT true val UNION ALL SELECT false)
   SELECT i1.val a, i2.val d, 1 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) FROM
 (SELECT t1.a, t1.cnt * t2.cnt cnt FROM t1, t2 WHERE t1.a = t2.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.

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

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). In particular, 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 was inserted. 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.

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 $\varphi_t = \{\{l_{1,1}, \dots, l_{1,k_1}\}, \dots, \{l_{n,1}, \dots, l_{n,k_n}\}\}$, where `ti` refers to the *database table* τ_i for node t_i as in Example 3.

- #εTab#: `SELECT 1 AS cnt`
- #intrTab#: `SELECT 1 AS val UNION ALL 0`
- #localProbFilter#: $(l_{1,1}$ `OR` $\dots$ `OR` $l_{1,k_1})$ `AND` $\dots$ `AND` $(l_{n,1}$ `OR` $\dots$ `OR` $l_{n,k_n})$
- #aggrExp#: `SUM(cnt) AS cnt`
- #extProj#: `t1.cnt` $\cdot \dots \cdot$ `tℓ.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, #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 $\iota : V \to \{1, \dots, o\}$ 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 $E(G_t) = \{\{u_1, v_1\}, \dots, \{u_n, v_n\}\}$.

- #εTab#: `SELECT 1 AS cnt`
- #intrTab#: `SELECT 1 AS val UNION ALL ... UNION ALL` u
- #localProbFilter#: `NOT` ($[\![u_1]\!] = [\![v_1]\!]$) `AND ... AND NOT` ($[\![u_n]\!] = [\![v_n]\!]$)
- #aggrExp#: `SUM(cnt) AS cnt`
- #extProj#: `t1.cnt` $\cdot \ldots \cdot$ `t`ℓ`.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 $\text{MinVC}(t, G) := G_t$ is defined as above. To this end, we use an additional column `card` for storing cardinalities.

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

- #εTab#: `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.

TODO:

- approximate memory used by db for one of the largest problems solved
- Supported DIMACS formats: cnf, td, tw, edge
- 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.

5 Experiments

We conducted a series of experiments using several benchmark sets for model counting and weighted model counting. Benchmark sets [16] and our results [17] are publicly available and also on github at daajoe/gpusat_experiments.

Measure, Setup, and Resource Enforcements. As we use different types of hardware in our experiments and other natural measures such as power consumption cannot be recorded with current hardware, we compare wall clock time and number of timeouts. In the time we include, if applicable, *preprocessing time* as well as *decomposition time* for computing 30 decompositions with a random seed and decomposition selection time. However, we avoid IO access on the CPU solvers whenever possible, i.e., we load instances into the RAM before we start solving. For parallel CPU solvers we allow access to 12 or 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 for model counting consisting of `fre/meel` benchmarks[9](1480 instances), and `c2d` benchmarks[10] (14 instances). For WMC, we used the overall 1091 instances from the `Cachet` benchmark set[11].

Benchmarked Solvers. In our experimental work, we present results for the most recent versions of publicly available #Sat solvers, namely, *c2d* 2.20 [9], *d4* 1.0 [29], *DSHARP* 1.0 [32], *miniC2D* 1.0.0 [33], *cnf2eadt* 1.0 [26], *bdd_minisat_all* 1.0.2 [41], and *sdd* 2.0 [10] (based on knowledge compilation techniques). We also considered rather recent approximate solvers *ApproxMC2*, *ApproxMC3* [5] and *sts* 1.0 [13], as well as CDCL-based solvers *Cachet* 1.21 [37], *sharpCDCL*[12], and *sharpSAT* 13.02 [40]. Finally, we also included multi-core solvers *gpusat* 1.0 [20], as well as *countAntom* 1.0 [4] on 12 physical CPU cores, which performed better than on 24 cores. Note that we benchmarked additional solvers, which we omitted from the presentation here and where we placed results online in our result data repository. For WMC, we considered the following solvers: *sts*, *gpusat1*, gpusat2, *miniC2D*, *Cachet*, *d4*, and *d-DNNF reasoner* 0.4.180625 (on top of d4 as underlying knowledge compiler). All experiments were conducted with default solver options. For solver gpusat2, we also benchmarked variant gpusat2(A+B) where we used 30 as threshold above which we apply the BST.

Benchmark Machines. The non-GPU solvers were executed on a cluster of 9 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, which is already a post-Spectre and post-Meltdown kernel[13]. 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.

5.1 Results

First, we present how existing preprocessors for #Sat and equivalence-preserving preprocessors for WMC influence the treewidth on the considered instances.

[9] See: tinyurl.com/countingbenchmarks
[10] See: reasoning.cs.ucla.edu/c2d
[11] See: cs.rochester.edu/u/kautz/Cachet
[12] See: tools.computational-logic.org
[13] Details on spectre and meltdown: spectreattack.com.

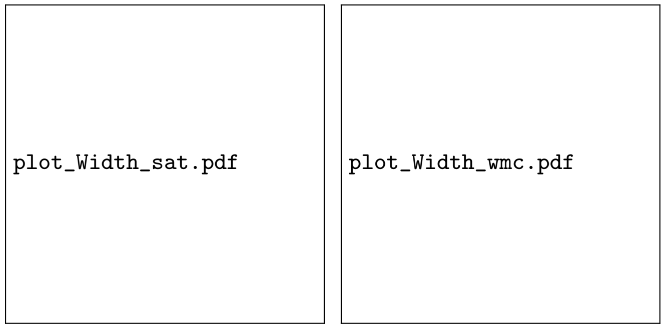

Figure 4: Width distribution of #Sat instances (left) before and after preprocessing (using both B+E and pmc). Width distribution of WMC instances (right) before and after preprocessing using pmc*. Results are based on the primal treewidth and presented in intervals. X-axis labels the intervals, y-axis labels the number of instances.

prob	pre	vMdn	cMdn	t[s] Mdn	to	t[s] Mdn pre	to	Mdn	50%	80%	90%	95%	min	max	mdn
#Sat	w/o pre	637	810	0.07	**6**	n/a	n/a	31	31	166	378	922	n/a	n/a	n/a
	pmc, B+E	**231**	350	0.02	**6**	0.06	192	**3**	**3**	**17**	201	**823**	-72	**755**	22
	pmc	**231**	189	0.03	**6**	**0.03**	**103**	**3**	4	19	228	**823**	-1839	547	**23**
	B+E	**231**	**185**	**0.02**	**6**	0.04	189	**3**	**3**	18	**192**	**823**	**-2**	633	**23**
WMC	w/o pre	**200**	519	0.04	**0**	n/a	n/a	28	28	40	43	54	n/a	n/a	n/a
	pmc*	**200**	**300**	**0.03**	**0**	**0.03**	**0**	**11**	**11**	**20**	**25**	**30**	**0**	**330**	**16**

Table 1: Overview on upper bounds of the primal treewidth for considered #Sat and WMC benchmarks before and after preprocessing. vMdn median of variables, cMdn median of clauses, t[s] Mdn of the decomposition runtime in seconds, maximum runtime t[s] Max, median Mdn and percentiles of upper bounds on treewidth, and min/max/mdn of the width improvement after preprocessing. Negative values indicate worse results.

Treewidth Analysis. We computed upper bounds on the primal treewidth for our benchmarks before and after preprocessing and state them in intervals. For model-count preserving preprocessing we explored both B+E Apr2016 [27] and pmc 1.1 [28]. For WMC, we used pmc with documented options $-\mathsf{vivification}$ $-\mathsf{eliminateLit}$ $-\mathsf{litImplied}$ $-\mathsf{iterate} = 10$ to preserve all the models, which we refer to by *pmc**. In this experiment, we used different timeouts. We set the timeout of the preprocessors to 900 seconds and allowed further 1800 seconds for the decomposer to get a detailed picture of treewidth upper bounds. Figure 4 (left) presents the width distribution of number of instances (y-axis) and their corresponding upper bounds (x-axis) for primal treewidth, both before and after preprocessing using B+E, pmc, and both preprocessors in combination (first pmc, then B+E) for #Sat. Table 1 (top) provides statistics on the benchmarks combined, including runtime of the preprocessor, runtime of the decomposer to obtain a decomposition, upper bounds on primal treewidth, and its improvements before and after preprocessing. Further, the table also lists the median of the widths of the obtained decompositions and their percentiles, which is the treewidth upper bound a given percentage of the instances have. Interestingly, overall we

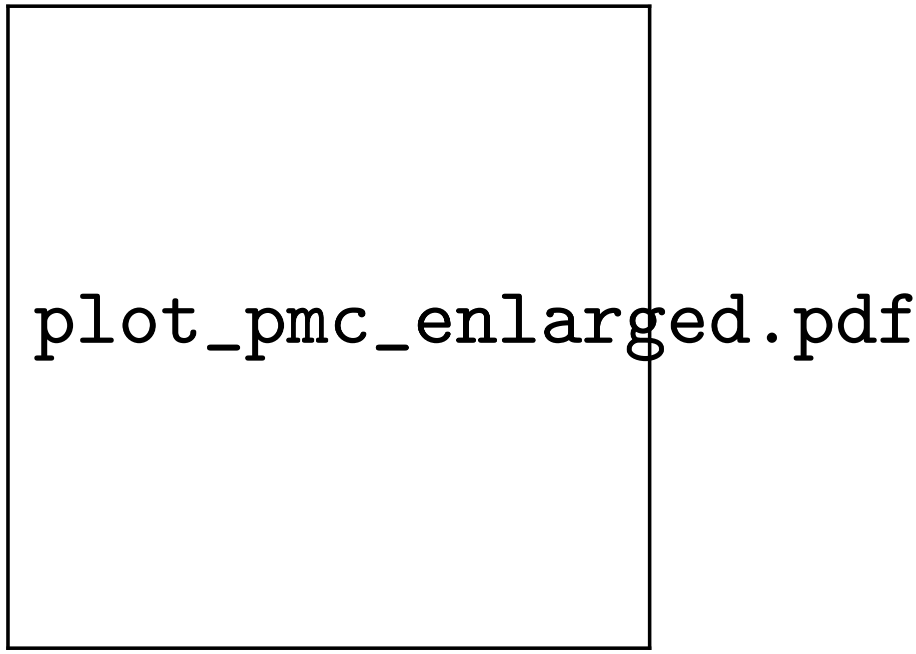

Figure 5: Runtime for the top 5 sequential and all parallel solvers over all the #Sat instances with pmc preprocessor. The x-axis refers to the number of instances and the y-axis depicts the runtime sorted in ascending order for each solver individually.

have that a majority of the instances after preprocessing has width below 20. In more details, more than 80% of the #Sat instances have primal treewidth below 19 after preprocessing, whereas 90% of the instances have treewidth below 192 for B+E. With pmc we observed a corner case where the primal treewidth upper bound increased by 1839, however, on average we observed a mean improvement on the upper bound of slightly above 23. The best improvement among the widths of all our instances was achieved with the combination of pmc and B+E where we improved the width by 755. Overall, both B+E and pmc managed to *drastically reduce* the widths, the decomposer ran below 0.1 seconds in median. Interestingly, even the upper bounds on the treewidth of the WMC instances reduced with pmc* as depicted in Figure 4 (right). In more detail, after preprocessing 95% of the instances have primal treewidth below 30, c.f., Table 1 (bottom).

Solving Performance Analysis. Figure 5 illustrates the top five sequential solvers, and all parallel counting solvers with preprocessor pmc in a cactus-like plot. Table 2 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

	solver	0-20	21-30	31-40	41-50	51-60	>60	best	unique	$\sum$	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	$\sum$	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	$\sum$	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	$\sum$	time[h]

Table 2: 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.

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 2 (top, mid) and Figure 5 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 2 (mid), gpusat2 solves even more instances as well as the other solvers. Still, it ranks fifth solving 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.

While we focus on #SAT with our implementation, we also conducted the experiments with WMC in order to compare our solver with gpusat1 in the setting for which it was designed. Table 3 (top) lists results of the top five best solvers

	solver	0-20	21-30	31-40	41-50	51-60	>60	best	unique	$\sum$	time[h]
with pmc*	miniC2D	858	**164**	**6**	**0**	**0**	3	13	**8**	**1031**	21.29
	gpusat1	**866**	158	0	**0**	**0**	0	**348**	4	1024	18.03
	gpusat2(A+B)	**866**	156	0	**0**	**0**	0	343	4	1022	**17.86**
	gpusat2	**866**	138	0	**0**	**0**	0	299	4	1004	22.43
	d4	810	106	0	**0**	**0**	0	46	0	916	55.36
	cachet	617	128	1	**0**	**0**	3	106	1	749	93.65
without pre	d4	82	501	**142**	**156**	**10**	**19**	111	**24**	**910**	**53.97**
	miniC2D	84	517	134	152	3	4	19	7	894	59.69
	gpusat2(A+B)	**86**	**527**	98	138	0	0	167	19	849	64.40
	gpusat2	**86**	511	98	138	0	0	131	7	833	68.61
	gpusat1	**86**	513	68	140	0	0	**182**	10	807	73.78
	cachet	60	447	100	145	2	9	118	1	763	89.80

Table 3: Number of WMC instances solved (with)out preprocessing. time[h] is the cumulated wall clock time in hours, where unsolved instances count as 900 seconds.

capable of solving WMC on our instances. Compared to gpusat1, our solver gpusat2 shows an improvement when the width of the instance is between 21 and 40, in more detail gpusat2 solves 44 instances more. After preprocessing with pmc*, one can observe that the majority of the instances has width below 20, c.f., Table 3 (bottom). As a result, gpusat2 does not provide significant improvement over gpusat1 there apart from small runtime improvements.

Currently, we are unable to measure the speed-up of the implementations in terms of the used cores, mainly due to the fact that we aimed for an implementation that is close to gpusat1 so that the improvements are actually from the architectural changes and not just from the different framework or drivers. Note that OpenCL does not support disabling certain cores on the GPU. We also benchmarked gpusat2 on an Nvidia GPU, whose runtimes are quite similar. We also provide preliminary data online with the experiments; which are however not conclusive yet. However, we we ran into bugs, which seems to be attributed to the OpenCL1.2 Nvidia driver. Therefore, we aim as future work for a new implementation in CUDA [8].

6 Related Work and Conclusion

Related Work. Fioretto et al. [21] introduced a solver for outputting a solution to the weighted CSP problem using a GPU. Their technique is effectively a version of dynamic programming on tree decompositions also known as bucket-elimination, which they limited to an incomplete elimination by introducing shortcuts and discarding non-optimal solutions in order to speed up the computation for the problem of outputting just one solution. While the underlying idea of a dynamic programming based solver still exists in our solver, gpusat2 is very different when just taking a slightly more detailed look. We approach the *counting question – not just outputting one solution*, which disallows certain simplifications. We can neither apply an incomplete bucket-elimination technique (mini-bucket elimination) nor discard non-optimal solutions. But then, we consider the binary case, which allows us to introduce various simplifications including the way we store the data enabling us to save memory and to avoid copying data. Also, we would like to point out that bucket-elimination is used in the decomposer htd to compute just the tree decomposition. In that way, our architecture is quite general as it completely separates the decomposition and the actual computation part resulting in a framework that can also be used for other problems. Moreover, we use more sophisticated data structures and split data whenever the data does not fit into the VRAM of the GPU. Finally, we balance between small width during the computation and not too small width as we want to employ the full computational power of the parallelization with the GPU. In the past, a variety of model counters and weighted model counters have been implemented based on several different techniques. We listed them in details in Section 5. However, here we want to highlight a few differences between our technique and knowledge compilation-based techniques as well as distributed computing. The solver d4 [29], which implements a knowledge compilation-based approach, employs heuristics to compute decompositions of an underlying hypergraph, namely the dual hypergraph, and uses this during the computation. Note that the following relationships are known for treewidth (i.e., the width of a tree decomposition of smallest width) of an arbitrary formula F, $\mathrm{inctw}(F) \leq \mathrm{dualtw}(F) + 1$ and $\mathrm{inctw}(F) \leq \mathrm{primtw}(F) + 1$, where inctw refers to the treewidth of the incidence graph, dualtw of the dual graph, and primtw of the primal graph. However, there is no such relationship between the treewidth of the primal and dual graph. We are currently unaware of how these theoretical results generalize to hypergraphs. Experimentally, it is easy to verify that a decomposition of the dual graph is often not useful in our context as it provides only decompositions of large width. When we consider parallel solving, a few words on distributed counting are in order. In fact, the model counter DMC [30] is intended for parallel computation on a cluster of computers using the message passing model (MPI). However, this distributed computation requires a separate setup of the cluster and exclusive access to multiple nodes. We focus on parallel counting with a shared memory model. For details, we refer to the difference between parallel and distributed computation [34].

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 [21]. Further, we are also interested in extending this work to projected model counting [15,18,19].

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. 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)
5. 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)
6. 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)
7. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (1970)
8. Cook, S.: CUDA Programming: A Developer's Guide to Parallel Computing with GPUs. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2013)
9. 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)
10. 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)
11. Domshlak, C., Hoffmann, J.: Probabilistic planning via heuristic forward search and weighted model counting. Journal of Artificial Intelligence Research **30** (2007)
12. 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)
13. 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)

14. Fichte, J.K., Hecher, M., Morak, M., Woltran, S.: Answer set solving with bounded treewidth revisited. In: Balduccini, M., Janhunen, T. (eds.) Proceedings of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'17). Lecture Notes in Computer Science, vol. 10377, pp. 132–145. Springer Verlag, Espoo, Finland (Jul 2017)
15. 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)
16. 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
17. Fichte, J.K., Hecher, M., Zisser, M.: Analyzed Benchmarks and Raw Data on Experiments for gpusat2 (Dataset) (Jul 2019), https://doi.org/10.5281/zenodo.3337727
18. 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)
19. 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
20. Fichte, J.K., Hecher, M., Woltran, S., Zisser, M.: Weighted model counting on the GPU by exploiting small treewidth. In: Azar, Y., Bast, H., Herman, G. (eds.) Proceedings of the 26th Annual European Symposium on Algorithms (ESA'18). Leibniz International Proceedings in Informatics (LIPIcs), vol. 112, pp. 28:1–28:16. Dagstuhl Publishing (2018)
21. 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)
22. 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)
23. Kask, K., Dechter, R., Larrosa, J., Cozman, F.: Bucket-tree elimination for automated reasoning. Tech. rep., Donald Bren School of Information and Computer Sciences University of California at Irvine (2001), https://www.ics.uci.edu/~dechter/publications/
24. Kleine Büning, H., Lettman, T.: Propositional logic: deduction and algorithms. Cambridge University Press, Cambridge, New York, NY, USA (1999)
25. Kloks, T.: Treewidth. Computations and Approximations, Lecture Notes in Computer Science, vol. 842. Springer Verlag (1994)
26. 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)
27. Lagniez, J., Lonca, E., Marquis, P.: Improving model counting by leveraging definability. In: Kambhampati, S. (ed.) Proceedings of 25th International Joint

Conference on Artificial Intelligence (IJCAI'16). pp. 751–757. The AAAI Press, New York City, NY, USA (Jul 2016)
28. Lagniez, J., Marquis, P.: Preprocessing for propositional model counting. In: Brodley, C.E., Stone, P. (eds.) Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI'14). pp. 2688–2694. The AAAI Press, Québec City, QC, Canada (2014)
29. 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)
30. Lagniez, J.M., Marquis, P., Szczepanski, N.: Dmc: A distributed model counter. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI'18. pp. 1331–1338. The AAAI Press (7 2018)
31. Larrosa, J.: Node and arc consistency in weighted csp. In: Dechter, R., Kearns, M., Sutton, R.S. (eds.) Proceedings of the 18th AAAI conference on Artificial Intelligence (AAAI'02). pp. 48–53. The AAAI Press, Edmonton, AB, Canada (Jul 2002)
32. 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)
33. 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)
34. Raynal, M.: Parallel computing vs. distributed computing: A great confusion? (position paper). In: Hunold, S., Costan, A., Giménez, D., Iosup, A., Ricci, L., Gómez Requena, M.E., Scarano, V., Varbanescu, A.L., Scott, S.L., Lankes, S., Weidendorfer, J., Alexander, M. (eds.) Proceedings of the Parallel Processing Workshops (Euro-Par'15). Lecture Notes in Computer Science, vol. 9523, pp. 41–53. Springer Verlag (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: 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)
38. 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)
39. Shapiro, L.G., Haralick, R.M.: Structural descriptions and inexact matching. IEEE Transactions on Pattern Analysis and Machine Intelligence **PAMI-3**(5), 504–519 (1981)
40. 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)

41. 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
42. 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)