Taming High Treewidth with Abstraction, Nested Dynamic Programming, and Database Technology

Markus Hecher(✉), Patrick Thier(✉), and Stefan Woltran(✉)

Institute of Logic and Computation, TU Wien, Vienna, Austria
{hecher,thier,woltran}@dbai.tuwien.ac.at

Abstract. Treewidth is one of the most prominent structural parameters. While numerous theoretical results establish tractability under the assumption of fixed treewidth, the practical success of exploiting this parameter is far behind what theoretical runtime bounds have promised. In particular, a naive application of dynamic programming (DP) on tree decompositions (TDs) suffers already from instances of medium width. In this paper, we present several measures to advance this paradigm towards general applicability in practice: We present nested DP, where different levels of abstractions are used to (recursively) compute TDs of a given instance. Further, we integrate the concept of hybrid solving, where subproblems hidden by the abstraction are solved by classical search-based solvers, which leads to an interleaving of parameterized and classical solving. Finally, we provide nested DP algorithms and implementations relying on database technology for variants and extensions of Boolean satisfiability. Experiments indicate that the advancements are promising.

1 Introduction

Treewidth [43] is a prominent structural parameter, originating from graph theory and is well-studied in the area of parameterized complexity [6,18,40]. For several problems hard for complexity class NP, there are results [12] showing so-called (fixed-parameter) tractability, which indicates a *fixed-parameter tractable (FPT)* algorithm running in polynomial time assuming that a given parameter (e.g., treewidth) is fixed. Practical implementations exploiting treewidth include generic frameworks [3,5,36], but also dedicated solvers that deal with problems ranging from (counting variants of) Boolean satisfiability (Sat) [25], over generalizations thereof [9,10] based on *Quantified Boolean Formulas (QBFs)*, to formalisms relevant to knowledge representation and reasoning [22]. For Sat, these solvers are of particular interest as there is a well-known correspondence between treewidth and resolution width [2]. QBFs extend Boolean logic by explicit universal and existential quantification over variables, which has applications in formal verification, synthesis, and AI problems such as planning [28]. Some of

L. Pulina and M. Seidl (Eds.): SAT 2020, LNCS 12178, pp. 343–360, 2020.
https://doi.org/10.1007/978-3-030-51825-7_25

these parameterized solvers are particularly efficient for certain fragments [37], and even successfully participated in problem-specific competitions [42].

Most of these systems are based on *dynamic programming (DP)*, where a tree decomposition (TD) is traversed in a post-order, i.e., from the leaves towards the root, and thereby for each TD node tables are computed. The size of these tables (and thus the computational efforts required) are bounded by a function in the treewidth of the instance. Although dedicated competitions [15] for treewidth advanced the state-of-the-art for efficiently computing treewidth and TDs [1,47], these DP approaches reach their limits when instances have higher treewidth; a situation which can even occur in structured real-world instances [38]. Nevertheless in the area of Boolean satisfiability, this approach proved to be successful for counting problems, such as, e.g., (weighted) model counting [24,25,44] and projected model counting [23].

To further increase the applicability of this paradigm, novel techniques are required which (1) rely on different levels of abstraction of the instance at hand; (2) treat subproblems originating in the abstraction by standard solvers whenever widths appear too high; and (3) use highly sophisticated data management in order to store and process tables obtained by dynamic programming.

Contributions. Above aspects are treated as follows.

1. To tame the beast of high treewidth, we propose *nested dynamic programming*, where only parts of an abstraction of a graph are decomposed. Then, each TD node also needs to solve a *subproblem* residing in the graph, but may involve vertices outside the abstraction. In turn, for solving such subproblems, the idea of nested DP is to subsequently repeat decomposing and solving more fine-grained graph abstractions in a nested fashion. This results not only in elegant DP algorithms, but also allows to deal with high treewidth. While candidates for obtaining abstractions often originate naturally from the problem, nested DP may require non-obvious sub-abstractions, for which we present a generic solution.
2. To further improve the capability of handling high treewidth, we show how to apply nested DP in the context of *hybrid solving*, where established, standard solvers (e.g., SAT solvers) and caching are incorporated in nested DP such that the best of two worlds are combined. Thereby, structured solving is applied to parts of the problem instance subject to counting or enumeration, while depending on results of subproblems. These subproblems (subject to search) reside in the abstraction only, and are solved via standard solvers.
3. We implemented a system based on a recently published tool called `dpdb` [24] for using database management systems (DBMS) to efficiently perform table manipulation operations needed during DP. Our system uses and significantly extends this tool in order to perform hybrid solving, thereby combining nested DP and standard solvers. As a result, we use DBMS for efficiently implementing the handling of tables needed by nested DP. Preliminary experiments indicate that nested DP with hybrid solving can be fruitful.

We exemplify these ideas on the problem of Projected Model Counting (#∃SAT) and discuss adaptions for other problems.

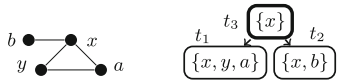

Fig. 1. Graph G (left), a TD $\mathcal{T}$ of graph G (right).

2 Background

Projected Model Counting. We define Boolean formulas in the usual way, cf., [28]. A literal is a Boolean variable x or its negation $\neg x$. A *(CNF) formula* φ is a set of *clauses* interpreted as conjunction. A clause is a set of literals interpreted as disjunction. For a formula or clause X, we abbreviate by $\mathrm{var}(X)$ the variables that occur in X. An *assignment* of φ is a mapping $I : \mathrm{var}(\varphi) \to \{0, 1\}$. The formula $\varphi[I]$ *under assignment* I is obtained by removing every clause c from φ that contains a literal set to 1 by I, and removing from every remaining clause of φ all literals set to 0 by I. An assignment I is *satisfying* if $\varphi[I] = \emptyset$. *Problem* #Sat asks to output the number of satisfying assignments of a formula. *Projected Model Counting* #∃Sat takes a formula φ and a set $P \subseteq \mathrm{var}(\varphi)$ of *projection variables*, and asks for $\#\exists\mathrm{Sat}(\varphi, P) := |\{I^{-1}(1) \cap P \mid \varphi[I] = \emptyset\}|$. Consequently, $\mathrm{Sat}(\varphi) := \#\exists\mathrm{Sat}(\varphi, \emptyset)$, and $\#\mathrm{Sat}(\varphi) := \#\exists\mathrm{Sat}(\varphi, \mathrm{var}(\varphi))$. #∃Sat is #·NP-complete [19] and thus probably harder than #Sat (#P-complete).

Tree Decomposition and Treewidth. We assume familiarity with graph terminology, cf., [17]. A *tree decomposition (TD)* [43] of a given graph G is a pair $\mathcal{T} = (T, \chi)$ where T is a rooted tree and χ assigns to each node $t \in V(T)$ a set $\chi(t) \subseteq V(G)$, called *bag*, such that (i) $V(G) = \bigcup_{t \in V(T)} \chi(t)$, (ii) $E(G) \subseteq \{\, \{u, v\} \mid t \in V(T), \{u, v\} \subseteq \chi(t) \,\}$, and (iii) 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 $t \in V(T)$, $\mathrm{type}(t) \in \{\mathit{leaf}, \mathit{join}, \mathit{intr}, \mathit{rem}\}$, the TD is called *nice*. A nice TD can be computed from a given TD $\mathcal{T}$ in linear time without increasing the width [31], assuming the width of $\mathcal{T}$ is fixed.

Example 1. Figure 1 depicts a graph G and a (non-nice) TD $\mathcal{T}$ of G of width 2.

Relational Algebra. We formalize DP algorithms by means of relational algebra [11], similar to related work [24]. A *table* τ is a finite set of *rows* r over a set $\mathrm{att}(\tau)$ of *attributes*. Each *row* $r \in \tau$ is a set of pairs (a, v) with $a \in \mathrm{att}(\tau)$ and v in *domain* $\mathrm{dom}(a)$ of a, s.t. for each $a \in \mathrm{att}(\tau)$ there is exactly one $(a, v) \in r$. Notably, apart from counters we use mainly binary domains in this paper.

Selection of rows in τ according to a Boolean formula φ is defined by $\sigma_\varphi(\tau) := \{r \mid r \in \tau, \varphi[\mathrm{ass}(r)] = \emptyset\}$, assuming that $\mathrm{ass}(r)$ refers to the truth assignment over the attributes of binary domain of a given row $r \in \tau$. Given a relation τ'

Listing 1: Table algorithm $\#\text{SAT}_t(\chi_t, \varphi_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for solving #SAT on node t of a nice tree decomposition, cf., [24].

In: Bag χ_t, bag formula φ_t, child tables $\langle \tau_1, \ldots \tau_\ell \rangle$ of t.
Out: Table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t := \{\{(\text{cnt}, 1)\}\}$
2 **else if** type$(t) =$ *intr, and* $a \in \chi_t$ *is introduced* **then**
3 $\quad \tau_t := \tau_1 \bowtie_{\varphi_t} \{\{(a,0)\}, \{(a,1)\}\}$
4 **else if** type$(t) =$ *rem, and* $a \notin \chi_t$ *is removed* **then**
5 $\quad \tau_t := {}_{\chi_t}G_{\text{cnt} \leftarrow \text{SUM(cnt)}}(\Pi_{\text{att}(\tau_1) \setminus \{a\}} \tau_1)$
6 **else if** type$(t) =$ *join* **then**
7 $\quad \tau_t := \dot{\Pi}_{\chi_t, \{\text{cnt} \leftarrow \text{cnt} \cdot \text{cnt}'\}}(\tau_1 \bowtie_{\bigwedge_{a \in \chi_t} a = a'} \rho_{\bigcup_{a \in \text{att}(\tau_2)} \{a \mapsto a'\}} \tau_2)$

with $\text{att}(\tau') \cap \text{att}(\tau) = \emptyset$, we refer to the *cross-join* by $\tau \times \tau' := \{r \cup r' \mid r \in \tau, r' \in \tau'\}$. Further, a *join (using* φ*)* corresponds to $\tau \bowtie_\varphi \tau' := \sigma_\varphi(\tau \times \tau')$. We define *renaming* of τ, given a set A of attributes, and a bijective mapping $m : \text{att}(\tau) \to A$ by $\rho_m(\tau) := \{(m(a), v) \mid (a, v) \in \tau\}$. τ *projected to* $A \subseteq \text{att}(\tau)$ is given by $\Pi_A(\tau) := \{r_A \mid r \in \tau\}$, where $r_A := \{(a, v) \mid (a, v) \in r, a \in A\}$. This is lifted to *extended projection* $\dot{\Pi}_{A,(a \leftarrow f)}$, assuming attribute $a \in \text{att}(\tau) \setminus A$ and arithmetic function $f : \tau \to \mathbb{N}$. Formally, we define $\dot{\Pi}_{A,(a \leftarrow f)}(\tau) := \{r_A \cup \{(a, f(r))\} \mid r \in \tau\}$. We use *aggregation by grouping* ${}_AG_{(a \leftarrow g)}$, where we assume $A \subseteq \text{att}(\tau), a \in \text{att}(\tau) \setminus A$ and an *aggregate function* $g : 2^\tau \to \text{dom}(a)$. We define ${}_AG_{(a \leftarrow g)}(\tau) := \{r \cup \{(a, g(\tau[r]))\} \mid r \in \Pi_A(\tau)\}$, where $\tau[r] := \{r' \mid r' \in \tau, r' \supseteq r\}$.

3 Towards Nested Dynamic Programming

A solver based on *dynamic programming (DP)* evaluates a given input instance $\mathcal{I}$ in parts along a given TD of a graph representation G of the instance. 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*, which is designed for a certain graph representation, and stores in τ_t results of problem parts of $\mathcal{I}$, thereby considering tables $\tau_{t'}$ for child nodes t' of t. DP works for *many problems*:

1. Construct a *graph representation* G of $\mathcal{I}$.
2. Compute (some) tree decomposition $\mathcal{T} = (T, \chi)$ of G.
3. Traverse the nodes of T in post-order (bottom-up tree traversal of T). At every node t of T during post-order traversal, execute a table algorithm that takes as input bag $\chi(t)$, a certain *bag instance* $\mathcal{I}_t$ depending on the problem, as well as previously computed child tables of t. Then, the results of this execution are stored in table τ_t.
4. Finally, interpret table τ_n for the root node n of T in order to *output the solution* to the problem for instance $\mathcal{I}$.

Having relational algebra and this paradigm at hand, we exemplarily show how to solve #SAT, required for solving #∃SAT later. To this end, we need the following graph representation for a given formula φ. The *primal graph* G_φ [44]

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 a TD $\mathcal{T} = (T, \chi)$ of G_φ and a node t of T. Then, we let bag instance φ_t of φ, called *bag formula*, be the clauses $\{\, c \mid c \in \varphi, \mathrm{var}(c) \subseteq \chi(t) \,\}$ entirely covered by the bag $\chi(t)$.

Now, the only ingredient that is still missing for solving #Sat via dynamic programming along a given TD, is the table algorithm #Sat$_t$. For brevity, table algorithm #Sat$_t$ as presented in Listing 1 shows the four cases corresponding to the four node types of a nice TD, as any TD node forms just an overlap of these four cases. Each table τ_t consists of rows using attributes $\chi(t) \cup \{\mathrm{cnt}\}$, representing an assignment of φ_t and cnt is a counter. Then, the table τ_t for a leaf node t, where $\mathrm{type}(t) = \mathit{leaf}$, consists of the empty assignment and counter 1, cf., Line 1. For nodes t with introduced variable $a \in \chi(t)$, we guess in Line 3 for each assignment 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 a remove node t, we project assignments of child tables to $\chi(t)$, cf., Line 5, and sum up counters of the same assignments. For join nodes, counters of equal assignments are multiplied (Line 7).

Example 2. Let $\varphi := \{\overbrace{\{\neg x, y, a\}}^{c_1}, \overbrace{\{x, \neg y, \neg a\}}^{c_2}, \overbrace{\{x, b\}}^{c_3}, \overbrace{\{x, \neg b\}}^{c_4}\}$. Observe that G of Fig. 1 is the primal graph G_φ and that there are 6 satisfying assignments of φ. We discuss selected cases of running algorithm #Sat$_t$ on each node t of TD $\mathcal{T}_{\mathrm{nice}}$ of Fig. 2 in post-order, thereby evaluating φ in parts. Observe that $\mathrm{type}(t_1) = \mathit{leaf}$. Consequently, $\tau_1 = \{\{(\mathrm{cnt}, 1)\}\}$, cf., Line 1. Nodes $t \in \{t_2, t_3, t_4\}$ are of $\mathrm{type}(t) = \mathit{intr}$. Thus, we cross-join table τ_1 with $\{\{(x, 0)\}, \{(x, 1)\}\}$ (two possible truth assignments for x), cf., Line 3, which is cross-joined with $\{\{(a, 0)\}, \{(a, 1)\}\}$, and then with $\{\{(y, 0)\}, \{(y, 1)\}\}$. Then, for node t_4 we additionally filter, cf., Line 3, those rows, where $\varphi_{t_4} = \{c_1, c_2\}$ is satisfied and obtain table τ_4. Node t_5 is of $\mathrm{type}(t_5) = \mathit{rem}$, where a is removed, i.e., by the properties of TDs, it is guaranteed that all clauses of φ using a are checked below t_5 and that no clause involving a will occur above t_4. Consequently, τ_5 is obtained from τ_4 by projecting to $\{x, y\}$ and summing up the counters cnt of rows of equal assignments correspondingly, cf., Line 5. Similarly, one proceeds with τ_6 and the right part of the tree, obtaining tables $\tau_7 - \tau_{10}$. In node t_{11}, we join common assignments of tables τ_6 and τ_{10}, and multiply counters cnt accordingly. Finally, we obtain 6 satisfying assignments, as expected. In all the tables the corresponding parts of assignment I, where x, y, b are set to 1 and a is set to 0 are highlighted.

Although these tables obtained via table algorithms might be exponential in size, the size is bounded by the width of the given TD of the primal graph G_φ. Still, practical results of such algorithms show competitive behaviour [3,25] up to a certain width. As a result, instances with high (tree-)width seem out of reach. Even further, if we lift the table algorithm #Sat$_t$ in order to solve problem #∃Sat, we are double exponential in the treewidth [23] and suffer from a rather complicated algorithm. To mitigate these issues, we present a novel approach to deal with high treewidth, by nesting of DP on abstractions of G_φ. As we will see, this not only works for #Sat, but also for #∃Sat with adaptions.

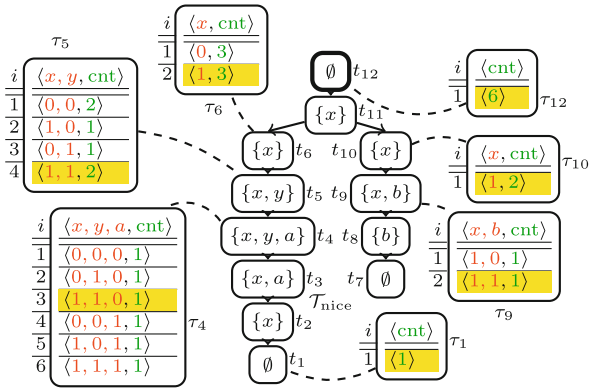

Fig. 2. Tables obtained by $\#\text{SAT}_t$ on $\mathcal{T}_{\text{nice}}$ for φ of Example 2.

3.1 Essentials for Nested Dynamic Programming

Assume that a set U of variables of φ, called *nesting variables*, appears *uniquely* in one TD node t. Then, one could do DP on the TD as before, but no truth value for any variable in U is stored. Instead, clauses involving U could be evaluated by nested DP within node t, since variables U appear uniquely in t. Indeed, for DP on the other (non-nesting) variables, only the result of this evaluation is essential. Now, before we can apply nested DP, we need abstractions with room for choosing nesting variables between the empty set and the set of all the variables. Inspired by related work [16,20,26,29], we define the *nested primal graph* N_φ^A for a given formula φ and a given set $A \subseteq \text{var}(\varphi)$ of *abstraction variables*. To this end, we say a path P in primal graph G_φ is a *nesting path (between u and v)* using A, if $P = u, v_1, \ldots, v_\ell, v$ ($\ell \geq 0$), and every vertex v_i is a *nesting variable*, i.e., $v_i \notin A$ for $1 \leq i \leq \ell$. Note that any path in G_φ is nesting using A if $A = \emptyset$. Then, the vertices of nested primal graph N_φ^A correspond to A and there is an edge between two vertices $u, v \in A$ if there is a nesting path between u and v. Observe that the nested primal graph only consists of abstraction variables and, intuitively, "hides" nesting variables in nesting paths of primal graph G_φ.

Example 3. Recall formula φ and primal graph G_φ of Example 2. Given abstraction variables $A = \{x, y\}$, nesting paths of G_φ are, e.g., $P_1 = x$, $P_2 = x, b$, $P_3 = b, x$, $P_4 = x, y$, $P_5 = x, a, y$. However, neither path $P_6 = y, x, b$, nor path $P_7 = b, x, y, a$ is nesting using A. Nested primal graph N_φ^A contains edge $\{x, y\}$ over vertices A due to paths P_4, P_5.

The nested primal graph provides abstractions of needed flexibility for nested DP. Indeed, if we set abstraction variables to $A = \text{var}(\varphi)$, we end up with full DP and zero nesting, whereas setting $A = \emptyset$ results in full nesting, i.e., nesting of all variables. Intuitively, the nested primal graph ensures that clauses subject to nesting (containing nesting variables) can be safely evaluated in exactly one node of a TD of the nested primal graph. To formalize this, we let *nestReach*(U)

Listing 2: Algorithm $\texttt{HybDP}_{\#\exists\text{SAT}}(\text{depth}, \varphi, P', A')$ for hybrid solving of $\#\exists$SAT by nested DP with abstraction variables A'.

In: Nesting depth ≥ 0, formula φ, projection variables $P' \subseteq \text{var}(\varphi)$, and abstraction variables $A' \subseteq \text{var}(\varphi)$.
Out: Number $\#\exists\text{SAT}(\varphi, P')$ of assignments.

```
1  φ, P ← BCP_And_Preprocessing(φ, P')
2  A ← A' ∩ P
3  if φ ∈ dom(cache)   /*Cache Hit occurred*/      then return cache(φ) · 2^|P'\P|
4  if var(φ) ∩ P = ∅ then return SAT(φ) · 2^|P'\P|
5  (T, χ) ← Decompose_via_Heuristics(N_φ^A)      /* Decompose */
6  width ← max_{t in T} |χ(t)| − 1
7  if width ≥ threshold_hybrid or depth ≥ threshold_depth /* Standard Solver */ then
8  |  if var(φ) = P then cache ← cache ∪ {(φ, #SAT(φ))}
9  |  else                cache ← cache ∪ {(φ, #∃SAT(φ, P))}
10 |  return cache(φ) · 2^|P'\P|
11 if width ≥ threshold_abstr /* Abstract via Heuristics & Decompose */ then
12 |  A      ← Choose_Subset_via_Heuristics(A, φ)
13 |  (T, χ) ← Decompose_via_Heuristics(N_φ^A)
14 n ← root(T)
15 τ ← {}      /* empty mapping */
16 for iterate t in post-order(T, n) /* Nested Dynamic Programming */      do
17 |  {t_1, ..., t_ℓ} ← children(T, t)
18 |  τ_t ← #∃SAT_t(depth, χ(t), φ_t, P, φ_t^A, A' \ A, ⟨τ_{t_1}, ..., τ_{t_ℓ}⟩)
19 cache ← cache ∪ {(φ, c)}   where Π_cnt(τ_n) = {{(cnt, c)}}
20 return cache(φ) · 2^|P'\P|
```

for any set $U \subseteq \text{var}(\varphi)$ of variables containing nesting variables ($U \not\subseteq A$), be the set of vertices of all nesting paths of G_φ between vertices a, b using A such that (i) both $a, b \in U$, or (ii) $a \in U \setminus A$. Intuitively, this definition ensures that from a given set U of variables, we obtain reachable (i) nesting and (ii) abstraction variables, needed to evaluate clauses over U. Then, assuming a TD $\mathcal{T}$ of N_φ^A, we say a set $U \subseteq \text{var}(\varphi)$ of variables ("*compatible set*") is *compatible* with a node t of $\mathcal{T}$, and vice versa, if (I) $U = \mathit{nestReach}(U)$, and (II) $U \cap A \subseteq \chi(t)$.

Example 4. Assume again formula φ, primal graph G_φ and abstraction variables $A = \{x, y\}$ of the previous example. Further, consider any TD (T, χ) of N_φ^A. Observe that $\mathit{nestReach}(\{b\}) = \{b, x\}$ due to nesting path b, x, i.e., $\{b\}$ is not a compatible set. However, $\{b, x\}$ is compatible with any node t of T where $x \in \chi(t)$. Indeed, to evaluate clauses $c_3, c_4 \in \varphi$, we need to evaluate both b and x. Similarly, $\{a, x\}$ is not a compatible set due to nesting path a, y, but $\{a, x, y\}$ is a compatible set. Also, $\{a, b, x, y\}$ is a compatible set.

By construction any nesting variable is in at least one compatible set. However, (1) a nesting variable could be even in several compatible sets, and (2) a compatible set could be compatible with several nodes of $\mathcal{T}$. Hence, to allow nested evaluation, we need to ensure that each nesting variable is evaluated only in one unique node t. As a result, we formalize for every compatible set U that is

Listing 3: Nested table algorithm $\#\exists\text{SAT}_t(\text{depth}, \chi_t, \varphi_t, P, \varphi_t^A, A', \langle \tau_1, \ldots, \tau_\ell \rangle)$ for solving $\#\exists$SAT on node t of a nice TD.

In: Nesting depth ≥ 0, bag χ_t, bag formula φ_t, projection variables P, nested bag formula φ_t^A, abstraction variables A', and child tables $\langle \tau_1, \ldots \tau_\ell \rangle$ of t.
Out: Table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t \leftarrow \{\{(\text{cnt}, 1)\}\}$
2 **else if** type$(t) =$ *intr, and* $a \in \chi_t$ *is introduced* **then**
3 $\quad \tau_t \leftarrow \tau_1 \bowtie_{\varphi_t} \{\{(a,0)\},\{(a,1)\}\}$
4 $\quad \tau_t \leftarrow \sigma_{\text{cnt}>0}(\dot{\Pi}_{\chi_t, \{\text{cnt} \leftarrow \text{cnt} \cdot \texttt{HybDP}_{\#\exists\text{SAT}}(\text{depth}+1,\, \varphi_t^A[\text{ass}],\, P \cap \text{var}(\varphi_t^A[\text{ass}]),\, A')\}} \tau_t)$
5 **else if** type$(t) =$ *rem, and* $a \notin \chi_t$ *is removed* **then**
6 $\quad \tau_t \leftarrow {}_{\chi_t}G_{\text{cnt} \leftarrow \texttt{SUM}(\text{cnt})}(\Pi_{\text{att}(\tau_1) \setminus \{a\}} \tau_1)$
7 **else if** type$(t) =$ *join* **then**
8 $\quad \tau_t \leftarrow \dot{\Pi}_{\chi_t, \{\text{cnt} \leftarrow \text{cnt} \cdot \text{cnt}'\}}(\tau_1 \bowtie_{\bigwedge_{a \in \chi_t} a = a'} \rho_{\bigcup_{a \in \text{att}(\tau_2)} \{a \mapsto a'\}} \tau_2)$

$\star$) Function ass refers to the respective truth assignment $I \colon \chi_t \to \{0,1\}$ of a given row $r \in \tau_t$.

subset-minimal, a *unique* node t compatible with U, denoted by $\text{comp}(U) := t$. For simplicity of our algorithms, we assume these unique nodes for U are introduce nodes, i.e., type$(t) =$ *intr*. We denote the union of all compatible sets U where $\text{comp}(U) = t$, by *nested bag variables* χ_t^A. Then, the *nested bag formula* φ_t^A for a node t of $\mathcal{T}$ equals $\varphi_t^A := \{c \mid c \in \varphi, \text{var}(c) \subseteq \chi_t^A\} \setminus \varphi_t$, where formula φ_t is defined above.

Example 5. Recall formula φ, TD $\mathcal{T} = (T, \chi)$ of G_φ, and abstraction variables $A = \{x, y\}$ of Example 3. Consider TD $\mathcal{T}' := (T, \chi')$, where $\chi'(t) := \chi(t) \cap \{x, y\}$ for each node t of T. Observe that $\mathcal{T}'$ is $\mathcal{T}$, but restricted to A and that $\mathcal{T}'$ is a TD of N_φ^A of width 1. Observe that only for compatible set $U = \{b, x\}$ we have two nodes compatible with U, namely t_2 and t_3. We assume $\text{comp}(U) = t_2$. Consequently, nested bag formulas are $\varphi_{t_1}^A = \{c_1, c_2\}$, $\varphi_{t_2}^A = \{c_3, c_4\}$, and $\varphi_{t_3}^A = \emptyset$.

Assume any TD $\mathcal{T}$ of N_φ^A using any set A of abstraction variables. Observe that the definitions of nested primal graph and nested bag formula ensure that any set S of vertices connected via edges in G_φ will "appear" among nested bag variables of some node of $\mathcal{T}$. Even more stringent, each variable $a \in \text{var}(\varphi) \setminus A$ appears *only* in nested bag formula φ_t^A of node t unique for a. These unique variable appearances allow to nest evaluating φ_t^A under some assignment to $\chi(t)$.

3.2 Hybrid Solving Based on Nested DP

Now, we have definitions at hand to discuss nested DP in the context of *hybrid solving*, which combines using both standard solvers and parameterized solvers exploiting treewidth. We first illustrate the ideas for the problem $\#\exists$SAT and then discuss possible generalizations in Sect. 3.3; a concrete implementation is presented in Sect. 4.

Listing 2 depicts our algorithm $\texttt{HybDP}_{\#\exists\text{SAT}}$ for solving $\#\exists$SAT. Note that the recursion is indirect in Line 18 through Line 4 of Listing 3 (discussed later).

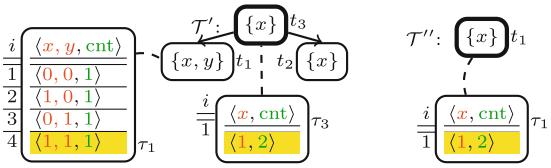

Fig. 3. Selected tables obtained by nested DP on TD $\mathcal{T}'$ of $N_\varphi^{\{x,y\}}$ (left) and on TD $\mathcal{T}''$ of $N_\varphi^{\{x\}}$ (right) for φ and projection variables $P = \{x, y\}$ of Example 6 via $\texttt{HybDP}_{\#\exists\text{SAT}_t}$.

Algorithm $\texttt{HybDP}_{\#\exists\text{SAT}}$ takes formula φ, projection variables P' and abstraction variables A'. The algorithm uses a global, but rather naive and simple cache mapping a formula to an integer, and consists of four subsequent blocks of code, separated by empty lines: (1) Preprocessing & Cache Consolidation, (2) Standard Solving, (3) Abstraction & Decomposition, and (4) Nested DP.

Block (1) spans Lines 1–3 and performs Boolean conflict propagation and preprocessing, thereby obtaining projection variables $P \subseteq P'$ (preserving satisfying assignments w.r.t. P'), sets A to $A' \cap P$ in Line 2, and consolidates cache with the updated formula φ. If φ is not cached, we do standard solving if the width is out-of-reach for nested DP in Block (2), spanning Lines 4–10. More concretely, if φ does not contain projection variables, we employ a SAT solver returning integer 1 or 0. If φ contains projection variables and either the width obtained by heuristically decomposing G_φ is above $\text{threshold}_{\text{hybrid}}$, or the nesting depth exceeds $\text{threshold}_{\text{depth}}$, we use a standard #SAT or #∃SAT solver depending on $\text{var}(\varphi) \cap P$. Block (3) spans Lines 11–13 and is reached if no cache entry was found in Block (1) and standard solving was skipped in Block (2). If the width of the computed decomposition is above $\text{threshold}_{\text{abstr}}$, we need to use an abstraction in form of the nested primal graph. This is achieved by choosing suitable subsets $E \subseteq A$ of abstraction variables and decomposing φ_t^E heuristically. Finally, Block (4) concerns nested DP, cf., Lines 14–20. This block relies on nested table algorithm $\#\exists\text{SAT}_t$, which takes parameters similar to table algorithm $\#\text{SAT}_t$, but additionally requires the nested bag formula for current node t, projection variables P and abstraction variables. Nested table algorithm $\#\exists\text{SAT}_t$ is sketched in Listing 3 and recursively calls for each row $r \in \tau_t$, $\texttt{HybDP}_{\#\exists\text{SAT}}$ on nested bag formula φ_t^A simplified by the assignment $\text{ass}(r)$ of the current row r. This is implemented in Line 4 by using extended projection, cf., Listing 1, where the count cnt of the respective row r is updated by multiplying the result of the recursive call $\texttt{HybDP}_{\#\exists\text{SAT}}$. Notably, as the recursive call $\texttt{HybDP}_{\#\exists\text{SAT}}$ within extended projection of Line 4 implicitly takes a given current row r, the function occurrences ass in Line 4 implicitly take this row r as an argument. As a result, our approach deals with high treewidth by recursively finding and decomposing abstractions of the graph. If the treewidth is too high for some parts, TDs of abstractions are used to guide standard solvers.

Example 6. Recall formula φ, set A of abstraction variables, and TD $\mathcal{T}'$ of nested primal graph N_φ^A given in Example 5. Restricted to projection set $P := \{x, y\}$, φ has two satisfying assignments, namely $\{x \mapsto 1, y \mapsto 0\}$ and $\{x \mapsto 1, y \mapsto 1\}$.

Listing 4: Nested table algorithm $\mathrm{QSAT}_t(\mathrm{depth}, \chi_t, \varphi_t, \varphi_t^A, A', \langle \tau_1, \ldots, \tau_\ell \rangle)$ for solving QSAT on node t of a nice tree decomposition.

In: Nesting depth ≥ 0, bag χ_t, bag QBF $\varphi_t = Q\,V.\gamma$, nested bag QBF φ_t^A, abstraction variables A', and child tables $\langle \tau_1, \ldots \tau_\ell \rangle$ of t.
Out: Table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t \leftarrow \{\emptyset\}$
2 **else if** type$(t) =$ *intr, and* $a \in \chi_t$ *is introduced* **then**
3 $\quad \tau_t \leftarrow \tau_1 \bowtie_{\varphi_t} \{\{(a,0)\},\{(a,1)\}\}$
4 $\quad \tau_t \leftarrow \sigma_{(Q=\exists \vee |\tau_t|=2^{|\chi_t|}) \wedge \mathtt{HybDP}_{\mathrm{QSAT}}(\mathrm{depth}+1, \varphi_t^A[\mathrm{ass}], A')}(\tau_t)$
5 **else if** type$(t) =$ *rem, and* $a \notin \chi_t$ *is removed* **then**
6 $\quad \tau_t \leftarrow \Pi_{\mathrm{att}(\tau_1) \setminus \{a\}} \tau_1$
7 **else if** type$(t) =$ *join* **then**
8 $\quad \tau_t \leftarrow \Pi_{\chi_t}(\tau_1 \bowtie_{\bigwedge_{a \in \chi_t} a=a'} \rho_{\bigcup_{a \in \mathrm{att}(\tau_2)} \{a \mapsto a'\}} \tau_2)$

$\star$) The cardinality of a table τ can be obtained via relational algebra (sub-expression): $|\tau| := c$, where $\{\{(\mathrm{card}, c)\}\} = {}_{\emptyset}G_{\mathrm{card} \leftarrow \mathtt{SUM}(1)}\tau$

Consequently, the solution to #∃SAT is 2. Figure 3 (left) shows TD $\mathcal{T}'$ of N_φ^A and tables obtained by $\mathtt{HybDP}_{\#\exists\mathrm{SAT}_t}(\varphi, P, A)$ for solving projected model counting on φ and P. Note that the same example easily works for #SAT, where $P = \mathrm{var}(\varphi)$.

Algorithm $\#\exists\mathrm{SAT}_t$ of Listing 3 works similar to algorithm $\#\mathrm{SAT}_t$, but uses attribute "cnt" for storing (projected) counts accordingly. We briefly discuss executing $\#\exists\mathrm{SAT}_{t_1}$ in the context of Line 18 of algorithm $\mathtt{HybDP}_{\#\exists\mathrm{SAT}_t}$ on node t_1 of $\mathcal{T}'$, resulting in table τ_1 as shown in Fig. 3 (left). Recall that $\mathrm{comp}(\{a,x,y\}) = t_1$, and, consequently, $\varphi_{t_1}^A = \{\{\neg x, y, a\}, \{x, \neg y, \neg a\}\}$. Then, in Line 4 of algorithm $\#\exists\mathrm{SAT}_t$, for each assignment $\mathrm{ass}(r)$ to $\{x,y\}$ of each row r of τ_1, we compute $\mathtt{HybDP}_{\#\exists\mathrm{SAT}_t}(\psi, P \cap \mathrm{var}(\psi), \emptyset)$ using $\psi = \varphi_{t_1}^A[\mathrm{ass}(r)]$. Each of these recursive calls, however, is already solved by BCP and preprocessing, e.g., $\varphi_{t_1}^A[\{x \mapsto 1, y \mapsto 0\}]$ of Row 2 simplifies to $\{\{a\}\}$.

Figure 3 (right) shows TD $\mathcal{T}''$ of N_φ^E with $E := \{x\}$, and tables obtained by $\mathtt{HybDP}_{\#\exists\mathrm{SAT}_t}(\varphi, P, E)$. Still, $\varphi_{t_1}^E[\mathrm{ass}(r)]$ for a given assignment $\mathrm{ass}(r) : \{x\} \to \{0,1\}$ of any row $r \in \tau_1$ can be simplified. Concretely, $\varphi_{t_1}^E[\{x \mapsto 0\}]$ evaluates to $\emptyset$ and $\varphi_{t_1}^E[\{x \mapsto 1\}]$ evaluates to two variable-distinct clauses, namely $\{\neg b\}$ and $\{y, a\}$. Thus, there are 2 satisfying assignments $\{y \mapsto 0\}$, $\{y \mapsto 1\}$ of $\varphi_{t_1}^E[\{x \mapsto 1\}]$ restricted to P.

Theorem 1. *Given formula φ, projection variables $P \subseteq \mathrm{var}(\varphi)$, and abstraction variables $A' \subseteq \mathrm{var}(\varphi)$. Then, $\mathtt{HybDP}_{\#\exists\mathrm{SAT}}(\varphi, P, A')$ correctly returns $\#\exists\mathrm{SAT}(\varphi, P)$.*

Proof (Sketch). Observe that (A): (T, χ) is a TD of nested primal graph N_φ^A such that $A \subseteq A' \cap P$. The interesting part of algorithm $\mathtt{HybDP}_{\#\exists\mathrm{SAT}}$ lies in Block (3), in particular in Lines 11–13. The proof proceeds by structural induction on φ. By construction, we have (B): Every variable of $\mathrm{var}(\varphi) \setminus A$ occurs in some nested bag formula φ_t^A as used in the call to $\#\exists\mathrm{SAT}_t$ in Line 18 for a unique

node t of T. Observe that $\#\exists\text{SAT}_t$ corresponds to $\#\text{SAT}_t$, whose correctness is established via invariants, cf., [24,44], only Line 4 differs. In Line 4 of $\#\exists\text{SAT}_t$, $\texttt{HybDP}_{\#\exists\text{SAT}}$ is called recursively on subformulas $\varphi_t^A[\text{ass}(r)]$ for each $r \in \tau_t$. By induction hypothesis, we have (C): these calls result to $\#\exists\text{SAT}(\varphi_t^A[\text{ass}(r)], P \cap \text{var}(\varphi_t^A[\text{ass}(r)]))$ for each $r \in \tau_t$. By (A), $\#\exists\text{SAT}_t$ as called in Line 18 stores only table attributes in $\chi_t \subseteq A \subseteq P$. Thus, by (C), recursive calls can be subsequently multiplied to cnt for each $r \in \tau_t$.

3.3 Generalizing Nested DP to Other Formalisms

Nested DP as proposed above is by far not restricted to (projected) model counting, or counting problems in general. In fact, one can easily generalize nested DP to other relevant formalisms, briefly sketched for the QBF formalism.

Quantified Boolean Formulas (QBFs). We assume QBFs of the form $\varphi = \exists V_1.\forall V_2 \ldots \exists V_\ell.\gamma$ using *quantifiers* $\exists, \forall$, where γ is a CNF formula and $\text{var}(\varphi) = \text{var}(\gamma) = V_1 \cup V_2 \cdots \cup V_\ell$. Given QBF $\varphi = Q\ V.\psi$ with $Q \in \{\exists, \forall\}$, we let $\text{qvar}(\varphi) := V$. For an assignment $I : V' \to \{0,1\}$ with $V' \subseteq V$, we let $\varphi[I] := \psi[I]$ if $V' = V$, and $\varphi[I] := Q(V \setminus V').\psi[I]$ if $V' \subsetneq V$. *Validity* of φ (QSAT) is recursively defined: $\exists V.\varphi$ is *valid* if there is $I\colon V \to \{0,1\}$ where $\varphi[I]$ is valid; $\forall V.\varphi$ is valid if for every $I\colon V \to \{0,1\}$, $\varphi[I]$ is valid.

Hybrid solving by nested DP can be extended to problem QSAT. To the end of using this approach for QBFs, we define the primal graph G_φ for a QBF $\varphi = \exists V_1.\forall V_2 \ldots \exists V_\ell.\gamma$ analogously to the primal graph of a Boolean formula, i.e., $G_\varphi := G_\gamma$. Consequently, also the nested primal graph is defined for a given set $A \subseteq \text{var}(\varphi)$ by $N_\varphi^A := N_\gamma^A$. Now, let $A \subseteq \text{var}(\varphi)$ be a set of abstraction variables, and $\mathcal{T} = (T, \chi)$ be a TD of N_φ^A and t be a node of T. Then, the *bag QBF* φ_t is given by $\varphi_t := \exists V_1.\forall V_2 \ldots \exists V_\ell.\gamma_t$ and the *nested bag QBF* φ_t^A for a set $A \subseteq \text{var}(\varphi)$ amounts to $\varphi_t^A := \exists V_1.\forall V_2 \ldots \exists V_\ell.\gamma_t^A$.

Algorithm $\texttt{HybDP}_{\text{QSAT}}$ is similar to $\texttt{HybDP}_{\#\exists\text{SAT}}$ of Listing 2, where the projection variables parameter P' is removed since P' constantly coincides with variables $\text{qvar}(\varphi)$ of the outermost quantifier. Further, Line 4 is removed, Lines 8 and 9 are replaced by calling a QSAT solver and nested table algorithm $\#\exists\text{SAT}_t$ of Line 18 is replaced by nested table algorithm QSAT_t as presented in Listing 4. Algorithm QSAT_t is of similar shape as algorithm $\#\exists\text{SAT}_t$, cf., Listing 3, but does not maintain counts cnt. Further, Line 4 of algorithm QSAT_t intuitively filters τ_t fulfilling the outer-most quantifier, and keeps those rows r of τ_t, where the recursive call to $\texttt{HybDP}_{\text{QSAT}}$ on nested bag formula simplified by the assignment $\text{ass}(r)$ of r succeeds. For ensuring that the outer-most quantifier Q is fulfilled, we are either in the situation that $Q = \exists$, which immediately is fulfilled for every row r in τ_t since r itself serves as a witness. If $Q = \forall$, we need to check that τ_t contains $2^{|\chi(t)|}$ many (all) rows. The cardinality of table τ_t can be computed via a sub-expression of relational algebra as hinted in the footnote of Listing 4. Notably, if $Q = \forall$, we do not need to check in Line 8 of Listing 4, whether all rows sustain in table τ_t since this is already ensured for both child tables τ_1, τ_2 of t. Then, if in the end the table for the root node of $\mathcal{T}$ is not empty,

it is guaranteed that either the table contains some (if $Q = \exists$) or all (if $Q = \forall$) rows and that φ is valid. Note that algorithm QSAT$_t$ can be extended to also consider more fine-grained quantifier dependency schemes.

Compared to other algorithms for QSAT using treewidth [9,10], hybrid solving based on nested DP is quite compact without the need of nested tables. Instead of rather involved data structures (nested tables), we use here plain tables that can be handled by modern database systems efficiently.

4 Implementation and Preliminary Results

We implemented a hybrid solver nestHDB[1] based on nested DP in Python3 and using table manipulation techniques by means of SQL and the *database management system (DBMS)* Postgres. Our solver builds upon the recently published prototype dpdb [24], which applied a DBMS for plain dynamic programming algorithms. However, we used the most-recent version 12 of Postgres and we let it operate on a tmpfs-ramdisk. In our solver, the DBMS serves the purpose of extremely efficient in-memory table manipulations and query optimization required by nested DP, and therefore nestHDB benefits from database technology.

Nested DP & Choice of Standard Solvers. We implemented dedicated nested DP algorithms for solving #SAT and #∃SAT, where we do (nested) DP up to threshold$_{\text{depth}} = 2$. Further, we set threshold$_{\text{hybrid}} = 1000$ and therefore we do not "fall back" to standard solvers based on the width (cf., Line 7 of Listing 2), but based on the nesting depth.

Also, the evaluation of the nested bag formula is "shifted" to the database if it uses at most 40 abstraction variables, since Postgres efficiently handles these small-sized Boolean formulas. Thereby, further nesting is saved by executing optimized SQL statements within the TD nodes. A value of 40 seems to be a nice balance between the overhead caused by standard solvers for small formulas and exponential growth counteracting the advantages of the DBMS. For hybrid solving, we use #SAT solver sharpSAT [48] and for #∃SAT we employ the recently published #∃SAT solver projMC [35], solver sharpSAT and SAT solver picosat [4]. Observe that our solver immediately benefits from better standard solvers and further improvements of the solvers above.

Choosing Non-nesting Variables & Compatible Nodes. TDs are computed by means of heuristics via decomposition library htd [1]. For finding good abstractions (crucial), i.e., abstraction variables for the nested primal graph, we use encodings for solver clingo [27], which is based on logic programming (ASP) and therefore perfectly suited for solving reachability via nesting paths. There, among a reasonably sized subset of vertices of smallest degree, we aim for a preferably large (maximal) set A of abstraction variables such that at the same time the resulting graph N_φ^A is reasonably sparse, which is achieved by minimizing the number of edges of N_φ^A. To this end, we use built-in (cost) optimization,

[1] Source code, instances, and detailed results are available at: tinyurl.com/nesthdb.

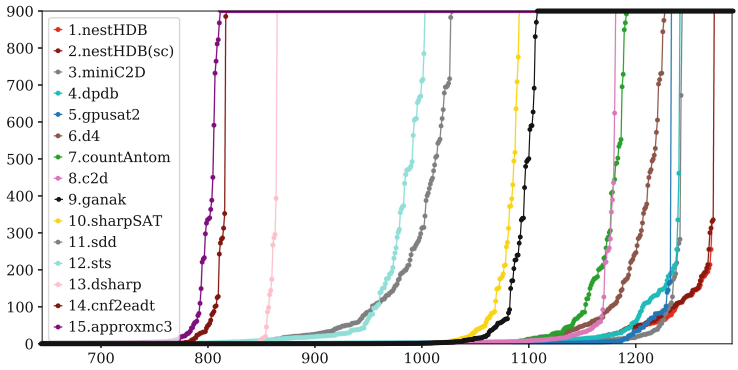

Fig. 4. Cactus plot of instances for #SAT, where instances (x-axis) are ordered for each solver individually by runtime[seconds] (y-axis). $\text{threshold}_{\text{abstr}} = 38$.

where we take the best results obtained by clingo after running at most 35 s. For the concrete encodings used in nestHDB, we refer to the online repository as stated above. We expect that this initial approach can be improved and that extending by problem-specific as well as domain-specific information might help in choosing promising abstraction variables A.

As rows of tables during (nested) DP can be independently computed and parallelized [25], hybrid solver nestHDB potentially calls standard solvers for solving subproblems in parallel using a thread pool. Thereby, the uniquely compatible node for relevant compatible sets U, as denoted in this paper by $\text{comp}(U)$, is decided during runtime among compatible nodes on a first-come-first-serve basis.

Benchmarked Solvers & Instances. We benchmarked nestHDB and 16 other publicly available #SAT solvers on 1,494 instances recently considered [24]. Among those solvers are single-core solvers miniC2D [41], d4 [34], c2d [13], ganak [46], sharpSAT [48], sdd [14], sts [21], dsharp [39], cnf2eadt [32], cachet [45], sharpCDCL [30], approxmc3 [8], and bdd_minisat [49]. We also included multi-core solvers dpdb [24], gpusat2 [25], as well as countAntom [7]. While nestHDB itself is a multi-core solver, we additionally included in our comparison nestHDB(sc), which is nestHDB, but restricted to a single core only. The instances [24] we took are already preprocessed by pmc [33] using recommended options `-vivification -eliminateLit -litImplied -iterate=10 -equiv -orGate -affine` for preserving model counts. However, nestHDB still uses pmc with these options also in Line 1 of Listing 2.

Further, we considered the problem #∃SAT, where we compare solvers projMC [35], clingo [27], ganak [46], nestHDB (see footnote 1), and nestHDB(sc) on 610 publicly available instances[2] from projMC (consisting of 15 *planning*, 60 *circuit*, and 100 *random* instances) and Fremont, with 170 *symbolic-markov* applications, and 265 *misc* instances. For preprocessing in Line 1 of Listing 2,

[2] Sources: tinyurl.com/projmc;tinyurl.com/pmc-fremont-01-2020.

benchmark set	solver	tw upper bound max	0-30	31-50	>50	Σ	time [h]
planning	nestHDB	**30**	**7**	0	0	**7**	**2.88**
	nestHDB(sc)	**30**	**7**	0	0	**7**	3.31
	projMC	26	6	0	0	6	3.01
	ganak	19	5	0	0	5	3.36
	clingo	4	1	0	0	1	4.00
circ	nestHDB	**99**	**34**	**10**	**16**	**60**	2.10
	nestHDB(sc)	**99**	**34**	4	14	52	4.60
	projMC	91	28	**10**	11	49	6.23
	ganak	**99**	**34**	**10**	**16**	**60**	**1.21**
	clingo	**99**	31	**10**	**16**	57	4.44
random	nestHDB	79	**30**	**20**	**17**	**67**	**10.91**
	nestHDB(sc)	79	**30**	**20**	15	65	11.29
	projMC	**84**	**30**	**20**	15	65	11.09
	ganak	19	19	0	0	19	23.18
	clingo	24	25	0	0	25	21.38
markov	nestHDB	23	62	0	0	62	31.98
	nestHDB(sc)	23	61	0	0	61	32.54
	projMC	8	54	0	0	54	33.65
	ganak	**59**	**64**	0	**4**	**68**	**30.32**
	clingo	3	38	0	0	38	37.54
misc	nestHDB	47	**38**	**17**	0	**55**	46.12
	nestHDB(sc)	47	**38**	13	0	51	48.20
	projMC	47	**38**	13	0	51	45.90
	ganak	44	**38**	15	0	53	**45.72**
	clingo	**63**	**38**	15	**1**	54	44.79
Σ	nestHDB	**99**	**171**	**47**	**33**	**251**	**93.99**
	nestHDB(sc)	**99**	170	37	29	236	99.95
	projMC	91	156	43	26	225	99.88
	ganak	**99**	160	25	20	205	103.78
	clingo	**99**	133	25	17	175	112.15

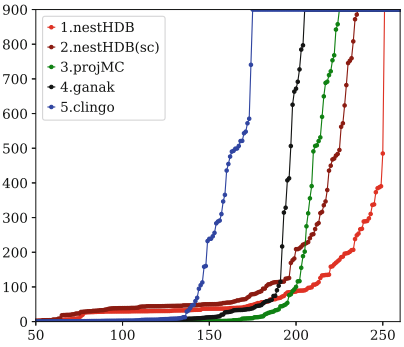

Fig. 5. Number of solved #∃Sat insts., grouped by upper bound intervals of treewidth (left), cactus plot (right). time[h] is cumulated wall clock time, timeouts count as 900 s. $\text{threshold}_{\text{abstr}} = 8$.

nestHDB uses pmc as before, but without options `-equiv -orGate -affine` to ensure preservation of models (equivalence).

Benchmark Setup. Solvers ran on a cluster of 12 nodes. Each node of the cluster is equipped with two Intel Xeon E5-2650 CPUs consisting of 12 physical cores each at 2.2 GHz clock speed, 256 GB RAM. For dpdb and nestHDB, we used Postgres 12 on a tmpfs-ramdisk (/tmp) that could grow up to at most 1 GB per run. Results were gathered on Ubuntu 16.04.1 LTS machines with disabled hyperthreading on kernel 4.4.0-139. We mainly compare total wall clock time and number of timeouts. For parallel solvers (dpdb, countAntom, nestHDB) we allow 12 physical cores. Timeout is 900 s and RAM is limited to 16 GB per instance and solver. Results for gpusat2 are taken from [24].

Benchmark Results. The results for #Sat showing the best 14 solvers are summarized in the cactus plot of Fig. 4. Overall it shows nestHDB among the best solvers, solving 1,273 instances. The reason for this is, compared to dpdb, that nestHDB can solve instances using TDs of primal graphs of widths larger than 44, up to width 266. This limit is even slightly larger than the width of 264 that sharpSAT on its own can handle. We also tried using minic2d instead

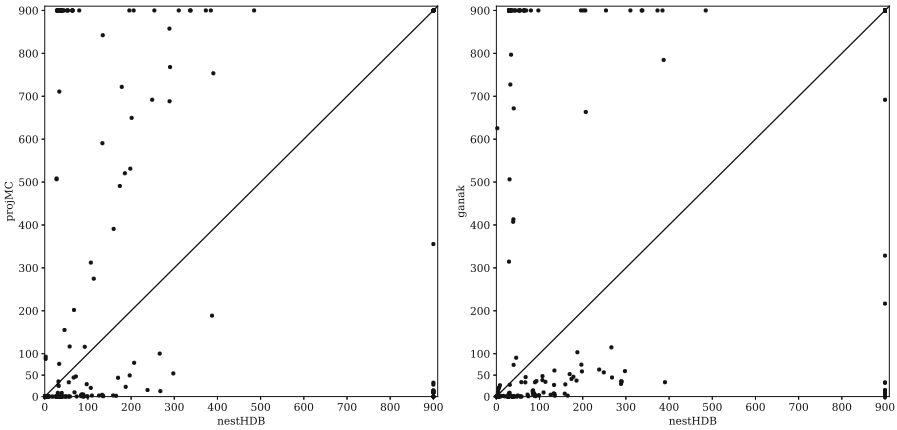

Fig. 6. Scatter plot of instances for #∃SAT, where the x-axis shows runtime in seconds of nestHDB compared to the y-axis showing runtime of projMC (left) and of ganak (right). $\text{threshold}_{\text{abstr}} = 8$.

of sharpSAT as standard solver for solvers nestHDB and nestHDB(sc), but we could only solve one instance more. Notably, nestHDB(sc) has about the same performance as nestHDB, indicating that parallelism does not help much on the instances. Further, we observed that the employed simple cache as used in Listing 2, provides only a marginal improvement.

Figure 5 (left) depicts a table of results on #∃SAT, where we observe that nestHDB does a good job on instances with low widths below $\text{threshold}_{\text{abstr}} = 8$ (containing ideas of dpdb), but also on widths well above 8 (using nested DP). Notably, nestHDB is also competitive on widths well above 50. Indeed, nestHDB and nestHDB(sc) perform well on all benchmark sets, whereas on some sets the solvers projMC, clingo and ganak are faster. Overall, parallelism provides a significant improvement here, but still nestHDB(sc) shows competitive performance, which is also visualized in the cactus plot of Fig. 5 (right). Figure 6 shows scatter plots comparing nestHDB to projMC (left) and to ganak (right). Overall, both plots show that nestHDB solves more instances, since in both cases the y-axis shows more black dots at 900 s than the x-axis. Further, the bottom left of both plots shows that there are plenty easy instances that can be solved by projMC and ganak in well below 50 s, where nestHDB needs up to 200 s. Similarly, the cactus plot given in Fig. 5 (right) shows that nestHDB can have some overhead compared to the three standard solvers, which is not surprising. This indicates that there is still room for improvement if, e.g., easy instances are easily detected, and if standard solvers are used for those instances. Alternatively, one could also just run a standard solver for at most 50 s and if not solved within 50 s, the heavier machinery of nested dynamic programming is invoked. Apart from these instances, Fig. 6 shows that nestHDB solves harder instances faster, where standard solvers struggle.

5 Conclusion

We presented nested dynamic programming (nested DP) using different levels of abstractions, which are subsequently refined and solved recursively. This approach is complemented with hybrid solving, where (search-intense) subproblems are solved by standard solvers. We provided nested DP algorithms for problems related to Boolean satisfiability, but the idea can be easily applied for other formalisms. We implemented some of these algorithms and our benchmark results are promising. For future work, we plan deeper studies of problem-specific abstractions, in particular for QSAT. We want to further tune our solver parameters (e.g., thresholds, timeouts, sizes), deepen interleaving with solvers like projMC, and to use incremental solving for obtaining abstractions and evaluating nested bag formulas, where intermediate solver references are kept during dynamic programming and formulas are iteratively added and (re-)solved.

Acknowledgements. The work has been supported by the Austrian Science Fund (FWF), Grants Y698, and P32830, as well as the Vienna Science and Technology Fund, Grant WWTF ICT19-065.

References

1. Abseher, M., Musliu, N., Woltran, S.: htd – a free, open-source framework for (customized) tree decompositions and beyond. In: Salvagnin, D., Lombardi, M. (eds.) CPAIOR 2017. LNCS, vol. 10335, pp. 376–386. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59776-8_30
2. Atserias, A., Fichte, J.K., Thurley, M.: Clause-learning algorithms with many restarts and bounded-width resolution. J. Artif. Intell. Res. **40**, 353–373 (2011)
3. Bannach, M., Berndt, S.: Practical access to dynamic programming on tree decompositions. Algorithms **12**(8), 172 (2019)
4. Biere, A.: PicoSAT essentials. JSAT **4**(2–4), 75–97 (2008)
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. Bodlaender, H., Koster, A.: Combinatorial optimization on graphs of bounded treewidth. Comput. J. **51**(3), 255–269 (2008)
7. Burchard, J., Schubert, T., Becker, B.: Laissez-faire caching for parallel #SAT solving. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 46–61. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_5
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: AAAI 2014, pp. 1722–1730. The AAAI Press (2014)
9. Charwat, G., Woltran, S.: Expansion-based QBF solving on tree decompositions. Fundam. Inform. **167**(1–2), 59–92 (2019)
10. Chen, H.: Quantified constraint satisfaction and bounded treewidth. In: ECAI 2004, pp. 161–170. IOS Press (2004)
11. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (1970)

12. Cygan, M., et al.: Parameterized Algorithms. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21275-3
13. Darwiche, A.: New advances in compiling CNF to decomposable negation normal form. In: ECAI 2004, pp. 318–322. IOS Press (2004)
14. Darwiche, A.: SDD: a new canonical representation of propositional knowledge bases. In: IJCAI 2011, pp. 819–826. AAAI Press/IJCAI (2011)
15. Dell, H., Komusiewicz, C., Talmon, N., Weller, M.: The PACE 2017 parameterized algorithms and computational experiments challenge: the second iteration. In: IPEC 2017, pp. 30:1–30:13. LIPIcs, Dagstuhl Publishing (2017)
16. Dell, H., Roth, M., Wellnitz, P.: Counting answers to existential questions. In: ICALP 2019. LIPIcs, vol. 132, pp. 113:1–113:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)
17. Diestel, R.: Graph Theory, Graduate Texts in Mathematics, vol. 173, 4th edn. Springer, Heidelberg (2012)
18. Downey, R.G., Fellows, M.R.: Fundamentals of Parameterized Complexity. TCS. Springer, London (2013). https://doi.org/10.1007/978-1-4471-5559-1
19. Durand, A., Hermann, M., Kolaitis, P.G.: Subtractive reductions and complete problems for counting complexity classes. Theoret. Comput. Sci. **340**(3), 496–513 (2005). https://doi.org/10.1016/j.tcs.2005.03.012
20. Eiben, E., Ganian, R., Hamm, T., Kwon, O.: Measuring what matters: a hybrid approach to dynamic programming with treewidth. In: MFCS 2019. LIPIcs, vol. 138, pp. 42:1–42:15. Dagstuhl Publishing (2019)
21. Ermon, S., Gomes, C.P., Selman, B.: Uniform solution sampling using a constraint solver as an oracle. In: UAI 2012, pp. 255–264. AUAI Press (2012)
22. Fichte, J.K., Hecher, M., Morak, M., Woltran, S.: Answer set solving with bounded treewidth revisited. In: Balduccini, M., Janhunen, T. (eds.) LPNMR 2017. LNCS (LNAI), vol. 10377, pp. 132–145. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61660-5_13
23. 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.) SAT 2018. LNCS, vol. 10929, pp. 165–184. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94144-8_11
24. Fichte, J.K., Hecher, M., Thier, P., Woltran, S.: Exploiting database management systems and treewidth for counting. In: Komendantskaya, E., Liu, Y.A. (eds.) PADL 2020. LNCS, vol. 12007, pp. 151–167. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-39197-3_10
25. Fichte, J.K., Hecher, M., Zisser, M.: An improved GPU-based SAT model counter. In: Schiex, T., de Givry, S. (eds.) CP 2019. LNCS, vol. 11802, pp. 491–509. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30048-7_29
26. Ganian, R., Ramanujan, M.S., Szeider, S.: Combining treewidth and backdoors for CSP. In: STACS 2017, pp. 36:1–36:17 (2017). https://doi.org/10.4230/LIPIcs.STACS.2017.36
27. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot ASP solving with clingo. TPLP **19**(1), 27–82 (2019). https://doi.org/10.1017/S1471068418000054
28. Giunchiglia, E., Marin, P., Narizzano, M.: Reasoning with quantified Boolean formulas. In: Handbook of Satisfiability, FAIA, vol. 185, pp. 761–780. IOS Press (2009). https://doi.org/10.3233/978-1-58603-929-5-761
29. Hecher, M., Morak, M., Woltran, S.: Structural decompositions of epistemic logic programs. CoRR abs/2001.04219 (2020). http://arxiv.org/abs/2001.04219

30. Klebanov, V., Manthey, N., Muise, C.: SAT-based analysis and quantification of information flow in programs. In: Joshi, K., Siegle, M., Stoelinga, M., D'Argenio, P.R. (eds.) QEST 2013. LNCS, vol. 8054, pp. 177–192. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40196-1_16
31. Kloks, T. (ed.): Treewidth: Computations and Approximations. LNCS, vol. 842. Springer, Heidelberg (1994). https://doi.org/10.1007/BFb0045375
32. Koriche, F., Lagniez, J.M., Marquis, P., Thomas, S.: Knowledge compilation for model counting: affine decision trees. In: IJCAI 2013. The AAAI Press (2013)
33. Lagniez, J., Marquis, P.: Preprocessing for propositional model counting. In: AAAI 2014, pp. 2688–2694. AAAI Press (2014)
34. Lagniez, J.M., Marquis, P.: An improved decision-DDNF compiler. In: IJCAI 2017, pp. 667–673. The AAAI Press (2017)
35. Lagniez, J., Marquis, P.: A recursive algorithm for projected model counting. In: AAAI 2019, pp. 1536–1543. AAAI Press (2019)
36. Langer, A., Reidl, F., Rossmanith, P., Sikdar, S.: Evaluation of an MSO-solver. In: ALENEX 2012, pp. 55–63. SIAM/Omnipress (2012)
37. Lonsing, F., Egly, U.: Evaluating QBF solvers: quantifier alternations matter. In: Hooker, J. (ed.) CP 2018. LNCS, vol. 11008, pp. 276–294. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98334-9_19
38. Maniu, S., Senellart, P., Jog, S.: An experimental study of the treewidth of real-world graph data (extended version). CoRR abs/1901.06862 (2019). http://arxiv.org/abs/1901.06862
39. Muise, C., McIlraith, S.A., Beck, J.C., Hsu, E.I.: Dsharp: fast d-DNNF compilation with sharpSAT. In: Kosseim, L., Inkpen, D. (eds.) AI 2012. LNCS (LNAI), vol. 7310, pp. 356–361. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30353-1_36
40. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford Lecture Series in Mathematics and Its Applications, vol. 31. OUP, Oxford (2006)
41. Oztok, U., Darwiche, A.: A top-down compiler for sentential decision diagrams. In: IJCAI 2015, pp. 3141–3148. The AAAI Press (2015)
42. Pulina, L., Seidl, M.: The 2016 and 2017 QBF solvers evaluations (QBFEVAL'16 and QBFEVAL'17). Artif. Intell. **274**, 224–248 (2019). https://doi.org/10.1016/j.artint.2019.04.002
43. Robertson, N., Seymour, P.D.: Graph minors II: algorithmic aspects of tree-width. J. Algorithms **7**, 309–322 (1986)
44. Samer, M., Szeider, S.: Algorithms for propositional model counting. J. Discrete Algorithms **8**(1), 50–64 (2010)
45. Sang, T., Bacchus, F., Beame, P., Kautz, H., Pitassi, T.: Combining component caching and clause learning for effective model counting. In: SAT 2004 (2004)
46. Sharma, S., Roy, S., Soos, M., Meel, K.S.: GANAK: a scalable probabilistic exact model counter. In: IJCAI 2019, pp. 1169–1176. ijcai.org (2019)
47. Tamaki, H.: Positive-instance driven dynamic programming for treewidth. J. Comb. Optim. **37**(4), 1283–1311 (2018). https://doi.org/10.1007/s10878-018-0353-z
48. Thurley, M.: sharpSAT – counting models with advanced component caching and implicit BCP. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 424–429. Springer, Heidelberg (2006). https://doi.org/10.1007/11814948_38
49. Toda, T., Soh, T.: Implementing efficient all solutions SAT solvers. ACM J. Exp. Algorithmics 21(1.12) (2015). Special Issue SEA 2014