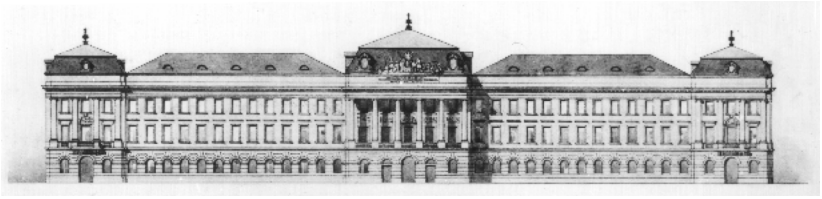
dbai

TECHNICAL REPORT



INSTITUT FÜR INFORMATIONSSYSTEME

ABTEILUNG DATENBANKEN UND ARTIFICIAL INTELLIGENCE

D-FLAT^2: Subset Minimization in Dynamic Programming on Tree Decompositions Made Easy

DBAI-TR-2015-93

Bernhard Bliem **Günther Charwat**
Markus Hecher **Stefan Woltran**

Institut für Informationssysteme
Abteilung Datenbanken und
Artificial Intelligence
Technische Universität Wien
Favoritenstr. 9
A-1040 Vienna, Austria
Tel: +43-1-58801-18403
Fax: +43-1-58801-18493
sekret@dbai.tuwien.ac.at
www.dbai.tuwien.ac.at

DBAI TECHNICAL REPORT
2015

TU WIEN
TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

DBAI TECHNICAL REPORT
DBAI TECHNICAL REPORT DBAI-TR-2015-93, 2015

D-FLATˆ2: Subset Minimization in Dynamic Programming on Tree Decompositions Made Easy

Bernhard Bliem [1] **Günther Charwat** [1] **Markus Hecher** [1]
Stefan Woltran [1]

Abstract. Many problems from the area of AI have been shown tractable for bounded treewidth. In order to put such results into practice, quite involved dynamic programming (DP) algorithms on tree decompositions have to be designed and implemented. These algorithms typically show recurring patterns that call for tasks like subset minimization. In this paper we present a novel approach to obtain such DP algorithms from simpler principles, where the DP formalization of subset minimization is performed automatically. We first give a theoretical account of our novel method, and then present D-FLATˆ2, a system that allows one to specify the core DP algorithm via answer set programming (ASP). We illustrate the approach at work by providing several DP algorithms that are more space-efficient than existing solutions, while featuring improved readability, reuse and therefore maintainability of ASP code. Experiments show that our approach also yields a significant improvement in runtime performance.

[1]TU Wien. E-mail: {bliem,gcharwat,hecher,woltran}@dbai.tuwien.ac.at

Acknowledgements: This work has been supported by the Austrian Science Fund (FWF): Y698, P25607, P25518.

Contents

1 Introduction

Many prominent NP-hard problems in the area of AI have been shown tractable for bounded treewidth. Thanks to Courcelle's theorem [9], it is sufficient to encode a problem as an MSO sentence in order to obtain such a result. To put this into practice, tailored systems for MSO logic are required, however. While there has been remarkable progress in this direction [26] there is still evidence that designing DP algorithms for the considered problems from scratch results in more efficient software solutions (cf. [11, 30]).

The actual design of these algorithms can be quite tedious, especially for problems located at the second level of the polynomial hierarchy like the AI problems circumscription, abduction, answer set programming or abstract argumentation (see [16, 21, 23, 24]). In many cases, the increased complexity of such problems is caused by co-NP hard subset minimization or maximization subproblems (e.g., minimality of models in circumscription). It is exactly the handling of these subproblems that makes the design of the DP algorithms difficult.

What we aim for in this article is thus the automatic generation of intricate DP algorithms from simpler principles. To the best of our knowledge, there is only a little amount of work in this direction. The D-FLAT system [2] – a declarative framework for rapid prototyping of DP algorithms on tree decompositions – offers a few built-ins for cost minimization and the handling of join nodes in standard DP algorithms; the LISP-based Autograph approach (see, e.g., [10]) on the other hand makes it possible to obtain a specification of the problem at hand via combinations of (pre-defined) fly-automata.

What we have in mind is different and motivated by recent developments in the world of answer set programming (ASP) [8]: For exploiting the full expressive power of ASP, a saturation programming technique (see, e.g., [27]) is often required for the encoding of co-NP subproblems. Several approaches for relieving the user from this task have been proposed [7, 18, 19] that employ metaprogramming techniques. For instance, in order to compute minimal models of a propositional formula, one can simply express the SAT problem in ASP together with a special minimize statement (recognized by systems like *metasp*). In this way, one obtains a program computing subset minimal models. Unfortunately, easy-to-use facilities like such minimize statements had no analog in the area of DP so far.

In this article, we propose a solution to this issue: We provide a new method for automatically obtaining DP algorithms for problems requiring minimization, given only an algorithm for a problem variant without minimization. For example, given a DP algorithm for SAT [33], our approach enables us to generate a new algorithm for finding only subset-minimal models. Making minimization implicit in this way makes the programmer's life considerably easier. Moreover, we present D-FLAT^2, which is an implementation of the new method and extends D-FLAT[1]. To underline the practical relevance of this work, we show elegant solutions for common AI problems using D-FLAT^2.

The contributions of this article are the following:

- We introduce a formal model of DP computations, abstracting from concrete algorithms.

[1]The systems D-FLAT and D-FLAT^2 are publicly available at `http://dbai.tuwien.ac.at/proj/dflat/system/` and `https://github.com/hmarkus/dflat-2/`, respectively.

Our results are therefore generally applicable, not just to a particular problem.

- We then show how our model captures typical DP computations for subset minimization problems.
- We discuss how computations can be compressed to ensure fixed-parameter tractability.
- Our main contribution is a formal definition of a transformation that turns non-minimizing computations into ones that perform minimization. We identify under which conditions this procedure is sound and give a formal proof.
- Finally, we discuss implementation issues. In contrast to DP algorithms that implement subset minimization directly, we present an implementation of our method, called D-FLAT^2, which not only yields more succinct algorithm specifications, but also avoids redundant computations. We illustrate our method at work by providing DP encodings for several AI problems, and show promising preliminary performance results.

This article is structured as follows. In Section 2, DP on tree decompositions is introduced. Section 3 contains our formal model of DP computations. In Section 4 we illustrate how the concepts are put into practice in the D-FLAT and D-FLAT^2 systems. Next, in Section 5 we show how common AI problems such as Circumscription and disjunctive ASP can be solved using our approach. Finally, Section 6 reports on experimental results.

Compared to the workshop paper [4], this article contains a full formal proof of the correctness of the method implemented in D-FLAT^2.

2 Dynamic Programming on Tree Decompositions

In this section we outline DP on tree decompositions. The ideas underlying this concept stem from the field of parameterized complexity. Many computationally hard problems become tractable in case a certain problem parameter is bound to a fixed constant. This property is referred to as *fixed-parameter tractability* [14], and the complexity class FPT consists of problems that are solvable in $f(k) \cdot n^{\mathcal{O}(1)}$, where f is a function that only depends on the parameter k, and n is the input size.

For problems whose input can be represented as a graph, an important parameter is *treewidth*, which measures "tree-likeness" of a graph. It is defined by means of tree decompositions (TDs) [32].

Definition 1. *A* tree decomposition *of a graph $G = (V, E)$ is a pair $\mathcal{T} = (T, \chi)$ where $T = (N, F)$ is a (rooted) tree and $\chi : N \to 2^V$ assigns to each node a set of vertices (called the node's* bag*), such that the following conditions are met:*

(1) For every $v \in V$, there exists a node $n \in N$ such that $v \in \chi(n)$.

(2) For every edge $e \in E$, there exists a node $n \in N$ such that $e \subseteq \chi(n)$.

(3) For every $v \in V$, the subtree of T induced by $\{n \in N \mid v \in \chi(n)\}$ is connected.

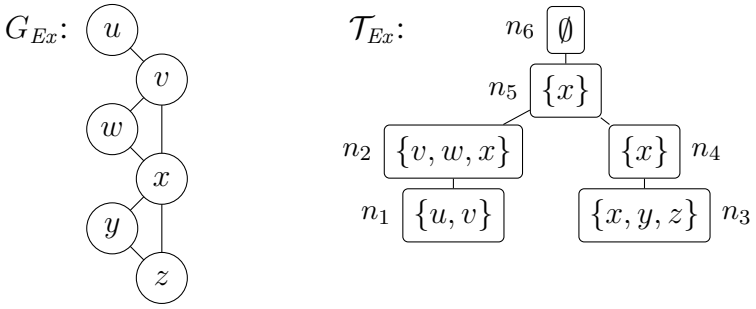


Figure 1: Primal graph G_{Ex} and a TD $\mathcal{T}_{Ex}$ of ϕ_{Ex}.

The width *of $\mathcal{T}$ is $\max_{n \in N} |\chi(n)| - 1$. The* treewidth *of a graph is the minimum width over all its tree decompositions.*

Although constructing a minimum-width TD is intractable in general [3], it is in FPT [5] and there are polynomial-time heuristics giving "good" TDs [6, 12, 13].

Example 1. *Let us consider the enumeration variant of the* SAT *problem. Given a propositional formula ϕ in CNF, we first have to find an appropriate graph representation. Here, we construct the* primal *graph G of ϕ, that is, vertices in G represent atoms of ϕ, and atoms occurring together in a clause form a clique in G. An example formula ϕ_{Ex}, its graph representation G_{Ex} and a possible TD $\mathcal{T}_{Ex}$ are given in Figure 1. The width of $\mathcal{T}_{Ex}$ is 2.*

Algorithms for DP on TDs generally traverse the TD in post-order. At each node, partial solutions for the subgraph induced by the vertices encountered so far are computed and stored in a data structure associated with the node. The size of the data structure is typically bounded by the TD's width and the number of TD nodes is linear in the input size. So if the width is bounded by a constant, the search space for subproblems is constant as well, and the number of subproblems only grows linearly for larger instances. We now illustrate the DP algorithm for SAT [33] on our running example; formal details are given in the next section.

Example 2. *Figure 2 illustrates the DP computation for* SAT. *The tables are computed as follows. For a TD node n, each table row r consists of data $\mathrm{D}(r)$, which stores partial truth assignments over atoms in $\chi(n)$. Here, $\mathrm{D}(r)$ only contains atoms that get assigned "true", atoms in $\chi(n) \setminus \mathrm{D}(r)$ get assigned "false". In r, all clauses covered by $\chi(n)$ must be satisfied by the partial truth assignment. The set $\mathrm{P}(r)$ contains so-called extension pointer tuples (EPTs) that denote the rows in the children where r was constructed from. First consider node n_1: here, $\chi(n_1) = \{u, v\}$ covers clause $(u \vee v)$, yielding three partial assignments for ϕ_{Ex}. In n_2, the child rows are extended, the partial assignments are updated (by removing atoms not contained in $\chi(n_2)$ and guessing truth assignments for atoms in $\chi(n_2) \setminus \chi(n_1)$). Here, clauses $(\neg v \vee w \vee x)$ and $(\neg w)$ have to be satisfied. Observe that row 2^1 is constructed from two different child rows. In n_3 we proceed as described before. In n_4, data related to removed vertices y and z are projected away. In n_5, additionally*

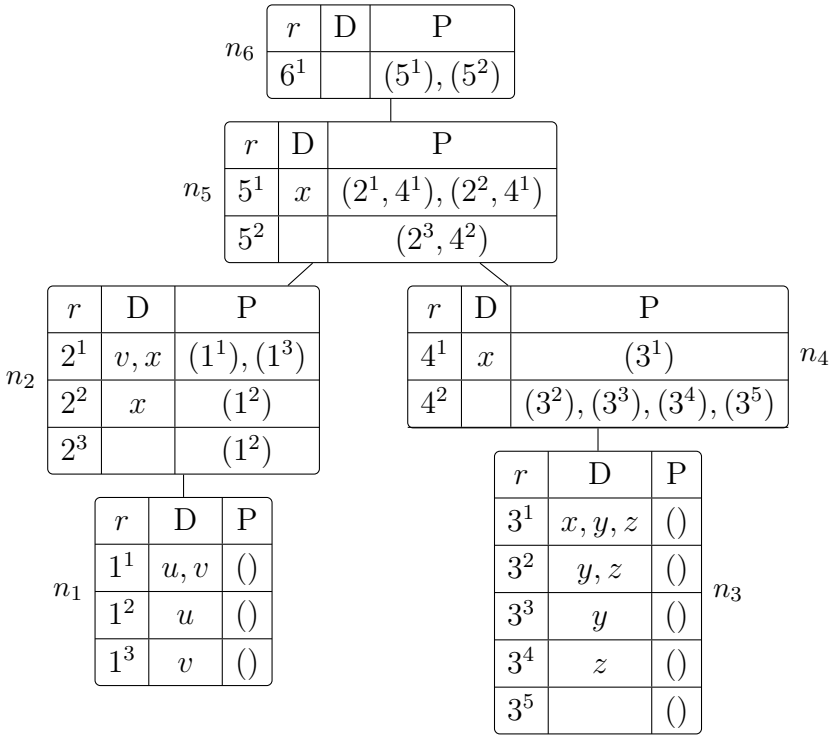


Figure 2: DP computation for SAT.

only partial assignments that agree on the truth assignment for common atoms are to be joined. We continue this procedure recursively until we reach the TD's root.

To decide whether a formula is satisfiable, it suffices to check if the table in the root node is non-empty. The overall procedure is in FPT time because the number of nodes in the TD is bounded by the input size (i.e., the number of atoms), and each node n is associated with a table of size at most $\mathcal{O}(2^{|\chi(n)|})$ (i.e., the number of possible truth assignments). For our example, ϕ_{Ex} is satisfiable due to existence of row 5^1. Solutions (models of ϕ_{Ex}) can be enumerated with linear delay by starting at the root and following the EPTs while combining the partial assignments associated with the rows. For instance, model $\{u, v, x, y, z\}$ is constructed by starting at 6^1 and following EPTs (5^1), $(2^1, 4^1)$, (1^1) and (3^1), thereby combining $\mathrm{D}(6^1) \cup \mathrm{D}(5^1) \cup \mathrm{D}(2^1) \cup \mathrm{D}(1^1) \cup \mathrm{D}(4^1) \cup \mathrm{D}(3^1)$.

The problems we will focus on in this paper are more involved. One such problem is the enumeration of the *subset-minimal* models of CNF formulas. Note that a naive implementation of this problem where the enumeration of models is used to filter out the minimal ones would violate the desired linear delay for enumerating the solutions. Indeed, genuine DP algorithms for such problem which involve subset minimization are needed. In the forthcoming section, we will present such an DP algorithm for $\subseteq$-MINIMAL SAT. Together with the above algorithm for SAT these algorithms will be used as a running example to illustrate our method.

3 A Formal Account of DP on TDs

In this section we formally introduce DP on TDs for subset optimization problems. First we define our data structure, called *computation*, which is a tree of tables resulting from the bottom-up traversal of the TD. We then define the *extensions* of rows in the tables, that are used to obtain (partial) solution candidates for the problem at hand. Additionally, we define their relation to so-called counterexamples. Counterexamples are witnesses for a solution candidate not being subset-minimal. Finally, we state how solutions are obtained, and define properties that have to be fulfilled by the tables in a computation in order to yield correct results. Section 3.2 then deals with compressing the tables, such that the size of the tables is bounded by the width of the TD. Finally, Section 3.3 introduces requirements for a computation without minimization to be eligible for transformation into a "minimizing" one (we call such a computation *augmentable*). We then show how this transformation can be achieved automatically, and prove that the solutions of the augmentable computation correspond to subset-minimal solutions of the original computation.

3.1 Computations

First, let us define the data structure used in our approach.

Definition 2. *A* computation *is a rooted ordered tree whose nodes are called* tables*. Each table R is a set of* rows *and each row $r \in R$ possesses*

- *some problem-specific* data $\mathrm{D}(r)$*,*
- *a non-empty set of* extension pointer tuples *(EPTs)* $\mathrm{P}(r)$ *such that each tuple is of arity k, where k is the number of children of R, and for each $(p_1, \ldots, p_k) \in \mathrm{P}(r)$ it holds that each p_i is a row of the i-th child of R,*
- *a* subtable $\mathrm{S}(r)$*, which is a set of* subrows*, where each subrow $s \in \mathrm{S}(r)$ possesses*
  - *some problem-specific data* $\mathrm{D}(s)$*,*
  - *a non-empty set of EPTs* $\mathrm{P}(s)$ *such that for each $(p_1, \ldots, p_k) \in \mathrm{P}(s)$ there is some $(q_1, \ldots, q_k) \in \mathrm{P}(r)$ with $p_i \in \mathrm{S}(q_i)$ for $1 \leq i \leq k$,*
  - *an inclusion status flag* $\mathrm{inc}(s) \in \{\mathrm{eq}, \subset\}$*.*

For rows or subrows a, b we write $a \approx b$, $a \leq b$ and $a < b$ to denote $\mathrm{D}(a) = \mathrm{D}(b)$, $\mathrm{D}(a) \subseteq \mathrm{D}(b)$ and $\mathrm{D}(a) \subset \mathrm{D}(b)$, respectively. For sets of rows or subrows R, S we write $\mathrm{D}(R)$ to denote $\bigcup_{r \in R} \mathrm{D}(r)$, and we write $R \approx S$, $R \leq S$ and $R < S$ to denote $\mathrm{D}(R) = \mathrm{D}(S)$, $\mathrm{D}(R) \subseteq \mathrm{D}(S)$ and $\mathrm{D}(R) \subset \mathrm{D}(S)$, respectively.

The reason that each row possesses a subtable is that we consider subset optimization problems, and we assume that algorithms for such problems use subtables to store potential counterexamples to a solution candidate being subset-minimal. The intuition of a subrow s for a row r is that s represents solution candidates that are subsets of the candidates represented by r. If one of these subset relations is proper, we indicate this by $\mathrm{inc}(s) = \subset$.

n_3

R			$\mathrm{S}(r)$			
r	D	P	s	D	P	inc
3^1	x, y, z	$()$	3^1_1	x, y, z	$()$	eq
			3^1_2	y, z	$()$	$\subset$
			3^1_3	y	$()$	$\subset$
			3^1_4	z	$()$	$\subset$
			3^1_5		$()$	$\subset$
3^2	y, z	$()$	3^2_1	y, z	$()$	eq
			3^2_2	y	$()$	$\subset$
			3^2_3	z	$()$	$\subset$
			3^2_4		$()$	$\subset$
3^3	y	$()$	3^3_1	y	$()$	eq
			3^3_2		$()$	$\subset$
3^4	z	$()$	3^4_1	z	$()$	eq
			3^4_2		$()$	$\subset$
3^5		$()$	3^5_1		$()$	eq

n_4

R			$\mathrm{S}(r)$			
r	D	P	s	D	P	inc
4^1	x	(3^1)	4^1_1	x	(3^1_1)	eq
			4^1_2		(3^1_2)	$\subset$
			4^1_3		(3^1_3)	$\subset$
			4^1_4		(3^1_4)	$\subset$
			4^1_5		(3^1_5)	$\subset$
4^2		(3^2)	4^2_1		(3^2_1)	eq
			4^2_2		(3^2_2)	$\subset$
			4^2_3		(3^2_3)	$\subset$
			4^2_4		(3^2_4)	$\subset$
4^3		(3^3)	4^3_1		(3^3_1)	eq
			4^3_2		(3^3_2)	$\subset$
4^4		(3^4)	4^4_1		(3^4_1)	eq
			4^4_2		(3^4_2)	$\subset$
4^5		(3^5)	4^5_1		(3^5_1)	eq

Figure 3: (Partial) DP computation for $\subseteq$-MINIMAL SAT without compression.

Example 3. *We extend our running example by now considering* $\subseteq$-MINIMAL SAT *(i.e., enumerating models that are subset-minimal w.r.t. the set of atoms that get assigned "true"). Figure 3 illustrates the computation for parts of our example. At* n_3*,* R *is computed as before. For any* $r \in R$*, each subrow* $s \in \mathrm{S}(r)$ *represents a partial "true" assignment that is a subset of the one in* r *(i.e.,* $\mathrm{D}(s) \subseteq \mathrm{D}(r)$*), and* $\mathrm{inc}(s)$ *is set appropriately. Now consider* n_4*, where* y *and* z *are removed. As in the* SAT *problem, we simply project away data related to the removed vertices. Observe that for instance* 4^1 *now contains redundant subrows* 4^1_2*,* 4^1_3*,* 4^1_4 *and* 4^1_5 *that contain the same data and inclusion flag (and only different EPTs). We overcome this problem in Section 3.2, where tables are* compressed *in order to remove redundancies. A complete example including compression is given in Section 3.2.*

The EPTs of a table row r are used for recursively combining the problem-specific data $\mathrm{D}(r)$ with data from "compatible" rows that are in descendant tables. The fact that each set of EPTs is required to be non-empty entails that for each (sub)row r at a leaf table it holds that $\mathrm{P}(r) = \{()\}$. We disallow rows with an empty set of EPTs because in the end we are only interested in rows that can be extended to complete solutions, consisting of one row per table. For this we introduce the notion of an *extension* of a table row.

Definition 3. *Let* $\mathcal{C}$ *be a computation and* R *be a table in* $\mathcal{C}$ *with* k *children. We inductively define the* extensions *of a row* $r \in R$ *as* $\mathrm{E}(r) = \{\{r\} \cup A \mid A \in \bigcup_{(p_1,\ldots,p_k)\in\mathrm{P}(r)} \{X_1 \cup \cdots \cup X_k \mid X_i \in$

$\mathrm{E}(p_i)$ *for all* $1 \leq i \leq k\}\}$.

Note that any extension $X \in \mathrm{E}(r)$ contains r and exactly one row from each table that is a descendant of R. If r is a row of a leaf table, $\mathrm{E}(r) = \{\{r\}\}$ because $\mathrm{P}(r) = \{()\}$.

While the extensions from the root table of a computation represent complete solution candidates, the purpose of subtables is to represent possible counterexamples that would cause a solution candidate to be invalidated. More precisely, for each extension X that can be obtained by extending a root table row r, we check if we can find an extension Y of an element $s \in \mathrm{S}(r)$ with $\mathrm{inc}(s) = \subset$ such that every element of Y is listed as a subrow of a row in X (i.e., we check if for every $y \in Y$ there is some $x \in X$ with $y \in \mathrm{S}(x)$). If this is so, then Y witnesses that X represents no solution because Y then represents a solution candidate that is a proper subset. For this reason, we need to introduce the notion of extensions (like Y) relative to another extension (like X).

Definition 4. *Let $\mathcal{C}$ be a computation, R be a table in $\mathcal{C}$ with k children, $r \in R$ be a row and $s \in \mathrm{S}(r)$ be a subrow of r. We first define, for any $X \in \mathrm{E}(r)$, a restriction of $\mathrm{P}(s)$ to EPTs where each element is a subrow of a row in X, as $\mathrm{P}_X(s) = \{(p_1, \ldots, p_k) \in \mathrm{P}(s) \mid r_i \in X,\ p_i \in \mathrm{S}(r_i)$ for all $1 \leq i \leq k\}$. Now we define the set of* extensions *of s* relative to *some extension $X \in \mathrm{E}(r)$ as $\mathrm{E}_X(s) = \{\{s\} \cup A \mid A \in \bigcup_{(p_1,\ldots,p_k) \in \mathrm{P}_X(s)} \{Y_1 \cup \cdots \cup Y_k \mid Y_i \in \mathrm{E}_X(p_i)$ for all $1 \leq i \leq k\}\}$.*

We can now formalize that the solutions of a computation are the extensions of those rows that do not have a subrow indicating a counterexample.

Definition 5. *Let R be the root table in a computation $\mathcal{C}$. We define the set of* solutions *of $\mathcal{C}$ as* $\mathrm{sol}(\mathcal{C}) = \{\mathrm{D}(X) \mid r \in R,\ X \in \mathrm{E}(r),\ \nexists s \in \mathrm{S}(r) : \mathrm{inc}(s) = \subset\}$

Example 4. *If n_4 in Figure 3 were the root of the TD, only 4^5 would yield a solution (i.e. $\{\}$, the interpretation where x, y, and z are all set to false), since all other rows contain a subrow where $\mathrm{inc}(s) = \subset$. This is indeed the only subset-minimal model of the formula consisting of the clauses encountered until n_4, i.e., $(\neg x \vee z) \wedge (\neg x \vee y \vee \neg z)$.*

Next we formalize requirements on subrows and their inclusion status to ensure that subrows correspond to subsets of their parent row, that each potential counterexample is represented by a subrow and that $\mathrm{inc}(\cdot)$ is used as intended.

Definition 6. *A table R is* normal *if the following properties hold:*

1. *For each $r \in R$, $s \in \mathrm{S}(r)$, $X \in \mathrm{E}(r)$ and $Y \in \mathrm{E}_X(s)$, it holds that $Y \leq X$, and $Y < X$ holds if and only if $\mathrm{inc}(s) = \subset$.*
2. *For each $r \in R$, $s \in \mathrm{S}(r)$ and $Y \in \mathrm{E}(s)$ there is some $r' \in R$ and $X' \in \mathrm{E}(r')$ such that $s \approx r'$ and $Y \approx X'$.*
3. *For each $q, r \in R$, $Z \in \mathrm{E}(q)$ and $X \in \mathrm{E}(r)$, if $Z \leq X$ holds, then there is some $s \in \mathrm{S}(r)$ and $Y \in \mathrm{E}_X(s)$ with $s \approx q$ and $Y \approx Z$.*

A computation is normal *if all its tables are normal.*

This definition ensures that it suffices to examine the root table of a normal computation in order to decide a subset minimization problem correctly, provided that the rows represent all solution candidates.

We will later show how a non-minimizing computation (i.e., one with empty subtables) satisfying certain properties can be transformed into a normal computation. In this transformation, we must avoid redundancies lest we destroy fixed-parameter tractability. For this, we first introduce how tables can be compressed without losing solution candidates.

3.2 Table Compression

To compress tables by merging equivalent (sub)rows, which is required for keeping the size of the tables bounded by the width of the TD, we first define an equivalence relation on rows, as well as one on subrows.

Definition 7. *Let R be a table and $r \in R$. We define an equivalence relation $\equiv_r$ over subrows of r such that $s_1 \equiv_r s_2$ if $s_1 \approx s_2$ and $\mathrm{inc}(s_1) = \mathrm{inc}(s_2)$.*

We use this notion of equivalence between subrows to compress subtables by merging equivalent subrows.

Definition 8. *Let R be a table and $r \in R$. We define a subtable $\mathrm{S}^*(r)$ called the* compressed subtable *of r that contains exactly one subrow for each $\equiv_r$-equivalence class. For any $s \in \mathrm{S}(r)$, let $[s]$ denote the $\equiv_r$-equivalence class of s and let s' denote the subrow in $\mathrm{S}^*(r)$ corresponding to $[s]$. We define s' by $s' \approx s$, $\mathrm{inc}(s') = \mathrm{inc}(s)$ and $\mathrm{P}(s') = \bigcup_{t \in [s]} \mathrm{P}(t)$.*

Once subtables have been compressed, we can compress the table by merging equivalent rows. For this, we first need a notion of equivalence between rows.

Definition 9. *We define an equivalence relation $\equiv_R$ over rows of a table R such that $r_1 \equiv_R r_2$ if $r_1 \approx r_2$ and there is a bijection $f : \mathrm{S}^*(r_1) \to \mathrm{S}^*(r_2)$ such that for any $s \in \mathrm{S}^*(r_1)$ it holds that $s \approx f(s)$ and $\mathrm{inc}(s) = \mathrm{inc}(f(s))$.*

When rows are equivalent, their compressed subtables only differ in the EPTs. We now define how such compressed subtables can be merged.

Definition 10. *Let R be a table, $r \in R$, and let $[r]$ denote the $\equiv_R$-equivalence class of r. For any $r' \in [r]$, let $f_{r'} : \mathrm{S}^*(r) \to \mathrm{S}^*(r')$ be the bijection such that for any $s \in \mathrm{S}^*(r)$ it holds that $s \approx f_{r'}(s)$ and $\mathrm{inc}(s) = \mathrm{inc}(f_{r'}(s))$. (The existence of $f_{r'}$ is guaranteed by Definition 9.) We define a subtable $\mathrm{mst}([r])$ (for "merged subtable") that contains exactly one subrow for each element of $\mathrm{S}^*(r)$. For any $s \in \mathrm{S}^*(r)$, let s' denote the subrow in $\mathrm{mst}([r])$ corresponding to s. We define s' by $s' \approx s$, $\mathrm{inc}(s') = \mathrm{inc}(s)$ and $\mathrm{P}(s') = \bigcup_{r' \in [r]} \mathrm{P}(f_{r'}(s))$.*

We use these equivalence relations to compress tables in such a way that all equivalent (sub)rows (according to the respective equivalence relation) are merged.

Definition 11. *Let R be a table. We now define a table* $\mathrm{compr}(R)$ *that contains exactly one row for each $\equiv_R$-equivalence class. For any $r \in R$, let $[r]$ denote the $\equiv_R$-equivalence class of r and let r' be the row in* $\mathrm{compr}(R)$ *corresponding to $[r]$. We define r' by $r' \approx r$,* $\mathrm{P}(r') = \bigcup_{q \in [r]} \mathrm{P}(q)$ *and* $\mathrm{S}(r') = \mathrm{mst}([r])$. *For any computation $\mathcal{C}$, we write* $\mathrm{compr}(\mathcal{C})$ *to denote the computation isomorphic to $\mathcal{C}$ where each table R in $\mathcal{C}$ corresponds to* $\mathrm{compr}(R)$.

Example 5. *Figure 4 illustrates the complete DP computation for our running example, including compression. Root node n_6 contains two rows, 6^1 and 6^2. 6^1 represents the subset-minimal models of our running example, since it is associated with a single subrow 6^2_1, where* $\mathrm{inc}(6^2_1) = \mathrm{eq}$. *Opposed to that, models represented by 6^1 are not subset-minimal, due to* $\mathrm{inc}(6^1_2) = \subset$. *We obtain the solutions by following the EPTs and combining* P *of the respective rows: We have* $\mathrm{P}(6^2) \cup \mathrm{P}(5^4) \cup \mathrm{P}(2^3) \cup \mathrm{P}(1^2) \cup \mathrm{P}(4^3) \cup \mathrm{P}(3^5) = \{u\}$ *and* $\mathrm{P}(6^2) \cup \mathrm{P}(5^5) \cup \mathrm{P}(2^4) \cup \mathrm{P}(1^3) \cup \mathrm{P}(4^1) \cup \mathrm{P}(3^1) = \{x, v, y, z\}$. *Thus the subset-minimal models (cf. solutions) are $\{u\}$ (i.e., u set to true, and v, x, y, z set to false) as well as $\{v, x, y, z\}$ (with u set to false).*

Regarding compression, consider the table stored at node n_4 in comparison to the non-compressed table for n_4 in Figure 3. For instance, due to Definition 10 4^1 now has subrow 4^1_2 that contains a set of EPTs, i.e., $\mathrm{P}(4^1_2) = \{(3^1_2), (3^1_3), (3^1_4), (3^1_5)\}$. *Additionally, due to Definition 11 we can merge complete rows. For instance, observe that the single row 4^2 is obtained by compressing rows 4^2, 4^3 and 4^4 of Figure 3, since the data of these rows as well as the data and the inclusion flags of all subrows coincide, i.e., they are contained in the same $\equiv_R$-equivalence class.*

We now show that compressing a table according to these definitions retains normality.

Lemma 1. *If a table R is normal, then so is* $\mathrm{compr}(R)$.

Proof sketch. Let R be a normal table and $R' = \mathrm{compr}(R)$. We prove conditions 1–3 of normality of R' separately. Let $r' \in R'$, $s' \in \mathrm{S}(r')$, $X' \in \mathrm{E}(r')$ and $Y'_{X'} \in \mathrm{E}_{X'}(s')$. Then we can find $r \in R$, $s \in \mathrm{S}(r)$, $X \in \mathrm{E}(r)$ and $Y \in \mathrm{E}_X(s)$ such that $r \approx r'$, $s \approx s'$, $X \approx X'$ and $Y \approx Y'_{X'}$. As R is normal, $Y \leq X$ holds, and $Y < X$ if and only if $\mathrm{inc}(s) = \subset$. This entails $Y'_{X'} \leq X'$, and $Y'_{X'} < X'$ if and only if $\mathrm{inc}(s') = \subset$ because $\mathrm{inc}(s') = \mathrm{inc}(s)$. This proves condition 1. Let $Y' \in \mathrm{E}(s')$. Then we can find $Y \in \mathrm{E}(s)$ such that $Y \approx Y'$. As R is normal, there is a row $t \in R$ with $t \approx s$ and an extension $T \in \mathrm{E}(t)$ such that $T \approx Y$. Then there is some $t' \in R'$ with $t' \approx t$ and $(T \setminus \{t\}) \cup \{t'\} \in \mathrm{E}(t')$, which proves condition 2. Let $q' \in R$ and $Z' \in \mathrm{E}(q')$ such that $Z' \leq X'$. Then we can find $q \in R$ and $Z \in \mathrm{E}(q)$ such that $Z \approx Z'$, so $Z \leq X$. As R is normal, there are $u \in \mathrm{S}(r)$ and $U \in \mathrm{E}_X(u)$ such that $u \approx q$ and $U \approx Z$. Then there is some $u' \in \mathrm{S}(r')$ with $u' \approx u$ and $(U \setminus \{u\}) \cup \{u'\} \in \mathrm{E}_{X'}(u')$, which proves condition 3. □

3.3 Normalizing Computations

Before we introduce our transformation from "non-minimizing" computations to "minimizing" ones, we define certain conditions that are prerequisites for the transformation. For this, we first define the set of all data of rows that have occurred in a table or any of its descendants.

Definition 12. *Let R be a table in a computation such that $R_1, \ldots, R_k$ are the child tables of R. We inductively define* $\mathrm{D}^*(R) = \bigcup_{r \in R} D(r) \cup \bigcup_{1 \leq i \leq k} D^*(R_i)$.

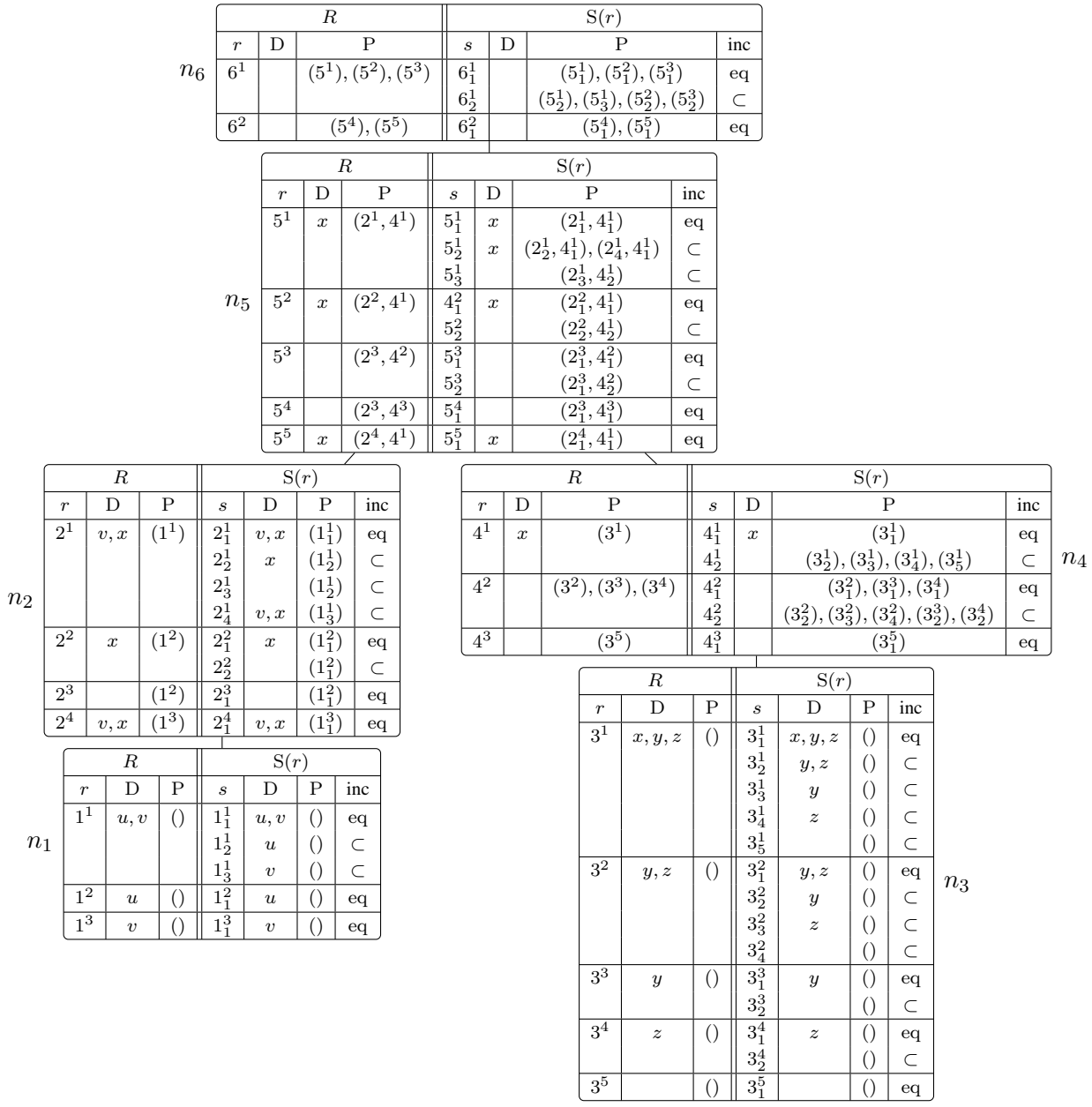


Figure 4: DP computation for $\subseteq$-MINIMAL SAT.

Now we define conditions that the tables in a computation must satisfy for being eligible for our transformation.

Definition 13. *Let R be a table in a computation such that $R_1, \ldots, R_k$ are the child tables of R, and let $r, r' \in R$. We say that $d \in \mathrm{D}(r)$ has been* illegally introduced *at r if there are $(r_1, \ldots, r_k) \in \mathrm{P}(r)$ such that for some $1 \leq i \leq k$ it holds that $d \notin \mathrm{D}(r_i)$ while $d \in \mathrm{D}^*(R_i)$. Moreover, we say that $d \in \mathrm{D}(r') \setminus \mathrm{D}(r)$ has been* illegally removed *at r if there is some $X \in \mathrm{E}(r)$ such that $d \in X$.*

We now define the notion of an *augmentable* table, i.e., a table that can be used in our transformation.

Definition 14. *We call a table* R augmentable *if the following conditions hold:*

1. *For all rows* $r \in R$ *it holds that* $\mathrm{S}(r) = \emptyset$.
2. *For all* $r, r' \in R$ *with* $r \neq r'$ *it holds that* $\mathrm{D}(r) \neq \mathrm{D}(r')$.
3. *For all* $r \in R$, $(r_1, \ldots, r_k) \in \mathrm{P}(r)$, $1 \leq h < j \leq k$, $H \in \mathrm{E}(r_h)$ *and* $J \in \mathrm{E}(r_j)$ *it holds that* $\mathrm{D}(H) \cap \mathrm{D}(J) \subseteq \mathrm{D}(r)$.
4. *No element of* $\mathrm{D}(R)$ *has been illegally introduced.*
5. *No element of* $\mathrm{D}(R)$ *has been illegally removed.*

We call a computation augmentable *if all its tables are augmentable.*

These requirements are satisfied by reasonable TD-based DP algorithms (cf. [30]) as these usually do not put arbitrary data into the rows. Rather, the data in a row is typically restricted to information about bag elements of the respective TD node. For instance, condition 3 mirrors condition 3 of Definition 1, and condition 2 is usually satisfied by reasonable FPT algorithms because they avoid redundancies in order to stay fixed-parameter tractable.

Now we describe how augmentable computations can automatically be transformed into normal computations that take minimization into account. For any table R in an augmentable computation, this allows us to compute a new table $\mathrm{aug}(R)$ if for each child table R_i the table $\mathrm{aug}(R_i)$ has already been computed and compressed to $\mathrm{compr}(\mathrm{aug}(R_i))$.

Definition 15. *We inductively define a function* $\mathrm{aug}(\cdot)$ *that maps each table* R *from an augmentable computation to a table. Let the child tables of* R *be called* $R_1, \ldots, R_k$. *For any* $1 \leq i \leq k$ *and* $r \in R_i$, *we write* $\mathrm{res}(r)$ *to denote* $\{q \in \mathrm{compr}(\mathrm{aug}(R_i)) \mid q \approx r\}$. *We define* $\mathrm{aug}(R)$ *as the smallest table that satisfies the following conditions:*

1. *For any* $r \in R$, $(r_1, \ldots, r_k) \in \mathrm{P}(r)$ *and* $(c_1, \ldots, c_k) \in \mathrm{res}(r_1) \times \cdots \times \mathrm{res}(r_k)$, *there is a row* $q \in \mathrm{aug}(R)$ *with* $q \approx r$ *and* $\mathrm{P}(q) = \{(c_1, \ldots, c_k)\}$.
2. *For any* $q, q' \in \mathrm{aug}(R)$ *such that* $q' \leq q$, $\mathrm{P}(q) = \{(q_1, \ldots, q_k)\}$ *and* $\mathrm{P}(q') = \{(q'_1, \ldots, q'_k)\}$ *the following holds: If for all* $1 \leq i \leq k$ *there is some* $s_i \in \mathrm{S}(q_i)$ *with* $s_i \approx q'_i$, *then there is a subrow* $s \in \mathrm{S}(q)$ *with* $s \approx q'$ *and* $\mathrm{P}(s) = \{(s_1, \ldots, s_k)\}$. *Moreover,* $\mathrm{inc}(s) = \subset$ *if* $q' < q$ *or* $\mathrm{inc}(s_i) = \subset$ *for some* s_i, *otherwise* $\mathrm{inc}(s) = \mathrm{eq}$.

For any augmentable computation $\mathcal{C}$, *we write* $\mathrm{aug}(\mathcal{C})$ *to denote the computation isomorphic to* $\mathcal{C}$ *where each table* R *in* $\mathcal{C}$ *corresponds to* $\mathrm{aug}(R)$.

Augmentable tables never have two different rows r, r' with $\mathrm{D}(r) = \mathrm{D}(r')$. Moreover, we defined $\mathrm{aug}(R)$ in such a way that for all $r \in R$ there is some $q \in \mathrm{aug}(R)$ with $r \approx q$. In the compression $\mathrm{compr}(\mathrm{aug}(R))$, we only merge rows and subrows having the same data. So with each row and subrow in $\mathrm{aug}(R)$ or $\mathrm{compr}(aug(R))$ we can associate a unique originating row in R. In fact, the extensions from an augmentable table R are in a one-to-one correspondence to the extensions of rows from $\mathrm{aug}(R)$. This is formalized by the following lemma, which we prove in the appendix.

Lemma 2. *Let R be a table from an augmentable computation and $Q = \mathrm{aug}(R)$. Then for any $r \in R$ and $Z \in \mathrm{E}(r)$ there are $q \in Q$ and $X \in \mathrm{E}(q)$ such that $r \approx q$ and $Z \approx X$. Also, for any $q \in Q$ and $X \in \mathrm{E}(q)$ there are $r \in R$ and $Z \in \mathrm{E}(r)$ such that $q \approx r$ and $X \approx Z$.*

The following lemma is central for showing that $\mathrm{aug}(\cdot)$ works as intended. For a full proof, see the appendix.

Lemma 3. *Let R be a table from an augmentable computation. Then the table $\mathrm{aug}(R)$ is normal.*

Proof sketch. Let R be a table in some augmentable computation such that $R_1, \ldots, R_k$ denote the child tables of R and let $Q = \mathrm{aug}(R)$. We use induction. If Q is a leaf table, then rows and extensions coincide and the construction of Q obviously ensures that Q is normal. If Q has child tables $Q_i = \mathrm{compr}(\mathrm{aug}(R_i))$ and all $\mathrm{aug}(R_i)$ are normal, all Q_i are normal by Lemma 1. Let $q \in Q$, $s \in \mathrm{S}(q)$, $\mathrm{P}(q) = \{(q_1, \ldots, q_k)\}$, $\mathrm{P}(s) = \{(s_1, \ldots, s_k)\}$, $X_i \in \mathrm{E}(q_i)$, $Y_i \in \mathrm{E}_{X_i}(s_i)$, $X = \{q\} \cup X_1 \cup \cdots \cup X_k$ and $Y = \{s\} \cup Y_1 \cup \cdots \cup Y_k$. As for normality condition 1, the construction of Q ensures $s \leq q$ and normality of Q_i ensures $Y_i \leq X_i$, so $Y \leq X$. To show that $\mathrm{inc}(s)$ has the correct value, first suppose $Y < X$ and $\mathrm{inc}(s) = \mathrm{eq}$. The latter would entail $Y_i = X_i$, so $s < q$, but then $\mathrm{inc}(s) = \subset$, which is a contradiction. So suppose $X \approx Y$ and $\mathrm{inc}(s) = \subset$. If $q < s$, there would be an illegal removal at the origin of s in R, contradicting that R is augmentable. So for some j there is a $d \in \mathrm{D}(X_j) \setminus \mathrm{D}(Y_j)$. Due to $X \approx Y$, $d \in \mathrm{D}(s)$ or $d \in \mathrm{D}(Y_h)$ for some $h \neq j$. In the first case, there is an illegal introduction at the origin of s in R. In the other case, $d \in \mathrm{D}(Y_h)$ entails $d \in \mathrm{D}(X_h)$. As row extensions in Q are in a one-to-one correspondence with those in R, and by augmentability of R, $d \in \mathrm{D}(X_j) \cap \mathrm{D}(X_h)$ entails $d \in \mathrm{D}(q)$. But then $d \in \mathrm{D}(s)$, which we already led to a contradiction.

For condition 2, let $Z_i \in \mathrm{E}(s_i)$ and $Z = \{s\} \cup Z_1 \cup \cdots \cup Z_k$. As $s \in \mathrm{S}(q)$, there are $p \in Q$ and $(p_1, \ldots, p_i) \in \mathrm{P}(p)$ with $p \approx s$ and $p_i \approx s_i$ This entails existence of $r \in R$ and $(r_1, \ldots, r_k) \in \mathrm{E}(r)$ with $r \approx p \approx s$ and $r_i \approx p_i$. By hypothesis, there are $q_i' \in Q_i$ and $X_i' \in \mathrm{E}(q_i')$ with $q_i' \approx s_i$ and $X_i' \approx Z_i$. Each q_i' originates from the unique $r_i \in R$ with $r_i \approx q_i'$. So $q_i' \in \mathrm{res}(r_i)$ holds and there are $q' \in Q$ and $X' \in \mathrm{E}(q')$ with $q' \approx r \approx s$ and $X' \approx Z$.

For condition 3, let $q' \in Q$, $\mathrm{P}(q') = \{(q_1', \ldots, q_k')\}$, $X' \in \mathrm{E}(q')$ and $X_i' \in \mathrm{E}(q_i')$ for all $1 \leq i \leq k$. Suppose $X' \leq X$ and, for the sake of contradiction, for some j there is a $d \in \mathrm{D}(X_j') \setminus \mathrm{D}(X_j)$. Then $d \in \mathrm{D}(q)$ or $d \in \mathrm{D}(X_h)$ for some $h \neq j$. In the first case, there is an illegal introduction at the origin of q in R. In the other case, $d \in \mathrm{D}(X_j) \cap \mathrm{D}(X_h)$ entails $d \in \mathrm{D}(q)$, which we already led to a contradiction. So $X_i' \leq X_i$ for each i. By hypothesis then there are $t_i \in \mathrm{S}(q_i)$ and $T_i \in \mathrm{E}_{X_i}(t_i)$ with $t_i \approx q_i'$ and $T_i \approx X_i'$. So there is a $t \in \mathrm{S}(q)$ with $t \approx q'$ and $\mathrm{P}(t) = \{(t_1, \ldots, t_k)\}$. Then $T = \{t\} \cup T_i \cup \cdots \cup T_k$ is in $\mathrm{E}_X(t)$ and $T \approx X'$. □

We can now state our main theorem, which says that exactly the subset-minimal solutions of an augmentable computation are solutions of the augmented computation.

Theorem 1. *Let $\mathcal{C}$ be an augmentable computation. Then* $\mathrm{sol}(\mathrm{aug}(\mathcal{C})) = \{S \in \mathrm{sol}(\mathcal{C}) \mid \nexists S' \in \mathrm{sol}(\mathcal{C}) : S' \subset S\}$.

Proof. Let R be the root table of an augmentable computation $\mathcal{C}$ and R' be the root table of $\mathcal{C}' = \mathrm{aug}(\mathcal{C})$. By Lemma 3, $\mathcal{C}'$ is normal. For the first direction, let $S \in \mathrm{sol}(\mathcal{C}')$. Then there are $r' \in R'$ and $X' \in \mathrm{E}(r')$ such that $\mathrm{D}(X') = S$ and for all $s' \in \mathrm{S}(r')$ it holds that $\mathrm{inc}(s') = \mathrm{eq}$. By Lemma 2, then there are $r \in R$ and $X \in \mathrm{E}(r)$ such that $X \approx X'$. As R is augmentable, $\mathrm{S}(r)$ is empty, so $S \in \mathrm{sol}(\mathcal{C})$ by Definition 5. We must now show that there is no solution in $\mathcal{C}$ smaller than S. For the sake of contradiction, suppose there is some $T \in \mathrm{sol}(\mathcal{C})$ with $T \subset S$. Then there are $q \in R$ and $Z \in \mathrm{E}(q)$ such that $\mathrm{D}(Z) = T$, hence $Z < X'$. By Lemma 2, then there are $q' \in R'$ and $Z' \in \mathrm{E}(q')$ such that $Z' \approx Z$. As R' is normal, due to $Z' < X'$, there is some $s' \in \mathrm{S}(r')$ such that $\mathrm{inc}(s') = \subset$. This contradicts $\mathrm{inc}(s') = \mathrm{eq}$, which we have seen earlier.

For the other direction, let $S \in \mathrm{sol}(\mathcal{C})$ be such that there is no $S' \in \mathrm{sol}(\mathcal{C})$ with $S' \subset S$. Then there are $r \in R$ and $X \in \mathrm{E}(r)$ such that $\mathrm{D}(X) = S$. By Lemma 2, then there are $r' \in R'$ and $X' \in \mathrm{E}(r')$ such that $X' \approx X$, and there are no $q' \in R'$ and $Z' \in \mathrm{E}(q')$ with $Z' < X$. Hence, as R' is normal, there cannot be a $s' \in \mathrm{S}(r')$ with $\mathrm{inc}(s') = \subset$. This proves that $S \in \mathrm{sol}(\mathcal{C}')$. □

Finally, we sketch that $\mathrm{aug}(\cdot)$ does not destroy fixed-parameter tractability.

Theorem 2. *Let $\mathcal{A}$ be an algorithm that takes as input an instance of size n and treewidth w along with a TD $\mathcal{T}$ of width w. Suppose $\mathcal{A}$ produces an augmentable computation $\mathcal{C}$ isomorphic to $\mathcal{T}$ in time $f(w) \cdot n^{\mathcal{O}(1)}$, where f is a function depending only on w. Then* $\mathrm{aug}(\mathcal{C})$ *can be computed in time $g(w) \cdot n^{\mathcal{O}(1)}$, where g again depends only on w.*

Proof sketch. We assume w.l.o.g. that each node in $\mathcal{T}$ has at most 2 children, as any TD can be transformed to this form in linear time without increasing the width [25]. As $\mathcal{A}$ runs in FPT time, i.e., in $f(w) \cdot n^{\mathcal{O}(1)}$ for a function f, no table in $\mathcal{C}$ can be bigger than $f(w) \cdot n^c$ for a constant c. To inductively compute $Q = \mathrm{aug}(R)$ for some table R in $\mathcal{C}$ with child tables $R_1, \ldots, R_k$ ($k \leq 2$), suppose we have already constructed each $Q_i = \mathrm{aug}(R_i)$ in FPT time. Then $|Q_i| = f_i(w) \cdot n^{c_i}$ for some f_i and c_i. We can compute $Q_i' = \mathrm{compr}(Q_i)$ in time polynomial in $|Q_i|$. Then $|Q_i'| = f_i'(w) \cdot n^{c_i'}$ for some f_i' and c_i'. Definition 15 suggests a straightforward way to compute Q in time polynomial in $|R|$ and $\sum_{1 \leq i \leq k} |Q_i'|$. So we can compute Q in FPT time. As $\mathcal{T}$ has size $\mathcal{O}(n)$, we can compute $\mathrm{aug}(\mathcal{C})$ in FPT time. □

4 Practical Realization

In this section we illustrate how the concepts introduced so far can be put into practice. We first recall the D-FLAT system, a system which enables specifications of DP algorithms on TDs via ASP. Then, we introduce a new variant of D-FLAT, called D-FLAT^2. While D-FLAT requires an explicit encoding of subset minimization within the DP algorithm, subset minimization is handled

automatically in D-FLAT^2, provided that the encoded problem produces an augmentable computation. Both systems use the library Htdecomp [13] for the generation of the tree decomposition and use state-of-the-art ASP technology from the Potassco family [20] for running the encodings of the DP algorithms.

4.1 D-FLAT: A Quick Tutorial

D-FLAT [1] is a framework for simplifying DP on TDs such that the user only has to provide an ASP program formulating the DP algorithm on TDs for solving the given graph-based problem. It automatically constructs a TD of the input graph in polynomial time (using heuristics). Then, the generated TD is traversed in post-order. At each DP node the tables containing the solution candidates are computed. There, an ASP solver is called with the following input:

1. the user-specified ASP encoding,
2. the input instance,
3. information about the current and child TD nodes (see Table 1), and
4. the partial solutions computed in the child TD nodes (see below).

Each model returned by the solver represents a partial solution of the current node.[2] In order to represent partial solutions, D-FLAT provides a general interface that supports (amongst others) rows *and* subrows of a table as introduced in Section 3. Additionally, for problems in NP, a simplified interface that *only* supports rows is available. We will now briefly describe the concepts behind these interfaces, as well as their relation to the formal concepts presented in Section 3.

Simplified interface (for problems in NP). The simplified interface supports tables that contain rows *without* subrows. When traversing the TD, tables for the already-visited child nodes are given to the user-specified encoding via predicates as listed in Table 2. Then, the partial solutions for the current TD node are computed via the user-specified encoding, and returned via the output predicates listed in Table 3. A row r consists of a set of *items* and *auxiliary items* that store problem-specific data $\mathrm{D}(r)$[3]. The output predicate $\texttt{extend}/1$ specifies the child row(s) that give rise to the partial solution encoded by the respective answer set.

Listing 1 contains the user-specified ASP encoding Π_{SAT} for the SAT problem. The encoding makes use of D-FLAT's input interface (see Tables 1 and 2) in the bodies of the rules, and the output interface (see Table 3) in the heads of the rules. Additionally, it expects that the input instance ϕ is encoded as follows. Atoms and clauses occurring in ϕ are given as ASP facts $\texttt{atom}(\cdot)$ and $\texttt{clause}(\cdot)$ respectively. Facts $\texttt{pos}(c, a)$ ($\texttt{neg}(c, a)$) denote that some atom a occurs positively (negatively) in clause c.

[2] We use colors to highlight input (red), output (orange) and input instance (blue) predicates.

[3] D-FLAT provides an efficient implementation for joining tables that emerge from different child nodes, which requires to specify items that are subject to set equivalence and auxiliary items that are subject to set union. Details are given in [1].

Table 1: Input predicates describing the tree decomposition.

Input predicate	Meaning
`final`	The current tree decomposition node is the root.
`childNode`(n)	n is a child of the current decomposition node.
`bag`(n, v)	Vertex v is contained in the bag of the decomposition node n.
`current`(v)	Vertex v is an element of the current bag.
`introduced`(v)	Vertex v is a current vertex but was in no child node's bag.
`removed`(v)	Vertex v was in a child node's bag but is not in the current one.

Table 2: Input predicates describing tables of decomposition child nodes.

Input predicate	Meaning
`childRow`(r, n)	r is a table row belonging to decomposition node n.
`childItem`(r, i)	$\mathrm{D}(r)$ of child row r contains item i.
`childAuxItem`(r, i)	$\mathrm{D}(r)$ of child row r contains auxiliary item i.

Table 3: Output predicates for constructing the table of the current decomposition node.

Output predicate	Meaning
`item`(i)	The data associated with the current row shall contain item i.
`auxItem`(i)	The data associated with the current row shall contain auxiliary item i.
`extend`(r)	The EPT of the current row shall contain a reference to child table row r.

Example 6. *We will now go through Listing 1 and explain how the partial solutions of our running example (depicted in Figure 2) are computed by D-FLAT. Let us first consider leaf node n_1 of $\mathcal{T}_{Ex}$. Since n_1 has no children, only Lines 6 and 12-19 are of interest to us. For all atoms in $\chi(n_1) = \{u, v\}$ we guess their truth assignment (Line 6). In case an atom gets assigned true, it is stored as* `item` *in the data of the computed row. Lines 12-14 specify the "current" clauses (i.e. clauses, such that all clause atoms occur in the bag of the current TD node). In our example, this yields clause $(u \vee v)$. Lines 16-17 denote whether such a clause is satisfied by the current truth assignment. Clause $(u \vee v)$ is satisfied by $\{u, v\}$, $\{u\}$ and $\{v\}$. However, $\{\}$ is no model of the clause, and removed by the constraint in Line 19. In n_2, $\{u\}$ is removed and $\{w, x\}$ is introduced to the bag. Whenever an atom was assigned false in a child row, Line 2 makes this explicit. We now* `extend` *each partial solution of the child node (Line 4). In Figure 2, this extension is denoted in column* P. *Their data (truth assignment), restricted to the current bag, is kept via Line 10. For introduced vertices, their truth assignment is guessed in Line 6. Similar to before, the "current" clauses are computed, and rows yielding unsatisfied current clauses are removed. In TD nodes having several children, we extend exactly one row per child table at a time (Line 4). Extended partial solutions have to agree on the truth assignment of atoms in the current bag*

```
1 % Define false atoms
2 f(R,X) ← childRow(R,N), bag(N,X), not childItem(R,X).
3 % Guess partial solutions to be extended
4 1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
5 % Guess truth value of introduced atoms
6 { item(A) : introduced(A) }.
7 % Only join rows coinciding on truth values of atoms
8 ← extend(X;Y), childItem(X,A), f(Y,A).
9 % True atoms are kept
10 item(X) ← extend(R), childItem(R,X), current(X).
11 % Current clauses
12 notCurr(C) ← pos(C,A), not current(A).
13 notCurr(C) ← neg(C,A), not current(A).
14 curr(C) ← clause(C), not notCurr(C).
15 % Define satisfied clauses
16 sat(C) ← curr(C), pos(C,A),     item(A).
17 sat(C) ← curr(C), neg(C,A), not item(A).
18 % Current clauses need to be satisfied
19 ← curr(C), not sat(C).
```

Listing 1: Π_{SAT}: D-FLAT encoding for solving SAT.

(Line 8). Consider join node n_5. For instance, since $\chi(n_5) \cap \mathrm{D}(2^1) = \chi(n_5) \cap \mathrm{D}(4^1)$, we can construct row 5^1 with $\mathrm{D}(5^1) = \{x\}$ and EPT $(2^1, 4^1)$.

General interface (for problems beyond NP). D-FLAT provides an interface that supports a more general data structure, called *item tree*. Here, we first briefly introduce item trees. Then, we show how item trees can be used to represent rows *with* subtables. For details we refer to [2]. Each node in the TD can be associated with an item tree. The predicates specifying item trees computed in the child nodes are given in Table 4, output predicates are given in Table 5. Similar to the simplified interface, each node in an item tree contains data, specified by `item`/2 and `auxItem`/2 predicates. Each item tree node additionally has an EPT that represents its origin, that contains extension pointers denoted by `extend`/2 (Note that these are now binary predicates). Each root-to-leaf path has a particular `length`/1, and the *level* of an item tree node is its depth on the path.

At the TD's root, each item tree node must be labeled with either `accept` or `reject` if it is a leaf, otherwise with `or`/1 or `and`/1. D-FLAT uses this to filter out solution candidates for which "counterexamples" exist: Only so-called *accepting* nodes are kept at the TD root, where a node is accepting if a) its label is `accept`, or b) its label is `or` and at least one child is accepting, or c) its label is `and` and all children are accepting. One can view the simplified interface as a special case of the interface for item trees, where the `length` of each root-to-leaf path is one, and the root node is of type `or`. Furthermore, contrary to tables, where each model returned by the ASP solver represents a row, for item trees a model represents a single root-to-leaf path in the item tree.

For solving subset minimization problems, we use item trees to represent tables R with subtables as follows. The `length` of each root-to-leaf path is two. The root node (level 0) of the item

Table 4: Input predicates describing item trees of child nodes in the decomposition.

Input predicate	Meaning
`atNode`(s, n)	s is an item tree node belonging to decomposition node n.
`rootOf`(s, n)	s is the root of the item tree at decomposition node n.
`sub`(r, s)	r is an item tree node with child s.
`childItem`(s, i)	The data of item tree node s contains item i.
`childAuxItem`(s, i)	The data of item tree node s contains auxiliary item i.

Table 5: Output predicates for constructing the item tree of the current decomposition node.

Output predicate	Meaning
`item`(l, i)	Item tree node at level l in the current root-to-leaf path shall contain item i.
`auxItem`(l, i)	Item tree node at level l in the current root-to-leaf path shall contain auxiliary item i.
`extend`(l, s)	The EPT associated with the item tree node at level l in the current root-to-leaf path shall contain a reference to item tree node s.
`length`(l)	The current root-to-leaf path has length l.
`or`(l)/`and`(l)	The node at level l in the current root-to-leaf path has type "or"/"and".
`accept`/`reject`	The leaf in the current root-to-leaf path has type "accept"/"reject".

tree is of type `or`. The nodes at level 1 are of type `and`. Here, the idea is to construct item trees, such that some (`or`) solution candidate (stored at level 1) is accepted if there is no counterexample (`and`) at level 2 that is smaller. For a row $r \in R$, we specify each item $i \in \mathrm{D}(r)$ by `item`$(1, i)$ (or `auxItem`$(1, i)$, respectively). $\mathrm{P}(r)$ is represented by predicates `extend`$(1, p_i)$ where p_i is a row in the i-th child of R. For a subrow $s \in \mathrm{S}(r)$, $i \in \mathrm{D}(s)$ is given via `item`$(2, i)$ predicates (or `auxItem`$(2, i)$), and EPTs $\mathrm{P}(s)$ are given by `extend`$(2, p_i)$ for some p_i. The inclusion status flag $\mathrm{inc}(s) = \subset$ is represented by `auxItem`$(2, \mathit{smaller})$, for $\mathrm{inc}(s) = \mathrm{eq}$ no flag is stored.

Example 7. *In the following we explain this concept on basis of* $\subseteq$-MINIMAL SAT. *In D-FLAT, it is again only required to specify a single ASP encoding, depicted in Listing 2. The encoding defines that data for both solution candidates and counterexamples are computed as in the* SAT *problem. Additionally, this encoding ensures that partial interpretations represented by counterexamples are subsets of partial interpretations represented by solution candidates. Line 1 defines the structure of the item trees used for the subset minimization problem. Similar to* Π_{SAT}*, Line 3 makes explicit if an atom is false in the data of an item tree node. Exactly one root-to-leaf path of each child item tree is extended (Lines 5-6). Then, for each level the truth assignment of introduced atoms is guessed. True atoms are stored via* `item`$/2$ *in the data of the row or subrow (Line 8). Root-to-leaf paths are only joined in case the respective item sets coincide on their partial interpretations for current atoms (Line 10). Truth assignments of extended child (sub)rows for current atoms are*

```
1 length(2). or(0). and(1).
2 % Define false atoms
3 f(S,X) ← atNode(S,N), sub(_,S), childNode(N), bag(N,X), not childItem(S,X).
4 % Guess root-to-leaf paths in item trees to be extended
5 extend(0,R) ← root(R).
6 1 { extend(L+1,S) : sub(R,S) } 1 ← extend(L,R), L<2.
7 % Guess truth value of introduced atoms
8 { item(2,A;1,A) : introduced(A) }.
9 % Only join root-to-leaf paths coinciding on atom truth values
10 ← extend(L,X;L,Y), atom(A), childItem(X,A), f(Y,A), L=1..2.
11 % True atoms are kept
12 item(L,X) ← extend(L,R), childItem(R,X), current(X), L=1..2.
13 % Current clauses
14 notCurr(C) ← pos(C,A), not current(A).
15 notCurr(C) ← neg(C,A), not current(A).
16 curr(C) ← clause(C), not notCurr(C).
17 % Define satisfied clauses
18 sat(L,C) ← curr(C), pos(C,A), item(L,A), L=1..2.
19 sat(L,C) ← curr(C), neg(C,A), not item(L,A), L=1..2.
20 % Current clauses need to be satisfied
21 ← curr(C), not sat(L,C), L=1..2.
22 % Interpretation at level 2 must be subset of that at level 1
23 ← item(2,A), not item(1,A).
24 % Update subset information; reject larger models at root
25 auxItem(2,smaller) ← extend(2,S), childAuxItem(S,smaller).
26 auxItem(2,smaller) ← atom(A), item(1,A), not item(2,A).
27 reject ← final, auxItem(2,smaller).
28 accept ← final, not reject.
```

Listing 2: D-FLAT encoding for solving $\subseteq$-MINIMAL SAT.

kept (Line 12). As in Listing 1, "current" clauses are computed (Lines 14-16), which have to be satisfied both by the partial interpretation stored for the solution candidate and the counterexample (Line 21). Line 23 guarantees that the truth assignment in a counterexample is a subset of (or equal to) that of a solution candidate. Flag `smaller` *denotes that the counterexample represents a proper subset of the solution candidate (Lines 25-26). In the final (i.e., root) node of the TD, solution candidates are rejected that still have a smaller counterexample, and accepted otherwise (Lines 27-28).*

Recall Figure 4 which contains the complete computation for our running example. Our encoding guarantees that the algorithm behaves exactly as described in the previous section. Observe that each model of the encoding matches a root-to-leaf path in the item tree, i.e., it contains exactly one row and subrow entry.

4.2 Towards D-FLAT^2

We implemented our approach in a system called D-FLAT^2 by extending D-FLAT. D-FLAT^2 avoids redundant computations of solution candidates and potential counterexamples. It supports minimization and maximization on user-specified items (e.g., for $\subseteq$-MINIMAL SAT, on atoms). D-FLAT^2 directly makes use of $\mathrm{aug}(\cdot)$ as given in Definition 15; it is thus a problem-independent software framework for subset optimization whose input is (1) an algorithm $\mathcal{A}$ for solving a problem without subset minimization, (2) an instance of this problem and (3) a tree decomposition (which actually can be generated from the input with the help of heuristics). By running $\mathcal{A}$ and transforming the resulting computation $\mathcal{C}$ into $\mathrm{aug}(\mathcal{C})$, Theorem 1 guarantees that the solutions are exactly the subset-minimal ones of $\mathcal{C}$.

Interface. D-FLAT^2 supports the simplified interface of D-FLAT. Internally, instead of using item trees for representing rows and subrows that explicitly represent potential counterexamples, D-FLAT^2 refines them by using so-called *reduced item trees*. Recall item trees and the simplified user interface for problems in NP, since reduced item trees work in a similar way. In a reduced item tree every root-to-leaf path is of length one. For a table R, rows $r \in R$ are stored as usual: each item $i \in \mathrm{D}(r)$ is given by `item`(i) (or `auxItem`(i), respectively). $\mathrm{P}(r)$ is represented by predicates `extend`(p_i) where p_i is a row in the i-th child of R. Instead of storing subrows explicitly, D-FLAT^2 only retains a set of so-called *counterexample pointers* associated with r. For a subrow $s \in \mathrm{S}(r)$, a counterexample pointer is a pair $(c, \mathrm{inc}(s))$, where c is a pointer to some $r' \in R$. By using counterexample pointers, it is no longer required to store counterexamples explicitly. Instead, solution candidates now also serve as countercandidates whenever they are referred to in a set of counterexample pointers. With this, it suffices to update solution candidates during the bottom-up traversal of the TD.

Program flow. Figure 5 illustrates the program flow of D-FLAT^2. The system parses the input instance and computes some tree decomposition heuristically using the Htdecomp library [13]. Then, in a first pass over the TD, solution candidates are computed as in D-FLAT. In contrast to D-FLAT, which computes counterexample candidates directly during the first (and only) pass, D-FLAT^2 traverses the TD a second time. There, the counterexample pointers are computed (along the lines of Definition 15). After the second pass, D-FLAT^2 is able to decide the problem or to materialize the solutions.

The two-pass approach allows us to exclude possible counterexamples that turn out not to correspond to a solution candidate after all. First we compute all rows without their subtables, then we delete rows that do not lead to solutions and finally we apply $\mathrm{aug}(\cdot)$ to the resulting tables. This way, we avoid storing pointers to counterexample candidates that appear in no extension at the root table.

Extensions. D-FLAT^2 generalizes our approach by also supporting problems where only a certain part of the data (instead of all data) is subject to minimization[4]. Towards this, D-FLAT^2 supports a new output predicate `optItem`/1, where `optItem`(i) means that item i is subject to op-

[4]Note that in the previous section we provided definitions and proofs only for the special case where minimization is applied to all data, as the generalization leads to more cumbersome notation but does not change the nature of the approach.

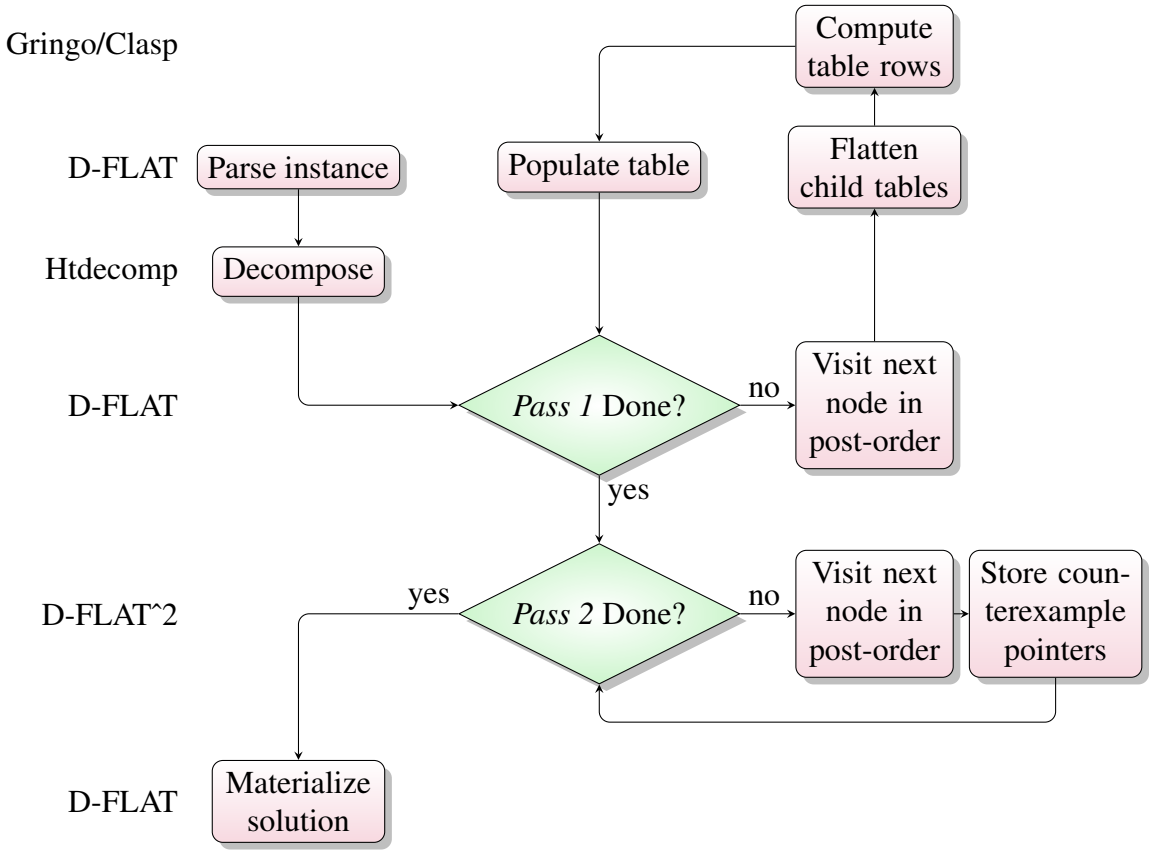


Figure 5: Flowchart for D-FLAT^2.

timization. Using this generalization, we are able to obtain DP algorithms for further AI problems like circumscription [29] (see the next section).

Beside minimization, D-FLAT^2 also supports subset maximization over sets of items. Basically this works analogous to Section 3 by maintaining inclusion flags eq and $\supset$ (instead of $\subset$); subtables are then used to store potential counterexamples to a solution candidate being subset-maximal.

For some problems we require counterexample candidates that are not solution candidates at the same time. An example will be given in Section 5, where we consider disjunctive ASP. There, possible counterexamples are models of a program reduct and not the program itself. In such cases, the rows representing such counterexample candidates can be marked with a special flag as "pseudo solution candidates" by means of the atom `auxItem(pseudo)`, which is taken into account by D-FLAT^2. Then, the only thing required from the user of D-FLAT^2 is an ASP encoding where each model corresponds to a single solution or counterexample candidate marked with `pseudo`.

5 Application to Common AI Problems

In the following we outline some practical applications where D-FLAT^2 is well suited.

5.1 SAT

Program $\Pi_{\texttt{optAllItems}} = \{\texttt{optItem(X)} \leftarrow \texttt{item(X)}\ .\}$ uses the optItem/1 predicate in a trivial way. With this single additional rule at hand, we obtain the D-FLAT^2 encoding

$$\Pi_{\subseteq\text{-MINIMAL SAT}} = \Pi_{\texttt{optAllItems}} \cup \Pi_{\text{SAT}}$$

which allows us to solve $\subseteq$-MINIMAL SAT by simply using the existing encoding for SAT and adding information on what to optimize. In this case, the basic specification $\Pi_{\texttt{optAllItems}}$ suffices, as in $\subseteq$-MINIMAL SAT we consider all items for minimization. D-FLAT^2 encoding $\Pi_{\subseteq\text{-MINIMAL SAT}}$ gives several advantages over the traditional D-FLAT encoding (see Listing 2): It allows us to again use the simplified interface (designed for problems in NP) and the overall length of the encoding is greatly reduced.

5.2 Circumscription

In the propositional case of Circumscription [29], we are given a theory T and sets of atoms P and Z, and we are interested in models M of T such that there is no model M' with $M' \cap P \subset M \cap P$ and $M \cap Z = M' \cap Z$. The formula whose classical models correspond to exactly those solutions is denoted by $\text{CIRC}(T; P; Z)$. We can model Circumscription in our approach by a slight modification of our $\subseteq$-MINIMAL SAT algorithm: We only put an atom x in an optimization item set if $x \in P$; and for any $x \in Z$ we add optimization items $t(x)$ or $f(x)$ if the item set contains x or not, respectively (thus making solution candidates with different interpretations of Z incomparable). By applying this technique we have that for any sibling rows r and r' in a table of the computation, the optimization item set of r is a subset of the one of r' iff $M_r \cap P \subseteq M_{r'} \cap P$ and $M_r \cap Z = M_{r'} \cap Z$, where M_r is a partial interpretation for row r following the EPTs. The program $\Pi_{\texttt{optForCirc}}$ (Listing 3) takes these considerations into account; moreover,

$$\Pi_{\text{CIRC}} = \Pi_{\texttt{optForCirc}} \cup \Pi_{\text{SAT}}$$

then gives a D-FLAT^2 implementation of the DP algorithm for Circumscription. As input it expects the theory T to be given as a CNF formula like for Π_{SAT} and the sets P and Z to be given using the predicates p/1 and z/1, respectively.

```
1 optItem(X)     ←      item(X), p(X).
2 optItem(t(X))  ←      item(X), z(X).
3 optItem(f(X))  ← not  item(X), z(X), current(X).
```

Listing 3: $\Pi_{\texttt{optForCirc}}$: used for solving Circumscription via $\Pi_{\text{CIRC}} = \Pi_{\texttt{optForCirc}} \cup \Pi_{\text{SAT}}$.

5.3 Disjunctive ASP

While a traditional TD-based DP algorithm for solving disjunctive ASP can be found in [24], here we solve the problem with D-FLAT^2. We first do so via reduction to Circumscription. In the following, for any interpretation, rule or set of atoms X, we write X' to denote the result of replacing each atom a in X with a new atom a'. Given a disjunctive logic program Π consisting of rules of the form $a_1 \vee a_2 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, not\ b_{m+1}, \ldots, not\ b_n$, we use the notation $h(r) := \bigvee_{1 \leq i \leq k} a_i$, $b^+(r) := \bigwedge_{1 \leq i \leq m} b_i$ and $b^-(r) := \bigwedge_{m+1 \leq i \leq n} \neg b_i$.

As shown in [31], an interpretation I is an answer-set of Π iff $I \cup I'$ is a model of $\bigwedge_{p \in HB(\Pi)}(p \equiv p') \wedge \textsc{Circ}(\bigcup_{r \in \Pi}\{((b^+(r) \wedge b^-(r')) \rightarrow h(r))\}; HB(\Pi); HB(\Pi)')$, where $HB(\Pi)$ denotes the Herbrand base of Π. In order to compute models of this formula, we can first calculate models of the Circumscription part and then remove those models (using the "`pseudo`" item for marking pseudo solution candidates), where the truth value of some atom a is different from the one of a'. This amounts to $\Pi_{\texttt{pseudoForASP}}$ (Listing 4), which can be used for solving disjunctive ASP by means of the combined program $\Pi_{\texttt{ASP}} = \Pi_{\texttt{pseudoForASP}} \cup \Pi_{\textsc{Circ}}$. The predicate `cor` in this encoding is assumed to be symmetric and occurring in input facts in order to associate each atom a with its corresponding primed variant a'. (We require that a and a' always occur together in some bag, which can be achieved by adding an edge (a, a') to the input graph.)

```
1 auxItem(pseudo) ← cor(A,B), current(A;B), item(A), not item(B).
2 auxItem(pseudo) ← extend(R), childAuxItem(R,pseudo).
```

Listing 4: $\Pi_{\texttt{pseudoForASP}}$: used for solving disjunctive ASP via $\Pi_{\texttt{ASP}} = \Pi_{\texttt{pseudoForASP}} \cup \Pi_{\textsc{Circ}}$.

In Listing 5, we present an alternative approach to solving disjunctive ASP, which does not resort to Circumscription. In this encoding, $\Pi'_{\texttt{ASP}}$, we generate solution candidates for all interpretations that are candidates for being a classical model of the input program, which is specified by means of the predicates `head`, `pos` and `neg`. A classical model M of a program P might be no answer set because some $M' \subset M$ is a model of the reduct P^M. To check this, we generate additional rows that only serve as counterexample candidates (like M') to the rows representing classical model candidates (like M). For any atom a from the current bag, if an item set contains a, then the corresponding interpretation sets a to true (otherwise to false). The rows representing potential counterexamples can additionally contain items of the form $\texttt{r}(a)$. This signifies that the atom a is false in the respective counterexample candidate but true in the classical model candidates that reference this counterexample candidate (by means of their counterexample pointers). In Lines 22 and 24, we make sure that any row containing an item $\texttt{r}(a)$ is marked with the "`pseudo`" item and will therefore not be considered as a solution but rather serves as a counterexample candidate only. Lines 27 and 28 are required to ensure that any counterexample candidate C of any solution candidate M only contains $r(a)$ for atoms a that are also contained in M.

```
1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
% Guess truth value/rule flag of introduced atoms
0 { item(A;r(A)) : atom(A), introduced(A) } 1.
% Make explicit when an atom is false or a rule is unsat
false(R,X)  ← childRow(R,N), bag(N,X), not childItem(R,X).
falser(R,X) ← childRow(R,N), bag(N,X), not childItem(R,r(X)).
unsat(R,X)  ← childRow(R,N), bag(N,X), not childAuxItem(R,X).
% Only join child item sets that coincide on common atoms
← extend(X;Y), atom(A), childItem(X,A),    false(Y,A).
← extend(X;Y), atom(A), childItem(X,r(A)), falser(Y,A).
% Only extend child item sets satisfying all removed rules
← extend(S), rule(X), removed(X), unsat(S,X).
% True atoms and satisfied rules remain so unless removed
item(X)    ← extend(S), childItem(S,X),    current(X).
item(r(X)) ← extend(S), childItem(S,r(X)), current(X).
% Through the guess, rules may become satisfied
auxItem(R) ← current(R;A), head(R,A),     item(A).
auxItem(R) ← current(R;A), pos(R,A),  not item(A).
auxItem(R) ← current(R;A), neg(R,A),      item(A).
% Rule is not in reduct if a negative body atom is set to true
auxItem(R) ← current(R;A), neg(R,A),      item(r(A)).
auxItem(pseudo) ← item(r(X)), current(X).
% Inherit pseudo flag from child nodes
auxItem(pseudo) ← extend(R), childAuxItem(R,pseudo).
optItem(S)    ← atom(S), item(S).
% Prevents r(S) at level 2 (reduct) if S is not true at level 1
optItem(r(S)) ← atom(S), item(S), not auxItem(pseudo).
optItem(r(S)) ← atom(S), item(r(S)).
```

Listing 5: $\Pi'_{\texttt{ASP}}$: D-FLAT^2 encoding for solving disjunctive ASP directly.

5.4 Abstract Argumentation

Problems from abstract argumentation [15] are further examples where our approach is reasonable. Given an object (A, R), where A is a set of arguments and $R \subseteq A \times A$, we call a set $S \subseteq A$ *admissible* if (1) $(a, b) \notin R$ for all $a, b \in S$ and (2) for each $s \in S$ and $r \in A$, $(r, s) \in R$ implies that there is some $q \in S$ with $(q, r) \in R$. S is *preferred* if it is a subset-maximal admissible set. For any $C \subseteq A$, we call $C^+ = C \cup \{a \mid \exists b \in C \text{ s.t. } (b, a) \in R\}$ the *range* of C. A set S is *semi-stable* if it is admissible and for every admissible $S' \subset S$, $S^+ \not\subset S'^+$ holds.

Listing 6 shows an encoding $\Pi_{\texttt{admissible}}$ for computing admissible sets. At first glance, it seems to be overcomplicated compared to the algorithm in [16]: For computing admissible sets it actually suffices to guess which introduced arguments are in S, whereas we guess in $\Pi_{\texttt{admissible}}$ which arguments are in the set, attackers or neither. In order for a guessed set to be a solution, every attacker has to be defeated by the time it is removed from the bag. Once it can be determined

that an attacker is defeated, its status changes from "attc" to "def". This additional complexity allows us to reuse $\Pi_{\texttt{admissible}}$ for computing semi-stable sets later on.

```
1 { extend(R)   : childRow(R,N) } 1 ← childNode(N).
% Guess whether an element is in, out or attacking (attc)
0 { item(in(A)); attc(A) } 1       ← introduced(A).
% Join only arguments with compatible status
nIn(S,A)  ← childRow(S,N), bag(N,A), not childItem(S,in(A)).
nDef(S,A) ← childRow(S,N), bag(N,A), not childAuxItem(S,def(A)), not
    childAuxItem(S,attc(A)).
← extend(S1), extend(S2), childItem(S1,in(A)),      nIn(S2,A).
← extend(S1), extend(S2), childAuxItem(S1,def(A)),  nDef(S2,A).
← extend(S1), extend(S2), childAuxItem(S1,attc(A)), nDef(S2,A).
% Inherit arguments that are in, defeated or attackers
item(in(A)) ← extend(S), childItem(S,in(A)),      current(A).
chdef(A)    ← extend(S), childAuxItem(S,def(A)),  current(A).
attc(A)     ← extend(S), childAuxItem(S,attc(A)), current(A).
% Set defeated arguments
auxItem(def(A))  ← current(A;B), att(B,A), item(in(B)).
auxItem(def(A))  ← chdef(A).
% Still remaining (undefeated) attackers
auxItem(attc(A)) ← attc(A), not auxItem(def(A)).
% Out-arguments are not allowed to be defeated/attackers
out(A) ← not attc(A), not chdef(A), current(A).
← auxItem(def(A)), out(A).
← out(A), current(A), item(in(B)), att(A,B).
% Assure that the set is conflict-free
← item(in(A)), item(in(B)), att(A,B).
% Remove candidates that leave attackers undefeated
← extend(S), childAuxItem(S,attc(A)), removed(A).
```

Listing 6: $\Pi_{\texttt{admissible}}$: D-FLAT encoding for admissible sets.

Using $\Pi_{\texttt{admissible}}$, we can compute preferred sets by simple subset maximization via

$$\Pi_{\texttt{preferred}} = \Pi_{\texttt{optAllItems}} \cup \Pi_{\texttt{admissible}}.$$

Furthermore, it is now also easy to compute semi-stable sets by means of $\Pi_{\texttt{optForSemiStable}}$ (Listing 7), using

$$\Pi_{\texttt{semiStable}} = \Pi_{\texttt{optForSemiStable}} \cup \Pi_{\texttt{admissible}}.$$

Although the code $\Pi_{\texttt{admissible}}$ could have been simplified for computing preferred sets, here it is indeed required because we need to find those admissible sets that have maximal range.

```
optItem(A) ← item(in(A)).
optItem(A) ← auxItem(attc(A)).
optItem(A) ← auxItem(def(A)).
```

Listing 7: $\Pi_{\texttt{optForSemiStable}}$: used for computing semi-stable sets.

6 Evaluation

In this section we compare our new system D-FLAT^2 (version 1.0.2) with D-FLAT (version 1.0.1). Both frameworks internally use ASP grounder Gringo 4.4.0 and solver Clasp 3.1.1. Additionally, we compare the performance to well-established systems for the respective problem domains.

6.1 Abstract Argumentation

This part focuses on experiments for problems from the area of abstract argumentation. In addition to D-FLAT and D-FLAT^2, we benchmarked the ASPARTIX system [17] that solves argumentation-related problems directly via ASP. The results for ASPARTIX were produced with Gringo 3.0.5, as it is not fully compatible with newer versions of Gringo. For evaluation we used so-called "grid-based" instances, where vertices are arranged on a $n \times m$ matrix, and edges connect horizontally, vertically and diagonally neighboring vertices. Each instance was run five times with different TDs, and every run was limited to one hour and three GB of memory.

System comparison. We considered the problem of enumerating all preferred extensions and compared the systems on grid-based instances with 40 to 65 nodes and treewidth 4. Figure 6 illustrates average runtimes and allocated memory together with the 95 % confidence interval. D-FLAT^2 showed the best performance, while D-FLAT is slightly slower and requires more memory. For ASPARTIX we observed timeouts for instances having more than 55 nodes.

Problem comparison. As D-FLAT^2 is based on D-FLAT, we compared these systems on several problems using grid-based instances with treewidth 4. Moreover, we analyzed the cost of computing preferred and semi-stable sets compared to only obtaining admissible sets. As instances have much more admissible sets than preferred sets, which would bias a performance comparison when doing explicit enumeration, we considered the counting variants of these problems. Results are summarized in Figure 7.

When counting admissible sets, D-FLAT^2 requires slightly more time and memory than D-FLAT due to the overhead imposed by using reduced item trees instead of item trees. For preferred sets, the inefficiency of computing redundant counter candidates in D-FLAT becomes evident. On the contrary, in D-FLAT^2 the difference in runtime for counting preferred instead of admissible sets is barely measurable (i.e. within the 95% confidence interval). Here, we observed that subset maximization comes for free for instances of small treewidth. Finally, for semi-stable sets, D-FLAT was not able solve instances with 500 vertices within the given memory limits. One reason is that for this problem many potential counter candidates have to be computed that turn out to be not even admissible. Thus, our two-phased approach of first computing (not necessarily maximal) solutions and then performing maximization obviously pays off in this case.

6.2 Subset-minimal SAT

Comparison to QBF. We also performed experiments for counting minimal models of randomly generated 3-CNF instances (conjunctive normal form with 3 literals in every clause). Results were compared to depQBF 4 [28], a state-of-the-art QBF solver and bloqqer 35 [22] in combination with

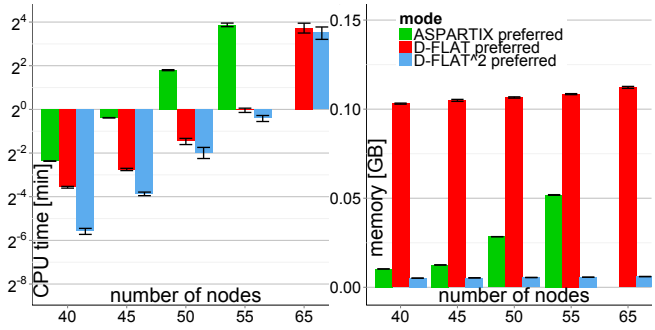


Figure 6: System comparison: Average CPU time (left) and maximum resident set (right).

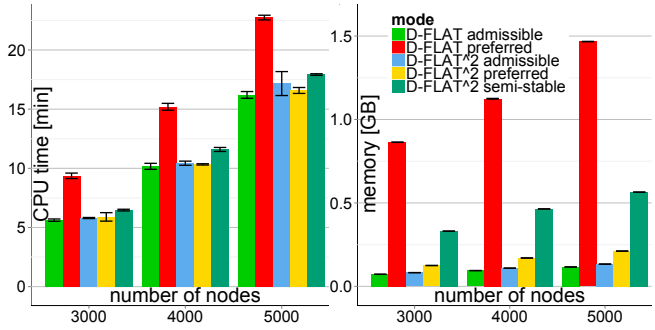


Figure 7: Problem comparison: Average CPU time (left) and maximum resident set (right).

depQBF, a tool for preprocessing QBFs. We considered randomly generated, satisfiable instances with 26 clauses and a varying number of variables. For each number of variables, five instances were generated and run three times. The treewidth of all generated instances did not exceed the value of 9. Figure 8 illustrates the obtained results, including the 95% confidence interval. The number of variables is shown on the x-axis.

Although bloqqer with depQBF is extremely fast, the preprocessing potentially yields simplifications which are not equivalence-preserving. Thus, bloqqer might not be well-suited for some applications. Furthermore, we let depQBF and bloqqer only output the first satisfying assignment (marked with an asterisk in the legend of the plot), since counting solutions is not directly supported by these tools. The actual runtime would, of course, increase in case of counting (via enumeration) since these tools require to incrementally call the solver until the formula (modified by previously computed solutions) is unsatisfiable. Figure 8 also reveals that (for instances of small treewidth) subset optimization in D-FLAT^2 almost comes for free, since SAT and $\subseteq$-MINIMAL SAT require approximately the same amount of resources. Note that we are going to conduct further experiments for the most recent version of depQBF released after submission deadline.

Comparison to ASP. In addition to before, we also counted minimal models by means of ASP. We used Potassco (Gringo 4.4.0 and Clasp 3.1.1) as a representative candidate among ASP solvers. Here, we again considered satisfiable 3-CNF instances. To guarantee an upper bound of 4 for the treewidth, we generated the instances following a grid-based structure (as above). Each bar in

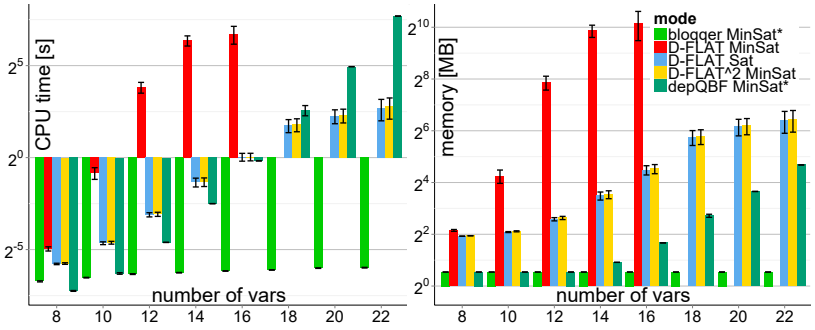


Figure 8: Comparison to QBF: Average CPU time (left) and maximum resident set (right).

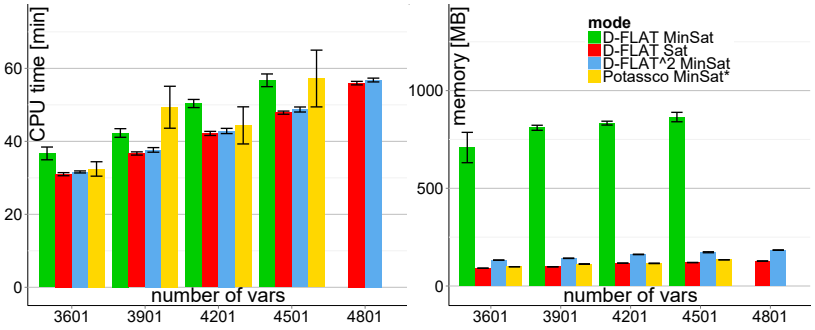


Figure 9: Comparison to ASP: Average CPU time (left) and maximum resident set (right).

Figure 9 presents the average time (or memory) over the five instances per number of variables and three runs per instance assuming a timeout of one hour. Additionally, the 95% confidence interval is given.

Again, Potassco only had to print the first satisfying assignment (marked with an asterisk), i.e. the actual runtime would increase in case of enumeration/counting. Note that the ASP approach is quite fast, but with instances of small treewidth, D-FLAT^2 quickly shows its advantage. Moreover, observe that compared to D-FLAT^2, Potassco shows potentially increased runtime consumption and a huge runtime deviation.

7 Conclusion

To put FPT results for bounded treewidth to use in the AI domain, often DP algorithms for problems involving tasks like subset minimization have to be designed. These algorithms exhibit common properties that are tedious to specify but can be automatically taken care of, as we have shown in this paper. In fact, we have provided a translation that turns a given DP algorithm for computing a set S of solution candidates (say, models of a formula) into a DP algorithm that computes only the subset-minimal elements of S (e.g., minimal models). We have shown the translation to be sound and to remain FPT whenever the original DP is. This is indeed superior to a naive way that computes all elements from S first and then filters out minimal ones in a post-processing step,

which would not yield an FPT algorithm in general.

We presented the D-FLATˆ2 system for DP on TDs realizing this idea with further generalizations (e.g., performing minimization only on a given set of atoms). Users of D-FLATˆ2 are only required to provide an ASP program that specifies an algorithm for a version of the problem without optimization. Our method then performs the optimization tasks in an automatic and uniform way, thus making the development of such algorithms significantly easier. Preliminary experiments indicate that our new approach brings significant advantages in terms of time and memory compared to previous solutions.

For future work, we would like to investigate the potential of other built-ins for DP algorithms; for instance, checks for connectedness could be treated in a similar way. In the long run, we anticipate a system that facilitates implementing DP algorithms but (in contrast to related systems such as *Sequoia*) keeps the overall design of the concrete DP algorithm in the user's hands.

References

[1] Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, Markus Hecher, and Stefan Woltran. D-FLAT: Progress report. Technical Report DBAI-TR-2014-86, TU Wien, 2014.

[2] Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, Markus Hecher, and Stefan Woltran. The D-FLAT system for dynamic programming on tree decompositions. In *Proc. JELIA*, volume 8761 of *LNCS*, pages 558–572, 2014.

[3] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, 1987.

[4] Bernhard Bliem, Günther Charwat, Markus Hecher, and Stefan Woltran. D-FLATˆ2: Subset minimization in dynamic programming on tree decompositions made easy. In *Proc. ASPOCP'15*, 2015.

[5] Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996.

[6] Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations I. Upper bounds. *Inf. Comput.*, 208(3):259–275, 2010.

[7] Gerhard Brewka, James P. Delgrande, Javier Romero, and Torsten Schaub. asprin: Customizing answer set preferences without a headache. In *Proc. AAAI*, pages 1467–1474. AAAI Press, 2015.

[8] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.

[9] Bruno Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990.

[10] Bruno Courcelle and Irène Durand. Computations by fly-automata beyond monadic second-order logic. *CoRR*, abs/1305.7120, 2013.

[11] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.

[12] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[13] Artan Dermaku, Tobias Ganzow, Georg Gottlob, Benjamin J. McMahan, Nysret Musliu, and Marko Samer. Heuristic methods for hypertree decomposition. In *Proc. MICAI*, volume 5317 of *LNCS*, pages 1–11. Springer, 2008.

[14] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999.

[15] Phan M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321–357, 1995.

[16] Wolfgang Dvořák, Reinhard Pichler, and Stefan Woltran. Towards fixed-parameter tractable algorithms for abstract argumentation. *Artif. Intell.*, 186:1–37, 2012.

[17] Uwe Egly, Sarah Alice Gaggl, and Stefan Woltran. Answer-set programming encodings for argumentation frameworks. *Argument and Computation*, 1(2):147–177, 2010.

[18] Thomas Eiter and Axel Polleres. Towards automated integration of guess and check programs in answer set programming: A meta-interpreter and applications. *TPLP*, 6(1-2):23–60, 2006.

[19] Martin Gebser, Roland Kaminski, and Torsten Schaub. Complex optimization in answer set programming. *TPLP*, 11(4-5):821–839, 2011.

[20] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Thomas Schneider. Potassco: The potsdam answer set solving collection. *AI Commun.*, 24(2):107–124, 2011.

[21] Georg Gottlob, Reinhard Pichler, and Fang Wei. Tractable database design and datalog abduction through bounded treewidth. *Inf. Syst.*, 35(3):278–298, 2010.

[22] Marijn Heule, Martina Seidl, and Armin Biere. A unified proof system for QBF preprocessing. In *Proc. IJCAR*, volume 8562 of *LNCS*, pages 91–106. Springer, 2014.

[23] Michael Jakl, Reinhard Pichler, Stefan Rümmele, and Stefan Woltran. Fast counting with bounded treewidth. In *Proc. LPAR*, volume 5330 of *LNCS*, pages 436–450. Springer, 2008.

[24] Michael Jakl, Reinhard Pichler, and Stefan Woltran. Answer-set programming with bounded treewidth. In *Proc. IJCAI*, pages 816–822, 2009.

[25] Ton Kloks. *Treewidth: Computations and Approximations*, volume 842 of *LNCS*. Springer, 1994.

[26] Joachim Kneis, Alexander Langer, and Peter Rossmanith. Courcelle's theorem – a game-theoretic approach. *Discrete Optimization*, 8(4):568–594, 2011.

[27] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[28] Florian Lonsing and Uwe Egly. Depqbf: An incremental QBF solver based on clause groups. *CoRR*, abs/1502.02484, 2015.

[29] John McCarthy. Circumscription – a form of non-monotonic reasoning. *Artif. Intell.*, 13(12):27–39, 1980.

[30] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford Lecture Series in Mathematics and its Applications. OUP, 2006.

[31] David Pearce, Hans Tompits, and Stefan Woltran. Characterising equilibrium logic and nested logic programs: Reductions and complexity. *TPLP*, 9(5):565–616, 2009.

[32] Neil Robertson and Paul D. Seymour. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.

[33] Marko Samer and Stefan Szeider. Algorithms for propositional model counting. *J. Discrete Algorithms*, 8(1):50–64, 2010.

A Appendix: Proofs

Lemma 1. *If a table R is normal, then so is* $\mathrm{compr}(R)$.

Proof. Let R be a normal table, let $R' = \mathrm{compr}(R)$, and let $R_1, \ldots, R_k$ denote the child tables of R (and R'). As for the first condition of normality of R', let $r' \in R'$, $s' \in \mathrm{S}(r')$, $X' \in \mathrm{E}(r')$ and $Y' \in \mathrm{E}_{X'}(s')$. For $1 \leq i \leq k$, the set $X' \cap R_i$ consists of a single element r_i, as can be seen via Definition 3. Similarly, there is a subrow s_i such that $Y' \cap \bigcup_{t \in R_i} \mathrm{S}(t) = \{s_i\}$. As Y' is an extension relative to X', $s_i \in \mathrm{S}(r_i)$ holds. The fact that r_i and s_i are part of extensions of r' and s', respectively, entails that $(r_1, \ldots, r_k) \in \mathrm{P}(r')$ and $(s_1, \ldots, s_k) \in \mathrm{P}(s')$. From the definition of $\mathrm{compr}(\cdot)$ we infer that then there is a row $r \in R$ having a subrow $s \in \mathrm{S}(r)$ such that $r \approx r'$, $s \approx s'$ and $(s_1, \ldots, s_k) \in \mathrm{P}(s)$. It must hold by Definition 2 that $(r_1, \ldots, r_k) \in \mathrm{P}(r)$ because $s \in \mathrm{S}(r)$, $(s_1, \ldots, s_k) \in \mathrm{P}(s)$ and $s_i \in \mathrm{S}(r_i)$. As $(r_1, \ldots, r_k) \in \mathrm{P}(r)$ and $(s_1, \ldots, s_k) \in \mathrm{P}(s)$, there are $X \in \mathrm{E}(r)$ and $Y \in \mathrm{E}_X(s)$ with $X \approx X'$ and $Y \approx Y'$. As R is normal, $Y \leq X$ holds, which proves that $Y' \leq X'$. By definition of $\mathrm{compr}(\cdot)$, it holds that $\mathrm{inc}(s) = \subset$ if and only if $\mathrm{inc}(s') = \subset$. Furthermore, due to normality of R, $Y < X$ holds if and only if $\mathrm{inc}(s) = \subset$. This proves that $Y' < X'$ holds if and only if $\mathrm{inc}(s') = \subset$.

To prove the second condition, let $r' \in R'$, $s' \in \mathrm{S}(r')$ and $Y' \in \mathrm{E}(s')$. Using the same reasoning as before, we can see that there is a subrow s of some row in R such that $s \approx s'$, and there are elements $s_i \in Y'$ such that $(s_1, \ldots, s_k) \in \mathrm{P}(s)$. Hence there is an extension $Y \in \mathrm{E}(s)$ with $Y \approx Y'$. By normality of R, there is a row $t \in R$ with $t \approx s$ and an extension $T \in \mathrm{E}(t)$ with $T \approx Y$. Similar to before, then there are elements $t_i \in T$ such that $(t_1, \ldots, t_k) \in \mathrm{P}(t)$. By definition of $\mathrm{compr}(\cdot)$, then there is a row $t' \in R'$ with $t' \approx t$ and $(t_1, \ldots, t_k) \in \mathrm{E}(t')$. Furthermore, there are $T_i \in \mathrm{E}(t_i)$ such that $T = \{t\} \cup T_1 \cup \cdots \cup T_k$. We can construct $T' = (T \setminus \{t\}) \cup \{t'\}$ and observe that $T' \in \mathrm{E}(t')$ and $T' \approx T = Y \approx Y'$ as well as $t' \approx s'$.

As for the third condition, let $q', r' \in R'$, $Z' \in \mathrm{E}(q')$ and $X' \in \mathrm{E}(r')$ such that $Z' \leq X'$. Similar to before, then there are rows $q, r \in R$ such that $q \approx q'$ and $r \approx r'$, there are elements $q_i \in Z'$ such that $(q_1, \ldots, q_k) \in \mathrm{P}(q)$, and there are elements $r_i \in X'$ such that $(r_1, \ldots, r_k) \in \mathrm{P}(r)$. Hence there are $Z \in \mathrm{E}(q)$ and $X \in \mathrm{E}(r)$ with $Z \approx Z'$ and $X \approx X'$, so $Z \leq X$. By normality of R, then there exist $u \in \mathrm{S}(r)$ and $U \in \mathrm{E}_X(u)$ such that $u \approx q$ and $U \approx Z$, and there are $u_i \in U$ such that $(u_1, \ldots, u_k) \in \mathrm{P}(u)$. Then by definition of $\mathrm{compr}(\cdot)$, there is a subrow $u' \in \mathrm{S}(r')$ with $u' \approx u$ and $(u_1, \ldots, u_k) \in \mathrm{P}(u')$. So $u' \approx q'$ and there is an extension $U' \in \mathrm{E}_{X'}(u')$ with $U' \approx U = Z \approx Z'$, which concludes the proof. □

Furthermore, we show that the extensions from rows in a table R are in a one-to-one correspondence to the extensions of rows from $\mathrm{compr}(R)$ such that the data of corresponding extensions is the same.

Lemma 4. *Let R be a table. Then for any $r \in R$ and $X \in \mathrm{E}(r)$ there are $r' \in \mathrm{compr}(R)$ and $X' \in \mathrm{E}(r')$ such that $r' \approx r$ and $X' \approx X$. Also, for any $r' \in \mathrm{compr}(R)$ and $X' \in \mathrm{E}(r')$ there are $r \in R$ and $X \in \mathrm{E}(r)$ such that $r \approx r'$ and $X \approx X'$.*

Proof. Let R be a table and $R' = \mathrm{compr}(R)$. For the first statement, let $r \in R$ and $X \in \mathrm{E}(r)$. Then there are $(r_1, \ldots, r_k) \in \mathrm{P}(r)$ and X_i such that $X = \{r\} \cup X_1 \cup \cdots \cup X_k$. By definition of

$\mathrm{compr}(\cdot)$, there is $r' \in R'$ with $r' \approx r$ and $(r_1, \ldots, r_k) \in \mathrm{P}(r')$. Then $X' = \{r'\} \cup X_1 \cup \cdots \cup X_k$ is in $\mathrm{E}(r')$ and $X' \approx X$. The second statement is proved symmetrically. □

As explained after our definition of $\mathrm{aug}(\cdot)$, each row and subrow in the (compressed) augmentation result has a unique originating row. We formalize this as follows.

Definition 16. *Let R be a table in some augmentable computation, let $Q = \mathrm{aug}(R)$ and $Q' = \mathrm{compr}(Q)$. We define a function $\mathrm{orig}_R(\cdot)$ that maps each row or subrow in Q or Q' to a row in R. For $q \in Q \cup Q'$ we define $\mathrm{orig}_R(q)$ to be the unique $r \in R$ with $r \approx q$. For $s \in \mathrm{S}(q)$ we define $\mathrm{orig}_R(s)$ to be the unique $r \in R$ with $r \approx s$.*

Let R be a table from an augmentable computation. The following lemma formalizes that for each (sub)row q in $\mathrm{aug}(R)$ with $(q_1, \ldots, q_k) \in \mathrm{P}(q)$, the originating row $\mathrm{orig}_R(q)$ in R has a corresponding EPT $(r_1, \ldots, r_k)$ such that each q_i is resulting from r_i. This will later be useful to establish that for each extension in Q we can also obtain one in R having the same data.

Lemma 5. *Let R be a table from an augmentable computation and let the child tables of R be called $R_1, \ldots, R_k$. We have $\big(\mathrm{orig}_{R_1}(q_1), \ldots, \mathrm{orig}_{R_k}(q_k)\big) \in \mathrm{P}(\mathrm{orig}_R(q))$ for any $q \in \mathrm{aug}(R)$ and $(q_1, \ldots, q_k) \in \mathrm{P}(q)$. Furthermore, for any $s \in \mathrm{S}(q)$ and $(s_1, \ldots, s_k) \in \mathrm{P}(s)$ it holds that $\big(\mathrm{orig}_{R_1}(s_1), \ldots, \mathrm{orig}_{R_k}(s_k)\big) \in \mathrm{P}(\mathrm{orig}_R(s))$.*

Proof. Let R be a table in some augmentable computation such that $R_1, \ldots, R_k$ denote the child tables of R, and let $Q = \mathrm{aug}(R)$ with child tables $Q_i = \mathrm{compr}(\mathrm{aug}(R_i))$. Moreover, let $q \in Q$ and $(q_1, \ldots, q_k) \in \mathrm{P}(q)$. By construction of $\mathrm{aug}(R)$, then there are $r \in R$ and $(r_1, \ldots, r_k) \in \mathrm{P}(r)$ such that $r \approx q$ and $q_i \in \mathrm{res}(r_i)$ for each $1 \leq i \leq k$. But then $q_i \approx r_i$, so $\mathrm{orig}_{R_i}(q_i) = r_i$ holds. Furthermore, $\mathrm{orig}_R(q) = r$ holds since $q \approx r$. Because of $(r_1, \ldots, r_k) \in \mathrm{P}(r)$, this entails $\big(\mathrm{orig}_{R_1}(q_1), \ldots, \mathrm{orig}_{R_k}(q_k)\big) \in \mathrm{P}(\mathrm{orig}_R(q))$.

As for the second statement, let $s \in \mathrm{S}(q)$ and $(s_1, \ldots, s_k) \in \mathrm{P}(s)$. By construction of $\mathrm{aug}(R)$, then there are $q' \in Q$ and $(q'_1, \ldots, q'_k) \in \mathrm{P}(q')$ such that $q' \approx s$ and $q'_i \approx s_i$ for each $1 \leq i \leq k$. We know that this entails $\big(\mathrm{orig}_{R_1}(q'_1), \ldots, \mathrm{orig}_{R_k}(q'_k)\big) \in \mathrm{P}(\mathrm{orig}_R(q'))$ by the first statement of this lemma. As we have seen that $q'_i \approx s_i$ and $q' \approx s$, it must hold that $\mathrm{orig}_R(s) = \mathrm{orig}_R(q')$ and $\mathrm{orig}_{R_i}(s_i) = \mathrm{orig}_{R_i}(q'_i)$, which proves the second statement. □

Lemma 2. *Let R be a table from an augmentable computation and $Q = \mathrm{aug}(R)$. Then for any $r \in R$ and $Z \in \mathrm{E}(r)$ there are $q \in Q$ and $X \in \mathrm{E}(q)$ such that $r \approx q$ and $Z \approx X$. Also, for any $q \in Q$ and $X \in \mathrm{E}(q)$ there are $r \in R$ and $Z \in \mathrm{E}(r)$ such that $q \approx r$ and $X \approx Z$.*

Proof. Let R be a table in some augmentable computation such that $R_1, \ldots, R_k$ denote the child tables of R, and let $Q = \mathrm{aug}(R)$ with child tables $Q_i = \mathrm{compr}(\mathrm{aug}(R_i))$. We prove the lemma by induction. First suppose R and Q are leaf tables. Then it can be easily verified using the definition of Q that the rows in Q and R are in a one-to-one correspondence. Formally, $|Q| = |R|$ holds and for each $r \in R$ there is a $q \in Q$ with $q \approx r$. As the EPTs in R and Q always consist of just the empty tuple, the data of any extension of a row in R or Q coincides with the data of that row.

Now suppose R and Q are arbitrary tables and both statements from the lemma hold for all R_i. We first attend to the first statement. Let $r \in R$ and $Z \in \mathrm{E}(r)$. Then there are $(r_1, \ldots, r_k) \in \mathrm{P}(r)$

and $Z_i \in \mathrm{E}(r_i)$ such that $Z = \{r\} \cup Z_1 \cup \cdots \cup Z_k$. By the induction hypothesis, there are $q_i \in Q_i$ and $X_i \in \mathrm{E}(q_i)$ such that $r_i \approx q_i$ and $X_i \approx Z_i$. As $r_i \approx q_i$, it holds that $q_i \in \mathrm{res}(r_i)$. Hence condition 1 from the construction of Q ensures that there is a row $q \in Q$ with $q \approx r$ and $\mathrm{P}(q) = \{(q_1, \ldots, q_k)\}$. Now clearly $X = \{q\} \cup X_1 \cup \cdots \cup X_k$ is contained in $\mathrm{E}(q)$, and $X \approx Z$.

As for the second statement, let $q \in Q$ and $X \in \mathrm{E}(q)$. Then there are $(q_1, \ldots, q_k) \in \mathrm{P}(q)$ and $X_i \in \mathrm{E}(q_i)$ such that $X = \{q\} \cup X_1 \cup \cdots \cup X_k$. By the induction hypothesis, there are $r_i \in R_i$ and $Z_i \in \mathrm{E}(r_i)$ such that $q_i \approx r_i$ and $X_i \approx Z_i$. Then $\mathrm{orig}_{R_i}(q_i) = r_i$ holds because of $q_i \approx r_i$. Let $r = \mathrm{orig}_R(q)$, so obviously $q \approx r$. By Lemma 5, $(q_1, \ldots, q_k) \in \mathrm{P}(q)$ entails $\big(\mathrm{orig}_{R_1}(q_1), \ldots, \mathrm{orig}_{R_k}(q_k)\big) \in \mathrm{P}(\mathrm{orig}_R(q))$, i.e., $(r_1, \ldots, r_k) \in \mathrm{P}(r)$. Now clearly $Z = \{r\} \cup Z_1 \cup \cdots \cup Z_k$ is contained in $\mathrm{E}(r)$, and $X \approx Z$. □

For subrows we show a result similar to the second statement from Lemma 2.

Lemma 6. *Let R be a table from an augmentable computation and $Q = \mathrm{aug}(R)$. Then for any $q \in Q$, $s \in \mathrm{S}(q)$ and $Y \in \mathrm{E}(s)$ there are $r \in R$ and $Z \in \mathrm{E}(r)$ such that $s \approx r$ and $Y \approx Z$.*

Proof. Let R be a table in some augmentable computation such that $R_1, \ldots, R_k$ denote the child tables of R, and let $Q = \mathrm{aug}(R)$ with child tables $Q_i = \mathrm{compr}(\mathrm{aug}(R_i))$. We prove the lemma by induction. First suppose R and Q are leaf tables. By definition of Q, for each subrow s in Q there is some $q' \in Q$ with $s \approx q'$. Obviously, $q' \approx \mathrm{orig}_R(q')$ and $\mathrm{orig}_R(s) = \mathrm{orig}_R(q')$, so $s \approx \mathrm{orig}_R(s)$, which proves the base case.

Now suppose R and Q are arbitrary tables and the statement holds for all R_i. Let $q \in Q$, $s \in \mathrm{S}(q)$, $Y \in \mathrm{E}(s)$ and $r = \mathrm{orig}_R(s)$. Then there are $(s_1, \ldots, s_k) \in \mathrm{P}(s)$ and $Y_i \in \mathrm{E}(s_i)$ such that $Y = \{s\} \cup Y_1 \cup \cdots \cup Y_k$. By the induction hypothesis, there are $r_i \in R_i$ and $Z_i \in \mathrm{E}(r_i)$ such that $s_i \approx r_i$ and $Y_i \approx Z_i$. Then $\mathrm{orig}_{R_i}(s_i) = r_i$. By Lemma 5, $(s_1, \ldots, s_k) \in \mathrm{P}(s)$ entails $\big(\mathrm{orig}_{R_1}(s_1), \ldots, \mathrm{orig}_{R_k}(s_k)\big) \in \mathrm{P}(\mathrm{orig}_R(s))$, so $(r_1, \ldots, r_k) \in \mathrm{P}(r)$. Now clearly $Z = \{r\} \cup Z_1 \cup \cdots \cup Z_k$ is contained in $\mathrm{E}(r)$, and $Y \approx Z$. □

Lemma 3. *Let R be a table from an augmentable computation. Then the table $\mathrm{aug}(R)$ is normal.*

Proof. Let R be a table in some augmentable computation such that $R_1, \ldots, R_k$ denote the child tables of R and let $Q = \mathrm{aug}(R)$. We inductively show that Q is normal if each $\mathrm{aug}(R_i)$ is normal.

Suppose R and Q are leaf tables. Then the EPTs of each row in R consist of just the empty tuple, which makes the definition of Q in this case equivalent to the following: For any $r \in R$, there is a row $q \in Q$ with $q \approx r$ and $\mathrm{P}(q) = \{()\}$. Furthermore, for each $q, q' \in Q$ with $q' \leq q$, there is a subrow $s \in \mathrm{S}(q)$ with $s \approx q'$ and $\mathrm{P}(s) = \{()\}$. Moreover, $\mathrm{inc}(s) = \subset$ if $q' < q$, otherwise $\mathrm{inc}(s) = \mathrm{eq}$. It can now easily be verified using Definition 6 that Q is normal.

Now suppose R and Q are arbitrary tables and all $\mathrm{aug}(R_i)$ are normal. Let also $Q_i = \mathrm{compr}(\mathrm{aug}(R_i))$. By Lemma 1, all Q_i are normal. Let $q \in Q$, $s \in \mathrm{S}(q)$, $X \in \mathrm{E}(q)$ and $Y \in \mathrm{E}_X(s)$ be arbitrary, and let $\mathrm{P}(q) = \{(q_1, \ldots, q_k)\}$ and $\mathrm{P}(s) = \{(s_1, \ldots, s_k)\}$. It follows that there are extensions $X_i \in \mathrm{E}(q_i)$ such that $X = \{q\} \cup X_1 \cup \cdots \cup X_k$, and there are extensions $Y_i \in \mathrm{E}_{X_i}(s_i)$ such that $Y = \{s\} \cup Y_1 \cup \cdots \cup Y_k$. To introduce auxiliary terminology, for any $x \in Q$ and $y \in \mathrm{S}(q)$, we write $\bar{x}$ to denote $\mathrm{orig}_R(x)$ and $\bar{y}$ for $\mathrm{orig}_R(y)$. Furthermore, for any $x \in Q_i$ and $y \in \mathrm{S}(x)$, we write $\bar{x}$ to denote $\mathrm{orig}_{R_i}(x)$ and $\bar{y}$ to denote $\mathrm{orig}_{R_i}(y)$.

To prove the first part of condition 1 of normality for Q, recall that property 2 from the definition of Q states that $s \approx q'$ for some $q' \in Q$ with $q' \leq q$. Hence $s \leq q$. By normality of Q_i it holds that $Y_i \leq X_i$. This proves $Y \leq X$.

For the second part of condition 1 of normality for Q, first suppose $Y < X$ while $\mathrm{inc}(s) = \mathrm{eq}$. The reason for the latter is that $\mathrm{inc}(s_i) = \mathrm{eq}$ for all s_i. For all $1 \leq i \leq k$, this entails that $Y_i \approx X_i$ due to the normality of Q_i. So $Y < X$ must be caused by $s < q$, but this would entail $\mathrm{inc}(s) = \subset$, which is a contradiction.

On the other hand, suppose $Y \approx X$ while $\mathrm{inc}(s) = \subset$. If there is some $d \in \mathrm{D}(q) \setminus \mathrm{D}(s)$, then for $Y \approx X$ to hold also $d \in \mathrm{D}(Y_j)$ for some $1 \leq j \leq k$. But then d has been illegally removed at $\bar{s}$, which contradicts R being augmentable. More precisely, $d \in \mathrm{D}(\bar{q}) \setminus \mathrm{D}(\bar{s})$ holds, and due to $Y \in \mathrm{E}(s)$ and Lemma 6 there is some $Z \in \mathrm{E}(\bar{s})$ such that $d \in \mathrm{D}(Z)$.

So $q \approx s$ and for some $1 \leq j \leq k$ there is a $d \in \mathrm{D}(X_j) \setminus \mathrm{D}(Y_j)$. For $Y \approx X$ to hold, $d \in \mathrm{D}(s)$ or there is some $1 \leq h \leq k$ such that $d \in \mathrm{D}(Y_h)$.

In case $d \in \mathrm{D}(s)$, d has been illegally introduced at $\bar{s}$, which contradicts R being augmentable. More precisely, $d \in \mathrm{D}(\bar{s})$, $(\bar{s_1}, \ldots, \bar{s_k}) \in \mathrm{P}(\bar{s})$ (by Lemma 5) and $d \notin \mathrm{D}(\bar{s_j})$ hold while $d \in \mathrm{D}^*(R_j)$ due to $d \in \mathrm{D}(X_j)$, which amounts to $d \in \mathrm{D}(Z)$ for some $Z \in \mathrm{E}(\bar{q_j})$.

In case $d \in \mathrm{D}(Y_h)$ for some $1 \leq h \leq k$, we have $h \neq j$ since $d \notin \mathrm{D}(Y_j)$. Then also $d \in \mathrm{D}(X_h)$ because $Y_h \leq X_h$ by normality of Q_h. Then, by Lemma 2, there are $H \in \mathrm{E}(\bar{q_h})$ and $J \in \mathrm{E}(\bar{q_j})$ such that $d \in \mathrm{D}(H)$ and $d \in \mathrm{D}(J)$. Because R is augmentable, also $d \in \mathrm{D}(\bar{q})$ and thus $d \in \mathrm{D}(q)$. As $q \approx s$, also $d \in \mathrm{D}(s)$. But then d has been illegally introduced at $\bar{s}$, as we have have already seen. This concludes the proof of the second part of condition 1 of normality for Q.

Now we show condition 2 of normality for Q. The reason for $s \in \mathrm{S}(q)$ is that there is some $q' \in Q$ with $\mathrm{P}(q') = \{(q'_1, \ldots, q'_k)\}$ and, for all $1 \leq i \leq k$, there is some $s_i \in \mathrm{S}(q_i)$ such that $s_i \approx q'_i$. Let $r = \mathrm{orig}_R(q')$ and $r_i = \mathrm{orig}_{R_i}(q'_i)$ and $Z \in \mathrm{E}(s)$ be arbitrary. Then there are extensions $Z_i \in \mathrm{E}(s_i)$ such that $Z = \{s\} \cup Z_1 \cup \cdots \cup Z_k$. By the induction hypothesis, for any $1 \leq i \leq k$ there is a row $q''_i \in Q_i$ and an extension $X''_i \in \mathrm{E}(q''_i)$ such that $q''_i \approx s_i \approx q'_i$ and $X''_i \approx Z_i$. As q'_i and q''_i have the same data and augmentable tables have at most one row having a certain data, also $(q''_1, \ldots, q''_k) \in \mathrm{res}(r_1) \times \cdots \times \mathrm{res}(r_k)$ holds. By construction of Q, this entails that there is some $q'' \in Q$ with $\mathrm{P}(q'') = \{(q''_1, \ldots, q''_k)\}$ and $q'' \approx r = q' \approx s$. We can construct $X'' = \{q''\} \cup X''_1 \cup \cdots \cup X''_k$ and observe that $X'' \in \mathrm{E}(q'')$. From $Z_i \approx X''_i$, for all $1 \leq i \leq k$, and $s \approx q''$ it follows that $Z \approx X''$, which proves condition 2 of normality for Q.

Finally we show condition 3 of normality for Q. Suppose there are $q, q' \in Q$ with $\mathrm{P}(q) = \{(q_1, \ldots, q_k)\}$ and $\mathrm{P}(q') = \{(q'_1, \ldots, q'_k)\}$, and there are $X \in \mathrm{E}(q)$ and $X' \in \mathrm{E}(q')$ such that $X' \leq X$.

Now for all $1 \leq i \leq k$ let $X_i \in \mathrm{E}(q_i)$ and $X'_i \in \mathrm{E}(q'_i)$. We first show that $X'_i \leq X_i$. Suppose to the contrary that for some $1 \leq j \leq k$ there is a $d \in \mathrm{D}(X'_j) \setminus \mathrm{D}(X_j)$. For $X' \leq X$ to hold, $d \in \mathrm{D}(q)$ or there is some $1 \leq h \leq k$ such that $d \in \mathrm{D}(X_h)$.

In case $d \in \mathrm{D}(q)$, d has been illegally introduced at $\bar{q}$, which contradicts R being augmentable. More precisely, $d \in \mathrm{D}(\bar{q})$, $(\bar{q_1}, \ldots, \bar{q_k}) \in \mathrm{P}(\bar{q})$ and $d \notin \mathrm{D}(\bar{q_j})$ hold while $d \in \mathrm{D}^*(R_j)$ due to $d \in \mathrm{D}(X'_j)$, which amounts to $d \in \mathrm{D}(Z)$ for some $Z \in \mathrm{E}(\mathrm{orig}_{R_j}(q'_j))$.

In case $d \in \mathrm{D}(X_h)$ for some $1 \leq h \leq k$, we have $h \neq j$ since $d \notin \mathrm{D}(X_j)$. Then there are $H \in \mathrm{E}(\bar{q_h})$ and $J \in \mathrm{E}(\bar{q_j})$ such that $d \in \mathrm{D}(H)$ and $d \in \mathrm{D}(J)$. Because R is augmentable, also

$d \in \mathrm{D}(\bar{q})$ and thus $d \in \mathrm{D}(q)$. But then d has been illegally introduced at $\bar{q}$, as we have have already seen.

Hence $X'_i \leq X_i$ for any $1 \leq i \leq k$. By the induction hypothesis, this entails that there are $s_i \in \mathrm{S}(q_i)$ and $Y_i \in \mathrm{E}_{X_i}(s_i)$ with $s_i \approx q'_i$ and $Y_i \approx X'_i$. By construction of Q, then there is a subrow $s \in \mathrm{S}(q)$ with $s \approx q'$ and $\mathrm{P}(s) = \{(s_1, \ldots, s_k)\}$. We can now construct $Y = \{s\} \cup Y_1 \cup \ldots Y_k$ and observe that $Y \in \mathrm{E}_X(s)$ and $Y \approx X'$, which concludes the proof. $\square$