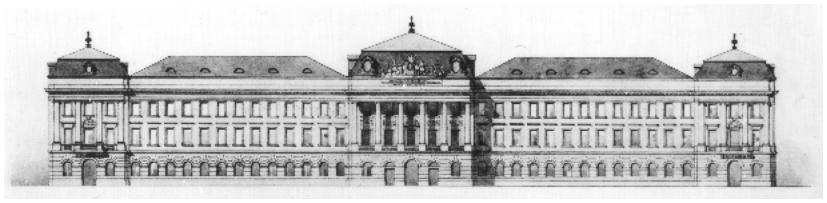
dbai

Technical Report



Institut für Informationssysteme

Abteilung Datenbanken und Artificial Intelligence

Treewidth-Preserving Modeling in ASP

DBAI-TR-2016-97

Manuel Bichler Bernhard Bliem
Marius Moldovan Michael Morak
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
2016

TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

DBAI Technical Report
DBAI Technical Report DBAI-TR-2016-97, 2016

Treewidth-Preserving Modeling in ASP

Manuel Bichler [1] Bernhard Bliem [1]
Marius Moldovan [1] Michael Morak [1]
Stefan Woltran [1]

Abstract. For ground ASP programs where an appropriate graph representation has bounded treewidth, algorithms that exploit this bound on the treewidth theoretically only require linear time for checking existence of an answer set. However, in practice such algorithms are hardly competitive against state-of-the-art CDCL-based solvers, which do not explicitly rely on bounded treewidth. In this work we investigate the hypothesis that CDCL-based solvers leverage small treewidth implicitly. We identify modeling constructs in non-ground ASP that significantly blow up the treewidth in the grounding and we give guidelines for modeling problems in non-ground ASP such that grounding preserves small treewidth of the input. We also experimentally show that a non-ground rule decomposition technique can decrease the treewidth of the grounding substantially. Finally, we identify a class of non-ground ASP programs that preserves bounded treewidth in the sense that the treewidth of the grounding only depends on the treewidth and the degree of the input graph instead of its size.

[1]Institute for Information Systems 184/2, Technische Universität Wien, Favoritenstrasse 9-11, 1040 Vienna, Austria. E-mail: [bichler,bliem,moldovan,morak,woltran]@dbai.tuwien.ac.at

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

Contents

1 Introduction

Answer Set Programming (ASP) [10, 20, 25, 28] is a well-established logic programming paradigm based on the stable model semantics of logic programs. Its main strength is a fully declarative language and the fact that, generally, each answer set of a given logic program directly describes a valid answer to the original question. Moreover, ASP solvers—see, e.g., [2, 3, 15, 24]—have made huge strides in efficiency and are increasingly used for industrial applications like planning or scheduling. A logic program usually consists of a set of logical implications (called rules) and a set of facts. Deciding the consistency problem, that is, whether a given disjunctive logic program has an answer set, is $\text{NExpTime}^{\text{NP}}$-complete in the combined complexity, where both the rules and facts are treated as input, and Σ^2_{P}-complete in the data complexity, where the set of rules is fixed; cf. [14].

In practice, when problems are modeled using the ASP logic programming language, the usual goal is to write a fixed program (i.e., set of rules) that solves the general problem. The concrete input is then supplied as a set of (ground) facts. The answer set solver takes the fixed program, plus the ground facts, and computes the answer sets, that is, the solutions to the original problem, as described earlier. Logic programming in general, and ASP in particular, have gained popularity because of this intuitive, declarative way to solve problems and the straightforward syntax. The following example illustrates this:

Example 1. *For an input graph specified as* `vertex(.)` *and* `edge(.,.)` *facts, the following program solves the three-colorability problem:*

```
1 { col(V,red), col(V,blue), col(V,green) } 1 :- vertex(V).
:- edge(X,Y), col(X,C), col(Y,C).
```

The first rule guesses exactly one of three colors for each vertex, while the second rule makes sure that no neighboring vertices have the same color. □

Evaluating an ASP program is usually a two-step approach (all current state-of-the-art solvers work in this way): First, the input program is grounded, that is, the variables in the program are replaced (in the worst case) by all possible, valid combinations of constants from the input domain. Secondly, a solver evaluates the (now) ground program, and computes the answer sets. Note that the grounding step is exponential in general. The difficulty of solving ASP programs is then often determined by certain parameters of this ground input program. Straightforward parameters include the size of the grounding, or the ratio between rules and atoms in the program. However, there are also more evolved, structural parameters. One such parameter is the treewidth of the ground program. The ground program is converted to a graph representation, and, intuitively, the treewidth then measures how close this graph is to a tree. It turns out that favorable theoretical runtime results can be established if the treewidth is small. Several algorithms have been proposed [26, 31] and implemented [29], but, despite linear runtime in theory, they have generally turned out to be hardly competitive when compared to state-of-the-art answer set solvers like *clasp* for consistency checking[1]. Furthermore, these algorithms are not designed to be general-purpose solvers, since they can only perform well on low-treewidth input programs. However, other solvers, like *clasp*, perform well consistently on such low-treewidth inputs, but also perform well on other instances. This begs the question whether existing CDCL-based solvers are inherently sensitive to low treewidth ASP programs.

It is the aim of this report to investigate the impact of the treewidth of ground programs on the runtime of existing state-of-the-art CDCL-based solvers. Our working hypothesis is that the answer to this question is affirmative. However, ASP problem encodings are usually written in non-ground form, in order to make use of the full, intuitive, and expressive ASP language. The actual input instance, consisting of ground facts, is then combined with the encoding, grounded, and then finally passed to the solver. Therefore, another interesting question arises: Assuming that the ground facts of the input instance, represented as a graph, already have low treewidth, which ASP language constructs may be used when writing the encoding, such that the low treewidth of the input instance is preserved in the grounded program? Furthermore, does the recently proposed approach of rule decomposition on the non-ground program [5, 7, 30] have an influence on the treewidth of the ground program obtained from it? Having these questions in mind, our main contributions can be stated as follows:

1. We perform an in-depth experimental investigation of different encodings for the same graph problems in order to identify those encodings that best preserve the low treewidth of the input graph (represented as ground facts) in the ground program obtained from combining the input graph with the respective encoding. We show that constructs like transitivity lead to a dramatic increase in the treewidth, while constructs like reachability do not, and identify the reasons for this, formulating several overarching guidelines for encoding problems in ASP in such a way that the treewidth of the input instance is preserved.

2. We investigate, via an experimental evaluation, the impact of a certain rewriting technique (viz. rule decomposition of non-ground programs, cf. [6]) on the treewidth of the corresponding ground program, that is, we compare the original encoding with the decomposed encoding for the same input graph. We show that this impact can be substantial and often reduces the treewidth by a large amount.

3. We isolate a class of ASP programs with restricted syntax that guarantees that for any encoding in the class and for any input graph of bounded treewidth and bounded degree, the treewidth of the corresponding ground program remains bounded as well. To be more precise, we showed that the treewidth and the degree of the grounded program only depends on the clique-width (a parameter more general than treewidth) and degree of the input graph. That is, for any non-ground ASP programs in this

[1]This, however, is another story for answer set *counting*, where these approaches prove to be highly competitive in some settings [17].

class, the treewidth of the grounded program does not depend on the size of the input graph.

The remainder of the paper is structured as follows. In the next section, we will introduce relevant background information regarding the ASP language and tree decompositions. Then, we report on our experimental evaluation regarding the impact of different encodings and rewriting strategies on the treewidth of the grounding. Section 4 provides our theoretical results on classes of programs which show robust treewidth with respect to the grounding step. In the final section, we conclude the paper with some observations and ideas for future work.

2 Preliminaries

General Definitions. We define two pairwise disjoint countably infinite sets of symbols: a set $\mathbf{C}$ of *constants* and a set $\mathbf{V}$ of *variables*. Different constants represent different values (*unique name assumption*). By $\mathbf{X}$ we denote sequences (or, with slight notational abuse, sets) of variables $X_1, \ldots, X_k$ with $k \geqslant 0$. For brevity, let $[n] = \{1, \ldots, n\}$, for any integer $n \geqslant 1$.

A (*relational*) *schema* $\mathcal{S}$ is a (finite) set of *relational symbols* (or *predicates*). We write p/n for the fact that p is an n-ary predicate. A *term* is a constant or variable. An *atomic formula* $\underline{a}$ over $\mathcal{S}$ (called $\mathcal{S}$-*atom*) has the form $p(\mathbf{t})$, where $p \in \mathcal{S}$ and $\mathbf{t}$ is a sequence of terms. An $\mathcal{S}$-*literal* is either an $\mathcal{S}$-atom (i.e. a positive literal), or an $\mathcal{S}$-atom preceded by the negation symbol "$\neg$" (i.e. a negative literal). For a literal ℓ, we write $dom(\ell)$ for the set of its terms, and $var(\ell)$ for its variables. This notation naturally extends to sets of literals. For brevity, we will treat conjunctions of literals as sets. For a domain $C \subseteq \mathbf{C}$, a (*total* or *two-valued*) $\mathcal{S}$-*interpretation* I is a set of $\mathcal{S}$-atoms containing only constants from C such that, for every $\mathcal{S}$-atom $p(\mathbf{a}) \in I$, $p(\mathbf{a})$ is true, and otherwise false. When obvious from the context, we will omit the schema-prefix.

A *substitution* from a set of literals L to a set of literals L' is a mapping $s : \mathbf{C} \cup \mathbf{V} \to \mathbf{C} \cup \mathbf{V}$ that is defined on $dom(L)$, is the identity on $\mathbf{C}$, and $p(t_1, \ldots, t_n) \in L$ (resp. $\neg p(t_1, \ldots, t_n) \in L$) implies $p(s(t_1), \ldots, s(t_n)) \in L'$ (resp., $\neg p(s(t_1), \ldots, s(t_n)) \in L'$).

Answer Set Programming (ASP). A *logic programming rule* is a universally quantified reverse first-order implication of the form

$$\mathcal{H}(\mathbf{X}, \mathbf{Y}) \leftarrow \mathcal{B}^{+}(\mathbf{X}, \mathbf{Y}, \mathbf{Z}, \mathbf{W}) \wedge \mathcal{B}^{-}(\mathbf{X}, \mathbf{Z}),$$

where $\mathcal{H}$ (the *head*), resp. $\mathcal{B}^{+}$ (the *positive body*), is a disjunction, resp. conjunction, of atoms, and $\mathcal{B}^{-}$ (the *negative body*) is a conjunction of negative literals, each over terms from $\mathbf{C} \cup \mathbf{V}$. For a rule π, let $H(\pi)$, $B^{+}(\pi)$, and $B^{-}(\pi)$ denote the set of atoms occurring in the head, the positive, and the negative body, respectively. Let $B(\pi) = B^{+}(\pi) \cup B^{-}(\pi)$. A rule π where $H(\pi) = \emptyset$ is called a *constraint*. Substitutions naturally extend to rules. We focus on *safe* rules where every variable in the rule occurs in the positive body. A rule is called *ground* if all its terms are constants. The grounding of a rule π w.r.t. a domain $C \subseteq \mathbf{C}$ is the set of rules $\mathit{ground}_C(\pi) = \{s(\pi) \mid s \text{ is a substitution, mapping } \mathit{var}(\pi) \text{ to elements from } C\}$.

A *logic program* Π is a finite set of logic programming rules. The schema of a program Π, denoted $\mathit{sch}(\Pi)$, is the set of predicates appearing in Π. We call a predicate *intensional* in Π if it occurs in the head of a rule of Π, and we call it *extensional* otherwise. An atom is *intensional* or *extensional* whenever the involved predicate is. The *active domain* of Π, denoted $\mathit{adom}(\Pi)$, with $\mathit{adom}(\Pi) \subseteq \mathbf{C}$, is the set of constants appearing in Π. A program Π is ground if all its rules are ground. The *grounding of a program* Π is the ground program $\mathit{ground}(\Pi) = \bigcup_{\pi\in\Pi} \mathit{ground}_{\mathit{adom}(\Pi)}(\pi)$. The *(Gelfond-Lifschitz) reduct* of a ground program Π w.r.t. an interpretation I is the ground program $\Pi^I = \{H(\pi) \leftarrow B^+(\pi) \mid \pi \in \Pi, B^-(\pi) \cap I = \emptyset\}$.

A $\mathit{sch}(\Pi)$-interpretation I is a *(classical) model* of a ground program Π, denoted $I \models \Pi$ if, for every ground rule $\pi \in \Pi$, it holds that $I \cap B^+(\pi) = \emptyset$ or $I \cap (H(\pi) \cup B^-(\pi)) \neq \emptyset$, that is, I satisfies π. I is a *stable model* (or *answer set*) of Π, denoted $I \models_s \Pi$ if, in addition, there is no $J \subset I$ such that $J \models \Pi^I$, that is, I is a subset-minimal model of the reduct Π^I. The set of answer sets of Π, denoted $\mathit{AS}(\Pi)$, are defined as $\mathit{AS}(\Pi) = \{I \mid I \text{ is a } \mathit{sch}(\Pi)\text{-interpretation, and } I \models_s \Pi\}$. For a non-ground program Π, we define $\mathit{AS}(\Pi) = \mathit{AS}(\mathit{ground}(\Pi))$. When referring to the fact that a logic program is intended to be interpreted under the answer set semantics, we often refer to it as an *ASP program*.

ASP Language Extensions. The ASP-Core-2 language specification for the standardized ASP language [11] defines several additional constructs to those discussed above, which we will briefly introduce here. Please see [11] for precise formal definitions of syntax and semantics. We will call rules that do not contain any such extensions *simple*.

Arithmetic expressions are atoms of the form

$$X \preccurlyeq \varphi(\mathbf{Y}),$$

where $\preccurlyeq \in \{<, \leqslant, =, \neq, \geqslant, >\}$ is a built-in relation, and φ is a mathematical expression built from constant numbers, variables from $\mathbf{Y}$, and the arithmetic connectives "$+$," "$-$," "$\times$," and "$\div$," with the obvious intuitive meaning. A (non-ground) rule π may contain such expressions in its body. Such a rule is safe, if (i) every variable in the arithmetic expression is safe, or (ii) $\preccurlyeq$ is the equality relation "$=$" and all variables in φ are safe, in which case also X is safe. The notion of grounding extends naturally to arithmetic expressions. Note, however, that ground arithmetic expressions can be directly evaluated. Thus, after grounding, rules with unsatisfied arithmetic expressions in the body are removed, and otherwise, the arithmetic expression is removed. Without loss of generality, we therefore assume that ground rules do not contain arithmetic expressions.

A *weak constraint* of the form

$$\pi[k, \mathbf{t}]$$

is a constraint π annotated with a term k representing a weight and a sequence of terms $\mathbf{t}$ occurring in π. The notion of grounding extends naturally to arithmetic expressions. Each answer set M is annotated by a total weight $w(M)$, which is the sum over all constants k for each tuple of constants $\mathbf{t}$, where the body of a corresponding weak constraint is satisfied in M. Note that k and $\mathbf{t}$ consist of constants after grounding.

An *aggregate atom* is an atom of the form

$$t \preccurlyeq \#agg\{\mathbf{t} : \varphi(\mathbf{X})\},$$

where t is a term; $\preccurlyeq \in \{<, \leqslant, =, \neq, \geqslant, >\}$ is a built-in relation; *agg* is one of *sum*, *count*, *max*, and *min*; $\mathbf{t} = \langle t_1, \ldots, t_n \rangle$ is a sequence of terms; and $\varphi(\mathbf{X})$ is a set of literals and arithmetic expressions, called the *aggregate body*. Aggregates may appear in rule bodies, with the following semantic meaning: Given an interpretation I, for each valid substitution s such that $s(\varphi(\mathbf{X})) \subseteq I$, take the tuple of constants $s(\mathbf{t})$. Let us denote this set with T. Now, execute the aggregate function on T as follows: for $\#count$, calculate $|T|$; for $\#sum$, calculate $\Sigma_{\mathbf{t} \in T} t_1$, where t_1 is the first term in $\mathbf{t}$; for $\#max$ and $\#min$, take the maximum and minimum term appearing in the first position of each tuple in T, respectively. Finally, an aggregate expression is true if the relation $\preccurlyeq$ between term t and the result of the aggregate function is fulfilled. Grounding of aggregates is done by rewriting aggregates into simple ground rules; see [22] for details.

A *choice rule* π is a rule

$$l\{\mathcal{H}(\mathbf{X}, \mathbf{Y})\}u \leftarrow \mathcal{B}^+(\mathbf{X}, \mathbf{Y}, \mathbf{Z}, \mathbf{W}) \wedge \mathcal{B}^-(\mathbf{X}, \mathbf{Z}),$$

where $\mathcal{B}^+$ and $\mathcal{B}^-$ are as before, l and u are terms (the upper and lower bound), and $\mathcal{H}$ is a set of *choice elements* of the form $a(\mathbf{V}) : \varphi(\mathbf{V}, \mathbf{V}')$, where φ is a set of literals over the variables $\mathbf{V}$ and $\mathbf{V}'$, and $\mathbf{V} \subseteq (\mathbf{X} \cup \mathbf{Y})$. Such a choice rule is a form of syntactic sugar, and stands for the rule

$$a(\mathbf{V}) \vee \overline{a}(\mathbf{V}) \leftarrow \mathcal{B}^+(\mathbf{X}, \mathbf{Y}, \mathbf{Z}, \mathbf{W}), \mathcal{B}^-(\mathbf{X}, \mathbf{Z}), \varphi(\mathbf{V}, \mathbf{V}'),$$

where $\overline{a}$ is an fresh relation, along with the two constraints of the form

$$\perp \leftarrow \mathcal{B}^+(\mathbf{X}, \mathbf{Y}, \mathbf{Z}, \mathbf{W}), \mathcal{B}^-(\mathbf{X}, \mathbf{Z}), k \preccurlyeq \#count\{\mathbf{V} : a(\mathbf{V}), \varphi(\mathbf{V}, \mathbf{V}')\},$$

where k and $\preccurlyeq$ are l and $>$, or u and $<$, respectively.

An *optimization statement* is a statement of the form

$$\#opt\{k, \mathbf{t} : \varphi(\mathbf{X})\},$$

which is another form to write a weak constraint of the form

$$\perp \leftarrow \varphi(\mathbf{X})[k', \mathbf{t}],$$

where $k' = k$ if $opt = maximize$, or $k' = -k$ if $opt = minimize$.

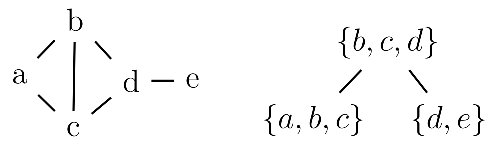


Figure 1: A graph with treewidth 2 and an (optimal) tree decomposition for it

Graph Representations. A ground ASP program can be represented as a graph structure. Two graph representations are often used in the literature. Firstly, the *primal graph* of a ground program Π is a graph $G = (V, E)$, where the set of vertices V contains a vertex v_a for every atom a appearing in Π. There is an edge $(v_a, v_b) \in E$, if atoms a and b appear together in some rule π in Π.

Secondly, the *incidence graph* of a ground program Π is a graph $G = (V, E)$, where the set of vertices V contains a vertex v_a (called *atom vertex*) for every atom a appearing in Π, and a vertex v_π (called *rule vertex*) for every rule π appearing in Π. There is an edge $(v_a, v_\pi) \in E$, if atom a appears in rule π.

Tree Decompositions. A *tree decomposition* of a graph $G = (V, E)$ is a pair $\mathcal{T} = (T, \chi)$, where T is a rooted tree and χ is a labeling function over nodes t of T, with $\chi(t) \subseteq V$ called the *bag of* t, such that the following holds: (i) for each $v \in V$, there exists a node t in T, such that $v \in \chi(t)$; (ii) for each $\{v, w\} \in E$, there exists a node t in T, such that $\{v, w\} \subseteq \chi(t)$; and (iii) for all nodes r, s, and t in T, such that s lies on the path from r to t, we have $\chi(r) \cap \chi(t) \subseteq \chi(s)$. The *width* of a tree decomposition is defined as the cardinality of its largest bag minus one. The *treewidth* of a graph G, denoted by $tw(G)$, is the minimum width over all tree decompositions of G.

Figure 1 shows a graph together with a tree decomposition of it that has width 2. This decomposition is optimal because the graph contains a cycle and thus its treewidth is at least 2.

To decide whether a graph has treewidth at most k is NP-complete [4]. For an arbitrary but fixed k however, this problem can be solved (and a tree decomposition constructed) in linear time [9].

The *primal (resp. incidence) treewidth* of a ground ASP program Π is the treewidth of its primal (resp. incidence) graph.

Clique-width. Clique-width is a more general parameter than treewidth in the sense that graphs of bounded treewidth also have bounded clique-width, but there are classes of graphs that have bounded clique-width and unbounded treewidth.

A *k-graph*, for $k > 0$, is a graph whose vertices are labeled by integers from $\{1, \ldots, k\} =: [k]$. We call the k-graph consisting of exactly one vertex v (say, labeled by $i \in [k]$) an *initial k-graph* and denote it by $i(v)$. Graphs can be constructed from initial k-graphs by means of repeated application of the following three operations:

- *Disjoint union* (denoted by $\oplus$);
- *Relabeling*: changing all labels i to j (denoted by $\rho_{i \to j}$);
- *Edge insertion*: connecting all vertices labeled by i with all vertices labeled by j via an edge (denoted by $\eta_{i,j}$); $i \neq j$; already existing edges are not doubled.

A construction of a k-graph G using the above operations can be represented by an algebraic term composed of $i(v)$, $\oplus$, $\rho_{i \to j}$, and $\eta_{i,j}$ ($i, j \in [k]$, and v a vertex). Such a term is then called a *cwd-expression defining* G. A *k-expression* is a cwd-expression in which at most k different labels occur.

As an example consider the complete bipartite graph $K_{n,n}$ with bipartition $A = \{a_1, \dots, a_n\}$ and $B = \{b_1, \dots, b_n\}$. A cwd-expression of $K_{n,n}$ using at most two labels is given by the following steps: (1) introduce all vertices in A using label 1, (2) introduce all vertices in B using label 2, (3) take the disjoint union of all these vertices, and (4) add all edges between vertices with label 1 and vertices with label 2, i.e., such a cwd-expression is given by $\eta_{1,2}(1(a_1) \oplus \cdots \oplus 1(a_n) \oplus 2(b_1) \oplus \cdots \oplus 2(b_n))$. As a second example consider the complete graph K_n on n vertices. A cwd-expression for K_n using at most two labels can be obtained by the following iterative process: Given a cwd-expression σ_{n-1} for K_{n-1}, where every vertex is labeled with label 1, one takes the disjoint union of σ_{n-1} and $2(v)$ (where v is the vertex only contained in K_n but not in K_{n-1}), adds all edges between vertices with label 1 and vertices with label 2, and then relabels label 2 to label 1. Formally, the cwd-expression σ_n for K_n is given by $(\rho_{2 \to 1}(\eta_{1,2}(\sigma_{n-1} \oplus 2(v_2)))$.

The *clique-width* of a graph G is the smallest integer k such that G can be defined by a k-expression. Our discussion above thus witnesses that complete (bipartite) graphs have clique-width 2. Furthermore, co-graphs also have clique-width 2 (co-graphs are exactly given by the graphs which do not contain an induced P_4 ,i.e., whenever there is a path (a, b, c, d) in the graph then $\{a, c\}$, $\{a, d\}$ or $\{b, d\}$ is also an edge of the graph) and trees have clique-width 3.

Monadic Second-Order Transductions. A *monadic second-order (MSO) transduction* [13] defines a function that maps "input graphs" to "output graphs", and this function is defined in terms of MSO logic.[2] To this end, an MSO transduction consists of MSO formulas $\chi, \delta_1, \dots, \delta_c, \theta_{1,1}, \theta_{1,2}, \dots, \theta_{c,c}$, where c is a positive integer, each δ_i has one free variable x, and each $\theta_{i,j}$ has two free variables x, y.

The purpose of the formula χ is to characterize the class of graphs for which the transduction is defined, namely those graphs G such that $G \vDash \chi$ holds.[3] We call a graph G an *input graph* if $G \vDash \chi$.

[2]In fact, MSO transductions apply not only to graphs but to relational structures in general. However, we only describe the special case of graphs for simplicity; it can be generalized in a straightforward way.

[3]To be more precise, χ is an MSO formula over the signature consisting of just the binary edge predicate, and $G \vDash \chi$ denotes that χ is satisfied by the relational structure that has $V(G)$ as its domain and that interprets the edge predicate as $E(G)$.

```
1 { p(X,Y) : edge(X,Y) } 1 :- vertex(X).
1 { p(X,Y) : edge(X,Y) } 1 :- vertex(Y).
pTrans(X,Y) :- p(X,Y).
pTrans(X,Z) :- pTrans(X,Y), pTrans(Y,Z).
:- vertex(X), vertex(Y), not pTrans(X,Y).
#show p/2.
```

Listing 1: ASP Hamiltonian Cycle encoding that uses transitivity.

We now explain the intended meaning of the δ_i formulas. Let G be an input graph. We denote the corresponding output graph (i.e., the result of the transformation on G) by H. The δ_i formulas define which vertices exist in H. For each $v \in V(G)$, there can be up to c copies of v in H. In fact, there is a copy v_i in H iff $(G, v) \vDash \delta_i$ (i.e., δ_i evaluates to true under G when the free variable x is interpreted as v).

Finally, the edges in H are specified using the $\theta_{i,j}$ formulas. For each pair of vertices $(v, w) \in V(G)^2$, there is an edge $(v_i, w_j) \in E(H)$ iff $(G, v, w) \vDash \theta_{i,j}$ in addition to $(G, v) \vDash \delta_i$ and $(G, w) \vDash \delta_j$.

It makes a difference whether an MSO transduction transforms an input graph into an output graph directly, or whether it transforms the incidence structure of an input graph into the incidence structure of an output graph.[4] MSO transductions that process graphs directly are useful because they allow us to conclude that the transformations they describe preserve bounded clique-width (i.e., the output graph's clique-width is bounded whenever the input graph's clique-width is bounded). On the other hand, MSO transductions that deal with the incidence structures of the input and output graphs guarantee that the corresponding transformations preserve bounded treewidth [13, Corollary 1.53].

3 Empirical Investigation of Modeling Techniques

In this section, we first show and describe different encodings for the graph problems Hamiltonian Cycle, Secure Set and Minimum Weighted Dominating Set. Next, we describe our experiments on finding the treewidths of the grounded encodings using traffic networks as input. We then show a correlation between the treewidth of the grounded program and a lower clingo running time for solving these problems. Finally, we investigate the impact of splitting non-ground rules up into smaller rules and find that this not only reduces the grounding size and primal treewidth but also the incidence treewidth of the ground program.

[4]An incidence structure of a graph is the structure whose domain consists not only of vertices but also of edges, and whose signature consists of the incidence relation.

```
1 1 { p(X,Y) : edge(X,Y) } 1 :- vertex(X).
2 1 { p(X,Y) : edge(X,Y) } 1 :- vertex(Y).
3 pTrans(X,Y) :- p(X,Y).
4 pTrans(X,Z) :- pTrans(X,Y), p(Y,Z).
5
6 :- vertex(X), vertex(Y), not pTrans(X,Y).
7 #show p/2.
```

Listing 2: Refined ASP Hamiltonian Cycle encoding that uses transitivity.

3.1 Encodings

3.1.1 Hamiltonian Cycle

We define the Hamiltonian Cycle problem as follows: Given a simple undirected graph $G = (V, E)$, with $V = \{1,2,...,n\}$, and a weight function w: $E \to \mathbb{R}^+$, we seek a cyclic permutation $\sigma = (1, \sigma(1), \sigma^2(1), ..., \sigma^{n-1}(1))$ of V, $\sigma^i(1)$ denoting the i$^{\text{th}}$ successor of vertex 1 (with $\sigma^0(1) = \sigma^n(1) = 1$), such that

$$w(\sigma) = \sum_{i=0}^{n-1} w(\{\sigma^i(1), \sigma^{i+1}(1)\})$$

is minimal. To this end, we will present several encodings, with transitivity, reachability and saturation approaches, respectively.

Listing 1 uses transitivity to find directed Hamiltonian cycles in a graph. In Lines 1 and 2 we guess for each vertex exactly one outgoing and one ingoing edge to be part of the path which turns into one cycle in actual solutions. Next, in Lines 3 and 4 we define the transitivity relation for edges that are selected to be part of the path. Finally, we check for connectivity, namely that we have exactly one cycle. As we stated in Line 3 that each edge in the guessed path is part of the transitivity relation, we know that a pair (a, b) of vertices is not in the transitivity relation if the guessed path does not connect a with b. We throw away solution candidates that contain such a pair in Line 5. Line 6 only filters what is being printed as the solution, namely the edges that are part of the cycle.

In Listing 2, we show a refined version of our ASP encoding for finding directed Hamiltonian cycles in a graph, using transitivity. The difference to the encoding in Listing 1 lies in Line 4. Given that the predicate `pTrans/2` can be inferred only in Lines 3 and 4, we now define the transitivity relation using predicate `p/2` instead of `pTrans/2` in the body of the rule in Line 4 once.

The encoding in Listing 3 replaces Lines 3 to 5 from the encodings in Listings 1 and 2 and ensures the connectedness of the solutions by means of reachability instead of transitivity. In Lines 3 and 4 of Listing 3 we first pick a starting vertex and then state that it is reachable from itself. In Line 5 we state that, if a vertex is reached and one of its outgoing edges is selected for the path, also the other endpoint is reached. Finally, in Line 6 we throw away all solution candidates in which there exist vertices that are not reached.

```
 1  1 { p(X,Y) : edge(X,Y) } 1 :- vertex(X).
 2  1 { p(X,Y) : edge(X,Y) } 1 :- vertex(Y).
 3  start(X) :- X = #min{ Y : vertex(Y) }.
 4  reach(X) :- start(X).
 5  reach(Y) :- reach(X), p(X,Y).
 6  :- vertex(X), not reach(X).
 7  #show p/2.
```

Listing 3: ASP Hamiltonian Cycle encoding that uses reachability.

```
 1  1 { p(X,Y) : edge(X,Y) } 1 :- vertex(X).
 2  1 { p(X,Y) : edge(X,Y) } 1 :- vertex(Y).
 3  s1(X) | s2(X) :- vertex(X).
 4  s1(X) :- saturate, vertex(X).
 5  s2(X) :- saturate, vertex(X).
 6  :- not saturate.
 7  saturate :- p(X,Y), s1(X), s2(Y).
 8  numVertices(N) :- N = #count{ X : vertex(X) }.
 9  saturate :- N #count{ X : s1(X) }, numVertices(N).
10  saturate :- N #count{ X : s2(X) }, numVertices(N).
11  #show p/2.
```

Listing 4: ASP Hamiltonian Cycle encoding that uses saturation.

Next, we present an encoding, in Listing 4, that finds again directed Hamiltonian cycles, now ensuring connectedness by means of saturation. The key idea here is that the guessed edges are connected iff for each partition of the vertices into two nonempty sets, there is a guessed edge between the two sets. Guessing edges in Lines 1 and 2 works like in the first three encodings. Next, in Line 3 we (tentatively) partition the vertex set into two sets, in Line 7 we derive `saturate` if there exists a guessed edge that connects the two partitions, and in Line 6 we say that `saturate` must be derived. Lines 4 and 5 then cause us to "saturate" the predicates that identify the partitions, hence the answer sets themselves do not encode partitions. However, the predicates that identify partitions play a crucial role in the subset-minimality check, which is done implicitly due to the ASP semantics: In order for a model to be an answer set, we go through all proper subsets and make sure none of them satisfies the reduct. In this particular case, if we have a model M containing some guessed set of edges that is not connected, then the reduct will have a model N that does not contain `saturate` and encodes a partition of the vertices such that no guessed edge connects the two parts. Because of the latter, N satisfies Line 7, and it trivially satisfies Line 6 because this rule is not present in the reduct. Hence N witnesses that M is no answer set because $N \subset M$. This means that every answer set encodes a set of guessed edges such that for every partition of the vertices into two parts there is a crossing edge. Lines 8 to 10 are merely used to make sure that we only check partitions where both parts are nonempty.

```
1  1 { p(X,Y) : edge(X,Y) } 1 :- vertex(X).
2  1 { p(X,Y) : edge(X,Y) } 1 :- vertex(Y).
3  s1(X) | s2(X) :- vertex(X).
4  s1(X) :- saturate, vertex(X).
5  s2(X) :- saturate, vertex(X).
6  :- not saturate.
7  saturate :- p(X,Y), s1(X), s2(Y).
8  all1 :- all1_upto(X), sup(X).
9  all1_upto(X) :- inf(X), s1(X).
10 all1_upto(Y) :- all1_upto(X), succ(X,Y), s1(Y).
11 all2 :- all2_upto(X), sup(X).
12 all2_upto(X) :- inf(X), s2(X).
13 all2_upto(Y) :- all2_upto(X), succ(X,Y), s2(Y).
14 saturate :- all1.
15 saturate :- all2.
16 notSucc(X,Z) :- vertex(X), vertex(Y), vertex(Z), X < Y, Y < Z.
17 succ(X,Y) :- vertex(X), vertex(Y), X < Y, not notSucc(X,Y).
18 notInf(X) :- succ(_,X).
19 inf(X) :- vertex(X), not notInf(X).
20 notSup(X) :- succ(X,_).
21 sup(X) :- vertex(X), not notSup(X).
22 #show p/2.
```

Listing 5: ASP Hamiltonian Cycle saturation encoding with loops.

The encoding in Listing 5 also uses saturation to establish connectedness. We use this encoding to check how avoiding aggregates can influence the treewidth. The approach is the same as in Listing 4 but replaces Lines 8 to 10, which use aggregates, by Lines 8 to 21 in which we check if for the current partition all vertices are contained in one set by means of creating a succession for the vertices and checking for each of them if they belong to `s1/1` and `s2/1`, respectively.

3.1.2 Secure Set

The Secure Set problem asks for a set of vertices S in a graph G such that for each subset $X \subseteq S$, $|N[X] \cap S| \geq |N[X] \setminus S|$ holds, where $N[X]$ is the closed neighborhood of X in G, i.e., the set X together with all vertices adjacent to some vertex in X. For this problem we used the encodings in Listing 6 to Listing 9, which have been introduced and thoroughly explained in [1]. The first three encodings use the concept of *loops*, the difference between the ones in Listing 6 and Listing 7 lying in the formula we use for determining if a set is secure or not, using either a `sum` or a `count` aggregate. The encoding in Listing 8 uses border vertices to define secure sets, besides the concept of *loops*. In Listing 9 we have an encoding that uses a different approach.

In order to avoid using loops we guess a partition of the attack set into active and inactive

```
edge(U,V) :- edge(V,U).
1 #count { V : inS(V) : vertex(V) }.
outS(V) :- vertex(V), not inS(V).
attackSet(V) :- inS(U), edge(U,V), outS(V).
inX(V) | outX(V) :- inS(V).
defendSet(V) :- inX(V).
defendSet(V) :- inX(U), edge(U,V), inS(V).
lt(U,V) :- vertex(U), vertex(V), U < V.
nsucc(U,W) :- lt(U,V), lt(V,W).
succ(U,V) :- lt(U,V), not nsucc(U,V).
ninf(V) :- lt(U,V).
nsup(U) :- lt(U,V).
inf(V) :- vertex(V), not ninf(V).
sup(V) :- vertex(V), not nsup(V).
okto(V,U) :- vertex(U), vertex(V), inf(U), outX(U).
okto(V,U) :- vertex(U), vertex(V), inf(U), outS(U).
okto(V,U) :- vertex(U), vertex(V), inf(U), inX(U), not edge(U,V).
okto(W,V) :- vertex(U),vertex(V),vertex(W),okto(W,U),succ(U,V),outX(V).
okto(W,V) :- vertex(U),vertex(V),vertex(W),okto(W,U),succ(U,V),outS(V).
okto(W,V) :- vertex(U),vertex(V),vertex(W),okto(W,U),succ(U,V), inX(V),
             not edge(V,W).
ok(V) :- sup(U), okto(V,U).
inactiveAttacker(V) :- inS(U), edge(U,V), ok(V), outS(V).
defended :- #sum { 1,V,pos : vertex(V), defendSet(V);
                   1,V,pos : vertex(V), inactiveAttacker(V);
                  -1,V,neg : vertex(V), attackSet(V) } >= 0.
inX(V)  :- defended, inS(V).
outX(V) :- defended, inS(V).
:- not defended.
:- inS(V), T = #count { U : edge(U,V) },
      #count { W : vertex(V), vertex(W), edge(V,W), inS(W) } < T / 2.
#show inS/1.
```

Listing 6: ASP Secure Set loop encoding.

attackers instead of a subset X of S. Further, for performance reasons all encodings for Secure Set we use in this technical report have an extra last constraint, compared to those in [1], that throws away all answer set candidates that contain vertices in S with less neighbors in S than outside of it.

3.1.3 Minimum Weighted Dominating Set

Finally, we will present two encodings for Minimum Weighted Dominating set. The task is to compute all vertex-weight-minimal dominating sets of an undirected graph $G = (V, E)$. A subset X of V is a dominating set of G if for each $v \in V$ the vertex is part of X or v is

```
edge(U,V) :- edge(V,U).
1 #count { V : inS(V) : vertex(V) }.
outS(V) :- vertex(V), not inS(V).
attackSet(V) :- inS(U), edge(U,V), outS(V).
inX(V) | outX(V) :- inS(V).
defendSet(V) :- inX(V).
defendSet(V) :- inX(U), edge(U,V), inS(V).
lt(U,V) :- vertex(U), vertex(V), U < V.
nsucc(U,W) :- lt(U,V), lt(V,W).
succ(U,V) :- lt(U,V), not nsucc(U,V).
ninf(V) :- lt(U,V).
nsup(U) :- lt(U,V).
inf(V) :- vertex(V), not ninf(V).
sup(V) :- vertex(V), not nsup(V).
okto(V,U) :- vertex(U), vertex(V), inf(U), outX(U).
okto(V,U) :- vertex(U), vertex(V), inf(U), outS(U).
okto(V,U) :- vertex(U), vertex(V), inf(U), inX(U), not edge(U,V).
okto(W,V) :- vertex(U),vertex(V),vertex(W),okupto(W,U),succ(U,V),outX(V).
okto(W,V) :- vertex(U),vertex(V),vertex(W),okupto(W,U),succ(U,V),outS(V).
okto(W,V) :- vertex(U),vertex(V),vertex(W),okupto(W,U),succ(U,V), inX(V),
             not edge(V,W).
ok(V) :- sup(U), okto(V,U).
inactiveAttacker(V) :- inS(U), edge(U,V), ok(V), outS(V).
size(N) :- N = #count { V : vertex(V) }.
defended :- #count { V,pos : vertex(V), defendSet(V);
                     V,pos : vertex(V), inactiveAttacker(V);
                     V,neg : vertex(V), not attackSet(V) } >= N, size(N).
inX(V)  :- defended, inS(V).
outX(V) :- defended, inS(V).
:- not defended.
:- inS(V), T = #count { U : edge(U,V) },
      #count { W : vertex(V), vertex(W), edge(V,W), inS(W) } < T / 2.
#show inS/1.
```

Listing 7: ASP Secure Set loop encoding with `count` aggregate.

adjacent to at least one $u \in X$. With Listing 10 and Listing 11 we will illustrate the effects of using an extra cost function with an aggregate. In both encodings we first guess which vertices should be in the dominating set, we state that a vertex is dominated if its neighbor is in the set and we set the constraint that all vertices must be dominated, in Lines 1 to 3. In Listing 10 we directly minimize over the weights of the vertices which are in the dominating set, in Line 4, while in Listing 11 we first calculate the sum of these weights in Line 4 and only then minimize over that value, in Line 5.

```
edge(U,V) :- edge(V,U).
1 #count { V : inS(V) : vertex(V) }.
outS(V) :- vertex(V), not inS(V).
attackSet(V) :- inS(U), edge(U,V), outS(V).
border(U) :- inS(U), outS(V), edge(U,V).
inX(V) | outX(V) :- border(V).
outX(V) :- inS(V), not border(V).
defendSet(V) :- inX(V).
defendSet(V) :- inX(U), edge(U,V), inS(V).
lt(U,V) :- vertex(U), vertex(V), U < V.
nsucc(U,W) :- lt(U,V), lt(V,W).
succ(U,V) :- lt(U,V), not nsucc(U,V).
ninf(V) :- lt(U,V).
nsup(U) :- lt(U,V).
inf(V) :- vertex(V), not ninf(V).
sup(V) :- vertex(V), not nsup(V).
okto(V,U) :- vertex(U), vertex(V), inf(U), outX(U).
okto(V,U) :- vertex(U), vertex(V), inf(U), outS(U).
okto(V,U) :- vertex(U), vertex(V), inf(U), inX(U), not edge(U,V).
okto(W,V) :- vertex(U),vertex(V),vertex(W),okto(W,U),succ(U,V),outX(V).
okto(W,V) :- vertex(U),vertex(V),vertex(W),okto(W,U),succ(U,V),outS(V).
okto(W,V) :- vertex(U),vertex(V),vertex(W),okto(W,U),succ(U,V), inX(V),
             not edge(V,W).
ok(V) :- sup(U), okto(V,U).
inactiveAttacker(V) :- inS(U), edge(U,V), ok(V), outS(V).
defended :- #sum { 1,V,pos : vertex(V), defendSet(V);
                   1,V,pos : vertex(V), inactiveAttacker(V);
                  -1,V,neg : vertex(V), attackSet(V) } >= 0.
inX(V)  :- defended, inS(V).
outX(V) :- defended, inS(V).
:- not defended.
:- inS(V), T = #count { U : edge(U,V) },
     #count { W : vertex(V), vertex(W), edge(V,W), inS(W) } < T / 2.
#show inS/1.
```

Listing 8: ASP Secure Set border loop encoding.

3.2 Experiments

In order to find out the treewidths of the groundings of the encodings when applied to traffic network instances, we first fed each combination of an encoding and an instance to *gringo 4.5.0* [19,21] and then ran *DynASP 2.0* [18] five times with options *-d -c1*, *-d -c4* and *-d -c5* each to determine the primal, incidence and incidence-weighted-primal treewidth, respectively and always took into consideration the smallest among the five resulting treewidths provided by *DynASP*. The incidence-weighted-primal treewidth is the treewidth of a modi-

```
edge(U,V) :- edge(V,U).
1 #count { V : inS(V) : vertex(V) }.
outS(V) :- vertex(V), not inS(V).
border(U,V):- inS(U), outS(V), edge(U,V).
attackSet(V) :- border(U,V).
activeAttacker(V) | inactiveAttacker(V) :- attackSet(V).
inX(U) : border(U,V) :- activeAttacker(V), outS(V).
defended :- inactiveAttacker(U), inX(V), edge(U,V).
defendSet(V) :- inX(V).
defendSet(V) :- inX(U), edge(U,V), inS(V).
defended :- #sum { 1,V,pos : vertex(V), defendSet(V);
                    1,V,pos : vertex(V), inactiveAttacker(V);
                   -1,V,neg : vertex(V), attackSet(V) } >= 0.
inX(V) :- defended, inS(V).
activeAttacker(V) :- defended, attackSet(V).
inactiveAttacker(V) :- defended, attackSet(V).
:- not defended.
:- inS(V), T = #count { U : edge(U,V) },
     #count { W : vertex(V), vertex(W), edge(V,W), inS(W) } < T / 2.
#show inS/1.
```

Listing 9: Alternative ASP Secure Set encoding.

```
{ in(X) : vertex(X) }.
dominated(Y) :- in(X), edge(X,Y).
:- vertex(X), not in(X), not dominated(X).
#minimize { W,X: in(X), weight(X,W) }.
```

Listing 10: ASP Minimum Weighted Dominating Set encoding.

fication of the incidence graph, such that the literals of all choice heads and weighted bodies (for details please see [32]) each form a clique. Weighted bodies appear in the grounding when using aggregate functions. The used instances represent rail traffic networks, such as metro, tram and train networks, of cities, metropolitan areas or states, having treewidths of 2, 3 and 4 and at most 70 vertices. The graphs were extracted from mass transit data feedsthat are publicly available using gtfs2graphs [16] and split by transportation type, such

```
{ in(X) : vertex(X) }.
dominated(Y) :- in(X), edge(X,Y).
:- vertex(X), not in(X), not dominated(X).
cost(C) :- C = #sum { W,X: in(X), weight(X,W) }.
#minimize { C : cost(C) }.
```

Listing 11: ASP Minimum Weighted Dominating Set encoding with aggregate.

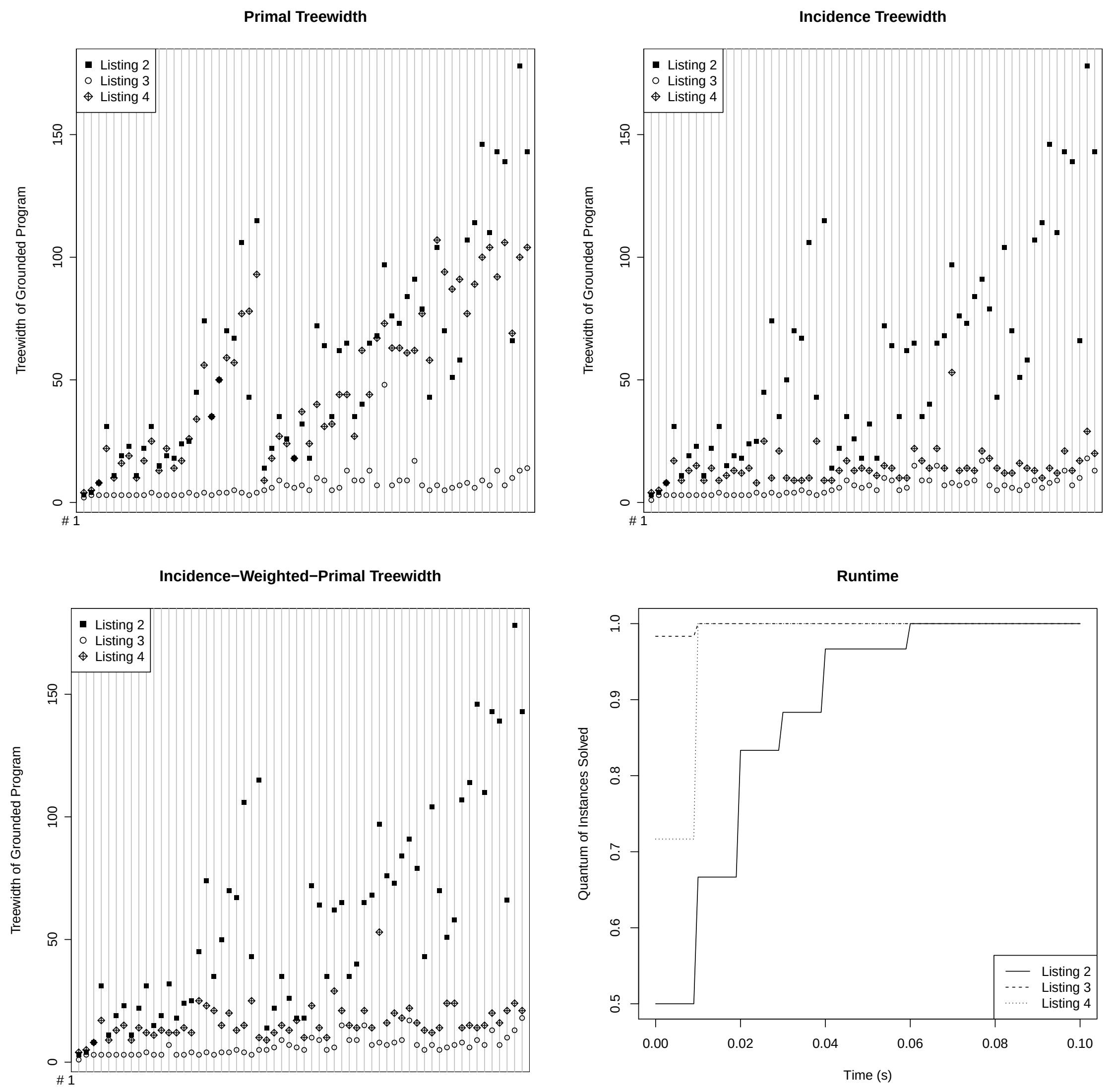


Figure 2: Treewidths of groundings of Hamiltonian Cycle encodings and quantum of instances per time, solved by the latter with *clingo*.

as tram, metro, train and combinations thereof. For Minimum Weighted dominating set we used the same instances to create three test sets. For the first we inserted for each vertex a weight of 1, for the second we inserted random weights between 1 and 10 and for the third random weights between 1 and 100. Further, to determine the running time for the encodings on our set of traffic networks, we used *clingo 4.5.4* [23] on a machine with an AMD Opteron 6308@3.5 GHz processor operated with Debian 8 (jessie, kernel 3.16.0-4-amd64).

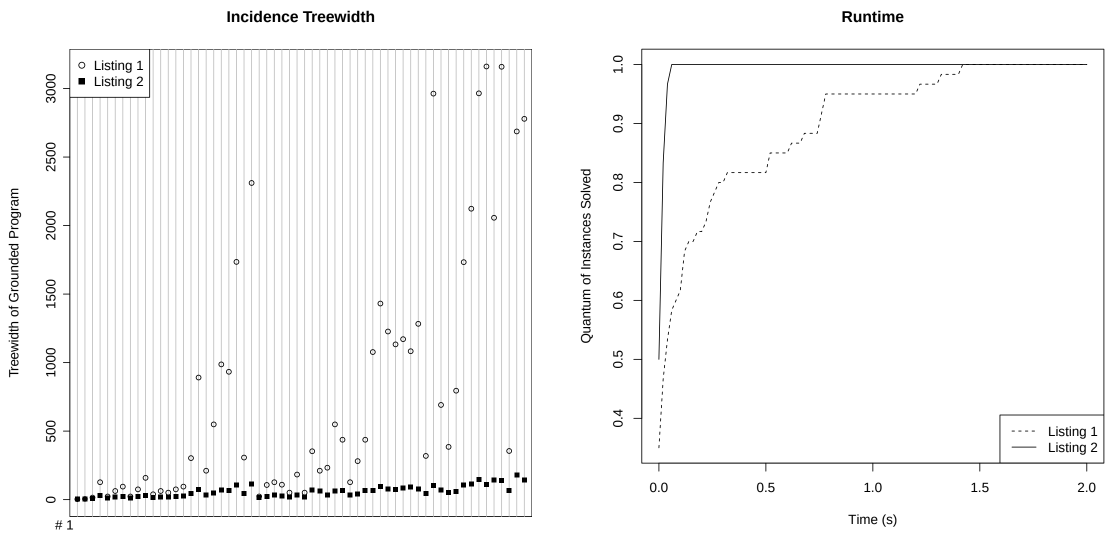


Figure 3: Treewidths of groundings of Hamiltonian Cycle encodings using transitivity and quantum of instances per time, solved by the latter with *clingo*.

3.2.1 Hamiltonian Cycle

In Figure 2 we present the primal, incidence and incidence-weighted-primal treewidths of the groundings of four Hamiltonian Cycle encodings, one using transitivity, one using reachability and one using saturation, all applied to each of the traffic networks. In each of the first three graphs, each vertical line stands for a traffic network instance. Further, we show the relation between the consumed time for solving and the quantum of instances solved. What leaps to the eye first is the fact that for all different types of graphs of the groundings, the encoding in Listing 2, that uses transitivity, is the one producing the highest treewidths. At the same time, the latter encoding is the slowest among the four compared in Figure 2. This can be explained by the fact that the number of `pTrans/2` instantiations can be quadratic in the number of vertices, leading to complex grounding graphs. This is why we dissuade from using such constructions that lead to the formation of a transitive closure. In addition, we recommend using encodings ensuring connectivity using the reachability approach. For the latter, materialized in the encoding in Listing 3, the structure of the grounding graphs is simpler due to the fact that instead of `pTrans/2` we have `reach/1` in Line 5 and the number of the latter are is linear in the number of vertices in the graph. This is why the grounding treewidths all remain linear or even constant in the treewidths of the original graphs. The incidence and incidence-weighted-primal treewidths for the groundings of the encoding based on saturation from Listing 4 also remain linear in dependence of the original treewidth of the instances, yet the primal treewidth also correlates with the size of the instance, in numbers of vertices, which can be explained by the presence of the rule in Line 7. Nevertheless, in matters of running time the encoding using saturation was only slightly slower than the one using reachability.

Looking at the charts in Figure 3 we can see that among the encodings using transitivity,

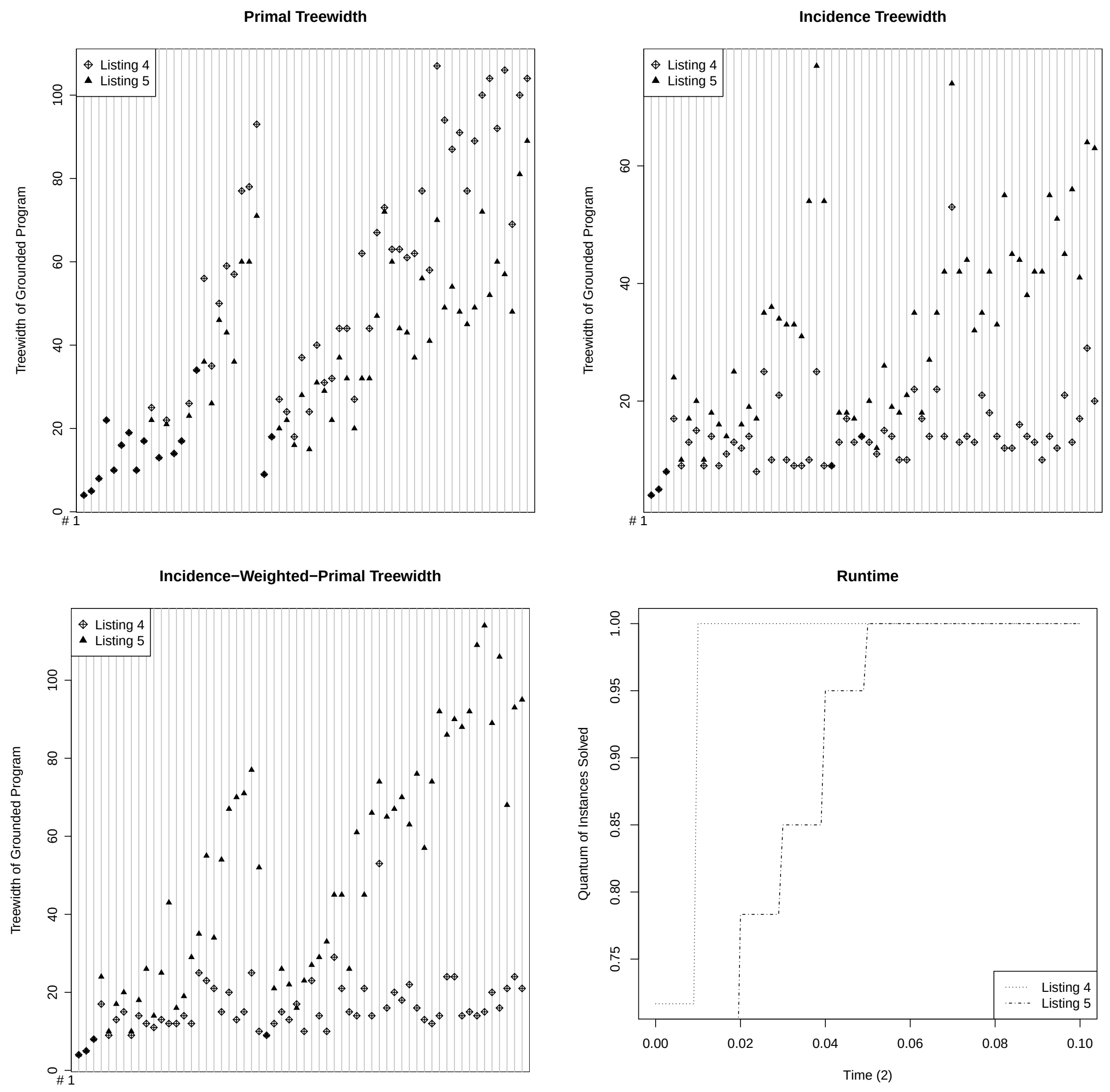


Figure 4: Treewidths of groundings of Hamiltonian Cycle encodings using saturation and quantum of instances per time, solved by the latter with *clingo*.

the one in Listing 2 performs better than the one in Listing 1 both in terms of treewidth and running time, although the difference lies solely in replacing one occurrence of `pTrans/2` with `p/2` in Line 4. This can be explained by the fact that the number of `pTrans/2` instantiations is quadratic in the number of vertices when the graph is connected while the number of `p/2` instantiations corresponds to the number of edges which is quadratic in the number of vertices only in case of a clique which is rather uncommon in traffic networks.

In Figure 4 we show how the two encodings for Hamiltonian Cycle using saturation behave w.r.t. running time and treewidth. The primal treewidth is correlated with the size of the input graph for both encodings rises dramatically with the former. However, incidence and incidence-weighted-primal treewidth grow with the size of the input graph for the encoding

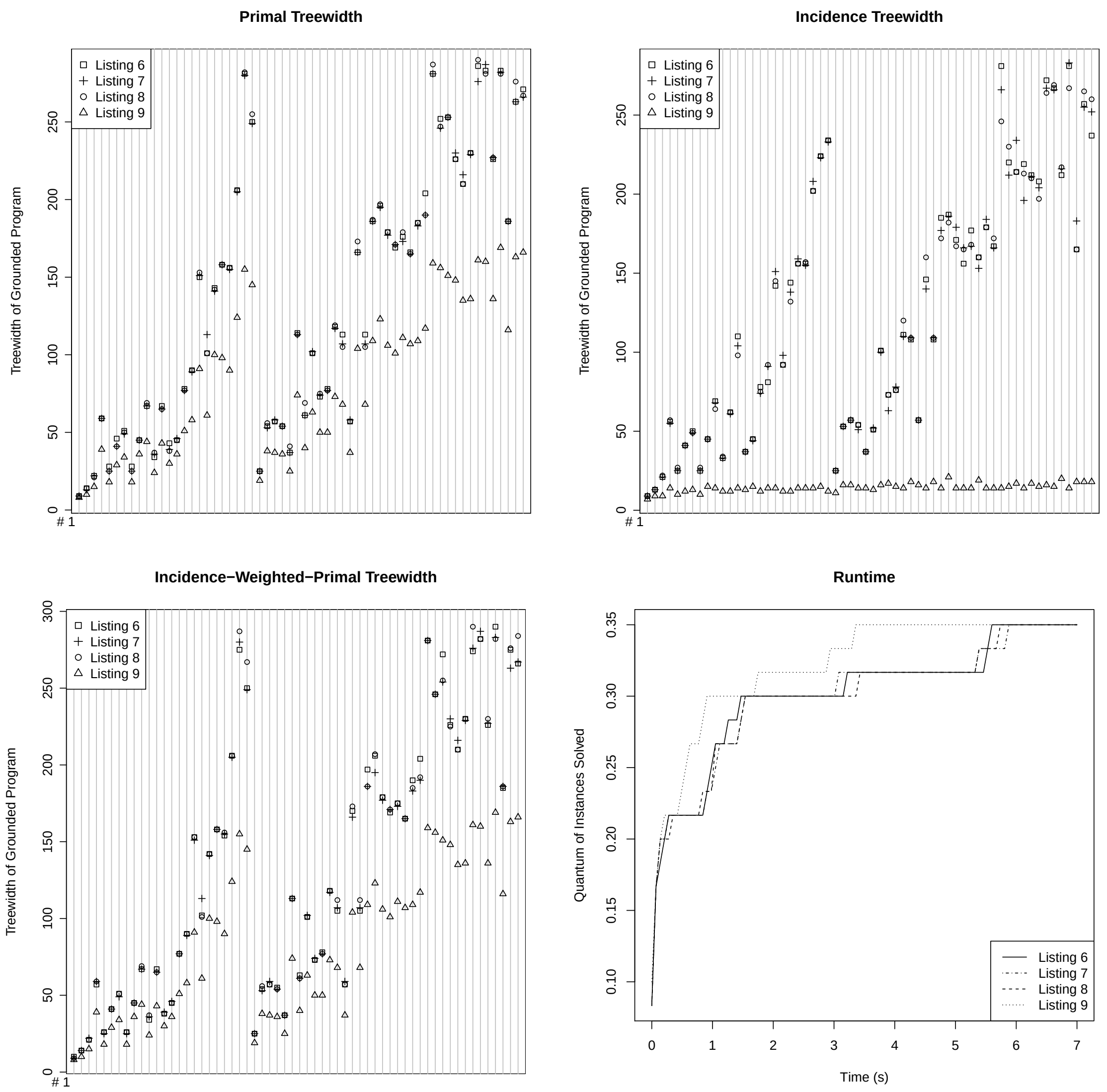


Figure 5: Treewidths of groundings of Secure Set encodings and quantum of instances per time, solved by the latter with *clingo*.

in Listing 5, while for the encoding in Listing 4 they remain linear in the treewidth of the original graph. The reason for the former behavior lies in the loops in Lines 8 to 13 from Listing 5, which result in complex structures of the grounding graphs. These are similar to those induced by a transitive closure, even if not as complex. This is why we advise against the use of loops. Furthermore, also here we can see a correlation between incidence and incidence-weighted-primal treewidths of the groundings and a lower running time for solving the problems with *clingo*.

3.2.2 Secure Set

Another example where loops have a negative impact on running time and treewidth is one of the Secure Set encodings, as we can see in Figure 5. The encodings that make use of a loop, namely those in Listings 6, 7 and 8, all have similar treewidths that depend on the sizes of the input instances, and are slower than the encoding in Listing 9, which avoids loops. The primal and incidence-weighted-primal treewidths for the groundings of the latter are lower than those for the groundings of the encodings with loops, yet they increase with the size of the input graph. Similarly to the behavior of the treewidths of grounded Hamiltonian Cycle encodings using saturation, also here, it is the incidence treewidth that remains linear or even constant in the treewidth of the original program. Unlike in the latter case, now the incidence-weighted-primal treewidth increases due to the restriction on the *count* aggregate.

3.2.3 Minimum Weighted Dominating Set

Let us look at the treewidths of Minimum Weighted Dominating Set groundings and the runtime performance for the two encodings on instances which contain vertices with weights of up to 10, in Figure 6. Primal and incidence-weighted-primal treewidths are only slightly lower for the encoding in Listing 10 than for the one in Listing 11. Incidence treewidth stays linear or even constant in the treewidth of the input instances for the latter, while for the former it increases with the number of vertices of the input instance. At the same time, the encoding in Listing 11 is much faster than the one in Listing 10, which leads to more complex incidence graphs as minimizing over a predicate that contains the sum of weights of vertices in the dominating set instead of directly minimizing on the latter weights. Please note that the treewidths are the same regardless of the values the weights of the vertices.

3.3 Impact of Rule Decomposition

As discussed earlier, when dealing with ASP programs, evaluation is usually a 2-step process. First, the program is grounded, that is, variables are replaced by all possible, valid combinations of symbols. Then, the solver is called on the ground program. The grounding process is exponential in the size of the rules of the program in general, but a recent rule decomposition tool named *lpopt*, presented in [6, 7] and originally based on an idea from [30], is able to reduce this exponentiality to the treewidth of the rules. Roughly, this rule decomposition tool works by representing non-ground rules as graphs, constructing a tree decomposition of these graphs, and then splitting the rules up into multiple smaller rules based on this decomposition. Interestingly, it seems that, apart from decreasing the grounding size, in certain instances also the treewidth of the ground program is reduced. Since the rules become smaller, the tool clearly reduces the primal graph treewidth of the ground program obtained from the original program after applying the rule decomposition. However, as the following experimental tests show, interestingly, also the incidence treewidth is reduced.

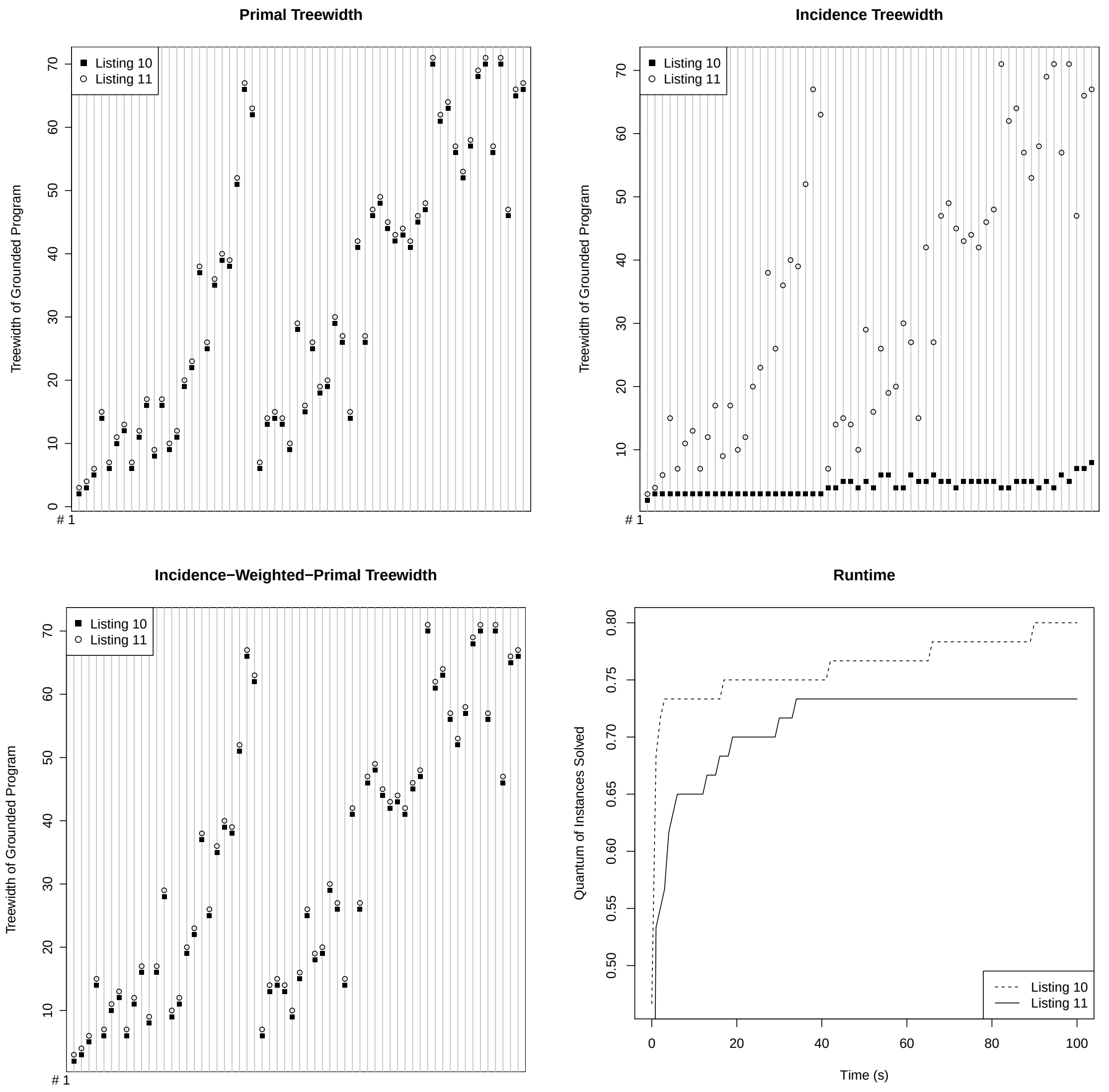


Figure 6: Treewidths of groundings of Minimum Weighted Dominating Set encodings and quantum of instances per time, solved by the latter with *clingo*.

3.3.1 Experiment Setup

For evaluating *lpopt*'s impact on ground incidence treewidth, we have considered three sources of problem instances:

1. The 200 2-QBF instances evaluated in [7], encoded in the new paradigm presented there, once before and once after being preprocessed by *DepQBF* [27]. For each instance, this paradigm yields a program with a long rule which makes it an archetype application for *lpopt*.

2. 20 different voting problem encodings taken from the *Democratix* project[5] [12] together with 315 instances from the *PrefLib* project[6].

3. Instances from the Fifth Answer Set Programming Competition 2014[7], providing two encodings for most of the 25 problems: one from 2013 and one from 2014, each with 20 instances. The impact of *lpopt* on grounding sizes and running times of these instances has already been investigated in [6].

Note that evaluation requires a logic program to be decomposable by *lpopt* at all, otherwise "common" groundings would be the same as *lpopt*-assisted ones. Due to not being *lpopt*-decomposable, the following instances could not be evaluated: "classic" 2-QBF encodings from [7], secure set encodings from [1] and Steiner tree problems taken from the *D-FLAT* project[8] [8].

For each instance, evaluation has been performed in the following manner:

1. (optional) Decompose the instance with *lpopt* in order to split up rules

2. Ground the instance using *gringo 4.5.4* [23]

3. Run *DynASP 2.0* [18] with options *-d -c4* on the grounding to determine its incidence treewidth

If, for any given instance, either the plain program or the *lpopt*-decomposed version ran into a timeout or a memory overflow in any of these steps, then this instance was considered incomparable.

For the 2-QBF instance set, evaluation was performed on an Intel Xeon E5345 2.33 GHz machine with *gringo* limits set to 10 minutes and 6 GB and *DynASP* limits set to 15 minutes and 48 GB. The voting instances were evaluated on the same machine with limits set to 10 minutes and 6 GB for both *gringo* and *DynASP*. The ASP competition instance sets were evaluated on an AMD Opteron 6308 3.5 GHz machine with the only limit being 5 minutes for *DynASP*.

3.3.2 Results

In the 2-QBF case, due to the very big rules, only few instances could have been grounded in the non-*lpopt*-assisted case within the time and memory limits. Additionally, *DepQBF*-preprocessed instances that have already been solved by the preprocessing were considered uninteresting and thus omitted from graphical representation. Therefore, as can be seen in Figure 7, out of the 400 instances (200 plain, 200 *DepQBF*-preprocessed), only 8 have been considered comparable (four plain, the same four *DepQBF*-preprocessed). Nevertheless, the

[5]`http://democratix.dbai.tuwien.ac.at/`

[6]`http://www.preflib.org/data/packs/soc.zip`

[7]`https://www.mat.unical.it/aspcomp2014/`

[8]`http://dbai.tuwien.ac.at/research/project/dflat/system/`

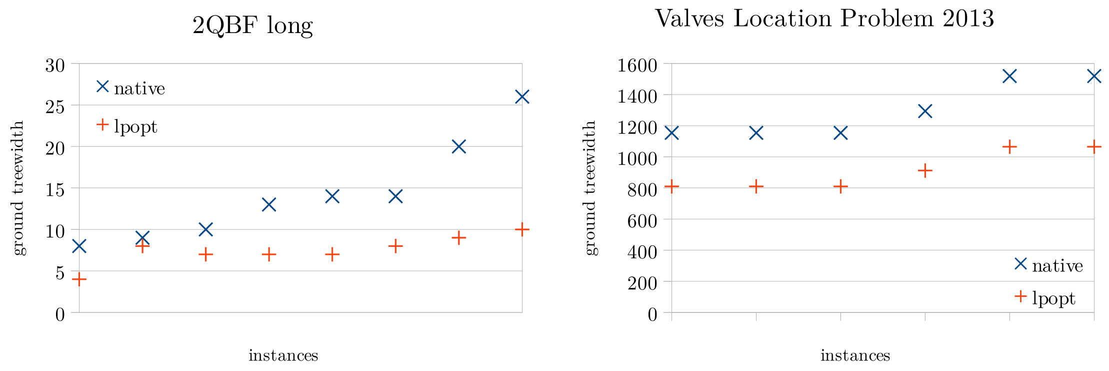


Figure 7: Incidence treewidths of groundings of comparable instances, with *lpopt* and without it ("native"). Instances are sorted by native treewidth. The left plot shows the results for the 8 comparable 2-QBF instances in the paradigm used in [7]. The right plot shows the results for the 6 comparable instances of the valves location problem in ASP competition's 2013 encoding.

2-QBF evaluation and Figure 7 suggest that the method of rule decomposition strongly reduces the ground incidence treewidths of these programs.

For voting, ground incidence treewidth was found to be de facto independent of whether *lpopt* was applied: 11 out of 20 problems have been found comparable (that is, contain comparable instances), each of which only shows in exceptional cases an *lpopt*-induced minimal change in ground incidence treewidth (both up and down).

The 187 comparable instances of the ASP competition show *lpopt* impact only in three problems:

The ground treewidth of the valves location problem of 2013, depicted right in Figure 7, benefits from the decomposition, although grounding size and solving time evaluation in [5] shows no significant impact. Here, *lpopt* is able to split up one rule without having to introduce auxiliary domain rules.

The ground treewidth of the weighted sequence problem of 2014 quite heavily suffers from the decomposition, though evaluation in [5] only shows a slight increase in grounding size (about 5 %). Here, *lpopt* introduced an auxiliary domain rule using an intensional predicate, which causes the observed deterioration. For a discussion on auxiliary domain rules and why they are needed, we refer to [6]. At this point, it suffices to mention that non-ground auxiliary domain rules potentially introduce cycles, which do not manifest themselves in the program's ground incidence graph if they only use extensional predicates (as then the body is grounded away).

The crossing minimization problem in the 2013 encoding also has its ground incidence treewidth increased when being treated by *lpopt*, though evaluation in [5] depicted a grounding size reduction by about 50% (again, with no significant runtime changes). Also here, an auxiliary domain rule is introduced using a guessed (and therefore intensional) predicate.

3.3.3 Discussion

The question of which predicates to select for auxiliary domain rules has already been posed in [5,6]. As of the current version of *lpopt*, this selection is being computed by a simple greedy algorithm that does not take into account whether predicates are extensional or intensional. The paper [6] raised awareness that the number of ground rules, and therefore the grounding size, depends on this selection. The work [5] suggested that the selection algorithm also respect whether candidate predicates are intensional or extensional, and avoid intensional ones. This would best be accomplished by incorporating rule decomposition into a grounder, since grounders like *gringo* are already aware of which predicates are intensional and which are extensional.

As the results show, rule decomposition is able to reduce the ground incidence treewidth of answer set programs. In a few cases, the treewidth deteriorated by rule decomposition, which may be avoided by including decomposition into the grounding process and preventing intensional predicates from being selected for auxiliary domain rules. If done so, our experiments show no reason for rule decomposition to increase ground incidence treewidth.

4 Theoretical Investigation of Modeling Techniques

Our observations from Section 3 suggest that a problem can often be modeled in multiple ways that lead to quite different relationships between the "input treewidth" (i.e., the treewidth of the input graph) and the "output treewidth" (i.e., the primal or incidence treewidth of the grounding). In particular, some encodings are "well-behaved" in the sense that they preserve bounded treewidth. By this we mean that the output treewidth can be bounded from above by a function that depends only on the input treewidth. Other encodings, however, may destroy bounded treewidth in the sense that the output treewidth cannot be bounded by the input treewidth alone but depends on the input size. We would like to identify which modeling techniques are responsible for this. In this section we first give some examples of such "ill-behaved" modeling techniques. We then characterize a class of non-ground ASP programs that excludes those techniques, and we prove that encodings from this class preserve bounded treewidth.

4.1 Modeling Techniques That Destroy Bounded Treewidth

In this section, we give a small selection of modeling techniques that have to be avoided if bounded treewidth is to be preserved. Note, however, that this is not a complete list and that other techniques may also destroy bounded treewidth. In the following, we assume that the input graph is directed and given by the unary vertex predicate `v` and the binary edge predicate `e`.

ASP grounders do not blindly instantiate the variables with all possible constants. Instead, they try to minimize the number of produced rules by suppressing ground rules whose body can never be satisfied by any model of the program. This property is also crucial for the treewidth of the resulting program, since blindly instantiating each variable with every constant almost always leads to unbounded output treewidth.

For our investigation, we therefore assume that a ground rule is suppressed from the grounding if it contains an extensional literal that is not a consequence of the input facts. This is the only assumption that we make on grounders in this work. For instance, this excludes that a grounding contains a rule `p(x,y) :- e(x,y)` if (x, y) is no edge in the input graph.

As a first example, consider the rule `p(X,Y) :- v(X), v(Y).` This destroys bounded treewidth even for graphs without any edges: The output graph (i.e., the primal or incidence graph of the grounding) contains the complete graph K_n as a minor, where n is the number of input vertices. The problem here is that this rule allows for a too liberal instantiation of the variables in the sense that the edges in the output graph are not restricted by edges in the input graph.

As an attempt to remedy this, we may impose the restriction on each rule r that all variables of r must be "chained together" by edge predicates in the positive body of r. Unfortunately this is not enough: Already a simple rule like `p(X,Z) :- e(X,Y), e(Y,Z)` destroys bounded treewidth. To see this, consider the class of all stars, i.e., undirected graphs where one vertex is adjacent to all other vertices, and let the class of input graphs be its directed version, where we have the directed edges (a, b) and (b, a) instead of an undirected edge $\{a, b\}$. Clearly this class has bounded treewidth because stars are trees. However, when given a star with n vertices, the output graph has linear treewidth because it contains the complete bipartite graph $K_{n-1,n-1}$ as a minor. Instead of further restricting the syntax of the rules, in this work we impose a restriction on the input graphs, namely that they have bounded degree.

Even for input graphs of bounded degree, it is crucial that the atoms that connect the variables in each rule appear in the *positive* body. Otherwise we could construct a program like the following.

```
p(X,Y) :- v(X), v(Y), not e(X,Y).
p(X,Y) :- v(X), v(Y), e(X,Y).
```

Here, for any input graph having n vertices, the output graph contains K_n as a minor.

Another observation is that it is important that the atoms that connect the variables of a rule are extensional. Otherwise the following program would be allowed, which materializes the transitive closure of the edge relation.

```
t(X,Y) :- e(X,Y).
t(X,Z) :- t(X,Y), e(Y,Z).
```

For any connected input graph, the output graph would again contain K_n as a minor. In the restriction that we explore in this work, we exclude this because in the second rule `X` is not part of an extensional atom. An alternative approach could be to avoid the construction of transitivity by disallowing a certain kind of cycles in a dependency graph of the program, similar to the concept of *tightness* in ASP. We will not pursue that approach in this work, however, and leave it for future work.

In fact, real-world grounders perform certain optimizations, which we ignore in this work, that would not lead to the treewidth being destroyed by some of our examples. For instance, if we have the rule `p(X,Y) :- v(X), v(Y)`, then K_n would not be a minor of the output graph of real-world grounders. The reason is that they may simplify every ground instantiation of this rule by removing both extensional atoms from the body because these atoms follow from the input facts and are thus true in every model. In our conception of a grounder, such simplifications are not performed. However, this is not a huge restriction because we could easily adapt this example to again destroy bounded treewidth even for grounders that perform such simplifications: We could replace both occurrences of the predicate `v` in the body by a new predicate `w` and add a rule like `{w(X) : v(X)}`, which guesses for each vertex x whether `w(x)` is true. A grounder would not be able to simplify the resulting rule `p(x,y) :- w(x), w(y)` because the body atoms are not deterministic consequences of the input.

4.2 A Treewidth-Preserving Class of Non-Ground ASP

The "ill-behaved" modeling techniques from Section 4.1 destroy bounded treewidth, but they are not necessary conditions. We still lack a positive result that indicates under which circumstances bounded treewidth is preserved. As we have seen, bounded treewidth is a relatively fragile property, meaning that already quite simple ASP programs destroy it. Hence we focus on preserving bounded treewidth together with bounded degree instead, which is more robust.

In this section, we present a class of ASP programs that preserves bounded treewidth for graphs of bounded degree. In fact, we even show that bounded clique-width together with bounded degree leads to groundings of bounded treewidth because bounded clique-width and bounded treewidth coincide on graphs of bounded degree [13, Corollary 1.53]. Before we define our class, we first need to formalize the notion of "chaining variables together" from Section 4.1. The idea is to restrict the structure of the output graph in a certain way by the structure of the input graph.

Definition 2. *Let r be a (non-ground) ASP rule. The* join graph *of r is the graph whose vertices are the variables in r and where there is an edge between two variables if these variables occur together in some positive body atom in r. The variables of r are* connected *if they occur in a positive extensional body atom in r and the join graph of r is connected.*

This allows us to define the class of programs that is of interest for this work.

Definition 3. *We call a (non-ground) ASP program Π* structure-restricted *if, for each rule r in Π, the variables of r are connected.*

We use this class to state the main theorem of this work.

Theorem 4. *Let Π be a structure-restricted ASP program and G be a graph. The primal and incidence graph of the grounding of Π together with G (as a set of facts) have bounded treewidth and bounded degree if G has bounded clique-width and bounded degree.*

We prove Theorem 4 using the framework of MSO transductions. First we discuss how we can turn a structure-restricted ASP program Π into a certain normal form to simplify the proof. Then we state the formulas that define our transduction, which formalizes how the incidence graph of the grounding of Π together with facts describing a graph G can be constructed from G. Finally, we discuss why this constitutes a proof of Theorem 4.

Simplifying the ASP program. Let Π be a structure-restricted ASP program. In the following, we use Π to construct a simplified program Γ together with a set of facts F such that, for every set of facts G that describe an input graph, $\Pi \cup G$ is equivalent to $\Gamma \cup F \cup G$.

To obtain Γ from Π, we first eliminate constants by the following rewriting: We introduce a new unary predicate C_a for each constant a. Then, for each constant a and each rule r containing a, we replace each occurrence of a by a new variable X_a and we add $C_a(X_a)$ to the positive body of r. Finally, for each such new predicate C_a, we put a fact $C_a(a)$ into F.

We call variables that we added to replace constants *constant variables* and all other variables *proper variables*. We denote the set of all new predicates C_a that we added to replace constants a by $\mathrm{CV}(\Gamma)$. Since Π is structure-restricted and thus has connected variables in each rule, Γ has connected proper variables in each rule, but the constant variables need not be connected. However, these will only be instantiated in a controlled way due to the facts we added to F.

Next, we eliminate propositional atoms and enforce that each rule in Γ contains at least one proper variable: We add an atom $\mathrm{dummy}(X)$ to the positive body of each rule in Γ, where X is a new proper variable and dummy is a new predicate, then replace each propositional atom q with $q(X)$ and put a fact $\mathrm{dummy}(a)$ into F, where a is an arbitrary new constant.

In the following, we consider Π (and thus Γ and F) to be fixed and write n to denote the maximum number of distinct variables that occur in a rule of Γ.

Specifying the input graphs. Input graphs are simple, ordered and directed graphs whose maximum degree is bounded by some constant d. The input structures of our MSO formulas encode such graphs as well as the facts in F, which we obtained from Π. Hence the domain of an input structure not only contains vertices of the input graph but also objects that correspond to constants in F. The signature of input structures contains a unary "vertex" relation and binary "edge" relation, which are used for specifying the input graph, as well as the unary relations in $\mathrm{CV}(\Gamma) \cup \{\mathrm{dummy}\}$ for encoding F.

Even though the MSO transduction that we present in this work operates on graphs directly instead of their incidence structures, it not only preserves bounded clique-width but also bounded treewidth. This is because (a) we restrict ourselves to simple input graphs of bounded degree, (b) our transformation preserves bounded degree, and (c) it is known that bounded clique-width and bounded treewidth coincide on simple graphs of bounded degree.

The following formula states which structures our transformation is defined for, by expressing that the maximum degree is at most d and each predicate in the set $\mathrm{CV}(\Gamma) \cup$

{dummy} has exactly one domain element in its extension.

$$\chi \;\equiv\; \forall x \neg \exists y_1 \cdots \exists y_{d+1} \left(\bigwedge_{1 \le i < j \le d+1} y_i \neq y_j \wedge \bigwedge_{i=1}^{d+1} \big(\text{edge}(x, y_i) \vee \text{edge}(y_i, x) \big) \right)$$
$$\wedge \bigwedge_{P \in \text{CV}(\Gamma) \cup \{\text{dummy}\}} \exists x \Big(P(x) \wedge \neg \exists y \big(x \neq y \wedge P(y) \big) \Big)$$

Auxiliary formulas. We will use the following auxiliary formulas to express that a vertex y is the i-th direct successor of a vertex x, for $1 \le i \le d$:

$$\text{edge}_i(x, y) \;\equiv\; \text{edge}(x, y) \wedge \neg \exists z \left(z < y \wedge \text{edge}(x, z) \wedge \bigwedge_{j=1}^{i-1} \neg\, \text{edge}_j(x, z) \right)$$

The proper variables of each rule r must be connected in the join graph. Hence, for any pair of proper variables x, y that occur together in r, we can observe that the body of every ground instantiation of r that substitutes x and y with vertices that are "too far apart" will always be false under any interpretation that encodes the input graph, since we cannot infer extensional atoms. Such instantiations will not be produced by a reasonable grounder. To be precise, two vertices are "too far apart" from another if the distance between them is at least n: There are at most n distinct variables in a rule, so we can only make at most $n-1$ steps from any vertex. Since the program is fixed and d is a constant, for each vertex v there is a constant number of vertices that we can reach from v in n steps. With this in mind, we define the formula $\text{reach}_{(i_1,\dots,i_k)}(x, y)$, for $0 \le k \le n$, to express that y is reachable from x via a sequence of vertices $(x_0, x_1, \dots, x_k)$ such that $x_0 = x$, $x_k = y$, and x_j is the i_j-th direct successor of x_{j-1}, for $1 \le j \le k$. We write ε to denote the empty tuple.

$$\text{reach}_\varepsilon(x, y) \;\equiv\; x = y$$
$$\text{reach}_{(i_1,\dots,i_k)}(x, y) \;\equiv\; \exists z \left(\text{reach}_{(i_1,\dots,i_{k-1})}(x, z) \wedge \text{edge}_{i_k}(z, y) \right) \quad \text{for } 1 \le k \le n$$

Since the maximum degree is at most d, we can uniquely identify each path of length at most n by the starting vertex and a number in $\{0, \dots, \sum_{i=1}^n d^i\}$. We call every element of $\{0, \dots, \sum_{i=1}^n d^i\}$ a *path number*. For every path number p, we write $\hat{p}$ to denote the *edge sequence* of p, i.e., the sequence of direct successor indices that is uniquely identified by p. (For our purposes it does not matter which edge sequence gets which path number, but we require that different edge sequences have different path numbers.)

There can be multiple paths from x to y. We will need to single out one of them for our transduction. For each path number p, we therefore define the formula $\text{fp}_p(x, y)$, which is true iff p is the smallest path number that encodes an actual path from x to y.

$$\text{fp}_p(x, y) \;\equiv\; \text{reach}_{\hat{p}}(x, y) \wedge \bigwedge_{q=0}^{p-1} \neg\, \text{reach}_{\hat{q}}(x, y)$$

We call a path number p *valid* from an input vertex a if there is an input vertex b such that $\mathrm{fp}_p(x,y)$ evaluates to true under $x \mapsto a,\ y \mapsto b$.

For each path number or constant p, we define an auxiliary formula $\mathrm{correct}_p(x,y)$ that is true iff y is interpreted by the domain element denoted by p (where the value of x is used as the starting vertex if p is a path number).

$$\mathrm{correct}_p(x,y) \;\equiv\; \begin{cases} \mathrm{fp}_p(x,y) & \text{if } p \text{ is a path number} \\ C_p(y) & \text{if } p \text{ is a constant} \end{cases}$$

Constructing the output graph. Using these auxiliary formulas, we define the remaining formulas of our MSO transduction.[9] The class of input graphs is characterized by the formula χ from above. Our transduction formalizes how we can turn an input graph represented as a set of facts G into the incidence graph of the program obtained by grounding $\Gamma \cup F$ together with G.

To make things easier to read, we do not use integers as the indices i, j of δ_i and $\theta_{i,j}$, but objects that make the intended meaning of the respective formulas clearer. For instance, for every propositional atom q, we will define a formula $\delta_q(x)$ that will evaluate to true for exactly one domain element that we use to interpret the free variable x; the intended purpose of the corresponding output vertex is to represent the atom q in the incidence graph of the grounding. Using objects such as q as an index obviously does not change the nature of the approach but can be seen as "syntactic sugar" since there is a constant number of objects that we use as indices in this way.

For every predicate P of (positive) arity k and for each sequence $p_1, \dots, p_{k-1}$ of path numbers or constants, we define a formula $\delta_{P[p_1,\dots,p_{k-1}]}(x)$ that is true iff we interpret x by an input vertex a such that each p_i that is a path number is valid from a. The idea is that for each sequence $b_1, \dots, b_{k-1}$ such that b_i is a constant or reachable from a in at most n steps we put a copy of a into the output graph to represent the ground atom $P(a, b_1, \dots, b_{k-1})$.

$$\delta_{P[p_1,\dots,p_{k-1}]}(x) \;\equiv\; \bigwedge_{i=1}^{k-1} \exists y\ \mathrm{correct}_{p_i}(x,y)$$

We proceed in a similar way with rules. In contrast to before, we only make copies for each possible value of the proper variables and we ignore the constant variables, since a reasonable grounder will only instantiate these by the constants they actually stand for. For each rule r in Γ with k proper variables ($k > 0$) and each sequence of path numbers $p_1, \dots, p_{k-1}$, we define the following formula.

$$\delta_{r[p_1,\dots,p_{k-1}]}(x) \;\equiv\; \mathrm{vertex}(x) \wedge \bigwedge_{i=1}^{k-1} \exists y\ \mathrm{correct}_{p_i}(x,y)$$

Here the conjunct $\mathrm{vertex}(x)$ makes sure that we only make copies of actual input vertices, but not of domain elements that only stand for constants.

[9] In fact our MSO transduction happens to be a first-order transduction.

We now define the formulas that specify the edges of the output graph. Let r be a rule with (a positive number of) proper variables $X_1, \dots, X_k$ and (a non-negative number of) constant variables $X_{k+1}, \dots, X_{k+m}$ that stand for constants we denote by $c_1, \dots, c_m$, respectively, and let $p_1, \dots, p_{k-1}$ be a sequence of path numbers. Let P be a predicate of (positive) arity l and let $A_P(r)$ be the set of tuples $(i_1, \dots, i_l)$ of integers such that $P(X_{i_1}, \dots, X_{i_l})$ occurs in r. We first define the following auxiliary formula:

$$\mathrm{compat}_{r,P}(x_1, \dots, x_{k+m}, y_1, \dots, y_l) \equiv \bigvee_{(i_1,\dots,i_l) \in A_P(r)} \bigwedge_{j=1}^{l} y_i = x_{i_j}$$

The intuition is that a structure satisfies $\mathrm{compat}_{r,P}(x_1, \dots, x_{k+m}, y_1, \dots, y_l)$ iff it interprets the variables of r in such a way that the ground instantiation of r according to the values of $x_1, \dots, x_{k+m}$ contains an atom $P(a_1, \dots, a_l)$ such that $a_1, \dots, a_l$ are exactly the values of $y_1, \dots, y_l$. We use this to make sure that whenever we instantiate the variables of r in such a way that the resulting ground rule r' contains $P(a_1, \dots, a_l)$, the incidence graph of the output program contains an edge between $P(a_1, \dots, a_l)$ and r'. For each sequence of path numbers or constants $q_1, \dots, q_{l-1}$, we therefore define the following formula:

$$\begin{aligned}\theta_{r[p_1,\dots,p_{k-1}],P[q_1,\dots,q_{l-1}]}(x,y) \equiv\ & \exists x_1 \cdots \exists x_{k+m} \exists y_1 \cdots \exists y_l \big(x = x_1 \wedge y = y_1 \\ & \wedge \mathrm{correct}_{p_1}(x, x_2) \wedge \cdots \wedge \mathrm{correct}_{p_{k-1}}(x, x_k) \\ & \wedge C_{c_1}(x_{k+1}) \wedge \cdots \wedge C_{c_m}(x_{k+m}) \\ & \wedge \mathrm{correct}_{q_1}(y, y_2) \wedge \cdots \wedge \mathrm{correct}_{q_{l-1}}(y, y_l) \\ & \wedge \mathrm{compat}_{r,P}(x_1, \dots, x_{k+m}, y_1, \dots, y_l) \big)\end{aligned}$$

Obtaining the actual incidence graph from the result. This MSO transduction defines a graph that differs in the following ways from the incidence graph of the grounding of Π together the facts describing the input graph: First, the output graph may contain several isolated vertices that are not in the incidence graph of the grounding. For instance, the formula $\delta_{\mathrm{edge}[p]}(x)$, where p is some path number, evaluates to true under every interpretation of x even if this vertex has no neighbors at all. This causes that the output graph may contain a vertex that corresponds to an atom $\mathrm{edge}(a, b)$ while there is no such edge in the input graph. However, this does not matter because we can just delete such vertices without destroying bounded treewidth or clique-width.[10]

The second difference between the output graph and the incidence graph of the grounding is that our MSO transduction does not take care of the facts encoding the input graph, whereas for every such fact the grounding contains a rule vertex that is connected to the respective atom vertex. It is easy to see that adding the missing rule vertices and connecting them with the respective atom vertices increases the treewidth or clique-width by at most 1.

[10]The class of graphs of bounded treewidth and the class of graphs of bounded clique-width are both hereditary (i.e., closed under taking subgraphs).

Finally, the output graph contains atom vertices involving the predicates from $\mathrm{CV}(\Gamma) \cup \{\mathrm{dummy}\}$, which are not present in the incidence graph of the grounding. Again, we can just delete these vertices and their incident edges without increasing the treewidth or clique-width.

Discussion. Recall that Π is fixed, d is bounded and the grounder does not instantiate two variables in a rule with vertices whose distance from each other is more than $n - 1$. Hence the number of produced instantiations of a rule r depends only on n and d. This implies that the incidence graph of the grounding also has bounded degree. As we have just seen, we can easily obtain this graph from the output graph of the transduction without significantly increasing the clique-width. Furthermore, as we have already mentioned, our transduction preserves bounded clique-width, and bounded clique-width coincides with bounded treewidth on graphs of bounded degree. It follows that grounding Π together with a set of facts describing an input graph of degree d leads to a program whose incidence treewidth depends only on d and the clique-width of the input graph. To prove Theorem 4 it remains to show that also the primal graph has both bounded degree and bounded treewidth. The fact that the incidence graph has bounded degree clearly implies that the primal graph also has bounded degree. It also implies that the primal graph has bounded treewidth because we can turn a tree decomposition of the incidence graph into one of the primal graph by replacing the rule vertices in the bags with all adjacent atom vertices, which increases the treewidth only by a constant.

5 Conclusion

In this work we investigated how the treewidth of grounded ASP programs is affected by different modeling constructs in the non-ground program. We first performed experiments with several problems and, for each problem, various modeling alternatives. As expected, certain language constructs lead to a significant increase in the treewidth of the grounding, when compared to the treewidth of the input graph. Interestingly, we observed that low treewidth of the grounding often correlates with low running time of state-of-the-art ASP solvers. We then showed that splitting non-ground rules up using a recently proposed rule decomposition technique may reduce not only the size of the groundings but also their treewidth.

Next, we turned to a theoretical analysis. As our experiments showed, different non-ground modeling variants lead to groundings of quite different treewidths and subsequent solver running times. It is therefore interesting which features of non-ground ASP encodings guarantee that the treewidth of the groundings is not significantly higher than the treewidth of the input. To clarify this, we introduced a syntactically restricted class of non-ground ASP called structure-restricted programs, and we showed that the treewidth of the grounding of a structure-restricted program together with some input depends only on the degree and treewidth (in fact even clique-width) of the input but not on its size.

Further research on the effect of the treewidth of a ground program on ASP solver running time is still needed to obtain a clear picture. Besides more experiments in this direction, a theoretical analysis of the solving process of CDCL-based solvers should be performed. Our experiments suggest that CDCL-based solving often implicitly takes advantage of low treewidth of the program. Another subject of future work is the incorporation of rule decomposition into ASP grounders. This allows us to avoid those auxiliary domain rules which increase the treewidth. Finally, we plan to study restrictions on non-ground programs other than structure-restrictedness to find other program classes that preserve bounded treewidth as well but may be employed in different situations.

References

[1] Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, and Stefan Woltran. Computing secure sets in graphs using answer set programming. *Journal of Logic and Computation*, 2015.

[2] Mario Alviano, Carmine Dodaro, Wolfgang Faber, Nicola Leone, and Francesco Ricca. WASP: A native ASP solver based on constraint learning. In *Proc. LPNMR*, pages 54–66, 2013.

[3] Mario Alviano, Wolfgang Faber, Nicola Leone, Simona Perri, Gerald Pfeifer, and Giorgio Terracina. The disjunctive datalog system DLV. In *Datalog Reloaded. Revised Selected Papers*, pages 282–301, 2010.

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

[5] Manuel Bichler. Optimizing non-ground answer set programs via rule decomposition. BSc Thesis, TU Wien. `http://dbai.tuwien.ac.at/proj/lpopt`, November 2015.

[6] Manuel Bichler, Michael Morak, and Stefan Woltran. lpopt: A rule optimization tool for answer set programming. In *Proc. LOPSTR 2016. To appear. See also CoRR abs/1608.05675*, 2016.

[7] Manuel Bichler, Michael Morak, and Stefan Woltran. The power of non-ground rules in answer set programming. In *Proc. ICLP. To appear. See also CoRR abs/1608.01856*, 2016.

[8] Bernhard Bliem, Michael Morak, and Stefan Woltran. D-FLAT: declarative problem solving using tree decompositions and answer-set programming. *TPLP*, 12(4-5):445–464, 2012.

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

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

[11] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giomvambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Francesco Ricca, and Torsten Schaub. ASP-Core-2 Input Language Format v2.03c, 2015. Accessed: 2016-06-27.

[12] Günther Charwat and Andreas Pfandler. Democratix: A declarative approach to winner determination. In *Proc. ADT 2015*, volume 9346 of *LNCS*, pages 253–269. Springer, 2015.

[13] Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach*, volume 138 of *Encyclopedia of mathematics and its applications*. Cambridge University Press, 2012.

[14] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[15] Islam Elkabani, Enrico Pontelli, and Tran Cao Son. Smodels^{A} - A system for computing answer sets of logic programs with aggregates. In *Proc. LPNMR*, pages 427–431, 2005.

[16] Johannes K. Fichte. daajoe/gtfs2graphs – a GTFS transit feed to graph format converter. `https://github.com/daajoe/gtfs2graphs`, 2016.

[17] Johannes K. Fichte, Markus Hecher, Michael Morak, and Stefan Woltran. Answer set solving using tree decompositions and dynamic programming — the dynASP2 system —. Technical Report DBAI-TR-2016-101, DBAI, Fakultät für Informatik an der Technischen Universität Wien, 2016.

[18] Johannes Klaus Fichte, Markus Hecher, Michael Morak, and Stefan Woltran. Counting answer sets via dynamic programming. In *Trends and Applications of Answer Set Programming (KI 2016)*, 2016.

[19] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Marius Lindauer, Max Ostrowski, Javier Romero, Torsten Schaub, and Sven Thiele. Potassco User Guide 2.0. Available at `https://sourceforge.net/projects/potassco/files/guide/2.0/guide-2.0.pdf`, 2015.

[20] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[21] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[22] Martin Gebser, Roland Kaminski, and Torsten Schaub. Grounding recursive aggregates: Preliminary report. *CoRR*, abs/1603.03884, 2016.

[23] 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.

[24] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187:52–89, 2012.

[25] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proc. ICLP/SLP*, pages 1070–1080, 1988.

[26] Michael Jakl, Reinhard Pichler, and Stefan Woltran. Answer-set programming with bounded treewidth. In *Proc. IJCAI'09*, 2009.

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

[28] Victor W. Marek and Mirosław Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In *The Logic Programming Paradigm – A 25-Year Perspective*, pages 375–398. Springer, 1999.

[29] Michael Morak, Reinhard Pichler, Stefan Rümmele, and Stefan Woltran. A dynamic-programming based ASP-solver. In *Proc. JELIA'10*, pages 369–372, 2010.

[30] Michael Morak and Stefan Woltran. Preprocessing of complex non-ground rules in answer set programming. In *Proc. ICLP*, pages 247–258, 2012.

[31] Reinhard Pichler, Stefan Rümmele, Stefan Szeider, and Stefan Woltran. Tractable answer-set programming with weight constraints: bounded treewidth is not enough. *Theory and Practice of Logic Programming, TPLP*, 14:141–164, 3 2014.

[32] Tommi Syrjänen. Lparse 1.0 user's manual. `http://www.tcs.hut.fi/smodels/lparse.ps.gz`, 2000.