Declarative Encodings of Acyclicity Properties*

Martin Gebser**, Tomi Janhunen, and Jussi Rintanen* **

Helsinki Institute for Information Technology HIIT
Department of Information and Computer Science
Aalto University, FI-00076 AALTO, FINLAND

Abstract. Many knowledge representation tasks involve trees or similar structures as abstract datatypes. However, devising compact and efficient declarative representations of such structural properties is non-obvious and can be challenging indeed. In this paper, we take a number of acyclicity properties into consideration and investigate various logic-based approaches to encode them. We use answer set programming as the primary representation language but also consider mappings to related formalisms, such as propositional logic, difference logic, and linear programming. We study the compactness of encodings and the resulting computational performance on benchmarks involving acyclic or tree structures.

1 Introduction

Numerous hard search tasks involve the construction of acyclic or tree structures. Constraint satisfaction and related methods are an important approach for solving many of these problems. For instance, Bayesian network structure learning, where directed acyclic graphs provide solution candidates, can be reduced to constraint optimization [15, 10]. Furthermore, constraint-based methods can be used to infer phylogenetic trees [4, 2], describing the evolution of living organisms, languages, and other evolving systems. Since acyclicity and the property of being a tree are no primitives in common constraint-based representation formalisms, the challenge of formulating such conditions in terms of more basic constraint expressions arises. Hence, in this work, we systematically investigate logic-based approaches to encode respective properties.

Although we rely on answer set programming (ASP) as the primary representation language for encodings, respective formulations in related formalisms, such as propositional satisfiability (SAT), difference logic (DL), and linear programming (LP), can be obtained through automatic polynomial translations from ASP. Linear translations to SAT exist whenever ASP rules are *tight* [11], i.e., if there are no circular positive dependencies through rules' prerequisites on the ground level. In the non-tight case, level mappings [16] can be used to bridge the semantic gap between ASP and SAT. Compact linear representations of level mappings can be achieved by means of difference constraints available in DL [21] or integer variables in LP [18].

The rest of this paper is organized as follows. After providing definitions of acyclicity and tree properties, we present encoding approaches and corresponding first-order

* The support from the Finnish Centre of Excellence in Computational Inference Research (COIN) funded by the Academy of Finland (under grant #251170) is gratefully acknowledged.

** Also affiliated with the University of Potsdam, Germany.
*** Also affiliated with Griffith University, Brisbane, Australia.

(a) Cyclic (b) Acyclic (c) Forest (d) Tree

Fig. 1: Directed example graphs with five vertices each

ASP formulations in Section 3. In Section 4, we evaluate these encodings on synthetic as well as application benchmarks, followed by conclusions in Section 5. The material on directed acyclic graphs was also presented in a short paper [13], which did not cover forests and trees. An extended draft (also treating chordal graphs) is available online.[1]

2 Acyclicity Properties

As usual, a *directed graph* G is a pair $\langle V, E\rangle$, where V is a finite set of *vertices* and $E \subseteq V \times V$ is a set of directed *edges*. For some $v \in V$, we denote the number of incoming or outgoing edges, respectively, by $\deg^-(v) = |\{u \mid \langle u, v\rangle \in E\}|$ and $\deg^+(v) = |\{u \mid \langle v, u\rangle \in E\}|$; v is a *root* (or *leaf*) of G if $\deg^-(v) = 0$ (or $\deg^+(v) = 0$). A *path* of length $k-1$ in G is a non-empty sequence $v_1, \ldots, v_k$ of vertices from V such that $\langle v_i, v_{i+1}\rangle \in E$ and $v_i \neq v_j$ for all $1 \leq i < j \leq k$. A sequence $v_0, v_1, \ldots, v_k$ is a *cycle* of length k in G if $\langle v_0, v_1\rangle \in E$, $v_0 = v_k$, and $v_1, \ldots, v_k$ is a path in G.

A *directed acyclic graph* is a directed graph G such that there is no cycle in G. A directed acyclic graph $G = \langle V, E\rangle$ is a *directed forest* if, for every $v_k \in V$, there is exactly one path $v_1, \ldots, v_k$ in G from a root v_1 of G to v_k. Given that any incoming edge $\langle v_{k-1}, v_k\rangle$ can be extended to a path $v_1, \ldots, v_{k-1}, v_k$ from a root v_1, the former condition is equivalent to requiring $\deg^-(v) \leq 1$ for all vertices $v \in V$. Finally, a *directed tree* is a directed forest $G = \langle V, E\rangle$ with a unique root, i.e., $\deg^-(v) = 0$ holds for exactly one $v \in V$. Some directed example graphs illustrating the introduced acyclicity properties are depicted in Figure 1.

An *undirected graph* $G = \langle V, E\rangle$ consists of a finite set V of *vertices* and a set E of undirected *edges* $\{u, v\}$ such that $u, v \in V$ and $u \neq v$. For some $v \in V$, $\deg(v) = |\{u \mid \{u, v\} \in E\}|$ denotes the number of edges including v; v is a *leaf* of G if $\deg(v) \leq 1$. A *path* $v_1, \ldots, v_k$ in G is defined as in the directed case except for replacing the requirement $\langle v_i, v_{i+1}\rangle \in E$ by $\{v_i, v_{i+1}\} \in E$ for all $1 \leq i < k$. A sequence $v_0, v_1, \ldots, v_k$ is a *cycle* of length $k \geq 3$ in G if $\{v_0, v_1\} \in E$, $v_0 = v_k$, and $v_1, \ldots, v_k$ is a path in G. An *undirected forest* is an undirected graph G such that there is no cycle in G. An *undirected tree* is an undirected forest G such that, for every pair $u, v \in V$, there is some path from u to v in G.

3 Encodings

In the following, we gradually develop different encodings of the graph-theoretic concepts introduced in the previous section in the first-order input language of the ASP grounder GRINGO [14]. The underlying principles, however, are of general applicability, and we outline particularities of respective SAT, DL, and LP formulations.

[1] `http://research.ics.aalto.fi/software/asp/etc/asp-acyclic.pdf`

```
1  #const n=5.
2  node(1..n).
3  pair(X,Y) :- node(X;Y), X != Y.
4  { edge(X,Y) } :- pair(X,Y).
```

Fig. 2: Encoding part for generating directed graphs

```
5  order(X,Y) :- pair(X,Y), not edge(X,Y).
6  order(X,Y) :- pair(X,Y), order(Y).
7  order(X) :- node(X), order(X,Y) : pair(X,Y).

9  :- node(X), not order(X).
```

Fig. 3: Inductive bottom-up encoding of acyclicity test

3.1 Directed Acyclic Graphs

To begin with, we consider directed graphs and their properties, focusing on the distinction of cyclic and acyclic graphs like the ones depicted in Figure 1(a) and 1(b).

Rules describing the (non-deterministic) generation of directed graphs are shown in Figure 2. The predicate `node`/1 provides the labels $1, \ldots, \mathbf{n}$ for vertices, where $\mathbf{n}$ is an integer constant standing for the number of vertices, and the symmetric predicate `pair`/2 represents the domain of directed edges given by all pairs of distinct vertices. Any subset of these pairs can be generated via the choice rule in line 4, permitting instances of `edge`/2 to hold without further preconditions. For example, the atoms characterizing the directed graph in Figure 1(a) are `edge(1,2)`, `edge(1,3)`, `edge(2,4)`, `edge(3,2)`, `edge(4,5)`, and `edge(5,3)`.

Acyclicity Checking To make sure that a generated directed graph $\langle V, E \rangle$ is acyclic, we may check whether there is a strict partial order $<$ over vertices in V such that $u < v$ if $\langle u, v \rangle \in E$. The rules in Figure 3 encode this approach in an inductive fashion, where ground instances of `order(X,Y)` and `order(X)` indicate the absence of cycles through edge candidates or vertices, respectively.

Reconsidering the example graph in Figure 1(a), `order(`u`,`v`)` can be derived for distinct vertices u and v such that $\langle u, v \rangle$ is not an edge. Since each vertex u has some successor v, there still is some atom `order(`u`,`v`)` for u that cannot be concluded in this way, e.g., `order(5,3)` for vertex `5`. Hence, `order(`u`)` remains underivable for all $u \in \{1, 2, 3, 4, 5\}$, and the cyclic graph in Figure 1(a) is rejected by means of the rules in Figure 3. Unlike this, vertex `5` has no successor in the acyclic graph shown in Figure 1(b), so that `order(5,`v`)` is derivable for $v \in \{1, 2, 3, 4\}$. This in turn yields `order(5)`, `order(4,5)`, and `order(4)`, given that `5` is the only successor of vertex `4`. Similarly, the derivation of `order(2,4)` leads to `order(2)`, and the establishment of `order(3,2)`, `order(3,4)`, and `order(3,5)` for the successors of vertex `3` allows for deriving `order(3)`. Finally, `order(1)` can be concluded in view of `order(1,2)` and `order(1,3)`. As derivable atoms comply with (ground instances of) the integrity constraint in line 9 of Figure 3, the graph in Figure 1(b) passes the acyclicity test.

Encoding Variants The encoding in Figure 3 is *non-tight*, i.e., it relies on well-foundedness in the presence of circular positive dependencies on the ground level. Level mappings (cf. [16, 21, 18]) furnish an alternative mechanism to express well-founded derivations, where difference constraints or integer variables allow for linear embed-

```
5  order(X,Y,1..n) :- pair(X,Y), not edge(X,Y).
6  order(X,Y,N-1)  :- pair(X,Y), order(Y,N).
7  order(X,N) :- node(X), order(X,Y,N) : pair(X,Y), N = 2..n.
8  order(X)   :- node(X), order(X,Y,1) : pair(X,Y).
```

Fig. 4: Unfolded bottom-up derivation of `order`/1

dings in DL and LP. A respective approach is taken by the *tight* ASP formulation of the predicate `order`/1 in Figure 4. In order to eliminate circular positive dependencies between rules' prerequisites and conclusions, the auxiliary predicates `order`/3 and `order`/2 include an additional argument for "step counting" that unfolds the derivation of a partial vertex order $<$ witnessing acyclicity. If such an order exists, its construction must be completed in at most **n** steps. For instance, the (total) order $1 < 3 < 2 < 4 < 5$ over vertices of the acyclic graph in Figure 1(b) is represented in terms of the atoms `order(5,2...5)`, `order(4,2...4)`, `order(2,2...3)`, and `order(3,2)` along with `order(`u`)` for $u \in \{1, 2, 3, 4, 5\}$. On the one hand, explicit step counting eliminates the potential of circular derivations, so that straightforward translations like completion [7] can be used to map the encoding in Figure 4 to SAT, DL, or LP. In fact, the tight ASP formulation of `order`/1 can be viewed as a directed counterpart of the SAT encoding for the undirected case in [8]. On the other hand, the step argument introduces an additional dimension increasing the number of atoms as well as constraints. For directed graphs $\langle V, E\rangle$, the size of ground instances grows from $\mathcal{O}(|E|)$ for the encoding in Figure 3 to $\mathcal{O}(|E|\times|V|)$, thus shifting from linear to quadratic space.

Variants of the encodings presented so far can be obtained by replacing the *unconditional* generation of edges in line 4 of Figure 2 by the following choice rule:

```
{ edge(X,Y) } :- pair(X,Y), order(Y).
```

The additional prerequisite `order(Y)` necessitates the absence of cycles through a vertex substituted for `Y` before admitting any edge from a predecessor substituted for `X`. Hence, the generation of edges with the above choice rule progresses successively from the leaves of a directed acyclic graph. Whether such *conditional* edge generation is advantageous or not is empirically investigated in Section 4. While the *leaf encodings* in Figure 3 and 4 describe "bottom-up" traversals starting from the leaves of a directed graph, acyclicity tests can likewise be performed "top-down" from roots, and a successive choice rule similar to the one above can optionally be used in *root encodings* instead of line 4 in Figure 2. Without other side constraints, bottom-up and top-down traversals appear fully symmetric, but in the contexts of forests and trees, considered in the following, the orientation of acyclicity tests may interact or interfere.

3.2 Directed Forests

In order to switch from directed acyclic graphs $\langle V, E\rangle$ to the more restrictive notion of directed forests, we have to make sure that $\deg^-(v) \leq 1$ holds for every vertex $v \in V$. For instance, the acyclic graph shown in Figure 1(b) is not a forest because $\deg^-(2) = \deg^-(4) = \deg^-(5) = 2$. The (sub)graph in Figure 1(c), however, is a forest with roots 1 and 3, given that $\deg^-(1) = \deg^-(3) = 0$ and $\deg^-(2) = \deg^-(4) = \deg^-(5) = 1$.

Five alternative approaches to test the additional requirement of at most one incoming edge per vertex are encoded by the rules in Figure 5(a)–(e). The integrity con-

straint shown in Figure 5(a) checks pairwise mutual exclusion by, for every vertex, enumerating distinct predecessor candidates and denying the joint occurrence of incoming edges for each pair of potential predecessors. While this approach is straightforward, its cubic space complexity, i.e., $\mathcal{O}(|V|^3)$, is a major bottleneck for scalability. The usage of one cardinality constraint per vertex in Figure 5(b) allows for a more compact (quadratic) representation since pairs of predecessor candidates are not explicitly enumerated. However, cardinality constraints of arbitrary arity are not directly available in SAT and DL input languages, and the rules in Figure 5(c)–(e) encode *normalizations* (cf. [1]) for the purpose of checking mutual exclusion.

Bidirectional Traversal The basic idea of the encoding in Figure 5(c) is to traverse the potential predecessors of a vertex from both ends of an (arbitrary) order. For convenience, we here rely on the natural order given by vertex indexes along with the background information that any pair of distinct vertices may possibly be connected by an edge (cf. line 3 of Figure 2). In case of a more restrictive predecessor selection, the candidates to be traversed and the order among them can of course be localized. However, the entry points for traversing potential predecessors of a vertex substituted for `X` are in line 10 and 11 of Figure 5(c) taken to be the vertices with the smallest and the greatest label, i.e., `1` and the integer constant represented by **n**. Starting from them, the rule in line 12 expresses that proceeding "upwards" or "downwards", respectively, to the next vertex in direction `D` is admissible if a current vertex substituted for `Y` does not have an edge to `X`. That is, a vertex substituted for `X` has at most one incoming edge if and only if some `Y` is encountered from both ends of the underlying order, as indicated by the atoms `unique(X,Y,1)` and `unique(X,Y,-1)`. Except for the corner case **n**=1, a respective `Y` can be assumed to be different from `X`, which per `X` eliminates one ground instance of the rule in line 13 for deriving `unique(X)`. Finally, the integrity constraint in line 14 requires ground instances of `unique(X)` for `X`=`1`, ..., `X`=**n** to hold, thus denying any graph with more than one incoming edge for some vertex.

The test based on bidirectional traversal filters the example graph in Figure 1(b) as follows. While `unique(1,`u`,`d`)` for $u \in \{1,2,3,4,5\}$ and $d \in \{1,-1\}$, `unique(3,1,1)`, `unique(3,1,-1)`, and thus `unique(1)` and `unique(3)` are derivable, such atoms remain underivable for other vertices. For instance, traversing the potential predecessors of vertex `2` yields `unique(2,1,1)` and `unique(2,`u`,-1)` for $u \in \{3,4,5\}$ only, so that `unique(2)` cannot be concluded from `unique(2,Y,1)` and `unique(2,Y,-1)` for any common ground substitution of `Y`. Unlike this, the test succeeds for the forest shown in Figure 1(c). In particular, `unique(2)` as well as `unique(`v`)` for $v \in \{4,5\}$ are derived in view of `unique(2,1,1)` and `unique(2,1,-1)` or `unique(`v`,3,1)` and `unique(`v`,3,-1)`, respectively.

Linear Traversal The approach encoded in Figure 5(d) inspects potential predecessors of a vertex substituted for `X` "downwards" from **n** while maintaining a counter flag in the third argument of the predicate `unique`/3. The latter is unconditionally initialized with `0` in line 10, and the rule in line 12 expresses that the flag can be decreased from `1` to `0` at any point (other than `X`) in traversing the indexes of vertices. Atoms of the form `unique(`v`,`u`,1)` indicate the absence of edges leading to v for the vertices with labels $u, \ldots,$ **n**, which is in line 11 checked for **n**, as well as when proceeding to any predecessor candidate with the next smaller label by applying the rule in line 13. The

```
10  :- node(Z), edge(X;Y,Z), X < Y.
```
(a) Pairwise mutual exclusion

```
10  :- node(Y), 2 { edge(X,Y) }.
```
(b) Cardinality constraint

```
10  unique(X,1,1)    :- node(X).
11  unique(X,n,-1)   :- node(X).
12  unique(X,Y+D,D) :- unique(X,Y,D), node(Y+D), not edge(Y,X).
13  unique(X) :- unique(X,Y,1;-1), n-2 < n*|X-Y|.
14  :- node(X), not unique(X).
```
(c) Bidirectional traversal

```
10  unique(X,n,0)    :- node(X).
11  unique(X,n,1)    :- node(X), not edge(n,X).
12  unique(X,Y-1,0) :- unique(X,Y,1), node(Y-1), X != Y-1.
13  unique(X,Y-1,C) :- unique(X,Y,C), node(Y-1), not edge(Y-1,X).
14  :- node(X), not unique(X,1,0).
```
(d) Linear traversal

```
10  unique(X,n,Y,0)      :- node(X;Y).
11  unique(X,n,Y,1)      :- node(X;Y), not edge(Y,X).
12  unique(X,I,I,C)      :- unique(X,J,J,C), I = (J+1)/2,
                            1 < J, J < 2*I.
13  unique(X,I,Y/2,C/2) :- unique(X,J,Y,C1), I = (J+1)/2, 0 < C,
                           unique(X,J,Y-1,C2), C = C1+C2, Y\2 == 0.
14  :- node(X), not unique(X,1,1,0).
```
(e) Tournament traversal

Fig. 5: Forest property tests for directed acyclic graphs

latter rule also forwards flag `0` to indicate the existence of at most one predecessor for a vertex v among $u, \ldots, \mathbf{n}$, and requiring `unique(`v`,1,0)` to hold by means of the integrity constraint in line 14 thus restricts the number of incoming edges per vertex to one or none. Note that the encoding in Figure 5(d) provides an ASP formulation of the well-known sequential counter approach from SAT [22] for the particular case of cardinality limited to at most one (cf. [19, 12]).

Tournament Traversal While linear traversal inspects vertices in (lexicographical) order, the encoding in Figure 5(e) aims at a symmetric partitioning of predecessor candidates. To this end, potential predecessors are viewed as leaves of a binary tree of depth $\lceil \log_2(\mathbf{n}) \rceil$, where `unique(X,n,Y,0)` and `unique(X,n,Y,1)` in line 10 and 11 provide base cases indicating whether the vertex substituted for `Y` may have an edge to `X` or not, respectively. The rule in line 13 then combines two such atoms, inspecting an even label `Y` along with `Y-1` to derive `unique(X,(J+1)/2,Y/2,`c`)` for $c \in \{0,1\}$: the numbers `(J+1)/2` and `Y/2` denote the round of traversal as well as the position of an outcome, and c=`0` or c=`1` represents again that the investigated predecessor candidates may have some or no edge to `X`. If the number of positions in a round is odd, the rule in line 12 additionally forwards previous outcomes for the last position, lacking an even partner, to the next round. Considering for instance vertex `5` and its incoming edge from `3` in Figure 1(c), the atom `unique(5,3,2,0)` is derived from `unique(5,5,3,0)` and `unique(5,5,4,1)`. In addition, `unique(5,3,1,0)`

and `unique(5,3,1,1)` jointly indicate that the vertices with labels `1` and `2` have no edge to `5`. Further combining successive outcomes yields `unique(5,2,1,0)` as well as `unique(5,2,2,1)` to eventually derive `unique(5,1,1,0)`, expressing that (at most) one edge to `5` has been encountered in traversing all candidates. In general, the integrity constraint in line 14 checks the respective condition for every vertex to filter directed acyclic graphs that are no forests. The presented tournament traversal approach to pairwisely compare elements as well as their aggregated outcomes resembles the so-called "commander encoding" from SAT [17], which has been generalized to cardinality constraints with arbitrary bound in [12].

Finally, note that (ground instances of) the encoding variants in Figure 5 do not involve circular positive dependencies, so that well-foundedness is not a bottleneck for corresponding SAT, DL, and LP formulations. However, SAT and DL input languages lack native support for cardinality constraints like the one in Figure 5(b),[2] and the scalability of explicit predecessor candidate pair enumeration, as in Figure 5(a), suffers from (cubic) space complexity. The normalizations in Figure 5(c)–(e) exemplify ways to encode cardinality limitations to at most one compactly, where the linear and tournament traversal formulations amount to ASP counterparts of respective approaches from SAT.

3.3 Directed Trees

Directed trees are directed forests $\langle V, E\rangle$ with one root. That is, all but one vertex in V must have some incoming edge, and since there may be at most one incoming edge per vertex, the number of edges must be $|V|-1$. Hence, either condition is sufficient to check that $\langle V, E\rangle$ is a tree. Moreover, $|V|-1$ edges are needed to connect the undirected version $\langle V, \{\{u, v\} \mid \langle u, v\rangle \in E\}\rangle$ of $\langle V, E\rangle$, so that connectedness provides a third criterion to distinguish trees. Figure 6 displays encodings of these equivalent conditions.

Edge Counting The cardinality constraint in Figure 6(a) holds if a directed forest $\langle V, E\rangle$ includes $|V|-1$ edges, and the (ground instance of the) integrity constraint in line 15 checks this condition. In view of $\mathtt{n}{=}5$, the three edges of the directed forest shown in Figure 1(c) are insufficient to pass the test, whereas it succeeds for the directed tree with four edges in Figure 1(d). Given that any pair of distinct vertices provides an edge candidate according to the declaration in line 3 of Figure 2, a ground instance of the cardinality constraint over edges involves $|V|^2-|V|$ atoms, and the bound $|V|-1$ goes beyond mutual exclusion testing, as encoded by the normalizations in Figure 5(c)–(e). While generalizations of the linear and tournament traversal approaches allow for dealing with greater cardinalities too, growing in space proportionally to the cardinality bound (cf. [22, 12]), we omit them here for brevity. However, since the number of edges cannot exceed $|V|-1$, the following integrity constraint is valid as well (if $V \neq \emptyset$):

```
:- n { edge(X,Y) }.
```

This can be viewed as a redundant formulation of an implied property of directed forests, and auxiliary atoms introduced in normalizations of the cardinality constraint in line 15 of Figure 6(a) may be reused to redundantly test the implied condition too. The effect of bounding the number of edges explicitly from above is assessed in Section 4.

[2] Cardinality constraints can be linearly translated to DL, e.g., by adopting the mapping in [20].

```
15  :- not n-1 { edge(X,Y) }.
```
(a) Edge counting

```
15  child(Y) :- edge(X,Y).
16  :- 2 { not child(X) : node(X) }.
```
(b) Root counting

```
15  reach(1) :- 0 < n.
16  reach(X) :- reach(Y), edge(X,Y).
17  reach(Y) :- reach(X), edge(X,Y).
18  :- node(X), not reach(X).
```
(c) Connectedness

Fig. 6: Tree property tests for directed forests

Unique Root The second approach encoded in Figure 6(b) focuses on the uniqueness of a root. To this end, the predicate `child`/1 is derived from `edge`/2 in line 15 to indicate vertices targeted by some edge. The (ground instance of the) integrity constraint in line 16 then denies graphs in which several vertices have no incoming edge, thus limiting the number of roots to (at most) one. For instance, the atoms `child(1)` and `child(3)` are not derived from `edge(1,2)`, `edge(3,4)`, and `edge(3,5)`, representing the edges of the forest shown in Figure 1(c), so that it is rejected by means of

```
:- 2 { not child(1), not child(2),
       not child(3), not child(4), not child(5) }.
```

For the tree in Figure 1(d), `child(3)` is additionally derivable from `edge(1,3)`, and thus the respective test succeeds. Similarly to the cardinality constraint in Figure 5(b), limiting incoming edges to at most one per vertex, the requirement of a unique root addresses mutual exclusion. Hence, normalizations analogous to those in Figure 5(c)–(e) are applicable. Notably, the auxiliary atoms introduced in either of the latter normalizations can also be explored to replace the rule in line 15 by these:

```
child(X) :- node(X), not unique(X,X, 1).
child(X) :- node(X), not unique(X,X,-1).
```
(1)

```
child(X) :- node(X), not unique(X,1,1).
```
(2)

```
child(X) :- node(X), not unique(X,1,1,1).
```
(3)

In combination with the bidirectional traversal encoding in Figure 5(c), the rules in (1) conclude the existence of some edge to a vertex substituted for `X` from a predecessor that blocks proceeding "upwards" or "downwards" to `X` along the order of candidates. With the linear and tournament traversal approaches in Figure 5(d) and 5(e), vertex `X` must have some predecessor if the counter flag `1` remains underivable as an outcome of inspecting all predecessor candidates, which is checked in the prerequisite of the rule in (2) or (3), respectively. Either of the alternative derivations preserves the role of the predicate `child`/1 to indicate the vertices with some predecessor, so that the integrity constraint in line 16 of Figure 6(b) or corresponding normalizations can be used unmodified to test the uniqueness of a root.

Connectedness The encoding in Figure 6(c) checks that the undirected version of a (directed) graph $\langle V, E \rangle$ is connected. As a matter of fact, $|V| - 1$ edges are necessary to connect all vertices in V, so that connectness provides a criterion for testing whether

a forest is a tree. In order to inductively explore connected vertices, the rule in line 15 picks the vertex with label `1` (if $V \neq \emptyset$) as an (arbitrary) starting point to proceed in either direction along edges by applying the rules in line 16 and 17. As a consequence, the predicate `reach`/1 provides the vertices connected to `1`, and (ground instances of) the integrity constraint in line 18 require all vertices of a graph to be connected. Reconsidering the forest in Figure 1(c), disconnectedness is revealed by deriving `reach(1)` and `reach(2)` but lacking `reach(`v`)` for $v \in \{3, 4, 5\}$. As the undirected versions of the other three graphs depicted in Figure 1 are connected, such atoms `reach(`v`)` are derivable in addition from the instances of `edge`/2 characterizing these graphs, so that they are compatible with the integrity constraint in line 18. Like the inductive derivation of `order`/1 in Figure 3, the connectedness encoding in Figure 6(c) induces circular positive dependencies, and a step argument ranging from `1` to **n**, similar to the one added in Figure 4, can be introduced for obtaining a tight variant. Finally, note that the orthogonal tests encoded in Figure 6 are not mutually exclusive but may be freely combined in order to inspect several complementary properties of a directed tree at once.

3.4 Undirected Forests and Trees

When switching from directed to undirected edges $\{u, v\}$, several directed paths connecting two vertices yield an undirected cycle. Therefore, acyclic graphs coincide with forests in the undirected case. However, given that roots cannot (necessarily) be identified unambiguously when edges lack orientation, only leaves v satisfying $\deg(v) \leq 1$ can be distinguished. For instance, the undirected version (ignoring edge orientation) of the example graph in Figure 1(d) has the leaves 2, 4, and 5, whereas no unique root can be determined among the inner vertices 1 and 3. In what follows, we focus on the description of encoding modifications enabling a transition from directed to undirected graphs.

As edge candidates represented by the atoms `pair(`u`,`v`)` and `pair(`v`,`u`)` refer to the same set $\{u, v\}$ of distinct vertices, first of all, a canonical edge representation is established by replacing the rule in line 3 of Figure 2 as follows:

```
3  pair(X,Y) :- node(X;Y), X < Y.
```

In this way, redundant representations are resolved via labels of vertices, e.g., the atom `edge(1,2)`, but not `edge(2,1)`, may be generated by applying the (non-deterministic) choice rule in line 4 of Figure 2. Beyond that, checking that an undirected graph is acyclic and thus a forest can be accomplished by rearranging the approach of the bottom-up acyclicity test in Figure 3 to handling leaves of an undirected graph. The basic idea of the encoding in Figure 7 is to successively derive `order(`v`)` for vertices v that do not belong to any cycle, as witnessed by `order(`u`)` for all but (at most) one vertex u in an edge $\{u, v\}$. Conversely, when each remaining vertex v has two or more edges $\{u, v\}$ such that `order(`u`)` does not hold, there must be some cycle through v.

Apart from line 8, the rules in Figure 7 are similar to the directed acyclicity test in Figure 3. The additional case in line 6 takes care of asymmetric edge representation in deriving an instance of `order(X,Y)` for an edge candidate $\{$`X`$, $`Y`$\}$ when `order(X)` or `order(Y)` indicates the absence of cycles involving $\{$`X`$, $`Y`$\}$. Given **n**$-1$ edge candidates per vertex substituted for `X`, the rule in line 8 applies once the existence of a cycle through `X` can be safely excluded. Its ground instance for **n**$=5$ and `X`$=3$ is as follows:

```
5  order(X,Y) :- pair(X,Y), not edge(X,Y).
6  order(X,Y) :- pair(X,Y), order(X).
7  order(X,Y) :- pair(X,Y), order(Y).
8  order(X) :- node(X), n-2 { order(X,Y) : pair(X,Y),
                              order(Y,X) : pair(Y,X) }.
9  :- node(X), not order(X).
```

Fig. 7: Inductive encoding of undirected acyclicity test

```
order(3) :- 3 { order(1,3), order(2,3),
                order(3,4), order(3,5) }.
```

That is, when all but one among the edge candidates including vertex `3` do not belong to any cycle, then `3` cannot be part of a cycle either. Corresponding ground instances first allow for deriving `order(2)`, `order(4)`, and `order(5)` from instances of `edge`/2 characterizing the undirected version of the example graph in Figure 1(d). After concluding that the leaves along with edges including them do not contribute to any cycle, merely the atom `order(1,3)` remains open, so that `order(1)` and `order(3)` are derived in turn. In general, instances of `order`/1 are derivable for all vertices, as required in view of the integrity constraint in line 9, if and only if an undirected graph is acyclic.

Encoding Variants The cardinality constraint in line 8 of Figure 7 applies when all but one edge candidate including a vertex substituted for `X` cannot belong to any cycle. Given the uniqueness of a remaining edge candidate, normalizations analogous to those in Figure 5(c)–(e) can be used again. To this end, a rule indicating the absence of an edge for a candidate $\{\texttt{X}, \texttt{Y}\}$ is augmented with a second case based on `order(Y)`, expressing that there is no cycle through a vertex substituted for `Y`. In view of such positive prerequisites, neither the rule in line 8 of Figure 7 nor normalizations of this *non-tight* ASP formulation provide derivations of `order(`v_i`)` for vertices in a cycle $v_0, \ldots, v_k$, since each v_i is included in (distinct) edges connecting it to its predecessor and successor along the cycle. However, an additional step argument like the one in Figure 4 allows for a *tight* encoding, simulating well-founded derivations without relying on circular positive dependencies, where $\lceil \mathbf{n}/2 \rceil$ steps are sufficient in the undirected case. In fact, in any but the final step of such a construction, the graph obtained by eliminating edges that include leaves from previous steps must yield at least two new leaves, or there certainly is some cycle otherwise. Moreover, the *unconditional* choice rule in line 4 of Figure 2 can likewise be replaced by the following two rules:

```
{ edge(X,Y) } :- pair(X,Y), order(Y).
{ edge(X,Y) } :- pair(X,Y), order(X).
```

The prerequisite `order(Y)` or `order(X)`, respectively, directs the generation of edges to proceed successively from leaves. As with respective encoding variants addressing directed graphs, the effect of *conditional* edge generation is empirically assessed below.

Trees In order to distinguish undirected trees among forests, the approach to allow for one root only, available in the directed case, is no longer applicable. To see this, note that the undirected version of the tree depicted in Figure 1(d) has the inner vertices 1 and 3 along with the leaves 2, 4, and 5. However, when the edge $\{3, 4\}$ is removed, the graph becomes a disconnected forest with the same inner nodes and leaves. Hence, merely counting inner nodes or leaves does not provide sufficient information to identify trees

directed acyclic graphs	leaf encoding non-tight	leaf encoding tight	root encoding non-tight	root encoding tight
CLASP	0.26	—	0.27	—
CLASP/SAT	2.41	1.59	1.55	1.16
LINGELING	5.65	2.53	5.19	2.49
Z3	1.69	—	1.67	—
CPLEX	682.54	623.17	639.38	687.46

Table 1: Average runtimes for directed acyclic graphs

or disconnected forests, respectively. Unlike this, the fact that $|V|-1$ edges are required to connect the vertices of a forest $\langle V, E\rangle$ applies both in the directed and the undirected case. Therefore, *edge counting* or checking *connectedness* are adequate approaches to distinguish undirected trees, and the respective encodings in Figure 6(a) and 6(c) as well as variants thereof can be used unmodified for this purpose.

4 Evaluation

To test our encodings, we ran CLASP (2.1.3), both as an ASP and SAT solver, the SAT solver LINGELING (ats-57807c8-131016), the DL solver Z3 (4.3.1), and the LP solver CPLEX (12.5.0.0) on a cluster of Linux machines. We used GRINGO (3.0.5) to ground first-order ASP formulations and the translators LP2SAT2 (1.18), LP2DIFF (1.33), and LP2MIP (1.18) for converting ground instances to SAT, DL, or LP, respectively. Our experiments took synthetic as well as application benchmarks into account.[3]

4.1 Synthetic Benchmarks

In the first series of experiments, we perturbed the search space for graphs with desired properties by adding randomly generated XOR-constraints over edges, varying both their length and number. The tables presented in the following provide average runtimes in seconds for 25 instances with $\mathtt{n}{=}25$ vertices. Runs exceeding 1000 seconds were counted as 1000 seconds, and an entry "—" marks that no run completed in time.

Table 1 gives average runtimes for deciding whether there is a solution subject to XOR-constraints, comparing the performance of the considered solvers on four different encodings of directed acyclic graphs. The non-tight and tight variants of the *leaf encoding* consist of the rules shown in Figure 2 along with the inductive or unfolded bottom-up acyclicity test, respectively, encoded in Figure 3 and 4. Their counterparts denoted by *root encoding* swap the orientation of acyclicity tests by traversing vertices top-down from roots instead of bottom-up from leaves. The runtimes of CLASP on ASP formulations in the first row exhibit no significant differences between the symmetric leaf and root encodings. That is, instances relying on their non-tight variants were solved easily by CLASP (with default settings), whereas all runs on the tight variants timed out. Similar behavior is observed with the Z3 solver, for which non-tight instances are mapped to DL and tight ones to plain SAT. However, LINGELING and CLASP, run as a SAT solver, both perform well on the translations by LP2SAT2, where the tight ASP formulations have some advantage over the level mappings introduced when translating non-tight instances. The performance of the LP solver CPLEX (with default settings) is rather unstable on all of our encoding variants. In particular, CPLEX does not seem to profit from integer variables to represent level mappings for non-tight instances.

[3] `http://research.ics.aalto.fi/software/asp/bench/treebm.tgz`

directed forests || trees	leaf encoding non-tight	leaf encoding tight	root encoding non-tight	root encoding tight	leaf encoding non-tight	leaf encoding tight	root encoding non-tight	root encoding tight
pairwise	10.38	13.30	9.53	12.03	8.78	184.16	12.31	14.12
cardinality	5.33	255.51	5.53	7.83	16.11	169.38	11.52	25.14
bidirectional	8.86	212.12	8.01	8.41	7.87	248.22	6.77	9.21
linear	3.86	365.09	4.48	6.57	5.33	221.69	7.08	8.02
tournament	6.39	256.61	6.62	8.99	7.14	219.01	7.48	6.44
			counting		251.44	290.33	252.14	50.17
			counting/ub		252.15	296.74	251.43	8.48
			connectedness		6.63	58.74	8.47	18.09

Table 2: Average runtimes for directed forests and trees

Average runtimes of CLASP on ASP formulations of directed forests are shown on the left-hand side of Table 2. We do not apply translators here because of their nonuniform treatment of cardinality constraints used in some encoding variants, whereas CLASP handles cardinality constraints in ground instances of ASP rules natively. The first five columns provide runtime results on the four previously considered encodings of directed acyclic graphs augmented with either set of rules in Figure 5(a)–(e) for limiting predecessors to at most one per vertex. Although switching to the more restrictive notion of directed forests reduces the admissible outcomes and makes the search for a solution harder, CLASP performs well on all non-tight encoding variants. However, except for pairwise mutual exclusion encoded in Figure 5(a), the remaining approaches still exhibit substantial difficulties with the tight variant of the leaf encoding to test acyclicity in a bottom-up fashion. Interestingly, such problems disappear with the root encoding swapping the orientation to top-down traversals. While this observation is certainly influenced by search heuristics of CLASP, it nonetheless demonstrates that subtle encoding details can have a significant impact. In the context of directed forests, traversals starting from roots rather than leaves may be more informative since the number of roots can never exceed that of leaves. In view of interactions or interferences between encoding parts and search heuristics, whether a particular modification is advantageous is still hardly predictable in general, so that systematic empirical investigations seem indispensable for fine-tuning encodings. Among the considered approaches to limit predecessors to at most one per vertex, the linear traversal encoded in Figure 5(d) tends to yield the best performance of CLASP. Only on the tight leaf encoding, checking mutual exclusion by explicitly enumerating pairs of distinct predecessor candidates stands out, even though the (cubic) space complexity of this approach is a bottleneck for scalability.

The right-hand side of Table 2 provides average runtimes for the encoding approaches in Figure 6(a)–(c) to distinguish directed trees among forests. Unless noted otherwise, the investigated encoding variants rely on the cardinality constraint in Figure 5(b) to rule out directed acyclic graphs that are no forests. Although the approach of *counting* over all edge candidates encoded in Figure 6(a) could be expected to yield poor performance, the average runtimes of CLASP are surprisingly short on the tight root encoding. In that context, the addition of a second cardinality constraint "ub" to redundantly assert $|V| - 1$ as upper bound on the number of edges of a directed tree $\langle V, E \rangle$ turns out to be helpful too. However, checking the *uniqueness* of a root by means of the same encoding approaches as presented for forests in Figure 5 leads to much more robust performance. While mutual exclusion via the enumeration of pairs or a cardinality constraint, respectively, rely on the rule in line 15 of Figure 6(b) to derive edge targets, we used its dedicated variants in (1), (2), and (3) for the bidirectional, lin-

undirected forests || trees	**unconditional edges** non-tight	tight	**conditional edges** non-tight	tight	**edge counting** non-tight	tight	**connectedness** non-tight	tight
cardinality	781.77	707.37	762.71	617.64	924.50	820.39	801.23	854.45
bidirectional	761.62	812.74	693.47	840.39	885.49	992.43	810.68	924.88
linear	800.16	806.34	802.25	831.39	960.59	809.99	804.17	820.35
tournament	800.02	760.77	802.18	972.76	970.47	874.31	801.24	911.62
			cardinality/ub		735.91	768.09	803.34	826.52
			bidirectional/ub		913.29	908.34	912.84	964.84
			linear/ub		—	791.52	847.79	819.73
			tournament/ub		803.75	689.36	826.86	829.45

Table 3: Average runtimes for undirected forests and trees

ear, and tournament traversal approaches. Apart from the tight leaf encoding on which CLASP struggles, the three normalizations along with the reuse of their auxiliary atoms yield some improvements in comparison to the first two approaches. The alternative of checking the *connectedness* of a directed forest, as encoded in Figure 6(c) or unfolded to a tight variant, respectively, also exhibits a robust performance of CLASP and even compensates difficulties on the tight leaf encoding to some extent. In summary, it appears promising to encode the properties of directed trees in terms of a limitation to (at most) one root node and/or connectedness, where manifold combinations with underlying encoding parts aiming at forests can in principle be conceived.

Turning from the directed case to undirected graphs, the left-hand side of Table 3 shows average runtimes of CLASP on several variants of the encoding in Figure 7. Since there is no apparent way to perform an acyclicity test top-down from roots or inner nodes, respectively, encoding variants are, for one, obtained by replacing the cardinality constraint in line 8 of Figure 7 by normalizations analogous to those in Figure 5(c)–(e). For another, we focus on *conditional edge generation* via choice rules requiring the absence of cycles through a vertex as a prerequisite for the addition of an edge including the vertex. The comparably long runtimes point at increased difficulty of search for a forest when edges lack orientation, and non-tight as well as tight encoding variants yield timeouts on 15 or more of the 25 instances. While the bidirectional traversal approach to check whether (at most) one edge including a vertex remains as yet unchecked to not belong to any cycle leads to the best performance of CLASP on non-tight instances, the use of cardinality constraints dominates on tight ones. Somewhat unexpectedly, the shortest average runtime is observed with the tight acyclicity test based on cardinality constraints. Moreover, both in the non-tight and the tight case, shortest runtimes are obtained with conditional edge generation, where the prerequisites of choice rules formulate the acyclicity requirement in a redundant fashion. However, the impact of such redundancy on solving performance does not exhibit a clear trend, and thus we do not additionally report respective runtime results for other graph structures.

The right-hand side of Table 3 provides average runtimes on encodings of undirected trees, combining the previously considered acyclicity test approaches with *edge counting* and/or *connectedness* checking, as encoded in Figure 6(a) and 6(c). Checking for the existence of (at least) $|V| - 1$ edges of a tree $\langle V, E \rangle$ tends to work better with tight than non-tight encoding variants to test acyclicity, where the addition of a redundant cardinality constraint "ub" asserting $|V| - 1$ as an upper bound pays off as well. In the absence of the latter, checking whether a forest is connected still yields more robust performance in combination with non-tight approaches to test acyclicity. Beyond that, we investigated hybrid variants incorporating cardinality constraints asserting the

Bayesian network	leaf encoding		root encoding		leaf encoding		root encoding	
CLASP || CPLEX	non-tight	tight	non-tight	tight	non-tight	tight	non-tight	tight
asia 1000	0.01	0.02	0.01	0.03	0.80	8.97	0.79	8.81
asia 10000	0.08	0.11	0.09	0.14	2.69	24.01	2.02	54.35
hailfinder 100	1126.24	—	981.86	—	1.17	—	1.10	—
insurance 100	20.56	83.94	11.96	117.75	42.27	—	68.02	—
mildew 100	0.43	2.86	0.35	711.70	26.35	—	8.87	—
mildew 1000	0.94	6.27	0.67	33.06	0.63	2407.47	0.64	3296.95
water 100	387.78	—	597.15	—	179.06	—	215.45	—
water 1000	2269.31	—	2593.09	—	533.11	—	2154.65	—

Table 4: Runtimes for Bayesian network structure learning

existence of $|V|-1$ edges as well as an encoding part for checking connectedness. The respective runtimes of CLASP in the lower right quarter, however, indicate performance declines due to crossing orthogonal formulations of the same property.

4.2 Application Benchmarks

Our second series of experiments deals with the application problem of Bayesian network structure learning [9, 10], which can be expressed in terms of optimization relative to ASP, SAT, or LP formulations. Given a set V of vertices, directed edge candidates C, and scores $s(v,P)$ for vertices $v \in V$ and parent sets $P \subseteq \{u \mid \langle u,v\rangle \in C\}$, the task consists of finding a directed acyclic graph $\langle V,E\rangle$ such that $E \subseteq C$ and $\sum_{v\in V,P=\{u\mid\langle u,v\rangle\in E\}} s(v,P)$ is maximal.

Among the considered solvers, CLASP and CPLEX support optimization, and Table 4 shows their runtimes on eight Bayesian network structure learning instances along with encodings varying the acyclicity test for directed graphs as in Table 1. In view of the increased hardness of performing optimization, we extended the time limit to 3600 seconds per run. The runtimes of CLASP on the left-hand and of CPLEX on the right-hand side indicate significant advantages of using non-tight encoding variants. Unlike with the synthetic benchmarks investigated above, CPLEX here seems to benefit from integer variables introduced by the translator LP2MIP to represent level mappings. Moreover, CLASP and CPLEX both tend to perform better on the leaf than on the root encoding variants, where differences are application-specific in view of the full symmetry of both approaches to test acyclicity. Although ASP formulations are automatically translated to LP, CPLEX yields comparable and on the hardest instances even better performance than CLASP when using non-tight encoding variants. This observation points out the potential behind translational approaches to solve ASP formulations.

5 Conclusions

Graphs satisfying acyclicity properties are frequent in knowledge representation tasks. As these properties do not appear as basic primitives in common constraint-based representation formalisms, formulating them compactly and efficiently is a recurring challenge. We investigated logic-based characterizations of acyclicity conditions and developed a systematic collection of encodings in the language of answer set programming.

While the size of ground instances is linear for the acyclicity test in Figure 3, encodings based on the full transitive closure of edges [4–6, 3] yield quadratic space complexity, even for sparse graphs. Furthermore, the LP formulations of acyclicity in [15, 10] require exponential space, whereas ours are proportional to graph size. Also note that the encoding variants for mutual exclusion testing in Figure 5 resemble diverse approaches from SAT to formulate cardinality constraints [22, 17, 19, 12].

Experiments on synthetic as well as application benchmarks indicate the relevance of representation approaches and constraint formulations, with particular emphasis on well-foundedness and cardinality constraints. As the discovery of a single "universal" encoding is highly unlikely, the alternatives investigated here provide a toolkit for representing graph structures in different applications. In fact, we believe that effective declarative means to specify such fundamental datatypes are important cornerstones for the development of robust constraint-based solving methods.

References

1. J. Bomanson and T. Janhunen. Normalizing cardinality rules using merging and sorting constructions. In *Proc. LPNMR'13*, pages 187–199, Springer, 2013.
2. M. Bonet and K. John. Efficiently calculating evolutionary tree measures using SAT. In *Proc. SAT'09*, pages 4–17, Springer, 2009.
3. G. Brewka, T. Eiter, and M. Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.
4. D. Brooks, E. Erdem, S. Erdogan, J. Minett, and D. Ringe. Inferring phylogenetic trees using answer set programming. *Journal of Automated Reasoning*, 39(4):471–511, 2007.
5. M. Çaylı, A. Karatop, E. Kavlak, H. Kaynar, F. Türe, and E. Erdem. Solving challenging grid puzzles with answer set programming. In *Proc. ASP'07*, pages 175–190, 2007.
6. M. Çelik, H. Erdoğan, F. Tahaoğlu, T. Uras, and E. Erdem. Comparing ASP and CP on four grid puzzles. In *Proc. RCRA'09*, CEUR, 2009.
7. K. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, Plenum Press, 1978.
8. J. Corander, T. Janhunen, J. Rintanen, H. Nyman, and J. Pensar. Learning chordal Markov networks by constraint satisfaction. In *Proc. NIPS'13*, pages 1349–1357, 2013.
9. James Cussens. Bayesian network learning by compiling to weighted MAX-SAT. In *Proc. UAI'08*, pages 105–112, AUAI Press, 2008.
10. James Cussens. Bayesian network learning with cutting planes. In *Proc. UAI'11*, pages 153–160, AUAI Press, 2011.
11. E. Erdem and V. Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3(4-5):499–518, 2003.
12. A. Frisch and P. Giannoros. SAT encodings of the at-most-k constraint. In *Proc. ModRef'10*, 2010.
13. M. Gebser, T. Janhunen, and J. Rintanen. ASP encodings of acyclicity properties. In *Proc. KR'14*, AAAI Press, 2014.
14. M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practice*. Morgan and Claypool Publishers, 2012.
15. T. Jaakkola, D. Sontag, A. Globerson, and M. Meila. Learning Bayesian network structure using LP relaxations. In *Proc. AISTATS'10*, JMLR.org, 2010.
16. T. Janhunen. Representing normal programs with clauses. In *Proc. ECAI'04*, pages 358–362, IOS Press, 2004.
17. W. Klieber and G. Kwon. Efficient CNF encoding for selecting 1 from n objects. In *Proc. CFV'07*, 2007.
18. G. Liu, T. Janhunen, and I. Niemelä. Answer set programming via mixed integer programming. In *Proc. KR'12*, pages 32–42, AAAI Press, 2012.
19. J. Marques-Silva and I. Lynce. Towards robust CNF encodings of cardinality constraints. In *Proc. CP'07*, pages 483–497, Springer, 2007.
20. M. Nguyen, T. Janhunen, and I. Niemelä. Translating answer-set programs into bit-vector logic. In *Proc. INAP/WLP'11*, pages 105–116, Springer, 2013.
21. I. Niemelä. Stable models and difference logic. *Annals of Mathematics and Artificial Intelligence*, 53(1-4):313–329, 2008.
22. C. Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In *Proc. CP'05*, pages 827–831, Springer, 2005.