ASP Encodings of Acyclicity Properties

Martin Gebser[*†] and **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 properties is non-obvious and can be challenging indeed. In this paper, we take acyclicity properties into consideration and investigate 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.

Introduction

Numerous hard computational tasks involve the construction of acyclic or tree structures. Constraint satisfaction and related methods are an important approach for solving many of these problems. 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.

Construction of acyclic graphs, trees, or chordal graphs shows up in numerous applications. For instance, Bayesian network structure learning, where directed acyclic graphs provide solution candidates, can be reduced to constraint optimization (Jaakkola et al. 2010; Cussens 2011). Furthermore, chordal Markov network learning amounts to the task of optimizing maximum weight spanning trees induced by chordal graphs (Corander et al. 2013). Chordality is a relaxation of strict acyclicity in which cycles of length three are allowed in an otherwise tree-structured undirected graph. Constraint-based methods can also be used to infer phylogenetic trees (Brooks et al. 2007; Bonet and John 2009), describing the evolution of living organisms, languages, and other evolving systems.

The basic problem to be solved with constraint-based formalisms in the above and other applications is constructing an acyclic or a tree structure subject to further conditions. That the choice among underlying edge candidates actually results in some graph structure with the desired properties must be enforced by corresponding constraints. Straightforward formulations, however, can become impractically large and inefficient, and our work sheds light on ways to encode such constraints succinctly and efficiently.

Although we rely on answer set programming (ASP) (Brewka, Eiter, and Truszczyński 2011) as the primary representation language for encodings, respective formulations in related formalisms, such as propositional satisfiability (SAT), SAT extended with 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* (Erdem and Lifschitz 2003), i.e., if there are no circular positive dependencies through rules' prerequisites on the ground level. In the non-tight case, level mappings (Janhunen 2004) 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 (Niemelä 2008) or integer variables in LP (Liu, Janhunen, and Niemelä 2012).

In this short paper, we present encoding approaches for directed acyclic graphs. Their extensions to forests and trees, in the directed and undirected case, as well as chordal graphs are addressed in an extended version of this paper.[1]

Directed Graphs

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 $v_1, \dots, v_k$ 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 $v_0, v_1, \dots, v_k$ is a *cycle* of length k in G if $\langle v_0, v_1\rangle \in E$, $v_0 = v_k$, and $v_1, \dots, v_k$ 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 $v_1, \dots, v_k$ in G from a root v_1 of G to v_k. Given

[*]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.

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

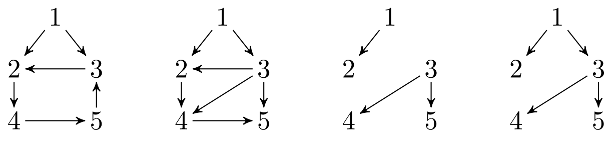
(a) Cyclic (b) Acyclic (c) Forest (d) Tree

Figure 1: Directed example graphs with five vertices each

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

Figure 2: Encoding part for generating directed graphs

that any incoming edge $\langle v_{k-1}, v_k \rangle$ can be extended to a path $v_1, \dots, v_{k-1}, v_k$ 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.

Encodings

We illustrate different encodings by means of first-order specifications in the input language of the ASP grounder GRINGO (Gebser et al. 2012). The underlying principles, however, are of general applicability, and we outline particularities of respective SAT, DL, and LP formulations.

Rules describing the (non-deterministic) generation of directed graphs are shown in Figure 2. The predicate `node`/1 provides the labels `1`, ..., **`n`** for vertices, where **`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 instance, 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. Given **`n`**=5, the following (simplified) ground instances are obtained when the label `3` is substituted for `X`:

```
order(3,1) :- not edge(3,1).
order(3,1) :- order(1).
order(3,2) :- not edge(3,2).
order(3,2) :- order(2).
order(3,4) :- not edge(3,4).
order(3,4) :- order(4).
order(3,5) :- not edge(3,5).
order(3,5) :- order(5).
```

```
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).
```

Figure 3: Inductive bottom-up encoding of acyclicity test

```
order(3)    :- order(3,1), order(3,2),
               order(3,4), order(3,5).
:- not order(3).
```

As expressed by `order(3,`v`)`, vertex `3` fulfills the condition of $<$ relative to a potential successor $v \in \{1, 2, 4, 5\}$ if there is no edge from `3` to v or if v has no path to `3`. Then, `order(3)` is derived once the existence of a cycle through vertex `3` can be safely excluded for all potential successors v of `3`. In particular, the positive prerequisites of corresponding ground rules for other vertices along with well-foundedness of derivations, as required in ASP, prohibit `order(3)` to hold if `3` belongs to some cycle. Given this, the integrity constraint denying models such that `order(3)` is false rejects any directed graph with a cycle through `3`, and respective ground rules establish the same for other vertices.

Reconsidering the example graph in Figure 1(a), we have that `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 are compatible 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. (Janhunen 2004; Niemelä 2008; Liu, Janhunen, and Niemelä 2012)) furnish an alternative mechanism to express well-founded derivations, where difference constraints or integer variables allow for linear embeddings 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 in-

```
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), N = 2..n,
                order(X,Y,N) : pair(X,Y).
8 order(X)   :- node(X),
                order(X,Y,1) : pair(X,Y).
```

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

stance, 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 (Clark 1978) 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 (Corander et al. 2013). 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 (non-deterministic) 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 the evaluation section below. 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 extended contexts the orientation of acyclicity tests may interact or interfere.

Notably, the predicate definitions in Figure 3 and 4 are both *stratified* (cf. (Apt, Blair, and Walker 1988)). That is, when instances of the predicate `edge`/2 are fixed, there is at most one (well-founded) model that can be deterministically determined. In the absence of circular positive dependencies, as with ground instances of the rules in Figure 4, the unique model or unsatisfiability is also obtained by evaluating (via unit propagation) the rules' completion. Except for variants introducing prerequisites for conditional edge generation, all encoding parts in the sequel are stratified, so that the choice of edges is the only source of non-determinism.

directed acyclic graphs	**leaf encoding** **non-tight**	**tight**	**root encoding** **non-tight**	**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

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) with default settings on a cluster of Linux machines. We used GRINGO (3.0.5) to ground first-order ASP formulations and the translators LP2SAT2 (1.18) (Janhunen and Niemelä 2011), LP2DIFF (1.33) (Janhunen, Niemelä, and Sevalnev 2009), and LP2MIP (1.18) (Liu, Janhunen, and Niemelä 2012) for converting ground instances to SAT, DL, or LP, respectively. In order to perturb the search space for directed acyclic graphs, we added randomly generated XOR-constraints over edges, systematically varying both the length and the number of such constraints.

Average runtimes in seconds for 25 instances with **n**=25 vertices are given in Table 1. Runs exceeding 1000 seconds were counted as 1000 seconds, and an entry "—" marks that no run completed in time. We compare 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 substantially benefit from the use of integer variables to represent level mappings for non-tight instances.

Conclusions

Graphs satisfying acyclicity properties are frequent in knowledge representation tasks. Since these properties do not appear as basic primitives in common constraint-based representation formalisms, formulating them compactly and efficiently is a recurring challenge. We have investigated logic-based characterizations of directed acyclic graphs and developed encodings in the language of answer set programming. Directed forests and trees, their undirected variants, as well as chordal graphs are addressed in an extended draft.[1]

Experiments indicate the relevance of representation approaches and constraint formulations, with particular emphasis on well-foundedness. While the size of ground instances is linear for the acyclicity test in Figure 3, encodings based on the full transitive closure of edges (Rintanen, Heljanko, and Niemelä 2004; Brooks et al. 2007; Çaylı et al. 2007; Cussens 2008; Çelik et al. 2009; Brewka, Eiter, and Truszczyński 2011) yield quadratic space complexity, even for sparse graphs. 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

Apt, K.; Blair, H.; and Walker, A. 1988. Towards a theory of declarative knowledge. In Minker, J., ed., *Foundations of Deductive Databases and Logic Programming*, 89–148. Morgan Kaufmann Publishers.

Bonet, M., and John, K. 2009. Efficiently calculating evolutionary tree measures using SAT. In Kullmann, O., ed., *Proceedings of the Twelfth International Conference on Theory and Applications of Satisfiability Testing (SAT'09)*, volume 5584 of *Lecture Notes in Computer Science*, 4–17. Springer.

Brewka, G.; Eiter, T.; and Truszczyński, M. 2011. Answer set programming at a glance. *Communications of the ACM* 54(12):92–103.

Brooks, D.; Erdem, E.; Erdogan, S.; Minett, J.; and Ringe, D. 2007. Inferring phylogenetic trees using answer set programming. *Journal of Automated Reasoning* 39(4):471–511.

Çaylı, M.; Karatop, A.; Kavlak, E.; Kaynar, H.; Türe, F.; and Erdem, E. 2007. Solving challenging grid puzzles with answer set programming. In Costantini, S., and Watson, R., eds., *Proceedings of the Fourth International Workshop on Answer Set Programming (ASP'07)*, 175–190.

Çelik, M.; Erdoğan, H.; Tahaoğlu, F.; Uras, T.; and Erdem, E. 2009. Comparing ASP and CP on four grid puzzles. In Gavanelli, M., and Mancini, T., eds., *Proceedings of the Sixteenth RCRA International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion (RCRA'09)*. CEUR Workshop Proceedings.

Clark, K. 1978. Negation as failure. In Gallaire, H., and Minker, J., eds., *Logic and Data Bases*, 293–322. Plenum Press.

Corander, J.; Janhunen, T.; Rintanen, J.; Nyman, H.; and Pensar, J. 2013. Learning chordal Markov networks by constraint satisfaction. In Burges, C.; Bottou, L.; Ghahramani, Z.; and Weinberger, K., eds., *Proceedings of the Twenty-seventh Annual Conference on Neural Information Processing Systems (NIPS'13)*, volume 26 of *Advances in Neural Information Processing Systems*, 1349–1357.

Cussens, J. 2008. Bayesian network learning by compiling to weighted MAX-SAT. In McAllester, D., and Myllymäki, P., eds., *Proceedings of the Twenty-fourth International Conference on Uncertainty in Artificial Intelligence (UAI'08)*, 105–112. AUAI Press.

Cussens, J. 2011. Bayesian network learning with cutting planes. In Cozman, F., and Pfeffer, A., eds., *Proceedings of the Twenty-seventh International Conference on Uncertainty in Artificial Intelligence (UAI'11)*, 153–160. AUAI Press.

Erdem, E., and Lifschitz, V. 2003. Tight logic programs. *Theory and Practice of Logic Programming* 3(4-5):499–518.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers.

Jaakkola, T.; Sontag, D.; Globerson, A.; and Meila, M. 2010. Learning Bayesian network structure using LP relaxations. In Teh, Y., and Titterington, D., eds., *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS'10)*, volume 9 of *JMLR Proceedings*, 358–365. JMLR.org.

Janhunen, T., and Niemelä, I. 2011. Compact translations of non-disjunctive answer set programs to propositional clauses. In Balduccini, M., and Son, T., eds., *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, volume 6565 of *Lecture Notes in Computer Science*, 111–130. Springer.

Janhunen, T.; Niemelä, I.; and Sevalnev, M. 2009. Computing stable models via reductions to difference logic. In Erdem, E.; Lin, F.; and Schaub, T., eds., *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *Lecture Notes in Artificial Intelligence*, 142–154. Springer.

Janhunen, T. 2004. Representing normal programs with clauses. In López de Mántaras, R., and Saitta, L., eds., *Proceedings of the Sixteenth European Conference on Artificial Intelligence (ECAI'04)*, 358–362. IOS Press.

Liu, G.; Janhunen, T.; and Niemelä, I. 2012. Answer set programming via mixed integer programming. In Brewka, G.; Eiter, T.; and McIlraith, S., eds., *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'12)*, 32–42. AAAI Press.

Niemelä, I. 2008. Stable models and difference logic. *Annals of Mathematics and Artificial Intelligence* 53(1-4):313–329.

Rintanen, J.; Heljanko, K.; and Niemelä, I. 2004. Parallel encodings of classical planning as satisfiability. In Alferes, J., and Leite, J., eds., *Proceedings of the Ninth European Conference on Logics in Artificial Intelligence (JELIA'04)*, volume 3229 of *Lecture Notes in Computer Science*, 307–319. Springer.