SAT modulo Graphs: Acyclicity

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. Acyclicity is a recurring property of solutions to many important combinatorial problems. In this work we study embeddings of specialized acyclicity constraints in the satisfiability problem of the classical propositional logic (SAT). We propose an embedding of directed graphs in SAT, with arcs labelled with propositional variables, and an extended SAT problem in which all clauses have to be satisfied and the subgraph consisting of arcs labelled *true* is acyclic. We devise a constraint propagator for the acyclicity constraint and show how it can be incorporated in off-the-shelf SAT solvers. We show that all existing encodings of acyclicity constraints in SAT are either prohibitively large or do not sanction all inferences made by the constraint propagator. Our experiments demonstrate the advantages of our solver over other approaches for handling acyclicity.

1 Introduction

SAT, the satisfiability problem of the propositional logic, has emerged as a powerful framework for solving combinatorial problems in AI and other areas of computer science. For many applications the basic SAT problem is sufficient, including AI planning and related state-space search problems [13], but a number of important applications involves the expression of constraints that cannot be effectively encoded as sets of clauses. For this reason, various extensions of SAT have been proposed, including SAT + linear arithmetics in the SAT modulo Theories (SMT) framework [25, 1]. Other instantiations of the SMT framework are possible, including bit vectors and arrays.

In this work, we consider extensions of SAT with graphs, initially focusing on satisfying an acyclicity constraint. Examples of combinatorial problems that involve acyclic graphs can be found in diverse areas. In machine learning, the structure learning problem for Bayesian networks is reducible to a MAXSAT problem and a main part of the reduction is about guaranteeing the acyclicity of resulting networks [4]. Acyclicity is implicit in inductive definitions (well-foundedness), and SAT solvers with efficient support for acyclicity constraints could be used in reasoning with logical languages that support inductive definitions [5]. Another closely related application is answer set programming, with which we have already experimented by using the technology presented in this paper [8]. Reasoning with physical networked systems – such as utility networks (power, water, communications) and transportation networks – often involves acyclicity and other graph constraints.

[⋆] Also affiliated with the University of Potsdam, Germany.
[⋆⋆] Also affiliated with Griffith University, Brisbane, Australia.

Reductions of acyclicity constraints to CNF SAT are known [4, 21], but, as we will show later, they either have prohibitively large size or sanction weak inferences. This motivates looking into specialized propagators for acyclicity. We believe the same to hold for many other graph problems, suggesting a wider framework of *SAT modulo Graphs* which could be viewed as an instantiation of the SMT framework. In comparison to many SMT theories, reasoning with many graph properties has a low overhead, enabling a tighter integration with SAT solvers than what is generally possible. For example, it turns out that running full acyclicity tests in every search node of a SAT solver is in general very feasible. We use ACYC-SAT to denote our SAT modulo Graphs framework instantiated with the acyclicity constraint.

Acyclicity constraints are expressible in SMT with linear arithmetics and, in particular, the fragment known as *difference logic* [15], which extends propositional logic by simple difference constraints of the form $x - y \geq k$. However, when the full generality of such constraints is not needed, the approach proposed in this paper can provide a simpler and more efficient reasoning framework. Moreover, we anticipate future extensions of the framework to cover other types of graph-related constraints that are not naturally and efficiently expressible in SMT with linear arithmetics.

Our research makes a number of new contributions. First, we present a propagator for the acyclicity constraint and propose an implementation inside the CDCL algorithm for SAT. We believe that the simplicity and efficiency of this propagator provides a significant advantage over alternative ways of reasoning with acyclicity. Our experiments will illustrate that substantial performance advantage can be gained this way. Second, our contribution can be viewed as initiating the study of graph-based constraints in the SMT framework [25, 1]. Earlier, SMT has been used most notably with theories for linear arithmetic and bit vectors. Concepts related to graphs, although important in many applications, have not been offered specialized support in the SMT framework.

The structure of the paper is as follows. First in Section 2 we propose an extension of the SAT problem for handling graphs and acyclicity constraints for them. In Section 3 we give some examples of the use of acyclicity constraints. Section 4 shows how a leading algorithm for solving the SAT problem can be extended with the acyclicity constraints. In Section 5 we present and evaluate alternatives to specialized acyclicity constraints, which is reduction of acyclicity to sets of clauses. Section 6 characterizes SAT modulo acyclicity in terms of a fragment of difference logic. In Section 7 we show that our implementation of acyclicity constraints generally and sometimes dramatically outperforms alternative approaches, including CNF encodings and difference logic solvers. Section 8 discusses related work and Section 9 concludes the paper.

2 Extending SAT with Acyclicity

We propose an extension of the standard SAT problem with acyclicity constraints. In addition to a set of clauses, the extended satisfiability problem includes a directed graph and a mapping from the arcs of the graph to propositional variables. A problem instance is satisfied if all clauses are satisfied, and there is no cycle in the graph such that for every arc in the cycle the corresponding propositional variable is true.

Definition 1. *An ACYC-SAT problem is a tuple* $\langle X, C, N, A, l \rangle$ *where*

1. *X is a finite set of propositional variables,*
2. *C is a set of clauses over X,*
3. *$G = \langle N, A \rangle$ is a directed graph with a finite set of nodes N and arcs $A \subseteq N \times N$, and*
4. *$l : A \to X$ is a labeling that assigns a propositional variable $l(n, n')$ to every arc (n, n') in the graph.*

Definition 2. *A solution to an ACYC-SAT problem $\langle X, C, N, A, l \rangle$ is a valuation $v : X \to \{0, 1\}$ such that all clauses in C are true under v, and the subgraph $\langle N, A_1 \rangle$ of $\langle N, A \rangle$ such that $A_1 = \{(n, n') \in A \mid v(l(n, n')) = 1\}$ is acyclic, that is, there is no non-empty directed path in $\langle N, A_1 \rangle$ from any node back to itself.*

A number of other graph problems could be handled in the same framework, in some cases with small modifications to our algorithms, including s-t-reachability, a node being on a simple path between two other nodes, connectivity, and so on. These and other graph properties show up in main applications of SAT solving, including verification (model-checking), control and diagnosis of networked systems. Our experiments suggest that explicit support for them may substantially improve the effectiveness of SAT-based methods in these applications.

3 Examples

Acyclicity shows up explicitly or implicitly in many types of graph problems. As shown in Section 5, best known encodings of acyclicity as clausal constraints in SAT have a trade-off between size and propagation strength. Next we illustrate how acyclicity can benefit also encodings of problems that do not explicitly appeal to acyclicity.

Example 1 (Hamiltonian cycles). Encoding of Hamiltonian cycles for directed graphs uses propositional variables for every arc, marking the arc either *true* or *false*, and consists of the following constraints, which we will explain in more detail below.

– Every node has exactly one incoming arc.
– Every node has exactly one outgoing arc.
– There is no cycle in the graph formed by all incoming arcs of all nodes except one node (an arbitrarily chosen “starting node”).

Given a directed graph $\langle N, A \rangle$ and a starting node $n_s \in N$, let a^n be propositional variables for arcs $(n, n_s) \in A$, expressing that (n, n_s) belongs to a Hamiltonian cycle, while arc variables $a^{n,n'}$ represent the same for arcs $(n, n') \in A$ to the remaining nodes $n' \in N \setminus \{n_s\}$. Then, given any arc $(n, n') \in A$, we use the following notation for the propositional variable expressing Hamiltonian cycle containment.

$$\alpha^{n,n'} = \begin{cases} a^n & \text{if} \quad n' = n_s \\ a^{n,n'} & \text{if} \quad n' \neq n_s \end{cases}$$

Moreover, for any node $y \in N$, let $x_1, \ldots, x_m$ and $z_1, \ldots, z_n$ denote the nodes x_i or z_j such that $(x_i, y) \in A$ or $(y, z_j) \in A$, respectively. Using the auxiliary propositions

$p^{y,0}, \dots, p^{y,m}$ and $q^{y,0}, \dots, q^{y,n}$ to represent that some incoming arc $(x_{i'}, y)$ or some outgoing arc $(y, z_{j'})$ with $i' \leq i$ or $j' \leq j$ belongs to a Hamiltonian cycle, the following formulas state that exactly one incoming and one outgoing arc must be picked for y.

$$\begin{array}{cccc} \neg p^{y,0} & \neg p^{y,i-1} \vee \neg \alpha^{x_i,y} & p^{y,i-1} \vee \alpha^{x_i,y} \leftrightarrow p^{y,i} & p^{y,m} \\ \neg q^{y,0} & \neg q^{y,j-1} \vee \neg \alpha^{y,z_j} & q^{y,j-1} \vee \alpha^{y,z_j} \leftrightarrow q^{y,j} & q^{y,n} \end{array}$$

Given such formulas for each node $y \in N$, models M such that the graph $\langle N, \{(n, n') \mid a^{n,n'} \in M\}\rangle$ is acyclic correspond to Hamiltonian cycles. □

The size of the above encoding is linear in the number of both arcs and nodes.

Example 2 (s-t reachability). Our encoding of s-t reachability for undirected graphs uses propositional variables for marking each node as reachable or unreachable, and arc variables $a^{n,n'}$ as well as $a^{n',n}$ for every edge $\{n, n'\}$ in the graph. The encoding uses the following constraints.

– The starting node n_s is *reachable*.
– A node n' is *reachable* if and only if it is the starting node or there is another node n so that the variable $a^{n,n'}$ for the arc $n \to n'$ is true and n is *reachable*.
– The graph corresponding to the arc variables is acyclic.

Given an undirected graph $\langle N, E\rangle$, this approach can be formulated in terms of the clauses $r^{n'} \to \bigvee_{\{n,n'\} \in E} a^{n,n'}$ for each node $n' \in N \setminus \{n_s\}$, two clauses $a^{n,n'} \to r^n$ and $a^{n',n} \to r^{n'}$ per edge $\{n, n'\} \in E$, and the unit clause r^{n_t} asserting that the target node n_t must be reached. By requiring $G = \langle N, \{(n, n') \mid a^{n,n'} \in M\}\rangle$ to be acyclic for a model M, any path in G must trace back to the starting node n_s. In particular, this applies for the mandatory path to n_t. □

Also this encoding is linear in the number of both edges and nodes. In many practical problems the degrees of nodes are bounded, small, or grow far slower than the number of nodes. In all of these cases the specialized acyclicity constraint leads to far smaller encodings than the use of clausal encodings (Section 5). Both encodings are easily adaptable to both directed and undirected graphs.

4 Acyclicity in SAT Solvers

The conflict-driven clause learning algorithm (CDCL) [16, 18, 17] is a leading systematic general-purpose algorithm for solving the SAT problem. The algorithm assigns truth-values to propositional variables, interleaved with calls to a propagator (inference rules), until an inconsistency (the empty clause) is inferred. The reasons for the inconsistency are analyzed, and a clause representing the assignments that led to the inconsistency is computed and added to the clause database (learned). Then the latest assignments are undone until the newly learned clause has exactly one unassigned variable, and the process of interleaved propagation and assignments resumes. Unsatisfiability is reported when learning the empty clause, and satisfiability is reported when all variables get assigned without obtaining a contradiction. The main propagation rule

```
1:  PROCEDURE propagator($x$);
2:  let $n_s \to n_e$ be the arc corresponding to variable $x$;
3:  traverse graph forwards from $n_e$, visiting nodes $n$
4:      mark $n$;
5:      IF $n = n_s$ THEN
6:          let $x_1, \ldots, x_k$ be variables for arcs on path $n_e \longrightarrow n_s$;
7:          initialize clause learning with $\neg x \vee \neg x_1 \vee \cdots \vee \neg x_k$;
8:          RETURN reporting contradiction;
9:  traverse graph backwards from $n_s$, visiting nodes $n$
10:     FOR EACH unassigned arc $n' \to n$ with $n'$ marked DO
11:         let $x'$ be the variable for arc $n' \to n$;
12:         let $P_1$ be the set of variables for arcs on path $n_e \longrightarrow n'$;
13:         let $P_2$ be the set of variables for arcs on path $n \longrightarrow n_s$;
14:         let $c = \neg x \vee \neg x' \vee \bigvee_{y \in P_1} \neg y \vee \bigvee_{y \in P_2} \neg y$;
15:         push $\neg x'$ in the propagation queue with $c$ as the reason;
```

Fig. 1. Propagator for acyclicity that is based on two depth-first traversals, one forwards from the end node of the added arc, and the other backwards from the starting node of the arc

in CDCL and other systematic SAT algorithms is Unit Propagation (UP) which infers l from $l \vee l_1 \vee \cdots \vee l_k$ and complements $\overline{l_1}, \ldots, \overline{l_k}$, and detects inconsistencies when all literals in a clause are false. Next we describe a propagator for acyclicity when the presence of arcs in the graph is indicated by propositional variables assigned *true*.

4.1 Propagator for Acyclicity

We consider the arcs of a graph to be either *enabled*, *disabled*, or *possible*, if the corresponding propositional variable is respectively assigned *true*, *false*, or unassigned.

When a propositional variable for an arc (n_s, n_e) is set *true*, we can infer new facts. Assume that the graph contains a cycle $n_1, n_2, \ldots, n_k, n_1$ and all arcs except (n_i, n_{i+1}) for some $i \in \{1, \ldots, k-1\}$ (or (n_k, n_1)) are now enabled and (n_i, n_{i+1}) is possible. Hence we conclude that (n_i, n_{i+1}) (or (n_k, n_1)) should be disabled, and therefore the corresponding propositional variable must be *false*, because otherwise there would be a cycle in the graph.

The above reasoning can be implemented by two depth-first traversals of the graph, formalized as the procedure in Figure 1. The first traversal identifies all nodes that can be reached from n_e through a path of enabled arcs. The second traversal identifies all nodes from which n_s can be reached through enabled arcs. Now, any arc from the former set of nodes to the latter has to be disabled and the corresponding propositional variables set false. There may be 0 or more such arcs. During the first traversal we also detect whether we can reach n_s from n_e, detecting a new cycle. If this is the case, we act as if the clause set had a clause stating that at least one of the arcs has to be disabled, and then run the CDCL learning algorithm starting as if this clause had just been falsified.

When an arc is disabled, no reasoning is required.

The amount of graph traversal can be reduced by observing that any cycle must be completely contained in a *strongly connected component* (SCC) of the graph. Hence when detecting cycles or inferring new literals, it is unnecessary to follow any arc from one SCC to another. SCCs can be recognized as a preprocessing step in linear time [24], and each node could be labeled with the index of its SCC.

4.2 Integration in a CDCL Implementation

The integration of the acyclicity constraint in the CDCL algorithm is straightforward: whenever a propositional variable x corresponding to an arc is set *true*, call the procedure propagator(x) (from Section 4.1), possibly adding new literals in the propagation queue, and report inconsistency if a cycle has emerged.

When a cycle has emerged, the CDCL clause learning process is initiated with a clause consisting of the negations of the propositional variables involved in the detected cycle. We call this clause the *cycle clause*. The cycle clause itself does not need to be added in the clause database.

When an almost-cycle has emerged, the corresponding cycle clause is added in the clause database, and the negation of the remaining unassigned arc variable is added in the propagation queue with the new cycle clause as its reason.

4.3 Preprocessing with Logical Simplifications

The use of non-clausal constraints impacts the use of preprocessors designed for standard CNF SAT problems. Standard preprocessors only look at the clause set, and not being aware of the non-clausal acyclicity constraint render some preprocessing methods incorrect. For example, variable elimination methods [22] and eliminating pure literals are incorrect in this context. Both can be made correct by leaving the arc variables – the only class of variables involved in the acyclicity constraint – out of consideration.

Preprocessing techniques that are *monotone*, that is, their results remain correct even if the preprocessing is only applied to a subset of the clauses, are directly applicable in our setting. Examples of monotone preprocessing techniques are *unit propagation lookahead* (also known as *failed literals*) and subsumption.

5 Comparison to Clausal Encodings

We compare the SAT algorithm extended with a built-in propagator for the acyclicity constraint to explicit encodings of the constraint we are aware of. Of particular interest are the *size* of the encodings, which determines how large or complicated graphs can be handled in practice, and the *propagation properties* of the encodings, which determine how well the encodings can prune search spaces. The following propagation properties are of interest.

INC	Is inconsistency (a cycle) detected with UP after all arcs forming a cycle are enabled?
BACK	For an enabled path $n_1, \ldots, n_k$, is arc (n_k, n_1) disabled by UP?

5.1 Explicit Enumeration of Cycles

The simplest encoding of acyclicity enumerates all possible cycles, and forbids enabling all arcs in each cycle. This leads to cycle clauses $\neg a_1 \vee \cdots \vee \neg a_n$ where $a_1, \ldots, a_n$ are variables for every arc in a cycle. The size of this encoding is in the worst case exponential in the number of nodes in the graph, and therefore in general impractical. We are not aware of prior uses of this encoding in any application. However, earlier works have – similarly to our propagator in Section 4.1 – generated some form of cycle clauses on-demand after detecting cycles by means external to the SAT solver [14].

This encoding propagates well. Every cycle is detected as soon as it emerges, so we have INC. When all but one arc in a potential cycle has been enabled, the remaining arc is disabled by unit propagation, so we have BACK.

5.2 Transitive Closure

In this encoding [21, 4], variables $t^{x,y}$ indicate that (x, y) belongs to the transitive closure of the relation corresponding to the underlying graph, that is, there is a (non-empty) directed path from x to y in the graph. Variables $a^{x,y}$ for arcs (x, y) imply $t^{x,y}$, and transitivity is expressed by $a^{x,y} \wedge t^{y,z} \to t^{x,z}$. Cycles are forbidden by $a^{x,y} \to \neg t^{y,x}$. This encoding is $\mathcal{O}(NM)$ size (for N nodes and M arcs), with $\mathcal{O}(N^2)$ variables. The encoding satisfies both INC and BACK, but it is often impractical [21], especially for complete graphs with its prohibitive $\mathcal{O}(N^3)$ size.

5.3 Topological Sorting with Indices

In this encoding, each node n in the graph is (nondeterministically) assigned an integer index $I(n)$ (typically encoded as a binary number with $\log N$ propositional variables). For each arc (n_1, n_2) there is a formula saying that if the arc is enabled, then $I(n_1) < I(n_2)$. While this encoding is very compact, the need to nondeterministically choose the indexing makes its propagation properties weak: even when all arcs are enabled or disabled, the indexing still has to be chosen before anything can be inferred about acyclicity. Hence this encoding satisfies neither of the propagation properties.

5.4 Tree Reduction

In this encoding [2] (which can be viewed as an efficient specialization of a SAT encoding of linear arithmetic constraints by Tamura et al. [23]), first the leaves (nodes without children) are identified and "removed", and the process is repeated until for acyclic graphs all nodes are guaranteed to be "removed". Essentially, we are assigning each node n an index $I(n)$ that is the maximum length of a path from n to a leaf node. We have to consider paths up to length $N - 1$. The encoding states that for each node n, $I(n) = k$ iff for all children n_0 of n we have $I(n_0) < k$ and for at least one child n_0 we have $I(n_0) = k - 1$. Finally, unit clauses state that $I(n) < N$ for every node n.

The number of clauses needed for each node is proportional to N times the number of arcs going out from it. Hence the total number of clauses is at most the product of the number of nodes and the number of arcs.

Violation of the acyclicity requirement is detected by unit propagation. Hence we have INC. The number of unit propagation steps is bounded by the size of the encoding. However, this encoding does not have the BACK property because leaf nodes are not recognized before all their outgoing arcs have been disabled. Hence no unit propagation takes place, and there is nothing else that could recognize the potential cycle.

5.5 Summary of Encoding Properties

The properties of the above encodings are summarized as follows.

encoding	size	propagation
Enumerative	$\mathcal{O}(v^v)$	INC, BACK
Transitive Closure	$\mathcal{O}(ev)$	INC, BACK
Tree Reduction	$\mathcal{O}(ev)$	INC
Topological Sort	$\mathcal{O}(v \log v + e \log v)$	-

The most compact encodings have the weakest propagation properties. The only encodings that have both of the important properties have a quadratic or an exponential size and are therefore impractical for graphs larger than some tens or hundreds of nodes. In contrast, by using a specialized propagator for acyclicity both of the propagation properties are satisfied, with linear time and space worst-case complexities.

6 Relation to Difference Logic

In this section, we provide a detailed analysis of the relationship between ACYC-SAT and *integer difference logic* (IDL) [15]. This logic is an extension of propositional logic by simple difference constraints of the form $x - y \geq k$ where x and y are integer variables and k is a constant. To streamline the forthcoming analysis, we assume a clausal representation rather than full propositional syntax. Moreover, the expressive power of the language can be further constrained by assuming particular values for the constant k. For our purposes, setting $k = 1$ is obvious as this amounts to constraints of the form $x > y$. In what follows, we compare SAT modulo acyclicity with an IDL fragment, denoted by IDL(1), based on formulas of the form

$$l_1 \vee \ldots \vee l_m \vee (x_1 > y_1) \vee \ldots \vee (x_n > y_n) \tag{1}$$

where $l_1 \vee \ldots \vee l_m$ is a propositional clause and $x_1 > y_1, \ldots, x_n > y_n$ difference constraints. Such formulas can express disequality, since $x \neq y$ is equivalent to $(x > y) \vee (y > x)$. As we shall see, equality $x = y$ is not modularly expressible using formulas of the form (1). To this end, it is essential that difference constraints may not be negated, since the formula $\neg(x > y)$ is equivalent to $x \leq y$, i.e., one half of equality.

Next we will establish *linear*, *faithful*, and *modular* translations between IDL(1) and ACYC-SAT. By linearity we mean transformation in linear time. For faithfulness, we identify integer variables used in IDL with nodes in ACYC-SAT and insist on a relatively tight correspondence of models. Finally, modularity means that the translation is feasible one expression at a time. An *LFM-translation* from a logic to another possesses all the three properties of linearity, faithfulness and modularity, and, if such a translation exists, we take this as an indication that the former can be straightforwardly expressed in the latter. Analogous frameworks based on polynomial translations have been used when ranking non-monotonic logics and logic programs on the basis of expressive power [11, 12].

Given a directed acyclic graph $G = \langle N, A \rangle$ and a node $n \in N$, we define the *elimination rank* of n in G, denoted by $\mathrm{er}_G(n)$, by setting $\mathrm{er}_G(n) = 0$ for any *root* node n and $\mathrm{er}_G(n) = i$ for any *non-root* node n that becomes a root once all nodes $n' \in N$ with $\mathrm{er}_G(n') < i$ have been eliminated from G.

Proposition 1. *There is an LFM-translation from ACYC-SAT to IDL(1).*

Proof sketch. An ACYC-SAT problem $\langle X, C, N, A, l \rangle$ conforming to Definition 1 can be linear-time translated into $\mathbf{T}_{\mathrm{IDL}}(C, A, l) = C \cup \{\neg l(x, y) \vee (x > y) \mid (x, y) \in A\}$. If v is a solution to the problem, a satisfying assignment v' for $\mathbf{T}_{\mathrm{IDL}}(C, A, l)$ is obtained by setting $v'(p) = v(p)$ for atomic propositions p and $v'(x) = \mathrm{er}_{\langle N, A' \rangle}(x)$ where $A' = \{(x, y) \in A \mid v(l(x, y)) = 1\}$. On the other hand, if v' is a satisfying assignment for $\mathbf{T}_{\mathrm{IDL}}(C, A, l)$, then a solution v can be extracted by setting $v(p) = v'(p)$ for atomic propositions p and $v(l(x, y)) = 1$ iff $v'(x) > v'(y)$ for $(x, y) \in A$. The translation $\mathbf{T}_{\mathrm{IDL}}$ is modular since clauses in C and arcs in A can be translated one-by-one. $\square$

Proposition 2. *There is an LFM-translation from IDL(1) to ACYC-SAT.*

Proof sketch. Let S be a set of formulas of the form (1) based on sets of propositional and integer variables X and V, respectively. The linear-time translation into ACYC-SAT is $\langle X, C, V, A, l \rangle$ where A is the set of arcs (x, y) for which $x > y$ appears in S, l is a labeling which assigns a new atom $l(x, y)$ to every $(x, y) \in A$, and $C = \mathbf{T}_{\mathrm{ACYC}}(S)$ contains a clause $l_1 \vee \ldots \vee l_m \vee l(x_1, y_1) \vee \ldots \vee l(x_n, y_n)$ for each extended clause (1) in S. The correspondence between solutions v to $\langle X, C, V, A, l \rangle$ and assignments v satisfying S is the same as in Proposition 1. The translation $\mathbf{T}_{\mathrm{ACYC}}$ is also modular as extended clauses can be translated independently of each other. $\square$

The expressive power of extended clauses (1) can be increased by allowing difference constraints of the form $x - y \geq 0$, or equivalently, of the form $x \geq y$. As discussed above, this amounts to negating difference constraints in (1) but we rather preserve the positive form of difference constraints and allow $x \geq y$ in extended clauses.

Theorem 1 (Intranslatability). *There is no faithful and modular generalization of the translation* $\mathbf{T}_{\mathrm{ACYC}}$ *from IDL(1) to IDL(0,1).*

Proof. The constraint $x > y$ is translated by $\mathbf{T}_{\mathrm{ACYC}}$ into a unit clause $l(x, y)$ and an arc (x, y) labeled by $l(x, y)$. This constraint is inconsistent with $y \geq x$ in IDL(0,1). Let us then assume a faithful and modular generalization of $\mathbf{T}_{\mathrm{ACYC}}$, which means $\mathbf{T}_{\mathrm{ACYC}}(y \geq x)$ should be independent of the respective translations of any other extended clauses. It is clear by the faithfulness of $\mathbf{T}_{\mathrm{ACYC}}$ that $\mathbf{T}_{\mathrm{ACYC}}(y \geq x)$ must be consistent as $y \geq x$ is satisfiable in IDL(0,1). Let v satisfy $\mathbf{T}_{\mathrm{ACYC}}(y \geq x)$ modulo acyclicity. Since v should be excluded in the presence of $\mathbf{T}_{\mathrm{ACYC}}(x > y)$, i.e., the unit clause $l(x, y)$, and subject to the semantics of SAT modulo acyclicity, we have that

1. $v(l(x, y)) = 0$ or
2. $v(l(y, v_1) \wedge \ldots \wedge l(v_n, x)) = 1$ for new atoms labeling arcs $(y, v_1), \ldots, (v_n, x)$ that form a path in the graph where $n \geq 0$ and $v_1, \ldots, v_n$ are potential new (and necessarily local) integer variables used in the translation of $y \geq x$.

If $n = 0$, then the second item reduces to $v(l(y, x)) = 1$. Due to the second item and the acyclicity property enforced in ACYC-SAT, $v(l(x, y)) = 1$ is not feasible. Thus $v(l(x, y)) = 0$ is necessary and since v was arbitrary, the translation $\mathbf{T}_{\mathrm{ACYC}}(y \geq x)$ must entail $\neg l(x, y)$. This reflects the fact that $\neg(x > y)$ is equivalent to $y \geq x$.

The translation $\mathbf{T}_{\mathrm{ACYC}}(x \geq y)$ entails $\neg l(y, x)$ by symmetry. Together, translations $\mathbf{T}_{\mathrm{ACYC}}(y \geq x)$ and $\mathbf{T}_{\mathrm{ACYC}}(x \geq y)$ are consistent with $l(y, z)$ and $l(z, x)$, i.e., the modular translations $\mathbf{T}_{\mathrm{ACYC}}(y > z)$ and $\mathbf{T}_{\mathrm{ACYC}}(z > x)$. This is because z is different from x and y, $\mathbf{T}_{\mathrm{ACYC}}(y \geq x)$ can only refer to $l(x, y)$ and $l(y, x)$, and thus adding $l(y, z)$ and $l(z, x)$ as unit clauses cannot interfere with consistency. Moreover, the selected arcs (y, z) and (z, x) do not create a cycle. A contradiction, since the theory $\{y \geq x, x \geq y, y > z, z > x\}$ in IDL(0,1) is inconsistent. $\square$

Theorem 1 shows formally that the expressive power of IDL(0,1) strictly exceeds that of IDL(1). This result, however, does not exclude the possibility for non-modular generalizations that, e.g., entirely embed difference constraints into clauses. But, on the other hand, achieving linearity may become a challenge due to interdependencies of integer variables. For instance, the transformation in [9] incurs a further logarithmic factor. To conclude the analysis in this section, we have identified a simple fragment of IDL that characterizes the expressive power of ACYC-SAT. This provides further insights into why an efficient implementation of the acyclicity extension can be expected.

7 Experimental Evaluation

We have implemented the acyclicity constraint propagator inside the MiniSAT solver, and then adapted it to the MiniSAT-based Glucose solver. We also extended the solvers' parsers with capabilities for reading graphs along with a SAT instance. All this is less than 500 lines of C++ code. Although our implementation does not try to amortize the costs of consecutive acyclicity tests, the propagator accounts only for a fraction of the total SAT solver runtime even with problem instances that intensively refer to graphs.

We empirically evaluated the performance of ACYC-SAT solvers on the Hamiltonian cycle problem as well as finding a directed acyclic graph, forest, or tree, subject to XOR-constraints over arcs. The problems and respective graph sizes in terms of nodes are indicated in the first two columns of Table 1. Per problem and graph size, we considered 100 (randomly generated) instances, that is, planar graphs in case of the Hamiltonian cycle problem or, otherwise, XOR-constraints to be fulfilled by a directed acyclic graph, forest, or tree, respectively. All experiments were run on a cluster of Linux machines, using a timeout of 3600 seconds per instance.

The evaluation includes our ACYC-SAT solvers Glucose-INC, Glucose-BACK, MiniSAT-INC, and MiniSAT-BACK, where the suffix INC indicates acyclicity propagation by detecting and denying cycles (using only the forwards traversal in Figure 1)

Problem	Size	Glucose-INC	Glucose-BACK	MiniSAT-INC	MiniSAT-BACK	Glucose-SAT	MiniSAT-SAT	Lingeling-SAT	Clasp-SAT	Clasp-ASP	Z3-SMT
Hamilton	100	0.21	0.07	**0.03**	0.04	224.14	275.00	2419.63	2600.90	0.95	2.45
	150	0.13	0.15	**0.10**	0.12	3440.00	3172.54	3536.02	—	20.16	50.64
Acyclic	25	0.08	0.05	0.05	**0.03**	2406.60	2934.30	1.61	1282.49	0.12	0.29
	50	2.34	**0.28**	1.64	0.29	3147.91	2988.30	17.09	—	0.76	7.61
	75	682.86	8.09	856.47	**4.76**	3241.00	3276.92	99.60	—	282.01	167.74
	100	2180.98	964.28	2172.01	**647.13**	3170.48	3176.70	2760.52	1984.10	831.33	2278.63
Forest	25	**0.59**	0.64	0.75	0.72	118.70	139.88	3.10	3.59	4.09	4.54
	50	**301.46**	304.44	466.56	498.00	1165.53	1438.49	667.24	1125.86	1039.26	1205.63
	75	**909.15**	1006.73	1011.05	920.43	2597.99	2708.27	1019.68	1470.12	1501.76	1755.28
	100	1349.29	1418.25	1271.86	**1269.47**	2882.20	2853.03	2131.73	2597.71	1632.94	2690.67
Tree	25	0.80	0.74	**0.67**	0.83	72.93	6.12	3.17	4.12	4.37	4.75
	50	**301.81**	315.83	564.05	544.43	815.09	1230.09	685.38	1126.76	1193.09	1208.36
	75	**947.61**	999.07	976.40	1025.02	2646.64	2749.26	1044.51	1633.95	1495.32	1726.56
	100	1348.91	1414.68	1330.81	**1224.28**	2882.36	2861.33	2239.12	2621.82	1995.19	2538.20

Table 1. Comparison between solvers for ACYC-SAT, SAT, ASP, and SMT on Hamiltonian cycle and directed acyclic graph problems

and BACK expresses that arc variables are also falsified to prevent cycles (using the entire propagator(x) routine in Figure 1). We compared our solvers to their base versions Glucose-SAT (3.0) and MiniSAT-SAT (2.2.0), run with plain SAT encodings based on Tree Reduction (cf. Section 5). Notably, for a graph with N nodes, the maximum length $I(n) = k$ from a node n to a leaf node is represented in terms of propositions for $I(n) \leq k, \ldots, I(n) \leq N-1$, thus exploiting the order encoding approach [23]. Furthermore, we have included Lingeling (ats-57807c8-131016) as an example of a SAT solver with a good performance that is unrelated to MiniSAT and Glucose. These SAT solvers are complemented by the combined SAT and Answer Set Programming (ASP) solver Clasp (3.0.4), run as Clasp-SAT on SAT encodings or as Clasp-ASP on more compact ASP encodings of Tree Reduction of size $\mathcal{O}(e)$. The ASP formalism includes an (implicit) acyclicity test which enables more compact encodings than with SAT. Similarly, difference logic, supported by Z3-SMT (4.3.2), allows for compact encodings of acyclicity constraints as described in Section 6 (see [19] for relations to ASP).

Table 1 gives average runtimes taking timeouts as 3600 seconds, while indicating timeouts in all runs by "—" as well as the minimum average runtime per row in boldface. The advantage of our ACYC-SAT solvers clearly shows on the Hamiltonian cycle instances, where the requirement of exactly one incoming and outgoing arc per node along with the global acyclicity condition (excluding the incoming arc of a fixed starting node) permit a very compact encoding. Given that all instances are solved easily, the efforts of Glucose-BACK and MiniSAT-BACK to falsify arc variables result in small overhead compared to Glucose-INC and MiniSAT-INC which merely check acyclicity. In fact, the higher average runtime of Glucose-INC on instances with 100 nodes is due to a single outlier, taking longer than the other 99 instances together. On the other hand, all four plain SAT solvers suffer from less compact encodings, leading to significantly higher runtimes and plenty of timeouts. The latter can also be observed in comparison to Clasp-ASP and Z3-SMT, whose average runtimes are still two orders of magnitude higher than the ones of our ACYC-SAT solvers.

Considering the problem of finding a directed acyclic graph subject to XOR constraints, the extended propagation of Glucose-BACK and MiniSAT-BACK pays off and significantly reduces the amount of the search needed in comparison to Glucose-INC and MiniSAT-INC. Advantages over Clasp-ASP and Z3-SMT still amount to one order of magnitude in average runtime. In general, plain SAT solvers have again the most difficulties. The exceptionally good performance of Lingeling-SAT on instances up to size 75 is mostly due to its preprocessing and inprocessing techniques, not shared by the other three SAT solvers. Differences in SAT solver engines become also apparent when comparing Clasp-SAT which fails on instances with 50 or 75 nodes but solves more instances of size 100 than Lingeling-SAT. Finally, the problems of finding a forest or tree fulfilling XOR-constraints add further restrictions on directed acyclic graphs in question. Therefore, these problems are harder for our ACYC-SAT solvers, and side constraints seem to dominate over differences in acyclicity propagation. Nevertheless, the acyclicity extensions of both Glucose and MiniSAT still have a significant edge over the other solvers.

8 Related Work

The way we integrate graph constraints in the SAT framework is highly analogous to the SMT framework [25, 1]. Typical implementations of difference logic in SMT solvers [3, 15, 20] involve graph-based algorithms. Rather than finding values to integer variables and then checking the inequalities, the satisfiability of a set of difference constrains can be decided by checking the existence of a negative-weight loop in the corresponding weighted graph using standard algorithms. The SAT modulo Graphs framework proposed in this paper exploits graph algorithms in the implementation but also suggests using graphs explicitly as a core concept in modeling. Reasoning techniques proposed in [15] include a counterpart to our rule that infers that an arc that would complete a cycle must be disabled. Based on an experimental comparison between our solvers and Z3, we believe that Z3 performs similar inferences (but were not able to confirm this by inspecting its source code): numbers of decisions and conflicts in the search performed by Z3 are comparable to our Glucose-BACK solver, and significantly lower than with our Glucose-INC solver, as shown in Figure 2 which plots the numbers of decisions for these solvers and all of the Acyclic instances. Plots for conflicts are similar. Z3 runtimes are about 15 times higher than those of Glucose-BACK for small instances, and more for bigger ones, which must be due to the far higher overhead of difference logic.

Constraints on graphs have earlier been of some interest in the automated reasoning and constraint programming communities. The works closest to ours are the following. Hoffmann and van Beek have recently presented an acyclicity constraint specialized for the Bayesian network learning problem, in an unpublished work [10]. Dooms et al. [6, 7] have proposed the CP(Graph) domain for constraint programming, in which variables have graphs as values and constraints express relations between different graphs.

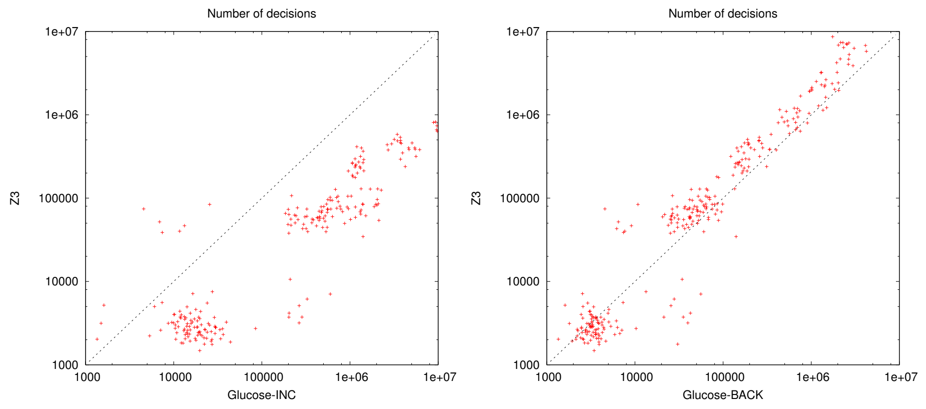

Fig. 2. Decisions for Z3 vs. Glucose-BACK and Glucose-INC on Acyclic instances from Table 1

9 Conclusion

We have presented a constraint propagator for a graph acyclicity constraint and its implementation in a SAT solver. In some important classes of SAT applications it is critical to express the acyclicity constraint compactly and to exploit it maximally to achieve efficient SAT solving. Such applications include Bayesian network learning, answer set programming, and reasoning about networked systems that are kept in an acyclic configuration, for example many electricity networks.

Our experiments show good scalability of our solvers in comparison to competing frameworks, including difference logic, and often dramatic improvements over acyclicity constraints encoded in the standard clausal SAT problem are apparent.

We are in the process of integrating acyclicity constraints in MAXSAT solvers, to be able to experiment with structure learning for Bayesian networks which is reducible to the weighted partial MAXSAT problem [4].

Future work includes addressing other important graph constraints, stemming from applications involving systems such as utility networks (power, water, telecommunications) and transportation. Graph constraints arising in these applications include reachability (one node is reachable from another), connectivity, and simple paths (a node is on a simple path between two given nodes). Similarly to acyclicity, these constraints do not appear to be expressible as clauses so that compactness (size less than quadratic) and strong propagations are both achieved, and unlike acyclicity, their expression in frameworks such as difference logic or ASP is not straightforward.

Acknowledgments

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

References

1. Audemard, G., Bertoli, P., Cimatti, A., Korniłowicz, A., Sebastiani, R.: A SAT based approach for solving formulas over Boolean and linear mathematical propositions. In: Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings. Number 2392 in Lecture Notes in Computer Science, Springer-Verlag (2002) 195–210
2. Corander, J., Janhunen, T., Rintanen, J., Nyman, H., Pensar, J.: Learning chordal Markov networks by constraint satisfaction. In Burges, C.J.C., Bottou, L., Welling, M., Ghahramani, Z., Weinberger, K., eds.: Advances in Neural Information Processing Systems 26. (2014) 1349–1357
3. Cotton, S., Maler, O.: Fast and flexible difference constraint propagation for DPLL(T). In: Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing. Number 4121 in Lecture Notes in Computer Science, Springer-Verlag (2006) 170–183
4. Cussens, J.: Bayesian network learning by compiling to weighted MAX-SAT. In: Proceedings of the Conference on Uncertainty in Artificial Intelligence, AUAI Press (2008) 105–112
5. Denecker, M., Ternovska, E.: A logic of nonmonotone inductive definitions. ACM Transactions on Computational Logic **9**(2) (2008) 14:1–14:52
6. Dooms, G., Deville, Y., Dupont, P.: Cp(graph): Introducing a graph computation domain in constraint programming. In van Beek, P., ed.: Principles and Practice of Constraint Programming - CP 2005. Volume 3709 of Lecture Notes in Computer Science., Springer-Verlag (2005) 211–225
7. Dooms, G., Katriel, I.: The minimum spanning tree constraint. In Benhamou, F., ed.: Principles and Practice of Constraint Programming – CP 2006. Volume 4204 of Lecture Notes in Computer Science., Springer-Verlag (2006) 152–166
8. Gebser, M., Janhunen, T., Rintanen, J.: Answer set programming as SAT modulo acyclicity. In: ECAI 2014. Proceedings of the 21st European Conference on Artificial Intelligence, IOS Press (2014)
9. Heljanko, K., Keinänen, M., Lange, M., Niemelä, I.: Solving parity games by a reduction to SAT. Journal for Computer and System Sciences **78**(2) (2012) 430–440
10. Hoffmann, H.F., van Beek, P.: A global acyclicity constraint for Bayesian network structure learning (September 2013) unpublished manuscript in the Doctoral Program of the International Conference on Principles and Practice of Constraint Programming.
11. Janhunen, T.: Evaluating the effect of semi-normality on the expressiveness of defaults. Artificial Intelligence **144**(1-2) (2003) 233–250
12. Janhunen, T.: Some (in)translatability results for normal logic programs and propositional theories. Journal of Applied Non-Classical Logics **16**(1-2) (2006) 35–86
13. Kautz, H., Selman, B.: Pushing the envelope: planning, propositional logic, and stochastic search. In: Proceedings of the 13th National Conference on Artificial Intelligence and the 8th Innovative Applications of Artificial Intelligence Conference, AAAI Press (1996) 1194–1201
14. Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT solvers. Artificial Intelligence Journal **157**(1) (2004) 115–137
15. Mahfoudh, M., Niebert, P., Asarin, E., Maler, O.: A satisfiability checker for difference logic. Proceedings of SAT 2002 – Theory and Applications of Satisfiability Testing **2** (2002) 222–230
16. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A new search algorithm for satisfiability. In: Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers., 1996 IEEE/ACM International Conference on. (1996) 220–227

17. Mitchell, D.G.: A SAT solver primer. EATCS Bulletin **85** (February 2005) 112–133
18. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of the 38th ACM/IEEE Design Automation Conference (DAC'01), ACM Press (2001) 530–535
19. Niemelä, I.: Stable models and difference logic. Annals of Mathematics and Artificial Intelligence **53**(1-4) (2008) 313–329
20. Nieuwenhuis, R., Oliveras, A.: DPLL(T) with exhaustive theory propagation and its application to difference logic. In: Proceedings of 17th the International Conference on Computer Aided Verification. Number 3576 in Lecture Notes in Computer Science, Springer-Verlag (2005) 321–334
21. Rintanen, J., Heljanko, K., Niemelä, I.: Parallel encodings of classical planning as satisfiability. In Alferes, J.J., Leite, J., eds.: Logics in Artificial Intelligence: 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004. Proceedings. Number 3229 in Lecture Notes in Computer Science, Springer-Verlag (2004) 307–319
22. Subbarayan, S., Pradhan, D.K.: NiVER: Non increasing variable elimination resolution for preprocessing SAT instances. In Hoos, H.H., Mitchell, D.G., eds.: Theory and Applications of Satisfiability Testing, 7th International Conference, SAT-2004. Vancouver, BC, Canada, May 10-13, 2004. Revised selected papers. Number 3542 in Lecture Notes in Computer Science, Springer-Verlag (2005) 276–291
23. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. Constraints **14**(2) (2009) 254–272
24. Tarjan, R.E.: Depth first search and linear graph algorithms. SIAM Journal on Computing **1**(2) (1972) 146–160
25. Wolfman, S.A., Weld, D.S.: The LPSAT engine & its application to resource planning. In: Proceedings of the 16th International Joint Conference on Artificial Intelligence, Morgan Kaufmann Publishers (1999) 310–315