Satisfiability algorithms for conjunctive queries over trees

James Cheney
LFCS, University of Edinburgh
jcheney@inf.ed.ac.uk

ABSTRACT

We investigate the satisfiability problem for conjunctions of constraints over ordered, unranked trees, including child, descendant, following-sibling, root, leaf, and first/last child constraints. We introduce new, symbolic approaches based on graph transformations, which simplify and check the consistency of a problem first, and delay blind search as long as possible. We prove correctness and termination for these algorithms. We also analyze the complexity of important special cases: binary and k-ary intersection of certain classes of XPath expressions. Our main complexity result is that binary intersection (for positive, simple navigational XPath over all axes) is tractable for expressions with a bounded number of changes in direction in the path, which is typically small.

Categories and Subject Descriptors

H.2.3 [**Database management systems**]: Languages—query languages; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—Complexity of proof procedures

General Terms

Algorithms, Theory

Keywords

XPath, trees, satisfiability

1. INTRODUCTION

Ordered, unranked, labeled trees are a common and useful data structure arising in many applications. Over the last ten years, XML has been developed as a standard syntax for such trees, and is now used in a wide variety of settings, including generating and processing HTML, Microsoft Office documents, configuration files, game content, music and scientific data. In XML processing settings, path expressions have become a very popular way to navigate through tree structures, and in particular the XPath [9] language supports a rich set of path expressions, including child, descendant, and sibling steps (and their converses), and root, leaf, child position, and label tests.

In applications based on XPath, it is often useful to know whether a path expression is *satisfiable*, or whether two or more path expressions intersect (that is, whether there is a single tree in which they all select a common node, starting from the root). In this paper we consider algorithms for the general satisfiability problem for conjunctive formulas in the language of ordered trees. These problems arise in a number of contexts, including query optimization and static analysis [15, 4].

The satisfiability and intersection problems for XPath expressions have been studied previously. Hidders [18] investigated the complexity of satisfiability for various fragments of XPath, showing in particular that satisfiability of arbitrary conjunctive formulas over ordered trees is NP-complete while satisfiability of single XPath expressions is in PTIME. Subsequently many other researchers have explored expressiveness and complexity aspects of logics over ordered and unordered trees, and both with and without a schema [5, 6, 7, 8, 14, 20].

Algorithmic aspects of satisfiability for conjunctive properties of trees have not been thoroughly investigated, although here has been some work on algorithms for the special case of intersection for fragments of XPath. Hammerschmidt et al. [17] give a quadratic algorithm for the special case of "downward" XPath expressions, along with a discussion showing that the complexity is linear in common cases. While Hidders' results show that the intersection problem for downward XPath expressions is NP-complete, Hammerschmidt et al.'s approach generalizes to show that k-ary intersection is solvable in $O(n^k)$ time, where n is the size of the problem. However, for larger fragments of XPath, the complexity of binary (or k-ary) intersection still appears to be open, including the complexity of determining whether two positive, filter-free XPath expressions have a nonempty intersection. Binary intersection is a common case, with applications to filtering and static analysis of XPath and XQuery [17, 4].

There are close connections between tree logics and various kinds of tree automata, which give general algorithms for solving such problems. However, the worst-case complexity of satisfiability in these logics is non-elementary [24]. Genevès et al. [14] developed a static analyzer for XPath and XML Schema constraints by reducing to a highly expressive modal mu-calculus over trees. These techniques can in prin-

ICDT 2011, March 21–23, 2011, Uppsala, Sweden.
Copyright 2011 ACM 978-1-4503-0529-7/11/0003 ...$10.00

ciple be used to solve satisfiability problems, and in practice may offer advantages due to the capability to handle schema and path constraints using a uniform framework, but appear difficult to analyze to obtain better bounds for special cases.

Since the general problem is NP-complete, in principle it can be translated to any other NP-complete problem. This approach has been taken in some work on static analysis for XPath or XQuery [3, 4]. Fast solvers for many such problems (such as Boolean satisfiability, integer linear programming, and constraint satisfaction problems) have been developed. In recent work [4], we explored translating tree satisfiability problems to existential first-order formulas over integer linear arithmetic, and then using a Satisfiability Modulo Theories (SMT) solver to solve the translated problems [10, 11, 21]. This approach is effective but is also opaque: it is difficult to diagnose performance problems or identify tractable special cases. Furthermore, while SAT or SMT solvers generally do provide satisfying instances that can be translated to satisfiers to the original problem, they do not usually provide proofs of unsatisfiability that provide insight into the original problem.

In this paper, we explore a direct, symbolic approach to solving tree satisfiability problems. It is common practice in work on XPath to visualize path expressions as so-called "tree-patterns" (see e.g. [18, 20]), whose vertices stand for vertices in a tree and whose edges stand for parent-child or ancestor-descendant relationships. A tree pattern can be matched to a tree by giving a function mapping the vertices of the pattern to the vertices of the tree in a way that satisfies the constraints specified by the edges, i.e., a homomorphism. Likewise, we can visualize a conjunctive formula over trees as a graph (which need not be a tree), and such a graph can be matched to a tree via a homomorphism.

Our approach essentially searches for a way to transform the graph representation of a problem to a form that is transparently satisfiable. This is analogous to symbolic techniques such as the "chase" for tableau queries [1], Robinson's algorithm for unification of first-order terms, or symbolic Gaussian elimination. However, since the problem is NP-complete, the transformations involve some nondeterministic guessing, introducing the need for backtracking in some situations. Our approach tries to avoid nondeterministic guessing as long as possible, a heuristic based on the observation that satisfiable problems always have small witnesses, but unsatisfiable problems often also have relatively shallow "proofs" of unsatisfiability. Moreover, a symbolic approach appears to have more potential for integration into modern SMT solvers [21, 2] than existing automata-theoretic techniques.

Other algorithms for rewriting or checking satisfiability of conjunctive queries, due to Olteanu et al. [23, 22], Lakshmanan et al. [20], and Benedikt and Koch [6], have some commonalities with our approach. Briefly, the difference is that Olteanu et al.'s and Benedikt and Koch's algorithms were developed for a different purpose, and were not designed for testing satisfiability and understanding special cases, while Lakshmanan's algorithm focuses on tree pattern queries and does not handle sibling steps. We relate our work to these approaches in more detail in Section 7.

1.1 Examples

Figure 1 shows an example constraint, loosely based on an example from Olteanu [22]. In this figure, we use single diagonal/vertical lines to denote downwards steps, and horizontal arrows to denote following-sibling steps. Also, an edge label + or $*$ stands for the transitive or reflexive transitive closure respectively. Dotted lines (see Figure 2) stand for document ordering constraints. Node labels indicate unary predicates such as root, leaf or labeling constraints.

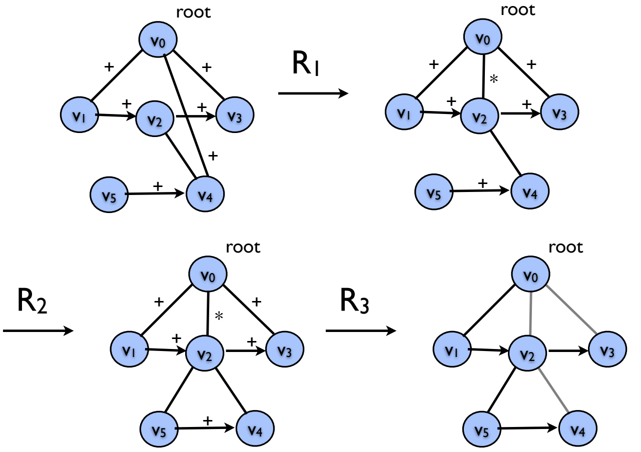


Figure 1: Example 1

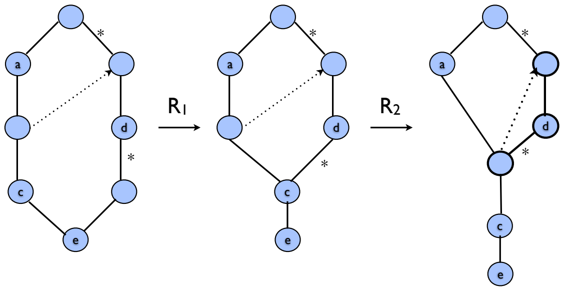


Figure 2: Example 2

For exposition purposes, the following description (and the figures) glosses over some aspects of the algorithm, such as the fact that we maintain the transitive closures of the descendant-or-self and document ordering edges. Instead, the figures just show enough edges to generate the full graph that the algorithm actually manipulates.

In Figure 1, we show how the first problem translates into the constraint graphs used by our algorithm. The basic idea of the algorithm is to search for parts of the graph that are not tree-like, and resolve them. In this example, the graph is acyclic, but nodes v_3 and v_4 are "join points", that is, have multiple paths from the root. In the first reduction step R_1, we resolve the join point v_4, by constraining v_0 to be an ancestor of v_2, the parent of v_4. Note that the other possibility is for $v_0 = v_2$ to be merged, but this makes the graph inconsistent because then v_0 would be both a sibling and child of v_0. This also resolves the join point v_3. However, v_4 remains a join point, since it is both a sibling and a child. Siblings always have the same parents, so in step R_2 we can safely add a child edge from v_2 to v_5. After these reductions, the graph is clearly satisfiable; we can read a satisfying tree directly from its structure, as shown in the last step R_3.

Note that our algorithm involves (essentially) no undirected search or backtracking for this example, whereas the algorithm of Olteanu et al. [22] takes around 10 reduction steps to normalize a similar problem to a satisfiable form (however that algorithm was not designed with satisfiability testing in mind, but rather for eliminating reverse axis steps from queries).

In the second example, shown in Figure 2, we show that a small graph is unsatisfiable, illustrating that our approach can find relatively small "proofs" of unsatisfiability as well as witnesses to satisfiability. Again, we focus on resolving join points. There are two join points: one at the parent of d and another at the lowermost node labeled e. We focus on e, which has two parents, and they must be equal if the graph is satisfiable in a tree. The first step R_1 merges them. Next, the node c is a join point, and it has a parent and an ancestor-or-self labeled d. If d is an ancestor-or-self of c then either they are equal or d is an ancestor-or-self of c's parent. Since d and c have different labels, they cannot be equal, so in step R_2 we make d an ancestor-or-self of c's parent. This leads to a graph with a nontrivial, irreflexive cycle: the highlighted subgraph in the rightmost part of Figure 2 is unsatisfiable because it requires d's parent to follow a descendant-or-self of d in the document order, which is impossible. There were no alternative ways to resolve the join point at e, so the graph is unsatisfiable.

1.2 Summary

Our main contributions are as follows:

- We give a constraint-solving style algorithm for testing satisfiability of unordered conjunctive queries over trees, with root, leaf, and node label constraints. We extend it to support document order and following-sibling constraints. We prove correctness (termination, soundness and completeness) for these algorithms, and discuss (without full proof of soundness) techniques for handling the next-sibling, first and last child constraints.

- We examine special cases of binary intersection for XPath queries, problems whose exact complexity remains open. We show that satisfiability of k-ary intersections of forward positive XPath queries (involving child, descendant-or-self, and following-sibling steps and downward-only filters) is solvable in $O(n^k)$ time. We extend this result to show that binary intersection for arbitrary simple XPath queries is solvable in $O(n^{3k-1})$ time, where k is the *degree of alternation*, or number of changes in direction in the problem, even in the presence of downward filters.

Although the symbolic algorithms we present do not offer new insights into worst-case complexity, they appear to be new and the deterministic part of the algorithm can be used as a fast approximate satisfiability test or to simplify queries before using another solver. In particular, simplifying a binary XPath intersection can decrease the degree of alternation of the problem, leading to speedups compared to a naive application of the binary intersection algorithm we give.

The rest of the paper is structured as follows. In the next section, we review the syntax and semantics of conjunctive queries over trees, using a graph formalism similar to the approach taken in Hidders [18] or Gottlob et al. [16]. In Section 3, we present a basic algorithm that suffices to decide satisfiability for unordered trees. We extend this to handle ordered trees with following-sibling constraints in Section 4. In Section 5, we study the binary intersection problem, or equivalently, satisfiability of constraints whose graphs are (undirected) cycles. In Section 6 we discuss and relate the two approaches. In Section 7 we discuss related and future work and Section 8 concludes.

2. BACKGROUND

Given a directed graph $G = (V, E)$, we write $in_G(v)$ for the set of E-predecessors of v (that is, $\{w \mid (w, v) \in E\}$ and symmetrically $out_G(v)$ for the set of E-successors of v.

For our purposes, a *tree* $T = (V_T, E_T, r_T, \lambda_T, <_T)$ over a given node label alphabet Σ is an ordered, directed acyclic graph (V_T, E_T) along with:

1. a root $r_T \in V_T$ such that $in_T(r_T) = \emptyset$ and $in_T(v)$ is a singleton for $v \neq r_T$.
2. a labeling function $\lambda_T : V_T \to \Sigma$ that assigns each vertex a label from Σ, and
3. a relation $<_T \subseteq V_T \times V_T$ that is a total ordering of the vertices V that is compatible with the partial ordering on vertices formed by the edges, in the sense that if $(x, y) \in E_T^*$ then $x \leq_T y$, and if $(x, y) \in E_T^*$ and $z \leq_T y$ then either $(x, z) \in E_T^*$ or $z \leq_T x$.

The global ordering $<$ induces a local ordering on the set of children of each node, such that visiting the nodes in increasing order according to $<$ is a preorder traversal of the tree. This presentation is equivalent to other common presentations of ordered trees, for example taking V to be a prefix-closed subset of $\mathbb{N}^*$, linearly ordered by the preorder traversal.

We write $precsib_T(v) = \{w <_T v \mid \exists z.(z, w) \in E \wedge (z, v) \in E\}$ for the set of preceding siblings and $follsib_T(v) = \{w >_T v \mid \exists z.(z, w) \in E \wedge (z, v) \in E\}$ for the set of following siblings of v in T.

Consider conjunctions ϕ of atomic formulas of the following forms:

$$\begin{array}{rcl} A & ::= & x \not\approx y \mid x \lhd y \mid x \lhd^* y \mid \mathsf{root}(x) \mid \mathsf{leaf}(x) \mid \mathsf{lab}_a(x) \\ & \mid & x \preceq y \mid x \twoheadrightarrow y \mid x \twoheadrightarrow^* y \mid \mathsf{fst}(x) \mid \mathsf{lst}(x) \end{array}$$

Here, x, y are variables, and a is one of a set of node labels Σ (which may be infinite and must be nonempty). The first line defines the *unordered* constraints, that is, those that are insensitive to document order. The constraint $x \not\approx y$ means that x and y are distinct nodes, constraint $x \lhd y$ says that y is a child of x, whereas $x \lhd^* y$ denotes the reflexive, transitive closure of the child relation. The unary formulas $\mathsf{root}, \mathsf{leaf}, \mathsf{lab}_a$ refer to the root, leaf, and labeling predicates. The ordered constraints include the document order $x \preceq y$, the next-sibling constraint $x \twoheadrightarrow y$, the following-sibling constraint $x \twoheadrightarrow^* y$, and first and last child constraints $\mathsf{fst}(x), \mathsf{lst}(x)$. In what follows, we write ϕ for conjunctions of atomic formulas and X typically denotes a set of variables of a formula ϕ.

We take as primitive the reflexive, transitive closures of the basic child and next-sibling constraints. We can define abbreviations for their transitive closures:

$$\begin{array}{rcl} x \lhd^+ y & \iff & x \lhd^* y \wedge x \not\approx y \\ x \twoheadrightarrow^+ y & \iff & x \twoheadrightarrow^* y \wedge x \not\approx y \\ x \prec y & \iff & x \preceq y \wedge x \not\approx y \end{array}$$

If we had taken the transitive closures as primitive, then defining their transitive reflexive closures would introduce a disjunction, taking us outside the conjunctive language.

We call a function $h : X \to V$ a *valuation.* We say that a valuation h *satisfies* an atomic formula A with free variables from X in T, written $T, h \models A$, provided one of the following hold:

$$\begin{array}{rcl}
T, h \models x \not\approx y & \iff & h(x) \neq h(y) \\
T, h \models x \lhd y & \iff & (h(x), h(y)) \in E_T \\
T, h \models x \lhd^* y & \iff & (h(x), h(y)) \in E_T^* \\
T, h \models \mathsf{root}(x) & \iff & h(x) = r_T \\
T, h \models \mathsf{leaf}(x) & \iff & \mathit{out}_T(h(x)) = \emptyset \\
T, h \models \mathsf{lab}_a(x) & \iff & \lambda_T(h(x)) = a \\
T, h \models x \preceq y & \iff & h(x) = h(y) \text{ or } h(x) <_T h(y) \\
T, h \models x \twoheadrightarrow y & \iff & h(x) \in \mathit{precsib}_T(h(y)) \text{ and} \\
& & \mathit{precsib}_T(h(y)) \cap \mathit{follsib}_T(h(x)) = \emptyset \\
T, h \models x \twoheadrightarrow^* y & \iff & h(x) = h(y) \text{ or } h(x) \in \mathit{precsib}_T(h(y)) \\
T, h \models \mathsf{fst}(x) & \iff & \mathit{precsib}_T(h(x)) = \emptyset \\
T, h \models \mathsf{lst}(x) & \iff & \mathit{follsib}_T(h(x)) = \emptyset
\end{array}$$

Moreover, given a conjunction of atomic constraints $\phi = A_1 \wedge \cdots \wedge A_n$, we write $T, h \models \phi$ to indicate that $T, h \models A_i$ for each i.

We adopt a graph representation of conjunctive formulas, similar to that employed in [18, 16, 7]. In detail, a *tree description graph* is a structure $G = (V_G, E_G, \lambda_G, \alpha_G)$ such that (V_G, E_G) is a graph, $\lambda : V_G \to \mathcal{P}(\{\mathsf{lab}_a, \mathsf{root}, \mathsf{leaf}, \mathsf{fst}, \mathsf{lst}\})$ maps each vertex to a set of labels, and $\alpha : E_G \to \mathcal{P}(\{\lhd, \lhd^*, \preceq, \not\approx, \twoheadrightarrow, \twoheadrightarrow^*\})$ maps each edge to a set of binary predicate symbols. Moreover, given T we say that a valuation $h : V_G \to V_T$ *satisfies* G if for each binary relation symbol $R \in \alpha(x, y)$ we have $T, h \models R(x, y)$ and for each unary predicate $P \in \lambda(x)$ we have $T, h \models P(x)$. In the common case where $V_G = V_T$ and h is the identity function, we write simply $T \models G$ instead of $T, h \models G$.

There is a standard translation from (positive, union-free) XPath expressions to conjunctive tree formulas [6, 18]. The formulas produced by this translation are trees, often called tree patterns; for simple (that is, filter-free) expressions the tree pattern is linear (that is, every node has indegree and outdegree at most 1, and there are unique source and sink nodes with outdegree or respectively indegree 0). In this paper, we leave out the details of XPath proper, and work directly with constraint graphs; these include positive XPath formulas as a special case.

The *conjunctive tree satisfiability problem* (or *TreeSAT* for short) is the problem of determining whether a conjunctive formula $\phi = A_1 \wedge \cdots \wedge A_n$ (over variables from X) is satisfiable by some tree T and valuation $h : X \to V_T$.

Theorem 1. *TreeSAT is NP-complete.*

Proof. The subproblem without leaf, first or last child, or next-sibling constraints is already NP-complete [18, 7]. We extend the approach taken in these proofs to show that satisfiability remains in NP in the presence of leaf, first/last child constraints, and next-sibling constraints.

To show that the problem is in NP, let T, h be a witness satisfying ϕ. We will construct a tree $T' = (V', E', <', \lambda', r')$ that also satisfies ϕ and whose size is polynomial in the size of ϕ, as follows. The vertices V' are the nodes of $h[X]$ in the range of h, the immediate parents of such nodes, and the root of T. The ordering $<'$ is the restriction of $<_T$ to V'. The edges are defined as follows:

$$\begin{array}{rcl}
E' & = & \{(v, w) \in V' \times V' \mid \\
& & \quad (v, w) \in E^+ \\
& & \quad \wedge \forall w' \in V'.(v, w') \in E^+ \implies (w, w') \in E^*\}
\end{array}$$

That is, the edges are those induced by the transitive closure of the edges of T, restricted to V'. The root r' is the root r of T, and the labeling λ' is the labeling of T restricted to V'.

Clearly, T' still has a root and is totally ordered. Some calculations suffice to show that T' is a tree, r' is still the root, etc. It is straightforward to show that the construction of T' preserves root, leaf, label, and first/last constraints, because the inclusion of parents of nodes in the range of h ensures that any node in the range of h that was previously a root, leaf, first or last child still is.

To see that the construction preserves child steps, note that if $(v, w) \in E$ where $v, w \in V'$ then (v, w) will also be in E', since w is a minimal descendant of v in V'. To see that it preserves descendant steps, we proceed by complete induction on the length of the path from v to w. If $v = w$ then there is nothing to prove since obviously $(v, w) \in (E')^*$. If there are no V'-nodes between v and w, then it is easy to see that $(v, w) \in E'$. Finally, if there is a node $v' \in V'$ strictly between v and w then clearly $(v, v') \in E^*$ and $(v', w) \in E^*$ imply by induction that $(v, v') \in (E')^*$ and $(v', w) \in (E')^*$, which implies $(v, w) \in (E')^*$. It is easy to see that the construction preserves following-sibling steps. To see that the construction preserves next-sibling steps, suppose x, y were mapped to adjacent siblings $u = h(x), v = h(y)$ by h. Then their common parent p is in V', and any descendant of p in V' that is strictly between u and v with respect to $<_T$ must have been a descendant of u, so p is not its immediate parent in T'. □

3. UNORDERED CONSTRAINTS

In this section, we present a complete, constraint-solving algorithm for determining satisfiability of formulas over the unordered constraints. We extend it to handle the ordered constraints in the next section.

We write $G\{A_1, \ldots, A_n\}$ to indicate that G is a graph containing constraints $A_1 \ldots, A_n$. If $v, w \in V_G$ then we write $G[v := w]$ for the graph obtained by merging v and w. We consider a simple form of *graph rewrite rules* of the form $G \leadsto G'$ where, in general, G' is either a collapsed version of G, an extension of G to include additional constraints, or a special symbol $\perp$ indicating failure.

The nondeterministic constraint-solving algorithm we are defining works as follows on a graph G:

1. Compute the transitive closure of the descendant-or-self edges and check for cycles.

2. Check for patterns that imply an equality between variables that are not already identified, particularly cycles involving descendant-or-self edges. If we find such a pair, merge the nodes.

3. Check for local inconsistencies, such as conflicting node labels or cycles involving irreflexive edges. Fail if any are found.

4. Check for "joins", or nodes with two incoming edges whose sources are not related. If one is found, guess an ordering between the incoming edges, and propagate it into G.

5. If there are no local inconsistencies, cycles, equality-propagation patterns, or joins, then G is in solved form and we return success (with solution G). A satisfying tree can be extracted from G.

3.1 Saturation

We will maintain structural invariants on constraint graphs, by keeping the graph saturated with respect to the following rules:

1. Containment: $(\lhd_G) \subseteq (\lhd_G^*) \cap (\not\approx)$.

$$G\{x \lhd y\} \quad\leadsto\quad G \cup \{x \lhd^* y, x \not\approx y\} \tag{1}$$

2. Reflexive, transitive closure: $(\lhd_G^*) = (\lhd_G^*)^*$.

$$G \quad\leadsto\quad G \cup \{x \lhd^* x\} \tag{2}$$

$$G\{x \lhd^* y, y \lhd^* z\} \quad\leadsto\quad G \cup \{x \lhd^* z\} \tag{3}$$

3. All nodes descend from the root.

$$G\{\mathsf{root}(x)\} \leadsto G \cup \{x \lhd^* y\} \quad (y \in V_G) \tag{4}$$

3.2 Merging

Next, we consider rules that merge nodes based on local properties of the graph.

1. The $\lhd^*$ relation is antisymmetric:

$$G\{x \lhd^* y, y \lhd^* x\} \quad\leadsto\quad G[x := y] \tag{5}$$

2. A leaf is maximal with respect to $\lhd^*$:

$$G\{\mathsf{leaf}(x), x \lhd^* y\} \leadsto G[x := y] \tag{6}$$

3. If two nodes have the same child, they are equal:

$$G\{x \lhd z, y \lhd z\} \leadsto G[x := y] \tag{7}$$

3.3 Consistency

We next introduce rules that check that the graph is locally consistent.

1. Inequality is irreflexive:

$$G\{x \not\approx x\} \leadsto \bot \tag{8}$$

2. A node has at most one label:

$$G\{\mathsf{lab}_a(x), \mathsf{lab}_b(x)\} \leadsto \bot \qquad (a \neq b) \tag{9}$$

3.4 Search

All of the rules so far are don't-care nondeterministic: no matter which order they are applied, we will arrive at a graph that is equisatisfiable to the original graph. (In fact, we can prove this directly by extending graphs to maintain an equivalence relation on the variables that have been merged, and showing that the rules above are confluent up to equivalence on these graphs. We omit the details.)

If the graph has been normalized under all of the above rules, then we know that it is acyclic, locally consistent, has at most one root with no parent, and so on. However, the constraint graph may still not be satisfiable. We wish to reduce the problem to a solved form that has a satisfiable subtree that can be extracted easily (in polynomial time). To obtain a solved form, it is helpful to first resolve all ambiguities about relationships between nodes that have a common descendant.

There are two kinds of situations that require nondeterministic guessing. First, if $x \lhd^* z, y \lhd^* z \in G$, and we do not know whether $x \lhd^* y$ or $y \lhd^* x$ holds, then (since x, y, z must all be on a common path from the root) there are three possibilities: either x comes before y, y comes before x or x and y are equal. Then we branch as follows:

$$G\{x \lhd^* z, y \lhd^* z\} \quad\leadsto\quad G[x := y] \tag{10}$$

$$G\{x \lhd^* z, y \lhd^* z\} \quad\leadsto\quad G \cup \{x \not\approx y, x \lhd^* y\} \tag{11}$$

$$G\{x \lhd^* z, y \lhd^* z\} \quad\leadsto\quad G \cup \{x \not\approx y, y \lhd^* x\} \tag{12}$$

The right hand sides are mutually exclusive, so if more than one rule applies, we may need to backtrack.

Second, if $x \lhd^* z, y \lhd z \in G$, then it is possible that x and z need to be merged in order to satisfy G, or for x to strictly precede y, but not for x to be strictly between y and z. Thus, we branch using the two rules:

$$G\{x \lhd^* z, y \lhd z\} \quad\leadsto\quad G[x := z] \tag{13}$$

$$G\{x \lhd^* z, y \lhd z\} \quad\leadsto\quad G \cup \{x \not\approx z, x \lhd^* y\} \tag{14}$$

Again, if more than one rule applies, we may need to backtrack — these rules are don't-know nondeterministic.

3.5 Additional rules

The above rules suffice to show correctness. We can also include some additional rules that may enable us to determine unsatisfiability more quickly than by using the bottom-up search rules:

$$\begin{aligned}
G\{x \lhd y, \mathsf{root}(y)\} &\quad\leadsto\quad \bot \\
G\{\mathsf{leaf}(x), x \lhd y\} &\quad\leadsto\quad \bot \\
G\{x \lhd y, x \lhd z, y \lhd^* w, z \lhd^* w\} &\quad\leadsto\quad G[y := z] \\
G\{x \lhd y, x \lhd^* z, y \lhd^* w, z \lhd^* w\} &\quad\leadsto\quad G[x := z] \\
G\{x \lhd y, x \lhd^* z, y \lhd^* w, z \lhd^* w\} &\quad\leadsto\quad G \cup \{y \lhd^* z, x \not\approx z\}
\end{aligned}$$

The first few rules are safe to apply eagerly while the last two rules may require backtracking.

3.6 Correctness

A graph G which cannot be reduced by any of the above rules is called a normal form. Correctness means showing that the algorithm terminates (in nondeterministic polynomial time), is complete (reduces every solvable problem to a normal form), and is sound (every consistent normal form is satisfiable). Soundness is the hard part for this style of algorithm, since we wish to avoid explicit enumeration of subtrees as much as possible.

Termination.

First, we show that the unordered algorithm terminates. It suffices to show that rules (1)–(14) terminate, when applied in an arbitrary order.

THEOREM 2 (TERMINATION). *Given an acyclic graph G, any sequence of applications of the rules (1)–(14) terminates.*

PROOF. The appropriate termination measure is $\mu(G) = (|V|, |V|^2 - |\lhd_G^*|)$, where we define $\mu(\bot) = (0, 0)$. The rules involving $\bot$ are trivially terminating. We order pairs lexicographically, and show that each rule decreases the measure. The saturation and search rules (if applicable) always increase the number of $\lhd^*$-edges and preserve the number of vertices, while the merging rules (if applicable) always decrease the number of vertices. □

Completeness.

Completeness means that if a graph is satisfiable, then it can be rewritten to a satisfiable normal form.

Lemma 1. *If $T \models G$ is satisfiable and not in normal form, then there is a rule instance $G \rightsquigarrow G'$ such that $T \models G'$.*

Proof. Suppose $T \models G$ and G is not normalized. Then some rule applies to G. If the rule is one of the deterministic rules (1)–(9), then $T \models G'$ because these rules preserve satisfiability. If one of rules (13) or (14) apply, then there exist $x \lhd^* z, y \lhd z \in G$. If x and z are assigned to the same node in T, then apply rule (13); the resulting $G' = G[x := z]$ is still satisfiable by T. Otherwise, $x \not\approx y$ and $x \lhd^* y$ must already hold in T, so $T \models G \cup \{x \not\approx y, x \lhd^* y\}$. The reasoning for rules (10)–(12) is similar, but we must observe that if $x \not\approx y$ in T and x, y have a common descendant then either $x \lhd^* y$ or $y \lhd^* x$ holds in T. □

Soundness.

By inspection, a normal form must be locally consistent (each node with at most one label, no parent of the root, no child of a leaf). It must also have no $\lhd^*$-cycles, other than the trivial reflexive edges $x \lhd^* x$. Finally, the search rules suffice to ensure that there is a unique choice of parent among the ancestors of each node. This is enough to ensure that there is a tree T that satisfies G with no further need for merging nodes or backtracking. We can (efficiently) extract this tree as follows.

We first show that G can be totally ordered by a relation $\ll_G$ that separates all nodes of G (and in particular, guarantees that all inequality constraints in G are satisfied):

Lemma 2. *If G is normal form, then there exists a total order $\ll_G$ such that if $x \lhd^* y \in G$ and $x \neq y$ then $x \ll_G y$.*

Proof. Since $\lhd^*$ has no nontrivial cycles, define $\ll_G$ as a topological sort of $\lhd_G^* \setminus \{(x,x) \mid x \in V_G\}$. □

The total ordering $\ll_G$ imposes stronger constraints on the ordering of the nodes in G than G itself does: for example, it forces nodes $x \lhd^* y$ to be distinct even though G might permit them to be equal. This suffices as long as $\ll_G$ is still consistent with *some* satisfying tree for G, as we will now show.

Fix $v_1 \ll_G \ldots \ll_G v_n$ to be the increasing sequence of elements of V_G. If G contains a root constraint, then v_1 must be the root, hence an ancestor of every other node. Whether or not G contains a root constraint, no node is a strict ancestor of v_1, so we can safely take v_1 to be the root of the satisfying tree we will build. For the other nodes, we define the set of *ancestors* of v in G to be the set

$$anc_G(v) = \{v_1\} \cup \{w \ll_G v \in V_G \mid w \lhd^* v \in G\} .$$

Note that we really do want to add v_1, not v, to ensure that every node besides v_1 has at least one ancestor. This set is totally ordered by $\ll_G$, so we define the *parent* of v in G (written $par_G(v)$) to be the $\ll_G$-maximum element of G, $\max anc_G(v)$.

Lemma 3. *Assume $v_1 \ll_G v$. Then if $w \lhd^* v \in G$ and $w \neq v$ then $w \lhd^* par_G(v) \in G$. Moreover, if $w \lhd v \in G$ then $par_G(v) = w$.*

Proof. For the first part, suppose $v_1 \ll_G v$ and $w \lhd^* v \in G$ where $w \neq v$. Then since both $par_G(v) \lhd^* v \in G$, and G is normalized, we must have either $w \lhd^* par_G(v) \in G$, or $par_G(v) \lhd^* w \in G$. In the first case we are done; otherwise, since $par_G(v)$ is the $\ll_G$-maximal element of $anc_G(v)$, and $w \in anc_G(v)$, we have $par_G(v) = w$, which implies $w \lhd^* par_G(v)$.

For the second part, since $w \lhd v$ we know that $w \lhd^* v$ and $w \neq v$. This implies that $w \lhd^* par_G(v)$ by the first part. Moreover, since $par_G(v) \lhd^* v \in G$ and the two are not equal we also know that $par_G(v) \lhd^* w \in G$. Since G is in normal form, we must therefore have $par_G(v) = w$. □

We can now construct a satisfying tree. For each i, we define T_i as $(V_i, E_i, v_1, \lambda_i, <_i)$ where

1. $V_i = \{v_1, \ldots, v_i\}$
2. $E_1 = \emptyset$
3. $E_{i+1} = E_i \cup \{(par_G(v_i), v_i)\}$
4. $\lambda_i : V_i \to \Sigma$ is $\lambda|_{\{v_1,\ldots,v_i\}}$, where λ is a total labeling satisfying $\lambda(v) = a$ whenever $\mathsf{lab}_a(v) \in G$, otherwise arbitrary.
5. $<_i$ is a preorder traversal of T_i such that if $v \ll_G w$ are siblings in T_i the $v \leq_i w$.

The labeling λ exists since G must have at most one node label for each edge.

Then T_n is a tree over V_G. We need to show that $T_n \models G$. We will show the stronger property that for each i, $T_i \models G_i$ where $G_i = G|_{\{v_1,\ldots,v_i\}}$, the restriction of G to formulas involving $\{v_1, \ldots, v_i\}$.

Lemma 4. *Let $T_1, \ldots, T_n$ be constructed as above. Then for each $i \in \{1, \ldots, n\}$, we have $T_i \models G_i$.*

Proof. Proof is by induction on i.

The base case is easy since G_1 has no irreflexive edges and by assumption, v_1 is the root of T and has the same label as specified in G (if any). Moreover, if $\mathsf{leaf}(v_1) \in G$ then $T \models \mathsf{leaf}(v_1)$.

For the inductive case, let formula $A \in G_{i+1}$ be given. First, suppose A is binary. It is easy to see that binary relations that were inherited from G_i are satisfied in T_i and are still satisfied in T_{i+1}. Thus, we need only consider new binary formulas that mention v_{i+1}. Suppose $w \to v \in G_{i+1}$ for some relation $(\to) \in \{\not\approx, \lhd, \lhd^*\}$. If $w \to v \in G_i$ then we are done since it still holds. If not, then one of w or v is v_{i+1}. If $w = v_{i+1}$, then by definition of $\ll_G$, we must have $v = v_{i+1}$, so clearly $\to$ cannot be $\not\approx$ or $\lhd$ since both are irreflexive; whereas $v_{i+1} \lhd^* v_{i+1} \in G_{i+1}$ since G is normalized. Now if $w \ll_G v_{i+1}$ then $v = v_{i+1}$ and there are several cases. If $(\to) = (\lhd)$, Lemma 3 implies that $par_G(v_{i+1}) = w$, hence $T_{i+1} \models par_G(v_{i+1}) \lhd v_{i+1}$. If $(\to) = (\lhd^*)$, then Lemma 3 applies again to show that $w \lhd^* par_G(v_{i+1}) \in G_{i+1}$. Since $par_G(v) \ll_G v_{i+1}$, we have

$w \lhd^* par_G(v_{i+1}) \in G_i$, so $T_i \models w \lhd^* par_G(v_{i+1})$. Hence $T_{i+1} \models w \lhd^* par_G(v_{i+1}) \lhd^* v_{i+1}$. Finally, if $(\rightarrow) = (\not\approx)$ then clearly $T_{i+1} \models w \not\approx v_{i+1}$ since $w \ll_G v_{i+1}$.

Now consider the unary formulas. Node labeling formulas still hold, and the root is still the root. For leaf, we need to be more careful. First, if $\mathsf{leaf}(v_{i+1}) \in G_{i+1}$ then clearly $T_{i+1} \models \mathsf{leaf}(v_{i+1})$ since we just added it as a leaf. If $\mathsf{leaf}(w) \in G_{i+1}$ for some other w then $\mathsf{leaf}(w) \in G_i$ so $T_i \models \mathsf{leaf}(w)$. The only way T_{i+1} could fail to satisfy $\mathsf{leaf}(w)$ is if w is the new parent of v_{i+1}. But in this case, G would not be in normal form since rule (6) would apply.

This completes the inductive case, so for all i we have $T_i \models G_i$. □

Theorem 3. *An unordered constraint graph in normal form is satisfiable.*

Proof. By the previous lemma, we construct $T_n \models G_n = G$. □

4. ORDERED CONSTRAINTS

In this section we show how to add support for following-sibling constraints. We sketch some steps towards also handling next-sibling and first/last constraints at the end of the section, but leave full proof of correctness for future work. We add the following rewriting rules:

1. The document order is reflexive, transitive, and antisymmetric:

$$G \rightsquigarrow G \cup \{x \preceq x\} \tag{15}$$
$$G\{x \preceq y, y \preceq x\} \rightsquigarrow G[x := y] \tag{16}$$
$$G\{x \preceq y, y \preceq z\} \rightsquigarrow G \cup \{x \preceq z\} \tag{17}$$

2. The following-sibling relation is reflexive, transitive, and antisymmetric:

$$G \rightsquigarrow G \cup \{x \twoheadrightarrow^* x\} \tag{18}$$
$$G\{x \twoheadrightarrow^* y, y \twoheadrightarrow^* x\} \rightsquigarrow G[x := y] \tag{19}$$
$$G\{x \twoheadrightarrow^* y, y \twoheadrightarrow^* z\} \rightsquigarrow G \cup \{x \twoheadrightarrow^* z\} \tag{20}$$

3. Containment:

$$G\{x \lhd^* y\} \rightsquigarrow G \cup \{x \preceq y\} \tag{21}$$
$$G\{x \twoheadrightarrow^* y\} \rightsquigarrow G \cup \{x \preceq y\} \tag{22}$$

4. Siblings have equal parents:

$$G\{p \lhd z, y \twoheadrightarrow^* z\} \rightsquigarrow G \cup \{p \lhd y\} \tag{23}$$

5. Siblings are totally ordered:

$$G\{x \twoheadrightarrow^* z, y \twoheadrightarrow^* z\} \rightsquigarrow G[x := y] \tag{24}$$
$$G\{x \twoheadrightarrow^* z, y \twoheadrightarrow^* z\} \rightsquigarrow G \cup \{x \twoheadrightarrow^* y, x \not\approx y\} \tag{25}$$
$$G\{x \twoheadrightarrow^* z, y \twoheadrightarrow^* z\} \rightsquigarrow G \cup \{y \twoheadrightarrow^* x, x \not\approx y\} \tag{26}$$

6. Siblings share ancestors:

$$G\{x \lhd^* z, y \twoheadrightarrow^* z\} \rightsquigarrow G[x := z] \tag{27}$$
$$G\{x \lhd^* z, y \twoheadrightarrow^* z\} \rightsquigarrow G \cup \{x \lhd^* y, x \not\approx z\} \tag{28}$$

7. The document order is a preorder traversal:

$$G\{x \lhd^* y, z \preceq y\} \rightsquigarrow G \cup \{z \preceq x\} \tag{29}$$
$$G\{x \lhd^* y, z \preceq y\} \rightsquigarrow G \cup \{x \lhd^* z\} \tag{30}$$

Rules (15)–(23) can be applied eagerly without backtracking, since they preserve satisfiability (we can check confluence by instrumenting the graphs with equivalence relations, as before). The remaining rules makes a nontrivial choice, so may introduce the need for backtracking.

4.1 Additional rules

There are some additional sound rules that may be helpful in pruning the search space, but are not strictly necessary for proving correctness. Here are a few such rules with a top-down flavor that can be added (without affecting termination, soundness or completeness):

$$\begin{aligned}
G\{x \lhd^* y, x \twoheadrightarrow^* y\} &\rightsquigarrow G[x := y] \\
G\{x \twoheadrightarrow^* y, p \lhd x\} &\rightsquigarrow G \cup \{p \lhd y\} \\
G \cup \{x \twoheadrightarrow^* y, x \lhd^* z, y \lhd^* z\} &\rightsquigarrow G[x := y] \\
G\{p \lhd^* x, x \twoheadrightarrow^* y\} &\rightsquigarrow G[p := x] \\
G\{p \lhd^* x, x \twoheadrightarrow^* y\} &\rightsquigarrow G \cup \{p \lhd^* y, p \not\approx x\}
\end{aligned}$$

Again, the last two may require backtracking while the first three are meaning-preserving.

4.2 Correctness

Termination and completeness hold by essentially the same arguments as before, observing that the new rewriting rules either decrease the number of vertices or increase the number of edges of G, and cover all cases.

The soundness argument is similar to before, but requires a stronger total ordering lemma:

Lemma 5. *If G is normal form, then there exists a total order $\ll_G$ satisfying:*

($\ll_1$) *If $x \preceq y \in G$ and $x \neq y$ then $x \ll_G y$.*

($\ll_2$) *If $x \lhd^* y \in G$ and $x \neq y$ then $x \ll_G y$.*

($\ll_3$) *If $x \twoheadrightarrow^* y \in G$ and $x \neq y$ then $x \ll_G y$.*

Proof. As with the earlier ordering, this is straightforward since the union of the irreflexive parts of the relations $\preceq, \lhd^*, \twoheadrightarrow^*$ is acyclic. Since $\lhd^*$ and $\twoheadrightarrow^*$ are contained in $\preceq$, it suffices to choose $\ll_G$ to be a topological sort of $\preceq \setminus \{(x, x) \mid x \in V_G\}$. □

Again, we take $v_1, \ldots, v_n$ to be an enumeration of V_G in increasing order of $\ll_G$. Now, to identify the parent of each node, we need to take the sibling ordering into account. For example, a formula such as $x \lhd^* y, y \twoheadrightarrow^* w$ normalizes easily, but its normal form does not contain an edge from x to w, so the previous definition of anc_G and par_G will not work. Instead, we need to include the ancestors of (previous) siblings among the candidate ancestors of a node. For v among $v_2, \ldots, v_n$ we define:

$$\begin{aligned}
anc_G(v) &= \{v_1\} \cup \{w \ll_G v \mid \exists x \neq w.\ w \lhd^* x \in G \\
&\qquad\qquad \text{and } x \twoheadrightarrow^* v \in G\} \\
par_G(v) &= \max_{\ll_G} anc_G(v)
\end{aligned}$$

Note that again, we really do want to add v_1, to ensure that every node has at least one ancestor (v_1 will always be the root). Also, the set of ancestors of v is linearly ordered by $\ll_G$ so the maximum is well-defined. We must now show that the definitions of ancestors and parent have the properties that will be needed in the proof.

LEMMA 6. *Let G be a normalized graph, and let $\ll_G$ and par_G be as defined above. Then:*

1. *If $v \ll_G w$ and $v \lhd^* w \in G$, then $v \lhd^* par_G(w) \in G$.*
2. *If $v \lhd w \in G$, then $v = par_G(w)$.*
3. *If $v \ll_G w$ and $v \twoheadrightarrow^* w \in G$, then $par_G(v) = par_G(w)$ (and both are defined).*

PROOF. For part (1), assume $v \ll_G w$ and $v \lhd^* w \in G$. Then clearly $w \twoheadrightarrow^* w \in G$ so $v \in anc_G(w)$. Since $v \lhd^* w$ and $par_G(w) \lhd^* w$, and since G is normalized, we must have either $v = par_G(w)$ or $v \lhd^* par_G(w) \in G$ or $par_G(w) \lhd^* v$. In the first two cases we are done; in the third case we must actually have $v = par_G(w)$ since $par_G(w)$ is maximal.

For part (2), assume $v \lhd w \in G$. Then by the first part, $v \lhd^* par_G(w) \in G$. By definition, we have $x \neq par_G(w)$ such that $par_G(w) \lhd^* x$ and $x \twoheadrightarrow^* w$. Since $v \lhd w$ holds and G is normalized, we must have $v \lhd x$ since otherwise rule (23) would apply. Thus, since $par_G(w) \neq x$, we must have $par_G(w) \lhd^* v$ since otherwise rules (13)–(14) would apply. Hence, by antisymmetry, $v = par_G(w)$.

For part (3), assume $v \ll_G w$ and $v \twoheadrightarrow^* w \in G$. Then neither v nor w can be the root, so $par_G(v)$ and $par_G(w)$ are well-defined. We will show that $anc_G(v) = anc_G(w)$, hence $par_G(v) = par_G(w)$. First, to show that $anc_G(v) \subseteq anc_G(w)$, let $p \neq x$ be given with $p \lhd^* x \in G$ and $x \twoheadrightarrow^* v \in G$. Then $x \twoheadrightarrow^* w$ so $p \in anc_G(w)$. Conversely, let $q \neq y$ with $q \lhd^* y \twoheadrightarrow^* w$ be given. Then since G is normalized, either $y \twoheadrightarrow^* v$ or $v \twoheadrightarrow^* y$. If $y \twoheadrightarrow^* v$ then $q \in anc_G(v)$. Otherwise, $v \twoheadrightarrow^* y \in G$ implies that $q \lhd^* v \in G$, since otherwise G would not be normalized with respect to rules (27)–(28). Consequently, again we can conclude $q \in anc_G(v)$. Since $anc_G(v) = anc_G(w)$, clearly $par_G(v) = par_G(w)$. □

LEMMA 7. *Let $T_1, \dots, T_n$ be constructed from the new version of par_G as before. Then for each $i \in \{1, \dots, n\}$, we have $T_i \models G_i$.*

PROOF. The proof is similar to the unordered case. The base case is easy to adjust since the root is not a sibling of any other node. We give the reasoning for the inductive cases involving the new document-order and following-sibling constraints only.

If $w \twoheadrightarrow^* v \in G_{i+1}$ is in G_i, then it is true in T_i and still true in T_{i+1}. If it is not already in G_i, then it must mention v_{i+1}. First suppose $w = v_{i+1}$. Then $v = v_{i+1}$ also, and clearly $T_{i+1} \models v_{i+1} \twoheadrightarrow^* v_{i+1}$. Otherwise, $w \ll_G v_{i+1}$ and $v = v_{i+1}$. By Lemma 6, we know that since $w \twoheadrightarrow^* v_{i+1} \in G_{i+1} \subseteq G$ and $w \ll_G v_{i+1}$, we have $par_G(w) = par_G(v)$. Thus, $T_{i+1} \models v \twoheadrightarrow^* w$.

If $w \preceq v \in G_{i+1}$, then a similar case analysis shows that if $w = v_{i+1}$ then $v = v_{i+1}$ and $v_{i+1} \preceq v_{i+1}$ holds in T_{i+1}. Otherwise, $w \ll_G v_{i+1} = v$ clearly implies $T_{i+1} \models w \preceq v_{i+1}$. Note that $\ll_G$ restricted to $\{v_1, \dots, v_{i+1}\}$ is a valid preorder traversal of each T_{i+1} by normalization of G under (29) and (30). □

4.3 First, last and next sibling constraints

It is possible to extend this approach to handle first child, last child, and next-sibling constraints, using the following additional rules:

1. The first or last sibling precedes/follows all others:

$$G\{\mathsf{fst}(x), p \lhd x, p \lhd y\} \leadsto G \cup \{x \twoheadrightarrow^* y\} \quad (31)$$
$$G\{\mathsf{lst}(y), p \lhd x, p \lhd y\} \leadsto G \cup \{x \twoheadrightarrow^* y\} \quad (32)$$

2. Adjacent siblings are siblings and not equal:

$$G\{x \twoheadrightarrow y\} \leadsto G \cup \{x \twoheadrightarrow^* y, x \not\approx y\} \quad (33)$$

3. Previous and next siblings are unique.

$$G\{x \twoheadrightarrow y, x' \twoheadrightarrow y\} \leadsto G[x := x'] \quad (34)$$
$$G\{x \twoheadrightarrow y, x \twoheadrightarrow y'\} \leadsto G[y := y'] \quad (35)$$

4. Siblings are totally ordered:

$$G\{x \twoheadrightarrow^* z, y \twoheadrightarrow z\} \leadsto G[x := z] \quad (36)$$
$$G\{x \twoheadrightarrow^* z, y \twoheadrightarrow z\} \leadsto G \cup \{x \twoheadrightarrow^* y, x \not\approx z\} \quad (37)$$

These rules are complete, terminating, and appear able to deal with most common cases, but difficult to prove sound by extending the previous argument. The problem is that the current definition of $\ll_G$ and par_G may not respect the additional constraints posed by first, last or next sibling. For example, if $p \lhd^* x, p \lhd^* y, \mathsf{fst}(x), \mathsf{fst}(y)$ then the existing definitions would take p to be the parent of both x and y, but they cannot both be first in a tree obtained by removing edges from this graph. Similarly, if the problem is of the form $p \lhd^* x, x \twoheadrightarrow y, p \lhd^* z$ then nothing stops us from taking the children of p to be $x \ll_G z \ll_G y$, but this violates the next-sibling constraint.

To make this work, it appears sufficient to construct T_{i+1} in such a way that if v_{i+1} has a parent assigned to it already in G, we use it, whereas if v_{i+1} is not connected to its nearest ancestor by a child edge, then we insert a surrogate parent between v_{i+1} and its closest ancestor. This would prevent both of the above problems, by ensuring that first, last, or next-sibling constraints are never mixed. However, the reasoning is delicate and so we leave establishing the soundness of (perhaps some extension of) the above rules for future work.

5. BINARY INTERSECTION

We now consider some special cases of intersection for Positive Navigational XPath expressions (henceforth, just XPath). In terms of conjunctive constraint graphs, a binary intersection is formed by equating two linear paths at their beginning and end points only. Such a problem is essentially the same as a constraint graph that is an undirected cycle (that is, each node has indegree and outdegree equal to 1).

Intersection for arbitrary XPath expressions over downward axes is NP-complete, and even satisfiability of a single expression with downward steps and upward filters is NP-complete. However, the satisfiability of binary intersection of downward XPath can be tested in quadratic time [17], and for simple XPath expressions the exact complexity is not known. By "simple" we mean expressions that are sequences of axis steps and node tests without any filters. In this section we show that binary (and k-ary, for fixed k) intersection remains tractable for XPath expressions involving only forward steps (including *downward filters*, that is, filters only involving forward steps), and we show that binary intersections of simple XPath expressions, or expressions with downward filters, are tractable if we fix the number of changes in direction in the problem.

Simple XPath expressions can be translated to conjunctive formulas in the standard way, yielding formulas whose graph is a linear sequence $p = x_1 \to_1 \cdots \to_n x_n$ (together with appropriate node annotations). The *binary (or* *k-ary) intersection problem* is the problem of determining whether the formula formed by equating the beginning and end points of two (or respectively k) paths is satisfiable. Note that for binary intersection this is equivalent to the problem of determining whether a closed loop

$$q = x_0 R_0 \cdots R_{n-1} x_n R_n x_0$$

is satisfiable, that is, whether the intersection $p \cap \epsilon$ is satisfiable.

We first discuss the special cases where only downward steps or forward steps are used, and then extend the analysis to handle arbitrary paths.

Downward problems.

For downward-only paths (including downward filters), intersection can be tested in quadratic time, as shown by Hammerschmidt et al. [17]. The idea of their algorithm is first to note that downward filters can be pruned (since they are always independently satisfiable), and test whether the main branches of the two path expressions are simultaneously satisfiable along a linear path by representing the path expressions as finite automata and testing emptiness of their intersection. More generally, the same technique can be used to show that k-ary downward intersection problems can be solved in $O(n^k)$ if the maximum path length is n.

Forward problems.

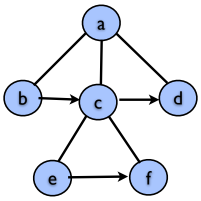
By a *forward path* we mean one that only uses forward steps ($\lhd$, $\lhd^*$, $\twoheadrightarrow$, $\twoheadrightarrow^*$), implicitly starts at the root, and may use label predicates (but no explicit root, first, last or leaf constraints). For convenience, write such a path p as $l_1 \to_1 \cdots \to_{n-1} l_n$, where each l_i is either in Σ or a wildcard $*$, and each $\to_i$ is a forward step $\lhd$, $\lhd^*$, $\twoheadrightarrow$. or $\twoheadrightarrow^*$. We will show that binary intersections of such paths can be solved in time $O(n^2)$, and more generally, k-ary intersections can be solved in time $O(n^k)$.

Given a path $p = x_1 \to_1 \cdots \to_{n-1} x_n$, let $\mathbf{C}$ and $\mathbf{S}$ be two new symbols not in Σ. We will use strings over $\Sigma \cup \{\mathbf{C}, \mathbf{S}\}$ to describe traversals of trees T in child-sibling form. These traversals (and their describing strings) will be called *tours* for brevity. First, let $\Sigma^{(=)}$ be the regular expression $a_1^*|\cdots|a_n^*$ where $\Sigma = \{a_1, \ldots, a_n\}$. That is, $\Sigma^{(=)}$ consists of arbitrary repetitions of a single symbol in Σ. Now define the set of valid tours as $\mathit{Tours} = \Sigma^{(=)}((\mathbf{C}\Sigma^{(=)})(\mathbf{S}\Sigma^{(=)})^*)^*$.

Essentially, the idea is that a tour represents a traversal of that part of a tree needed to witness the satisfiability of a forward path. This is not just a path in T, but may include siblings visited by following-sibling or next-sibling steps. A tour may also include zero or more symbols representing the node label visible at a given node. Thus, a tour is like a path, but meanders and makes (possibly redundant) node label observations rather than going straight to its destination.

For example, Figure 3 shows a tree along with regular expressions describing all its tours.

Given a string $s \in \mathit{Tours}$, define a tree T_s as follows. If s is in $\Sigma^{(=)}$, then define T to be a single-node tree whose label is consistent with the node label determined by s. Otherwise, s must be of the form $\Sigma^{(=)}\mathbf{C}\Sigma^{(=)}(\mathbf{S}\Sigma^{(=)})^*s'$, where s' is a tour. Suppose in particular that there are n occurrences of $\mathbf{S}$ in the prefix of s before s'. Then construct $T_{s'}$ and build a new tree T_s extending $T_{s'}$ whose root and first n children are consistent with the node labels in the beginning of s, and whose last child is the root of $T_{s'}$. Let $\mathit{tours}(T)$ be the set of all strings $s \in \mathit{Tours}$ such that T_s is a subtree of T. Note that this can be done consistently no matter how we split the $\Sigma^{(=)}$ substrings, since they can contain at most one distinct node label.



a^*, $a^*\mathbf{C}b^*$, $a^*\mathbf{C}c^*$, $a^*\mathbf{C}d^*$
$a^*\mathbf{C}b^*\mathbf{S}c^*$, $a^*\mathbf{C}c^*\mathbf{S}d^*$
$a^*\mathbf{C}c^*\mathbf{C}e^*$, $a^*\mathbf{C}c^*\mathbf{C}f^*$
$a^*\mathbf{C}b^*\mathbf{S}c^*\mathbf{S}d^*$
$a^*\mathbf{C}b^*\mathbf{S}c^*\mathbf{C}e^*$,
$a^*\mathbf{C}b^*\mathbf{S}c^*\mathbf{C}f^*$,
$a^*\mathbf{C}b^*\mathbf{S}c^*\mathbf{C}e^*\mathbf{S}f^*$

Figure 3: Tours

We define a regular expression $\mathit{reg}(p)$ over language $\Sigma \cup \{\mathbf{C}, \mathbf{S}\}$ as follows:

$$\begin{aligned}
\mathit{reg}(a) &= a^* \\
\mathit{reg}(*) &= \Sigma^{(=)} \\
\mathit{reg}(p \lhd q) &= \mathit{reg}(p) \cdot \mathbf{C} \cdot \Sigma^{(=)} \cdot (\mathbf{S} \cdot \Sigma^{(=)})^* \cdot \mathit{reg}(q) \\
\mathit{reg}(p \lhd^* q) &= \mathit{reg}(p) \cdot (\mathbf{C} \cdot \Sigma^{(=)} \cdot (\mathbf{S} \cdot \Sigma^{(=)})^*)^* \cdot \mathit{reg}(q) \\
\mathit{reg}(p \twoheadrightarrow q) &= \mathit{reg}(p) \cdot \mathbf{S} \cdot \Sigma^{(=)} \cdot \mathit{reg}(q) \\
\mathit{reg}(p \twoheadrightarrow^* q) &= \mathit{reg}(p) \cdot (\mathbf{S} \cdot \Sigma^{(=)})^* \cdot \mathit{reg}(q)
\end{aligned}$$

Because of reflexive steps, $\mathit{reg}(p)$ can contain strings that are not tours. For example, $\mathit{reg}(a \lhd^* b)$ contains ab, which does not correspond to a consistent node labeling. However, we can filter these nonsensical strings out using the regular expression *Tours*. These strings correspond to trees that satisfy the structural part of p but not its node labeling constraints.

LEMMA 8. *A tree T matches forward path p if and only if $\mathit{reg}(p) \cap \mathit{tours}(T) \neq \emptyset$.*

PROOF. For the first part, we show by induction on p that if p is satisfiable in T then it is satisfiable on a subtree T_s generated by some tour s that matches $\mathit{reg}(p)$.

For the second part, if $\mathit{reg}(p) \cap \mathit{tours}(T) \neq \emptyset$, then let s be some tour in the intersection. We show by induction that p matches T_s, which is a subtree of T. Since forward queries are monotone, p matches T. □

LEMMA 9. *An intersection of forward paths $p_1 \cap \cdots \cap p_k$ is satisfiable if and only if $\mathit{reg}(p_1) \cap \cdots \cap \mathit{reg}(p_k) \cap \mathit{Tours} \neq \emptyset$.*

PROOF. We need to show that if $p_1 \cap \cdots \cap p_k$ is satisfiable if and only if there exists an $s \in \mathit{reg}(p_1) \cap \cdots \cap \mathit{reg}(p_k) \cap \mathit{Tours}$ such that $p_1 \cap \cdots \cap p_k$ is satisfied on T_s. The reverse direction is immediate. For the forward direction, consider a tree T satisfying the intersection (at some node v). Each p_i is satisfied on some tour of T ending at v. Take the union of all of these tour subtrees (or, just take the largest tour ending at v, that always visits the first child whenever it takes a child step). Let s be the string defining this tour. By monotonicity, T_s will satisfy all of the p_i, so by the previous lemma, each p_i will contain s. □

THEOREM 4. *Satisfiability of the intersection of k forward path expressions (with or without downward filters) of size n can be determined in $O(n^k)$ time.*

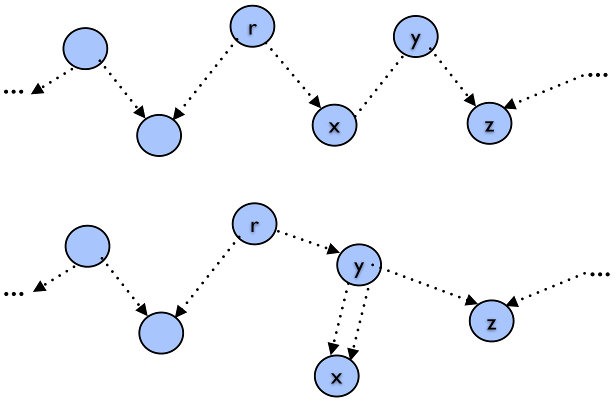


Figure 4: The simplified, alternating form of a simple intersection problem, and one merging step. Here, dotted lines stand for directed paths, not edges.

Proof. By the same reasoning as in Hammerschmidt et al. [17], it suffices to ignore downward filters since they are always satisfiable and can be handled independently of the rest of the problem. By the previous lemma, we construct the regular expressions $reg(p_1), \ldots, reg(p_k)$ describing the tours of trees satisfying $p_1, \ldots, p_k$. We test (by the standard product construction) whether the intersection $reg(p_1) \cap \cdots \cap reg(p_k) \cap Tours$ is simultaneously satisfiable. Since the size of the finite automaton generating *Tours* is independent of k or n, the total time needed is $O(n^k)$. □

Alternating problems.

We now consider the structure of arbitrary simple intersection problems. Such a problem is essentially a cyclic graph, if we ignore the direction of the edges. An *inflection point* is a node on the cycle whose indegree or outdegree is two; if the indegree is two then it is called a *local maximum* and if the outdegree is two then it is called a *local minimum.* An intersection problem is a cycle that alternates between forward and backward subpaths, separated by alternating inflection points. The *degree of alternation* of an intersection problem is the number of local minima in it. Since minima and maxima alternate, this is the same as the number of local maxima.

Lemma 10. *If there are more than two inflection points, then the problem can be either solved, or simplified to an equivalent form where the root is a local minimum, in linear time.*

Proof. This can be done by merging each node directed into the root with the root, and checking local satisfiability, until none are left or the graph becomes unsatisfiable. If the root is not a local minimum in the final graph, then the graph has been collapsed to a single satisfiable root node. Otherwise, return the simplified graph. □

Theorem 5. *Given an intersection problem with an explicit root node and alternation degree at most k, its satisfiability can be determined in time $T_k = O(n^{3k-1})$ for $k > 0$ (and linear time if $k = 0$).*

Proof. First, assume the problem is normalized as in the previous lemma, so that the root is a local minimum.

If there are just two inflection points, then the problem is a forward binary intersection so can be solved using the $O(n^2)$ algorithms above. This is $T_1 = O(n^{3-1})$.

If there are more than two inflection points, then there must be at least four. Suppose the cycle is of the form: $r \to^* x \leftarrow^* y \to^* z \cdots r$. (See also Figure 4). Since $r \lhd^* y \to^* x$, in any solution y must be located somewhere along the path from r to x (either equal to one of the nodes on that path, or between two adjacent nodes related by a $\lhd^*$-edge). There are at most $O(n)$ places where y can go, so we consider all possibilities such that the (binary, forward) intersection subproblem between y and x is satisfiable. If there are no such possibilities, then the whole problem cannot be satisfied, so we terminate and report unsatisfiability.

If we can insert y into the path from p to x such that the subproblem between y and x is satisfiable, then note that this subproblem can be handled independently of the rest of the problem since it involves only forward steps. Then, since y has no in-edges, and the part between y and x is satisfiable and independent of the rest of the problem, we can prune it, yielding a cycle of the form $r \to^* y \to^* z \cdots r$ which has one less alternation. Since y can be placed in at most $O(n)$ places along the path from r to x, and checking satisfiability between x and y takes $O(n^2)$ time, this takes $O(n^3)T_{k-1} = O(n^3 \cdot n^{3(k-1)-1}) = O(n^{3k-1})$. □

Moreover, if there is no root node we can simply try all k possibilities:

Corollary 1. *Any intersection problem with alternation degree k can be tested for satisfiability in time $O(kn^{3k-1})$.*

As with the downward and forward intersection cases, we can handle arbitrary downward filters by checking their satisfiability independently. We cannot lift the restriction to allow backward filters, since satisfiability even for single expressions with downward steps and downward/upward filters is already NP-complete [18]. On the other hand, it appears unproblematic to allow downward filters with single node identity constraints, since these yield independent subproblems that can also be solved independently of the main problem in $O(n^2)$ time. Moreover, it may be possible to refine the analysis to identify tractable special cases involving non-downward filters. This is left for future work.

6. DISCUSSION

In this paper we have considered two strategies for solving conjunctive constraints over trees. First, we investigated a constraint-solving approach that propagates simplifications first, then tries nondeterministic guessing to normalize the graph to a form from which we can extract a tree. If time is at a premium, we can speculatively simplify the graph first in the hope that it will be easy to see that it is satisfiable or unsatisfiable, then give up, or give the unsolved subproblems to an external solver. Further experimental work needs to be done to compare this approach with other applicable techniques, particularly reductions to SAT or SMT solvers.

Second, we investigated the special case where the problem is the intersection of two (simple, positive, navigational) XPath expressions. In this case we were able to identify a fixed-parameter tractability criterion, namely, the degree of alternation of the problem. In practice, most XPath intersection problems seem to involve small expressions with few changes in direction. Thus, this result shows that many common cases of XPath intersection are in PTIME, despite the NP-completeness of the general problem as shown by Hidders [18].

The two techniques exhibit different perspectives on conjunctive satisfiability problems over trees: a local view which manipulates syntactic patterns involving small numbers of constraints, and a global view that examines structural properties of the whole graph. They are complementary in the following sense: Given an intersection problem, we can first apply the simplifications of the constraint-satisfaction algorithm to decrease its degree of alternation. In particular, it appears possible to define a variant of the constraint-solving algorithm that maintains a sparse representation of the unsolved part of the problem, in particular retaining the simple cyclic structure of intersection problems. We have prototyped variations of this idea using Constraint Handling Rules [12], a logic programming formalism for constraint-solving algorithms. Conversely, given a constraint problem in this sparse form, we can look for independent subproblems that are instances of the intersection problem, and can thus be solved independently in polynomial time. Heuristics might be developed for the local constraint-solving algorithm to resolve join points that help factor the graph into intersection problems that can be solved more efficiently.

7. RELATED AND FUTURE WORK

As discussed in the introduction, Hidders [18] established that the problem we consider here is NP-complete and considered various special cases, some of which are in PTIME. Hammerschmidt et al. [17] studied the intersection problem for downward XPath, showing that it is solvable in quadratic time by a reduction to reachability for deterministic finite automata. Björklund et al. [7] studied the containment and satisfiability problems for conjunctive tree formulas, focusing on the complexity of subproblems where only certain axes are allowed to be used.

Lakshmanan et al. [20] investigated algorithms for tree pattern satisfiability, including tree patterns with node equality constraints (hence also intersection problems) and some schema constraints but not sibling or document ordering steps. Their approach is based on rewriting tree pattern queries to propagate node label information and eliminate ambiguities. They use additional predicates $OTSP(x, y)$ (x and y are on the same path) and $COUS(x, y)$ (x and y are cousins, that is, not on the same path). However, their algorithm (and its correctness proof) is not presented in full.

Olteanu et al. [23, 22] gives an algorithm for transforming XPath queries involving both forward and backward axes into (possibly exponentially many) queries involving just forward axes. This algorithm is also based on rewriting and has much in common with the one in this paper. However, Olteanu's algorithm is somewhat more complicated, since it treats the forward and backward versions of axes separately, whereas we translate both forward and reverse steps to the same formulas. Olteanu's algorithm has also mainly been applied to (acyclic) XPath query rewriting rather than satisfiability testing, and imposes some constraints on the structure of graphs, whereas our algorithm applies to arbitrary conjunctive constraint graphs.

Benedikt and Koch [6] present an alternative algorithm for converting arbitrary (positive) tree formulas to tree pattern queries. This algorithm can also be used for satisfiability testing, but it appears to be intended as an alternative proof of the prior expressiveness result established by Olteanu, not as a practical satisfiability algorithm. Benedikt and Koch's algorithm starts by guessing one of exponentially many preorder traversals of the vertices and then checking that the constraints are consistent with the guessed ordering. Since the first step of the algorithm has an exponential branching factor, it *always* takes exponential time to determine unsatisfiability, even in cases (e.g. a large irreflexive cycle) that our algorithm would determine unsatisfiable without any nondeterministic guessing.

In previous work, Benedikt and Cheney [4] employed a reduction from existential positive formulas over trees to existential constraints over the natural numbers, using standard Satisfiability Modulo Theories (SMT) solvers to solve the resulting formulas. This approach worked well for disjointness and independence analysis problems, but it is opaque and does not yield insight into the structure of the problem or reasons for poor performance in some cases. In contrast, the symbolic algorithm in this paper makes the structure of the problem clearer, and (at least indirectly) helped us gain insight into the binary intersection case.

It is possible to reduce satisfiability (and containment) to other problems, such as the monadic second-order logic over trees [13], or modal mu-calculus [14], for which standard decision procedures exist. As discussed in [4], we have experimented with the monadic second order logic approach, using the MONA tool [19]. Genevès et al. [14] present an XPath and XML Schema static analysis tool based on model-checking in a modal mu-calculus over binary trees. Our experience with a (preliminary) version of their solver and with MONA has been that performance is not competitive with techniques based on a reduction to SMT problems, but it is not clear whether this is due to engineering or algorithmic differences (since neither Genevès et al.'s solver nor competitive SMT solvers are available as source code).

In this paper we did not consider formulas with union (disjunction), negation or universal quantification, which are considered in several of the above approaches. These additions can make the satisfiability problem much harder; the general case of first-order satisfiability over trees is nonelementary [24]. Benedikt and Koch's survey [6] provides an excellent overview of the known complexity and expressiveness results for larger fragments of XPath.

We have also considered only satisfiability in the absence of a schema. There is a great deal of work on XPath satisfiability modulo a schema [5, 14, 20], largely establishing that the problem is at least as hard as the schema-free case, and sometimes much harder. Benedikt et al. [3] employ a SAT solver approach to test (un)satisfiability of downward XPath expressions modulo a schema. The most immediately relevant work is by Björklund et al. [8], who show that conjunctive tree query satisfiability modulo a schema is NP-complete for all interesting combinations of axes. That is, it is no better (and no worse) than the schema-free case. It may be possible to boost complete SAT solver techniques for satisfiability modulo a schema efficiently by using constraint-solving or global analyses to preprocess the problem.

One motivation for our symbolic approach has been to gain insight into the structure of the problem in a way that can be combined with so-called *theory-propagation* [21] techniques that have become popular in the Satisfiability Modulo Theories (SMT) community. The SMT community seems not to have investigated satisfiability modulo theories of trees. We plan to investigate the possibility of lifting our approach to solve existential first-order formulas over trees using the "splitting on demand" approach explored by Barrett et al. [2], possibly as an extension to existing SMT solvers. In order to interface with SMT solvers using theory-propagation and splitting on demand, we will need to augment constraint solving to handle negated atomic formulas and to produce (preferably small) witnesses to unsatisfiability.

8. CONCLUSIONS

We presented a symbolic algorithm for determining satisfiability of conjunctive formulas over trees, a generalization of the satisfiability and intersection problems for positive, navigational XPath. Although the problem it solves is NP-complete, our approach delays nondeterministic guessing until the local implications of constraints have been checked, so it can find witnesses to satisfiability and provide concise proofs of unsatisfiability quickly, without an exponential search. We investigated the structure of binary intersection problems for positive, navigational XPath with downward-only filters, and showed that forward k-ary intersection problems are solvable in polynomial time. We also identified the degree of alternation as a key factor in the complexity of binary intersection problems, and showed that binary intersection of simple navigational XPath expressions of bounded degree of alternation is tractable. The complexity of binary intersection for arbitrary (simple) XPath expressions remains open.

9. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in SAT modulo theories. In M. Hermann and A. Voronkov, editors, *LPAR*, volume 4246 of *Lecture Notes in Computer Science*, pages 512–526. Springer, 2006.

[3] M. Benedikt, A. Bonifati, S. Flesca, and A. Vyas. Verification of tree updates for optimization. In *CAV*, 2005.

[4] M. Benedikt and J. Cheney. Destabilizers and independence of XML updates. *PVLDB*, 3(1), 2010. To appear.

[5] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. *J. ACM*, 55(2):1–79, 2008.

[6] M. Benedikt and C. Koch. XPath leashed. *ACM Comput. Surv.*, 41(1):1–54, 2008.

[7] H. Björklund, W. Martens, and T. Schwentick. Conjunctive Query Containment over Trees. In *DBPL*, 2007.

[8] H. Björklund, W. Martens, and T. Schwentick. Optimizing conjunctive queries over trees using schema information. In *MFCS*, 2008.

[9] J. Clark and S. DeRose. XML path language (XPath) version 1.0. W3C recommendation, World Wide Web Consortium, 1999.

[10] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[11] B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[12] T. Frühwirth. *Constraint Handling Rules.* Cambridge University Press, 2009.

[13] P. Genevès and N. Layaïda. Deciding XPath containment with MSO. *Data Knowl. Eng.*, 63(1):108–136, 2007.

[14] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. In *PLDI*, 2007.

[15] G. Ghelli, K. Rose, and J. Siméon. Commutativity analysis for XML updates. *ACM TODS*, 33(4):1–47, 2008.

[16] G. Gottlob, C. Koch, and K. U. Schulz. Conjunctive queries over trees. *J. ACM*, 53(2):238–272, 2006.

[17] B. Hammerschmidt, M. Kempa, and V. Linnemann. On the Intersection of XPath Expressions. In *IDEAS*, 2005.

[18] J. Hidders. Satisfiability of XPath expressions. In *DBPL*, 2003.

[19] N. Klarlund and A. Møller. Mona v. 1.4 user manual. Technical Report BRICS NS-01-1, U. Aarhus, 2001.

[20] L. V. S. Lakshmanan, G. Ramesh, H. Wang, and Z. J. Zhao. On testing satisfiability of tree pattern queries. In M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, editors, *VLDB*, pages 120–131. Morgan Kaufmann, 2004.

[21] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to *DPLL*(T). *J. ACM*, 53(6):937–977, 2006.

[22] D. Olteanu. Forward node-selecting queries over trees. *ACM Trans. Database Syst.*, 32(1):3, 2007.

[23] D. Olteanu, H. Meuss, T. Furche, and F. Bry. Xpath: Looking forward. In *EDBT '02: Proceedings of the Worshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering-Revised Papers*, pages 109–127, London, UK, 2002. Springer-Verlag.

[24] L. Stockmeyer. The Complexity of Decision Problems in Automata Theory and Logic. Technical report, MIT, 1974.