Trees in trees: is the incomplete information about a tree consistent? *

Eryk Kopczyński

Institute of Informatics, University of Warsaw
erykk@mimuw.edu.pl

Abstract

We are interested in the following problem: given a tree automaton $\mathcal{A}$ and an incomplete tree description P, does a tree T exist such that T is accepted by $\mathcal{A}$ and consistent with P? A tree description is a tree-like structure which provides incomplete information about the shape of T. We show that this problem can be solved in polynomial time as long as $\mathcal{A}$ and the set of possible arrangements that can be forced by P are fixed. We show how our result is related to an open problem in the theory of incomplete XML information.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases XML, tree automata, incomplete tree descriptions, Euler cycle

Digital Object Identifier 10.4230/LIPIcs.CSL.2011.367

1 Introduction

In [2] and [9], the authors study the problem of incomplete data in relational databases, and classify the complexity of various computational problems associated with incompleteness, like consistency. These results have become very influential, and are now used for many practical applications with integration and exchange in relational databases [1, 8, 12].

But what about databases with more structure than a relational database, for example, XML documents [14]? Unlike a relational database, which is just a collection of tables, an XML document is ordered in a tree-like fashion. There has been some early work on incomplete information in XML [3, 10]. The paper [4] aims to provide a classification of problems associated with incompleteness for XML documents, like [2] and [9] did for relational databases.

Elements that can appear in XML documents are defined with DTDs [14]. A DTD (document type definition) is a set of declarations (a kind of a grammar) that define which elements may appear in an XML document, and how they are related. For example, consider a DTD for a database which describes the structure of employment in a company. Such a `company` might form a `group`; the description of each group might start with a `name` or not, followed by a description of a `leader` or not, followed by descriptions of `persons` employed in this group and smaller `groups` which are parts of the given group. The description of a `leader` consists of a description of a `person`. The description of a `person` consist of a `name`. The XML document below would be consistent with that DTD.

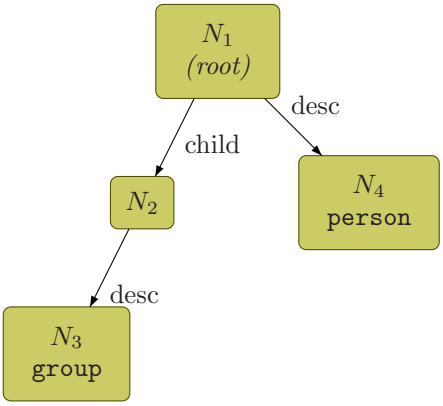
* Supported by the *Querying and Managing Navigational Databases* project realized within the Homing Plus programme of the Foundation for Polish Science, cofinanced by the European Union from the Regional Development Fund within the Operational Programme Innovative Economy ("Grants for Innovation").

Computer Science Logic 2011 (CSL'11).
Editor: Marc Bezem; pp. 367–380

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
<!DOCTYPE company [...]>
<company>
  <group>
    <name>Example Co.</name>
    <leader>
      <person><name>Smith</name></person>
      </leader>
    <group>
      <name>sales</name>
      <leader>
        <person><name>Baker</name></person>
        </leader>
      <person><name>Jones</name></person>
      </group>
    <group>
      <name>research</name>
      <person><name>Black</name></person>
      <person><name>White</name></person>
      </group>
    </group>
  </company>
```

An incomplete description of such a document might, for example, state that our document X contains four distinct nodes N_1, N_2, N_3, N_4 such that N_1 is the root, N_2 is a child of N_1, N_3 is a descendant of N_2, N_4 is a descendant of N_1, and N_3 is a `group` and N_4 is a `person`. The document above is consistent with this incomplete description (take N_1 to be the description of the company, N_2 to be the main group, N_3 to be the *research* group, and N_4 to be Black).



Note that this information is incomplete on several levels. It is possible that there are nodes about which we have no information at all (*open world assumption*). We don't know how many levels are between N_2 and N_3. In the assignment above, N_4 (Black) was a child of N_3 (research), although it was not explicitly stated in the description. In general, it would be also possible that N_2 could be a descendant of N_4—although our DTD forbids this, since a `person` cannot contain a `group` as a descendant.

[4] considers several types of such incomplete tree descriptions; we are interested in *incomplete DOM-trees* which enforce the mapping from the incomplete tree description to the tree to be injective, i.e., N_4 we know that is not equal to N_2 or N_3. Four *axes* are allowed to appear in the incomplete description: *next sibling*, *sibling*, *child*, *descendant*.

One of the problems investigated by [4] is the following:

▶ Problem 1.1. Let D be a fixed DTD. Given an incomplete tree description t, is there an XML document X which is consistent with both D and t?

Theorem 5.28 from [4] shows that this consistency problem is in PTIME for incomplete tree descriptions which do not use the *descendant* axis (only *child*, *sibling*, and *next sibling*). The case whether we can extend the result to also allow tree descriptions using this relation has been left open. Even the case where the tree description allows the *descendant* relation only on the topmost level and the shape of the descendant trees was completely fixed was an open question.

In this paper, we show how to solve this special case, and then show a polynomial algorithm for almost solving Problem 1.1. We say "almost", because there is a subtle difference between our and [4] understanding of when a tree is consistent with an incomplete description. However, we believe that our definition is also well motivated: although there are trees matching a given description according to [4] and not matching according to us, they are quite unnatural.

However, we prefer to work with theoretically more pure notions, rather than XML documents. Thus, instead of XML, we deal with binary trees with vertices labelled by elements of alphabet Σ, and instead of DTDs we deal with finite automata. The second change makes our results more general (each DTD corresponds to a finite automaton, but automata are more general than DTDs); another generalization is that we don't use the specific four axes listed above, but rather allow them to be defined using a regular expression. Still, we need to choose the axes from a fixed language $\mathcal{L}$ to obtain good algorithmic results. Our generalization from XML to generic trees also allows us to use our results for hierarchical data other than XML documents; for example, we could want to know whether a correct program in some programming language (or, more generally, a word in a context free language) exists that includes given keywords and symbols in given structural relationships; or whether there is an evolution of a branching process which exhibits given behaviors.

The paper is structured as follows. In Section 2 we provide the definitions required to understand the problem. In Section 3 we show how to solve the special case above in polynomial time (Theorem 3.2). This is used to explain techniques which are then used in Section 4 to solve the general problem for any tree descriptions (Theorem 4.3). In section 5 we show how to translate [4]'s problem from the XML world to the world of automata over binary trees, and point out the subtle difference that could not be solved with our methods.

For completeness, in Section 6 we show that the assumptions about the fixed size of automaton and the fixed set of languages cannot be lifted from Theorems 3.2 and 4.3. The problems become NP complete without these assumptions.

2 Preliminaries

An *unlabeled tree* is a finite $\tau \subset \{0,1\}^*$ such that whenever $uw \in \tau$, also $u \in \tau$. The empty word ϵ is the *root* of the tree, $w0$ and $w1$ are the *children* of w, and $w \in \tau$ is a *leaf* iff $w0, w1 \notin \tau$.

For a tree τ, by $\mathrm{port}(\tau)$ we denote the set of $x \in \{0,1\}^*$ such that $x \notin \tau$ and all proper prefixes of x are in τ. In other words, $\mathrm{port}(\tau)$ is the set of x such that $\tau \cup \{x\}$ is also a tree. The elements of $\mathrm{port}(\tau)$ are called *ports*; this name emphasizes our open world assumption – we don't treat these locations as places where the tree ends, but rather as variables: the tree can be extended ("grown") in arbitrary way from each port. In other words, our trees are similar to contexts used in the algebraic theory of trees ([5]), except that ports appear in every applicable location, not in a single one.

A *tree over* Σ is a function $T : \tau \to \Sigma$, where τ is an unlabeled tree. By $\mathrm{port}(T)$ we denote $\mathrm{port}(\mathrm{dom}(T))$.

Let T and U be two trees over Σ. We say that T is an *extension* of U ($T \subseteq U$) iff for each $u \in \mathrm{dom}(U)$ we have $u \in \mathrm{dom}(T)$ and $T(u) = U(u)$. We say that U is a *full subtree* of T at offset w iff for each u we have that $u \in \mathrm{dom}(U)$ iff $wu \in \mathrm{dom}(T)$, and $U(u) = T(wu)$. We say that U is an *inner subtree* of T at offset w iff an extension of U is a full subtree of T at offset w. If $U_1, U_2, \ldots, U_n$ are inner subtrees of T at offsets $w_1, w_2, \ldots, w_N$ respectively, we say that they are *disjoint inner subtrees* of T iff they share no common node, i.e., the sets $w_i\mathrm{dom}(U_i)$ are disjoint.

A (nondeterministic tree) *automaton* is a tuple $\mathcal{A} = (\Sigma, Q, q_I, \delta)$, where $q_0 \in Q$ and $\delta \subseteq \Sigma \times Q \times Q \times Q$.

Let T be a tree over Σ, and $\mathcal{A} = (\Sigma, Q, q_I, \delta)$ be an automaton. A *run* of $\mathcal{A}$ over T is a tree ρ over Q such that:

- $\mathrm{dom}(\rho) = \mathrm{dom}(T) \cup \mathrm{port}(T)$,
- For each $w \in \mathrm{dom}(T)$ we have $(T(w), \rho(w), \rho(w0), \rho(w1)) \in \delta$.

A run is *accepting* iff $\rho(\epsilon) = q_I$. We say that a tree T is *accepted* by $\mathcal{A}$ iff there is an accepting run of $\mathcal{A}$ on T. Sets of all trees accepted by some automaton $\mathcal{A}$ are called *regular languages* of trees, and can be equivalently defined using different kinds of automata, logic, algebra, etc.

This definition is a bit different from the usual one (e.g. [6]): in the usual definition, we would have a set of final states F, and force $\rho(x) \in F$ for each $x \in \mathrm{port}(T)$. As mentioned above, in our intentions (open world assumption) ports are places where the tree can extended in any way. If we want to make sure that our "incomplete" trees can be extended to trees which are accepted according to the usual definition, it is enough to assume that each state of $\mathcal{A}$ is productive, i.e., there is an accepting run on some tree which uses this state. Also, if we want to block some port x so that the tree cannot be extended there and the run ρ ends in a final state in this port, it is enough to add one extra character EOT to Σ and one extra state q_{EOT} to Q, and a transition $(\mathrm{EOT}, q, q_{\mathrm{EOT}}, q_{\mathrm{EOT}})$ for each $q \in F$. By putting $T(x) = \mathrm{EOT}$ we ensure that $\rho(x) \in F$ and $\rho(x0) = \rho(x1) = q_{\mathrm{EOT}}$, and since there is no transition with q_{EOT} on top, we can no longer extend the tree at $x0$ or $x1$ without losing acceptance. This allows us to simulate the usual definition with ours.

We will also use trees in their graph theoretical meaning. A *rooted tree* is a structure (V, E, v_R), where $E \subseteq V \times V$ and $v_R \in V$, such that for each $v \in V$ there is exactly one path from v_R to v in the graph (V, E). (This path is trivial for $v = v_R$.) By vE we denote the set of vertices $w \in V$ such that $(v, w) \in E$.

3 Special case

Let $\mathcal{A} = (\Sigma, Q, q_I, \delta)$ be a fixed automaton on binary trees over alphabet Σ.

▶ Problem 3.1. given trees $U_1, \ldots, U_N$. Decide whether there is a tree T accepted by $\mathcal{A}$ which includes $U_1, \ldots, U_N$ as disjoint inner subtrees, and one of the trees appears at offset ϵ.

The assumption that one of the trees appears at offset ϵ is to simplify the presentation by eliminating some of the special cases connected with the root. If we don't want such an assumption, it is enough to add an empty tree to our sequence.

Note that although the trees are disjoint, it is possible that e.g. U_2 will be connected to a port of U_3. It is also possible that U_2 is not directly connected to a port of U_3, but there is a path of vertices in between which do not belong to any U_i.

▶ **Theorem 3.2.** *Assuming that $|Q|$ is fixed, the problem 3.1 above can be solved in time polynomial in $|\Sigma|$ and the size of trees $U_1, \ldots, U_N$.*

Proposition 6.1 in Section 6 below shows that the problem becomes NP complete without the assumption that $|Q|$ is fixed.

We will present the proof of the theorem above in the following way. We will show a sequence of simpler algorithms; for each of them, we will identify the major problem with it and show how to fix it. This sequence will converge to a correct algorithm running in deterministic polynomial time.

We will start with some additional definitions.

3.1 Multiplicity vectors

We call functions $f : Q \to \mathbb{Z}$ *multiplicity vectors*. For $x \in \mathbb{Z}$, we say $f \geq x$ iff $f(q) \geq x$ for all $q \in Q$; $f \leq x$ is defined similarly. By $[q]$ we denote the multiplicity vector such that $[q](q) = 1, [q](r) = 0$ for $r \neq q$. For a set of multiplicity vectors S, by $S[q > 0]$ we denote $\{f \in S : f(q) > 0\}$. If S_1 and S_2 are sets of multiplicity vectors, $S_1 + S_2 = \{f_1 + f_2 : f_1 \in S_1, f_2 \in S_2\}$, and $S_1 - S_2 = \{f_1 - f_2 : f_1 \in S_1, f_2 \in S_2\}$.

We say that a tree T **realizes** a multiplicity vector f from $q \in Q$ iff $f \geq 0$ and there is a valid run ρ of $\mathcal{A}$ over T such that for each state r, r appears in the ports of T at least $f(r)$ times (i.e., the cardinality of $\rho^{-1}(r) \cap \mathrm{port}(T)$ is at least $f(r)$), and $\rho(\epsilon) = q$.

We denote by $A(U, q, n)$ the set of multiplicity vectors f such that $f \leq n$ and there is a tree $T \supseteq U$ which realizes f from q.

▶ **Lemma 3.3.** *Sets $A(U, q, n)$ can be calculated in time polynomial in $|U|$, S, and n.*

Proof. We start with calculating $A(\emptyset, q, n)$ for each q.

Define the sequence of sets of multiplicity vectors $A_i(q, n)$ as follows:

- $A_0(q, n) = \{f : 0 \leq f \leq n, f \leq [q]\}$
- $A_{i+1}(q, n)$ is a set of f such that $0 \leq f \leq n$ and there exist $f_1 \in A_i(q_1, n)$ and $f_2 \in A_i(q_2, n)$ such that $(x, q, q_1, q_2) \in \delta$ and $f \leq f_1 + f_2$.

▶ Proposition 3.4. If $A_i(q, n) = A_{i+1}(q, n)$ for each $q \in Q$, then $A(\emptyset, q, n) = A_i(q, n)$.

The proof is straightforward. This proposition allows us to calculate $A(\emptyset, q, n)$ for each q and n inductively.

▶ Proposition 3.5. Let U be a tree such that U_1 and U_2 are full subtrees of U at offsets 0 and 1, respectively. Then $A(U, q, n)$ is the set of f such that there exist $f_i \in A(U_i, q_i, n)$ for $i = 1, 2$ such that $(U(\epsilon), q, q_1, q_2) \in \delta$ and $f \leq f_1 + f_2$.

Again, the proof is straightforward. This proposition allows us to calculate $A(U, q, n)$ by induction over subtrees of U. ◀

3.2 Algorithm I

We start with a strongly non-deterministic algorithm.

1. We guess a permutation of our set of trees: $U_{d_1}, U_{d_2}, U_{d_3}, \ldots, U_{d_N}$. This specifies the order of subtrees in our goal tree T. If U_i appears at offset w_i in the goal tree T, and w_i is a prefix of w_j, then the tree U_i will appear before U_j in this permutation. If w_i and w_j are incomparable, they can be ordered arbitrarily.
2. $f := [q_I]$. At each stage of our algorithm, f is the multiplicity vector which says how many occurrences of each state we have in our ports.
3. for $i := 1$ to N:

4. We guess $q \in Q$.
5. If $f(q) = 0$, then we fail.
6. $f := f - [q]$
7. $f := f + a$, where a is one of the elements of $A(U_{d_i}, q, N)$.
8. We accept.

In our loop, we fill one of our ports in some state q with the tree U_{d_i}. This means we used one of the ports specified by $f(q)$, but U_{d_i} gives us new ports according to $A(U_{d_i}, q, N)$. It is enough to use multiplicity vectors f such that $f \leq N$ because we will only have to use N states.

3.3 Algorithm II

Except for Step 1, all the steps of Algorithm I are easy to determinize in polynomial time. Instead of analyzing one choice, we construct the sets of all possible choices.

1. We guess a permutation of our set of trees, just like in Algorithm I.
2. $S := \{[q_I]\}$
3. for $i := 1$ to N:
4. $\quad P := \emptyset$;
5. $\quad$ for $q \in Q$ do
6. $\quad\quad R := S$;
7. $\quad\quad R := R[q > 0]$; // use only vectors which had a port with state q
8. $\quad\quad R := R - [q]$; // fill this port
9. $\quad\quad R := R + A(U_{d_i}, q)$; // we get new states in ports below U_{d_i}
10. $\quad\quad P := P \cup R$;
11. $\quad$ S := P;
12. We accept.

If in the end we have obtained a non-empty set S, we have won (constructed a tree which contains our required subtrees). If not, it probably means that we have guessed the permutation incorrectly.

Now, our goal is to avoid guessing the permutation.

3.4 Algorithm III

In Algorithm III we do not guess the permutation. Instead, we do not care if we have used up a state which we have not obtained yet — i.e., we remove Step 7 from Algorithm II, and allow our multiplicity vectors to go negative.

At the end of our algorithm we check whether S has an element $f \geq 0$. If no, it means that we had no chance – whatever permutation we would pick in Algorithm II or Algorithm I, we would get an empty set at the end. What if S contains a non-negative element? This does not yet mean that the tree exists, because it is still possible that it was impossible to do a good ordering. For example, suppose we have two states q and r which are unreachable from q_I, U_1 is an empty tree, and the tree U_2 realizes $[q]$ from r and U_3 realizes $[r]$ from q. Algorithm III will accept, even if the trees U_2 and U_3 can only be used in states which are not reachable from q_I.

3.5 Algorithm IV

Thus, we need to make sure that a correct permutation exists. For this, we use the following lemma:

▶ **Lemma 3.6.** *For $i = 1 \dots N$, let $f_i \in A(U_i, q_i, N)$. Then a permutation of trees for the Algorithm I which generates a positive answer, such that in step 3 for each i we choose to use state q_{d_i} as q and multiplicity vector f_{d_i} as a, exists iff the following two conditions are satisfied:*

- *Euler condition:* $[q_I] + \sum(f_i - [q_i]) \geq 0$
- *Connectedness: Let $G = (V, E)$ be the graph such that $V = \{q_i : i \in \{1 \dots n\}\}$, and $(p, q) \in E$ iff there exists an i such that $p = q_i$ and $f_i(q) > 0$. Then there must be a path from q_I to each state in V. edge $p \to q$. Then there must be a path from q_I to each state in V.*

This lemma is a generalization of the classic Euler theorem about a condition for existence of an Euler path in a graph. Also the proof generalizes the proof of Euler's result. This technique is essentially equivalent to Theorem 3.1 from [7] and Proposition 3 from [11].

Proof. We try to build the tree in an arbitrary way, filling up ports with our trees whenever the state matches. If we end up with all trees used, we are done. Otherwise, there is some tree U from r_0 which cannot be inserted because all ports of type r_0 have been already used. We can assume that r_0 was already used as a connecting state in our construction; otherwise, there would be no connection between the used and unused trees, which means that the connectedness condition is not satisfied. We construct a sequence of states $r_0, r_1, r_2, \dots, r_m$ and of unused trees $U_{i_0}, U_{i_1}, \dots$, such that $q_{i_k} = r_{k+1}$ and $f_{i_k}(r_k) > 0$, until we no longer can find an unused tree U_{i_m} with $f_{i_m}(r_m) > 0$. This construction must end in the state $r_m = r_0$; if it ended in any other state, it would violate the Euler condition (the state r_m would be used in the root more times that it would be produced in a port). We find the place in our construction where r_0 is used for connection, and insert the trees $U_{i_{m-1}}, \dots, U_{i_0}$ there. We continue this operation until all trees are used. ◀

Algorithm III checked for the Euler condition, but we have to modify it to also check for connectedness. This can be done by choosing the (connected) graph G, or even better, guessing the spanning tree of G. Since $V \subseteq Q$ and $|Q|$ is fixed, we have a fixed number of possible spanning trees. Thus, we modify Algorithm III in the following way:

1. We start by guessing a rooted tree $\tau = (V, E, q_I)$. (A fixed number of possible guesses.)
2. Now, for each edge $e \in E$ we guess i_e, the index of the tree which is forced to realize this edge. (Thus, we have $N^{|V|-1}$ possible guesses, which again is polynomial.)
3. We execute Algorithm III, except that in step 5, if $i = i_{(q_1,q_2)}$ for some $e \in E$, then we use only q_1 as q, and in general we can only use states from V; and in step 9 we restrict $A(t_{d_i}, q, n)$ to multiplicity vectors f such that $f(q_2) > 0$.

Algorithm IV is correct and works in deterministic polynomial time (when guesses are replaced by looping over the whole subset).

4 Incomplete tree descriptions

In this section we generalize the results from the section below to a case where we can force the trees U_i to appear in the result tree in a specific pattern.

As in the previous section, let $\mathcal{A} = (\Sigma, Q, q_I, \delta)$ be a fixed tree automaton over Σ. Also let let $\mathcal{L}$ be a finite set of regular (word) languages over $\{0, 1\}$.

An **incomplete tree description**, or a **pattern** for short, is a $P = (V^P, E^P, p_0, C, L)$, where:

- (V^P, E^P, p_0) is a rooted tree,
- $C : V^P \to P(\Sigma)$ assigns a set of possible elements of alphabet to each subpattern,
- $L : E^P \to \mathcal{L}$ assigns one of the languages in $\mathcal{L}$ to each edge of our tree.

For $p \in V^P$, $P[p]$ is the subpattern obtained from P by moving the root p_0 to p and restricting V^P to vertices accessible from p.

▶ **Definition 4.1.** A tree T **matches** an incomplete tree description P iff $T(\epsilon) \in c(P)$, and for each $p \in p_0 E^P$, a tree U_p matching $P[p]$ is an inner subtree of T at offset $w_p \neq \epsilon$, and they are disjoint inner subtrees.

▶ Problem 4.2. Given an incomplete tree description P, is there a tree T accepted by $\mathcal{A}$ which matches P?

Note that the set of trees which matches a given incomplete description P is a regular language of trees; and an intersection of two regular languages is also a regular language. However, an automaton recognizing trees consistent with P would be of size exponential in $|P|$, so such a view is not practical for us.

▶ **Theorem 4.3.** *Let $\mathcal{A}$ and $\mathcal{L}$ be fixed. Problem 4.2 is solvable in time polynomial in the size of P.*

Propositions 6.1 and 6.2 in Section 6 below show that the problem becomes NP complete without the assumption that $\mathcal{A}$ and $\mathcal{L}$ are fixed, respectively.

Note that the Problem 3.1 is the special case of Problem 4.2 where $\mathcal{L} = \{0, 1, (0 + 1)^*\}$ and $(0 + 1)^*$ is only allowed to appear at edges starting in p_0 (not counting some minor differences regarding the root of the result tree).

Proof. For words $u, v \in \{0, 1\}^*$, we say that $u \equiv v$ iff for each two words $t, w \in \{0, 1\}^*$ and each $L \in \mathcal{L}$ we have $tuw \in L$ iff $tvw \in L$. It is well known that the relation $\equiv$ has a finite index. Let M be the set of equivalence classes of $\equiv$; let $[w] \in M$ be the equivalence class of a word $w \in \{0, 1\}^*$. M is equipped with a concatenation operation given by $[w_1][w_2] = [w_1 w_2]$. For a $L \in \mathcal{L}$, let $[L] = \{[w] : w \in L\}$; note that for each $m \in M$ either $m \subseteq L$ or m is disjoint with L. The set M is called the *syntactic monoid* of $\mathcal{L}$ (see e.g. [13]; although syntactic monoids are more commonly known for single languages, our extension of this notion to a finite family of languages is quite obvious).

Let P be an incomplete tree description, and $N = |P|$.

We will need to extend our definition of multiplicity vectors to take the elements of M into account together with states. An **extended multiplicity vector** (EMV) is a function $f : Q \times M \to \mathbb{Z}$. We say that a tree T realizes an EMV f from $q \in Q$ iff there is a valid run ρ of $\mathcal{A}$ over T such that $\rho(\epsilon) = q$, and for each $r \in Q$ and $m \in M$, $\left|\rho^{-1}(r) \cap \mathrm{port}(T) \cap m\right| \geq f(r, m)$. We define all operations on EMVs and sets of EMVs in the same way as for multiplicity vectors.

If f is an EMV and $m \in M$ we denote by mf the EMV such that $mf(q, m_1) = \sum_{m_2 : mm_2 = m_1} f(q, m_2)$.

For $p \in V_P$ and a state $q \in Q$, we denote by $A(p, q, n)$ the set of EMVs f such that $0 \leq f \leq n$, and there is a tree U which matches $P[p]$ and realizes f from q.

Now, it is enough to show that $A(p_0, q, n) \neq \emptyset$. Thus, all we need is the following lemma:

▶ **Lemma 4.4.** *The set $A(p, q, n)$, where $n \leq N$, can be calculated in polynomial time for each $p \in V_P$.*

Thus, to check whether a tree accepted by $\mathcal{A}$ and matching P exists, it is enough to check whether $A(p_0, q_I, 0) \neq \emptyset$. ◀

To prove 4.4 we will need one more technical lemma:

▶ **Lemma 4.5.** *Let $q \in Q$ and $x \in \Sigma$. Let set $A^0(x, q, n)$ be the set of EMVs f which are realized from q by a tree which has x in its root, and $0 \leq f \leq n$. The set $A^0(x, q, n)$ (where $n \leq N$) can then be calculated in polynomial time.*

Proof. Let $A^0_0(x, q, n)$ be the set of EMVs which can be realized by a tree which has x in its root and ports in both of its children.

Let $A^0_{k+1}(x, q, n)$ be the set of EMVs f such that either $f \in A^0_0(x, q, n)$, or the following conditions are satisfied for some EMVs f_0 and f_1, states $q_0, q_1 \in Q$, and letters $x_0, x_1 \in \Sigma$:

- $f \leq f_0 + f_1$
- $f_0 \in [0]A^0_k(x_0, q_0, n)$
- $f_1 \in [1]A^1_k(x_1, q_1, n)$
- $0 \leq f \leq n$

It is straightforward to check that the sequence $A^0_k(x, q, n)$ is increasing for each x, q, n, and its limit is $A^0(x, q, n)$. ◀

Proof of Lemma 4.4. We prove the lemma by induction over subpatterns. Let $n' = n + |pE_P|$. From inductive hypothesis we can calculate the $A(p', r, n)$ for each $p' \in pE_P$ and each $r \in Q$.

1. $A := \emptyset$
2. for each rooted tree $\tau = (V^\tau, E^\tau, v^\tau)$ such that $V^\tau \subseteq Q \times M$ and $v^\tau = (q, [\epsilon])$:
3. for each assignment $\alpha : E^\tau \to pE_P \cup \{p\}$:
4. $\quad B := \bigcup_{x \in C(P)} A^0(x, q, n')$
5. $\quad \mathrm{Restrict}_\alpha(B, (q, [\epsilon]), p)$
6. $\quad$ For each $p' \in pE_P$:
7. $\quad\quad$ For each $(r, m) \in Q \times [L(p, p')] \cap V^\tau$:
8. $\quad\quad\quad A := mA(p', r, n')$
9. $\quad\quad\quad \mathrm{Restrict}_\alpha(A, (r, m), p')$
10. $\quad\quad\quad$ let $B := B - [(r, m)] + A$
11. $\quad$ Let $A := A \cup \{f \in B : 0 \leq f \leq n\}$
12. return A

$\mathrm{Restrict}_\alpha(B, (q, m), p)$ is defined as follows:

1. For each $((q_1, m_1), (q_2, m_2)) \in \alpha^{-1}(p)$:
2. $\quad$ if $(q_1, m_1) \neq (q, m)$ then $B := \emptyset$
3. $\quad B := B[(q_2, m_2) > 0$

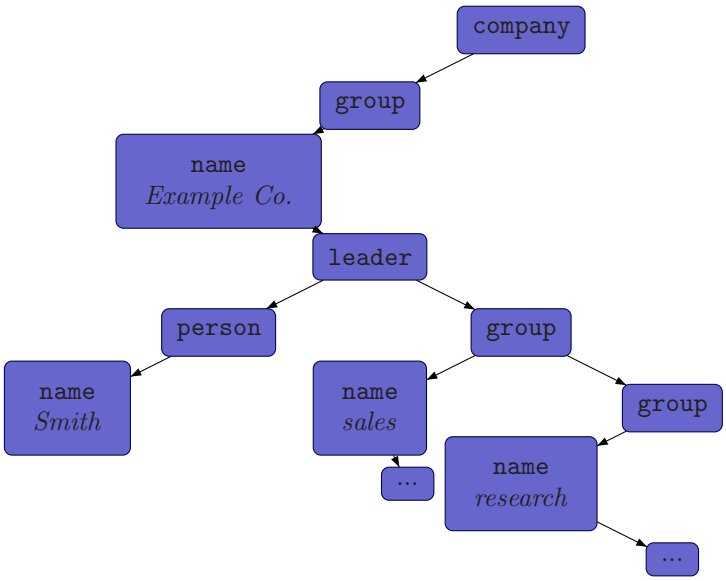
This algorithm is based on a idea which is very similar to that of Algorithm IV. The differences is that it takes the syntactic monoid M into account, and calculates the EMV instead of non-emptiness. ◀

5 Application to XML

In this section, we compare the problem solved in Theorem 4.3 to the open question from [4]: is it possible to remove the condition of $\downarrow^*$-freeness from Theorem 5.28? In other words, for a fixed DTD D, and an incomplete DOM-tree t, is it possible to find out whether there exists an XML document X which satisfies D and is consistent with t, in time polynomial in size of t?

It is beyond scope of this paper to include the full definition of XML documents, DTDs, and incomplete DOM trees, thus we just give examples and list the important differences:

- XML documents are trees, but they are unranked trees (with a sibling ordering), not binary ones. However, we can use the standard encoding of unranked trees in binary trees: if a vertex v of the unranked tree is assigned w in our binary tree, then the first child of v (if any) is assigned $w0$, and the next sibling of v (if any) is assigned $w1$. This allows us to convert any unranked tree (or, more accurately, an unranked forest) to a binary tree, and vice versa. The following picture shows the XML document from the introduction, represented as a binary tree.

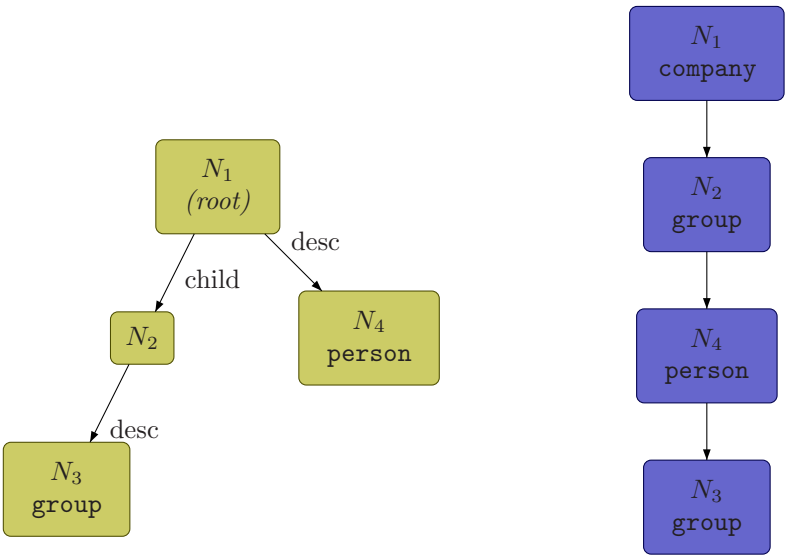


- In [4], the question is whether a tree consistent with a given DTD exists. Essentially, for each type of a node, a DTD (document type definition) gives a regular expression which describes which sequences of node types can be allowed as children of a node of given type. For example, we can use the following DTD to define valid company descriptions:

```
<!ELEMENT company (group)>
<!ELEMENT group (name? leader? (group|person)*)>
<!ELEMENT leader (person)>
<!ELEMENT person (name)>
<!ELEMENT name (#PCDATA)>
```

In our results, we use tree automata instead of DTDs. Our result is more general, since tree automata can verify whether a tree is consistent with a DTD, but not the other way around – tree automata are more powerful. Properties such as *if a group has a leader and only one subgroup, then this subgroup cannot have a leader* cannot be easily described with a DTD, but can be described with an automaton.

- Incomplete DOM-trees allow using markings: in terms of patterns, each subpattern can have marking which forces it to appear in a specific location, which is one of the following: *root, leaf, first child, last child.* These markings can be easily checked by extending the automaton $\mathcal{A}$ and the alphabet Σ, or by enforcing EOT on the applicable child, so our result covers this.
- Subpatterns are allowed to have only one of the following relations to their bigger nodes: *next sibling*, *younger sibling*, *child*, *descendant.* This corresponds to picking a language 1, 1^*, 01^*, $0(0+1)^*$, respectively, as $L(p, p')$. Thus, our result again generalizes the XML case.
- The last difference cannot be easily solved with our means. In case of [4], a tree T is considered to match pattern P if there is an injective mapping ϕ from V_P to vertices of T such that for each edge $(p_1, p_2) \in E_P$ we have $\phi(p_2) = \phi(p_1)v$ where $v \in L(p_1, p_2)$. There is a subtle difference with our definition. Consider again the pattern from the introduction (on the left), and the following tree (on the right):



Is the tree to the right consistent with the pattern on the left? According to the definition from [4], yes ($\phi(v)$ is the node of the tree which is labelled with the same N_i as v). According to our definition, no: the inner subtree which matches the subpattern rooted at N_2 would have to include N_4, which means that it would not be disjoint with the inner subtree which matches the subpattern N_4. This is the only case where there is a difference: if N_4 is below N_3, above N_2, between N_2 and N_3, or below N_2 but not above N_3, then the tree matches the description according to both definitions.

In general the situation can be more complicated (there could be long sequences of descendants which the definition from [4] would allow to interleave in arbitrary ways). We don't see how to use inductive reasoning to exactly solve the case from [4]. However, we think that it is not natural to interleave vertices which correspond to different subpatterns, and for this reason both definitions are motivated equally well.

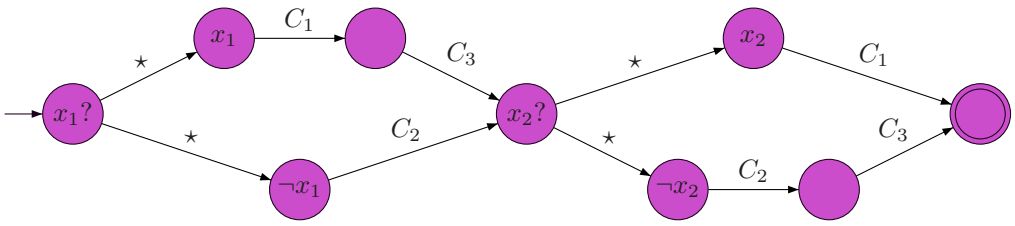
6 Hardness results

In this section we show that the assumptions about the fixed size of Q and $\mathcal{L}$ cannot be removed from Theorems 3.2 and 4.3.

▶ Proposition 6.1. Without assumption that $|Q|$ is fixed, Problem 1 is NP complete, even in the word case, and when all tress U_i contain only one letter.

This has been essentially proven in [4]; we provide a short proof for completeness.

Proof. The problem is in NP because the witness must be polynomial. We reduce the problem of CNF-SAT satisfiability to Problem 1. Let $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_n$ be a CNF formula, with variables $x_1, \ldots, x_m$. Our language will be $\Sigma = \{\star, C_1, \ldots, C_n\}$. For each x_i our automaton contains two paths, one for x_i and one for $\neg x_i$; on each path we accept a letter corresponding to all clauses that are satisfied by choosing given literal. Tree U_i asks whether C_i appears anywhere in the resulting word. The following picture shows the automaton used to decide whether a formula $\phi = C_1 \wedge C_2 \wedge C_3$ is satisfiable, where $C_1 = x_1 \vee x_2$, $C_2 = \neg x_1 \vee \neg x_2$, and $C_3 = x_1 \vee \neg x_2$.



◀

Note that we could get a more straightforward reduction from the Hamiltonian circuit problem if we could say that the result tree T is covered by U_is completely. After a small modification, our technique allows to solve Problems 3.1 and 4.2 even in this case (or in the case where we have a subset $\Delta \subseteq \Sigma$ and labels from Δ are only allowed to appear when explicitly requested by the description).

Also note that we could encode the reduction from the proof of Proposition 6.1 in a two letter alphabet, but then the trees (or word) U_i need to be longer. Restricting to both an alphabet of fixed size and trees U_i of size 1 again yields a polynomial algorithm (which is similar to one given in Lemma 3.3, but we count the number of occurrences of each label, not each state; a similar problem has been solved in [11]).

▶ Proposition 6.2. Without assumption that $\mathcal{L}$ is fixed, Problem 2 is NP complete, even for a very simple (fixed) automaton ($|\sigma| = 1, |Q| = 2, |\delta| = 2$), and incomplete tree descriptions of depth 2 (i.e., consisting just of a root and its direct subpatterns).

Proof. Our automaton says that the tree consists of a single path. Thus, we have $\Sigma = \{a\}, Q = \{q_I, q_N\}, \delta = \{(a, q_I, q_I, q_N), (a, q_I, q_N, q_I)\}$.

Again, the problem is in NP because the witness must be polynomial, and we prove hardness by reducing CNF satisfiability. Let $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_n$ be a CNF formula, with variables $x_1, \ldots, x_m$. Let D_i be the set of sequences of m bits S such that C_i is satisfied when x_j is satisfied iff $S_j = 1$. The subpattern P_i says there is a vertex labeled with a at position $D_i\{0 + 1\}^*$; this regular language can be defined in polynomial size (if defined as a

regular expression or a DFA; the syntactic monoid is of exponential size, though). Since the tree must have only one path, all D_i parts much correspond to the same word, which means that all C_i are satisfiable simultaneously. ◀

We don't know whether the problem is already NP complete for a trivial automaton. However, if we use the injective definition of consistence with an incomplete description from [4], and a trivial automaton, then the problem is again NP complete in the word case. Indeed, we can reduce the 3-CNF satisfiability problem: for each of the m variables we have a subpattern p_i, and $L(p_0, p_i) = 1^{im} + 1^{im+i}$; putting the tree matching the subpattern at offset 1^{im} means that x_i is false, and offset 1^{im+i} means that x_i is true. Then, for each clause we add additional subpatterns to respective p_i's, so that if the three variables have been given the only assignment which does not satisfy the clause, we have to fit three different offsets into two slots (choosing any other assignment gives us more space). Note that the syntactic monoid M – even the one which recognizes all languages L at once – is still of polynomial size in this case.

7 Conclusion

We have shown how to determine in polynomial time whether there exists a tree which is accepted by the given automaton $\mathcal{A}$ and is consistent with given incomplete description P, assuming that both the automaton $\mathcal{A}$ and the set of possible axes in P are fixed. This brings us closer to a complete understanding of the consistence problem for XML documents [4] and other hierarchical data. Although our goal was to improve the results of [4], we had to use another definition of when a tree matches a incomplete description in order to make our method work. Our definition differs in just one subtle detail — while the original one allowed several branches of the incomplete description to be mixed together and appear as a single branch in the result tree, our does not allow that — they can become a single branch, but the tree matching one of the subpatterns must completely precede the tree matching other subpatterns, no mixing is allowed. We believe that such mixing is not natural, so our result is also interesting. Still, the original consistency problem, for the original definition, remains open.

One of techniques we have been using is Lemma 3.6, which is essentially the classical Euler theorem about existence of Euler path, generalized to the case where an "edge" can have many endpoints. This technique has been also used recently in [11] (and also less recently in [7]). We believe that it is worth investigating whether this method has more applications for various kinds of branching structures in automata theory and computer science in general.

Acknowledgements I want to thank Pablo Barcelo and Filip Murlak for introducing me to these problems and their helpful comments, and Wojtek Czerwiński and Paweł Parys for the atmosphere of research at AUTOBÓZ 2010.

References

1 Serge Abiteboul and Oliver M. Duschka. Complexity of answering queries using materialized views. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '98, pages 254–263, New York, NY, USA, 1998. ACM.

2 Serge Abiteboul, Paris Kanellakis, and Gosta Grahne. On the representation and querying of sets of possible worlds. *SIGMOD Rec.*, 16:34–48, December 1987.

3 Serge Abiteboul, Luc Segoufin, and Victor Vianu. Representing and querying xml with incomplete information. *ACM Trans. Database Syst.*, 31:208–254, March 2006.

4 Pablo Barceló, Leonid Libkin, Antonella Poggi, and Cristina Sirangelo. Xml with incomplete information. *J. ACM*, 58:4:1–4:62, December 2010.

5 Mikolaj Bojańczyk and Igor Walukiewicz. Forest algebras. In *Logic and Automata'08*, pages 107–132, 2008.

6 H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

7 Javier Esparza. Petri nets, commutative context-free grammars, and basic parallel processes. *Fundam. Inf.*, 31:13–25, July 1997.

8 Ronald Fagin, Phokion Kolaitis, Renée Miller, and Lucian Popa. Data exchange: Semantics and query answering. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *Database Theory - ICDT 2003*, volume 2572 of *Lecture Notes in Computer Science*, pages 207–224. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-36285-1_14.

9 Tomasz Imieliński and Witold Lipski, Jr. Incomplete information in relational databases. *J. ACM*, 31:761–791, September 1984.

10 Yaron Kanza, Werner Nutt, and Yehoshua Sagiv. Querying incomplete information in semistructured data. *Journal of Computer and System Sciences*, 64(3):655 – 693, 2002.

11 Eryk Kopczynski and Anthony Widjaja To. Parikh images of grammars: Complexity and applications. In *Proceedings of the 2010 25th Annual IEEE Symposium on Logic in Computer Science*, LICS '10, pages 80–89, Washington, DC, USA, 2010. IEEE Computer Society.

12 Maurizio Lenzerini. Data integration: a theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, pages 233–246, New York, NY, USA, 2002. ACM.

13 Jean-Eric Pin. Syntactic semigroups. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of formal languages, vol. 1*, pages 679–746. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

14 W3C. Extensible markup language (xml) 1.1 (second edition). Available on: `http://www.w3.org/TR/xml11/`, 2006.