Optimal Probabilistic Generators for XML Corpora

Serge Abiteboul[†‡] serge.abiteboul@inria.fr
Yael Amsterdamer[†♣] yaelamst@post.tau.ac.il
Daniel Deutch[†‡♠] deutchd@cs.bgu.ac.il
Tova Milo[♣] milo@cs.tau.ac.il
Pierre Senellart[♡] pierre@senellart.com

[†]Inria Saclay – Île-de-France ; 4 rue J.Monod, 91893 Orsay Cedex, France
[‡]ENS Cachan ; 61, Av. du Président Wilson, 94235 Cachan Cedex, France
[♣]Tel Aviv University ; P.O. Box 39040, Tel Aviv 69978, Israel
[♠]Ben Gurion University ; P.O.B. 653 Beer-Sheva 84105, Israel
[♡]Institut Télécom ; Télécom ParisTech ; CNRS LTCI ; 46 rue Barrault, 75634 Paris Cedex 13, France

Abstract

We study the problem of, given a corpus of XML documents and its schema, finding *an optimal probabilistic model* (optimality meaning maximizing the likelihood of the corpus to be generated). We present an efficient algorithm for finding the best probabilistic model, in absence of constraints. We further study the problem in presence of integrity constraints (key, inclusion, and domain constraints) and consider in this case two different kinds of generators: a *continuation-test* generator that performs, while generating, some tests of schema satisfiability; these tests allow avoiding the violation of constraints (but as we show, are costly to implement), and a *restart* generator that may generate an invalid document and then restart and try again.

Résumé

Étant donnés un corpus de documents XML et son schéma, nous étudions le problème de déterminer un *modèle probabiliste optimal* (maximisant les chances de générer ce corpus). Nous montrons comment obtenir le modèle probabiliste optimal, en l'absence de contraintes. Nous étudions aussi le problème en présence de contraintes d'intégrité (clés, inclusions et contraintes de domaine) et considérons dans ce cas deux types de générateurs : (i) des générateur basés sur des tests de continuation qui évitent de générer des documents invalides au prix de tests coûteux ; et (ii) des générateurs qui produisent des documents jusqu'à en obtenir un qui soit valide au prix d'un nombre potentiellement très grand de productions de documents inutiles.

Keywords

DTD, probabilistic XML, XML generator, XML Schema, corpus fitting

1 Introduction

We are concerned with the following problem: given a corpus of XML documents, find a "best model" for this corpus. We consider *generative models*, and optimality means maximizing the likelihood of the corpus to be generated. There are two aspects in the problem. The first is to find a type (e.g., in an XML schema language such as DTD or XML Schema) the documents conform to. This has been intensively studied (see, e.g., [21, 19, 15, 8]). The second aspect is given such a type, find probabilities to "guide" this type, that in some sense maximize the particular corpus. This is the contribution of the present work: given a document corpus and a type for its documents, we show how to find *the best probabilistic model*.

Such a probabilistic model has a variety of usages:

Testing. The model can be used to generate (many) samples of the documents for test purposes. For instance, the document may describe some workflow sessions and the samples be used to stress-test a new functionality.

Explaining. The type is already useful to explain the corpus to users. The probabilities provide extra information on the semantics of data. E.g., in DBLP, how many journal articles there are vs. conference ones, or how many authors a paper has on average.

Querying. One can get an approximation of query answers by "evaluating queries" on this model in the style of query answering on probabilistic databases. For instance, one can verify the probability that journal articles have page numbers.

Type mining. Given a corpus, many possible types are such that all documents in the corpus satisfy them. To choose between them, one can use measures such as compactness (how small the type is) or precision (how much it rules out documents outside of the corpus). It turns out one can also use as a quality measure how well a probabilistic model for this type fits the corpus.

This variety of usages motivates the present work.

For types, we consider a very general notion that is essentially based on automata specifying the labels of the children of nodes with a certain label. This suggests the following *nondeterministic generator* for all documents satisfying a particular schema. Start with a single node with the root label. The children of a node of label l are generated using the automaton A_l corresponding to its label. Starting from the start state of A_l the generator nondeterministically chooses an accepted run of the automaton corresponding to a word $a_1...a_n\$$ in $L(A_l)$ (where \$ is a special terminating symbol) thereby generating a sequence of children respectively labeled $a_1...a_n$. To obtain a *probabilistic generator*, one attaches to the transitions of the automata probabilities to be selected. This provides the *skeleton* of the document. One needs also to feed in data values (at the leaves) following some specified distribution. The entire generation process may be interpreted as tree rewriting specified as ActiveXML documents [1].

Such a type together with the probabilities provides a probabilistic model for documents. Our contribution consists in determining the "best" such model for a given corpus of documents and a specific type. More precisely, we need to determine the probabilities to attach to the automata transitions that make the corpus most likely.

Model	Shorthand	Main results	Sec.
XML schema with/without constraints	schema	Formalization of the model	2.2, 2.3
Nondeterministic generator	nd-generator	Definition of the concept of a schema-based generator	3.1
Probabilistic generator	p-generator	An algorithm for finding the best probabilistic model for a document corpus based on a given schema, and a proof that termination probability is 1	3.2, 4.2
Restart generator	r-generator	Definition and discussion about the restart overhead	3.3, 5.2
Continuation-test generator	ct-generator	An algorithm for finding the best probabilistic model for a document corpus based on a given *binary* schema	3.3, 5.1

Table 1: Summary of Results

We first introduce an elegant way of obtaining these probabilities. The documents of a particular corpus are type-checked. For each automaton, we count the number of times each transition is chosen. We prove that using the relative frequencies of the transitions yields probabilities that optimize the generation of the corpus.

However, real applications also often involve (in addition to types) semantic constraints that greatly complicate the issue. We consider the three main kinds of constraints considered in practice, namely (unary) key, inclusion, and domain constraints. The main difficulty is that during generation we may reach some states where some of the transitions do not constitute real alternatives: following a particular transition, there is no chance of generating an instance obeying the constraints. This motivates our considering two kinds of generators, *restart generators* and *continuation-test* generators, as follows.

Restart generator. A run of a *restart* generator is quite simple. Ignore the constraints and generate a skeleton. Check whether there exists a value assignment for this skeleton so that the resulting document satisfies the constraints. If this fails, restart. Unfortunately, we show that for some input instances, there is virtually no chance of generating a skeleton that can be turned into a document satisfying the constraints, rendering restart-generators a problematic solution in general (although efficient in some cases).

Continuation-test generator. A run of a *continuation-test* generator is somewhat more complex. At every point of generation where there is more than one option, we invoke a continuation-test to check which of the options are feasible (i.e., for which there is a continuation that leads to a document satisfying the constraints). Thus we never choose a transition that takes us to a dead end and document generation always succeeds. The price we pay for this is performing the continuation-test, which we show is NP-complete.

To compute the best probabilistic model for the *continuation-test* generator, we have to assume that choices are binary. (We will explain why.) Again, we type-check the documents

of the corpus. We count the number of times each transition was chosen, but this time only in cases where there was more than one option with continuation. We prove that this gives optimal probabilities.

The present work focuses on establishing formal foundations for probabilistic generators for XML; practical implementations of the techniques presented here, as well as their experimental study, will follow as a next step (see Section 7 for future work).

Paper organization In Section 2 we provide definitions and background required for the rest of the paper. Generators are defined in Section 3. In Sections 4 and 5 we study the problem of finding the best probabilistic generators without and with constraints respectively. Related work is considered in Section 6 and we conclude with future work in Section 7. For ease of reading, the models and results considered in this paper are summarized in Table 1.

2 Preliminaries

In this section, we first introduce basic definitions for XML document and document corpora. We then consider schemas and constraints.

2.1 XML Documents and Corpus

An *XML document* is an unranked, ordered, and labeled tree. Given an XML document $d = (V, E)$, we use $root(d)$ for the root node of d. Let $\mathcal{L} = \mathcal{L}_\mathrm{l} \cup \mathcal{L}_\mathrm{i}$ be a finite domain of labels, where $\mathcal{L}_\mathrm{l}$ and $\mathcal{L}_\mathrm{i}$ are two disjoint sets of labels for leaves and internal nodes resp. We denote by $label : V \to \mathcal{L}$ the labeling function of the nodes, mapping leaf (internal) nodes to leaf (internal) labels. Given a node $v \in V$, $label_\downarrow(v) \in \mathcal{L}^*\$$ is the sequence of labels of the children of v, from left to right, with an additional terminating symbol $\$ \notin \mathcal{L}$. We assume that (only) the leaves are further assigned values from a countably infinite domain $\mathcal{V}$ by the function val.

Example 2.1. Consider the following XML document $\bar{d}$.

```
<Dept>
  <Head>Martha B.</Head>
  <Seniors>
    <Emp>
      <Name>Martha B.</Name>
      <Tel>123-5234</Tel>
      <Tel>123-5357</Tel>
    </Emp>
  </Seniors>
  <Juniors></Juniors>
</Dept>
```

This document describes the phone book of a department containing one senior employee as a member (who is also the department head), Martha B.: The root node v_0 is the one labeled with *Dept*, i.e., $root(\bar{d}) = v_0$ and $label(v_0) = Dept$. Let v_1 be the node such that $label(v_1) = Emp$. Then $label_\downarrow(v_1) = Name\ Tel\ Tel\$$. Similarly, if $label(v_2) = Name$, then $label_\downarrow(v_2) = \$$ (i.e., this is a leaf node with no children), but this node has a value, $val(v_2) =$ "*Martha B.*".

An *XML corpus* is then a finite bag of documents. Let $\mathcal{D}$ be the universal domain of all documents over $\mathcal{L}$. To represent a corpus we use a function $D : \mathcal{D} \to \mathbb{N}$, which maps each document d to the number of times d appears in the corpus. Let $supp(f)$ be the set

of all values x in the domain of the function f s.t. $f(x) \neq 0$. We use $|D|$ to denote the (necessarily finite) size of the corpus counting duplicates, i.e., $\sum_{d \in supp(D)} D(d)$.

2.2 Schema

We start by recalling the notion of schemas as specifications of valid XML documents. We consider first schemas with no constraints, and then in Section 2.3 we extend our definition to the general case where constraints are allowed. Also, to simplify the definitions, our model follows that of Document Type Definitions (DTDs). *However, we stress the model can be extended in a straightforward manner to a schema defined in the XML Schema language.* Let $\mathcal{Q}$ be a finite domain of states.

Definition 2.2. A *schema* S is a tuple $(r, \mathcal{A}_\downarrow)$, where $r \in \mathcal{L}_\mathrm{i}$ is the root label, and $\mathcal{A}_\downarrow$ is a partial function mapping an internal label $l \in \mathcal{L}_\mathrm{i}$ to a deterministic finite-state automaton (DFA) $\mathcal{A}_\downarrow(l) = A_l$ [1], whose language is $L(A_l) \subseteq \mathcal{L}^*\$$. An XML document d is said to be *verified* by a schema S if $label(root(d)) = r$ and for every internal node v of d, $l = label(v) \in \mathcal{L}_\mathrm{i}$ and $label_\downarrow(v) \in L(A_l)$.

We refer to the DFA A_l as *the deriving automaton* of l, and to the set of all such automata for the labels of a document d as the deriving automata of d.

Remark 2.3. Note by the definition, every word accepted by the automata must terminate with a \$, and contain no other \$'s. We also put a few additional restrictions on the model, to simplify further definitions. First, we assume the states of each deriving automaton form a disjoint subset of $\mathcal{Q}$. Second, we assume that the order the automatons are called is fixed, *Breadth-First Left-To-Right (BF-LTR)*. The order of invocation is irrelevant for verification but is important for the documents generation that is discussed in the sequel.

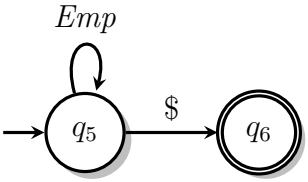

Figure 1: The $A_{Seniors}$ / $A_{Juniors}$ DFAs

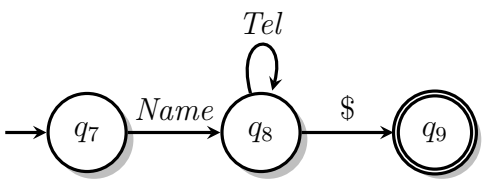

Figure 2: The A_{Emp} DFA

Example 2.4. Consider the schema $\bar{S}$ for general documents that describe a department of employees, like in Example 2.1. In this case, assume that $\mathcal{L}_\mathrm{i} = \{Dept, Seniors, Juniors, Emp\}$, $\mathcal{L}_\mathrm{l} = \{Head, Name, Tel\}$, and $r = Dept$. A_{Dept} is simply composed of a sequence of states q_0 to q_4, and $L(A_{Dept}) = Head\ Seniors\ Juniors\ \$$. $A_{Seniors}$, $A_{Juniors}$,[2] and A_{Emp}, depicted in Figures 1 and 2, are such that $L(A_{Seniors}) = L(A_{Juniors}) = Emp^*\ \$$ and $L(A_{Emp}) = Name\ Tel^*\ \$$. Note that $\bar{S}$ verifies the document $\bar{d}$ from Example 2.1.

[1] It is common to use *regular expressions* for the allowed sequences of children labels in a schema [20, 18]; the reasons for our choice of automata instead will become apparent when we discuss generators below.

[2] To simplify the example we use here the same deriving automaton for *Seniors* and *Juniors*.

2.3 Introducing Constraints

We continue by adding global constraints to the model we had so far. Following previous work on constraints in XML schema languages, we consider three major types of constraints on the values of the leaves.

Definition 2.5. A *schema with constraints* is a pair $\langle S^{\mathrm{u}}, C\rangle$, where S^{u} is a schema (without constraints) and C is a set of constraints on labels from $\mathcal{L}_{\mathrm{l}}$, of the following three types.

Key constraint Given a label $l \in \mathcal{L}_{\mathrm{l}}$, we denote by $uniq(l)$ the constraint that the value of each l-labeled leaf is unique (among all values of of l-labeled leaves in the document)[3].

Inclusion constraint Given two labels $l, l' \in \mathcal{L}_{\mathrm{l}}$, we denote by $l \subseteq l'$ the constraint that the values of l-labeled leaves are included in those of l'-labeled leaves.

Domain Constraint Given a label $l \in \mathcal{L}_{\mathrm{l}}$, we denote by $l \subseteq dom(l)$ the constraint that in any document, the values of l-labeled nodes are in $dom(l)$, a subset of $\mathcal{V}$.

We assume, in the sequel, that inclusion constraints $l \subseteq l'$ are only given when $dom(l) = dom(l')$, or when there are no domain constraints on l, l' (which we consider a practical scenario). When that is not the case, the combination of domain and inclusion constraints may change the domain of possible values for some of the labels, e.g. the "actual" domain of l may become $dom(l) \cap dom(l')$ and must be re-computed.

3 Generators

In this section, we consider various generators. First we consider nondeterministic generators, then probabilistic ones, and finally generators under constraints.

3.1 Nondeterministic Generator

Schemas are typically considered as *acceptors* for verifying XML documents. A schema accepts or rejects a given XML document. But it is also possible to see a schema as a *nondeterministic generator* (nd-generator). This is in the same sense that a DFA can be also seen as a word generator. Starting from the start state, one can choose nondeterministically transitions until reaching an accepting state, mimicking the run of the automaton on a word in the language, thus generating that word. For each node of label l, we can use the automaton A_l to nondeterministically generate the children of that node. Since a schema does not perform verification on the leaf values, an nd-generator generates *XML document skeletons*, which consist only of the labeled nodes, and into which leaf values can be later injected (see Section 7).

More precisely, generating a document skeleton d can be described as follows:

1. Generate a new root $root(d)$ with a label r and add it to a *todo* queue Q.
2. While Q is not empty, pop the node v at the head of the queue. Let l be the label of v and q the start state of A_l.

[3]We are considering here only *unary* keys, defined on single values and not combinations thereof.

3. Choose one transition in A_l from q to some state q'.
4. If the transition from q to q' is annotated with \$ (i.e., we have finished generating children for v) return to step 2.
5. Otherwise the transition is annotated with some label l. Generate v', a child for v such that $label(v') = l$. If $l \in \mathcal{L}_{\text{i}}$ add v' to Q. Set $q = q'$ and return to step 3.

The generation process thus ends when the *todo* queue at step 2 is empty, i.e., the deriving automata of all the generated inner nodes reached an accepting state. This means that the inner nodes generated last have only leaves as children (since we are going in a BF-LTR order).

Example 3.1. Reconsider the automaton A_{Emp} depicted in Figure 2 as a generator. Assume that we have already generated an *Emp*-labeled node v, and now we are generating its children. We start from state q_7 and when v has no children. We have only one option for the next transition, moving to q_8. Since the transition is annotated with *Name*, we generate the first child node and label it with *Name*. From q_8 we have two options: a transition to itself, in which case we generate an additional child, labeled *Tel*, and a transition to q_9, in which case no more children are generated for v.

Remark 3.2. Given such a nondeterministic generator, one can easily construct an Active XML document that generates the same documents. Active XML is much more general and allows specifying generators that will be introduced further in this paper, including the generators for handling schemas with constraints. For each label, a particular function is used for generating the children of nodes with that label. The functions use a *choose* function that introduces the nondeterminism. To introduce probabilistic choices, the *random* function can be used; *random* takes a rational number between 0 and 1 and generates T or F with a probability determined by that number. (The function *choose* is simply *random*(0.5)). Finally, *guards* (i.e., conditions controlling the firing of the functions), can be used to guarantee that BF-LTR order is followed. It also allows describing different strategies for generating the document (notably different orders).

Next, we define the notion of a *generation trace*, which describes the process of document generation in terms of the nondeterministic choices taken by the generator.

Definition 3.3. A *generation trace of a node* v, whose deriving automaton is A and where $label_{\downarrow}(v) = a_1...a_n\$$, is a sequence $\langle q_0, a_1\rangle, \langle q_1, a_2\rangle, ..., \langle q_n, \$\rangle$ where $q_0, ..., q_n \in \mathcal{Q}$ and the transition function δ of A is such that $\delta(q_{i-1}, a_i) = q_i$ for all $1 \leqslant i \leqslant n$ and $\delta(q_n, \$)$ is an accepting state. A *generation trace of a document* is then the concatenation of all the generation traces of all its inner nodes, in the order they were performed.

We next show that using an nd-generator indeed generates the documents verified by the corresponding schema.

Proposition 3.4. *For a schema S, the set $D \subseteq \mathcal{D}$ of all documents that S generates as an nd-generator is exactly the set of all documents S verifies.*

Proof (sketch). Assume that some document d is verified by S. For each internal node v in d there is an automaton $A_l = \mathcal{A}_{\downarrow}(label(v))$; A_l performs a sequence of state transitions on $label_{\downarrow}(v)$ and reaches an accepting state (since S verifies d). Take the sequence of pairs of state and transition label to be the generation trace for v. Concatenating all the generation

traces of the inner nodes according to our BF-LTR order will give a valid generation trace of S for d. The other direction is also simple – it is easy to see that if an automaton generates nondeterministically a sequence of child labels, it also accepts this sequence of labels; hence if a finite d is generated by S it is also verified by it. □

3.2 Probabilistic Generator

For practical purposes, we are not only interested in generating all possible finite documents that match some XML schema, but rather want to generate them according to some probability distribution. For that we introduce the notion of probabilistic generator, where the nondeterministic choices are associated with probabilities.

Definition 3.5. A *probabilistic generator* (p-generator) S is a pair $\langle S^{\mathrm{u}}, \textit{t-prob}\rangle$, where S^{u} is a schema, and *t-prob* is a function $\mathcal{Q} \times \mathcal{L} \to [0,1]$ mapping the transitions of the deriving automata of d to probabilities, such that for every $q \in \mathcal{Q}$, $\sum_{l\in\mathcal{L}} \textit{t-prob}(q,l) = 1$, and for every transition (q,l) which is not a part of any automaton, $\textit{t-prob}(q,l)$ is 0.

The probabilistic generation process is then very similar to the nondeterministic one, except that from each automaton state q, the generator *randomly draws* the next transition (q,l), according to *t-prob*.

Document probability. Let d be a document skeleton. For each internal node v in d, the probability of $\textit{label}_{\downarrow}(v)$ is the product of probabilities of all the transitions in its generation trace; the probability of d is then the product of all such probabilities over all its nodes. It is implied that we assume here independence of the probabilistic events associated with transitions (and naturally, independence in generation of different documents).

Example 3.6. Let us assign probabilities to the transitions in the schema described in Example 2.4. Assume that $\textit{t-prob}(q_5, \textit{Emp}) = 0.3$, $\textit{t-prob}(q_5, \$) = 0.7$, $\textit{t-prob}(q_8, \textit{Tel}) = 0.6$ and $\textit{t-prob}(q_8, \$) = 0.4$ (all other transitions have probability 1). We can now compute the probability of generating the document skeleton $\bar{d}$ in Example 2.1. The following table shows for each node its generation trace and the computation of generation probability. Since all internal nodes in $\bar{d}$ have unique labels, we use them here as node identifiers.

Node	**Generation trace**	**Probability**
Dept	$\langle q_0, \textit{Head}\rangle, \langle q_1, \textit{Seniors}\rangle, \langle q_2, \textit{Juniors}\rangle, \langle q_3, \$\rangle$	$1 \cdot 1 \cdot 1 \cdot 1 = 1$
Seniors	$\langle q_5, \textit{Emp}\rangle, \langle q_5, \$\rangle$	$0.3 \cdot 0.7 = 0.21$
Juniors	$\langle q_5, \$\rangle$	0.7
Emp	$\langle q_7, \textit{Name}\rangle, \langle q_8, \textit{Tel}\rangle, \langle q_8, \textit{Tel}\rangle, \langle q_8, \$\rangle$	$1 \cdot 0.6 \cdot 0.6 \cdot 0.4 = 0.144$
Total		$0.21 \cdot 0.7 \cdot 0.144 \approx 0.021$

The last row shows the total probability to generate $\bar{d}$ with the p-generator, which is the product of the probabilities of the inner nodes.

3.3 Generators with Constraints

As before, we would like to define a model for the generation of XML documents that are verified by a given schema with constraints on the values. However, in the presence of constraints, a generator that only makes independent choices may be unsuitable, as shown by the next example.

Example 3.7. Let us now consider a schema based on $\bar{S}$ from Example 2.4, but with the following additional constraints on the values:

- $uniq(Name)$: the employee names are unique.
- $Tel \in$ 123-5$\{0,..,9\}^3$: the department phone numbers always start with 123-5, and then some 3 digits.
- $Head \subseteq Name$: the name of the department head must be a name of an employee in the department.

Note that a document generated according to our schema may list a head but no member employees, in violation of constraint 3. We can try to enforce that there is at least one employee, by setting $t\text{-}prob(q_5, Emp)$ to 1 (either in $A_{Seniors}$ or $A_{Juniors}$). However, such a generator will never halt. Another possibility could be modifying the automaton itself to enforce, e.g., that there is at least one junior / senior employee; but the resulting generator will no longer correspond to the schema and particularly will not generate $\bar{d}$ from Example 2.1 (or a similar document, where Martha B. is a junior employee).

We suggest two kinds of generation models which deal with the problem described in the above example: *restart generators* which try to generate a document, check if it is invalid, and if so start the process over again; and *continuation-test generators*, which may perform a test for the existence of a continuation that leads to a valid document. As we explain later, the generator can use this test to avoid generating invalid documents.

Restart generators. We start by defining more formally the notion of a restart generator (r-generator). An r-generator G is a pair $\langle G^{\mathrm{p}}, C\rangle$, where G^{p} is a p-generator, and C is a set of constraints (as in a constrained schema). The operation of G is composed of two main steps which may be repeated.

1. Generating, probabilistically, a document skeleton d matching the schema of G^{p}. This step can be done simply by invoking G^{p}.
2. Checking, given d and C whether there *exists a valid value assignment to the leaves of* d. If not, d is discarded and we start over.

Some important questions which arise here are whether the test of the second step is decidable, and if so whether it can be computed efficiently. We show that the answer is "yes" to both questions and how to compute the answer in Section 5.2. An r-generator is very simple, but may generate many invalid documents before generating a valid one, if at all. This is a major pitfall, which leads us to consider the next kind of generators.

Continuation-test generators. A continuation-test generator (ct-generator) G makes calls to a *continuation test* after probabilistically choosing the next step, and before actually taking it. The continuation-test takes as input (1) a schema S, (2) a generation trace ξ of the partial document generated so far, and (3) $a \in \mathcal{L} \cup \{\$\}$, the choice G considers. The continuation test returns as output false if and only if there exists no document d

verified by S such that $\xi, \langle q, a\rangle$ is a prefix of the generation trace for d. In such a case, the generator chooses one of the other transitions going out of q; the generator may further record the output of the test to avoid performing it twice on the same transition.

Intuitively, the continuation test guides the generator by testing if a possible next step can lead (eventually) to a valid document; if not, then the generator will not make this step. In a sense, the continuation test is the only reasonable Boolean test to perform here: if the test returns true when there is no continuation, an invalid document will be generated; in contrast, if the test returns false when there is a continuation, there are some valid documents (that may be in the corpus) that will never be generated, regardless of the probabilities assigned to transitions.

Note that, in the absence of constraints (when $C = \varnothing$), there are no invalid document skeletons and both r-generators and ct-generators are the same as p-generators.

3.4 Quality and Optimality Measures

For a given XML schema, there are many possible generator instances (for each model described above). In order to compare different generators we define measures of generator quality and optimality, as follows.

Fitting a corpus. The main goal of this paper, besides defining the models for different kinds of generators, is to describe how to *learn the probability distributions from a corpus of XML documents*. Thus, the quality of a generator depends on the probability of generating the corpus documents. Formally,

Definition 3.8. Given a generator G and for every document $d \in \mathcal{D}$, let $\Pr(d\,|\,G)$ be the probability for G to generate d. Let $D : \mathcal{D} \to \mathbb{N}$ be a document corpus. Then the *quality* of G with respect to D, denoted $\mathit{quality}(G, D)$, is $\prod_{d \in \mathit{supp}(D)} \Pr(d\,|\,G)^{D(d)}$ (recall that $D(d)$ is the number of occurrences of d in D).

Note that if we multiply $\mathit{quality}(G, D)$ by c_D, the multinomial coefficient of D as a bag,[4] the result is exactly the probability for G to generate D.

Optimal generator. We say that a probabilistic generator G *conforms* to a schema S if their structure is identical (as for instance in the extension of a schema to an r-generator or ct-generator). Given a schema S, a class $\mathcal{G}$ of generators conforming to S, and a document corpus D, we say that a generator $G \in \mathcal{G}$ is *optimal* for S, $\mathcal{G}$, D if for each generator $G' \in \mathcal{G}$, $\mathit{quality}(G, D) \geqslant \mathit{quality}(G', D)$. When $\mathcal{G}$ is understood, we say that it is optimal for S, D. We call the problem of finding the optimal generator (for a given S and D) `OPT-GEN`.

4 The Unconstrained Case

In this section, we first show quality bounds for generators, then study optimal generators for schemas without constraints. We will consider constraints in Section 5.

[4] c_D is the number of distinct permutations of the bags elements (in particular, if D is a set, $c_D = |D|!$).

4.1 An Upper Bound for Quality

We start by considering an upper bound of quality for a corpus. We will later discuss whether this boundary can be achieved by the types of generators we defined, or by others.

Given a corpus D, consider a generator that would generate each document d in D with probability $\frac{D(d)}{|D|}$, i.e., according to its relative frequency. The quality of this generator would be $q_D = \prod_{d \in supp(D)} \left(\frac{D(d)}{|D|}\right)^{D(d)}$. We can show that this is indeed the maximal possible quality of a generator for D, *independently* from the XML schema the generator conforms to, and even the type of generator, as the following propositions holds.

Proposition 4.1. *Let D be a corpus and G a generator. Then quality$(G, D) \leqslant q_D$.*

The proof is based on the following lemma (which follows from results in [10]).

Lemma 4.2. *Let $\alpha_1 \dots \alpha_n$ be n positive integers. We define the function $f_n : \mathbb{R}_+^n \to \mathbb{R}_+$ as: $(p_1, \dots, p_n) \mapsto f_n(p_1, \dots, p_n) = \prod_{i=1}^n p_i^{\alpha_i}$ Then the maximum of f_n under the constraint $\sum_{i=1}^n p_i \leqslant 1$ is obtained when $p_i = \frac{\alpha_i}{\sum_{j=1}^n \alpha_k}$ for $1 \leqslant i \leqslant n$ and only then.*

It is easy to obtain a generator that achieves this optimal quality. The generator ignores any schema information, and simply randomly draws documents from the corpus, according to their relative frequency. We argue that this is not a good generator. First, if the corpus is very large, this generator will be much less compact than the ones we study, so not appropriate for explanation or query evaluation. Furthermore, this generator suffers from *over-fitting*: it cannot generate any documents other than those already in the corpus, and thus it is not appropriate for, e.g., testing. We want generators that also generate documents that are similar to, yet different than, those in the corpus.

4.2 An Optimal Generator

We next consider the problem of finding the optimal probabilistic generator out of those conforming to a given schema, in the unconstrained case.

Theorem 4.3. *We can solve* `OPT-GEN` *(without constraints) in time $O(|S| + |D|)$ where $|S|$ is the size of the schema S and $|D|$ is the total size of the corpus D (i.e., the sum of the size of all distinct elements in D, plus a binary encoding of their multiplicity).*

Proof. Consider Algorithm 1, which gets a schema as input and computes a probability for each transition. In lines 2–3 the schema is used for verifying the input corpus documents, and in the process the number of times each transition (q, a) was chosen is recorded in $freq(q, a)$ (also considering the frequency of each document in the corpus). Then in lines 4–6 we assign each transition (q, a) as probability the relative number of times it was chosen after reaching q. If there is no data for a certain state (i.e., it was not reached during the verification), we give equal probabilities to its outgoing transitions.

By construction, Algorithm 1 outputs a generator which has the same structure as S. It is easy to check that the transition probabilities from each node in an automaton of the generator sum up to 1 (this is enforced by the normalization in line 6).

Lines 1, 4–5, and 6 require a time linear in S. The loop in lines 2–3 consists in running the schema on each $d \in D$ and therefore require a time linear in the size of D.

Input: schema S, corpus D of documents verified by S
Output: p-generator G conforming to S

foreach *transition* (q, a) *in an automaton of* S **do** $freq(q, a) \longleftarrow 0$;
foreach $d \in supp(D)$ **do**
    $\xi \longleftarrow$ the generation trace of d by S;
    foreach $\langle q, a \rangle$ *in* ξ **do**
        $freq(q, a) \longleftarrow freq(q, a) + D(d)$;
foreach *state* q *in an automaton of* S **do**
    $total(q) \longleftarrow 0$;
    $out(q) \longleftarrow 0$;
    foreach *transition* (q, a) *in an automaton of* S **do**
        $out(q) \longleftarrow out(q) + 1$;
        $total(q) \longleftarrow total(q) + freq(q, a)$;
$G \leftarrow \langle S, t\text{-}prob \rangle$ s.t. $\forall q \in \mathcal{Q}, a \in \mathcal{L} \cup \{\$\}$ $t\text{-}prob(q, a) = \frac{1}{out(q)}$ if $total(q) = 0$, otherwise $t\text{-}prob(q, a) = \frac{freq(q,a)}{total(q)}$;
return G;

Algorithm 1: Algorithm for `OPT-GEN` (no constraints)

It is still to be shown that the output G of Algorithm 1 has maximum quality among all generators that conform to S. The quality of G is:

$$quality(G, D) = \prod_{d \in supp(D)} \Pr(d|G)^{D(d)} = \prod_{d \in supp(D)} \prod_{q \text{ in } S} \prod_{(q,a) \text{ in } S} \left(\frac{freq(q, a)}{total(q)} \right)^{D(d) \cdot \#\langle q,a \rangle \text{ in } d\text{'s trace by } S}$$
$$= \prod_{q \text{ in } S} \prod_{(q,a) \text{ in } S} \left(\frac{freq(q, a)}{\sum_{(q,a') \text{ in } S} freq(q, a')} \right)^{freq(q,a)}$$

whereas, similarly, every probabilistic generator G' conforming to S has quality:

$$quality(G', D) = \prod_{q \text{ in } S} \prod_{(q,a) \text{ in } S} p(q, a)^{freq(q,a)}$$

for some assignment $p(q, a)$ verifying, for each state q of S, $\sum_{(q,a) \text{ in } S} p(q, a) = 1$. Observe that there is no constraint between the $p(q, a)$'s for transitions of different origins (q, a) and (q', a'). We can then look independently for each state q which assignment of $p(q, a)$ maximizes the term $\prod_{(q,a) \text{ in } S} p(q, a)^{freq(q,a)}$ under the summing constraint. Lemma 4.2 shows that this is exactly the assignment made by G.[5] □

To be of practical use, the generator returned by Algorithm 1 needs to have a guarantee of almost always termination, which is not a consequence of Theorem 4.3. Even the maximal probability of generating the corpus documents (for which the generator halts) may still be far from 1, and thus the probability of not halting may not be negligible. However, we can

[5] When $total(q) = 0$, the value of this term is 1 for any assignment of $p(q, a)$, and in particular for the uniform distribution in Algorithm 1.

show that our construction guarantees termination. (The proof is by an adaptation of a corresponding result in [10].)

Theorem 4.4. *The generator returned by Algorithm 1 has a termination probability of* 1.

5 The Case with Constraints

We now allow constraints, as defined in Section 3.3. We consider the computation of an optimal probabilistic generator given a constrained schema. Recall that we study two interesting classes of generators: continuation-test generators (ct-generators) and restart generators (r-generators). We start with continuation-test generators.

5.1 Continuation-Test Generators

We first study the complexity of continuation tests. To do that, we need to adapt some known result:

Lemma 5.1 (adapted from [12, 14])**.** *The satisfiability of an XML schema with unary key, inclusion and domain constraints is NP-complete w.r.t. the size of the schema.*

Proof (sketch). A similar claim is proved in [12], which follows, in turn, the proof in [14]. Both models in [12, 14] are more expressive than ours (which means that NP membership carries over), but the hardness results are given even for a very simple model, a deterministic restriction of DTDs (which is less expressive than ours). One last required adaptation follows from the fact that their results are for key and inclusion constraints but not for domain constraints. To account for domain constraints, we briefly review the proof used in [12]. The proof there is by encoding the schema with constraints as a Presburger formula, and showing that the formula is satisfiable if and only if the schema with constraints is satisfiable. To extend the proof to also account for domain constraints in our settings, we first observe that a domain constraint on l restricts the set of valid document skeletons only if the domain is finite and there is a key constraint on l; in this case the domain constraint is expressible as an inequality specifying that the number of occurrences of l is smaller than the domain size. So, we add the relevant inequalities to the Presburger formula, and the proof technique of [12] can still be used. □

We call a partial generation trace *valid* for a generator G if it is a prefix of a generation trace of a valid document skeleton by G. We now have:

Proposition 5.2. *Let $S = \langle S^{\mathrm{u}}, C\rangle$ be a schema with constraints and G a generator conforming to S^{u}. Let ξ be a partial generation trace valid for G, and let (q, a) be a possible next transition for G after ξ. Whether $\xi, \langle q, a\rangle$ is a valid (partial) generation trace for G is NP-complete.*

Proof (sketch). NP-hardness follows from Lemma 5.1. To prove inclusion in NP, we construct in polynomial time a schema that is satisfiable if and only if the continuation test succeeds, and then use the NP algorithm of [12] to decide. □

Input: constrained schema S, corpus D of documents verified by S
Output: ct-generator G conforming to S

1 **foreach** *transition* (q, a) *in an automaton of* S **do** $freq(q, a) \longleftarrow 0$;
2 **foreach** $d \in supp(D)$ **do**
  $\xi \longleftarrow$ the generation trace of d by S;
  foreach $\langle q, a \rangle$ *in* ξ **do**
    if $\exists a' \in \mathcal{L} \cup \{\$\}$ *s.t.* (q, a') *is a transition in* S **then**
      $\xi' \longleftarrow$ the prefix of ξ before $\langle q, a \rangle$ (exclusive);
3 **if** $cont(S, \xi', a') = T$ **then**
4 $freq(q, a) \longleftarrow freq(q, a) + D(d)$;
5 Compute *total* and *out* as in Algorithm 1 lines 4-5;
6 $G \leftarrow$ ct-generator based on S and where $\forall q \in \mathcal{Q}, a \in \mathcal{L} \cup \{\$\}$ $t\text{-}prob(q, a) = \frac{1}{out(q)}$ if $total(q) = 0$, otherwise $t\text{-}prob(q, a) = \frac{freq(q,a)}{total(q)}$;
return G;

Algorithm 2: Algorithm for `OPT-GEN` (constraints, ct-generators)

Finding an optimal binary ct-generator. To solve `OPT-GEN` for ct-generators, we assume that the schema has a particular property, namely that it is "binary". A schema is *binary* if for each state of each automaton in the schema, there are at most two possible transitions. We will discuss the case of non-binary schemas afterwards. Recall that $\mathrm{FP}^{\mathrm{NP}}$ is the class of problems solvable by polynomial-time computation algorithms that are allowed calls to an NP oracle. Formally, we show:

Theorem 5.3. *Given a binary schema with constraints S and a corpus, we can find an optimal ct-generator in time FP^{NP}.*

Proof. Algorithm 2 computes the optimal ct-generator in time polynomial in the size of S, while making calls to an oracle *cont* that performs continuation tests. Generally, Algorithm 2 is very similar to Algorithm 1, except that the frequency of taking a transition is only recorded in situations where there exists a second optional transition, which according to the oracle *does not lead to a dead end*. The time complexity of the algorithm follows from the complexity of Algorithm 1, and the calls for *cont* in line 3.

It is still to be shown that the output G of Algorithm 1 has maximum quality among all the ct-generators that conform to S. This proof is very similar to the proof of Proposition 4.1, only that this time when we maximize the term $quality(G', D) = \prod_{(q,a) \text{ in } S} p(q, a)^{freq(q,a)}$, $freq(q, a)$ refers to the number of times the transition (q, a) was taken when there was a second choice with continuation. In other cases every ct-generator must have chosen the only possibility with probability 1. □

Generation time. Without constraints, it was trivially the case that a document was generated in time linear in its size. However, for continuation-test generators it is no longer the case, because of the complexity of continuation-tests:

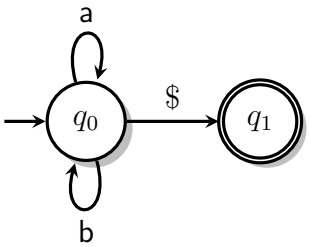

Figure 3: A_3 - A DFA with 3 Choices

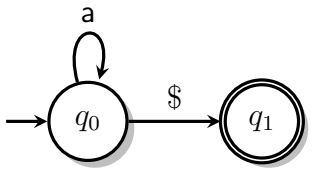

Figure 4: The DFA $A_{tradeoff}$

Proposition 5.4. *The generator produced by Algorithm 2 generates a document in time exponential in the size of the document (assuming $P \neq NP$).*

Termination probability. The question of whether the generator returned by Algorithm 2 stops with probability 1 is more complicated than its counterpart for the unconstrained case. In particular, the result in [10] cannot be used here, because it relies on the probability of termination from each given label being fixed; however, in presence of constraints, this probability may vary according to the context in which the label appears in the partial XML tree. The following example gives intuition for the difficulty.

Example 5.5. Let $\mathcal{L}_\mathrm{i} = \{r, \mathsf{c}, \mathsf{d}\}$, $\mathcal{L}_\mathrm{l} = \{\mathsf{a}, \mathsf{b}\}$, and S a schema with constraints, s.t. $L(r) = \mathsf{a}^+\mathsf{b}^+(\mathsf{c} \mid \varepsilon)\$$, $L(\mathsf{c}) = \mathsf{ad}$, $L(\mathsf{d}) = \{\mathsf{ddd}, \varepsilon\}$ and the only constraints are $\mathsf{b} \subseteq \mathsf{a}$, $uniq(\mathsf{b})$. Let the corpus D contains a single document, <r><a/><b/></r>. Then during the verification of D, we never get c or d, and so assign uniform probability distributions to the transitions of their automatons (we could use any other arbitrary probability assignment). specifically, this gives a probability of $\frac{1}{2}$ for deriving ddd from d. Then, during generation, we could start by generating one a and two b's; as a result of the inclusion constraint, the generator must then generate a c, to have another a as its child; but then the termination probability from d is less than 1, as shown in a similar example of [10].

This particular example of termination with probability < 1 uses transitions that are not used in the validation phase; it is still open whether eliminating such transitions with no evidence in the corpus would ensure almost certain termination, or whether there is a way of assigning probabilities to these transitions that gives the same guarantee.

We conclude the discussion on ct-generators by a remark on non-binary choices.

Non-binary choices. We have assumed so far in this section that the schemas were binary. When this is not the case, finding the optimal generator is still open. We study here two options for handling the non-binary case via slight variations of the model: (1) turning the choices into binary choices, and (2) keeping probabilities for all combinations of valid choices. We present these options by example.

Consider the following constrained schema. The deriving automaton A_3 of the root label $r \in \mathcal{L}_\mathrm{i}$ is shown in Figure 3 with $\mathsf{a}, \mathsf{b} \in \mathcal{L}_\mathrm{l}$. ($A_3$ accepts $(\mathsf{a} \mid \mathsf{b})^*\$$.) Observe it has a ternary choice. We also assume that b has a key constraint and domain cardinality 1.

Consider (1). We show two ways of turning the ternary choice into a binary one. A third possibility is not considered here.

First, one decides whether a is produced or not and then (if an a is not produced) whether b is produced or whether we are done with the children of r. We use a probability

assignment *t-prob*: we choose to produce a with probability $t\text{-}prob(q_0, \mathsf{a})$ and to produce b (given that we have not produced a) with probability $t\text{-}prob(q_0, \mathsf{b})$. Of course, as before, we use continuation tests to avoid reaching dead ends during generation, and in the probability learning, as in Algorithm 2. Alternatively, one can choose whether we are done with r first, and, if we are not done, whether we produce a or b. This yields $t\text{-}prob'$.

Consider the single document corpus <r><a/><b/></r>. We can compute the transition probabilities: $\begin{cases} t\text{-}prob(q_0, \mathsf{a}) = \frac{1}{3} & t\text{-}prob'(q_0, \$) = \frac{1}{3} \\ t\text{-}prob(q_0, \mathsf{b}) = 1 & t\text{-}prob'(q_0, \mathsf{a}) = \frac{1}{2} \end{cases}$

This yields, for the probability of generating the corpus: $\frac{1}{3} \times \frac{2}{3} \times 1 \times \frac{2}{3} = \frac{4}{27}$ using the first alternative, and $\frac{2}{3} \times \frac{1}{2} \times \frac{2}{3} \times \frac{1}{2} \times \frac{1}{3} = \frac{1}{27}$ using the second one: the quality of the generator depends of the way the choice has been made binary.

Now consider (2). We keep the ternary choices but assign a probability to each possible subset of the transitions of size more than 1. For the example, this yields:

a, b, $ are all available	**only a, $ are available**
$t\text{-}prob(q_0, \mathsf{a}) = \frac{1}{2}$	$t\text{-}prob'(q_0, \mathsf{a}) = 0$
$t\text{-}prob(q_0, \mathsf{b}) = \frac{1}{2}$	
$t\text{-}prob(q_0, \$) = 0$	$t\text{-}prob'(q_0, \$) = 1$

which gives a probability of generating the corpus of $\frac{1}{2} \times \frac{1}{2} \times 1 = \frac{1}{4}$.

In both cases, we can obtain an optimal generator *for this particular class of generators*. For (1), this suffers from the inelegance of the arbitrary ordering of the transitions that is chosen and affects the outcome. For (2), this may result in a large number of parameters.

5.2 Restart Generators

We next consider restart generators. First, we show that given a generated document skeleton, we can check its validity efficiently (and if invalid, restart). Then, however, we show that the *number of restarts* may be unboundedly large; and this can hold particulary for generators that are optimal (i.e. best fit to the corpus).

Proposition 5.6. *Given a schema with constraints $\langle S^{\mathrm{u}}, C\rangle$, and a document skeleton d valid for S^{u}, verifying the validity of d with respect to $\langle S^{\mathrm{u}}, C\rangle$ can be done in PTIME.*

Proof. We consider again the schema satisfiability test from [12], which is tested through the satisfiability of the formula $\varphi \wedge \psi$. The variables $x_1, ..., x_n$ in the formula represent the numbers of occurrences of nodes labeled with $l_1, ..., l_n$. In this case, if $d = (V, E)$, we will take the assignment of each x_i to be $a_i^d = |\{v \in V \mid label(v) = l_i\}|$. Since d is valid for S^{u}, we know that this assignment for $x_1, ..., x_n$ satisfies φ, which is the part of the formula that expresses the validity of the document for the schema S^{u}.

It is left to find a satisfying assignment for ψ, that expresses validity with respect to the constraints in C. For that we must also find an assignment for the variables $v_1, ..., v_n$ that represent the number of unique values for each label. We can ignore here inner node labels, since we only have values on the leaves (unlike [12]). Once we find values for $v_1, ..., v_n$ we can be sure that there exists a valid assignment for the leaf values, for the generated

document skeleton. Let us initialize a directed graph $G = (V, E)$, such that there is a node $n(v_i)$ for every variable, node $n(0)$ and $n(a_i^d)$ for $1 \leqslant i \leqslant n$, and add the edges $(n(0), n(v_i))$, $(n(v_i), n(a_i^d))$ for each i. In G a directed edge (a, b) expresses that $a \leqslant b$. ψ connects, using $\wedge$, sub-formulas of the 4 following types:

$$(1)\ v_i \leqslant x_i \qquad (2)\ v_i = 0 \leftrightarrow x_i = 0 \qquad (3)\ v_i = x_i \qquad (4)\ v_i \leqslant v_j$$

In addition, for each domain constraint $l_i \in dom(l_i)$ we add $v_i \leqslant |dom(l_i)|$ (recall that we only need to verify validity w.r.t. constraints of finite domains).

We will generate the sub-formulas according to the constraints in the schema, while using the value assigned to each x_i instead of a variable, and also updating G in the process, as follows. We can ignore sub-formulas of the first kind, as they were already embedded in the initialization of G; for sub-formulas of the second kind, if indeed $x_i = 0$, we will add the edge $(n(v_i), n(0))$; otherwise we will add $(n(1), n(v_i))$, adding a new node $n(1)$ if necessary; for $v_i = x_i$ we will add $(n(a_i^d), n(v_i))$; for $v_i \leqslant v_j$ we will add $(n(v_i), n(v_j))$; and finally for $v_i \leqslant k$ we will add $(n(v_i), n(k))$, adding the node $n(k)$ if necessary. Then we will take $G^* = (V, E^*)$, the transitive closure of G.

We claim that ψ is satisfiable iff in G^* there exists no edge $(n(k), n(k'))$ s.t. $k' < k$.

For the one direction, assume that there exists no such edge $(n(k), n(k'))$, and let us assign to each v_i the minimal k such that $(n(v_i), n(k)) \in E^*$ (i.e., the lowest upper bound for v_i). By the initialization there must exist such a k. It is straightforward to verify that every sub-formula of ψ is satisfied. E.g., assume that $v_i = x_i$. By the construction, $(n(v_i), n(a_i^d))$ and $(n(a_i^d), n(v_i))$ are in E, E^*. Assume by contradiction that v_i is assigned a value $k < a_i^d$; then $(n(v_i), n(k)) \in E^*$ and thus also $(n(a_i^d), k)$, which gives a contradiction. Assigning v_i a value $k > a_i^d$ contradicts the choice of minimal upper bounds as values.

For the other direction, assume that there exists such edge $(n(k), n(k'))$. This edge is in E^*, thus by the definition of transitive closure there is a path from $n(k)$ to $n(k')$ in E; this path represents a sequence of inequalities $k \leqslant y_1, y_1 \leqslant y_2, ..., y_t \leqslant k'$, and clearly, those inequalities cannot all be satisfied together. Thus ψ is not satisfiable.

Finally, the complexity of generating G, generating G^*, and checking for an edge $(n(k), n(k'))$ s.t. $k' < k$ is polynomial in n (the number of labels) and C, and logarithmic in the size of the document skeleton (as we also need to save a representation for the number of nodes in the skeleton). Thus the skeleton validity test is in PTIME. □

The quality of an r-generator vs. the restart overhead. We next examine how many times we will restart (i.e., what is the expected the number of generated invalid documents). In particular, we show that there is a tradeoff between the optimality of an r-generator in terms of quality, and its restart overhead.

Example 5.7. Consider a simple schema $S_{tradeoff}$, which consists of a root label r, whose automaton is depicted in Figure 4. The regular language of this automaton is $\mathsf{a}^*\$$. Let $\mathcal{L}_l = \{\mathsf{a}\}$ and let the set of constraints $C = \{uniq(\mathsf{a}), \mathsf{a} \in \{0\}\}$ (a can have only one value, 0).[6] Consider a document corpus which consists only of the document d, whose root has a single child a with the value 0.

[6] We could also construct more complicated examples, where the value domains are infinite.

Now, the only probabilistic parameter that can be varied in an r-generator is the probability to choose the transition from q_0 to itself. Denote this parameter by α. In this case, maximizing the quality of the generator means maximizing the probability for generating d. Since this is an r-generator, generating d means (perhaps) generating some finite number of invalid document skeletons, restarting after each time, and then generating d in the last invocation of the generator. The probability of generating an invalid document skeleton in a single invocation of the generator is the probability of choosing (q_0, a) twice or more, i.e., α^2. The probability of generating d in a single invocation is the probability of choosing (q_0, a) once and $(q_0, \$)$ once, i.e., $\alpha \cdot (1-\alpha)$. Then the total probability of generating d is the probability of generating d in the first invocation, in the second one, etc., that is: $\sum_{k=0}^{+\infty} \alpha(1-\alpha)(\alpha^2)^k =_{(*)} \alpha(1-\alpha)\frac{1}{1-\alpha^2} = \frac{\alpha}{1+\alpha}$. Since we use the formula for converging infinite geometric series, the transition $(*)$ requires that $\alpha < 1$ (if $\alpha = 1$, $\Pr(d)$ is obviously 0). We would like to find the maximum of probability of generating d, and as the function is monotonically increasing in $[0, 1)$, the closer α gets to 1 the higher the quality of the generator is. Let us choose α to be $1 - \varepsilon$, for some arbitrarily small $\varepsilon > 0$.

What is the expectation for the number of times our "nearly-optimal" generator restarts? The probability of generating a valid document in one invocation is $1-\alpha^2$. Define a random variable X that measures the number of times the r-generator restarts. Since the different invocations of the r-generator are independent, X has a geometric distribution, and we get the following (let $\varepsilon' = 2\varepsilon - \varepsilon^2$): $\mathbb{E}[X] = \frac{1-(1-\alpha^2)}{1-\alpha^2} = \frac{(1-\varepsilon)^2}{1-(1-\varepsilon)^2} = \frac{1-\varepsilon'}{\varepsilon'}$. This shows that the expected number of restarts tends towards $+\infty$ as $\varepsilon \to 0$.

Remark 5.8. A conclusion from the example is that maximizing the likelihood of the corpus may not be the best measure for the quality of r-generators, and finding different quality measures for such generators will be considered in future research.

Remark 5.9. An additional conclusion from this example is that Algorithm 1 does not return the optimal r-generator. Intuitively, unlike in a p-generator, in an r-generator we only need to maximize the relative probability of the corpus w.r.t. the set of *valid* documents.

6 Related Work

Various models for probabilistic XML documents exist in the literature (e.g. [11, 3]); see [4] for a review of such models and a comparison of their expressiveness. The model that we consider here is not of a probabilistic document but rather of a probabilistic *schema*; in particular our model allows to define infinitely many documents, in contrast to the finitely many documents (*worlds*) in the models above. Probabilistic schemas were also considered in a recent work [6] that suggested the use of recursive Markov chains [13] for modeling and querying probabilistic XML. The model of [6] can be seen as a straightforward extension of p-generator where global states and labels are uncoupled.

As noted in Remark 3.2, the different models presented in this paper, including nondeterministic, probabilistic and constrained generators, can also be captured by Active XML [2] and tree rewriting. This suggests a variety of new interesting research questions that can be further studied in future work.

The starting point of this work assumes that we are given a schema; there are many works on *schema inference* from a corpus of documents (e.g. [21, 7, 9, 19, 16]). These works

complement our work in two senses: first, we can use the inferred schemas as inputs; second, our results can be used to measure quality of inferred schemas, based on the quality of the optimal generator conforming to them. There are other measurements for schemas quality (see [5] for a recent work), and combining them with our measurement is an intriguing future research task.

Our work also has strong connections with the recent work of [12, 14]. They consider satisfiability tests for XML schemas with constraints, and prove that these tests are NP-complete; we used an adaptation of this result to show NP-completeness of the continuation tests. Note that in contrast to our work, the work of [12, 14] focuses on satisfiability, and thus the model that is used there is not probabilistic.

On the technical level, our work is also related to other (non-XML) probabilistic models. In particular, *Probabilistic Context-Free Grammars* (PCFGs) [17, 10] are a common model for the probabilistic generation of strings, used heavily in natural language processing, bioinformatics, and more. We have noted that our algorithm for the non-constrained case is inspired by [10]; we are not aware of an equivalent result in the presence of constraints on strings. Applying our results to this area is an intriguing future research task.

7 Conclusion

We have studied the problem of finding an optimal probabilistic model for a given corpus and type of XML documents. We have shown how to view the model as a probabilistic generator. We have provided elegant solutions both for the case without and with constraints. For the latter, we have studied two kinds of generators, ct-generators and r-generators, studied algorithms for finding optimal generators, and analyzed the advantages and disadvantages of both kinds.

We have focused in this paper on generators for XML document *skeletons*, i.e., without data values. As future work we will further explore the generation of data values. Another challenging research direction follows from the two different kinds of generators that we have introduced in presence of constraints. Recall that a ct-generator always generates valid documents (but generation is costly), while an r-generator avoids the cost of continuation test but may restart often. This suggests combining both approaches to obtain better performing generators, that generate faster large numbers of valid documents. In particular, one would like to estimate how likely is a restart and use a continuation test only when it pays to avoid costly restarts. This is left for future research. More possibilities for future research lie in, e.g., considering other kinds of constraints such as non-unary keys (in a limited manner, as [14] proves that satisfiability in this case is undecidable), different or random orders of generation, parallelism, and more. Some of these directions may be studied by further extending our model to Active XML. Last but not least, it would be interesting to experiment with the generators that were formally introduced here.

Acknowledgments. We would like to thank Yann Ollivier for insightful comments. This work has been supported in part by the Advanced European Research Council grant Webdam on Foundations of Web Data Management, grant agreement 226513 (`http://webdam.inria.fr/`), by the EU project Mancoosi, by the Israel Science Foundation and by the US-Israel Binational Science Foundation.

References

[1] S. Abiteboul, O. Benjelloun, and T. Milo. The Active XML project: an overview. *VLDB J.*, 17(5), 2008.

[2] S. Abiteboul, P. Bourhis, A. Galland, and B. Marinoiu. The AXML artifact model. In *TIME*, 2009.

[3] S. Abiteboul, T.-H. H. Chan, E. Kharlamov, W. Nutt, and P. Senellart. Aggregate queries for discrete and continuous probabilistic XML. In *ICDT*, 2010.

[4] S. Abiteboul, B. Kimelfeld, Y. Sagiv, and P. Senellart. On the expressiveness of probabilistic XML models. *VLDB J.*, 18(5), 2009.

[5] T. Antonopoulos, F. Geerts, W. Martens, and F. Neven. Generating, sampling and counting subclasses of regular tree languages. In *ICDT*, 2011.

[6] M. Benedikt, E. Kharlamov, D. Olteanu, and P. Senellart. Probabilistic XML via Markov chains. *PVLDB*, 3(1), Aug. 2010.

[7] G. J. Bex, W. Gelade, F. Neven, and S. Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. In *WWW*, 2008.

[8] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise DTDs from XML data. In *VLDB*, 2006.

[9] G. J. Bex, F. Neven, and S. Vansummeren. Inferring XML schema definitions from XML data. In *VLDB*, 2007.

[10] Z. Chi and S. Geman. Estimation of probabilistic context-free grammars. *Comput. Linguist.*, 24(2), 1998.

[11] S. Cohen, B. Kimelfeld, and Y. Sagiv. Incorporating constraints in probabilistic XML. In *PODS*, 2008.

[12] C. David, L. Libkin, and T. Tan. Efficient reasoning about data trees via integer linear programming. In *ICDT*, 2011.

[13] K. Etessami and M. Yannakakis. Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations. *JACM*, 56(1), 2009.

[14] W. Fan and L. Libkin. On XML integrity constraints in the presence of DTDs. *JACM*, 49(3), 2002.

[15] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: a system for extracting document type descriptors from XML documents. In *SIGMOD*, 2000.

[16] W. Gelade, T. Idziaszek, W. Martens, and F. Neven. Simplifying XML schema: Single-type approximations of regular tree languages. In *PODS*, 2010.

[17] K. Lary and S. J. Young. The estimation of stochastic context-free grammars using the inside-outside algrithm. *Computer, Speech and Language*, 4, 1990.

[18] W. Martens, F. Neven, T. Schwentick, and G. J. Bex. Expressiveness and complexity of XML schema. *ACM Trans. Database Syst.*, 31(3), 2006.

[19] T. Milo and D. Suciu. Type inference for queries on semistructured data. In *PODS*, 1999.

[20] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Internet Technol.*, 5(4), 2005.

[21] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *SIGMOD*, 1998.