Certain and Possible XPath Answers

Sara Cohen
School of Computer Science and Engineering
Hebrew University of Jerusalem
Jerusalem, Israel
sara@cs.huji.ac.il

Yaacov Y. Weiss
School of Computer Science and Engineering
Hebrew University of Jerusalem
Jerusalem, Israel
yyweiss@cs.huji.ac.il

ABSTRACT

Formulating an XPath query over an XML document is a difficult chore for a non-expert user. This paper introduces a novel approach to ease the querying process. Instead of specifying a query, the user simply marks *positive examples* $\mathcal{X}^+$ of nodes that fit her information need. She may also mark *negative examples* $\mathcal{X}^-$ of undesirable nodes. A deductive method, to suggest additional nodes that will interest the user, is developed in this paper.

To be precise, a node y is a *certain answer* if every query returning all positive examples $\mathcal{X}^+$, and not returning any negative example from $\mathcal{X}^-$, must also return y. Similarly, y is a *possible answer* if there exists a query returning $\mathcal{X}^+$ and y, while not returning any node in $\mathcal{X}^-$. Thus, y is likely to be of interest to the user if y is a certain answer, and unlikely to be of interest if y is not even a possible answer. The complexity of finding certain and possible answers, with respect to various classes of XPath, is studied.

It is shown that for a wide variety of XPath queries (including child and descendant axes, wildcards, branching and attribute constraints), certain and possible answers can be found efficiently, provided that $\mathcal{X}^+$ and $\mathcal{X}^-$ are of bounded size. To prove this result a novel algorithm is developed.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*Query languages*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Query formulation*

General Terms

Algorithms, Theory

1. INTRODUCTION

Formulating an XPath query over an XML document is a difficult (if not impossible) chore for a non-expert user. In order to correctly formulate a query, the user must be intimately familiar with the XPath syntax. In addition, the user must have complete familiarity with the XML schema of the document she is querying. However, often users are not familiar with all details of relevant XML schema. In fact, they may be completely oblivious of the XML structure, as they may view XML documents as HTML, via a style-sheet transformation. Obviously, lack of knowledge of the query language and the document format completely preclude non-expert users from querying XML documents.

This paper introduces a novel approach to ease the querying process. The underlying assumption of this paper is that the user has an information need that can be expressed as an XPath query of some type. In fact, the user may not even be interested in deriving all answers to the XPath query, but, rather, in finding *enough answers* of interest. We assume that the user does not have the expertise to formulate her query. Therefore, instead of specifying a query, the user simply marks *positive examples* $\mathcal{X}^+$ (and possibly *negative examples* $\mathcal{X}^-$) of nodes that fit (resp. do not fit) her information need. Note that she may even mark these in the HTML rendering of the XML document, as long as the underlying XML node can be recognized by the system. The system then suggests additional nodes that might be of interest to the user, given the user examples.

Deciding which nodes may interest the user is challenging. We introduce a precise semantics to this question, by defining certain and possible answers. Intuitively, a node y is a *certain answer* if every XPath query returning all positive examples $\mathcal{X}^+$, and not returning any negative example from $\mathcal{X}^-$, must also return y. Similarly, y is a *possible answer* if there exists an XPath query returning $\mathcal{X}^+$ and y, while not returning any node in $\mathcal{X}^-$. Thus, assuming the the user's underlying information need is expressible in XPath, y is likely to be of interest to the user if y is a certain answer, and unlikely to be of interest if y is not even a possible answer.

In this paper we consider XPath queries with child and descendant axes, wildcards, branching, and attribute conditions of the form $@_a\ \theta\ c$ where a is the name of an attribute, θ is one of $=, \leq, \geq$ and c is a constant. We demonstrate the notions of certain and possible answers with a simple example, below. Example 3.6 is more intricate, and takes into consideration variations in the document structure.

EXAMPLE 1.1. *Consider the XML document in Figure 1, listing books for sale. We have numbered the book-labeled nodes for easy reference. Suppose Nodes 1 and 2 (hereafter referred to as x_1 and x_2) are chosen by the user as positive examples, i.e., $\mathcal{X}^+ = \{x_1, x_2\}$. Then, for example, x_4 is a* possible answer *as there is an XPath query returning*

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EDBT/ICDT '13, March 18 - 22 2013, Genoa, Italy
Copyright 2013 ACM 978-1-4503-1598-2/13/03 ...$15.00.

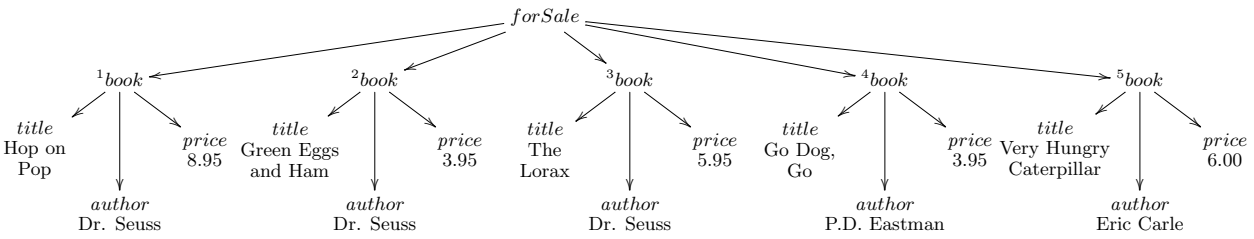

Figure 1: XML document with a simple structure, listing books for sale.

x_1, x_2, x_4. *(In fact, without negative examples, every node is a possible answer, as witnessed by the XPath query returning all nodes in the document.) Node* x_3 *is a* certain answer *as every query returning* x_1 *and* x_2 *will also return* x_3. *To see this observe that the* most precise *XPath query returning* x_1 *and* x_2 *will return books with author "Dr. Seuss" and price between 3.95 and 8.95, and* x_3 *satisfies both these properties.*

Suppose now that x_5 *is chosen by the user as a* negative example*, i.e.,* $\mathcal{X}^- = \{x_5\}$. *It is not difficult to see that* x_4 *is no longer a possible answer, as any XPath query returning* x_1 *and* x_2*, cannot differentiate between* x_4 *and* x_5.[1]

Determining certain and possible answers is a nontrivial problem, and this is especially true for documents with less regular structures. In this paper we study the complexity of these problems. Our main contributions are:

- We develop a *novel method to ease the querying process* via the notions of certain and possible answers. We emphasize that this method is suitable for non-expert users who cannot formulate XPath queries, and who are interested in viewing sufficiently many (but not necessarily all) results for their information needs.
- The problems of determining certain and possible answers are abstracted to the more general *query existence problem*: given $\mathcal{X}^+$ and $\mathcal{X}^-$, determine whether there exists a query that returns all of $\mathcal{X}^+$ and none of $\mathcal{X}^-$.
- We introduce *query automata* as a compact summary of queries returning the positive examples $\mathcal{X}^+$.
- We characterize query existence by an intricate set-based algorithm, which allow us to determine whether there exists a query in the language of an automata that does not return any of $\mathcal{X}^-$.
- Finally, by proving a bound on the size of the sets that our algorithm produces, we show that determining certain answers, possible answers and query existence are all in polynomial time in the size of the document, provided that $\mathcal{X}^+$ and $\mathcal{X}^-$ are of bounded size. This bound on $\mathcal{X}^+$ and $\mathcal{X}^-$ is natural, as these sets are hand-picked by the user. Our results hold for a wide variety of XPath queries, including child and descendant axes, wildcards, branching and attribute constraints.

[1]We consider a fragment of XPath that does not allow attribute conditions involving disjunction or $\neq$. If these were allowed, then x_4 would remain a possible answer. However, such an XPath class would often yield useless results in the context of our paper, as we can use these constructs to list all allowed or forbidden values.

This paper is structured as follows. Section 2 discusses related work. Section 3 introduces necessary definitions, including the formal definitions of the problems of interest. Sections 4–6 present our polynomial algorithm for XPath queries with child and descendant axes, as well as wildcards. Extensions that allow for attributes and branching are presented in Section 7. Finally, in Section 8, we conclude.

2. RELATED WORK

The notions of certain and possible answers have been studied extensively in the context of data exchange, e.g. [5, 12], and are applied when the queried database is not known precisely. In this paper, we take a completely different approach. In particular, we assume that the database is known, but the user query is only partially known (via the information conveyed by $\mathcal{X}^+$ and $\mathcal{X}^-$). To the best of our knowledge certain and possible answers have not been studied before in this context.

While this paper is the first to consider the problems of determining certain and possible answers for unknown queries, there are three related areas of past work. First, there is a wealth of articles which aim at simplifying the querying process for the non-expert user. Second, there are several works that aim at explaining the presence (or absence) of query results. Third, the problem of learning XPath queries (or tree patterns) from user examples has been studied in the past. We discuss each of these areas below.

XML Querying for the Non-Expert User. There seems to be a consensus that querying XML by a non-expert user is difficult. Hence, there are many papers that aim at simplifying this task. Most work on this topic take one of the two following approaches: use of keywords or flexible interpretations of queries.

There has been considerable research on *XML keyword search,* e.g., [14,16,18], suggesting that users formulate their queries over XML as keyword searches. The burden is then upon the system to return results that will be of interest to the user. Different semantics and ranking methods have been developed. While keyword searches are extremely simple for formulation, they are generally not very expressive. Thus, for example, users cannot use inequalities to constrain the results, nor can they explicitly state the relationships between various keywords.

A complementary approach is to allow users to formulate queries as trees, and then *flexibly interpret* these query trees, to return answers that are somewhat similar to the user's query, e.g., [7,22,24]. This results in a more expressive

query language, which is both advantageous (since it allows the user to more precisely express her information need), as well as disadvantageous (as it makes querying again less intuitive).

The approach suggested in this paper is complementary to both of the above. We do not require the user to formulate her information need as a query at all, but, instead, she marks positive and negative examples from the document, and then the system automatically finds additional nodes of interest.

Explaining Query Results. Recently, there has been considerable interest in explaining query results to a user, i.e., in justifying why certain tuples do, or do not, appear in a query result. Most of this work, e.g., [10,15,17,20], assumes that the query is given. Answers take the form of provenance of tuples, or explanations as to which query clause caused tuples to be eliminated. This is very different from our framework in which only the answers are given, and not the queries.

Somewhat more related are [11, 25, 26]. In [26], the focus is on generating a query that almost returns a given output (with a precise definition for "almost"). In [25] missing tuples in a query result are explained by generating a query that returns the query result, as well as the missing tuples. Finally, in [11] the problem of synthesizing view definitions from data is studied. There are many differences between these works and ours. One obvious difference is that these works consider relational databases and conjunctive queries, and the results are not immediately applicable to other settings. However, perhaps the most significant difference is that these papers focus on deriving queries, while we focus on determining whether nodes are potentially interesting answers for the user. This is a crucial difference, as our notions of certain and possible answers take into consideration a huge (actually infinite) number of different queries, at the same time. To be precise, finding all queries that return $\mathcal{X}^+$, while not returning $\mathcal{X}^-$ is intractable, due to the potentially mammoth number of such queries; whereas we show that determining certain and possible answers can be solved in polynomial time.

Learning XPath and Tree Patterns. There is a wealth of work on using machine learning techniques to learn XPath queries or tree patterns from positive and negative examples. Most often, these works are in the context of wrapper induction. Below we discuss some of the more theoretical related work on learning queries.

In [23], an algorithm for learning XPath queries from examples was presented. The algorithm was shown to be polynomial for certain types of XPath queries (e.g., the queries must be anchored), when only positive examples are given. The presence of negative examples rendered their learning problem to be NP-complete. The goal of [23] is rather different from that of this paper (and thus, the techniques employed are also different). In particular, one of the main results of [23] states that for every XPath query there exists a *characteristic set* (i.e., a set of XML fragments with marked positive examples) for which the query can be learned, using their algorithm, in polynomial time. Our goal is different, as we do not expect to learn the user's query (which might require many examples), but rather to find additional answers that are consistent with the input provided by the user.

Slightly more related are the works of [1, 2, 4, 6, 8, 9, 19, 21]. The goal of these papers is to learn a query (e.g., in the classes XPath, tree automata, tree patterns, (k,l)-contextual tree languages), usually using equivalence or membership questions. Intuitively, equivalence questions provide the user with a query (or examples of results of a query) and ask the user whether the computer-provided query is equivalent to that of the user. Membership questions provide the user with an example, and ask the user whether this example should be marked positively. Equivalence questions are considered more natural for a user, as membership questions often provide the user with a document that is completely different from the document of current interest, and ask questions about this different document. Many negative results have been shown, most notably, that tree automata cannot be learned in polynomial time with equivalence questions [4], as well as a similar negative result for learning even simple fragments of XPath [8, 9]. Several of these works [19, 21] have been experimentally tested against wrapper induction benchmarks, and have been shown to work extremely well.

The results of [1, 2, 4, 6, 8, 9] are, once again, completely different from (and complementary to) those of ours. Most significantly, we do not ask the user questions of any type, nor do we attempt to precisely learn the user's query of interest. Instead, we generalize the user's examples by developing an algorithm that determines whether a node in the document is returned by the intersection (resp. union) of all XPath queries that are consistent with the user examples, in order to provide the user with additional certain (resp. possible) answers of interest. Note that we neither arbitrarily choose some consistent XPath query to return to the user,[2] nor do we burden the user with questions. When the precise user's query is required for some purpose, our approach is not sufficient (and instead, those of [8,9] can be used). However, in many contexts, the user is simply interested in seeing *enough* nodes of interest (e.g., for imprecise querying of XML). In these cases, we expect our approach to be very useful. The negative results on learnability from the above papers strengthen the motivation for our approach, as precisely learning the user's query will often not be possible in polynomial time, even when the user is willing to answer questions. Thus, [1, 2, 4, 6, 8, 9] do not provide solutions for ad-hoc querying, which is the goal of this paper.

3. DEFINITIONS

XML Documents. Let $\mathcal{N}_e$ and $\mathcal{N}_a$ be fixed infinite sets of *element* and *attribute names*. Let $\mathcal{D}_<$ be a fixed infinite and totally ordered *domain of values* and $\mathcal{D}$ be a fixed infinite (non-ordered) *domain of values*. We model XML documents as unranked trees, where each node has a label taken from $\mathcal{N}_e$. In addition, a node can have any number of attributes, with names from $\mathcal{N}_a$ and values in $\mathcal{D}_< \cup \mathcal{D}$. We formally define documents below.

DEFINITION 3.1 (DOCUMENT). *A* document *is a tuple* $d = (\mathcal{V}, \mathcal{E}, r, lab, @)$ *where*

- $(\mathcal{V}, \mathcal{E}, r)$ *is an unranked tree with nodes* $\mathcal{V}$*, edges* $\mathcal{E}$ *and root* $r \in \mathcal{V}$*;*

[2]Our algorithm is capable of returning such an arbitrary XPath query if desired.

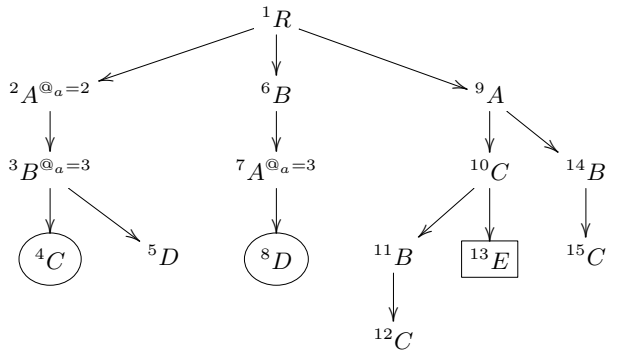

Figure 2: XML document d.

- *lab is a function from $\mathcal{V}$ to $\mathcal{N}_e$, which associates each node with a label;*
- *for each $a \in \mathcal{N}_a$, $@_a$ is a partial function from $\mathcal{V}$ to $\mathcal{D}$ or from $\mathcal{V}$ to $\mathcal{D}_<$, which associates nodes with attributes and values.*

REMARK 1. *To simplify the presentation, our nodes do not have data associated directly with them, rather only via attributes. To model nodes with data (e.g., as in Figure 1), we simply assume some attribute $@_{val}$ for each node, with which the data of the node is associated.*

A small document d appears in Figure 2. We will use d as a running example throughout the paper. The nodes are numbered for easy reference; we will refer to these nodes as $x_1, \ldots, x_{15}$. (The circles and box can be ignored for the time being.) Note that nodes x_2, x_3, x_7 have associated attributes.

Query Syntax. We consider XPath queries with child and descendant axes, wildcards, branching and attribute constraints, as defined next.

DEFINITION 3.2 (QUERY). *A* query *is an expression q of the form*

$$q ::= \quad \downarrow l \quad | \quad \Downarrow l \quad | \quad q/q \quad | \quad q[q] \quad | \quad q[@_a\ \theta\ c]\,,$$

where $\downarrow$ and $\Downarrow$ represent the child and descendant axes, respectively, $l \in \mathcal{N}_e \cup \{*\}$ is a label or wildcard, q/q is the concatenation of two queries, $q[q]$ is used to express branching conditions and $@_a\ \theta\ c$ constrains the value of attribute $a \in \mathcal{N}_a$, with $\theta \in \{\leq, \geq, =\}$ and $c \in \mathcal{D}_< \cup \mathcal{D}$.

Of course, we will assume that θ is "=" if $c \in \mathcal{D}$, i.e., if c is part of an unordered domain. Note that for technical reasons, we will not allow queries to contain duplicate occurrences of the same branching conditions, e.g., $\downarrow D[\Downarrow A][\Downarrow B]$ is allowed, while $\downarrow D[\Downarrow A][\Downarrow A]$ is not.

To make the presentation clear, we use lowercase letters from the end of the alphabet $(z, y, \ldots)$ to denote nodes in a document. We use lowercase letters from the beginning of the alphabet $(a, b, \ldots)$ to denote names of attributes and uppercase letters from the beginning of the alphabet to denote element names $(A, B, \ldots)$.

Query Semantics. We formally define the semantics of our queries, by induction on the structure of a query.

DEFINITION 3.3 (RESULTS). *Let d be a document, x and y be nodes in d, and q be a query. We say that q* returns *y from x, written $d \models q(x, y)$, if*

- *$q =$ "$\downarrow l$": y is a child of x and either $l = *$ or $lab(y) = l$;*
- *$q =$ "$\Downarrow l$": y is a descendant of x and either $l = *$ or $lab(y) = l$;*
- *$q =$ "q_1/q_2": there is a node z such that $d \models q_1(x, z)$ and $d \models q_2(z, y)$;*
- *$q =$ "$q_1[q_2]$": $d \models q_1(x, y)$ and there is some z such that $d \models q_2(y, z)$*
- *$q =$ "$q_1[@_a\ \theta\ c]$": $d \models q_1(x, y)$ and $@_a$ is defined on y with a value c_a such that $c_a\ \theta\ c$.*

Finally, we define the result of q on d as

$$q(d) = \{x \mid d \models q(r, x)\}\,,$$

where r is the root node of d.

EXAMPLE 3.4. *To demonstrate our query semantics, we present several queries with their results over d (Figure 2):*

$$\downarrow * \ / \downarrow * \ / \downarrow * \,(d) = \{x_4, x_5, x_8, x_{11}, x_{13}, x_{15}\}$$
$$\Downarrow A[@_a \leq 3] \,/\, \Downarrow C(d) = \{x_4\}$$
$$\downarrow * \,[\downarrow B][\downarrow C/\downarrow B] \,/\, \Downarrow E(d) = \{x_{13}\}\,.$$

Classes of Queries. We will be interested in various classes of XPath queries. To specify a particular class, we will use the format $\mathcal{C}(F)$ where F is a subset of features among $\{*, \downarrow, \Downarrow, [\ ], @\}$. Thus, F lists the features that queries in the class may contain; $*$, $\Downarrow$ and $\downarrow$, indicate the allowed presence of $*$, $\Downarrow$ and $\downarrow$, respectively, in the query; $[\ ]$ allows the use of the query form $q[q]$ and $@$ allows the use of the query form $q[@_a\ \theta\ c]$. We abstractly denote a class of queries as $\mathcal{C}$, when the particular set of features is not of importance.

Problems of Interest. Given a set of nodes, from an XML document, chosen by the user, we wish to determine which additional nodes are certainly of interest, and which are possibly of interest, to the user. These two sets are formally defined next.

We fix a class of queries $\mathcal{C}$, a document d and subsets $\mathcal{X}^+$ and $\mathcal{X}^-$ of nodes in d, called *positive* and *negative examples*, respectively. We use $\mathcal{Q}_\mathcal{C}(\mathcal{X}^+, \mathcal{X}^-)$ to denote the set of queries

$$\{q \in \mathcal{C} \mid \mathcal{X}^+ \subseteq q(d) \wedge \mathcal{X}^- \cap q(d) = \emptyset\}\,.$$

Thus, $\mathcal{Q}_\mathcal{C}(\mathcal{X}^+, \mathcal{X}^-)$ contains the queries in $\mathcal{C}$ that return all positive examples, and no negative examples. These queries are said to be *consistent* with $\mathcal{X}^+$ and $\mathcal{X}^-$.

DEFINITION 3.5 (CERTAIN AND POSSIBLE ANSWERS). *The set of* certain *and* possible answers *with respect to $\mathcal{C}$, $\mathcal{X}^+$ and $\mathcal{X}^-$, are defined as follows*

$$Cert_\mathcal{C}(\mathcal{X}^+, \mathcal{X}^-) := \bigcap_{q \in \mathcal{Q}_\mathcal{C}(\mathcal{X}^+, \mathcal{X}^-)} q(d)$$
$$Poss_\mathcal{C}(\mathcal{X}^+, \mathcal{X}^-) := \bigcup_{q \in \mathcal{Q}_\mathcal{C}(\mathcal{X}^+, \mathcal{X}^-)} q(d)$$

Intuitively, y is a certain answer if *every* query that is consistent with $\mathcal{X}^+$ and $\mathcal{X}^-$, also returns y. Similarly, y is a possible answer if *there exists* a query that is consistent with $\mathcal{X}^+$ and $\mathcal{X}^-$, and that also returns y.

REMARK 2. *The sets of certain and possible answers are dependent on the class of queries $\mathcal{C}$, e.g., it is possible for y to be a certain answer with respect to a class $\mathcal{C}_1$, while not being a certain answer with respect to $\mathcal{C}_2$. The subclasses of XPath considered in this paper were chosen for two reasons. First, they capture the most common XPath features. Second, they are restricted enough to allow interesting generalizations, which might not be possible if the language was richer. To see why, consider allowing disjunctions of XPath expressions. Then, intuitively, $\mathcal{Q}_\mathcal{C}(\mathcal{X}^+, \mathcal{X}^-)$ would contain the query that is the disjunction of all precise expressions describing the nodes in $\mathcal{X}^+$, and thus, $Cert_\mathcal{C}(\mathcal{X}^+, \mathcal{X}^-)$ would not generalize $\mathcal{X}^+$ in an interesting manner.*

REMARK 3. *All the definitions and results in this paper assume that there is a single document d, containing positive and negative examples. The restriction to a single document is only to simplify the presentation. However, all results immediately apply even if the examples are taken from multiple documents.*

REMARK 4. ***As $\mathcal{X}^+$ and $\mathcal{X}^-$ are hand-picked by the user, we will always assume that they are of bounded size,*** *i.e., $\mathcal{X}^+$ and $\mathcal{X}^-$ have a constant number of nodes, even though d is unbounded. Since our goal is to enable ad-hoc imprecise querying, we do not see it likely for the user to choose an exorbitant number of examples.*

We demonstrate these notions with the following example.

EXAMPLE 3.6. *Consider document d from Figure 2. Suppose that $\mathcal{X}^+$ contains the circled nodes, i.e., $\mathcal{X}^+ = \{x_4, x_8\}$. To make the discussion more intuitive, we consider the class of XPath queries $\mathcal{C}(\downarrow, \Downarrow, *)$ (i.e., without branching and attribute conditions). Observe first that x_{15} is a certain answer. This is immediate as no query in $\mathcal{C}(\downarrow, \Downarrow, *)$ can distinguish between x_4 and x_{15}.*

Next, we show that x_5 is also certain answer. To see this, observe that every query returning x_4, x_8 must be either one of the following three queries, or must contain one of these queries: (1) $\Downarrow A \,/\, \Downarrow *$, (2) $\Downarrow B \,/\, \Downarrow *$, (3) $\downarrow * \,/\, \downarrow * \,/ \downarrow *$. All three of these queries also return x_5.

Node x_{12} is a possible answer, as, e.g., the query $\Downarrow A \,/\, \Downarrow *$ returns $\mathcal{X}^+\{x_4, x_8\}$ and x_{12}. However, x_{12} is not a certain answer, as $\downarrow * \,/ \downarrow * \,/ \downarrow *$ returns $\mathcal{X}^+$, but not x_{12}.

Suppose now that the user has marked the boxed node x_{13} as a negative example, i.e., $\mathcal{X}^- = \{x_{13}\}$. It is easy to see that x_{12} is now a certain answer; every query returning $\{x_4, x_8\}$, and not returning x_{13}, also returns x_{12}.

Thus, small efforts on the part of the user, to mark nodes as positive or negative examples, can immediately allow many deductions on the part of the system as to which additional nodes can interest the user.

One way to determine if a node y is a certain answer, is to generate $\mathcal{Q}_\mathcal{C}(\mathcal{X}^+, \mathcal{X}^-)$, and then check if y is also returned by all these queries. Obviously, this is highly inefficient as there can be an exponential number of queries that return $\mathcal{X}^+$. (For example, if q returns at least $\mathcal{X}^+$, then replacing any label in q with a wildcard will also yield a query returning at least $\mathcal{X}^+$.) If constraints on attributes are allowed, then there can be an infinite number of queries that return $\mathcal{X}^+$.

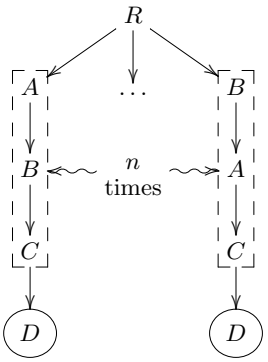

Figure 3: Document d' with an exponential number of minimal queries returning the positive examples.

An alternative approach is to find all *minimal queries* returning $\mathcal{X}^+$ and not $\mathcal{X}^-$, and only check whether y is returned by these. This is the approach we took in our explanations in Example 3.6. To formalize this notion, we say that q is minimal with respect to $\mathcal{X}^+$ if $\mathcal{X}^+ \subseteq q(d)$ and there is no q' such that $\mathcal{X}^+ \subseteq q'(d)$ and $q' \subset q$ (i.e., q' is contained in q for all documents). Unfortunately, as the following example demonstrates, even if there are only two positive examples and no negative examples at all, there can still be an exponential number of minimal queries returning $\mathcal{X}^+$, and hence this strategy is also not feasible.

EXAMPLE 3.7. *Consider the document d' in Figure 3. The notation in the figure indicates that there are n repetitions of the paths in the boxes, i.e., each of the two paths to D are of length $3n+2$. It is easy to see that for $\mathcal{X}^+$ containing precisely the two circled nodes, there are an exponential number of minimal queries. In particular, every XPath query $q_1/\Downarrow C/\cdots/q_n/\Downarrow C/\Downarrow D$ where $q_i \in \{\Downarrow A, \Downarrow B\}$ for $i \leq n$, is minimal.*

Example 3.7 demonstrates the need for a novel approach to determining certain and possible answers. Our first step in solving these problems is to observe that both the problem of determining possible answers and the problem of determining certain answers, can be reduced to a single general existence problem, defined next.

PROBLEM 1 (QUERY EXISTENCE). *Let $\mathcal{X}^+$ and $\mathcal{X}^-$ be sets of nodes in a document d, and let $\mathcal{C}$ be a class of queries. The* query existence *problem is to determine whether the set $\mathcal{Q}_\mathcal{C}(\mathcal{X}^+, \mathcal{X}^-)$ is not empty.*

We now show the relationship between this problem and those of finding possible and certain answers.

PROPOSITION 3.8. *Let d be a document, $\mathcal{X}^+$ and $\mathcal{X}^-$ be sets of nodes, y be a node and $\mathcal{C}$ be a class of queries. Then,*

1. $y \in Cert_\mathcal{C}(\mathcal{X}^+, \mathcal{X}^-) \iff \mathcal{Q}_\mathcal{C}(\mathcal{X}^+, \mathcal{X}^- \cup \{y\}) = \emptyset$;
2. $y \in Poss_\mathcal{C}(\mathcal{X}^+, \mathcal{X}^-) \iff \mathcal{Q}_\mathcal{C}(\mathcal{X}^+ \cup \{y\}, \mathcal{X}^-) \neq \emptyset$.

We conclude that an efficient solution to the query existence problem provides us with a efficient algorithm for determining both certain and possible answers. Such a solution is the focus of this paper.

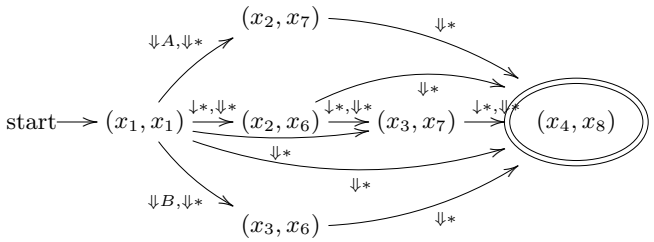

Figure 4: Query automaton $\mathcal{A}(\mathcal{C}_{path}, \{x_4, x_8\})$.

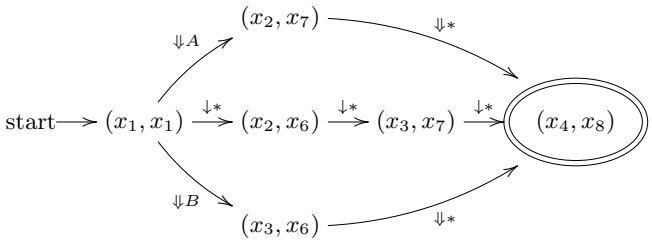

Figure 5: Normalization $\mathcal{A}^{\mathrm{n}}$ of $\mathcal{A}$ from Figure 4.

4. QUERY AUTOMATA FOR $\mathcal{X}^+$

Solving the query existence problem, i.e., determining if $\mathcal{Q}_{\mathcal{C}}(\mathcal{X}^+, \mathcal{X}^-) \neq \emptyset$, requires taking both $\mathcal{X}^+$ and $\mathcal{X}^-$ into consideration. In our algorithm, $\mathcal{X}^+$ is used to form a *query automaton*, and special types of *verification mappings*, from the query automaton to $\mathcal{X}^-$, are used to characterize query existence. This section focuses on the construction of the query automaton, while verification mappings are introduced in Section 5.

In this section, we fix a document $d = (\mathcal{V}, \mathcal{E}, r, lab, @)$ and a bounded set $\mathcal{X}^+$. In order to solve the query existence problem, we introduce the notion of a query automaton. This structure will be used to concisely represent all queries in $\mathcal{C}$ that return $\mathcal{X}^+$, i.e., the set of queries $\mathcal{Q}_{\mathcal{C}}(\mathcal{X}^+, \emptyset)$. The construction of this automaton depends on the specific class $\mathcal{C}$. We start by considering the class $\mathcal{C}(*, \Downarrow, \downarrow)$, denoted $\mathcal{C}_{path}$. Later, (in Section 7) we will consider larger (and smaller) classes of queries and show how to adapt our automata and algorithm for these classes.

REMARK 5. *When considering the class $\mathcal{C}_{path}$, the problem at hand bears some similarity to that of inferring common patterns from strings, e.g. [3, 13]. However, previous work has focused on finding a single minimal common pattern (with respect to different notions of minimality), and not on determining whether another string is satisfied by all (or some) common pattern. In addition, our assumption of a bound on $\mathcal{X}^+$ and $\mathcal{X}^-$ provides us with new cases that are efficiently inferable, requiring the development of new techniques.*

For each $x \in \mathcal{X}^+$, we use $\mathcal{V}_x$ to denote the set of nodes on the path from root of d to x. Intuitively, $\mathcal{V}_x$ is the set of *nodes of interest* for x in d. It contains all nodes that a query in $\mathcal{C}_{path}$ can "examine" while returning y.[3]

We show how to construct a query automaton for a set of nodes $\mathcal{X}^+$ and the class $\mathcal{C}_{path}$. Let x be a node in $\mathcal{X}^+$. In the following, we use $\mathcal{N}_{e,x}$ to denote the set containing the wildcard, as well as all labels associated with a node in $\mathcal{V}_x$.

CONSTRUCTION 1. *The query automaton $\mathcal{A}(\mathcal{C}_{path}, x) = (\mathcal{S}, \Sigma, \delta, s_0, s_f)$ is defined as follows:*

1. $\mathcal{S} = \mathcal{V}_x$;
2. $\Sigma = \{\downarrow l \mid l \in \mathcal{N}_{e,x}\} \cup \{\Downarrow l \mid l \in \mathcal{N}_{e,x}\}$;
3. $\delta(z, q)$ *is the set of nodes* z' *such that* $d \models q(z, z')$;
4. $s_0 = r$, *i.e., the root of* d;
5. $s_f = x$;

The query automaton $\mathcal{A}(\mathcal{C}_{path}, \mathcal{X}^+)$ is simply the product of the automata $\mathcal{A}(\mathcal{C}_{path}, x)$, for all x in $\mathcal{X}^+$.

If s is a state in $\mathcal{A}(\mathcal{C}_{path}, \mathcal{X}^+)$, then observe that $\delta(s, \downarrow l)$ must always either be empty, or contain a singleton. This holds since a state s is a tuple of nodes $(z_1, \ldots, z_k)$ from d and each node z_i can have a single child among V_{x_i}, i.e., among the nodes on the path to x_i.

The *language* of $\mathcal{A}$, written $L(\mathcal{A})$, is the set of all queries *accepted by* $\mathcal{A}$, where q is accepted by $\mathcal{A}$ if there are queries $q_1, \ldots, q_n$ and states $s_1, \ldots, s_n$, with $s_n = s_f$, such that

- $q = q_1 / \cdots / q_n$;
- for all $1 \leq i \leq n$, it holds that $s_i \in \delta(s_{i-1}, q_i)$.

EXAMPLE 4.1. *Recall the document d from Figure 2. Let $\mathcal{X}^+ = \{x_4, x_8\}$. The query automata $\mathcal{A}$ for $\mathcal{X}^+$ and $\mathcal{C}_{path}$ appears in Figure 4. We have only drawn states s that are* between s_0 *and s_f, i.e., s is reachable from the starting state s_0, and s_f is reachable from s.*

Observe that the starting state (x_1, x_1) contains only the root node and the final state (x_4, x_8) corresponds precisely to $\mathcal{X}^+$. There is a transition $\Downarrow A$ (as well as $\Downarrow *$) from (x_1, x_1) to (x_2, x_7) since x_2 and x_7 are both descendants of x_1, and they have the same label A. Similarly, there is a transition $\downarrow *$ from (x_3, x_7) to (x_4, x_8) as (1) x_4 is a child of x_3 and (2) x_8 is a child of x_7.

It is not difficult to see that $\mathcal{A}$ accepts precisely the language of queries returning $\{x_4, x_8\}$ over d. This language includes the three queries (1) $\Downarrow A / \Downarrow *$, (2) $\Downarrow B / \Downarrow *$, and (3) $\downarrow * / \downarrow * / \downarrow *$ discussed in Example 3.6, as well as additional queries, such as $\Downarrow *$, and $\Downarrow * / \downarrow *$.

Our automata satisfy several important properties.

THEOREM 4.2. *Let $\mathcal{X}^+$ be a set of nodes and $\mathcal{A}$ be the automaton $\mathcal{A}(\mathcal{C}_{path}, \mathcal{X}^+)$. Then,*

1. *$\mathcal{A}$ is acyclic;*
2. *$\mathcal{A}$ is polynomial in the size of d if $\mathcal{X}^+$ is of constant size;*
3. *the language of $\mathcal{A}$ is precisely the set of queries in $\mathcal{C}_{path}$ returning $\mathcal{X}^+$, i.e., $L(\mathcal{A}) = \mathcal{Q}_{\mathcal{C}_{path}}(\mathcal{X}^+, \emptyset)$.*

According to Theorem 4.2, to solve the query existence problem, it is sufficient to find a query q in $L(\mathcal{A}(\mathcal{C}_{path}, \mathcal{X}^+))$ for which $\mathcal{X}^- \cap q(d) = \emptyset$. For purposes of efficiency, we prefer

[3]The notion of "examining" a node when returning y should be self-explanatory. However, to be more precise, in Definition 3.3, deciding that $d \models q(r, y)$ may involve subproblems of the format $d \models q_1(z, w)$. Nodes z, w are thus, examined, when determining that y is returned.

to consider a *normalized version* of our automata. Normalized automata will have at most one transition between any pair of states, i.e., for any two states s_1 and s_2, and for any $\alpha, \alpha' \in \Sigma$, if $s_2 \in \delta(s_1, \alpha)$ and $s_2 \in \delta(s_1, \alpha')$, then $\alpha = \alpha'$. Note that this property will not naturally occur in our automata, since, e.g., if $s_2 \in \delta(s_1, \downarrow a)$, then $s_2 \in \delta(s_1, \downarrow *)$, $s_2 \in \delta(s_1, \Downarrow a)$, $s_2 \in \delta(s_1, \Downarrow *)$, by our construction.

Let $\mathcal{A}$ be an automaton, defined as above. The *normalization* of $\mathcal{A}$, denoted $\mathcal{A}^{\mathrm{n}}$ is derived from $\mathcal{A}$ by the following two steps:

- **Removing Unneeded Wildcards.** For each pair of states s_1, s_2 in $\mathcal{A}$, if there is some $l \neq *$ such that $s_2 \in \delta(s_1, \downarrow l)$, then remove s_2 from $\delta(s_1, \downarrow *)$. Similarly, if there is some $l \neq *$ such that $s_2 \in \delta(s_1, \Downarrow l)$, then remove s_2 from $\delta(s_1, \Downarrow *)$.
- **Removing Unneeded Descendants.** For each pair of states s_1, s_2 in $\mathcal{A}$, if there is some l such that $s_2 \in \delta(s_1, \downarrow l)$, then remove s_2 from $\delta(s_1, \Downarrow l)$.

  Similarly, for each pair of states s_1, s_2 in $\mathcal{A}$, if there is a path of transitions of length greater than one from s_1 to s_2, then remove s_2 from $\delta(s_1, \Downarrow l)$.

It is not difficult to see that this process is unambiguous, and is independent of the order of application (due to the automaton being acyclic). For example, removal of unneeded descendants is order independent, since a descendant transition from s_1 to s_2 is only removed if there is some other path of transitions from s_1 to s_2. Hence, this descendant transition is not needed when determining which other descendant transitions to remove.

EXAMPLE 4.3. *The normalized version $\mathcal{A}^{\mathrm{n}}$ of the automata $\mathcal{A}$ from Figure 4, appears in Figure 5. Observe that $\mathcal{A}$ and $\mathcal{A}^{\mathrm{n}}$ have the same set of states, the same starting state and the same final state. However, in $\mathcal{A}^{\mathrm{n}}$ we have removed transitions that were somewhat superfluous, as they could be derived from other transitions in the automaton.*

For example, there is a single transition $\Downarrow A$ from (x_1, x_1) to (x_2, x_7) in $\mathcal{A}^{\mathrm{n}}$, whereas there were two transitions ($\downarrow A$ and $\Downarrow *$) in $\mathcal{A}$. Similarly only the transition $\downarrow *$ from (x_3, x_7) to (x_4, x_8) was retained in $\mathcal{A}^{\mathrm{n}}$. Finally, observe that there are pairs of states for which a transition exists in $\mathcal{A}$, but there is none at all in $\mathcal{A}^{\mathrm{n}}$. For example, there is no transition from (x_1, x_1) to (x_4, x_8) in $\mathcal{A}^{\mathrm{n}}$; the $\Downarrow *$ transition appearing in $\mathcal{A}$ was removed as there are paths of length greater than one from (x_1, x_1) to (x_4, x_8) in $\mathcal{A}$.

Observe that the language of $\mathcal{A}^{\mathrm{n}}$ contains precisely three queries (those discussed already in Example 3.6): (1) $\Downarrow A\,/\Downarrow *$, (2) $\Downarrow B\,/\Downarrow *$, and (3) $\downarrow *\,/\downarrow *\,/\downarrow *$.

It is easy to see that the normalization $\mathcal{A}^{\mathrm{n}}$ of $\mathcal{A}$ has the following properties.

PROPOSITION 4.4. *Let $\mathcal{A}^{\mathrm{n}}$ be the normalization of $\mathcal{A}$. Then, for all states s,*

- *there is at most one state s_1 and one query $\downarrow l_1$ such that $s \in \delta(s_1, \downarrow l_1)$;*
- *if there are states s_1 and s_2 and queries $\Downarrow l_1$ and $\Downarrow l_2$ such that $s \in \delta(s_1, \Downarrow l_1)$ and $s \in \delta(s_2, \Downarrow l_2)$, then $l_1 = l_2$.*

Normalization removes queries from the language, but preserves all *minimal queries*. This is a crucial characteristic that allows us to show the following result. (In Theorem 4.5, we use $q \subseteq q'$ to denote the fact that $q(d) \subseteq q'(d)$, for all documents d.)

THEOREM 4.5. *Let $\mathcal{A} = \mathcal{A}(\mathcal{C}_{path}, \mathcal{X}^+)$ be a query automata and $\mathcal{A}^{\mathrm{n}}$ be its normalization. Then,*

1. *$L(\mathcal{A}^{\mathrm{n}}) \subseteq L(\mathcal{A})$.*
2. *For all $q' \in L(\mathcal{A})$, there exists a $q \in L(\mathcal{A}^{\mathrm{n}})$ such that $q \subseteq q'$.*
3. *$\mathcal{Q}_{\mathcal{C}_{path}}(\mathcal{X}^+, \mathcal{X}^-) \neq \emptyset$ if and only if there exists a query $q \in L(\mathcal{A}^{\mathrm{n}})$ such that $\mathcal{X}^- \cap q(d) = \emptyset$.*

5. VERIFICATION MAPPINGS FOR $\mathcal{X}^-$

The previous section defined query automata that capture the $\mathcal{X}^+$ part of the query existence problem. This section focuses on $\mathcal{X}^-$, in order to provide a characterization, based on $\mathcal{X}^-$ and the query automaton, that will allow us to answer the query existence problem. To this end, we introduce *single-node and multi-node verification mappings*.

We fix $\mathcal{A}^{\mathrm{n}}$ to be the normalization of $\mathcal{A}(\mathcal{C}_{path}, \mathcal{X}^+)$. A sequence of states $\bar{s} = s_0, \ldots, s_n$ in $\mathcal{A}^{\mathrm{n}}$ is an *accepting sequence* if the following conditions hold:

- $s_n = s_f$, i.e., is the accepting state *and*
- for all $1 \leq i \leq n$, there exists q_i in the alphabet of $\mathcal{A}^{\mathrm{n}}$ such that $s_i \in \delta(s_{i-1}, q_i)$.

Since $\mathcal{A}^{\mathrm{n}}$ is normalized, there will be precisely one q_i for which $s_i \in \delta(s_{i-1}, q_i)$. Hence, we will refer to q_i unambiguously in Definition 5.1 below. Note that there is a natural correspondence between accepting sequences and words in $L(\mathcal{A}^{\mathrm{n}})$.

Let $x \in \mathcal{X}^-$ be a node. Let $\mathcal{V}_x$ be, as before, the set of nodes on the path from the root of d to x. Define $\mathcal{V}_x^\epsilon$ as $\mathcal{V}_x \cup \{\epsilon\}$. We extend $\models$ (from Definition 3.3) to deal with ϵ by defining:

- for all d, q and y, $d \models q(y, \epsilon)$;
- for all d, q and y, if $y \neq \epsilon$, then $d \not\models q(\epsilon, y)$.

The purpose of this "dummy node" ϵ will soon be apparent. Intuitively, it is used to mark paths that do not (or cannot) continue in the database.

DEFINITION 5.1 (SNQV-MAPPING). *Let $s_0, \ldots, s_n$ be an accepting sequence in $\mathcal{A}^{\mathrm{n}}$. A function $\mu : \{s_0, \ldots, s_n\} \to \mathcal{V}_x^\epsilon$ is a* single-node query-verification mapping *(or* snqv-mapping *for short) for $s_0, \ldots, s_n$ and x if all the following conditions hold:*

1. *$\mu(s_0) = r$ (where r is the root of d);*
2. *$\mu(s_n) = x$ or $\mu(s_n) = \epsilon$;*
3. *for all i, it holds that $d \models q_i(\mu(s_{i-1}), \mu(s_i))$.*

Intuitively, an snqv-mapping μ for $s_0, \ldots, s_n$ and x indicates whether x can be returned by query $q = q_1/\cdots/q_n$ formed by states $s_0, \ldots, s_n$. If μ maps all states to nodes other than ϵ, then it constitutes a proof that x is returned by q over d. Conversely, if $x \in q(d)$, then there is an snqv-mapping μ for which $\mu(s_f) = x$. Hence, $x \notin q(d)$, if and only if all snqv-mapping μ for $s_0, \ldots, s_n$ map s_n $(= s_f)$ to ϵ.

EXAMPLE 5.2. *Consider node x_{13} from document d (Figure 2). Let $\bar{s}_1, \bar{s}_2, \bar{s}_3$ be the accepting sequences from $\mathcal{A}^n$ (Figure 5):*

$$\bar{s}_1 = (x_1, x_1), (x_2, x_7), (x_4, x_8)$$
$$\bar{s}_2 = (x_1, x_1), (x_2, x_6), (x_3, x_7), (x_4, x_8)$$
$$\bar{s}_3 = (x_1, x_1), (x_3, x_6), (x_4, x_8)\,,$$

and $\mu_1, \mu_1', \mu_2, \mu_3$ be the mappings:

$$\begin{array}{lll} \mu_1(x_1,x_1) = x_1 & \mu_1(x_2,x_7) = x_9 & \mu_1(x_4,x_8) = \epsilon \\ \mu_1'(x_1,x_1) = x_1 & \mu_1'(x_2,x_7) = x_9 & \mu_1'(x_4,x_8) = x_{13} \\ \mu_2(x_1,x_1) = x_1 & \mu_2(x_2,x_6) = x_9 & \mu_2(x_3,x_7) = x_{10} \\ & & \mu_2(x_4,x_8) = x_{13} \\ \mu_3(x_1,x_1) = x_1 & \mu_3(x_3,x_6) = \epsilon & \mu_3(x_4,x_8) = \epsilon \end{array}$$

Mappings μ_1 and μ_1' are both snqv-mappings for $\bar{s}_1$ and x_{13}. Note that μ_1' is a witness to the fact that the query defined by $\bar{s}_1$ returns node x_{13}. Similarly, μ_2 and μ_3 are snqv-mappings for $\bar{s}_2$ and $\bar{s}_3$, respectively, and x_{13}. Every snqv-mapping for $\bar{s}_3$ and x_{13} maps (x_4, x_8) to ϵ. This holds as the query defined by $\bar{s}_3$, i.e., $\Downarrow B/\Downarrow *$, does not return x_{13}.

We extend the notion of an snqv-mapping to multiple nodes. Let $\mathcal{X}^- = \{x_1, \ldots, x_k\}$. We define $\mathcal{V}^{\epsilon}_{\mathcal{X}^-} = \mathcal{V}^{\epsilon}_{x_1} \times \cdots \times \mathcal{V}^{\epsilon}_{x_k}$.

DEFINITION 5.3 (MNQV-MAPPING). *Let $s_0, \ldots, s_n$ be an accepting sequence in $\mathcal{A}^n$. A function $\mu : \{s_0, \ldots, s_n\} \to \mathcal{V}^{\epsilon}_{\mathcal{X}^-}$ is a* multi-node query-verification mapping *(or* mnqv-mapping *for short) for $s_0, \ldots, s_n$ and $x_1, \ldots, x_k$ if, for all $j \leq k$, the projection μ_j of μ on its j-th components[4] is an snqv-mapping for x_j.*

Observe that if *every* mnqv-mapping μ for $s_0, \ldots, s_n$ and $x_1, \ldots, x_k$ maps s_n $(= s_f)$ to the tuple $\bar{\epsilon} = (\epsilon, \ldots, \epsilon)$, then there exists a query $q \in L(\mathcal{A}^n)$ for which $\mathcal{X}^- \cap q(d) = \emptyset$. Thus, such query is a proof for non-emptiness of the set $\mathcal{Q}_{\mathcal{C}_{path}}(\mathcal{X}^+, \mathcal{X}^-)$. We demonstrate this observation next.

EXAMPLE 5.4. *Consider $\mathcal{X}^- = \{x_{10}, x_{13}\}$. Then,*

$$\begin{array}{ll} \mu(x_1,x_1) = (x_1,x_1) & \mu(x_2,x_6) = (x_9,x_9) \\ \mu(x_3,x_7) = (x_{10},x_{10}) & \mu(x_4,x_8) = (\epsilon,x_{13}) \end{array}$$

is an mnqv-mapping for $\bar{s}_2 = (x_1, x_1), (x_2, x_6), (x_3, x_7), (x_4, x_8)$ and $\mathcal{X}^-$, while

$$\begin{array}{ll} \mu'(x_1,x_1) = (x_1,x_1) & \mu'(x_3,x_6) = (\epsilon,\epsilon) \\ & \mu(x_4,x_8) = (\epsilon,\epsilon)\,. \end{array}$$

is an mnqv-mapping for $\bar{s}_3 = (x_1, x_1), (x_3, x_6), (x_4, x_8)$.

Note that μ indicates that the query defined by $\bar{s}_2$ is not a witness for query existence, as this query will return x_{13} (one of the nodes in $\mathcal{X}^-$). However, as all mnqv-mappings for $\bar{s}_3$ will map (x_4, x_8) to $\bar{\epsilon}$, the query defined by $\bar{s}_3$ is a witness for the query existence problem.

6. DETERMINING QUERY EXISTENCE

In this section we show how to use the notions of mnqv-mappings, in order to determine query existence. Let s be a state in $\mathcal{A}^n$ and let $\bar{s}$ be an accepting sequence containing s. We define

$$Img_{\bar{s}}(s) := \{\bar{y} \mid \exists \mu \text{ for } \bar{s} \text{ and } x_1, \ldots, x_k \text{ s.t. } \mu(s) = \bar{y}\}\,.$$

Thus, $Img_{\bar{s}}(s)$ is the set containing the images of s in all mnqv-mappings for $\bar{s}$ and $x_1, \ldots, x_k$. Obviously, $Img_{\bar{s}}(s) \subseteq \mathcal{V}^{\epsilon}_{\mathcal{X}^-}$, but $Img_{\bar{s}}(s)$ can also be much smaller.

Now, from our discussion in the previous section, it is easy to see the following result, which follows, essentially, by the definitions.

PROPOSITION 6.1. *The set $\mathcal{Q}_{\mathcal{C}_{path}}(\mathcal{X}^+, \mathcal{X}^-)$ is not empty if and only if there exists an accepting sequence $\bar{s}$ such that $Img_{\bar{s}}(s_f) = \{\bar{\epsilon}\}$.*

Note that the accepting sequence $\bar{s}$ provides us with a query (defined by this sequence of transitions) that returns all of $\mathcal{X}^+$ and none of $\mathcal{X}^-$.

Proposition 6.1 provides us with a procedure for determining query existence, namely, consider all accepting sequences $\bar{s}$, compute $Img_{\bar{s}}(s_f)$ and check whether this set is precisely $\{\bar{\epsilon}\}$. Obviously, however, such a procedure is highly inefficient, e.g., since it must consider a possibly exponential number of accepting sequences.

To avoid explicit enumeration of all accepting sequences, define $AllImg(s)$ as

$$AllImg(s) := \{Img_{\bar{s}}(s) \mid \bar{s} \text{ is an accepting sequence}\}\,.$$

Note that $AllImg(s)$ is a set of sets. Then, the following is a corollary of Proposition 6.1.

COROLLARY 6.2. *The set $\mathcal{Q}_{\mathcal{C}_{path}}(\mathcal{X}^+, \mathcal{X}^-)$ is not empty if and only if $\{\bar{\epsilon}\} \in AllImg(s_f)$.*

Corollary 6.2 again provides us with an approach to solving the query existence problem, namely, first compute the set $AllImg(s_f)$, and then check if the set $\{\bar{\epsilon}\}$ is contained in $AllImg(s_f)$. In order for this approach to be efficient, we will show that for all s,

1. $AllImg(s)$ is of polynomial size *and*
2. $AllImg(s)$ can be computed in polynomial time.

Note that neither of these claims is trivial, as a straightforward computation requires us to consider each of the possibly exponential number of accepting sequences, each of which might, conceivably give rise to a distinct set. Notwithstanding this difficulty, we will present a procedure for computing $AllImg(s)$, and will show that this procedure runs in polynomial time (thus, proving both of the above claims).

6.1 Computing $AllImg(s)$

We will show how compute $AllImg(s)$ inductively, by considering the states s in $\mathcal{A}^n$ in a topological ordering. Since $\mathcal{A}^n$ is acyclic (Theorem 4.2), this ordering is well-defined.

Initial State. For the initial state s_0, it is easy to see that

$$AllImg(s_0) = \{\{\bar{r}\}\}\,,$$

where $\bar{r}$ is the tuple containing only the document root node. This follows immediately from the definition, since $Img_{\bar{s}}(s_0) = \{\bar{r}\}$, for all accepting sequences $\bar{s}$.

[4]Formally, this projection μ_j is defined in the following fashion: $\mu_j(s_i) = y_j$ if $\mu(s_i) = (y_1, \ldots, y_k)$.

Non-Initial State. We now consider a non-initial state s. Before describing our computation of $AllImg(s)$, we show how $Img_{\bar{s}}(s)$ can be expressed as a function of $Img_{\bar{s}}(t)$, where t is the state preceding s. For this purpose, we define the function $Next$ below.

Formally, let $\bar{s}$ be an accepting sequence containing s and let t be the state immediately preceding s in $\bar{s}$. Recall that since $\mathcal{A}^{\mathrm{n}}$ is normalized, there will be a single transition q_t from t to s.

We show how to define $Img_{\bar{s}}(s)$, in terms of $Img_{\bar{s}}(t)$. Let $\bar{y} = (y_1, \ldots, y_k)$ be a tuple in $Img_{\bar{s}}(t)$. For each $l \leq k$, define

$$Z_l := \{z \in \mathcal{V}^{\epsilon}_{x_l} \mid d \models q_t(y_l, z)\}\,.$$

Moreover, if s is the accepting state s_f, then we remove from Z_l all nodes that are different from x_l and from ϵ. Now, define

$$Next((y_1, \ldots, y_k), q_t) = Z_1 \times \cdots \times Z_k\,. \tag{1}$$

For sets of tuples, Ψ, we similarly define

$$Next(\Psi, q_t) = \cup_{\bar{y} \in \Psi} Next(\bar{y}, q_t)\,. \tag{2}$$

We show the following result.

PROPOSITION 6.3. *Let $\bar{s}$ be an accepting sequence, s be a state in $\bar{s}$ and t be the state preceding s in $\bar{s}$. Then,*

$$Img_{\bar{s}}(s) = Next(Img_{\bar{s}}(t), q_t)\,.$$

Proposition 6.3 is useful as it relates sets $Img_{\bar{s}}(s)$ and $Img_{\bar{s}}(t)$. However, in order to avoid considering all accepting sequences, we must be able to relate the sets $AllImg(s)$ and $AllImg(t)$. This is precisely what we do next.

Again, let t be a state with a single transition q_t to s, in $\mathcal{S}^{\mathrm{n}}$. Define

$$AllNext(AllImg(t), q_t) = \{Next(\Psi, q_t) \mid \Psi \in AllImg(t)\}\,.$$

Observe that $AllNext(AllImg(t), q_t)$ is a set of sets, just like $AllImg(t)$. Now the following result easily follows from the definition of $AllImg(s)$ and from Proposition 6.3.

COROLLARY 6.4. *Let s be a non-initial state of $\mathcal{A}^{\mathrm{n}}$ and let T be the set of all nodes from which s is reachable in a single transition. Then,*

$$AllImg(s) = \bigcup_{t \in T} AllNext(AllImg(t), q_t)\,.$$

We conclude this section by observing that Corollary 6.4 provides us with an algorithm to compute $AllImg(s)$, for all states s: For the initial state s_0, recall that $AllImg(s_0) = \{\{\bar{r}\}\}$. Next, iterate over all remaining states in topological order, and use the definition of $AllNext$ to compute $AllImg(s)$ in terms of all previously computed $AllImg(t)$ (with transitions to s).

6.2 Bounding the Size of $AllImg(s)$

Section 6.1 provided us with a method to compute the set of sets $AllImg(s)$, for all states s. It is easy to observe that this computation is polynomial in the sizes of the previously computed sets $AllImg(t)$. Hence, if we can show that $AllImg(s)$ is always of polynomial size in the input (and not dependent on the iteration step), we derive an efficient method of determining query existence. Thus, bounding the size of $AllImg(s)$ is the topic of this section. We start with some necessary definitions.

First, we define a partial order over tuples in $\mathcal{V}^{\epsilon}_{\mathcal{X}^-}$. Let $\bar{y} = (y_1, \ldots, y_k)$ and $\bar{z} = (z_1, \ldots, z_k)$ be tuples in $\mathcal{V}^{\epsilon}_{\mathcal{X}^-}$. We say that $\bar{y}$ *precedes* $\bar{z}$, written $\bar{y} \preceq \bar{z}$, if for each $i \leq k$, one of the following conditions hold: (1) $y_i = z_i$; (2) y_i is an ancestor of z_i; *or* (3) $z_i = \epsilon$. Given a set of tuples Ψ, we say that $\bar{y}$ is a *minimal tuple* in Ψ, if $\bar{y} \prec \bar{z}$, for all $\bar{z} \in \Psi$.

An arbitrary set of tuples from $\mathcal{V}^{\epsilon}_{\mathcal{X}^-}$ may not have a minimal tuple (with respect to $\preceq$), as not every two tuples are comparable. However, we can show the following property of $Img_{\bar{s}}(s)$.

THEOREM 6.5. *For every s in $\mathcal{A}^{\mathrm{n}}$ and each accepting sequence $\bar{s}$, there is a minimal tuple in $Img_{\bar{s}}(s)$.*

Corollary 6.6 follows immediately.

COROLLARY 6.6. *For every s in $\mathcal{A}^{\mathrm{n}}$, each set in $AllImg(s)$ has a minimal tuple.*

Minimal tuples are useful, as they are the only tuple of interest in a set of tuples, when propagating with a query of the form $\Downarrow l$, as Proposition 6.7 shows.

PROPOSITION 6.7. *Let $\bar{y}_i$ be minimal among $\{\bar{y}_1, \cdots, \bar{y}_n\}$. Then, for any $l \in \mathcal{N}_e \cup \{*\}$,*

$$Next(\{\bar{y}_1, \cdots, \bar{y}_n\}, \Downarrow l) = Next(\bar{y}_i, \Downarrow l)\,.$$

We will use $|AllImg(s)|$ to denote the total size of s, including all members of all sets, i.e.,

$$|AllImg(s)| = \sum_{\Psi \in AllImg(s)} |\Psi|\,.$$

Note that $|AllImg(s)| \leq \sum_{\bar{s}} |Img_{\bar{s}}(s)|$, but $|AllImg(s)|$ can also be significantly smaller, since $AllImg(s)$ is a set of sets, and thus, a single copy of duplicate sets is retained.

In the following, $h(s)$ is the *height* of s, i.e., the number of nodes on the longest path from the initial state s_0 to s. For nodes in d, we define h similarly, i.e., $h(x)$ is the number of nodes on the path from r to x. Finally, we will use $h(\mathcal{X}^-)$ to denote the maximum of all heights of nodes in $\mathcal{X}^-$, i.e., $h(\mathcal{X}^-) := \max\{h(x) \mid x \in \mathcal{X}^-\}$.

THEOREM 6.8. *Let m be $h(\mathcal{X}^-)$ and k be $|\mathcal{X}^-|$. Then, for any state s in $\mathcal{A}^{\mathrm{n}}$,*

$$|AllImg(s)| \leq (m+1)^k h(s)\,.$$

We now derive the main result of this section.

THEOREM 6.9. *Determining non-emptiness of the query set $\mathcal{Q}_{\mathcal{C}_{path}}(\mathcal{X}^+, \mathcal{X}^-)$ is in polynomial time, if $\mathcal{X}^+$ and $\mathcal{X}^-$ are of constant size.*

Due to Proposition 3.8, the following is immediate.

COROLLARY 6.10. *Determining whether a node y is in $Cert_{\mathcal{C}_{path}}(\mathcal{X}^+, \mathcal{X}^-)$ and whether y is in $Poss_{\mathcal{C}_{path}}(\mathcal{X}^+, \mathcal{X}^-)$ is in polynomial time, if $\mathcal{X}^+$ and $\mathcal{X}^-$ are of constant size.*

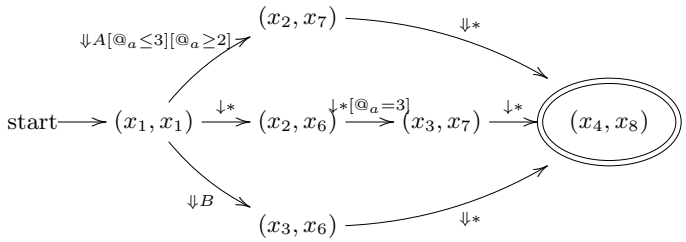

Figure 6: Normalized automaton for class $\mathcal{C}(*, \downarrow, \Downarrow, @)$.

7. OTHER CLASSES OF QUERIES

The previous sections considered the class of queries $\mathcal{C}_{path}$. Note that the class of queries does not have bearing on the algorithm, for the most part; however it is critical in defining the query automaton. This section considers other classes of queries and shows how to adapt query automata to these cases. In particular, we will show how to deal with specific features. These constructions can be combined in a straight-forward manner, when considering classes that contain several of these features.

7.1 Classes with Attributes

Attributes can not be directly encoded in the automata presented this far since, unlike labels, (1) attributes allow for inequality conditions and (2) there may be multiple attributes associated with a single node. However, extending our results to deal with attributes is still quite easy. The only change is in the language of our automaton (which will allow queries with attributes) and in their construction. In the following we detail only the main difference in construction, i.e., how the transitions are determined.

Given $\mathcal{X}^+ = \{x_1, \ldots, x_k\}$, let $\mathcal{A}^{\text{n}}$ be the normalized automaton constructed in Section 4 for the language $\mathcal{C}_{path}$. Let $s = (z_1, \ldots, z_k)$ and $s' = (z'_1, \ldots, z'_k)$ be states such that there is a transition σ from s to s'. Now, let A be the set of attributes a common to nodes $z'_1, \ldots, z'_k$, with values taken from $\mathcal{D}_<$. Define

$$a_{min} := \min\{@_a(z_i) \mid i \leq k\}$$
$$a_{max} := \max\{@_a(z_i) \mid i \leq k\}$$

Let B be the set of attributes b common to nodes $z'_1, \ldots, z'_k$, with values taken from $\mathcal{D}$, for which there is a single value b_{val} such that $@_b(z_i) = b_{val}$, for all i.

Now, we replace the transition σ with the transition

$$\sigma \cdot_{a \in A} ([@_a \geq a_{min}][@_a \leq a_{max}]) \cdot_{b \in B} ([@_b = b_{val}]),$$

where $\cdot_{a \in A}$ and $\cdot_{b \in B}$ are string concatenation, applied to all $a \in A$ and $b \in B$, respectively. Intuitively, σ is replaced with a stricter transition that also enforces all possible attribute constraints.

Example 7.1. *The normalized automaton in Figure 5 was created for the set $\mathcal{X}^+ = \{x_4, x_8\}$ and the class of queries $\mathcal{C}_{path}$. Figure 6 contains the normalized automaton for the same set $\mathcal{X}^+$, but this time for class $\mathcal{C}(*, \downarrow, \Downarrow, @)$.*

It is not difficult to show that besides these changes, the algorithm can remain the same, and still solves query existence in polynomial time. Note, in particular, that definitions and results for verification mappings are given in terms of satisfaction of queries (i.e., the operator $\models$ from Definition 3.3), and hence, are independent of the language chosen.

7.2 Classes with Branching

In essence, dealing with branching again involves adapting the transitions of the query automaton, albeit, in a much more intricate manner. In order to simplify the presentation, we will assume in this section that branching is allowed, while attributes are not. Combining both branching and attributes is rather straight-forward, when using the techniques of both this section, and those of the previous.

From a technical standpoint, branching introduces significant challenges. We focus in this section on the difficulties that arise, explaining where straightforward extensions of our previous results fail to provide a polynomial solution. Due to space limitations, this section only contains a hint to our mechanism used to overcome the new problems.

When defining our query automata in Section 4, the alphabet was of the form $\downarrow l$ and $\Downarrow l$ (where l is a label or wildcard). Similarly to the way attributes were dealt with, one can enrich the alphabet of query automata to include $\downarrow l[\bar{q}]$ and $\Downarrow l[\bar{q}]$ where $[\bar{q}]$ denotes a series of branching conditions $[q_1] \cdots [q_n]$ (each of which may, in turn, also include branching conditions). In such a manner, one can define an automaton that returns precisely the language of all queries (which may include branching) that return $\mathcal{X}^+$.

Example 7.2. *Consider the automaton in Figure 4. In order to capture all branching conditions, the transition from (x_1, x_1) to (x_2, x_6) could be augmented with more queries, such as*

$$\downarrow * [\downarrow *] \qquad \downarrow * [\downarrow * / \downarrow D] \qquad \Downarrow * [\Downarrow D],$$

among many, many others. Note that for all three of the queries q above, it holds that $d \models q(x_1, x_2)$ and $d \models q(x_1, x_6)$.

Let $\mathcal{C}_{branch}$ be the class of queries $\mathcal{C}(*, \Downarrow, \downarrow, [])$. Let d be a document and $\mathcal{X}^+$ be a set of nodes. Then, the query automaton $\mathcal{A}(\mathcal{C}_{branch}, \mathcal{X}^+)$ is defined similarly to Construction 1, with the following change to the alphabet. The alphabet Σ of $\mathcal{A}(\mathcal{C}_{branch}, \mathcal{X}^+)$ contains all expressions of the form $\downarrow l[q_1] \cdots [q_m]$ and $\Downarrow l[q_1] \cdots [q_m]$, where q_i are queries in $\mathcal{C}_{branch}$. In other words, Σ contains all single step (i.e., single axes) XPath queries with arbitrary branching.[5] Note that the transition function remains the same (it is defined in terms of query satisfaction, for all queries in the alphabet).

With essentially the same proof as that of Theorem 4.2, it is easy to show that $\mathcal{A}(\mathcal{C}_{branch}, \mathcal{X}^+)$ is acyclic and that the language of $\mathcal{A}$ is precisely those queries in $\mathcal{C}_{branch}$ that return $\mathcal{X}^+$. However, Claim 2 of Theorem 4.2 no longer holds, as there may now be an exponential number of transitions.

Fortunately, the size of $\mathcal{A}(\mathcal{C}_{branch}, \mathcal{X}^+)$ is not of particular concern, as our algorithm for checking query existence uses a normalized version of an automaton. Therefore, our next step is to extend the normalization process to automata of the form $\mathcal{A}(\mathcal{C}_{branch}, \mathcal{X}^+)$. In a similar spirit to the normalization defined in Section 4, we remove unneeded wildcard transitions, (by removing transitions involving $*$, when there

[5]To ensure a finite number of transitions in the automaton, recall that we do not allow expressions in which the same branching condition appears multiple times, e.g., $\downarrow D[\Downarrow A][\Downarrow A]$.

is a more specific one with a label), and we remove unneeded descendant transitions (by removing transitions involving $\Downarrow$, when these are implied by other transitions in the automaton). In addition to these parts of the normalization process described earlier, we also perform a third step:

- **Removing Unneeded Branching.** Remove transitions of the form $\downarrow l[q_1]\cdots[q_n]$ (resp. $\Downarrow l[q_1]\cdots[q_n]$) between states that have a transition $\downarrow l[q'_1]\cdots[q'_m]$ (resp. $\Downarrow l[q'_1]\cdots[q'_m]$) such that all of q_i are among q'_j.

It is quite easy to show that the result of normalization, denoted $\mathcal{A}^{\mathrm{n}}(\mathcal{C}_{branch}, \mathcal{X}^+)$, satisfies properties in the spirit of those of Proposition 4.4. In particular, each state s will have at most one incoming transition with a child axis (i.e., of the form $\downarrow l[q_1]\cdots[q_m]$), and all incoming transitions will have precisely the same branching conditions. This hinges on the crucial observation that, in $\mathcal{A}(\mathcal{C}_{branch}, \mathcal{X}^+)$, if s has incoming transitions $\Downarrow l[q_1]$ and $\Downarrow l[q_2]$, then s will also have the incoming transition $\Downarrow l[q_1][q_2]$ (and hence the former two will be removed during normalization).

In addition, as in Theorem 4.5, with essentially the same proof one can show that $\mathcal{Q}_{\mathcal{C}_{branch}}(\mathcal{X}^+, \mathcal{X}^-)$ is not empty if and only if there exists a query $q \in L(\mathcal{A}^{\mathrm{n}}(\mathcal{C}_{branch}, \mathcal{X}^+))$ such that $\mathcal{X}^- \cap q(d) = \emptyset$.

Given the above results, the algorithm used for checking for query existence in Section 6 can be used for the class $\mathcal{C}_{branch}$. However, this does not immediately show that determining non-emptiness of $\mathcal{Q}_{\mathcal{C}_{branch}}(\mathcal{X}^+, \mathcal{X}^-)$ is in polynomial time. There are three difficulties with this process:

1. As described so far, $\mathcal{A}^{\mathrm{n}}$ is derived from $\mathcal{A}$. However, since $\mathcal{A}$ may have exponentially many transitions, it is not possible to generate $\mathcal{A}$, and thereafter derive $\mathcal{A}^{\mathrm{n}}$, in polynomial time.

2. The transitions in $\mathcal{A}^{\mathrm{n}}$ may have exponentially large branching conditions. To understand why, recall that a transition containing the branching condition $[q]$ into a state $(z_1, \ldots, z_k)$ implies that the subtree rooted at z_i (in the document) satisfies q, for all i. Thus, the most restrictive transition into $(z_1, \ldots, z_n)$ will have a branching condition that expresses all the common aspects of the subtrees rooted at $z_1, \ldots, z_n$. Obviously, this branching condition can be exponential in size (as there can, potentially, be exponentially many commonalities to the subtrees rooted at $z_1, \ldots, z_n$).

   Thus, we may conclude that even if it were possible to directly generate $\mathcal{A}^{\mathrm{n}}$ (while bypassing the need to generate $\mathcal{A}$), simply representing $\mathcal{A}^{\mathrm{n}}$ in a straight-forward fashion (by writing the branching conditions on its transitions) can take exponential time (and space).

3. One of the basic operations performed while running our algorithm that checks for query existence is to check whether nodes on the paths to $\mathcal{X}^-$ also satisfy queries on the transitions in $\mathcal{A}^{\mathrm{n}}$. Since branching conditions may be of exponential size, it is not clear how to check for satisfaction in less than exponential time.

We now present a result allowing us to bypass these three problems. Recall that all incoming transitions to a state have the same branching conditions, and that there is at most one transition between any two states. The following result shows that it is possible to iterate over the normalized version $\mathcal{A}^{\mathrm{n}}$ of $\mathcal{A}(\mathcal{C}_{branch}, \mathcal{X}^+)$, to generate mnqv-mappings, without explicit generation of the branching conditions in the transitions. To this end, Theorem 7.3 states that it is possible to check satisfaction of the queries on transitions of $\mathcal{A}^{\mathrm{n}}$ without their explicit generation. Note that this is all that is necessary in order to generate mnqv-mappings.

THEOREM 7.3. *Let $\mathcal{A}^{\mathrm{n}}$ be the normalization of the automaton $\mathcal{A}(\mathcal{C}_{branch}, \mathcal{X}^+)$. Let s_1 and s_2 be states in $\mathcal{A}^{\mathrm{n}}$, such that $s_2 \in \delta(s_1, \sigma[\bar{q}])$, where σ is of the form $\Downarrow l$, $\Downarrow *$, $\downarrow l$ or $\downarrow *$, and $[\bar{q}]$ is a series of branching conditions. Let z_1 and z_2 be nodes in document d. Then, without explicit generation of $[\bar{q}]$, it is possible to determine whether $d \models \sigma[\bar{q}](z_1, z_2)$ in polynomial time, if $\mathcal{X}^+$ is of bounded size.*

Given Theorem 7.3 and the discussion above, we can immediately conclude that query existence is in polynomial time even if branching is allowed.

COROLLARY 7.4. *Query existence, determining certain answers and determining possible answers is in polynomial time, if $\mathcal{X}^+$ and $\mathcal{X}^-$ are of bounded size, for $\mathcal{C}(\downarrow, \Downarrow, *, [\,])$.*

Combining the results in this section and those in Section 7.1, we get the following, stronger, result.

COROLLARY 7.5. *Query existence, determining certain answers and determining possible answers is in polynomial time, if $\mathcal{X}^+$ and $\mathcal{X}^-$ are of bounded size, for $\mathcal{C}(\downarrow, \Downarrow, *, [\,], @)$.*

7.3 Classes Omitting Features

The previous two sections dealt with adding in additional features to the class of queries. In this section, we consider omitting some of the features.

If $\mathcal{C}$ does not contain $\downarrow$, $\Downarrow$ or $*$, we simply do not add transitions for queries with these features during our construction process (see Construction 1). An easy analysis shows that our algorithm remains correct when removing these transitions. Thus, combined with Corollary 7.5 we can show the following.

COROLLARY 7.6. *Query existence, determining certain answers and determining possible answers is in polynomial time, if $\mathcal{X}^+$ and $\mathcal{X}^-$ are of bounded size, for* any *class $\mathcal{C}(F)$, where $F \subseteq \{\downarrow, \Downarrow, *, [\,], @\}$.*

Not adding edges with these features can significantly reduce the size of the query automata. In some cases, this will lead to an automata that has a polynomially sized language. In these cases, query existence will be polynomial even if $\mathcal{X}^+$ and $\mathcal{X}^-$ are unbounded in size.

THEOREM 7.7. *Let $\mathcal{C}(F)$ be a class of queries, d be a document, $\mathcal{X}^+$ and $\mathcal{X}^-$ be sets of nodes from d.*

1. *If F does not contain $\Downarrow$, $[\,]$, then determining whether $\mathcal{Q}_{\mathcal{C}(F)}(\mathcal{X}^+, \mathcal{X}^-)$ is not empty is in polynomial time, even if $\mathcal{X}^+$ and $\mathcal{X}^-$ are unbounded in size.*

2. *If F does not contain $\Downarrow$, then then determining whether $\mathcal{Q}_{\mathcal{C}(F)}(\mathcal{X}^+, \mathcal{X}^-)$ is not empty is in polynomial time if $\mathcal{X}^+$ is bounded, even if $\mathcal{X}^-$ is unbounded.*

Omitting language features and bounding one of $\mathcal{X}^+$ or $\mathcal{X}^-$ does not always reduce the complexity of query existence, as demonstrated in the following theorem. Note that the second claim in Theorem 7.8 is an easy adaptation of a result in [23] to our setting.

THEOREM 7.8. *Let $\mathcal{C}(F)$ be a class of queries, d be a document, $\mathcal{X}^+$ and $\mathcal{X}^-$ be sets of nodes from d.*

1. *If F contains $\Downarrow$, but not $[]$, then it is NP-complete to determine whether $\mathcal{Q}_{\mathcal{C}(F)}(\mathcal{X}^+, \mathcal{X}^-)$ is not empty, if $\mathcal{X}^-$ is unbounded in size, even if $|\mathcal{X}^+| = 2$.*
2. *[23] If F contains $[], \downarrow$, then it is NP-hard to determine whether $\mathcal{Q}_{\mathcal{C}(F)}(\mathcal{X}^+, \mathcal{X}^-)$ is not empty, if $\mathcal{X}^+$ is unbounded in size, even if $|\mathcal{X}^-| = 1$.*

8. CONCLUSION

The goal of this work is to enable an intuitive interface for querying XML. Instead of formulating XPath queries against a document, the user simply marks some nodes of interest $\mathcal{X}^+$, and can also indicate that some nodes $\mathcal{X}^-$ are not of interest. Additional nodes that may interest the user are defined via the notions of certain and possible answers.

The main result of this paper is an efficient algorithm for determining query existence (where query existence generalizes both the problem of determining certain answers and the problem of determining possible answers). Our algorithm is based on the notion of a query automaton, which efficiently summarizes the language of queries returning $\mathcal{X}^+$, and on the notion of verification mappings. Effort is required to prove that the algorithm is in polynomial time, when $\mathcal{X}^+$ and $\mathcal{X}^-$ are bounded. Thus, this paper presents a positive result—that determining certain and possible answers is in polynomial time, even for the class of XPath including child, descendant, wildcards, branching and attribute constraints.

There are many directions for future work. One important problem is that of ranking possible answers. The current work is binary in nature. A node either is, or is not, a possible answer. However, in practice, possible answers should not all be thought of as equally likely to be of interest, and hence the ability to rank possible answers is of importance. Another related problem is that of returning approximately certain (or possible) answers, in the case of an inconsistent set of positive and negative examples. Approximation may also be a useful tool if the number of positive and negative examples grows prohibitively large (e.g., when used in contexts other than ad-hoc querying). Finally, for an implementation, it is important to leverage the document schema in order to more efficiently return certain and possible answers. For example, one can take advantage of similarities in document structure in order to reduce the number of positive/negative examples considered (as examples with the same structure introduce some redundancies).

9. REFERENCES

[1] T. Amoth, P. Cull, and P. Tadepalli. On exact learning of unordered tree patterns. *Machine Learning*, 44:211–243, 2001.

[2] T. R. Amoth, P. Cull, and P. Tadepalli. Exact learning of tree patterns from queries and counterexamples. In *COLT*, pages 175–186, 1998.

[3] D. Angluin. Finding patterns common to a set of strings. In *STOC*, pages 130–141, 1979.

[4] D. Angluin. Negative results for equivalence queries. *Machine Learning*, 5(2):121–150, July 1990.

[5] M. Arenas and L. Libkin. XML data exchange: Consistency and query answering. *J. ACM*, 55(2), 2008.

[6] H. Arimura, H. Ishizaka, and T. Shinohara. Learning unions of tree patterns using queries. *Theor. Comput. Sci.*, 185(1):47–62, 1997.

[7] N. Augsten, D. Barbosa, M. Bohlen, and T. Palpanas. Tasm: Top-k approximate subtree matching. In *ICDE*, pages 353 –364, 2010.

[8] J. Carme, M. Ceresna, and M. Goebel. Query-based learning of XPath expressions. In *ICGI*, pages 342–343, 2006.

[9] M. Ceresna. *Supervised Learning of Wrappers from Structured Data Sources*. PhD thesis, Vienna University of Technology, 2005.

[10] A. Chapman and H. V. Jagadish. Why not? In *SIGMOD*, pages 523–534. ACM, 2009.

[11] A. Das Sarma, A. Parameswaran, H. Garcia-Molina, and J. Widom. Synthesizing view definitions from data. In *ICDT*, pages 89–103, 2010.

[12] C. David, L. Libkin, and F. Murlak. Certain answers for XML queries. In *PODS*, pages 191–202, 2010.

[13] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[14] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: Ranked keyword search over XML documents. In *SIGMOD*, 2003.

[15] M. Herschel, M. A. Hernández, and W.-C. Tan. Artemis: a system for analyzing missing answers. *Proc. VLDB Endow.*, 2:1550–1553, August 2009.

[16] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs. In *ICDE*, pages 367–378, 2003.

[17] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *PVLDB*, 1(1):736–747, 2008.

[18] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. In *PODS*, pages 173–182, 2006.

[19] R. Kosala, M. Bruynooghe, J. Van Den Bussche, and H. Blocked. Information extraction from web documents based on local unranked tree automaton inference. In *IJCAI*, pages 403–408, 2003.

[20] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. Why so? or why no? functional causality for explaining query answers. In *Management of Uncertain Data*, pages 3–17, 2010.

[21] S. Raeymaekers, M. Bruynooghe, and J. Bussche. Learning (k,l)-contextual tree languages for information extraction from web pages. *Machine Learning*, 71(2-3):155–183, June 2008.

[22] D. Shasha, J. T.-L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, pages 39–52, 2002.

[23] S. Staworko and P. Wieczorek. Learning twig and path queries. In *ICDT*, pages 140–154, 2012.

[24] Y. Tian and J. Patel. Tale: A tool for approximate large graph matching. In *ICDE*, pages 963–972, 2008.

[25] Q. T. Tran and C.-Y. Chan. How to conquer why-not questions. In *SIGMOD*, pages 15–26, 2010.

[26] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *SIGMOD*, pages 535–548. ACM, 2009.