IPSJ Transactions on Databases Vol.7 No.2 70–82 (June 2014)

Regular Paper

An Algorithm for Finding top-K Valid XPath Queries

Kosetsu Ikeda[1,a)] Nobutaka Suzuki[1,b)]

Received: December 22, 2013, Accepted: April 3, 2014

Abstract: Suppose that we have a DTD and XML documents valid against the DTD, and consider writing an XPath query to the documents. Unfortunately, a user often does not understand the entire structure of the documents exactly, especially in the case where the documents are very large and/or complex, or the DTD has been updated but the user misses it. In such cases, the user tends to write an invalid XPath query. However, it is difficult for the user to correct the query by hand due to his/her lack of exact knowledge about the entire structure of the documents. In this paper, we propose an algorithm that finds, for an XPath query q, a DTD D, and a positive integer K, top-K XPath queries most syntactically close to q among the XPath queries conforming to D, so that a user select an appropriate query among the K queries. We also present some experimental studies.

Keywords: XPath, DTD, XML

1. Introduction

Suppose that we have a DTD D and XML documents valid against D, and let us consider writing an XPath query to the documents. Unfortunately, a user often does not understand the entire structure of the documents exactly, especially in the case where the documents is a very large and/or complex or D has been updated but the user misses the update. In such cases, the user tends to write an incorrect XPath query q in the sense that q is not "valid" against the documents or the answer of q is disappointing due to his/her structural misunderstanding of the documents. However, it is difficult for the user to correct q by hand due to his/her lack of exact knowledge about the entire structure of the documents. On the other hand, a query q written by a user is at least an important "hint" in order to find a correct query, even if q is incorrect.

Therefore, in this paper we propose an algorithm that finds, for an (possibly incorrect) XPath query q, a DTD D, and a positive integer K, top-K XPath queries syntactically close to q among the XPath queries valid against D, so that a user may select a desirable query from the top-K queries.

As a brief example of our algorithm, let us consider the following simple DTD D.

```
<!ELEMENT site          (people)>
<!ELEMENT people        (person)*>
<!ELEMENT person        (name, email, phone?)>
<!ELEMENT name          (#PCDATA)>
<!ELEMENT email         (#PCDATA)>
<!ELEMENT phone         (#PCDATA)>
<!ATTLIST person id ID #REQUIRED>
```

Suppose that a user wants `name` element of the person whose id is "123" and that he/she tries to use an XPath query q = `/person[@id = "123"]/naem`, which is not valid against D. Our algorithm finds XPath queries "syntactically close" to q based on the *edit distance* between XPath queries, proposed in this paper. In this example, our algorithm lists the following top-K XPath queries syntactically close to q (assuming that $K = 3$). Each XPath query q' is followed by the edit distance between q and q', assuming that the cost of relabeling l with l' is the normalized string edit distance between l and l' [16].

```
1. //person[@id = "123"]/name              (0.75)
2. //people/person[@id = "123"]/name       (1.75)
3. /site/people/person[@id = "123"]/name   (2.25)
```

As above, by our algorithm the user can obtain top-K correct XPath queries syntactically close to q without modifying q by hand even if he/she does not know the exact structure of D. Although the above DTD D is very small, DTDs used in practice are larger and more complex [4]. In such a situation, a user tends not to understand the entire structure of a DTD exactly, and thus our algorithm is helpful for writing correct XPath queries on such DTDs.

In this paper, we focus on XPath queries using child, descendant-or-self, following-sibling, preceding-sibling, and attribute axes. Although our XPath queries support no upward axes, this usually gives little problem since the majority of XPath queries uses only downward axes [11]. Thus, we believe that our algorithm is useful to correct a large number of XPath queries.

An overview of our algorithm is as follows. Let q be an XPath query and D be a DTD D. To obtain top-K queries syntactically close to q under D, we first compute the set of valid queries obtained by correcting q, then select top-K queries close to q among the valid queries. To obtain such a set of valid queries, we construct a graph called "xd-graph" (**Fig. 1**). The important point of xd-graph is that valid queries obtained by correcting q are mapped to paths from the start node to the accepting node. For example, consider the XPath query q and the xd-graph in Fig. 1. The path $n_0 \to a_0 \to d_1 \to c_2$ on the xd-graph represents a valid query $/\downarrow::a/\downarrow::d/\downarrow::c$, which is obtained by inserting $/\downarrow::a$ to q

[1] University of Tsukuba, Tsukuba, Ibaraki 305–8550, Japan
[a)] lumely@slis.tsukuba.ac.jp
[b)] nsuzuki@slis.tsukuba.ac.jp

and substituting the label of the first location step ↓:: e with d. Similarly, the other paths from the start node n_0 to the accepting node c_2 represent valid queries obtained by correcting q. Another important point is that, for any path p from the start node to the accepting node, the cost of p represents the cost of correcting the input query to the valid query represented by p. Therefore, once an xd-graph is obtained, it suffices to solve the K shortest paths problem over the xd-graph to obtain top-K valid queries syntactically close to q.

Related Work

Reference [5] proposes an algorithm that finds valid tree pattern queries most syntactically close to an input query. Their algorithm and ours are incomparable due to the underlying data models; in their data model a tree is unordered and a schema is represented by a DAG, while we use DTD (recursion is supported) and a tree is ordered. Since in our data model a schema allows cycles and a query allows sibling axes ($\rightarrow^+$, $\leftarrow^+$), their algorithm cannot be applied to our data model. Note that Choi investigated 60 DTDs and 35 of the DTDs are recursive [4], which suggests that it is meaningful to support recursive schemas. Besides query correction, several related but different approaches have been studied for XML; query expansion, inexact queries, interaction, keyword search, etc. Reference [18] takes a query expansion approach instead of correcting queries. References [2], [3], [7], [8] deal with a top-K query evaluation for XML documents to derive inexact answers, i.e., evaluating a "relaxed" version of the input query, if it is unsatisfiable. Inexact querying is also studied in Refs. [13], [14], in which a user can write an XQuery query without specifying exact connections between elements. Reference [17] proposes an interactive system for generating XQuery queries. There has been a number of studies on XML keyword search (e.g., Refs. [12], [20], [21]), which are especially suitable for users that are not familiar with XML query languages. Several XML editors (e.g., XMLSpy [1]) support auto-complete for XPath query editing, but they do not support listing K correct XPath queries. Finally, this paper is a revised version of Ref. [10]. Besides a number of minor fixes, this paper provides (i) a discussion on location step exchange operation (Section 4) and (ii) a revised version of experimental results (Section 7).

The rest of this paper is organized as follows. Section 2 gives some preliminaries. Section 3 defines edit operations to XPath queries. Section 4 shows that finding top-K valid XPath queries is NP-hard if location step exchange is allowed as an edit operation. Section 5 introduces *xd-graph*, which forms the basis of our algorithm. Section 6 gives algorithms for finding top-K valid XPath queries. Section 7 shows some experimental results. Section 8 summarizes this paper.

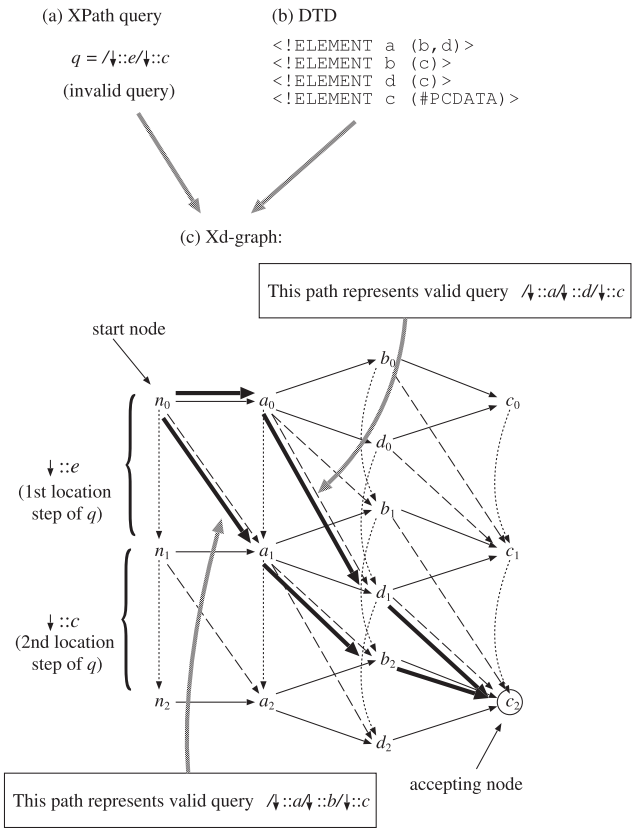

Fig. 1 Overview of our algorithm.

2. Preliminaries

Let Σ_e be a set of labels (element names) and Σ_a be a set of attribute names with $\Sigma_e \cap \Sigma_a = \emptyset$. A *DTD* is a triple $D = (d, \alpha, s)$, where d is a mapping from Σ_e to the set of regular expressions over Σ_e, α is a mapping from Σ_e to 2^{Σ_a}, and $s \in \Sigma_e$ is the *start label*. For example, the DTD in Section 1 is a triple (d, α, site), where $d(\text{site}) = \text{people}$, $d(\text{people}) = \text{person}^*$, $d(\text{person}) = (\text{name}, \text{emailaddress}, \text{phone?})$, $d(\text{name}) = \epsilon$, $\alpha(\text{name}) = \{\text{id}\}$, and $\alpha(e) = \emptyset$ for any element $e \neq$ name.

By $L(d(a))$ we mean the language of $d(a)$. For labels b, c, if there is a string $str \in L(d(a))$ such that $str[i] = c$ and $str[j] = b$ with $i < j$ ($i > j$), then we say that b can be *right* (resp., *left*) to c in $d(a)$, where $str[i]$ denotes the ith character of str. For example, e can be right to c in $d(a) = c(f|e)^*$.

For a DTD $D = (d, \alpha, s)$ and labels $a, b \in \Sigma_e$, b is *reachable* from a in D if (i) $a = b$ or b appears in $d(a)$, or (ii) for some label a', a' is reachable from a and b appears in $d(a')$. In the following, we assume that any label in a DTD is reachable from the start label of the DTD.

In this paper, we use XPath queries using child (↓), descendant-or-self (↓*), following-sibling ($\rightarrow^+$), preceding-sibling ($\leftarrow^+$), and attribute (@) axes. The set of such XPath queries is denoted XP. Formally, XP is the set of XPath queries defined in **Table 1**. Thus, an *XPath query* (*query* for short) q in XP can be denoted

$$/ax[1] :: l[1][exp[1]]/ \cdots /ax[m] :: l[m][exp[m]], \qquad (1)$$

Table 1 Syntax of XP.

XP	::=	"/" RelativePath | "/" RelativePath "@" Attribute
RelativePath	::=	LocationStep | LocationStep "/" RelativePath
LocationStep	::=	Axis "::" Label | Axis "::" Label Predicate
Axis	::=	"↓" | "↓*" | "$\rightarrow^+$" | "$\leftarrow^+$"
Label	::=	(any label in Σ_e)
Attribute	::=	(any label in Σ_a)
Predicate	::=	"[" Exp "]"
Exp	::=	PredPath | PredPath Op Value
PredPath	::=	RelativePath | "@" Attribute | RelativePath "@" Attribute
Op	::=	"=" | "<" | ">" | "=<"| "=>"
Value	::=	"" (any string other than "") ""

where $ax[i] \in$ Axis and $l[i] \in \Sigma_e$ for $1 \le i \le m-1$, $exp[i] \in$ Exp for $1 \le i \le m$, $ax[m] \in$ Axis $\cup \{@\}$, and $l[m] \in \Sigma_a$ if $ax[m] = @$, $l[m] \in \Sigma_e$ otherwise. If the ith location step has no predicate, then we write $exp[i] = \epsilon$.

Let q be a query in (1). For indexes i, j such that $ax[i] \in \{\downarrow, \downarrow^*\}$ and that $ax[i+1], \cdots, ax[j] \in \{\rightarrow^+, \leftarrow^+\}$, we say that l is the *parent label* of $l[j]$ in q if (i) $ax[i] = \downarrow$ and $l = l[i-1]$, or (ii) $ax[i] = \downarrow^*$, l is reachable from $l[i-1]$, and $l[i]$ appears in $d(l)$. For example, if $q = /\downarrow::a/\downarrow::b/\rightarrow^+::c/\leftarrow^+::d$, then a is the parent label of b, c, d in q.

Let $D = (d, \alpha, s)$ be a DTD. Then q is (syntactically) *valid* against D if the following conditions hold.

- $ax[1] = \downarrow$ and $l[1] = s$, or, $ax[1] = \downarrow^*$ and $l[1] \in \Sigma_e$
- The following condition holds for every $2 \le i \le m$
  - $ax[i] = \downarrow$ and $l[i]$ appears in $d(l[i-1])$,
  - $ax[i] = \downarrow^*$ and $l[i]$ is reachable from $l[i-1]$ in D,
  - $ax[i] = \rightarrow^+$ and $l[i]$ can be right to $l[i-1]$ in $d(l)$, where l is the parent label of $l[i]$ (the case where $ax[i] = \leftarrow^+$ is defined similarly), or
  - $ax[i] = @$, $i = m$, and $l[i] \in \alpha(l[i-1])$.
- For every $1 \le i \le m$ with $exp[i] \ne \epsilon$, query $/\downarrow::l[i]/exp[i]$ is valid against DTD $(d, \alpha, l[i])$.

By $|q|$ we mean the number of location steps in q, e.g., if $q = /\downarrow::a/\downarrow::b[\leftarrow^+::d]$, then $|q| = 3$. If a query q has neither predicate nor attribute axis, then we say that q is *simple*.

3. Edit Operations to XPath Query

In this section, we define edit operations to queries. We propose the following six kinds of *edit operations*.

- *Axis substitution:* substitutes axis ax with ax', denoted $ax \rightarrow ax'$. For example, by applying $\downarrow \rightarrow \downarrow^*$ to $/\downarrow::a$ we obtain $/\downarrow^*::a$.
- *Label substitution:* substitutes label l with l', denoted $l \rightarrow l'$. For example, by applying $a \rightarrow b$ to $/\downarrow::a$ we obtain $/\downarrow::b$.
- *Location step insertion:* inserts location step $ax :: l$, denoted $\epsilon \rightarrow ax :: l$. For example, by applying $\epsilon \rightarrow \downarrow::b$ to the tail of $/\downarrow::a$ we obtain $/\downarrow::a/\downarrow::b$.
- *Location step deletion:* deletes location step $ax :: l$, denoted $ax :: l \rightarrow \epsilon$. For example, by applying $\downarrow::a \rightarrow \epsilon$ to the first location step of $/\downarrow::a/\downarrow::b$ we obtain $/\downarrow::b$.
- *Location step exchange:* exchanges adjacent two location steps. For example, by applying this edit operation to $/\downarrow::a/\downarrow^*::b$ we obtain $/\downarrow^*::b/\downarrow::a$.
- *Predicate exchange:* exchanges the predicates of adjacent two location steps. For example, by applying this edit operation to $/\downarrow::a[b/d]/\downarrow^*::c$ we obtain $/\downarrow::a/\downarrow^*::c[b/d]$.

We next define the *position* of a location step ls, denoted $pos(ls)$. Let $q = /ax[1] :: l[1][exp[1]]/\cdots/ax[m] :: l[m][exp[m]] \in$ XP. We define that $pos(ax[i] :: l[i]) = i$ for $1 \le i \le m$. As for location steps in predicates, let $exp[i] = ax'[1] :: l'[1][exp'[1]]/\cdots/ax'[n] :: l'[n][exp'[n]]$. Then we define that $pos(ax'[j] :: l'[j]) = i.j$ for $1 \le j \le n$. The position of a location step in $exp'[j]$ can be defined similarly. For example, let $q = /\downarrow::a/\downarrow::b[\downarrow::d[\downarrow::g]]/\rightarrow^+::c$. Then $pos(\downarrow::b) = 2$, $pos(\downarrow::d) = 2.1$, and $pos(\downarrow::g) = 2.1.1$. By $[op]_{pos}$, we mean an edit operation op applied to the location step at position pos. If op is an edit operation inserting a location step ls, then $[op]_{pos}$ inserts ls just after the location step at pos.

Let $q \in$ XP. An *edit script* for q is a sequence of edit operations having a position in q. For an edit script s for q, by $s(q)$ we mean the query obtained by applying s to q. For example, let $s = [\epsilon \rightarrow \downarrow::b]_1\,[c \rightarrow f]_3$ and $q = /\downarrow^*::a/\downarrow::d/\downarrow::c$. Then we have $s(q) = /\downarrow^*::a/\downarrow::b/\downarrow::d/\downarrow::f$.

Throughout this paper, we assume the following. Let $U = \{\downarrow, \downarrow^*\}$, $S = \{\rightarrow^+, \leftarrow^+\}$, and $A = \{@\}$.

- An axis can be substituted with an axis of "same kind" only, that is, $ax \in U$ (resp., S, A) can be substituted with an axis in U (resp., S, A) only.
- A location step $ax :: l$ can be inserted to a query only if $ax \in U$ and $l \in \Sigma_e$.

A *cost function* assigns a cost to an edit operation. By $\gamma(op)$ we mean the *cost* of an edit operation op, where γ is a cost function. In the following, we assume that $\gamma(op) \ge 0$. A cost function can be a general function as well as a constant. For example, $\gamma(op)$ can be a string edit distance between l and l' if $op = l \rightarrow l'$. For an edit script $s = op_1 op_2 \cdots op_n$, by $\gamma(s)$ we mean the *cost* of s, that is, $\gamma(s) = \sum_{1 \le i \le n} \gamma(op_i)$. For a DTD D, a query q, and a positive integer K, the *K optimum edit script* for q under D is a sequence of edit operations $s_1, \cdots, s_K$ if (i) each of $s_1(q), \cdots, s_K(q)$ is valid against D, (ii) $\gamma(s_1) \le \cdots \le \gamma(s_K)$, and (iii) $s_1, \cdots, s_K$ are optimum, that is, for any edit script s for q such that $s(q)$ is valid against D, $s(q) \in \{s_1(q), \cdots, s_K(q)\}$ or $\gamma(s) \ge \gamma(s_K)$. We say that $s_1(q), \cdots, s_K(q)$ are *top-K* queries *syntactically close* to q under D.

4. Operations Causing Intractability

In this section, we show that location step exchange and predicate exchange make finding top-K valid queries intractable. Let us consider the following decision problem, called *query correction problem*.

Input: A DTD D, a query q, and a positive integer K.
Problem: Determine whether there is an edit script s to q such that $\gamma(s) \le K$ and that $s(q)$ is valid against D.

We have the following result.

Theorem 1 If location step exchange is allowed, the query correction problem is NP-hard.

Proof We reduce the directed Hamiltonian path problem, which is NP-complete, to the query correction problem. The directed Hamiltonian path problem is defined as follows.

Input: A directed graph $H = (V, E)$ and nodes $u, v \in V$.
Problem: Determine whether H contains a directed Hamiltonian path from u to v.

Let $H = (V, E)$ and $u, v \in V$ be an instance of the directed Hamiltonian path problem, where $V = \{v_1, v_2, \cdots, v_k\}$. From this instance, we define a DTD D, a query q, and a positive integer K. First, D is defined to simulate H. Formally, Σ_e, Σ_a, and $D = (d, \alpha, s')$, are defined as follows.

$$\Sigma_e = \{v_1, v_2, \cdots, v_k\},$$
$$\Sigma_a = \emptyset,$$
$$s' = u,$$
$$d(v_i) = seq_i, \quad (1 \le i \le k)$$

where seq_i is any sequence of labels such that v_j occurs in seq_i iff $v_i \to v_j \in E$.

Second, query q is defined as follows.

$$q = /\downarrow::v_1/\downarrow::v_2/\cdots/\downarrow::v_k.$$

As for the costs of edit operations, we define that the cost of a location step exchange is one and that the costs of the other edit operations are ∞. Let $K = \frac{1}{2}k^2(k+1)$.

In the following, we show that H has a directed Hamiltonian path from u to v iff there is an edit script s to q such that $\gamma(s) \le K$ and that $s(q)$ is valid against D.

($\Rightarrow$) Assume that H has a directed Hamiltonian path from u to v, say $u = v_{i_1} \to v_{i_2} \to \cdots \to v_{i_k} = v$. By the definition of D, query $q' = /\downarrow::v_{i_1}/\downarrow::v_{i_2}/\cdots/\downarrow::v_{i_k}$ is valid against D. Since $\{v_{i_1}, v_{i_2}, \cdots, v_{i_k}\} = V$, there is an edit script s to q consisting of only location step exchanges such that $\gamma(s) \le \frac{1}{2}k^2(k+1) = K$ and that $s(q) = q'$.

($\Leftarrow$) Assume that H has no directed Hamiltonian path from u to v. Then any path from u to v on H of length k visits some same node more than once. This and the definition of D imply that in order to make q valid, we need to use a location step deletion, a location step insertion, or a label substitution, which costs ∞. Thus there is no edit script s to q such that $\gamma(s) \le K$ and that $s(q)$ is valid. □

We also have the following.

Theorem 2 If predicate exchange and label substitution are allowed at the same time, the query correction problem is NP-hard.

Proof Again we reduce the directed Hamiltonian path problem to the query correction problem. Let $H = (V, E)$ and $u, v \in V$ be an instance of the directed Hamiltonian path problem, where $V = \{v_1, v_2, \cdots, v_k\}$. From this instance, we define a DTD D, a query q, and a positive integer K. First, D is defined to simulate H. Formally, Σ_e, Σ_a, and $D = (d, \alpha, s')$, are defined as follows.

$$\Sigma_e = \{v\} \cup \{v_1, v_2, \cdots, v_k\} \cup \{a_1, a_2, \cdots, a_k\},$$
$$\Sigma_a = \emptyset,$$
$$s' = u,$$
$$d(a_i) = \epsilon, \quad (1 \le i \le k)$$
$$d(v_i) = seq_i, \quad (1 \le i \le k)$$

where seq_i is a sequence of labels such that v_j occurs in seq_i iff $v_i \to v_j \in E$ or $v_j = a_i$.

Second, query q is defined as follows.

$$q = /\downarrow::v[\downarrow::a_1]/\downarrow::v[\downarrow::a_2]/\cdots/\downarrow::v[\downarrow::a_k].$$

As for the costs of edit operations, we define that for any $l' \in \Sigma_e$

$$\gamma(l \to l') = \begin{cases} 1 & \text{if } l = v, \\ \infty & \text{otherwise,} \end{cases}$$

the cost of a predicate exchange is one, and that the costs of the other edit operations are ∞. Let $K = \frac{1}{2}k^2(k+1) + k$.

We can show similarly to Theorem 1 that H has a directed Hamiltonian path $u = v_{i_1} \to v_{i_2} \to \cdots \to v_{i_k} = v$ iff there is an edit script s to q such that $s(q) = /\downarrow::v_{i_1}[\downarrow::a_{i_1}]/\downarrow::v_{i_2}[\downarrow::a_{i_2}]/\cdots/\downarrow::v_{i_k}[\downarrow::a_{i_k}]$, $\gamma(s) \le K$, and that $s(q)$ is valid against D. □

Thus, it is unlikely that we can find top-K valid queries efficiently, if we use location step exchange or predicate exchange. In the following sections, we consider finding top-K valid queries without location step exchange.

Finally, let us consider the expressive power of the edit operations. Under the following restriction, the six edit operations are "complete" in the sense that any query can be transformed into another arbitrary query by using these edit operations.

- No new predicate can be added to a query.

We believe that this restriction is reasonable since (i) it is unnatural to add a predicate that is not written by a user and (ii) it is hardly possible to "infer" an appropriate predicate from an "empty" predicate. The completeness can be shown easily since a query q can be transformed into another query q' by deleting every location step of q and inserting every location step of q'. In fact, the completeness still holds even if location step exchange and predicate exchange are omitted, although more edit operations may be required to correct a query. For example, let $q = /\downarrow::a/\downarrow::b$ and $q' = /\downarrow::b/\downarrow::a$. If location step exchange is allowed, q can be transformed into q' by just one edit operation, otherwise at least two edit operations are required. In summary, since the completeness is preserved, we believe that our edit operations have an expressive power enough to handle the problem of correcting invalid queries, even if location step exchange and predicate exchange are omitted.

5. Xd-Graph Representing Valid Queries

In this section, we introduce a graph called *xd-graph*, which forms the basis of our algorithm. An xd-graph is constructed from an XPath query and a DTD, and as we will see below, the xd-graph represents the set of valid queries obtained by correcting the input query.

Throughout this section, we assume that each query is simple.

5.1 Overview

For a query q and a DTD D, in order to obtain top-K queries syntactically close to q under D, we first need to compute the set of valid queries obtained by correcting q, then select top-K queries close to q among the set of valid queries. However, it is not obvious how to obtain such a set of valid queries in which top-K queries can be found easily. To cope with this problem, in this paper we construct a graph called "xd-graph" from q and D, as shown in **Fig. 2**. An xd-graph has a start node and an accepting node, and valid queries obtained by correcting q are mapped to paths from the start node to the accepting node. Thus, in order to obtain top-K queries syntactically close to q under D, it suffices to solve the K shortest paths problem over the xd-graph.

Let us show the structure of xd-graph. As shown in Fig. 2 (d), an xd-graph consists of $|q| + 1$ DTD graphs and some edges connecting the DTD graphs, where the DTD graph of D represents the parent-child relationships between labels occurring in D (Fig. 2 (c)). The reason why we use such multiple DTD graphs is that we have to represent every edit operation to each location step of q. As shown in **Fig. 3**, the edges between the first and sec-

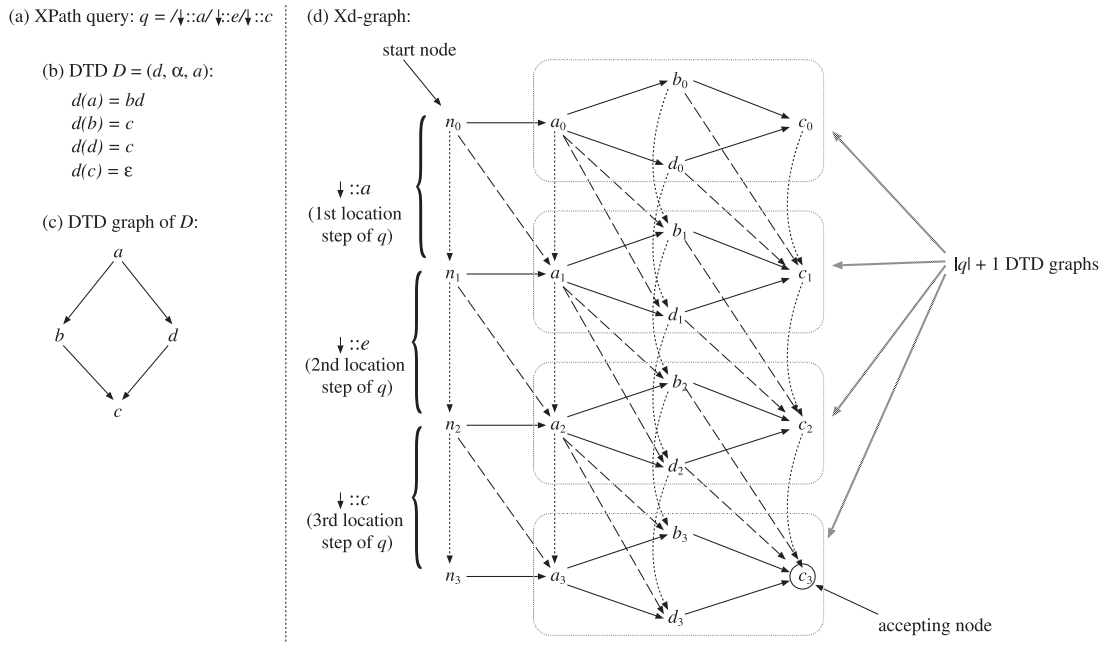

Fig. 2 An example of xd-graph.

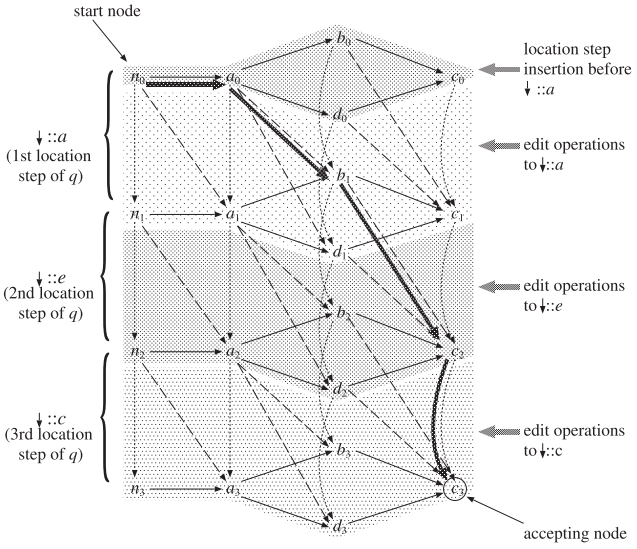

Fig. 3 Xd-graph.

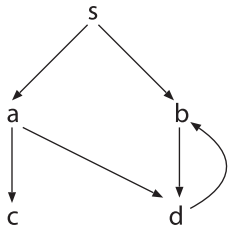

Fig. 4 A DTD graph $G(D)$.

ond DTD graphs represent edit operations to the first location step $\downarrow::a$, the edges between the second and third DTD graphs represent edit operations to the second location step $\downarrow::e$, and so on. To identify the edit operation of an edge, an xd-graph has several kinds of edges; a "horizontal" edge $l_i \rightarrow l'_i$ corresponds to a location step insertion, each "slant" edge $l_{i-1} \dashrightarrow l'_i$ corresponds to a label substitution, and each "vertical" edge $l_{i-1} \dashrightarrow l_i$ corresponds to a location step deletion. For example, consider the edges on the path $n_0 \rightarrow a_0 \dashrightarrow b_1 \dashrightarrow c_2 \dashrightarrow c_3$ (the thick path in Fig. 3).

- The first "horizontal" edge $n_0 \rightarrow a_0$ represents inserting location step $\downarrow::a$ before the first location step $\downarrow::a$.
- The second "slant" edge $a_1 \dashrightarrow b_1$ represents substituting the label of the first location step $\downarrow::a$ with b.
- The third "slant" edge $b_1 \dashrightarrow c_2$ represents substituting the label of the second location step $\downarrow::e$ with c.
- The last "vertical" edge $c_2 \dashrightarrow c_3$ represents deleting the third location step $\downarrow::c$.

Thus, the thick path represents correcting $q = /\downarrow::a/\downarrow::e/\downarrow::c$ to valid query $/\downarrow::a/\downarrow::b/\downarrow::c$. Each edge has a cost according to the edit operation associated with the edge. Thus, the cost of each path p from the start node to the accepting node represents the cost of correcting the input query to the valid query represented by p.

In the following, we first present the detailed examples of xd-graph (Section 5.2), then give the formal definitions (Section 5.3).

5.2 Xd-Graph Examples

To construct an xd-graph, we need a graph representation of DTD. The *DTD graph* $G(D)$ of a DTD $D = (d, \alpha, s)$ is a directed graph (V, E), where $V = \Sigma_e$ and $E = \{l \rightarrow l' \mid l'$ is a label appearing in $d(l)\}$. For example, **Fig. 4** is the DTD graph of $D = (d, \alpha, s)$, where $d(s) = ba^*$, $d(a) = c|d$, $d(b) = d$, $d(c) = \epsilon, d(d) = b|\epsilon$.

Now let us illustrate xd-graph. We first present the following three cases by examples, then define xd-graph formally.

Case A) Only child ($\downarrow$) can be used as an axis.

Case B) Descendant-or-self ($\downarrow^*$) can be used as well as $\downarrow$.

Case C) Sibling axes ($\rightarrow^+$, $\leftarrow^+$) can be used as well as $\downarrow$ and $\downarrow^*$.

Case A) Only child ($\downarrow$) can be used as an axis.

Let us first illustrate the xd-graph constructed from a simple query $q = /\downarrow::a/\downarrow::d$ and the DTD graph $G(D)$ in Fig. 4. Since only $\downarrow$ axis is allowed, it suffices to consider location step

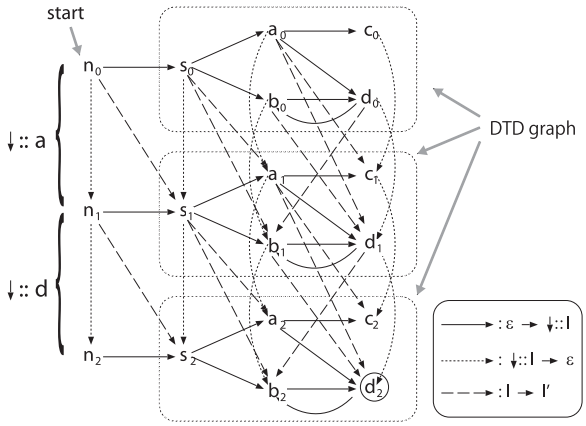

Fig. 5 An xd-graph $G(q, G(D))$.

insertion, location step deletion, and label substitution. **Figure 5** shows xd-graph $G(q, G(D))$. The xd-graph is constructed from 3 copies of $G(D)$ with their nodes connected by several edges. Here, n_0, n_1, n_2 are newly added nodes, which correspond to the "root node" in the XPath data model. Each node is subscripted, e.g., the node s in $G(D)$ is denoted s_0 on the topmost DTD graph of $G(p, G(D))$, s_1 on the second topmost DTD graph, and so on, as shown in Fig. 5.

We have the following three kinds of edges in an xd-graph.

- A "horizontal" edge $l \to l'$ corresponds to a location step insertion.
- A "slant" edge $l \dashrightarrow l'$ corresponds to a label substitution.
- A "vertical" edge $l \dashrightarrow l'$ corresponds to a location step deletion.

More concretely, let us first consider horizontal edge $n_0 \to s_0$ in Fig. 5. This edge means "moving from the root node to child node s, using no location step of q." In other words, the edge $n_0 \to s_0$ represents adding a location step $\downarrow::\ s$, that is, the edge represents an edit operation $[\epsilon \to \downarrow::\ s]_0$. Let us next consider slant edge $s_0 \dashrightarrow b_1$ in Fig. 5. This edge means "moving from node s to child node b using the first location step $\downarrow::\ a$ of q." Since the target node is b rather than a, we have to substitute the label of $\downarrow::\ a$ with b, that is, the edge $s_0 \dashrightarrow b_1$ represents $[a \to b]_1$. Finally, consider vertical edge $b_1 \dashrightarrow b_2$ in Fig. 5. This edge means "staying the same node b by ignoring (deleting) the second location step $\downarrow::\ d$ of q." Thus the edge $b_1 \dashrightarrow b_2$ represents $[\downarrow::\ d \to \epsilon]_2$.

In Fig. 5, n_0 is called *start node* and d_2 is called *accepting node*. Each path from the start node to the accepting node represents a simple query valid against D obtained by correcting q. For example, let us consider a path $p = n_0 \to s_0 \dashrightarrow a_1 \dashrightarrow d_2$ in Fig. 5. Recall that $q = /\downarrow::\ a/\downarrow::\ d$. The first edge $n_0 \to s_0$ represents a location step insertion $[\epsilon \to \downarrow::\ s]_0$. The second edge $s_0 \dashrightarrow a_1$ represents a label substitution $[a \to a]_1$, i.e., the first location step "$\downarrow::\ a$" of q is unchanged. Similarly, the location step "$\downarrow::\ d$" of q is unchanged. Thus, p represents a query $q' = /\downarrow::\ s/\downarrow::\ a/\downarrow::\ d$, which is obtained by applying $[\epsilon \to \downarrow::\ s]_0[a \to a]_1[d \to d]_2$ to q. Note that q' is valid against D.

Case B) Descendant-or-self ($\downarrow^*$) can be used as well as $\downarrow$.

In this case, we can use $\downarrow^*$ axes as well as $\downarrow$ axes. Let us first consider an edit operation inserting location step $\downarrow^*::\ l$ to a query.

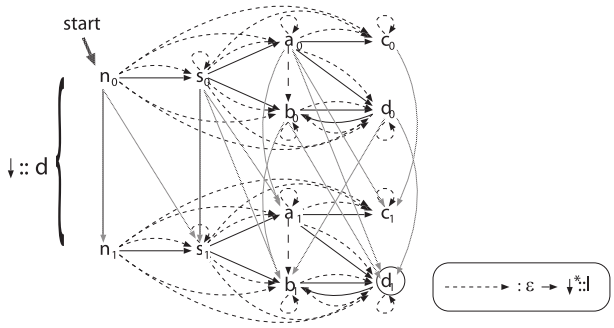

Fig. 6 Edges representing location step insertion.

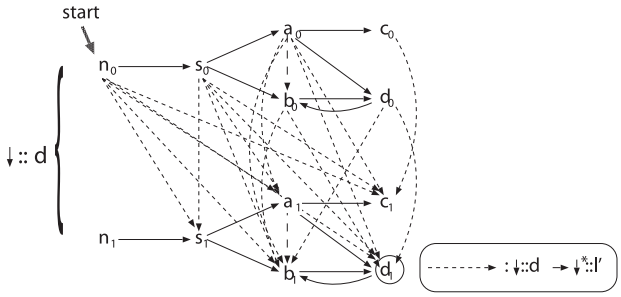

Fig. 7 Edges representing axis substitution.

For this insertion, we add edges representing the edit operation to an xd-graph. **Figure 6** shows the xd-graph constructed from the DTD graph in Fig. 4 and a query $q = /\downarrow::\ d$. Each dashed edge in Fig. 6 represents a location step insertion. For example, $s_0 \dashrightarrow d_0$ means "moving from node s to node d via $\downarrow^*$ axis, using no location step of q," that is, inserting a location step $\downarrow^*::\ d$ at position 0 of q, i.e., $[\epsilon \to \downarrow^*::\ d]_0$. As stated before, every path from the start node to the accepting node represents a simple query valid against D, which is obtained by correcting q. For example, $n_0 \to s_1 \dashrightarrow d_1$ represents a simple query $/\downarrow::\ s/\downarrow^*::\ d$ obtained by applying $[d \to s]_1[\epsilon \to \downarrow^*::\ d]_1$ to $q = /\downarrow::\ d$.

Let us next consider axis substitution between $\downarrow$ and $\downarrow^*$. **Figure 7** shows the xd-graph constructed from the same DTD graph as above and the same query $q = /\downarrow::\ d$. In the figure, for simplicity we omit some of the edges representing location step insertion, location step deletion, and label substitution. In Fig. 7, a dashed edge represents substituting $\downarrow::\ a$ with $\downarrow^*::\ l$. For example, $n_0 \dashrightarrow a_1$ means "moving from the root node to a with $\downarrow^*$ axis," i.e., substituting $\downarrow::\ d$ with $\downarrow^*::\ a$. Here, consider path $p = n_0 \to s_0 \dashrightarrow d_1$ in Fig. 7. p represents a query $/\downarrow::\ s/\downarrow^*::\ d$, which is obtained by applying $[\epsilon \to \downarrow::\ s]_0[\downarrow \to \downarrow^*]_1$ to $q = /\downarrow::\ d$.

Finally, substituting $\downarrow^*$ with $\downarrow$ can be represented by a slant edge similar to label substitution ($l \to l'$), and the deletion of a location step using $\downarrow^*$ axis can be handled similarly to the location step deletion in Case A.

Case C) Sibling axes ($\rightarrow^+$, $\leftarrow^+$) can be used as well as $\downarrow$ and $\downarrow^*$.

Let us consider handling $\rightarrow^+$ and $\leftarrow^+$ axes. **Figure 8** shows the xd-graph constructed from the same DTD graph as above and a query $q = /\rightarrow^+::\ d$. First, let us consider edges connecting the same labels having distinct subscripts, e.g., $s_0 \to s_1$ and $a_0 \to a_1$. Such an edge means that the position does not change (ignoring $\rightarrow^+::\ d$ of q) and $\rightarrow^+::\ d$ is deleted from q.

Let us next consider dashed edges connecting "sibling labels."

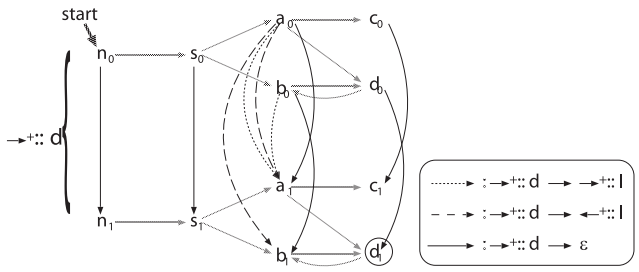

Fig. 8 Edges dealing with $\rightarrow^+$ and $\leftarrow^+$ axes.

For example, we have four edges between a_0, b_0 and a_1, b_1 (e.g., $a_0 \dashrightarrow b_1$, $b_0 \dashrightarrow a_1$) since a and b are siblings in $d(s) = ba^*$. A dashed edge $\dashrightarrow$ represents substituting a sibling axis ($\rightarrow^+$ or $\leftarrow^+$) with $\rightarrow^+$, and another dashed edge $\dashrightarrow$ represents substituting a sibling axis with $\leftarrow^+$. For example, $a_0 \dashrightarrow b_1$ means "moving from node a to b via $\leftarrow^+$ axis," that is, substituting the location step $\rightarrow^+:: d$ of q with $\leftarrow^+:: b$. An xd-graph has no edge violating a DTD, e.g., Fig. 8 does not have edge $b_0 \dashrightarrow a_1$ since $d(s) = ba^*$ and a cannot be left to b.

5.3 Formal Definition of Xd-Graph

Let $D = (d, \alpha, s)$ be a DTD, $G(D) = (V, E)$ be the DTD graph of D, and $q = /ax[1] :: l[1]/\cdots/ax[m] :: l[m]$ be a simple query. Let $G_i(D) = (V_i, E_i)$ be a graph obtained by adding a subscript i to each node of $G(D)$, that is, $V_i = \{l_i \mid l \in V\}$ and $E_i = \{l_i \rightarrow l'_i \mid l \rightarrow l' \in E\}$ for $0 \le i \le m$. The *xd-graph* for q and $G(D)$, denoted $G(q, G(D))$, is a directed graph (V', E'), where

$$\begin{aligned} V' &= \{n_0, n_1, \cdots, n_m\} \cup V_0 \cup V_1 \cup \cdots \cup V_m, \\ E' &= E_{insc} \cup (E'_0 \cup E'_1 \cup \cdots \cup E'_m) \\ &\quad \cup (F_1 \cup F_2 \cup \cdots \cup F_m). \end{aligned} \tag{2}$$

Here, E_{insc} in Eq. (2) is the set of edges inserting $\downarrow:: l$ (corresponding to "$\epsilon \rightarrow\downarrow:: l$" in Fig. 5), that is, $E_{insc} = \{n_0 \rightarrow s_0, \cdots, n_m \rightarrow s_m\} \cup (E_0 \cup \cdots \cup E_m)$, where E_i is the set of edges of $G_i(D)$. E'_i in Eq. (2) is the set of edges inserting $\downarrow^*:: l$ (corresponding to "$\epsilon \rightarrow\downarrow^*:: l$" in Fig. 6) and is defined as follows.

$$E'_i = \{n_i \rightarrow l_i \mid l_i \in V_i\} \cup \{l_i \rightarrow l'_i \mid l' \text{ is reachable from } l \text{ in } D\}.$$

F_i in Eq. (2) is the set of edges between $G_{i-1}(D)$ and $G_i(D)$ defined as follows. We have two cases to be considered.
1) The case where $ax[i] \in \{\downarrow, \downarrow^*\}$: $F_i = D_i \cup C_i \cup A_i$, where

$$D_i = \{n_{i-1} \rightarrow n_i\} \cup \{l_{i-1} \rightarrow l_i \mid l \in V\}, \tag{3}$$

$$C_i = \{n_{i-1} \rightarrow s_i\} \cup \{l_{i-1} \rightarrow l'_i \mid l \rightarrow l' \in E\},$$

$$\begin{aligned} A_i &= \{n_{i-1} \rightarrow l_i \mid l_i \in V_i\} \\ &\quad \cup \{l_{i-1} \rightarrow l'_i \mid l' \text{ is reachable from } l \text{ in } D\}, \end{aligned} \tag{4}$$

Here, D_i is the set of edges corresponding to "$\downarrow:: l \rightarrow \epsilon$" in Fig. 5, C_i is the set of edges corresponding to "$l \rightarrow l'$" in Fig. 5, and A_i is the set of edges corresponding to "$\downarrow:: d \rightarrow\downarrow^*:: l$" in Fig. 7.
2) The case where $ax[i] \in \{\leftarrow^+, \rightarrow^+\}$: $F_i = D_i \cup L_i \cup R_i$, where

$$\begin{aligned} L_i &= \{l_{i-1} \rightarrow l'_i \mid \\ &\quad l' \text{ can be left to } l,\ l'' \text{ is the parent label of } l, l' \text{ in } d(l'')\}, \\ R_i &= \{l_{i-1} \rightarrow l'_i \mid \\ &\quad l' \text{ can be right to } l,\ l'' \text{ is the parent label of } l, l' \text{ in } d(l'')\}, \end{aligned}$$

and D_i is the same as the previous case. L_i (resp., R_i) is the set of edges corresponding to "$\rightarrow^+:: d \rightarrow \leftarrow^+:: l$" (resp., "$\rightarrow^+:: d \rightarrow \rightarrow^+:: l$") in Fig. 8.

Finally, we define the cost of an edge in $G(q, G(D)) = (V', E')$. Suppose that $\gamma(l \rightarrow l')$, $\gamma(ax \rightarrow ax')$, $\gamma(\epsilon \rightarrow ax :: l)$, and $\gamma(ax :: l \rightarrow \epsilon)$ are defined for any $l, l' \in \Sigma_e$ and any axes ax, ax'. Then the cost of an edge $e \in E'$, denoted $\gamma(e)$, is defined as follows.

- The case where $e \in E_{insc}$: We can denote $e = l_i \rightarrow l'_i$. Since this edge represents inserting a location step $\downarrow:: l'$, $\gamma(e) = \gamma(\epsilon \rightarrow\downarrow:: l')$.
- The case where $e \in E'_i$: We can denote $e = l_i \rightarrow l'_i$. Since this edge represents inserting a location step $\downarrow^*:: l'$, $\gamma(e) = \gamma(\epsilon \rightarrow\downarrow^*:: l')$.
- The case where $e \in D_i$: We can denote $e = l_{i-1} \rightarrow l_i$. Since this edge represents deleting a location step $ax[i] :: l[i]$, $\gamma(e) = \gamma(ax[i] :: l[i] \rightarrow \epsilon)$.
- The case where $e \in C_i$: We can denote $e = l_{i-1} \rightarrow l'_i$. Since this edge represents substituting $ax[i]$ with $\downarrow$ and substituting $l[i]$ with l', $\gamma(e) = \gamma(ax[i] \rightarrow\downarrow) + \gamma(l[i] \rightarrow l')$.
- The case where $e \in A_i$: We can denote $e = l_{i-1} \rightarrow l'_i$. Since this edge represents substituting $ax[i]$ with $\downarrow^*$ and substituting $l[i]$ with l', $\gamma(e) = \gamma(ax[i] \rightarrow\downarrow^*) + \gamma(l[i] \rightarrow l')$.
- The case where $e \in L_i$: We can denote $e = l_{i-1} \rightarrow l'_i$. Since this edge represents substituting $ax[i]$ with $\leftarrow^+$ and substituting $l[i]$ with l', $\gamma(e) = \gamma(ax[i] \rightarrow\leftarrow^+) + \gamma(l[i] \rightarrow l')$. The case where $e \in R_i$ can be defined similarly.

For example, assume that $\gamma(ax \rightarrow ax') = 0$ if $ax = ax'$, $\gamma(l \rightarrow l') = 0$ if $l = l'$, and that $\gamma(op) = 1$ for any other edit operation op. Then for the path $p = n_0 \rightarrow s_0 \dashrightarrow a_1 \dashrightarrow d_2$ in Fig. 5, we have $\gamma(p) = \gamma(\epsilon \rightarrow\downarrow:: s) + (\gamma(\downarrow\rightarrow\downarrow) + \gamma(a \rightarrow a)) + (\gamma(\downarrow\rightarrow\downarrow) + \gamma(d \rightarrow d)) = 1 + 0 + 0 = 1$.

6. Algorithm for Finding top-K Queries

In this section, we present an algorithm for finding top-K queries syntactically close to an input query under a DTD. We first consider the case where a query is simple, then present an algorithm for queries in XP.

6.1 Algorithm for Simple Query

Let D be a DTD, Σ_e be the set of labels in D, $q = /ax[1] :: l[1]/\cdots/ax[m] :: l[m]$ be a simple query, and $G(q, G(D)) = (V', E')$ be the xd-graph for q and $G(D)$. Moreover, let $n_0 \in V'$ be the start node and $(l[m])_m \in V'$ be the accepting node of $G(q, G(D))$. If $l[m] \notin \Sigma_e$ (due to user's typo), then the label $l \in \Sigma_e$ "most similar" to $l[m]$ is selected and $l_m \in V'$ is used as the accepting node[*1]. Currently, we select $l \in \Sigma_e$ such that the edit distance between l and $l[m]$ is the smallest.

By the definition of xd-graph, in order to find top-K queries syntactically close to q under D, it suffices to solve the K shortest paths problem over the xd-graph $G(q, G(D))$ between the start

[*1] $G(q, G(D))$ can also have multiple accepting nodes by adding a new "accepting" node n and edges from each node in V_m to n. But since this approach tends to output "too diverse" answers, we currently use a single accepting node.

node and the accepting node. The resulting K shortest paths represent the top-K queries syntactically close to q under D. Formally, this algorithm can be described as follows.

Input: A DTD $D = (d, \alpha, s)$, a simple query $q = /ax[1] :: l[1]/\cdots/ax[m] :: l[m]$, and a positive integer K.

Output: Top-K queries syntactically close to q under D.

(1) Construct the DTD graph $G(D)$ of D.

(2) Construct the xd-graph $G(q, G(D))$ for q and $G(D)$.

(3) Solve the K shortest paths problem [*2] on $G(q, G(D))$ between the start node and the accepting node.

(4) Let $q_1, \cdots, q_K$ be the queries represented by the K paths obtained above. Return $q_1, \cdots, q_K$.

Let us give a simple example. We use query q and DTD D in Fig. 2, thus we have $q = /\downarrow::a/\downarrow::e/\downarrow::c$ and $D = (d, \alpha, a)$, where $d(s) = bd$, $d(a) = c$, $d(b) = c$, $d(c) = \epsilon$. Let $K = 2$. For simplicity, we only consider child axis (the other axes are omitted), and suppose that the cost of each edit operation is one except that $\gamma(l \to l') = 0$ whenever $l = l'$. In line 1 of the algorithm, we obtain the DTD graph $D(G)$ shown in Fig. 2 (c). In line 2, we obtain the xd-graph shown in Fig. 2 (d), where n_0 is the start node and c_3 is the accepting node. Now, in step 3 we solve the K shortest paths problem on the xd-graph and obtain the following two shortest paths.

- $n_0 \dashrightarrow a_1 \dashrightarrow b_2 \dashrightarrow c_3$. The second edge $a_1 \dashrightarrow b_2$ represents substituting the label of the second location step $\downarrow::e$ of q with b, while the other edges do nothing (substituting a label with the same one). Thus we have $/\downarrow::a/\downarrow::b/\downarrow::c$.
- $n_0 \dashrightarrow a_1 \dashrightarrow d_2 \dashrightarrow c_3$. The second edge $a_1 \dashrightarrow d_2$ represents substituting the label of the second location step $\downarrow::e$ of q with d, while the other edges do nothing. Thus we have $/\downarrow::a/\downarrow::d/\downarrow::c$.

The above two queries are returned in line 4.

We have the following.

Theorem 3 Let D be a DTD, q be a simple query, and K be a positive integer. Then the above algorithm outputs top-K queries syntactically close to q under D.

Proof(sketch) Let $q = /ax[1] :: l[1]/\cdots/ax[m] :: l[m]$ be a simple XPath query and D be a DTD. It suffices to show that the xd-graph $G(q, G(D))$ of q and D is "sound" (every path from the start node to the accepting node corresponds to a valid query) and "complete" (every valid query obtained by some edit script to q is represented by a path from the start node to the accepting node in the xd-graph). Let q_k be the prefix of q of length k, that is, $q_k = /ax[1] :: l[1]/\cdots/ax[k] :: l[k]$. In particular, $q_0 = \epsilon$. In the following, we show by induction on $|q|$ that for any node l_k in $G(q, G(D))$, the following two statements hold.

- (Soundness) Any path from the start node to l_k represents a valid query retrieving element l obtained by applying an edit script to q_k.
- (Completeness) Any valid query retrieving l obtained by applying an edit script to q_k is represented by a path from the start node to l_k.

Basis: $|q| = 0$ and $q = \epsilon$. Thus $G(q, G(D))$ contains only one DTD graph (G_1 in **Fig. 9**). Since $q = \epsilon$, only location step insertions to q are allowed. Thus it is easy to verify that for each node l_0 in $G(q, G(D))$, the above two statements hold.

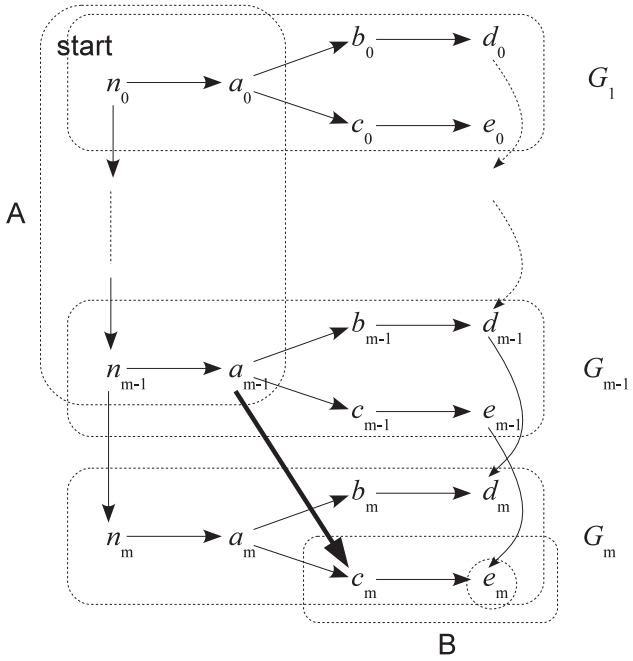

Fig. 9 Xd-graph.

Induction: Assume as the induction hypothesis that if $|q| < m$, then for any node l_k in $G(q, G(D))$, the above two statements hold. Consider the case of $|q| = m$. Since the soundness is rather clear, we only consider the completeness. Consider an edge between G_{m-1} and G_m, say $a_{m-1} \to c_m$ (see Fig. 9). We have the following observations.

- By the induction hypothesis, any valid query retrieving a obtained by applying an edit script to q_{m-1} is represented by a path from the start node to a_{m-1}, that is, the subgraph A of Fig. 9 is complete.
- By definition, the edges between G_{m-1} and G_m cover all the edit operations to the mth location step of q.
- Assuming that c_m is the "document root," we can show similarly to the basis case that any valid query p retrieving e (the label of the accepting node) obtained by applying location step insertions to ϵ is represented by a path from c_m to e_m. That is, the subgraph B of Fig. 9 is complete.

The above three observations imply that any valid query retrieving l obtained by applying an edit script to q is represented by a path from the start node to l_m, where l_m is the accepting node. Hence $G(q, G(D))$ is complete. □

Let us consider the time complexity of this algorithm. First, we consider the size of $G(q, G(D))$. For every node n in $G(p, G(D))$, the number of edges leaving n is in $O(|\Sigma_e|)$. Since the number of nodes in $G(q, G(D))$ is in $O(|q| \cdot |\Sigma_e|)$, the total number of edges in $G(q, G(D))$ is in $O(|q| \cdot |\Sigma_e|^2)$. Let us next consider solving the K shortest paths problem on $G(q, G(D))$. There are a number of algorithms for solving this problem (e.g., Refs. [6], [15]), and we use the extended Dijkstra's algorithm. The time complexity of Dijkstra's algorithm is $O(K \cdot |E| \cdot \log |V|)$, where E is the set of edges and V is the set of nodes. Since the number of edges in

[*2] There are a number of algorithms for solving K shortest paths problem (e.g., Refs. [6], [15]). Here we can use any of them.

the xd-graph is in $O(|q| \cdot |\Sigma_e|^2)$ and that of nodes is in $O(|q| \cdot |\Sigma_e|)$, the time complexity for solving the K shortest paths problem over the xd-graph is in $O(K \cdot |q| \cdot |\Sigma_e|^2 \cdot \log(|q| \cdot |\Sigma_e|))$. This is the time complexity of the algorithm.

Thus we have the following.

Theorem 4 Let D be a DTD, Σ be the set of labels in D, q be a simple XPath query, and K be a positive integer. Then top-K queries syntactically close to q under D can be obtained in $O(K \cdot |q| \cdot |\Sigma|^2 \cdot \log(|q| \cdot |\Sigma|))$ time. □

6.2 Algorithm for Queries in XP

The algorithm proposed in the previous subsection can handle only simple queries. In this subsection, we extend the algorithm so that it handles any queries in XP.

We present an algorithm that finds, for a query $q \in$ XP and a DTD D, top-K queries syntactically close to q under D. We first give some definitions. Let $q = /ax[1]::l[1][exp[1]]/\cdots/ax[m]::l[m][exp[m]] \in$ XP. By $sp(q)$ we mean the *selection path* of q obtained by dropping every predicate in q and the last location step of q if $ax[m] = @$; that is,

$$sp(q) = \begin{cases} /ax[1]::l[1]/\cdots/ax[m-1]::l[m-1] & \text{if } ax[m] = @, \\ /ax[1]::l[1]/\cdots/ax[m]::l[m] & \text{otherwise.} \end{cases}$$

Suppose that $ax[m] = @$. By definition the set of edit operations applicable to $ax[m]::l[m]$ is $S = \{ax[m]::l[m] \to \epsilon\} \cup \{l[m] \to l \mid l \in \alpha(l[m-1])\}$. We say that $op_1, \cdots, op_K$ are K optimum edit operations for $ax[m]::l[m]$ if $op_1, \cdots, op_K \in S$, $op_i \neq op_j$ for any $i \neq j$, $\gamma(op_1) \leq \cdots \leq \gamma(op_K)$, and $\gamma(op_K) \leq op$ for any $op \in S \setminus \{op_1, \cdots, op_K\}$ (We assume that $op_{|S|+1} = \cdots = op_K = nil$ with $\gamma(nil) = \infty$ if $|S| < K$).

We now present the algorithm. To find top-K queries syntactically close to a query q under a DTD D, we again construct an xd-graph $G(sp(q), G(D))$ and solve the K shortest paths problem on the xd-graph. But since q may not be simple, before solving the K shortest paths problem we modify $G(sp(q), G(D))$ as follows [*3].

- Suppose $exp[i] \neq \epsilon$. The cost of deleting location step $ax[i]::l[i][exp[i]]$ should be $\gamma(ax[i]::l[i] \to \epsilon) + \gamma(exp[i] \to \epsilon)$, where "$exp[i] \to \epsilon$" stands for the delete operations that delete every location step in $exp[i]$ (line (3-a) below).
  We also have to consider correcting $exp[i]$. To do this, we call the algorithm for query $/l[i]/exp[i]$ and DTD $(d, \alpha, l[i])$ recursively. The obtained result is incorporated into $G(sp(q), G(D))$ by using the gadget in **Fig. 10** (node l_i corresponds to $l[m]$); the obtained K optimum edit scripts are assigned to the K edges $e_1, \cdots, e_K$ in the gadget (line (3-b)).
- If $ax[m] = @$, we have to modify $G(sp(q), G(D))$ in order to incorporate the K optimum edit operations for $ax[m]::l[m]$ (line 4).

FindKPaths(D, q, K)

Input: A DTD $D = (d, \alpha, s)$, a query $q = /ax[1]::l[1][exp[1]]/\cdots/ax[m]::l[m][exp[m]]$, and a positive integer K.

[*3] Since it is fairly difficult to correct the right hand side and the comparison operator of $exp[i]$ exactly, we focus on correcting the left hand side of $exp[i]$.

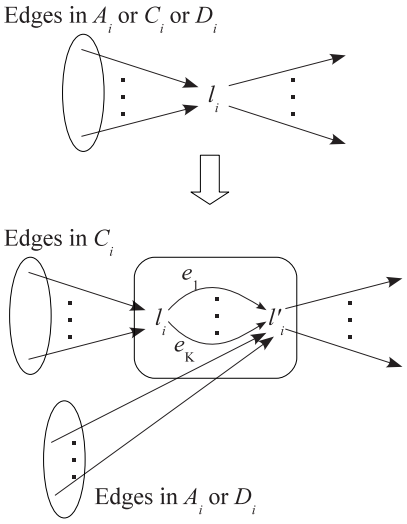

Fig. 10 Node l_i and its gadget, where l'_i is a new node and $e_1, \cdots, e_K$ are new edges.

Output: Top-K queries syntactically close to q under D.

(1) Construct the DTD graph $G(D)$ of D.
(2) Construct the xd-graph $G(sp(q), G(D))$ for q and $G(D)$.
(3) For each $1 \leq i \leq m$ with $exp[i] \neq \epsilon$, modify $G(sp(q), G(D))$ as follows.
  (a) For each edge $e \in D_i$ (defined in Eq. (3)), let $\gamma(e) \leftarrow \gamma(e) + \gamma(exp[i] \to \epsilon)$.
  (b) For each node $l_i \in V_i$, do the following (i) – (iii).
    (i) Replace l_i with its corresponding gadget (Fig. 10).
    (ii) Call FindKPaths(D', q', K), where $D' = (d, \alpha, l_i)$ and $q' = /l_i/exp[i]$ [*4]. Let $s'_1, \cdots, s'_K$ be the result.
    (iii) $\gamma(e_j) \leftarrow \gamma(s'_j)$ for every $1 \leq j \leq K$.
(4) If $ax[m] = @$, modify $G(sp(q), G(D))$ as follows.
  (a) Replace the accepting node l_{m-1} of $G(sp(q), G(D))$ with its corresponding gadget (Fig. 10).
  (b) Let $op_1, \cdots, op_K$ be the K optimum edit operations for $ax[m]::l[m]$.
  (c) $\gamma(e_j) \leftarrow \gamma(op_j)$ for every $1 \leq j \leq K$.
(5) Solve the K shortest paths problem on $G(sp(q), G(D))$ modified as above.
(6) Let $q_1, \cdots, q_K$ be the queries represented by the K paths obtained above. Return $q_1, \cdots, q_K$.

Let us explain the algorithm by an example. For simplicity, we assume that the cost of each edit operation is one except that $\gamma(l \to l') = 0$ whenever $l = l'$. We also assume that only child axes are allowed (the other axes are omitted). Let $K = 2$, $q = /\downarrow::a/\downarrow::b[\downarrow::e]/\downarrow::c$ be a query, and $D = (d, \alpha, a)$ be a DTD, where $d(a) = bd$, $d(b) = c$, $d(d) = ce$, and $d(c) = d(e) = \epsilon$. In step 1 of the algorithm, we obtain the DTD graph $G(D)$ shown in **Fig. 11**. In step 2, $sp(q) = /\downarrow::a/\downarrow::b/\downarrow::c$ and we obtain the xd-graph $G(sp(q), G(D))$ shown in **Fig. 12**, where n_0 is the start node and c_3 is the accepting node. In this xd-graph, the costs of four edges $n_0 \dashrightarrow a_1$, $a_1 \dashrightarrow b_2$, $b_2 \dashrightarrow c_3$, $d_2 \dashrightarrow c_3$ are zero (the edges labeled by "0" in Fig. 12), while the costs of the other edges

[*4] Since l_i is added as the first location step of q', for each recursive call we assume that $\gamma(n_0 \to l) = 0$ if $l = (l_i)_0$ and $\gamma(n_0 \to l) = \infty$ otherwise, where n_0 is the start node of the constructed xd-graph in the recursive call.

are one (their labels are omitted). In step 3, since the second location step $\downarrow::\ b[\downarrow::\ e]$ of q has a predicate, $G(sp(q), G(D))$ is modified by replacing five nodes a_2, b_2, c_2, d_2, e_2 with their corresponding gadgets, as shown in **Fig. 13**. For example, consider the gadget having two nodes b_2 and b'_2. This gadget has two edges $b_2 \xrightarrow{A} b'_2$ and $b_2 \xrightarrow{B} b'_2$, where the former represents substituting e with c in the predicate and the latter represents deleting the predicate of q. Note that, due to step (3-a), the costs of "vertical" edges $n_1 \dashrightarrow n_2$, $a_1 \dashrightarrow a'_2$, $b_1 \dashrightarrow b'_2$, $c_1 \dashrightarrow c'_2$, $d_1 \dashrightarrow d'_2$, and $e_1 \dashrightarrow e'_2$ are increased by one (the edges labeled by "2" in Fig. 13), since these edges now represent deleting $\downarrow::\ b[\downarrow::\ e]$ instead of deleting $\downarrow::\ b$. Over this modified graph, we solve the K shortest paths problem between n_0 and c_3 (step 5). The followings are the three shortest paths whose costs are one.

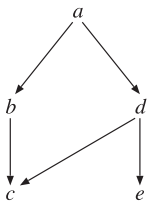

Fig. 11 DTD graph $G(D)$.

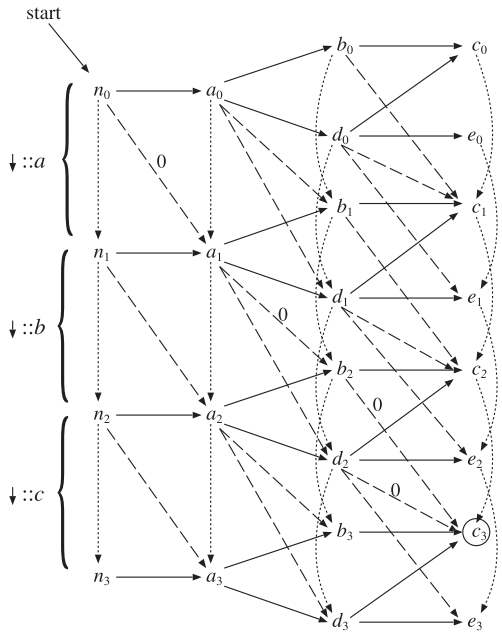

Fig. 12 Xd-graph $G(sp(q), G(D))$.

- $n_0 \dashrightarrow a_1 \dashrightarrow b_2 \xrightarrow{A} b'_2 \dashrightarrow c_3$. The third edge $b_2 \xrightarrow{A} b'_2$ represents substituting e with c in the predicate of q, while the other edges do nothing (substituting a label with the same one). Thus we obtain $/\downarrow::\ a/\downarrow::\ b[\downarrow::\ c]/\downarrow::\ c$.
- $n_0 \dashrightarrow a_1 \dashrightarrow b_2 \xrightarrow{B} b'_2 \dashrightarrow c_3$. The third edge $b_2 \xrightarrow{B} b'_2$ represents deleting the predicate of q, while the other edges do nothing. Thus we obtain $/\downarrow::\ a/\downarrow::\ b/\downarrow::\ c$.
- $n_0 \dashrightarrow a_1 \dashrightarrow d_2 \xrightarrow{A} d'_2 \dashrightarrow c_3$. The second edge $a_1 \dashrightarrow d_2$ represents substituting b with d in the second location step $\downarrow::\ b[\downarrow::\ e]$, while the other edges do nothing. Thus we obtain $/\downarrow::\ a/\downarrow::\ d[\downarrow::\ e]/\downarrow::\ c$.

Since $K = 2$, arbitrary two of the above three are returned in step 6 (ties are broken arbitrary).

We have the following.

Theorem 5 Let D be a DTD, $q \in$ XP a query, and K be a positive integer. Then FindKPaths outputs top-K queries syntactically close to q under D.

Proof(sketch) Let $q = /ax[1] :: l[1][exp[1]]/\cdots/ax[m] :: l[m][exp[m]]$. We show the completeness of the graph obtained in lines (1) to (4) of the algorithm. Every update to $sp(q)$ is covered by $G(sp(q), G(D))$ by Theorem 4. Thus we have to consider (a) updates to the predicates in q and (b) update to the attributes in q. Consider first (a). Consider a location step $ax[i] :: l[i][exp[i]]$ of q. Due to the definition of update operations, the possible updates to this location step are as follows.

(1) The whole location step is deleted.

(2) This location step is not deleted. In this case, $l[i]$ is replaced

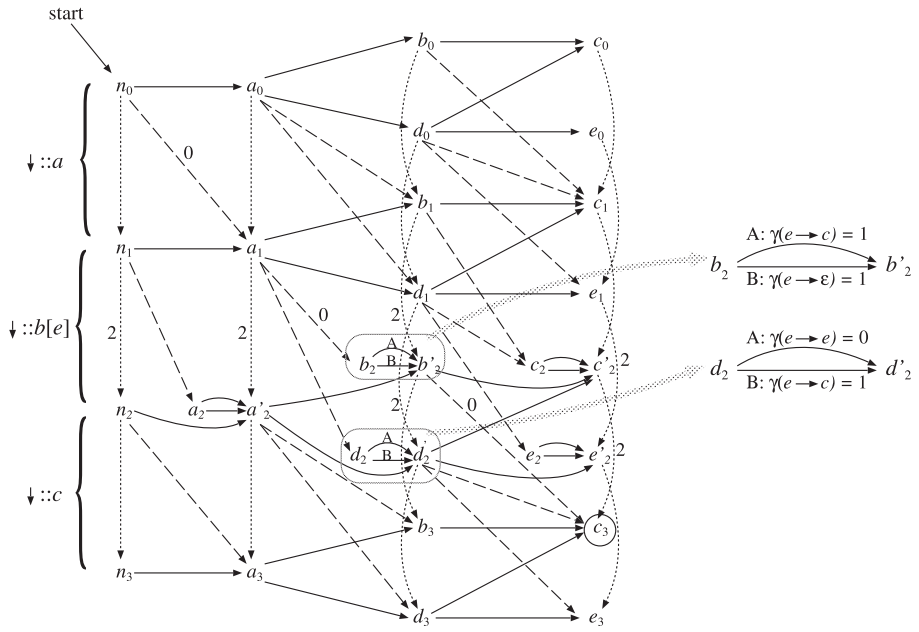

Fig. 13 The graph obtained by modifying $G(sp(q), G(D))$.

by some label and $exp[i]$ is updated by some update script. (1) is covered by line (3-a) and (2) is covered by line (3-b) of the algorithm. As for (b), the possible updates to the attributes are covered by line (4). Thus the graph obtained in line (3) is complete. □

Let us consider the running time of the algorithm. Let $q = /ax[1] :: l[1][exp[1]]/\cdots/ax[m] :: l[m][exp[m]]$. Let $mnl(q)$ be the *maximum nest level* of q, that is,

$$mnl(q) = \begin{cases} 0 & \text{if } q \text{ is simple,} \\ 1 + \max_{1\le i\le m}(mnl(exp[i])) & \text{otherwise.} \end{cases}$$

For example, if $q = /\downarrow::a/\downarrow::b[\downarrow::d[\downarrow::e]]/\downarrow::c$, then $mnl(q) = 2$. First, consider the case where $mnl(q) = 1$, i.e., no $exp[i]$ has a predicate. In this case, since FindKPaths is called $|V_i| = |\Sigma_e|$ times (step (3-b)), by Theorem 4 the algorithm runs in $O(K \cdot |sp(q)| \cdot |\Sigma_e|^2 \cdot \log(|sp(q)| \cdot |\Sigma_e|) + \sum_{1\le i\le m} |\Sigma_e| \cdot (K \cdot |exp[i]| \cdot |\Sigma_e|^2 \cdot \log(|exp[i]| \cdot |\Sigma_e|))) = O(K \cdot |q| \cdot |\Sigma_e|^3 \cdot \log(|q| \cdot |\Sigma_e|))$ time. In general, due to step (3) the running time of the algorithm is increased by a factor of $|\Sigma_e|$ as $mnl(q)$ increases by one. Thus, the algorithm runs in $O(K \cdot |q| \cdot |\Sigma_e|^{2+mnl(q)} \cdot \log(|q| \cdot |\Sigma_e|))$ time. This suggests that the algorithm may run inefficiently if q has deeply nested predicates. However, XPath queries usually contains very few such predicates, and as we will see below, by pruning unnecessary edges and nodes of xd-graphs the algorithm can run more efficiently. Therefore, we believe that the algorithm runs efficiently for most of XPath queries.

Pruning Xd-Graph

An xd-graph may contain unnecessary nodes, e.g., in Fig. 5 the accepting node d_2 is unreachable from c_0, c_1, and c_2, and thus these three nodes are unnecessary. By pruning such nodes, we can save space and time. Such a pruning is effective especially if a DTD has a tree-like structure. For example, suppose that the DTD graph $D(G)$ is a complete k-ary tree and that query q contains no sibling axis and no predicate. For a leaf node n in $D(G)$, the number of nodes from which n is reachable is in $O(\log|\Sigma_e|)$. Thus the size of the xd-graph can be reduced from $O(|q| \cdot |\Sigma_e|^2)$ to $O(|q| \cdot \log^2|\Sigma_e|)$, and the time complexity of the algorithm in this subsection can be reduced to $O(K \cdot |q| \cdot \log^{2+mnl(q)}|\Sigma_e| \cdot \log(|q| \cdot \log|\Sigma_e|))$.

On the other hand, pruning needs a top-down traversal from the start node and a bottom-up traversal from the accepting node. This can be done in $O(|\Sigma_e|)$.

We also make an experiment to evaluate the effect of this pruning. This is shown in Section 7.2.

7. Experimental Results

In this section, we present two experimental results. The first experiment evaluates the "quality" of the output of the algorithm, and the second experiment evaluates the effect of pruning xd-graph.

The algorithm is implemented in Ruby, and the experiments are performed on Apple Xserve with Mac OS X Server 10.6.8, Xeon 2.26 GHz CPU, 6GB Memory, and Ruby-1.9.3. Our current implementation covers the queries in XP except the ones using attribute axis (@) or the comparison operators ("Op" in Table 1).

Table 2 XPath queries (correct queries).

1.	/site/closed_auctions/closed_auction/annotation/description/text/keyword
2.	//closed_auction//keyword
3.	/site/closed_auctions/closed_auction//keyword
4.	/site/closed_auctions/closed_auction[annotation/description/text/keyword]/date
5.	/site/closed_auctions/closed_auction[descendant::keyword]/date
6.	/site/open_auctions/open_auction/bidder[following-sibling::bidder]
7.	/site/open_auctions/open_auction/bidder[preceding-sibling::bidder]

In the following, we use the shorthand notations for child and descendant-or-self axes, i.e., "↓::" is omitted and "//" is used instead of "/↓*::."

7.1 Quality of the output of the Algorithm

For a DTD D and an incorrect query q written by a user, there are a number of queries similar to q under D, and thus our algorithm need to output a result containing the "correct query" that the user requires. We evaluate the ratio at which the results of the algorithm contain the correct queries.

The outline of this experiment is as follows. We first prepare a set of pairs (q_c, q_i), where q_c is a correct query (a query a user should write) and q_i is an incorrect query (a query a user actually writes). Then for each pair (q_c, q_i), we execute the algorithm to obtain top-K queries syntactically close to q_i and check weather the top-K queries contain q_c.

Let us give the details of the experiment. The experiment is achieved by the following five steps.

(1) The DTD used in this experiment is auction.dtd of XMark [19], which is a recursive schema. As for XPath queries, we use XPath queries of XPathMark [9]. These queries have a natural interpretation over documents generated with XMark. Therefore they simulate realistic query needs of a potential user of the the auction site. Among the queries of XPathMark, we choose seven queries that can be handled by the current implementation of our algorithm (**Table 2**). They are treated as "correct queries" q_c.

(2) XPathMark also purveys a query in natural language and a condition for each XPath query. For example, for the XPath query

```
//closed_auction//keyword,
```

the corresponding query and condition (enclosed in curly brackets) are as follows.

```
Keywords in annotations of closed auctions
{descendant}
```

This condition means "only descendant axis is available." These are called "questions."

(3) We request seven people to solve the 7 questions obtained in step 2. That is, for each question they are asked to write an XPath query that coincides with what the question means. In this step they can see auction.dtd at any time. We obtain $7 \times 7 = 49$ answers (i.e., queries written by users) in total.

(4) We check the above 49 queries by hand and find 20 incorrect ones as shown in **Table 3**. Now we obtain 20 pairs (q_c, q_i) of correct queries and incorrect queries such that q_c and q_i share the same question.

(5) For each incorrect query q_i of the 20 pairs (q_c, q_i) and each $K = 1, \cdots, 10$, we execute the algorithm for q_i under auction.dtd and check whether the corresponding correct query

Table 3 Incorrect queries written by users.

1.	/closed_auctions/closed_auction/annotation/description/text/keyword
2.	/site/closed_auctions/closed_auction/annotation/keyword
3.	/closed_auction/annotation/keyword
4.	/closed_auctions/closed_auction/annotation/keyword
5.	/closed_auctions/closed_auction/annotation/description//keyword
6.	/site//closed_auction/annotation/keyword
7.	//closed_auction/annotation/keyword
8.	/closed_auctions/closed_auction//keyword
9.	/open_auction/following-sibling::bidder
10.	/site/open_auctions/open_auction/following-sibling::bidder
11.	/open_auctions/open_auction/bidder/following-sibling::bidder
12.	/site/open_auctions/open_auction/following-sibling::bidder
13.	/open_auction/bidder/following-sibling::bidder
14.	/open_auction/preceding-sibling::bidder
15.	/site/open_auctions/open_auction/preceding-sibling::bidder
16.	/open_auctions/open_auction/bidder/preceding-sibling::bidder
17.	/site/open_auctions/open_auction/preceding-sibling::bidder
18.	/open_auction/bidder/preceding-sibling::bidder
19.	/site/open_auctions/bdder/following-sibling::bidder
20.	/site/open_auctions/bdder/preceding-sibling::bidder

q_c is contained in the output of the algorithm. We use the following simple cost function. This is determined in an ad-hoc manner for no particular reason.

$\gamma(l \to l')$ = the normalized string edit distance [16] between l and l',

$$\gamma(ax \to ax') = \begin{cases} 0 & \text{if } ax = ax', \\ 2 & \text{otherwise,} \end{cases}$$

$\gamma(\epsilon \to ax :: l) = 1$,

$\gamma(ax :: l \to \epsilon) = 2$.

Figure 14 illustrates the result. As shown in the figure, the algorithm fairly succeeds in generating top-K queries containing correct queries.

However, the ratio does not reach 100% due to the three incorrect queries 5, 6, and 7 in Table 3. Since the cost of location step deletion is set to be larger than that of location step insertion, incorrect queries containing redundant location steps tend not to be contained in the result of the algorithm. More concretely, one of the incorrect query is the following,

```
/closed_auctions/closed_auction/annotation/descr-
iption//keyword
```

and the corresponding correct query is as follows. The algorithm does not predict it since it needs to delete two location steps /annotation and /description (and to insert one location step /site).

```
/site/closed_auctions/closed_auction//keyword
```

In this experiment, we use a simple ad-hoc cost function and we might obtain a more better result if we use a more sophisticated cost function. This is an important future work.

7.2 Execution Time of the Algorithm

Since the size of an xd-graph may become very large, pruning of xd-graph is important to obtain top-K queries efficiently. We evaluate the execution time of the algorithm, as follows.

(1) Pruning of xd-graph becomes more effective as the accepting node is near to the start node. We divide the queries shown in Table 3 into two sets. First set Q_1 contains queries 1–8 whose target element (accepting node) is "keyword", which is far from the start node (the distance between the start node and the accepting node is 6 in the DTD graph of auction.dtd). Second set Q_2 contains queries 9–20 whose target element is "bidder", which is near from the start node (the distance between the start node and the accepting node is 3).

(2) For each set Q_1 and Q_2 and each $K = 1, \cdots, 10$, we execute the algorithm with the same cost function of the previous experiment and measure its execution time.

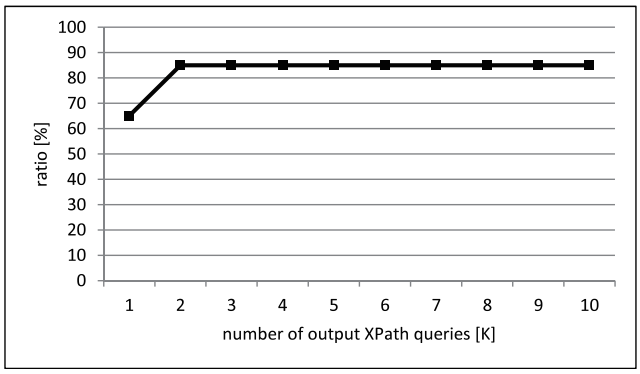

Fig. 14 Ratios at which the outputs contain correct answers.

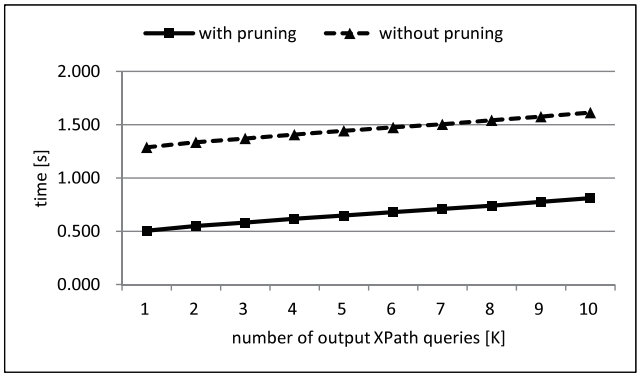

Fig. 15 Execution time with/without pruning of the algorithm for queries targeting "far" nodes.

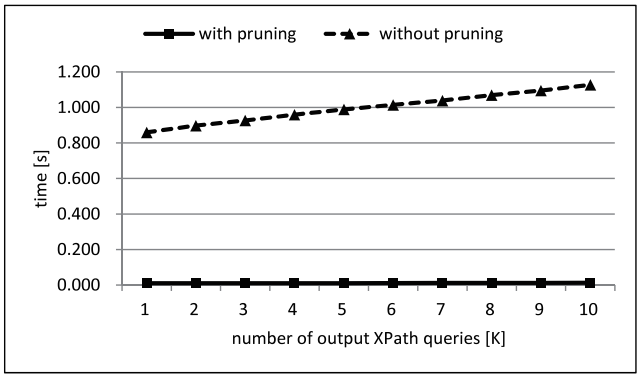

Fig. 16 Execution time with/without pruning of the algorithm for queries targeting "near" nodes.

Figure 15 plots the average execution times for Q_1, and **Fig. 16** for Q_2, with/without pruning. With pruning the average execution time for Q_1 is about 0.51 to 0.81 seconds, while without pruning execution requires about twice the time in the average. On the other hand, with pruning the average execution time for Q_2 is about 10 milliseconds, while without pruning the average execu-

tion time is increased by a factor of 85 to 113.

Thus, with pruning the algorithm runs efficiently and the pruning brings a much reduction of the execution time of the algorithm especially for queries targeting near nodes.

8. Conclusion

In this paper, we proposed an algorithm that finds, for a query q, a DTD D, and a positive integer K, top-K queries syntactically close to q under D. Experimental results suggest that the algorithm outputs "correct" answers efficiently in many cases.

As a future work, we should devise a method for determining reasonable costs of edit operations automatically, since it may be difficult for users to specify the cost of each edit operation exactly. Possibly, slack costs cause a localized solution. Therefore they need to be determined carefully.

Acknowledgments The authors are grateful to the reviewers' valuable comments that improved this paper.

References

[1] ALTOVA: XMLSpy, available from ⟨http://www.altova.com/jp/xmlspy.html⟩.
[2] Amer-Yahia, S., Cho, S. and Srivastava, D.: Tree Pattern Relaxation, *Proc. EDBT*, pp.89–102 (2002).
[3] Amer-Yahia, S., Lakshmanan, L.V. and Pandit, S.: FleXPath: Flexible structure and full-text querying for XML, *Proc. SIGMOD*, pp.83–94 (2004).
[4] Choi, B.: What are real DTDs like?, *Proc. WebDB*, pp.43–48 (2002).
[5] Cohen, S. and Brodianskiy, T.: Correcting queries for XML, *Inf. Syst.*, Vol.34, No.8, pp.690–710 (2009).
[6] Eppstein, D.: Finding the k shortest paths, *SIAM J. Computing*, Vol.28, No.2, pp.652–673 (1998).
[7] Fazzinga, B., Flesca, S. and Furfaro, F.: XPath query relaxation through rewriting rules, *IEEE Trans. Knowledge and Data Engineering*, Vol.23, pp.1583–1600 (2011).
[8] Fazzinga, B., Flesca, S. and Pugliese, A.: Retrieving XML data from heterogeneous sources through vague querying, *ACM Trans. Internet Technol.*, Vol.9, No.2 (2009).
[9] Franceschet, M.: XPathMark: An XPath Benchmark for the XMark Generated Data, *Proc. XSym'05*, pp.129–143 (2005).
[10] Ikeda, K. and Suzuki, N.: Finding top-K Correct XPath Queries of User's Incorrect XPath Query, *Proc. DEXA'12*, pp.116–130 (2012).
[11] Ives, Z.G., Halevy, A.Y. and Weld, D.S.: An XML query engine for network-bound data, *The VLDB Journal*, Vol.11, No.4, pp.380–402 (2002).
[12] Li, G., Feng, J., Wang, J. and Zhou, L.: Effective keyword search for valuable lcas over XML documents, *Proc. ACM CIKM*, pp.31–40 (2007).
[13] Li, Y., Yu, C. and Jagadish, H.V.: Schema-Free XQuery, *Proc. VLDB*, pp.72–83 (2004).
[14] Li, Y., Yu, C. and Jagadish, H.V.: Enabling Schema-Free XQuery with meaningful query focus, *The VLDB Journal*, Vol.17, pp.355–377 (2008).
[15] Martins, E.: K-th Shortest Paths Problem, available from ⟨http://www.mat.uc.pt/˜eqvm/OPP/KSPP/KSPP.html⟩.
[16] Marzal, A. and Vidal, E.: Computation of Normalized Edit Distance and Applications, *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol.15, pp.926–932 (1993).
[17] Morishima, A., Kitagawa, H. and Matsumoto, A.: A machine learning approach to rapid development of XML mapping queries, *Proc. ICDE*, pp.276–287 (2004).
[18] Schenkel, R. and Theobald, M.: Feedback-Driven Structural Query Expansion for Ranked Retrieval of XML Data, *Proc. EDBT*, pp.331–348 (2006).
[19] Schmidt, A., Waas, F., Kersten, M., Carey, M.J., Manolescu, I. and Busse, R.: XMark: A Benchmark for XML Data Managemet, *Proc. VLDB*, pp.974–985 (2002).
[20] Termehchy, A. and Winslett, M.: Using structural information in XML keyword search effectively., *ACM Trans. Database Syst.*, Vol.36, No.1, p.4 (2011).
[21] Xu, Y. and Papakonstantinou, Y.: Efficient keyword search for smallest LCAs in XML databases, *Proc. ACM SIGMOD*, pp.527–538 (2005).

Kosetsu Ikeda received his bachelor's degree in library and information science from University of Tsukuba in 2011, and his M.E. degree in information science from Tsukuba University in 2013. He has been a Ph.D. student of Graduate School of Library, Information and Media Studies, University of Tsukuba. His current research interests are XML query processing and Web data management.

Nobutaka Suzuki received his bachelor's degree in information and computer sciences from Osaka University in 1993, and his M.E. and Ph.D. degrees in information science from Nara Institute of Science and Technology in 1995 and 1998, respectively. He was with Okayama Prefectural University as a Research Associate in 1998–2004. In 2004, he joined University of Tsukuba as an Assistant Professor. Since 2009, he has been an Associate Professor of Graduate School of Library, Information and Media Studies, University of Tsukuba. His current research interests include database theory and structured documents.

(Editor in Charge: *Hiroshi Sakamoto*)