An Algorithm for Finding
K Correct XPath Expressions

IKEDA KOSETSU

Graduate School of Library, Information and Media Studies

University of Tsukuba

March 2013

Contents

Chapter 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 D exactly, especially in the case where D is a very large and/or complex DTD 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 does not conform to D or the answer of q is disappointing due to his/her structural misunderstanding of D. 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 D. 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 "similar"(syntactically close) to q among the XPath queries conforming to D, in order 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, emailaddress, phone?)>
<!ELEMENT name         (#PCDATA)>
<!ELEMENT emailaddress (#PCDATA)>
<!ELEMENT phone        (#PCDATA)>
<!ATTLIST person id ID #REQUIRED>
```

Suppose that a user wants `name` element in the person whose id is "11517" and that he/she tries to use an XPath query $q =$ `/person[@id = "11517"]/naem`, which does not conform to D. Our algorithm finds XPath queries similar to q based on the *edit distance* between XPath queries, introduced in this paper. In this example, our algorithm lists the following top-K XPath queries similar 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' [14].

```
1. //person[@id = "11517"]/name              (0.75)
2. //people/person[@id = "11517"]/name       (1.75)
3. /site/people/person[@id = "11517"]/name   (2.25)
```

As above, by our algorithm the user can obtain top-K correct XPath queries similar 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 to write correct XPath queries on such DTDs.

In this paper, we focus on an XPath fragment using child, descendant-or-self, following-sibling, preceding-sibling, and attribute axes. Although our XPath fragment supports no upward axes, this gives usually no problem since the majority of XPath queries uses only downward axes[9]. Thus, we believe that our algorithm is useful to correct a large number of XPath queries.

There have been a number of eminent studies related to this paper. Ref. [5] proposes an algorithm that finds valid tree pattern queries most similar 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 supporting multiple type for element name (as in XML Schema), while we use DTD (recursion is supported) and a tree is ordered. 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. Ref. [17] proposes the node insertion operation that is also proposed in this paper. Ref. [16] takes a query expansion approach instead of correcting queries. Refs. [2, 3, 8, 7] 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. [11, 12], in which a user can write an XQuery query without specifying exact connections between elements. Ref. [15] proposes an interactive system for generating XQuery queries. There has been a number of studies on XML keyword search (e.g., [20, 10, 19]), which are especially suitable for users that are not familiar with XML query languages. Moreover, several XML editors (e.g., XMLSpy [1]) support autocomplete for XPath query editing, but they do not support listing K correct XPath queries.

The rest of this paper is organized as follows. Chapter 2 gives some preliminary definitions. Chapter 3 defines four edit operations to XPath queries. Chapter 4 illustrates the DTD graph and the xd-graph that are used in our algorithm. Chapter 5 shows an algorithm for finding K correct XPath queries most similar to input XPath query, and then shows an extension to the algorithm so that the algorithm handle XPath predicates. Chapter 6 shows some experimental results. Chapter 7 summarizes the paper.

Chapter 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 Chapter 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 $\mathit{str} \in L(d(a))$ such that $\mathit{str}[i] = c$ and $\mathit{str}[j] = b$ with $i < j$ ($i > j$), then we say that b can be *right* (resp., *left*) to c in $d(a)$, where $\mathit{str}[i]$ 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.

The XPath fragment used in this paper, denoted XP, is a set of location paths using child ($\downarrow$), descendant-or-self ($\downarrow^*$), following-sibling ($\rightarrow^+$), preceding-sibling ($\leftarrow^+$), and attribute (@) axes.

Table 2.1 Syntax of XP

XP	::=	"/" RelativePath | "/" RelativePath "@" Attribute
RelativePath	::=	LocationStep | LocationStep "/" RelativePath
LocationStep	::=	Axis "::" Nodetest | Axis "::" Nodetest Predicate
Axis	::=	"$\downarrow$" | "$\downarrow^*$" | "$\rightarrow^+$" | "$\leftarrow^+$"
Nodetest	::=	Label | "$*$"
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 '"') '"'

Formally, XP is defined in Table 2.1. Thus an *XPath query* (*query* for short) q in XP can be denoted

$$/ax[1] :: l[1][exp[1]]/\cdots/ax[m] :: l[m][exp[m]], \tag{2.1}$$

where $ax[i] \in$ Axis and $l[i] \in \Sigma_e$ for $1 \leq i \leq m-1$, $exp[i] \in$ Exp for $1 \leq i \leq m$, $ax[m] \in \text{Axis} \cup \{@\}$, 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 (2.1) containing no '$*$' as node test. 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 *conforms* to 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 $2 \leq i \leq m$
  - $ax[i] = \downarrow$ 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 $1 \leq i \leq m$ with $exp[i] \neq \epsilon$, query $/\downarrow :: l[i]/exp[i]$ conforms to DTD $(d, \alpha, l[i])$.

Let q be a query in (2.1) containing '$*$'s as node tests. Then q *conforms* to D if for some $l_1 \in L(l[1]), \cdots, l_m \in L(l[m])$, $/ax[1] :: l_1[exp[1]]/\cdots/ax[m] :: l_m[exp[m]]$ conforms to D.[*1] By $|q|$ we mean the number of location steps in q, e.g., if $q = /\downarrow :: a/\downarrow :: *[\leftarrow^+ :: d]$, then $|q| = 3$. If a query q has neither predicate nor attribute axis, then we say that q is *simple*.

[*1] $L(l[i]) = \{l[i]\}$ if $l[i]$ is a label, $L(l[i]) = \Sigma_e$ if $l[i] =$ '$*$'.

Chapter 3

Edit Operations to XPath Query

In this chapter, we define edit operations to queries. We use the following four kinds of *edit operations.*

- *Axis substitution:* substitutes axis ax with ax', denoted $ax \to ax'$. For example, by applying $\downarrow \to \downarrow^*$ to $/\downarrow :: a$ we obtain $/\downarrow^* :: a$.
- *Label substitution:* substitutes label (possibly '$*$') l with l', denoted $l \to l'$. For example, by applying $a \to b$ to $/\downarrow :: a$ we obtain $/\downarrow :: b$.
- *Location step insertion:* inserts location step $ax :: l$, denoted $\epsilon \to ax :: l$. For example, by applying $\epsilon \to \downarrow :: b$ to the tail of $/\downarrow :: a$ we obtain $/\downarrow :: a/\downarrow :: b$.
- *Location step deletion:* deletes location step $ax :: l$, denoted $ax :: l \to \epsilon$. For example, by applying $\downarrow :: a \to \epsilon$ to the first location step of $/\downarrow :: a/\downarrow :: b$ we obtain $/\downarrow :: b$.

We next define the *position* of a location step ls, denoted $pos(ls)$. Let $q = /ax[1] :: l[1][exp[1]]/\cdots/ax[m] :: l[m][exp[m]] \in \text{XP}$. We define that $pos(ax[i] :: l[i]) = i$ for $1 \leq i \leq m$. 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 $1 \leq j \leq n$. The position of a location step in $exp'[j]$ can be defined similarly. For example, let $q = /\downarrow :: a/\downarrow :: b[\downarrow :: *[\downarrow :: g]]/\to^+ :: c$. Then $pos(\downarrow :: b) = 2$, $pos(\downarrow :: *) = 2.1$, 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 $q \in \text{XP}$. 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 $s = [\epsilon \to \downarrow :: b]_1\,[c \to f]_3$ 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 $U = \{\downarrow, \downarrow^*\}$, $S = \{\to^+, \leftarrow^+\}$, 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 $\gamma(op) \geq 0$. 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, $\gamma(s) = \sum_{1 \geq i \geq n} \gamma(op_i)$. 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 $s_1(q), \cdots, s_K(q)$ conforms to D, (ii) $\gamma(s_1) \leq \cdots \leq \gamma(s_K)$, and (iii) $s_1, \cdots, s_K$ are optimum, that is, for any edit script s for q such that $s(q)$ conforms to D, $s(q) \in \{s_1(q), \cdots, s_K(q)\}$ or $\gamma(s) \geq \gamma(s_K)$. We say that $s_1(q), \cdots, s_K(q)$ are *top-K* queries *similar* to q under D.

Chapter 4

Xd-Graph Representing Queries Conforming to DTD

In this chapter, we introduce a graph called *xd-graph*, which forms the basis of our algorithm. Throughout this section, we assume a simple query.

To find top-K queries similar to a query q under a DTD D, we take the following approach.

1. We first construct an xd-graph for q and D. The graph is designed so that each path in the graph represents a simple query q' such that (a) q' is obtained by applying some edit script to q and that (b) q' conforms to D.
2. Then we solve the K shortest paths problem on the xd-graph. The result corresponds to top-K queries similar to q under D. The details of this step are presented in Chapter 5.

4.1 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 $E = \{l \to l' \mid l' \text{ is a label appearing in } d(l)\}$. For example, Fig. 4.1 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 (assuming that no '$*$' can be used), 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 ($\to^+$, $\leftarrow^+$) can be used as well as $\downarrow$ and $\downarrow^*$.

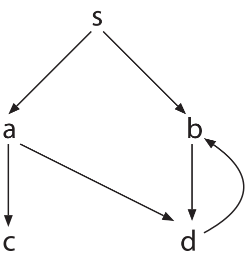

Figure 4.1 a DTD graph $G(D)$

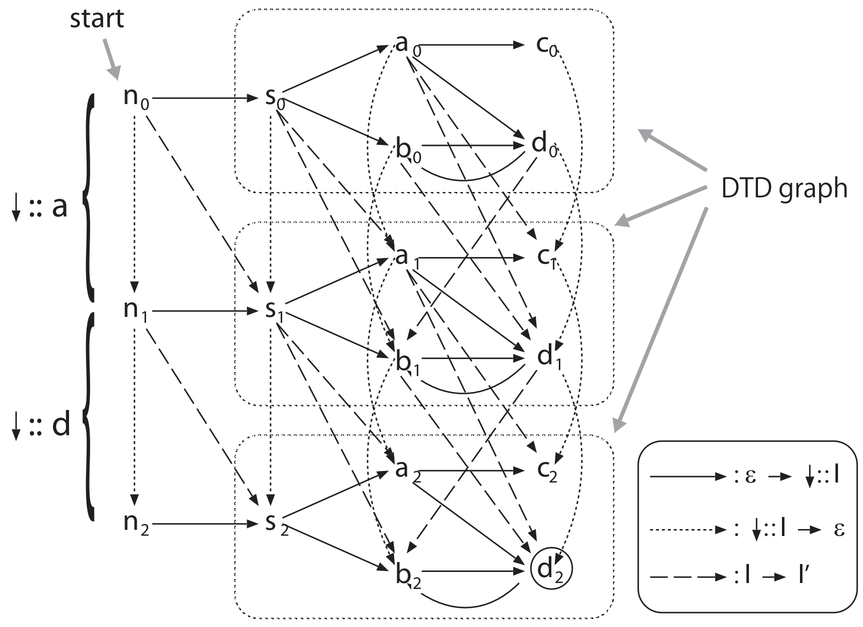

Figure 4.2 an xd-graph $G(q, G(D))$

Case A)

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. 4.1. Since only $\downarrow$ axis is allowed, it suffices to consider location step insertion, location step deletion, and label substitution. Fig. 4.2 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. 4.2.

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

- A "horizontal" edge $l \rightarrow l'$ 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.

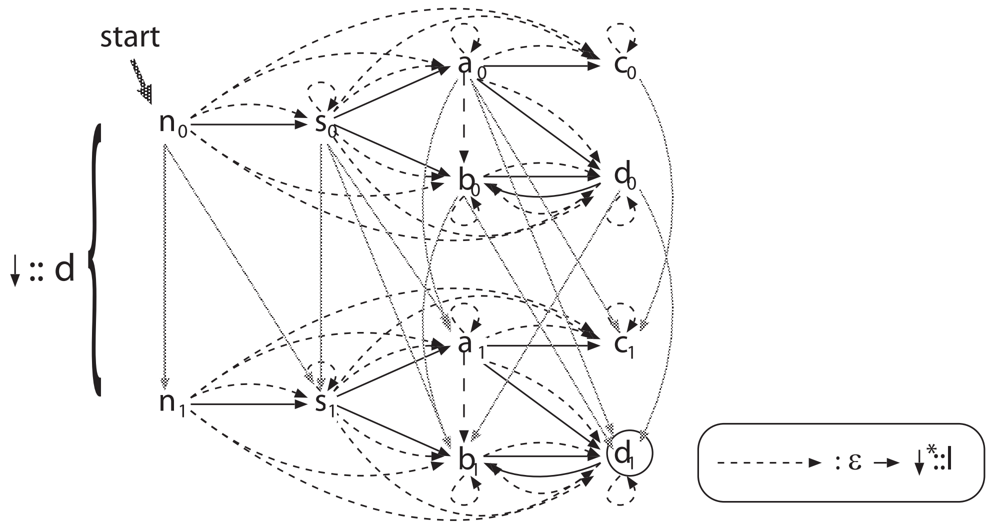

Figure 4.3 Edges representing location step insertion

More concretely, let us first consider horizontal edge $n_0 \to s_0$ in Fig. 4.2. 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. 4.2. 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. 4.2. 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. 4.2, 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 conforming to 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. 4.2. 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' conforms to D.

Case B)

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. For this insertion, we add edges representing the edit operation to an xd-graph. Fig. 4.3 shows the xd-graph constructed from the DTD graph in Fig. 4.1 and a query $q = /\downarrow :: d$. Each dashed edge in Fig. 4.3 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 conforming to D, which is obtained by

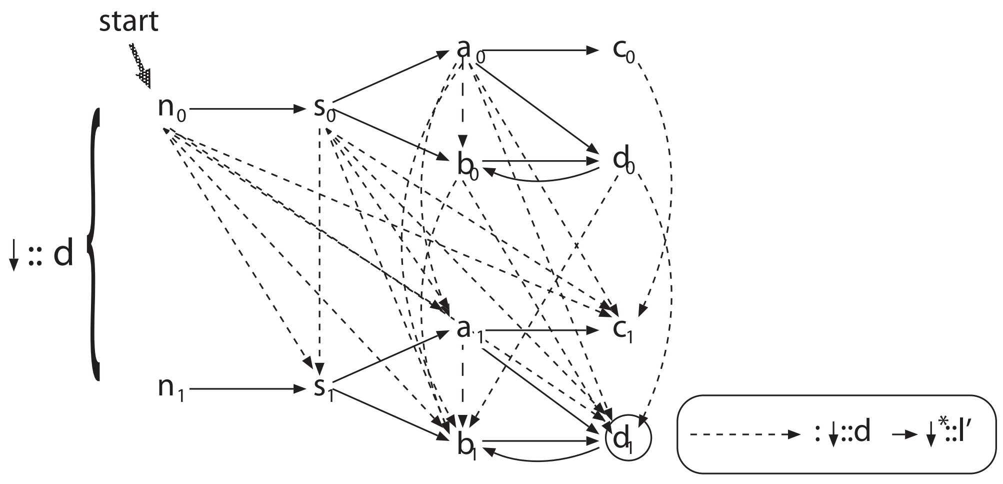

Figure 4.4 Edges representing axis substitution

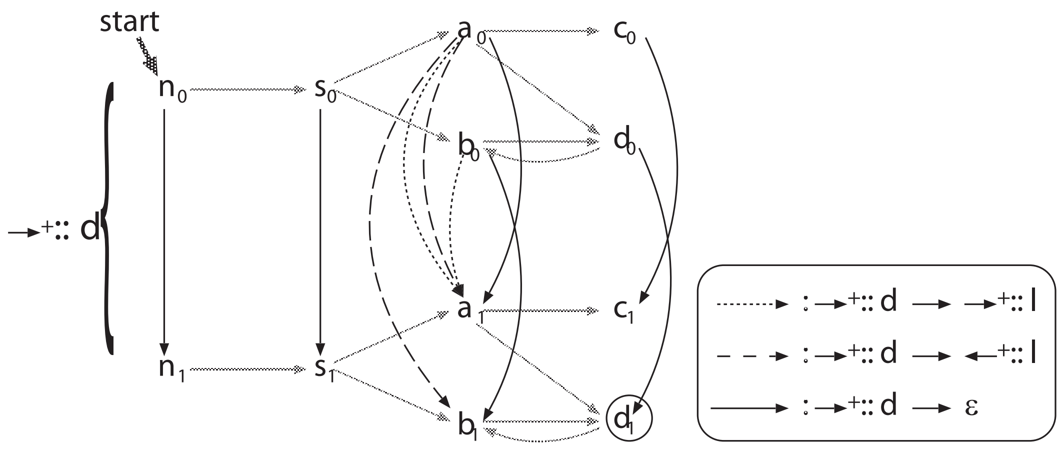

Figure 4.5 Edges dealing with $\rightarrow^{+}$ and $\leftarrow^{+}$ axes

correcting q. For example, $n_0 \rightarrow s_1 \dashrightarrow d_1$ represents a simple query $/\downarrow:: s/\downarrow^{*}:: d$ obtained by applying $[d \rightarrow s]_1[\epsilon \rightarrow \downarrow^{*}:: d]_1$ to $q = /\downarrow:: d$.

Let us next consider axis substitution between $\downarrow$ and $\downarrow^{*}$. Fig. 4.4 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. 4.4, 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 $p = n_0 \rightarrow s_0 \dashrightarrow d_1$ in Fig. 4.4. p represents a query $/\downarrow:: s/\downarrow^{*}:: d$, which is obtained by applying $[\epsilon \rightarrow \downarrow:: s]_0[\downarrow \rightarrow \downarrow^{*}]_1$ to $q = /\downarrow:: d$.

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

Case C)

Let us consider handling $\rightarrow^+$ and $\leftarrow^+$ axes. Fig. 4.5 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., $s_0 \rightarrow s_1$ and $a_0 \rightarrow a_1$. 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". 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. 4.5 does not have edge $b_0 \dashrightarrow a_1$ since $d(s) = ba^*$ and a cannot be left to b.

Wildcard Node Test)

To handle wild card node test '$*$', we duplicate each "slant" edge in Fig. 4.2. For example, between s_0 and a_1 we use two edges $s_0 \dashrightarrow a_1$ and $s_0 \overset{*}{\dashrightarrow} a_1$ instead of a single edge $s_0 \dashrightarrow a_1$. The former of the two represents substituting a label with a as in Case A, and the latter represents substituting a label with '$*$' rather than a. Similarly, each dashed edge in Fig. 4.4 is duplicated.

4.2 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 $0 \leq i \leq m$. The *xd-graph* for q and $G(D)$, denoted $G(q, G(D))$, is a directed graph (V', E'), where

$$\begin{aligned} V' &= \{n_0, \cdots, n_m\} \cup V_0 \cup \cdots \cup V_m, \\ E' &= E_{insc} \cup (E'_0 \cup \cdots \cup E'_m) \cup (F_1 \cup \cdots \cup F_m). \end{aligned} \tag{4.1}$$

Here, E_{insc} in (4.1) is the set of edges inserting $\downarrow :: l$ (correspond to "$\epsilon \rightarrow \downarrow :: l$" in Fig. 4.2), 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 (4.1) is the set of edges inserting $\downarrow^* :: l$ (corresponding to "$\epsilon \rightarrow \downarrow^* :: l$" in Fig. 4.3) and define 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 (4.1) 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 $ax[i] \in \{\downarrow, \downarrow^*\}$: $F_i = D_i \cup C_i \cup C_i^* \cup A_i \cup A_i^*$, where

$$\begin{aligned}
D_i &= \{n_{i-1} \to n_i\} \cup \{l_{i-1} \to l_i \mid l \in V\}, \\
C_i &= \{n_{i-1} \to s_i\} \cup \{l_{i-1} \to l_i' \mid l \to l' \in E\}, \\
C_i^* &= \{n_{i-1} \xrightarrow{*} s_i\} \cup \{l_{i-1} \xrightarrow{*} l_i' \mid l \to l' \in E\}, \\
A_i &= \{n_{i-1} \to l_i \mid l_i \in V_i\} \cup \{l_{i-1} \to l_i' \mid l' \text{ is reachable from } l \text{ in } D\}, \\
A_i^* &= \{n_{i-1} \xrightarrow{*} l_i \mid l_i \in V_i\} \cup \{l_{i-1} \xrightarrow{*} l_i' \mid l' \text{ is reachable from } l \text{ in } D\}.
\end{aligned} \tag{4.2}$$

Here, D_i is the set of edges corresponding to "$\downarrow:: l \to \epsilon$" in Fig. 4.2, C_i is the set of edges corresponding to "$l \to l'$" in Fig. 4.2, and A_i is the set of edges corresponding to "$\downarrow:: d \to \downarrow^*:: l$" in Fig 4.4. C_i^* (A_i^*) is the set of "duplicated" edges of C_i (resp., A_i) to handle '$*$'.

2) The case where $ax[i] \in \{\leftarrow^+, \to^+\}$: $F_i = D_i \cup L_i \cup R_i$, where

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

and D_i is the same as the previous case. L_i (resp., R_i) is the set of edges corresponding to "$\to^+:: d \to \ \leftarrow^+:: l$" (resp., "$\to^+:: d \to \ \to^+:: l$") in Fig. 4.5.

Finally, we define the cost of an edge in $G(q, G(D)) = (V', E')$. Suppose that $\gamma(l \to l')$, $\gamma(ax \to ax')$, $\gamma(\epsilon \to ax :: l)$, and $\gamma(ax :: l \to \epsilon)$ 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 $e = l_i \to l_i'$. Since this edge represents inserting a location step $\downarrow:: l'$, $\gamma(e) = \gamma(\epsilon \to \downarrow:: l')$.
- The case where $e \in E_i'$: We can denote $e = l_i \to l_i'$. Since this edge represents inserting a location step $\downarrow^*:: l'$, $\gamma(e) = \gamma(\epsilon \to \downarrow^*:: l')$.
- The case where $e \in D_i$: We can denote $e = l_{i-1} \to l_i$. Since this edge represents deleting a location step $ax[i] :: l[i]$, $\gamma(e) = \gamma(ax[i] :: l[i] \to \epsilon)$.
- The case where $e \in C_i$: We can denote $e = l_{i-1} \to l_i'$. Since this edge represents substituting $ax[i]$ with $\downarrow$ and substituting $l[i]$ with l', $\gamma(e) = \gamma(ax[i] \to \downarrow) + \gamma(l[i] \to l')$. The case where $e \in C_i^*$ can be defined similarly.
- The case where $e \in A_i$: We can denote $e = l_{i-1} \to l_i'$. Since this edge represents substituting $ax[i]$ with $\downarrow^*$ and substituting $l[i]$ with l', $\gamma(e) = \gamma(ax[i] \to \downarrow^*) + \gamma(l[i] \to l')$. The case where $e \in A_i^*$ can be defined similarly.
- The case where $e \in L_i$: We can denote $e = l_{i-1} \to l_i'$. Since this edge represents substituting $ax[i]$ with $\leftarrow^+$ and substituting $l[i]$ with l', $\gamma(e) = \gamma(ax[i] \to \leftarrow^+) + \gamma(l[i] \to l')$. The case where $e \in R_i$ can be defined similarly.

For example, assume that $\gamma(ax \to ax') = 0$ if $ax = ax'$, $\gamma(l \to l') = 0$ if $l = l'$, and that $\gamma(op) = 1$ for any other edit operation op. Then for the path $p = n_0 \to s_0 \dashrightarrow a_1 \dashrightarrow d_2$ in Fig. 4.2, we have $\gamma(p) = \gamma(\epsilon \to \downarrow:: s) + (\gamma(\downarrow \to \downarrow) + \gamma(a \to a)) + (\gamma(\downarrow \to \downarrow) + \gamma(d \to d)) = 1 + 0 + 0 = 1$.

Chapter 5

Algorithm for Finding top-K Queries

In this section, we present an algorithm for finding top-K queries similar 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.

5.1 Method 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 similar to q under D, it suffices to solve the K shortest paths problem over the xd-graph $G(q, G(D))$ between the start node and the accepting node. The resulting K shortest paths represent the top-K queries similar to q under D. Thus we have the following.

Theorem 1 Let D be a DTD, q be a simple query, and K be a positive integer. Then the above method outputs top-K queries similar to q under D. $\square$

Proof(sketch) Let p be a simple XPath query and D be a DTD. It suffices to show that the xd-graph $G(p, G(D))$ of p and D is "sound" and "complete". For the soundness, it is easy to show that every path p from the start node to the accepting node in $G(p, G(D))$ is "correct", that is, the XPath query represented by p is correct w.r.t. D. Let me next consider the completeness. Consider inserting an additional edge $l_i \to l_j$ such that there is no edge between l_i and l_j in $G(p, G(D))$. Then it is easy to verify that $l_i \to l_j$ does not correctly represent any edit operation to p. $\square$

[*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, I currently use a single accepting node.

Let us consider the time complexity of this method. 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., [13, 6]), and we use the extended Dijkstra's algorithm. The time complexity of the 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 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 method.

Thus we have the following.

Theorem 2 Let D be a DTD, Σ be the set of labels in D, p be a simple XPath query, and K be a positive integer. Then K optimum correct edit scripts for p under D can be obtained in $O(K \cdot |p| \cdot |\Sigma|^2 \cdot \log(|p| \cdot |\Sigma|))$ time.

5.2 Algorithm for General Query

We present an algorithm that finds, for a query $q \in$ XP and a DTD D, top-K queries similar 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 $op \in S \setminus \{op_1, \cdots, op_K\}$ (I 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 similar 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.[*2]

- 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).

[*2] 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 $exp[i]$.

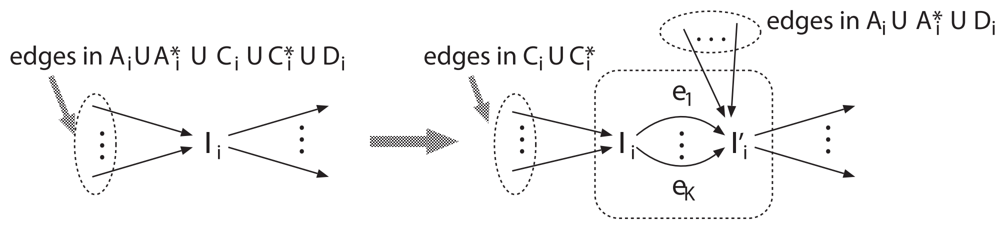
edges in A_i U A*_i U C_i U C*_i U D_i
l_i
edges in C_i U C*_i
edges in A_i U A*_i U D_i
e_1
e_K
l_i
l'_i

Figure 5.1 Node l_i and its gadget, where l'_i is a new node and $e_1, \cdots, e_K$ are new edges.

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. 5.1 (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 $ax[m] = @$, I 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 $K$.
Output: $K$ optimum edit scripts $s_1, \cdots, s_K$ for $q$ 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. (4.2)), 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. 5.1).
         ii. Call FindKPaths($D', q', K$), where $D' = (d, \alpha, l_i)$ and $q' = /l_i/exp[i]$.[*3] 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. 5.1).
     (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. Delete the nodes unreachable from the accepting node in $G(sp(q), G(D))$.
  6. Solve the $K$ shortest paths problem on $G(sp(q), G(D))$ modified as above.
  7. Let $s_1, \cdots, s_K$ be the result of line 6. Return $s_1, \cdots, s_K$.
```

The above algorithm runs in $O(K \cdot |q| \cdot |\Sigma_e|^2 \cdot \log(|q| \cdot |\Sigma_e|))$ time. I also have the following.

Theorem 3 Let D be a DTD, $q \in$ XP a query, and K be a positive integer. Then the algorithm outputs K optimum edit scripts for q under D. □

[*3] Since l_i is added as the first location step of q', for each recursive call I 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.

Pruning Xd-Graph

An xd-graph may contain unnecessary nodes, e.g., in Fig. 4.2 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 $O(K \cdot |q| \cdot \log^2|\Sigma_e| \cdot \log(|q| \cdot \log|\Sigma_e|))$.

On the other hand, pruning needs a top-down traverse from the start node and a bottom-up traverse 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 Chapter 6.1.

Chapter 6

Experimental Results

In this chapter, we present two experimental results. The first experiment evaluates the execution time, and the second experiment evaluates the "quality" of the output of the algorithm. The algorithm is implemented in Ruby, and the experiments are performed on Apple Xserve with Mac OS X Server 10.6.8, Xeon 2.26GHz CPU, 6GB Memory, and Ruby-1.9.3.

6.1 Running 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. We create a set Q of 10 queries shown in Table 6.1. These queries are generated by XQGen [21], which is an XPath expression generator, under auction.dtd of XMark [18]. The average size of the queries in Q is 4.1. Two of the queries contains predicates and the others are simple.
2. For each query obtained above and for each $K = 1, \cdots, 10$, we execute the algorithm and measure its execution time. In this experiment, we use the following simple cost function.

$$\begin{aligned} \gamma(l \to l') &= \text{ the normalized string edit distance between } l \text{ and } l' \\ \gamma(ax \to ax') &= \begin{cases} 0 & \text{if } ax = ax' \\ 0.5 & \text{otherwise} \end{cases} \\ \gamma(\epsilon \to ax :: l) &= \gamma(ax :: l \to \epsilon) = 1 \end{aligned}$$

Fig. 6.3 plots the average execution times for Q, with/without pruning. With pruning the average execution time for Q is about 30 to 250 milliseconds, but without pruning the average execution time is increased by a factor of 4 to 100. Thus, with pruning the algorithm runs efficiently and the pruning brings a huge reduction of the execution time of the algorithm.

Table 6.1 Generated queries by XQGen for evaluating execution time

1.	//listitem/parlist
2.	//closed_auctions/closed_auction/happiness
3.	//text/parlist/listitem/listitem
4.	//closed_auctions/buyer
5.	//categories/description
6.	//date/mail
7.	//regions/namerica/shipping
8.	/site/closed_auction/closed_auctions/price
8.	//africa/item[description/parlist/text]
10.	/site/regions/africa/item[mailbox/mail]

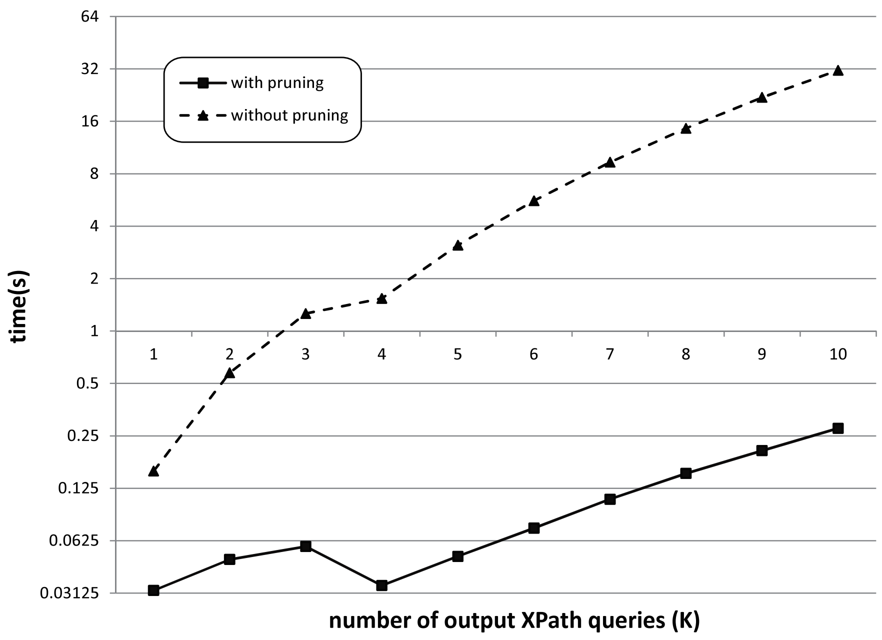

Figure 6.1 Execution time with/without pruning of the algorithm

6.2 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

Write a minimum XPath query q satisfying the following two conditions.

1. The target element of q is "start".
2. q must use an "interval" element.

Figure 6.2 An example of a question

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 similar to q_i and calculate the ratio at which the top-K queries contain q_c.

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

1. We generate 30 queries shown in Table 6.2 by using XQGen under auction.dtd of XMark. The average size of these queries is about 5.4. There queries are treated as "correct queries".
2. XQGen can generate XPath queries containing only $\downarrow$ and $\downarrow^*$ axes. Thus we choose randomly about 20% of the XPath queries obtained in 3) (in this case, four XPath queries), and we insert location steps using sibling axes to the chosen XPath queries. The following is an example.

   /s/a/c

   $\downarrow$

   /s/a/following-sibling::a/c

3. For each query q_c obtained above, We make a "question", which describes the meaning of q_c in words. Fig. 6.2 shows an example of a simple question for `//interval/start`. Each question is carefully described so that it does not permit more than one correct queries. We obtain 30 questions.
4. We request six people to solve the 30 questions obtained in step 2. That is, for each question they are asked to write a query whose semantics coincides with what the question means. In this step they can see auction.dtd at any time. We obtain 180 answers (i.e., queries written by users) in total.
5. We checked the 180 queries by hand and find 17 incorrect ones. Now we obtain 17 pairs (q_c, q_i) of correct queries and incorrect queries such that q_c and q_i share the same question.
6. For each query q_i of the 17 incorrect queries and each $K = 1, \cdots, 10$, we execute the algorithm for q_i and check whether the corresponding correct query q_c is contained in the output of the algorithm. We use the same cost function as the previous experiment. Fig. 6.3 illustrates the result.

As shown in the figure, the algorithm fairly succeeds in generating top-K queries containing correct queries. The reason why the ratio does not reach 100% is as follows. Auction.dtd contains a cycle and a correct query traverses the cycle, but a user write an incorrect query that "skips" the intermediate elements on the cycle and the algorithm cannot predict the correct query since too much elements

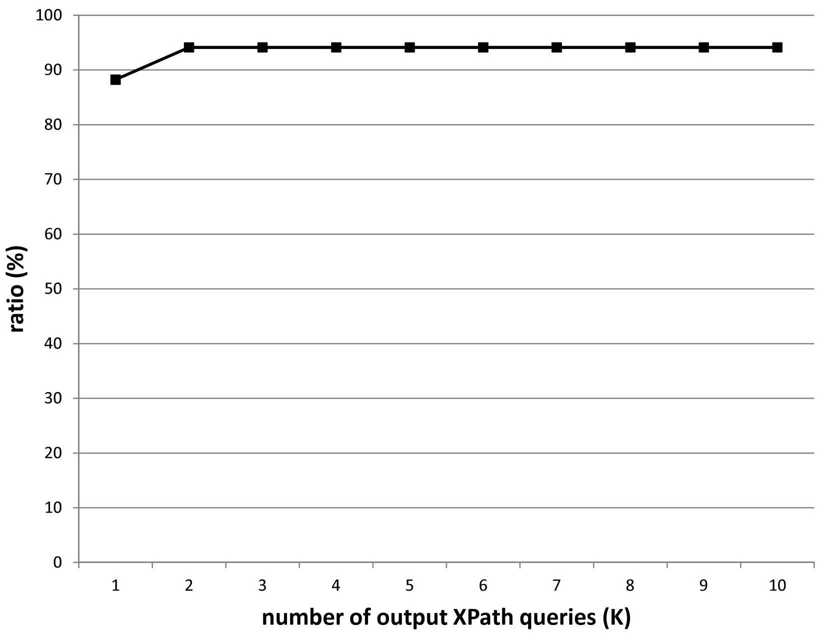

Figure 6.3 The Ratio at which the outputs of the algorithm contains correct answers

are skipped. More concretely, the query written by a user is the following,

```
//closed_auctions/closed_auction/annotation/description/text
```

and the corresponding correct query is as follows. The algorithm does not predict it since four elements are skipped.

```
//closed_autcions/closed_auction/annotation
      /description/parlist/listitem/parlist/listitem/text
```

Table 6.2 XPath queries of “right answers”

1.	//interval/start
2.	//listitem/text/preceding-sibling::text
3.	//annotation/description/parlist/listitem/parlist/listitem
4.	//closed_auction/annotation/description/parlist
5.	//category/description/parlist/listitem/parlist/listitem/text/emph
6.	/site/open_auctions/open_auction/annotation/description/text/bold
7.	//regions/asia/item/mailbox/mail/from
8.	//item/description/parlist/listitem/text/emph/keyword
9.	/site/regions/africa/item/following-sibling::item/mailbox/mail/to
10.	/site/regions/africa/item/quantity
11.	/site/regions/asia/item/mailbox
12.	//africa/item/mailbox/mail/following-sibling::mail
13.	//open_auction/annotation/description/text/emph
14.	//closed_auctions/closed_auction/annotation/description/parlist/listitem/parlist/listitem/text
15.	//person/phone
16.	//closed_auctions/closed_auction/annotation/author
17.	//regions/africa/item/mailbox
18.	//mailbox/mail/date
19.	//africa/item/name
20.	/site/regions/australia/item/description/parlist
21.	//europe/item/payment
22.	//watches/watch
23.	//regions/australia/item/description/parlist/listitem/parlist/listitem
24.	//open_auction/bidder/increase
25.	//regions/australia/item/shipping
26.	//description/parlist
27.	/site/regions/australia
28.	//samerica/item/description/text/emph
29.	//open_auctions/open_auction/interval/start
30.	//bidder/following-sibling::bidder/personref

Chapter 7

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 similar 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.

Another future work is to improve the algorithm completely, and evaluates more essentially. It is necessary to evaluate from three points of view. Firstly, a control experiment is needed against the related works. Secondly, the effect of handing following-sibling, preceding-sibling and attribute axes should be evaluated. Thirdly, "quality" of the output should be evaluated for various costs. It is suggested that a location step close to the root is more important than a location step far from the root. Therefore, the quality of the output can probably be improved by inclining costs in the former side.

Acknowledgment

The author would first like to express my sincere gratitude to warm encouragement and support of my adviser, Associate Professor Nobutaka Suzuki in pursuing this study. Completion of this thesis would be impossible without his help.

The author would especially like to thank Associate Professor Atsuyuki Morishima for his valuable suggestions and support.

The author is also grateful to Associate Professor Tetsuo Sakaguchi for his daily perceptive comments. His gratitude also goes to Professors Tsutsuji Satoh, Makoto Matsumoto, Hidehiko Hasegawa and Hiroshi Itsumura for their comments, suggestions and criticisms.

The author wishes to extend to thanks to my pleasant friends Tomomi Mitsuishi, Kazuma Hasegawa, Hiroki Mizumoto, Naoto Watanabe, Yui Yasunaga, Masai Takanari, Kazuma Horie, Rong Chen and other colleagues of the university with which he affiliate.

Last but not least, a special word of appreciation goes to my parents.

Bibliography

[1] ALTOVA. “XMLSpy”. (online), available from ⟨ http://www.altova.com/jp/xmlspy.html⟩ , (accessed 2013-01-10)

[2] Amer-Yahia, S., Cho, S., Srivastava, D.: Tree pattern relaxation. In: Proc. EDBT. pp. 89–102 (2002)

[3] Amer-Yahia, S., Lakshmanan, L.V., Pandit, S.: Flexpath: Flexible structure and full-text querying for xml. In: Proc. SIGMOD. pp. 83–94 (2004)

[4] Choi, B.: What are real dtds like? In: Proc. WebDB. pp. 43–48 (2002)

[5] Cohen, S., Brodianskiy, T.: Correcting queries for xml. Information Systems 34(8), 690–710 (2009)

[6] Eppstein, D.: Finding the k shortest paths. SIAM J. Computing 28(2), 652–673 (1998)

[7] Fazzinga, B., Flesca, S., Furfaro, F.: Xpath query relaxation through rewriting rules. IEEE Transactions on Knowledge and Data Engineering 23, 1583–1600 (2011)

[8] Fazzinga, B., Flesca, S., Pugliese, A.: Retrieving xml data from heterogeneous sources through vague querying. ACM Trans. Internet Technol. 9(2), 7:1–7:35 (May 2009), `http://doi.acm.org/10.1145/1516539.1516542`

[9] Ives, Z.G., Halevy, A.Y., Weld, D.S.: An xml query engine for network-bound data. The VLDB Journal 11(4), 380–402 (2002)

[10] Li, G., Feng, J., Wang, J., Zhou, L.: Effective keyword search for valuable lcas over xml documents. In: Proc. ACM CIKM. pp. 31–40. CIKM ’07, ACM (2007)

[11] Li, Y., Yu, C., Jagadish, H.V.: Schema-free xquery. In: Proc. VLDB. pp. 72–83 (2004)

[12] Li, Y., Yu, C., Jagadish, H.V.: Enabling schema-free xquery with meaningful query focus. The VLDB Journal 17, 355–377 (May 2008)

[13] Martins, E.: K-th shortest paths problem, http://www.mat.uc.pt/ eqvm/OPP/KSPP/KSPP.html

[14] Marzal, A., Vidal, E.: Computation of normalized edit distance and applications. IEEE Transactions on Pattern Analysis and Machine Intelligence 15, 926–932 (1993)

[15] Morishima, A., Kitagawa, H., Matsumoto, A.: A machine learning approach to rapid development of xml mapping queries. In: Proc. ICDE. pp. 276–287 (2004)

[16] Schenkel, R., Theobald, M.: Feedback-driven structural query expansion for ranked retrieval of xml data. In: Proc. EDBT. pp. 331–348 (2006)

[17] Schlieder, T.: Schema-driven evaluation of approximate tree-pattern queries. In: Proceedings of the 8th International Conference on Extending Database Technology: Advances in Database Technology. pp. 514–532. EDBT '02, Springer-Verlag, London, UK, UK (2002), `http://dl.acm.org/citation.cfm?id=645340.650204`

[18] Schmidt, A., Waas, F., Kersten, M., Carey, M., Manolescu, I., Busse, R.: Xmark: A benchmark for xml data managemet. In: Proc. VLDB. pp. 974–985 (2002)

[19] Termehchy, A., Winslett, M.: Using structural information in xml keyword search effectively. ACM Trans. Database Syst. 36(1), 4 (2011)

[20] Xu, Y., Papakonstantinou, Y.: Efficient keyword search for smallest lcas in xml databases. In: Proc. ACM SIGMOD Conf. pp. 527–538. ACM (2005)

[21] Y.Wu, Lele, N., R.Aroskar, Chinnusamy, S., Brenes, S.: Xqgen: an algebra-based xpath query generator for micro-benchmarking. In: Proc. CIKM. pp. 2109–2110 (2009)