A Study on Algorithms for Finding Correct XPath Queries

Kosetsu IKEDA

Graduate School of Library, Information and Media Studies

University of Tsukuba

July 2016

XML 問合せ式における修正候補発見アルゴリズムに関する研究

要旨

記述した問合せ式が所望の結果を返さない場合，問合せ式を正しく修正する必要がある．このような場合に，正しい問合せ式の記述を支援する手法が利用できれば有用である．これまで，関係データベースにおいては，SQL 文の記述を支援するための手法が提案されてきた．一方，XML(Extensible Markup Language) は関係データベースと比べてかなり複雑な構造を有している．このため，XML において問合せ式の記述を支援するのはより困難であり，有効な手法はほとんど存在しないのが現状である．

問合せ式の記述を支援するためには，データ構造に関する正確な情報を把握する必要がある．そのような情報を得る方法として，データを走査してデータ構造に関する情報を収集する方法と，データの代わりにスキーマを参照する方法の 2 つが考えられる．しかし，前者にはいくつかの問題がある．まず，プライバシーやセキュリティ上の理由から，データの全体または一部を参照するのが困難な場合がある．次に，データのサイズはスキーマのそれより極めて大きいため，計算資源の限られた環境においては巨大なデータを処理することが困難な場合がある．そこで本論文では後者に着目し，スキーマから得られる構造情報に基づいて問合せ式の記述を支援することについて考察する．また，問合せ式の記述を支援する場合，問合せ式記述の際に支援を行う（要素名の補完など）方法と，正しくない問合せ式が記述された場合にその式を正しいものに修正する方法とが考えられる．これらの方法は互い

に対立するものではなく，併用することも可能である．とりわけ後者は，記述された問合せ式が所望の結果を返さない時や，スキーマが更新されて問合せ式が正しくないものとなった場合などに有効である．本論文では後者に着目し，利用者によって記述された XPath 式を正しいものに修正する問題について考察する．

データを参照せずスキーマに基づいて問合せ式の振る舞いを解析することを静的解析 (static analysis) といい，本論文で考察する問題も静的解析の 1 つである．ここで，XML における静的解析問題は多くの場合において計算困難であることが知られている．例えば，XPath 充足可能性問題 (XPath satisfiability problem) は XML において最も広く知られている静的解析問題のうちの 1 つである．この問題は一般には決定不能であることが示されており，child 軸と descendant-or-self 軸のみが許される場合においても NP 困難であることが分かっている．また，XPath 包含判定問題 (XPath containment problem) も XML においてよく知られている静的解析問題であるが，この問題も決定不能であることが知られている．本論文で考察する XPath 式修正問題についても，この問題が効率よく解けるか否かは自明でなく，アルゴリズム・計算複雑性の観点からこの問題の性質を究明することは重要な課題である．そこで本論文では，XPath 式修正問題において，スキーマや XPath 式に対する制約の強さの観点から，計算困難な部分と効率よく解ける部分を明らかにすることを主な目的とする．

本論文の主な貢献は以下の通りである．

1. これまで，XPath 式に対する修正を形式的に定義し議論した研究は存在しなかった．本論文では，XPath 式に対する編集操作の形式的な定義を与え，２つの編集操作のクラス core および extended を定める．これらの定義により，XPath 式修正問題の計算複雑性およびアルゴリズムに関する形式的な議論を可能としている．これに加えて，XPath 式の 2 つの部分クラス simple および XP を定義する．これら編集操作および XPath のクラスが，XPath 式修正問題の計算複雑さに影響を与える主要な要因であることを明らかにする．特に，編集操作として extended を許した

場合，XPath 式修正問題が計算困難であることを明らかにする．

2. 編集操作として core のみを用いると仮定した上で，XPath 式修正問題を解くためのアルゴリズムを提案する．このアルゴリズムに関する議論を通じて，XPath 式が simple なものである場合，XPath 式修正問題が多項式時間可解であることを明らかにする．さらに，XPath 式が XP に属する場合において，XPath 式修正問題が多項式時間可解であるための十分条件を明らかにする．

編集操作は XPath 式修正問題を形式化するために必要不可欠であり，この問題の計算複雑さにも影響を与える．本論文では，4 章において XPath に対する 6 種の編集操作を定義する．それは，(1) 軸の置換，(2) ラベルの置換，(3) ロケーションステップの挿入，(4) ロケーションステップの削除，(5) ロケーションステップの交換，(6) 述語の交換，である．ここで，(1) から (4) までの編集操作を core といい，(5) および (6) を extended という．これらの編集操作は，任意の XPath 式を他の任意の XPath 式に変換できるという意味で完全 (complete) であり，この完全性は編集操作を core なものに限定しても成り立つという性質を有する．この問題の計算複雑さに影響を与えるもう一つの要因は XPath 式である．本論文では，XPath 式の 2 つの部分クラス simple と XP を定義する．XP は，軸として child, descendant-or-self, following-sibling, preceding-sibling, attribute を用いた XPath 式からなる集合である．また，XP に属する XPath 式 q が述語も属性軸も使用しないものであるとき，q は simple であるという．これら定義に基づいて，XPath 式修正問題を「XPath 式 q と DTDD に対して，D の下で q に最も近い (編集距離の最も小さい)XPath 式を求める問題」として定義し，この問題の計算複雑さを明らかにする．まず，編集操作として core に加えて extended の使用が可能である場合，XPath 式を simple なものに限定したとしても XPath 式修正問題が NP 困難であることを，有向ハミルトン経路問題 (directed Hamiltonian path problem) からの帰着により示す. 次に，ロケーションステップの交換を用いないと仮定しても，述語の交換とラベルの置換を許した場合，述語に含まれるどのロケーションステッ

プも述語を含まないという制約が課された XPath 式 (XP の部分クラスに相当) に対しても，XPath 式修正問題が NP 困難となることを有向ハミルトン経路問題からの帰着により示す．一方，編集操作を core に限定した場合，XPath 式として XP に属するものを許した場合でも，多くの場合において効率良く解けることを後述のアルゴリズムにより明らかにする．

XPath 式の修正を行う場合，その式に構文的に近くかつ正しい XPath 式は複数存在し得るため，そのような XPath 式の中から望ましい式を選択できることが望ましい．そこで本論文では，編集操作を core に限定した場合において，XPath 式 q，DTDD，および正整数 K に対して，D の下で q に最も近い K 個の XPath 式を求めるアルゴリズムを構成する．ただし，たとえ編集操作が core に限定されていたとしても，q に最も近い K 個の XPath 式が効率良く得られるか否かは自明でない．この問題に対処するため，アルゴリズムを次の方針に基づいて構成する．(1) まず, q を修正して得られる，スキーマに妥当な XPath 式の集合を求める．(2) 次に, (1) で得られた XPath 式の中から，q に最も近い K 個の式を選択する．(1) の性質を満たし, かつ, q に最も近い K 個の XPath 式を効率良く選択することが可能な XPath 式集合を表現するため，５章において xd-graph という新しいグラフを提案する．Xd-graph の最も重要な点は，q を修正して得られる妥当な XPath 式がグラフ上の開始ノードから受理ノードへの経路に対応していることである．もう１つの重要な点は，開始ノードから受理ノードへの任意の経路 p に対して，p の重みが「入力 XPath 式 q を p で表される妥当な XPath 式に修正するためのコスト」に一致していることである．したがって，q に最も近い K 個の XPath 式を得るためには，xd-graph を求めた上で，この xd-graph 上で K 最短経路問題を解けばよいことになる．６章では，この考えに基づいて，入力 XPath 式 q と DTDD に対して，D の下で q に最も近い K 個の XPath 式を効求めるアルゴリズムを構成する．そして，本アルゴリズムが simple な XPath 式に対して多項式時間で動作することを示す．また，XP に属する XPath 式に対して，述語のネストの深さが定数で抑えられている場合に，提案アルゴリズムは多項式時間で動作することを示す．さらに，評価実験において，適切な枝刈り処理を行う

ことにより，本アルゴリズムが実 DTD の下で効率よく XPath 式を修正可能であることを明らかにする．

正規木文法 (regular tree grammar) は DTD より真に表現力の高いスキーマ言語であり，W3C XML Schema や RELAX NG などのスキーマ言語の形式モデルとしても用いられている．本論文では，正規木文法にも対応できるように上述のアルゴリズムを拡張する．DTD と正規木文法との最も大きな違いは，前者は一つの要素名に対して 1 つの型しか割り当てることができないのに対し，後者は一つの要素名に対して複数の型を割り当てることが可能であることである．7 章では，この「一つの要素名に対して複数の型を割りあて可能」という性質を xd-graph の定義に採り入れて拡張することにより，xd-graph と同様の性質を有する xg-graph を構成する．この xg-graph を用いることにより，入力 XPath 式 q と正規木文法 G に対して，G の下で q に最も近い K 個の XPath 式を求めるアルゴリズムを構成する．さらに，このアルゴリズムの計算複雑さについても考察し，正規木文法は DTD より真に表現力が高いにもかかわらず，DTD の場合と同様の効率で提案アルゴリズムが動作可能であることを明らかにする．

A Study on Algorithms for Finding Correct XPath Queries

Abstract

If a query written by a user does not return a desirable answer, then the user have to correct the query. In such cases, a method for helping users to write correct queries is much useful. For relational databases (RDBs), methods for helping users to write correct SQL queries have been proposed. On the other hand, Extensible Markup Language (XML) essentially has a much more complex data structure than RDB. Thus, helping users to write correct queries is a much more difficult problem, and very few effective methods for dealing with the problem have been proposed so far.

In order to help users to write correct queries, information about correct data structures is required. There are two possible approaches to obtain such information: (1) gathering structural information from data and (2) referring schema information instead of data. However, the former approach has some drawbacks. First, it is sometimes impossible to access some or entire part of data due to privacy and/or security reasons. Second, the size of data is extremely larger than that of schema, and large data is hard to be processed in environments with small resources. In such situations, it is useful to help users to write correct queries by using schema rather than data. Therefore, this dissertation focuses on the latter approach and considers correcting queries by using the structural information of schema. Assuming this approach, there are two possibilities to help users to write

correct queries. One is to help users while writing a query (e.g., completing element names), the other is to help users after a query is written. These are not incompatible to each other, and we can use both of them together. In particular, the latter is much useful if a user writes a query but the query returns undesirable results, or a correct query becomes invalid due to schema updates. This dissertation focuses on the latter and considers correcting XPath queries written by users.

Problems of analyzing the behavior of queries over schema without referring data are called *static analysis*, and the problem considered in this dissertation is a kind of static analysis problem. It is known that XML static analysis problems are intractable in many cases. For example, the XPath satisfiability problem is one of the most popular XML static analysis problem. This problem is shown to be undecidable in general case, and remains NP-hard even if only child and descendant-or-self axes are allowed. The XPath containment problem is another popular XML static analysis problem, and it is also shown that the problem is undecidable in general case. It is not clear whether the XPath query correction problem considered in this dissertation can be solved efficiently or not, and careful considerations are required to investigate the nature of the problem in terms of algorithm and computational complexity. Therefore, this dissertation aims at clarifying the boundaries between the tractability and the intractability of the XPath query correction problem, in terms of the restrictions on schemas and XPath queries.

The main contribution of this dissertation are the following.

1. A formal definition of edit operations to XPath query is presented for the first time. Two classes of edit operations, *core* and *extended*, are defined. These enable formal discussions about complexities and algorithms for correcting XPath queries. Besides this, two XPath fragments *simple* and *XP* are given in order to investigate the complexity of the XPath query correction problem. As stated later, these two factors surely affects the complexity of the problem. In particular, the XPath query correction problem is shown to be intractable if the extended edit operations are allowed.

2. Algorithms for solving the XPath query correction problem are presented, assuming that only the core edit operations are available. It is shown that the algorithms run efficiently for simple XPath queries. Moreover, sufficient conditions under which the algorithms run efficiently for queries in XP are identified.

Edit operation is the key to formalize the XPath query correction problem and also affects the complexity of the XPath query correction problem. In Chapter 4, six edit operations to XPath queries are proposed: (1) axis substitution, (2) label substitution, (3) location step insertion, (4) location step deletion, (5) location step exchange, and (6) predicate exchange. Above (1) to (4) are called *core* edit operations and (5) and (6) are called *extended* edit operations. The six edit operations are complete in the sense that any XPath query can be transformed into another arbitrary XPath query by using these edit operations, and the completeness still holds for the core edit operations. The other major factor of the complexity of the problem is XPath fragment. In this dissertation, two XPath fragments *simple* and *XP* are presented. In short, XP is the set of XPath queries using child, descendant-or-self, following-sibling, preceding-sibling, and attribute axes. In particular, a query q in XP is *simple* if q uses neither predicate nor attribute axes. Based on these definitions, the *XPath query correction problem* is defined to find, for an XPath query q and a DTD D, the query that is syntactically closest (i.e., having the least edit distance) to q. By reducing the directed Hamiltonian path problem to the XPath query correction problem, it is shown that if extended edit operations are allowed, then the problem becomes NP-hard even for simple XPath queries. It is also shown that if predicate exchange and label substitution are allowed at the same time, the problem is NP-hard even if each predicate of an XPath query is a simple location step having no predicate, which corresponds to a subclass of XP. On the other hand, by using the algorithms presented below, it is shown that if only core edit operations are allowed, then the problem is tractable in many cases even if queries in XP are allowed.

Since there may be more than one correct XPath query syntactically close to a query q, it is desirable for users to be able to choose a preferable query from queries obtained by correcting q. Therefore, the main algorithm proposed in this dissertation is designed for finding top-K queries syntactically close to a query q under a DTD D. However, even if the set of edit operations to queries is restricted to core, it is not clear if top-K queries syntactically close to q under D can be obtained efficiently. To cope with this problem, the algorithm proposed in this dissertation takes the following approach: (1) compute the set of valid queries obtained by correcting q, then (2) select top-K queries close to q among the valid queries. To model the set of valid queries in (1) that enables efficient computations of top-K queries, a novel graph called *xd-graph* is proposed. The most 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. 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. Based on this idea, in Chapter 6 an algorithm for finding top-K queries that runs in polynomial time for simple XPath queries is proposed. As for queries in XP, it is shown that the algorithm runs in polynomial time if the nest level of a predicate is bounded by a constant. It is also shown that the algorithm can efficiently find top-K queries under a real-world DTD by pruning unnecessary nodes and edges of an xd-graph appropriately.

Regular tree grammar is strictly more expressive than DTD and is used to model major powerful schema languages such as W3C XML Schema and RELAX NG. The above proposed algorithm is extended in order to handle such powerful schemas. The main difference between DTD and regular tree grammar is that the former assigns exactly one type to one element name while the latter is able to assign more than one type to one element name. In Chapter 7, a novel graph called

xg-graph is introduced by extending the definition of xd-graph to cope with this property inherent to regular tree grammar. Based on this xg-graph, an algorithm for finding, for an XPath query q and a regular tree grammar G, top-K queries syntactically close to q under G is presented. Then it is shown that the algorithm runs as efficient as the previous algorithm designed for DTD, despite the fact that regular tree grammar is strictly more expressive than DTD.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The more complex the data structure becomes, the more difficult it is to write "correct" queries. For relational databases (RDBs), a number of methods for helping users to write correct SQL queries have been proposed (e.g., [76, 63, 36, 19]). On the other hand, Extensible Markup Language (XML) essentially has a much more complex data structure than RDB, but very few studies on helping users to write correct queries for XML have been made so far.

In order to help users to write correct queries, we need information about correct data structures. There are two possible approaches to obtain such information: (1) gathering structural information from data and (2) referring schema information instead of data. However, the former approach has some drawbacks. First, it is sometimes impossible to access some or entire part of data due to privacy and/or security reasons. Second, the size of data is extremely larger than that of schema, and large data is hard to be processed in environments with small resources. In such situations, it is useful to help users to write correct queries by using schema rather than data. Therefore, this dissertation focuses on the latter approach and considers correcting queries by using the structural information of schema.

In general, there are two approaches to help users to write correct queries. One is to help users while writing a query (e.g., suggesting some keywords or subexpressions), the other is to help users *after* a query is written (e.g., correcting

a query that returns undesirable results). These approaches are not incompatible to each other, and we can use both of them together. In particular, the latter approach is much useful if a user writes a query but the query returns undesirable results, or a correct query becomes invalid due to schema updates. This dissertation focuses on the latter approach and considers correcting XPath queries written by users.

Problems of analyzing the behavior of query over schema without referring data are called *static analysis*, and the problem considered in this dissertation is a kind of static analysis problem. It is known that XML static analysis problems are intractable in many cases. For example, the XPath satisfiability problem, which is the most popular XML static analysis problem, is shown to be undecidable in general case, and remains NP-hard even if only child and descendant-or-self axes are allowed [11]. The XPath containment problem is another popular XML static analysis problem, which is shown to be undecidable [52]. The XML type checking problem is also a popular XML static analysis problem. Again it is shown that the problem is undecidable [1, 2] and remains intractable even if a number of restrictions are imposed on schemas and queries [44, 45]. Thus, whether the query correction problem considered in this dissertation can be solved efficiently or not is not clear, and careful considerations are required to investigate the nature of the problem in terms of computational complexity. Therefore, this dissertation aims at clarifying the boundaries between the tractability and the intractability of the XPath query correction problem, in terms of the restrictions on schemas and XPath queries.

The problem addressed in this dissertation is described as follows:

Input: A DTD D, a query q, and a positive integer K.

Problem: top-K XPath queries syntactically close to q among the XPath queries valid against the schema

As a brief example of the problem, let us consider the following simple DTD D as a schema.

```
<!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

$$q = \texttt{/person[@id = "123"]/nama},$$

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 dissertation. 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' [47].

```
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 the 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, schemas used in practice are larger and more complex [15]. In such a situation, a user tends not to understand the entire structure of a schemas exactly, and thus our algorithm is helpful for writing correct XPath queries on such schemas. Moreover, since the algorithm is based on the edit distance between XPath queries, we can

change the cost of an edit operation, if necessary. For example, if a user wants "concise" XPath queries that prefers descendant-or-self axes to child axes wherever possible, it suffices to decrease the costs of deleting child axis and inserting descendant-or-self axis.

The main contributions of this dissertation are the following threefold.

1. The notion of correcting XPath query has not been formalized so far. In this dissertation, a formal definition of edit operations to XPath query is presented for the first time. This enables formal discussions about complexities and algorithms for correcting XPath queries. The edit operations has two classes called "core" and "extended", which affect the (in)tractability of the XPath query correction problem.

2. Two XPath fragments "simple" and "XP" are given in order to investigate the complexity of the XPath query correction problem. Here, XP is the set of XPath queries using child, descendant-or-self, following-sibling, preceding-sibling, and attribute axes. In particular, a query q is *simple* if $q \in XP$ and q uses neither predicate nor attribute axes. It is shown that if core and extended edit operations are allowed, then the XPath query correction problem becomes intractable even for simple XPath queries. On the other hand, it is shown that if only core extended edit operations are allowed, then the XPath query correction problem can be solved efficiently in many cases.

3. Algorithms for solving the XPath query correction problem are presented, assuming that only the core edit operations are available. It is shown that the algorithms run efficiently for simple XPath queries. Also, sufficient conditions under which the algorithms run efficiently for queries in XP are identified.

An overview of our main algorithm is as follows. Let q be an XPath query and D be a DTD. 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. 1.1). 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. 1.1. The path $n_0 \rightarrow a_0 \rightarrow d_1 \rightarrow c_2$ on the xd-graph represents a valid query $/\downarrow::a/\downarrow::d/\downarrow::c$, which is obtained by inserting $/\downarrow::a$ to q and substituting the label of the first location step $\downarrow::e$ 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. It is also shown that the algorithm can efficiently find top-K queries under a real-world DTD by pruning unnecessary nodes and edges of an xd-graph appropriately.

This dissertation considers regular tree grammar as well as DTD. Regular tree grammar is strictly more expressive than DTD, and is used to model major powerful schema languages such as W3C XML Schema and RELAX NG. The above proposed algorithm is extended in order to handle such powerful schemas. The main difference between DTD and regular tree grammar is that the former assigns exactly one type to one element name while the latter is able to assign more than one type to one element name. In Chapter 7, a novel graph called xg-graph is introduced by extending the definition of xd-graph to cope with this property inherent to regular tree grammar. Based on this xg-graph, an algorithm for finding, for an XPath query q and a regular tree grammar G, top-K queries syntactically close to q under G is presented. Then it is shown that the algorithm runs as efficient as the previous algorithm designed for DTD, regular tree grammar is strictly more expressive than DTD nonetheless.

Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 shows related works and position of this dissertation. Chapter 3 gives some preliminary definitions of XPath and DTD. Chapter 4 defines edit operations to XPath queries and considers the complexity of the problem and shows that finding top-K valid XPath queries is NP-hard if extended edit operations are allowed. Chapter 5 introduces xd-graph, which forms the basis of our algorithm. Chapter 6 gives algorithms for finding top-K valid XPath queries under DTDs, assuming that only core edit operations are allowed. Chapter 7 extends the algorithms to use regular tree grammar as a schema instead of DTD. Chapter 8 shows some experimental results. Chapter 9 discusses the results of this dissertation and some future works. Chapter 10 summarizes this dissertation.

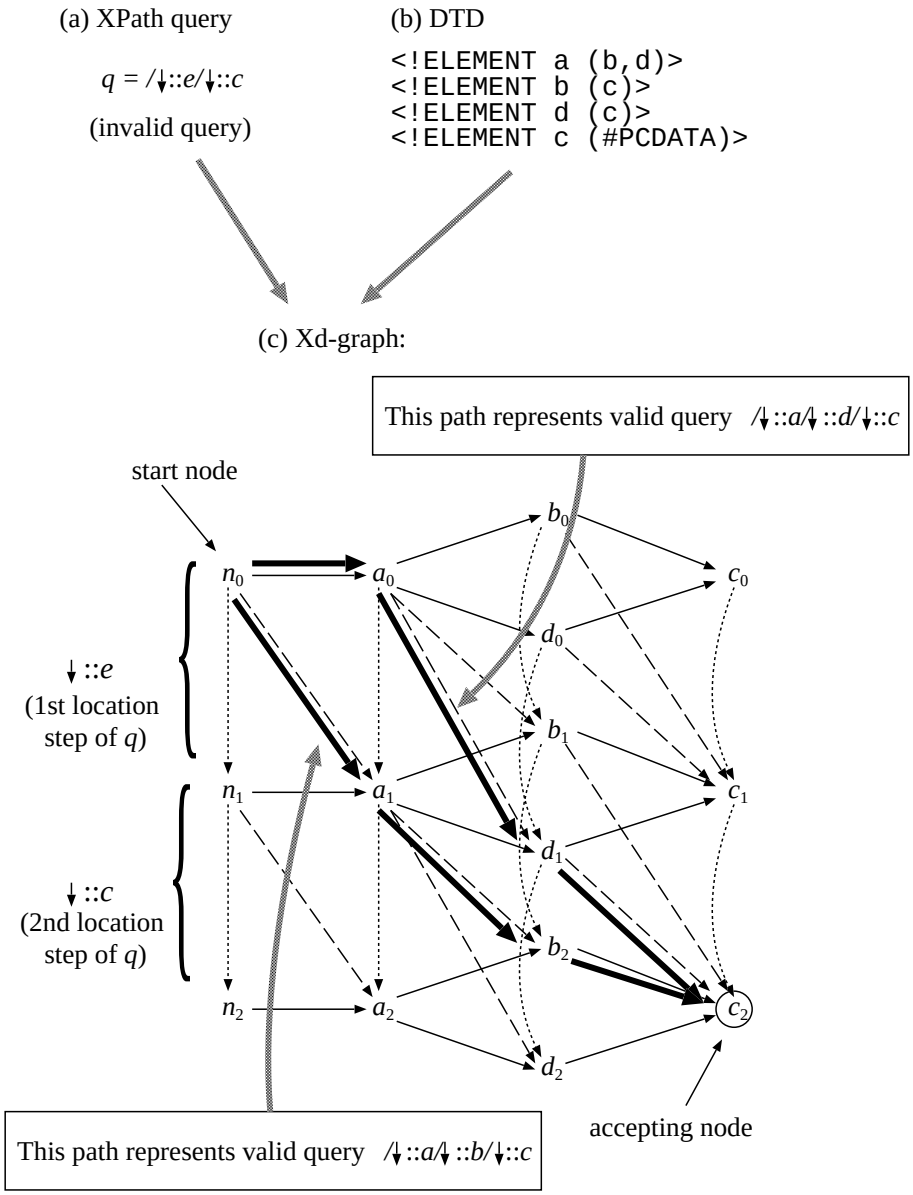


Figure 1.1: Overview of our algorithm

Chapter 2

Related Works

2.1 Edit Operation to Various Data Structure

Correcting XPath queries in this dissertation are essentially based on edit operations to XPath queries. Edit operation was firstly proposed in terms of strings by Levenshtein [40]. This consists of

- insertion of a symbol,
- deletion of a symbol, and
- substitution of a symbol for another symbol.

These have widely been accepted as the basis of edit operations. Then some extensions were proposed, e.g., *transposition* of two adjacent character, *merge*, and *split* [74, 60]. *Alignment* is a notion similar to edit distance, which is widely used to compare DNA sequences [62, 75, 22, 29, 12].

The notion of edit operation is also extended to handle other data structures that are more complex than string. In particular, tree is the most popular data structure to which edit operations are applied. The basic edit operations to trees consist of insertion of a node, deletion of a node, and relabeling of a node. Unlike string, a tree is either unordered or ordered, and thus we have two kinds of tree

edit distance problem: the *unordered* tree edit distance problem and the *ordered* tree edit distance problem. Whether a tree is ordered or not significantly affects the complexity of the problem. Actually, the former problem is shown to be NP-hard [86]. Moreover, the problem is shown to be MAXSNP-hard [9], and thus it is unlikely that there exists an efficient approximation algorithm for solving the problem. On the other hand, the latter problem is fortunately tractable. An algorithm for solving the problem was firstly introduced by Tai [70], then Zhang and Shasha proposes a new algorithm to the problem [85]. However, these algorithms run in $O(n^4)$ in the worst case. Recently, Demaine et al. reduced the upper bound complexity of the problem, in which their algorithm run in $O(n^3)$ time [17]. Pawlik proposed another $O(n^3)$ algorithm [55]. Besides string and tree, edit operations are also defined in terms of other data structures. For example, edit operations to graphs are used in order to measure the structural distance between two graphs [25]. In [65, 66] edit operations between a schema and an XML document are proposed, which are used to correct invalid XML documents into valid ones. Edit operations between two schemas are also proposed [30, 38, 39]. These edit operations are used to detect changes between old and new schemas.

2.2 XPath Satisfiability and Related Problems

The notion close to the syntactical validity of XPath in this dissertation is XPath satisfiability. Here, the *XPath satisfiability problem* is to decide, for a DTD D and an XPath query q, whether there exists an XML document d valid against D such that the answer of q on d is not empty. The main difference between the two notions is that the XPath satisfiability uses a more strict condition than the syntactical validity of XPath, and thus the XPath satisfiability problem has higher complexity than the syntactical validity of XPath.

In fact, assuming that no restriction is imposed on DTDs, the XPath satisfiability problem is shown to be NP-hard even if an XPath query uses only child

and descendant-or-self axes [10, 11]. Moreover, the problem remains intractable even under *fixed* DTDs [67]. Due to this situation, a number of studies on exploring subclasses of DTD under which the satisfiability problem is tractable have been made. For example, Ref. [27] considers the problem under non-recursive DTDs. However, non-recursiveness does not broaden the tractable class of XPath. On the other hand, Ref. [49] considers the problem under disjunction-free DTD, and shows that under disjunction-free DTDs the satisfiability problem becomes tractable for some XPath fragments. However, alternation '|' cannot be used under disjunction-free DTDs, which means that disjunction-freeness is too restrictive from a practical point of view. Besides these restricted DTDs, more practical subclasses of DTD have been proposed. It is shown that the problem becomes tractable under duplicate-free DTDs (i.e., DTDs in which no content model contains two occurrences of the same label) for XPath queries using only child and descendant-or-self axes [68, 69]. Moreover, disjunction-capsuled DTD (DC-DTD) and its extensions have been proposed. Here, a DTD is disjunction-capsuled if each alternation '|' is enclosed by '*' or '+'. Under these DTDs, the satisfiability problem is tractable if only child and descendant-or-self axes are allowed [32, 31, 33, 34]. Moreover, it is shown that most of real-world DTDs are categorized into DC-DTD and its extensions [34, 64].

Ref. [28] takes an approach different to the above studies. The study propose a system for solving the satisfiability problem by transforming a DTD and an XPath query into MSO-logic formulas and then solving the problem with decision procedures for MSO-logic formulas. This approach does not require restricting DTDs but requires exponential time in the worst case.

Another popular XML static analysis problem related to XPath is the XPath query containment problem [61]. For XPath queries p and q, q *contains* p if it holds that whenever a document d matches p d also matches q. It is shown that the XPath query containment problem is undecidable in general case [52], and remains coNP-hard in several restricted cases [78]. The XPath equivalence

problem is a problem similar to above, which is to decide whether given two XPath queries always return the same result. This problem is also shown to be intractable [48].

2.3 XML Query Correction and Related Studies

Currently, XML is widely used to represent various kind of data, and XPath [81] is the most popular query language for XML. Moreover, XPath is contained in transformation languages such as XSLT [83] and XQuery [82]. In such languages, XPath expressions are used to select elements to be transformed.

The ordinary XPath processors [37, 8] only verify the syntax of an inputs XPath expression according to the XPath specification. However, such XPath processors do not check the validity of an input XPath query in terms of a schema. On the other hand, our algorithm corrects an XPath expression that is invalid against a schema even though syntactically valid.

Although a number of studies on XPath have been made so far, studies on correcting XPath queries are unexpectedly not many. Ref. [16, 13] 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 and a query are represented by a DAG, while we use DTD and regular tree grammar as schema (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 [15], which suggests that it is meaningful to support recursive schemas. Besides this, the major limitation of their algorithm is that their algorithm outputs a DAG even if an input query is restricted to be a tree. Therefore, their algorithm cannot be directly applied to XPath query correction.

Besides query correction, several related but different approaches have been

studied for XML; query expansion, inexact queries, interaction, keyword search, minimizing query, etc. Ref. [58] proposes the node insertion operation that is also proposed in this dissertation. Ref. [57] takes a query expansion approach instead of correcting queries. Refs. [5, 6, 21, 20] 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. [42, 43], in which a user can write an XQuery query without specifying exact connections between elements. Ref. [50] proposes an interactive system for generating XQuery queries in which an XPath query is interactively created by an algorithm based on the interactive learning algorithm for regular expression [7]. There has been a number of studies on XML keyword search (e.g., [84, 41, 14, 71, 53, 72]), which are especially suitable for users that are not familiar with XML query languages. Refs. [4, 77, 56, 23] proposes minimizing XPath Queries for optimizing a query more efficiently but maintain outputs. Several XML editors (e.g., XMLSpy [3]) support auto-complete for XPath query editing, but they do not support listing K correct XPath queries.

Chapter 3

Preliminaries

In this chapter, we give some definitions related to XPath and DTD.

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 represented by a triple (d, α, site), where

$$\begin{aligned} d(\text{site}) &= \text{people}, \\ d(\text{people}) &= \text{person}^*, \\ d(\text{person}) &= (\text{name}, \text{emailaddress}, \text{phone?}), \\ d(\text{name}) &= \epsilon, \\ \alpha(\text{name}) &= \{\text{id}\}, \\ \alpha(e) &= \emptyset \quad \text{for any element } e \in \Sigma_e \backslash \{name\}. \end{aligned}$$

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 $d(a) = c(f|e)^*$.

Table 3.1: Syntax of XP

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

For a DTD $D = (d, \alpha, s)$ and labels $a, b \in \Sigma_e$, b is *reachable* from a in D if

- $a = b$ or b appears in $d(a)$, or
- 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 dissertation, we use XPath queries using child ($\downarrow$), descendant-or-self ($\downarrow^*$), 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 3.1. Thus, an *XPath query* (*query* for short) q in XP can be denoted

$$/ax[1] :: l[1][exp[1]]/ \cdots /ax[m] :: l[m][exp[m]], \tag{3.1}$$

where

$$\begin{aligned} ax[i] &\in \text{Axis}, \\ l[i] &\in \Sigma_e (1 \le i \le m-1), \\ exp[i] &\in \text{Exp}(1 \le i \le m), \\ ax[m] &\in \text{Axis} \cup \{@\}, \\ l[m] &\in \begin{cases} \Sigma_a & \text{if } ax[m] = @, \\ \Sigma_e & \text{otherwise.} \end{cases} \end{aligned}$$

If the ith location step has no predicate, then we write $exp[i] = \epsilon$. Although XP supports no upward axes, this usually gives little problem since the majority of XPath queries uses only downward axes[35].

Let q be a query in (3.1). 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

- $ax[i] = \downarrow$ and $l = l[i-1]$, or
- $ax[i] = \downarrow^*$, 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 $1 \leq i \leq m$ with $exp[i] \neq \epsilon$, 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*.

Chapter 4

Edit Operations to XPath Query and Intractability

In this chapter, we firstly define edit operations to queries and show that some edit operations make finding top-K valid queries intractable.

4.1 Edit Operations to XPath Query

We propose the following six kinds of *edit operations*.

1. *Axis substitution:* substitutes axis ax with ax', denoted $ax \rightarrow ax'$. For example, by applying $\downarrow \rightarrow \downarrow^*$ to $/\downarrow :: a$ we obtain $/\downarrow^* :: a$.

2. *Label substitution:* substitutes label l with l', denoted $l \rightarrow l'$. For example, by applying $a \rightarrow b$ to $/\downarrow :: a$ we obtain $/\downarrow :: b$.

3. *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 $/\downarrow :: a/\downarrow :: b$.

4. *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 $/\downarrow::b$.

5. *Location step exchange:* exchanges adjacent two location steps. For example, by applying this edit operation to $/\downarrow::a/\downarrow^*::b$ we obtain $/\downarrow^*::b/\downarrow::a$.

6. *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 $/\downarrow::a/\downarrow^*::c[b/d]$.

Above 1 to 4 are called *core* edit operations and 5 and 6 are called *extended* edit operations.

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 $1 \leq i \leq m$. 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 $q = /\downarrow::a/\downarrow::b[\downarrow::d[\downarrow::g]]/\rightarrow^+::c$. 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 dissertation, we assume the following. Let $U = \{\downarrow, \downarrow^*\}$, $S = \{\rightarrow^+, \leftarrow^+\}$, and $A = \{@\}$.

- An axis can be substituted with an axis of the "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,

$$\gamma(s) = \sum_{1 \leq i \leq n} \gamma(op_i).$$

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 $s_1, \cdots, s_K$ satisfying the following conditions.

1. Each of $s_1(q), \cdots, s_K(q)$ is valid against D.
2. $\gamma(s_1) \leq \cdots \leq \gamma(s_K)$.
3. $s_1, \cdots, s_K$ 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 $\gamma(s) \geq \gamma(s_K)$.

We say that $s_1(q), \cdots, s_K(q)$ are *top-K* queries *syntactically close* to q under D.

4.2 Edit Operations Causing Intractability

For an query q and a DTD D, top-K queries syntactically close to q under D may not be found efficiently if all the edit operation defined in the previous section are allowed. Actually, in this section we show that the extended edit operations 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 $\gamma(s) \leq K$ 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 even if a query is simple.*

Proof We reduce the directed Hamiltonian path problem, which is NP-complete [26], 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.

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

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

Second, query q is defined as follows.

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

For example, let us consider the directed graph shown in Fig. 4.1(A). From this graph, we obtain the DTD in Fig. 4.1(B) and the query in fig. 4.1(C) according to the above reduction. 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 $\gamma(s) \leq K$ and that $s(q)$ is valid against D.

($\Rightarrow$) Assume that H has a directed Hamiltonian path from u to v, say $u = v_{i_1} \rightarrow v_{i_2} \rightarrow \cdots \rightarrow v_{i_k} = v$. 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 $\gamma(s) \leq \frac{1}{2}k^2(k+1) = K$ 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 $\gamma(s) \leq K$ 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 even if each predicate of a query is a simple location step having no predicate.*

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.

$$\begin{aligned} \Sigma_e &= \{v\} \cup \{v_1, v_2, \cdots, v_k\} \cup \{a_1, a_2, \cdots, a_k\}, \\ \Sigma_a &= \emptyset, \end{aligned}$$

$$\begin{aligned} s' &= u, \\ d(a_i) &= \epsilon, \quad (1 \le i \le k) \\ d(v_i) &= seq_i, \quad (1 \le i \le k) \end{aligned}$$

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. □

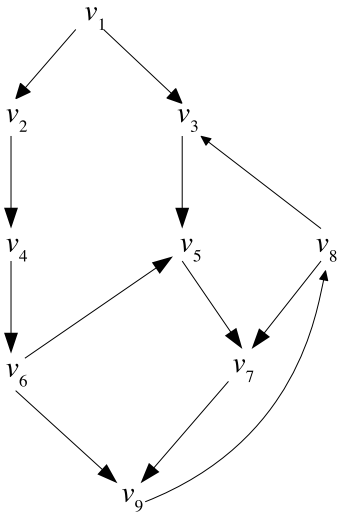
Thus, it is unlikely that we can find top-K valid queries efficiently, if we use the extended edit operations. In the following chapters, we consider finding top-K valid queries assuming that only the core edit operations are allowed.

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 the extended edit operations 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 extended edit operations are omitted.

(A) directed graph $H = (V,E)$



(B) DTD $D = (d, v_1, \alpha)$

$$
\begin{aligned}
d(v_1) &= v_2 v_3 \\
d(v_2) &= v_4 \\
d(v_3) &= v_5 \\
d(v_4) &= v_6 \\
d(v_5) &= v_7 \\
d(v_6) &= v_5 v_9 \\
d(v_7) &= v_9 \\
d(v_8) &= v_3 v_7 \\
d(v_9) &= v_8
\end{aligned}
$$

(C) XPath query q

$$/\downarrow::v_1/\downarrow::v_2/\downarrow::v_3/\downarrow::v_4/\downarrow::v_5/\downarrow::v_6/\downarrow::v_7/\downarrow::v_8/\downarrow::v_9$$

Figure 4.1: A directed graph $H = (V, E)$, a DTD $D = (d, v_1, \alpha)$, and a query q.

Chapter 5

Xd-Graph Representing Set of Valid Queries

In this chapter, 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, an xd-graph represents the set of valid queries obtained by correcting the input query.

Throughout this chapter, 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 dissertation we construct a graph called "xd-graph" from q and D, as shown in Fig. 5.1. 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

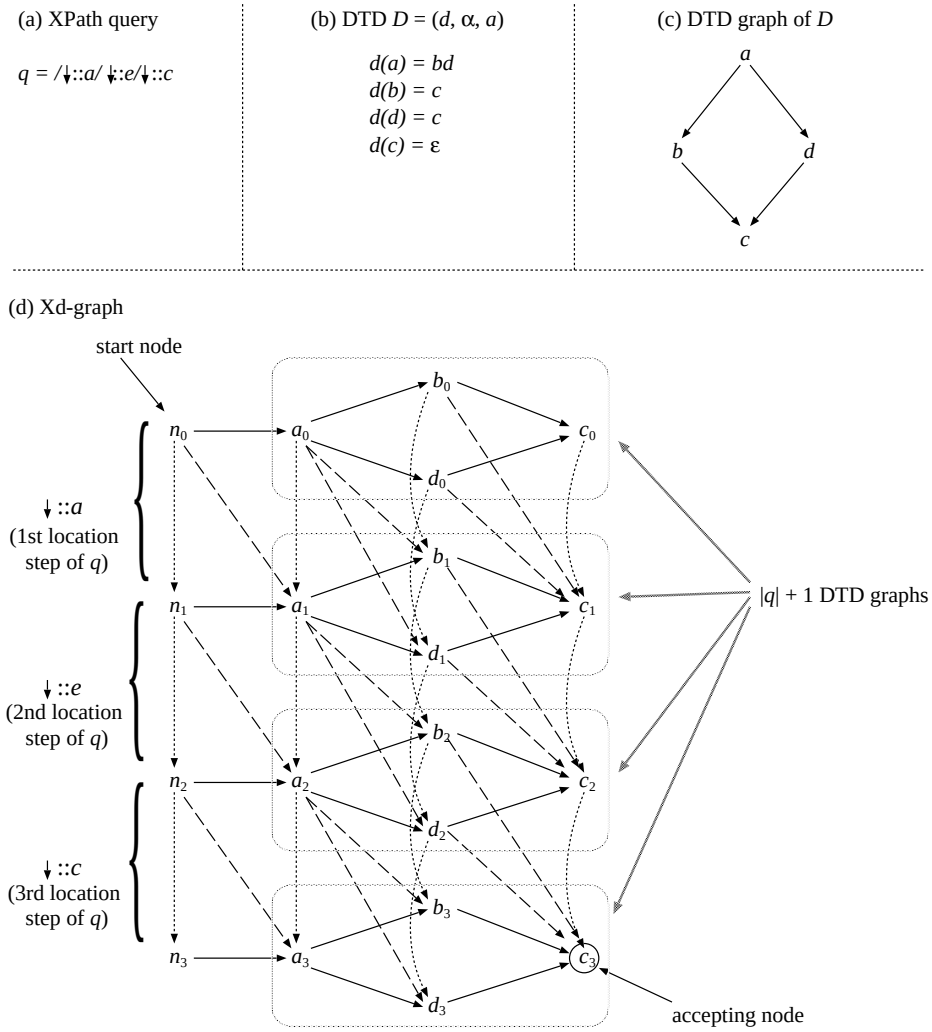


Figure 5.1: An example of xd-graph

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. 5.1(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. 5.1(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. 5.2, the edges between the first and second 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. 5.2).

- 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).

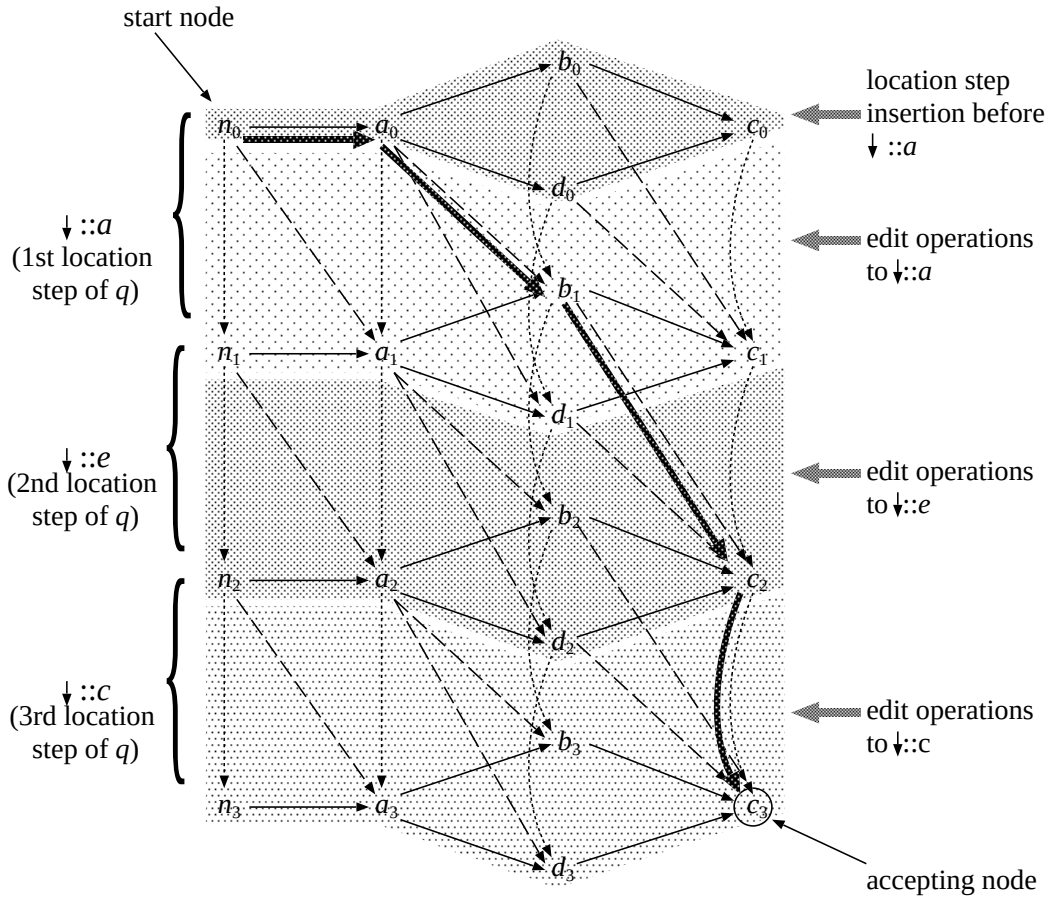


Figure 5.2: Xd-graph

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

$$V = \Sigma_e, E = \{l \rightarrow l' \mid \text{l' is a label appearing in d(l)}\}.$$

For example, Fig. 5.3 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.

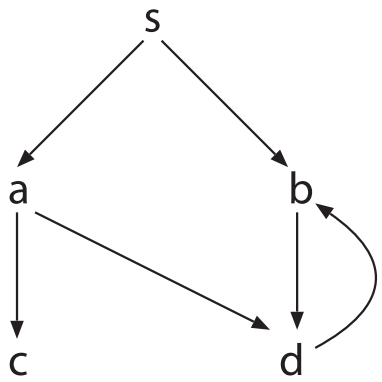


Figure 5.3: a DTD graph $G(D)$

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. 5.3. Since only $\downarrow$ axis is allowed, it suffices to consider location step insertion, location step deletion, and label substitution. Figure 5.4 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. 5.4.

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.

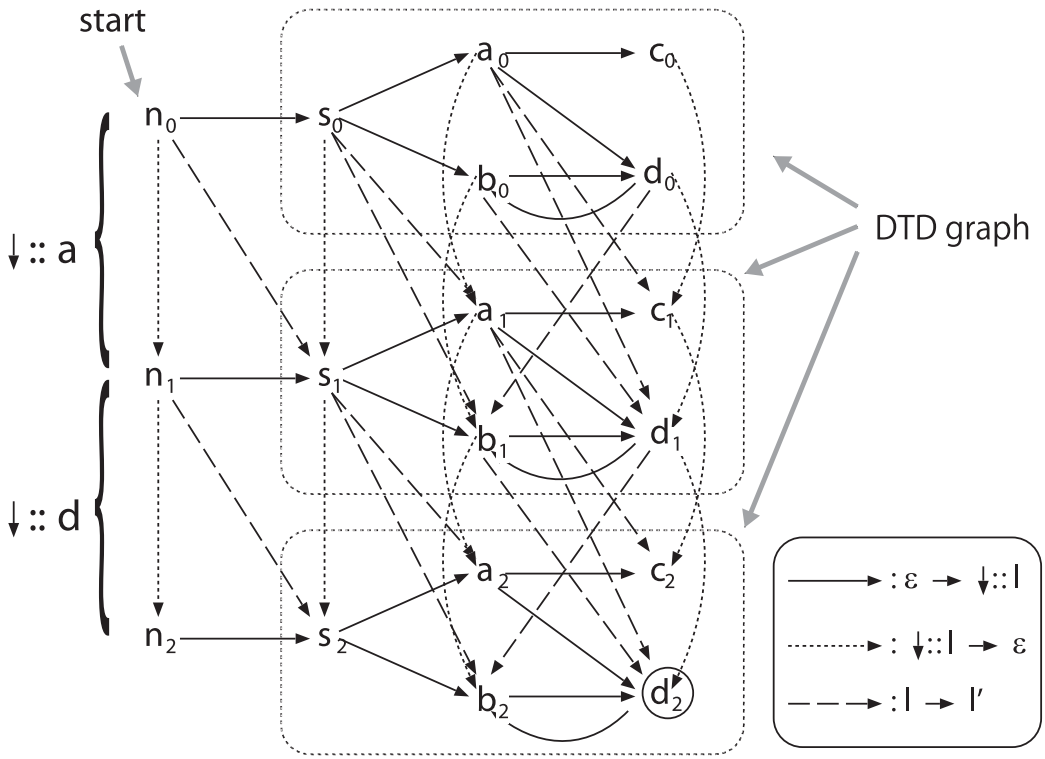


Figure 5.4: An xd-graph $G(q, G(D))$

- A "vertical" edge $l \dashrightarrow l'$ corresponds to a location step deletion.

More concretely, let us first consider horizontal edge $n_0 \rightarrow s_0$ in Fig. 5.4. This edge means "moving from the root node to child node s, using no location step of q". In other words, the edge $n_0 \rightarrow s_0$ represents adding a location step $\downarrow::s$, that is, the edge represents an edit operation $[\epsilon \rightarrow \downarrow::s]_0$. Let us next consider slant edge $s_0 \dashrightarrow b_1$ in Fig. 5.4. 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 $[a \rightarrow b]_1$. Finally, consider vertical edge $b_1 \dashrightarrow b_2$ in Fig. 5.4. 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 $[\downarrow::d \rightarrow \epsilon]_2$.

In Fig. 5.4, 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

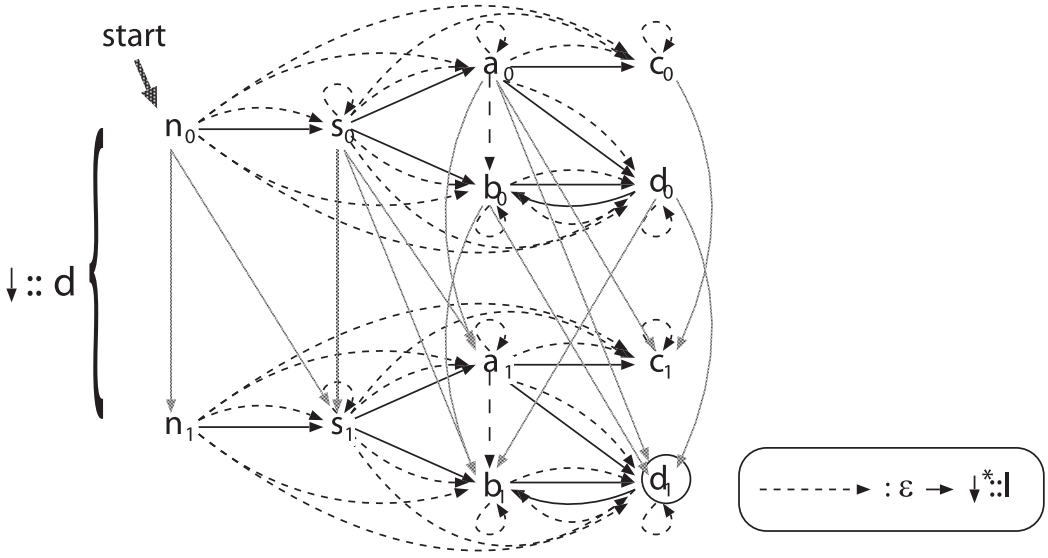


Figure 5.5: Edges representing location step insertion

against D obtained by correcting q. For example, let us consider a path $p = n_0 \rightarrow s_0 \dashrightarrow a_1 \dashrightarrow d_2$ in Fig. 5.4. Recall that $q = /\downarrow::a/\downarrow::d$. The first edge $n_0 \rightarrow s_0$ represents a location step insertion $[\epsilon \rightarrow \downarrow::s]_0$. The second edge $s_0 \dashrightarrow a_1$ represents a label substitution $[a \rightarrow a]_1$, 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 $[\epsilon \rightarrow \downarrow::s]_0[a \rightarrow a]_1[d \rightarrow d]_2$ 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. For this insertion, we add edges representing the edit operation to an xd-graph. Fig. 5.5 shows the xd-graph constructed from the DTD graph in Fig. 5.3 and a query $q = /\downarrow::d$. Each dashed edge in Fig. 5.5 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., $[\epsilon \rightarrow \downarrow^*::d]_0$. As stated before,

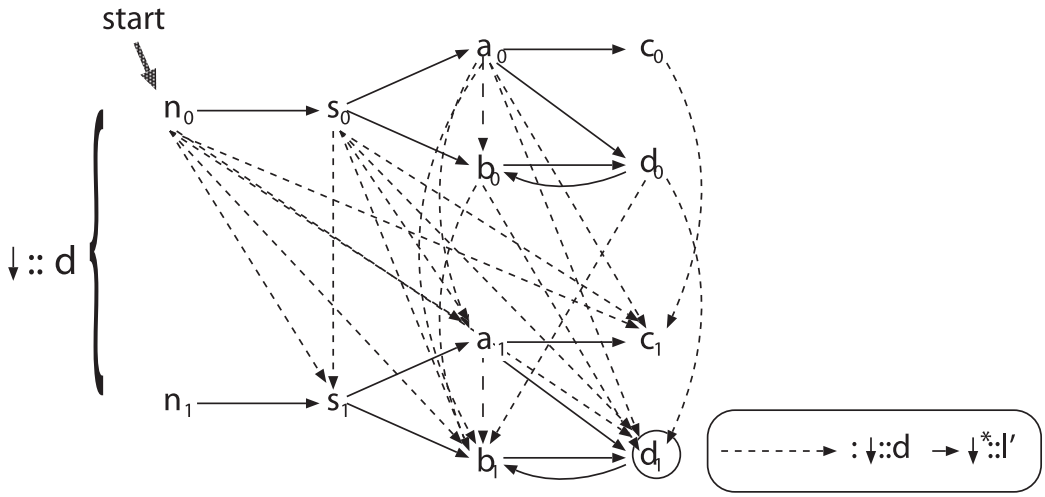


Figure 5.6: Edges representing axis substitution

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, $n_0 \rightarrow s_1 \dashrightarrow d_1$ 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. 5.6 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. 5.6, 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. 5.6. 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.

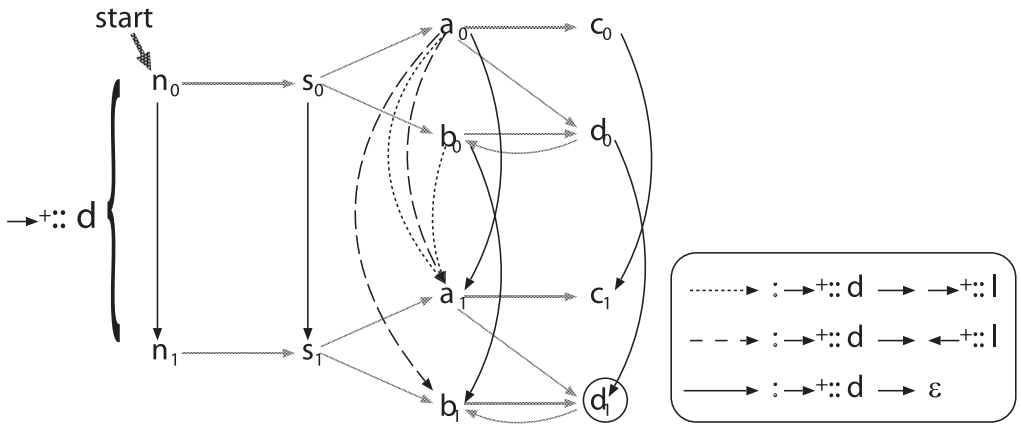


Figure 5.7: Edges dealing with $\rightarrow^+$ and $\leftarrow^+$ axes

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

Let us consider handling $\rightarrow^+$ and $\leftarrow^+$ axes. Fig. 5.7 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. 5.7 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 us now give the 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. Moreover, 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 $E_i = \{l_i \to l'_i \mid l \to l' \in E\}$ for $0 \leq i \leq m$. Then the *xd-graph* for q and $G(D)$, denoted $G(q, G(D))$, is a directed graph (V', E'), where

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

Here, E_{insc} in (5.1) is the set of edges inserting $\downarrow :: l$ (corresponding to "$\epsilon \to \downarrow :: l$" in Fig. 5.4), that is,

$$E_{insc} = \{n_0 \to s_0, \cdots, n_m \to s_m\} \cup (E_0 \cup \cdots \cup E_m),$$

where E_i is the set of edges of $G_i(D)$. E'_i in (5.1) is the set of edges inserting $\downarrow^* :: l$ (corresponding to "$\epsilon \to \downarrow^* :: l$" in Fig. 5.5) and is defined as follows.

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

F_i in (5.1) 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 $ax[i] \in \{\downarrow, \downarrow^*\}$: $F_i = D_i \cup C_i \cup A_i$, where

$$\begin{aligned} D_i &= \{n_{i-1} \to n_i\} \cup \{l_{i-1} \to l_i \mid l \in V\}, && (5.1) \\ C_i &= \{n_{i-1} \to s_i\} \cup \{l_{i-1} \to l'_i \mid l \to l' \in E\}, \\ A_i &= \{n_{i-1} \to l_i \mid l_i \in V_i\} \cup \{l_{i-1} \to l'_i \mid l' \text{ is reachable from } l \text{ in } D\}. && (5.2) \end{aligned}$$

Here, D_i is the set of edges corresponding to "$\downarrow :: l \to \epsilon$" in Fig. 5.4, C_i is the set of edges corresponding to "$l \to l'$" in Fig. 5.4, and A_i is the set of edges corresponding to "$\downarrow :: d \to \downarrow^* :: l$" in Fig 5.6.

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

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

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. 5.7.

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. 5.4, we have $\gamma(p) = \gamma(\epsilon \rightarrow\downarrow:: s) + (\gamma(\downarrow\rightarrow\downarrow) + \gamma(a \rightarrow a)) + (\gamma(\downarrow\rightarrow\downarrow) + \gamma(d \rightarrow d)) = 1 + 0 + 0 = 1$.

Chapter 6

Algorithm for Finding top-K Queries under DTDs

In this chapter, 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

[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.

$G(q, G(D))$ between the start 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. 5.1, 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. 5.1(c). In line 2, we obtain the xd-graph shown in Fig. 5.1(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 $/ \downarrow :: a/ \downarrow :: b/ \downarrow :: c$.

[2]There are a number of algorithms for solving K shortest paths problem (e.g., [46, 18]). Here we can use any of them.

- $n_0 \dashrightarrow a_1 \dashrightarrow d_2 \dashrightarrow c_3$. 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.

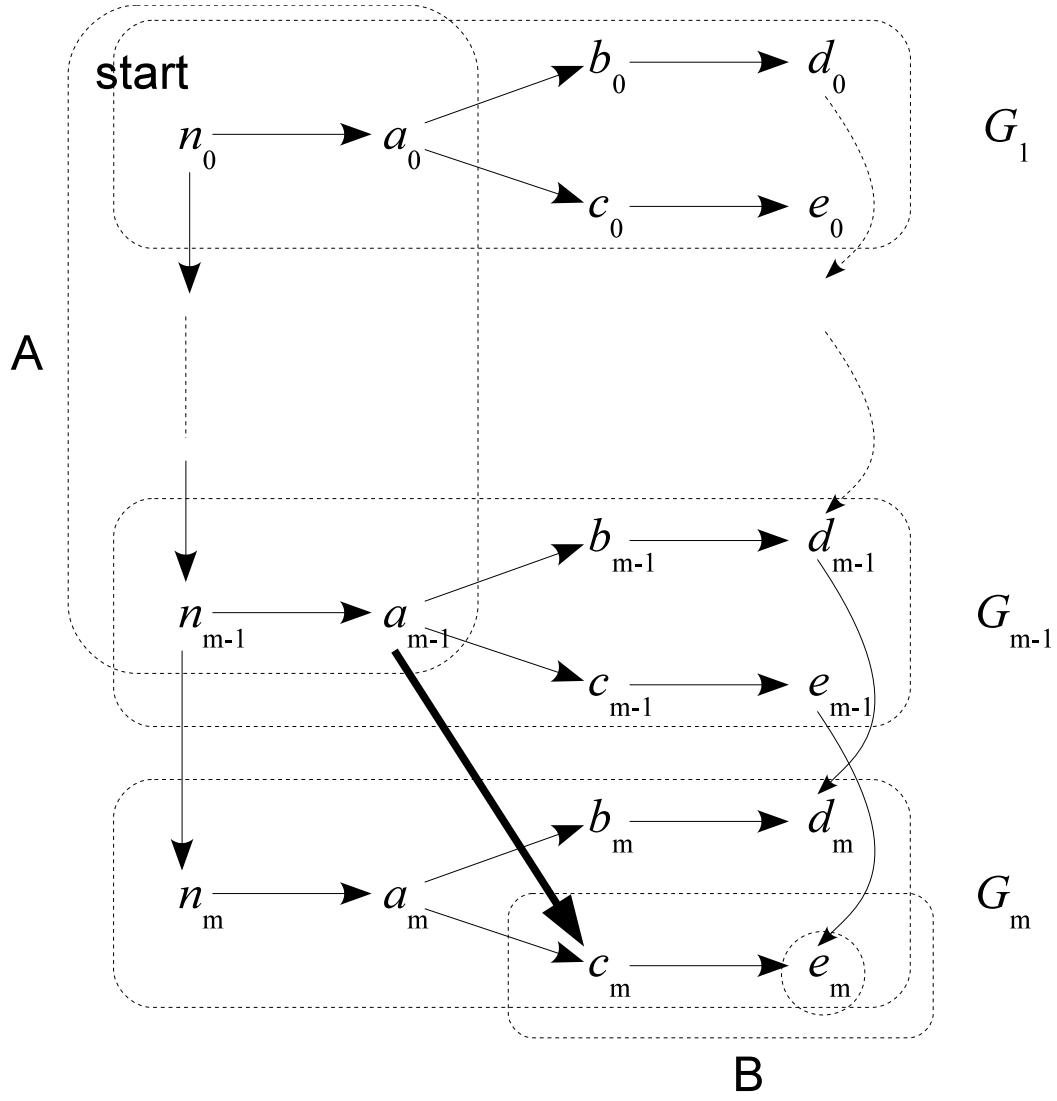


Figure 6.1: Xd-graph

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 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. 6.1). 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.

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 $a_{m-1} \rightarrow c_m$ (see Fig. 6.1). 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. 6.1 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. 6.1 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., [46, 18]), 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 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.* □

Let D be a DTD, $q = /ax[1] :: l[1]/ \cdots /ax[m] :: l[m]$ be a query, and $G(q, G(D))$ be the xd-graph for q and D. The proposed algorithm assumes that the label of the accepting node of $G(q, G(D))$ coincides with that of the last location step of q, that is, $l[m]_m$ is the accepting node of $G(q, G(D))$. Here, consider

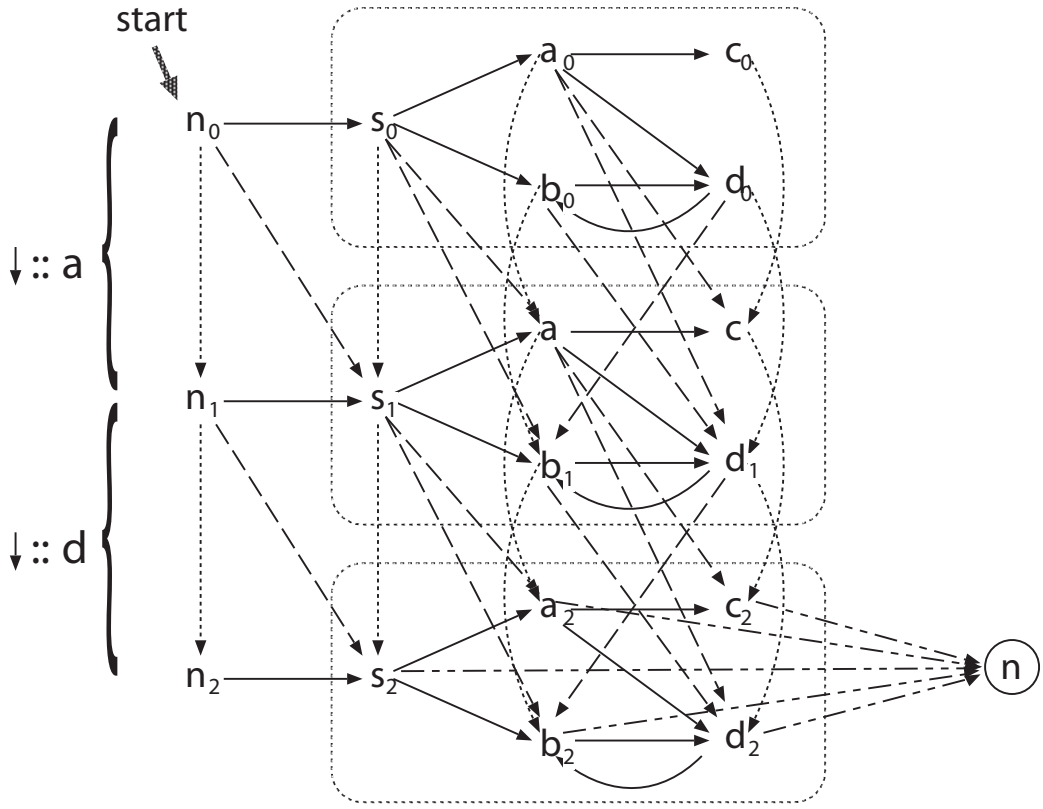


Figure 6.2: Adding a new accepting node n to $G(q, G(D))$

the case that a user write a query q. Then the label of the last location step of q represents the most symbolic label that the user wants to retrieve, thus the user tends not to write a completely wrong label at the last location step of q. Therefore, in most cases we believe that the above assumption is a reasonable one. However, if a user prefers a more general query correction, we can relax the above assumption by extending the xd-graph, as follows (see Fig. 6.2)

1. Add a new accepting node n to $G(q, G(D))$.

2. For each node l_i in $G(q, G(D))$, add and edge $l_i \rightarrow n$.

3. For each edge $l_i \rightarrow n$ added above, let $\gamma(l_i \rightarrow n) = 0$.

On the other hand, the drawback of the above extension is that the approach tends

to output "too many" corrected queries. Thus the above extension should be used only if no desirable query correction is obtained under the original xd-graph.

6.2 Algorithm for Queries in XP

The algorithm proposed in the previous section can handle only simple queries. In this section, 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]

[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 $exp[i]$.

- Suppose $exp[i] \neq \epsilon$. The cost of deleting location step $ax[i] :: l[i][exp[i]]$ should be $\gamma(ax[i] :: l[i] \rightarrow \epsilon) + \gamma(exp[i] \rightarrow \epsilon)$, where "$exp[i] \rightarrow \epsilon$" 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. 6.3 (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.
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. (5.1)), let $\gamma(e) \leftarrow \gamma(e) + \gamma(exp[i] \rightarrow \epsilon)$.

   (b) For each node $l_i \in V_i$, do the following (i) – (iii).

      i. Replace l_i with its corresponding gadget (Fig. 6.3).

      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.

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

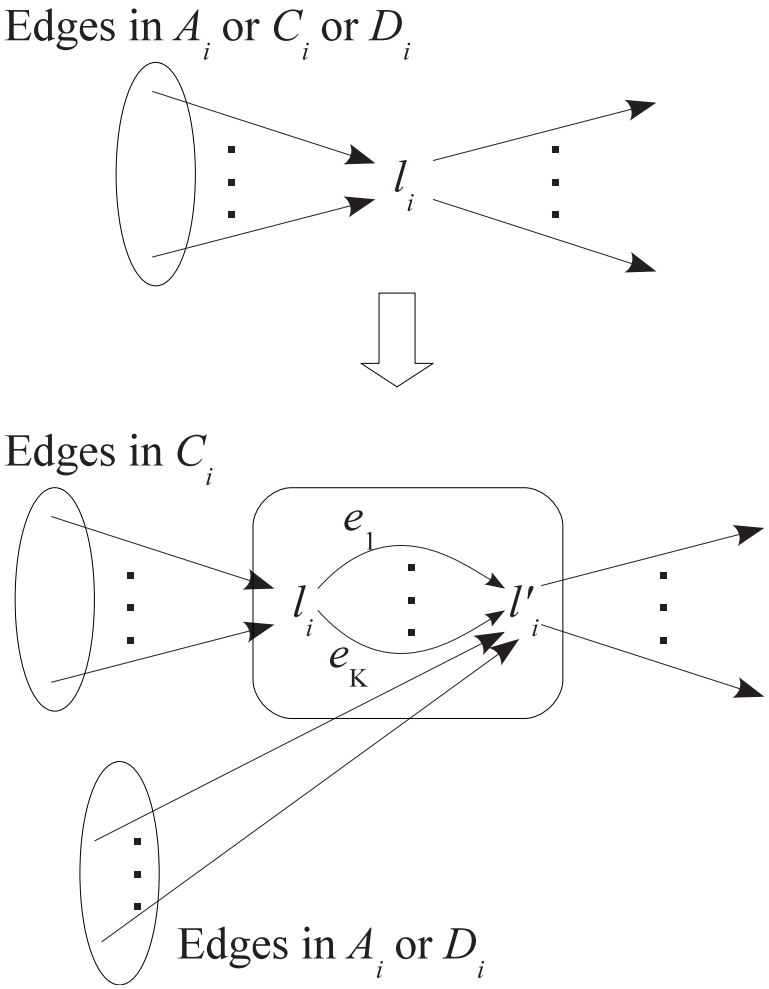


Figure 6.3: Node l_i and its gadget, where l'_i is a new node and $e_1, \cdots, e_K$ are new edges.

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. 6.3).

   (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 $q_1, \cdots, q_K$.

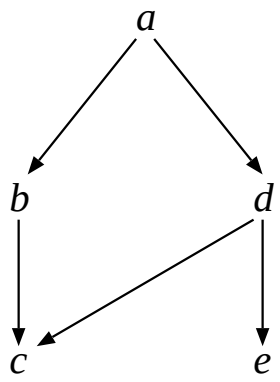


Figure 6.4: DTD graph $G(D)$.

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. 6.4. 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. 6.5, 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. 6.5), while the costs of the other edges 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. 6.6. 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 $e_1 \dashrightarrow e_2'$

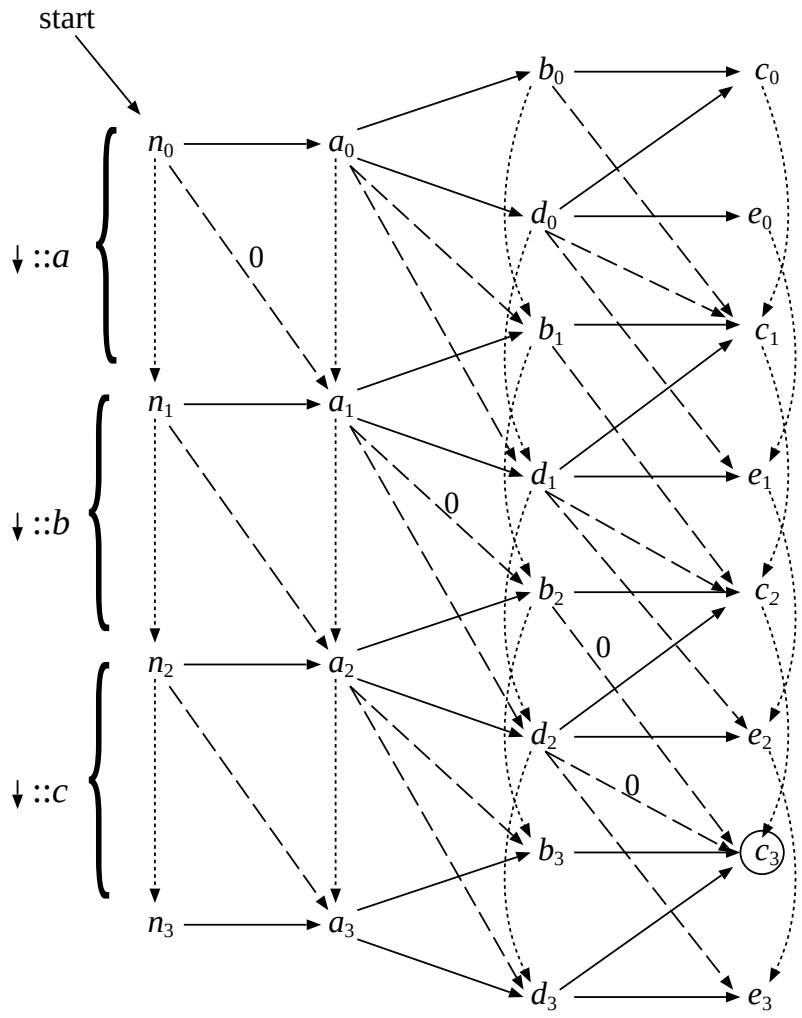


Figure 6.5: Xd-graph $G(sp(q), G(D))$.

are increased by one (the edges labeled by "2" in Fig. 6.6), 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.

- $n_0 \dashrightarrow a_1 \dashrightarrow b_2 \overset{A}{\rightarrow} b_2' \dashrightarrow c_3$. The third edge $b_2 \overset{A}{\rightarrow} b_2'$ 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 $/\downarrow :: a/\downarrow :: b[\downarrow :: c]/\downarrow :: c$.

- $n_0 \dashrightarrow a_1 \dashrightarrow b_2 \overset{B}{\rightarrow} b_2' \dashrightarrow c_3$. The third edge $b_2 \overset{B}{\rightarrow} b_2'$ represents deleting the predicate of q, while the other edges do nothing. Thus we obtain $/\downarrow::a/\downarrow::b/\downarrow::c$.

- $n_0 \dashrightarrow a_1 \dashrightarrow d_2 \overset{A}{\rightarrow} d_2' \dashrightarrow c_3$. 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 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 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 $q = /ax[1] :: l[1][exp[1]]$ $/\cdots/ax[m] :: l[m][exp[m]]$. By $mnl(q)$ we mean the *maximum nest level* of q, that

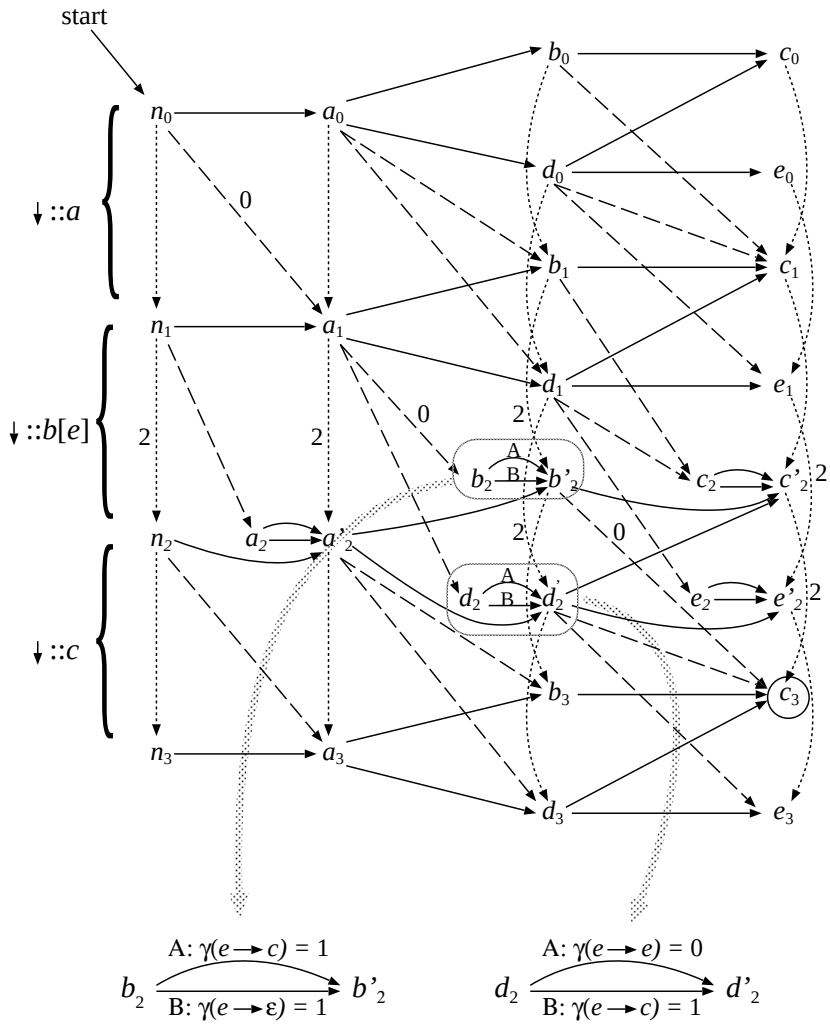


Figure 6.6: The graph obtained by modifying $G(sp(q), G(D))$.

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. Thus we have the following.

Theorem 6 *Let D be a DTD, $q \in$ XP be a query, and K be a positive integer. Then* FindKPaths*(D, q, K) runs in polynomial time of $|D|$ and $|q|$ if $mnl(q)$ is constant.*

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. 5.4 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 section 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, the pruning itself can be done very efficiently. Actually, the pruning needs (1) a top-down traversal from the start node and (2) a bottom-up traversal from the accepting node, each of which can be done by a breath-first traversal. Since a breath-first traversal can be done in $O(|V| + |E|)$ time for a graph (V, E) [73] and the numbers of nodes and edges of an xd-graph are in $O(|\Sigma_e|)$, the pruning can be done in

$$O(|\Sigma_e| + |\Sigma_e|) = O(|\Sigma_e|).$$

We also make an experiment to evaluate the effect of this pruning. This is shown in Section 8.2.

Chapter 7

Algorithm for Regular Tree Grammar

In this chapter, we extend the algorithms to use regular tree grammar as a schema instead of DTD. Regular tree grammar is a general schema model for XML including local tree grammar, which is equivalent to DTD [51]. Since regular tree grammar can assign more than one type (non-terminal) to a label, we cannot use the definition of xd-graph in that shape. In the following, we firstly define regular tree grammar formally, then extend xd-graph for regular tree grammar (xg-graph). By using xg-graph, we describe how to find correct XPath queries.

7.1 Regular Tree Grammar

A *regular tree grammar* is a 4-tuple $G = (N, T, S, P)$, where N is a set of *non-terminals*, T is a set of *terminals*, $S \in N$ is a set of *start symbols*, P is a set of *productions* of the form $X \rightarrow ar$ such that $X \in N, a \in T$, and r is a regular expression over N. We say that X is the *left-hand side* of the production, ar is the *right-hand side*, a is the *label*, and r is the *content model*.

For example, the DTD in Chapter 1 can be represented by a regular tree gram-

mar $G = (N, T, S, P)$, where

$$\begin{aligned} N &= \{Site, People, Person, Name, Pcdata\}, \\ T &= \{site, people, person, name, pcdata\}, \\ S &= \{Site\}, \\ P &= \{Site \rightarrow site(People), People \rightarrow people(Person^*), \\ &\quad Person \rightarrow person(Name), Name \rightarrow Pcdata(Pcdata), \\ &\quad Pcdata \rightarrow pcdata(\epsilon)\}. \end{aligned}$$

For a regular tree grammar $G = (N, T, S, P)$ and labels a, b, we say that b is *reachable* from a in G if either one of the conditions is satisfied.

- $a = b$, or there are productions $X \rightarrow ar$ and $X' \rightarrow br'$ in P such that X' occurs in r.
- For a label a', a' is reachable from a, and there are productions $X \rightarrow a'r$ and $X' \rightarrow br'$ such that X' occurs in r.

Let t be a tree. An *interpretation* I of t against G is a mapping from each node e in t to a state $I(e)$ that satisfies the following conditions.

- If e is the root of t, $I(e)$ is an initial state.
- There is a production $X \rightarrow ar$ in G such that $I(e) = X$, a is a terminal of e, and that $I(e_0)I(e_1)...I(e_m)$ matches r, where $e_0, e_1, ..., e_m$ are the child nodes of e.

By $L(G)$ we mean the *language* of G. Then $t \in L(G)$ if and only if there is an interpretation of t against G.

It is shown that regular tree grammar is strictly more expressive than DTD [51]. Actually, both W3C XML Schema [79, 80] and RELAX NG [54] can be modeled by regular tree grammar. For example, let us consider the RELAX NG schema

shown in Fig. 7.1. In this schema, an `item` element can be of type `CD` or `Book`, and the `CD` and `Book` types have different content models. Such types cannot be modeled by any DTD since each element must have exactly one content model. On the other hand, regular tree grammar can handle such types. The following regular tree grammar $G = (N, T, S, P)$ corresponds to the RELAX NG schema in Fig. 7.1.

$$
\begin{aligned}
N &= \{Catalog, CD, Book, Title, Artist, Author, Pcdata\} \\
T &= \{\text{catalog}, \text{item}, \text{title}, \text{artist}, \text{author}, \text{pcdata}\} \\
S &= \{Catalog\} \\
P &= \{Catalog \rightarrow \text{catalog}(CD^{+}\ Book^{+}), \\
&\quad\ CD \rightarrow \text{item}(Title\ Artist), \\
&\quad\ Book \rightarrow \text{item}(Title\ Author), \\
&\quad\ Title \rightarrow \text{title}(Pcdata), \\
&\quad\ Artist \rightarrow \text{artist}(Pcdata), \\
&\quad\ Author \rightarrow \text{author}(Pcdata), \\
&\quad\ Pcdata \rightarrow \text{pcdata}(\epsilon)\}
\end{aligned}
$$

7.2 Xg-Graph

In this section, we introduce a graph called xg-graph. This is an extended version of xd-graph that can handle non-terminal of regular tree grammar. Throughout this section, we assume that each query is simple. The general case is considered in Section 7.3.2.

Similar to Section 6.1, we obtain K optimum edit scripts against a query q under $G = (N, T, S, P)$ as follows.

```
<?xml version="1.0" encoding="Shift_JIS" ?>
<rng:grammar
  xmlns:rng="http://relaxng.org/ns/structure/1.0">

  <rng:start>
    <rng:ref name="Catalog" />
  </rng:start>

  <rng:define name="Catalog">
    <rng:element name="catalog">
      <rng:oneOrMore>
        <rng:ref name="CD" />
      </rng:oneOrMore>
      <rng:oneOrMore>
        <rng:ref name="Book" />
      </rng:oneOrMore>
    </rng:element>
  </rng:define>

  <rng:define name="CD">
    <rng:element name="item">
      <rng:ref name="Title" />
      <rng:ref name="Artist" />
    </rng:element>
  </rng:define>

  <rng:define name="Book">
    <rng:element name="item">
      <rng:ref name="Title" />
      <rng:ref name="Author" />
    </rng:element>
  </rng:define>
   <rng:define name="Title">
    <rng:element name="title">
      <rng:text />
    </rng:element>
  </rng:define>

  <rng:define name="Artist">
    <rng:element name="artist">
      <rng:text />
    </rng:element>
  </rng:define>

  <rng:define name="Author">
    <rng:element name="author">
      <rng:text />
    </rng:element>
  </rng:define>

</rng:grammar>
```

Figure 7.1: An example of RELAX NG schema

1. Construct a production-graph $G(P)$ from P in G.

2. Construct an xg-graph $G(q, G(P))$ from q and $G(P)$. It consists of all paths that represent XPath expressions valid against G obtained by applying edit scripts on q.

3. Solve K shortest paths problem over $G(q, G(P))$ to obtain top-K optimum edit scripts against q under G. The details is considered in Section 7.3.

7.2.1 Production-Graph

To construct an xg-graph, we need to represent productions of regular tree grammar as a graph. Therefore, we first define production-graph.

Let $G = (N, T, S, P)$ be a regular tree grammar. A *production-graph* $G(P) = (V, E)$ is a directed graph, where

$$
\begin{aligned}
V &= \{(X, a) \mid X \to ar \in P\}, \\
E &= \{(X, a) \to (X', a') \mid X \to ar \in P, X' \text{ occurs in } r, X' \neq \text{Pcdata}, \\
&\quad X' \to a'r' \in P \text{ for some } a' \in T \text{ and some regular expression } r' \text{ over } N\}.
\end{aligned}
$$

For example, a production-graph $G(P)$ for regular tree grammar $G = (N, T, S, P)$ is shown in Fig. 7.2, where

$$
\begin{aligned}
N &= \{S, A, B, C, D, Pcdata\}, \\
T &= \{s, a, b, c, d, pcdata\}, \\
S &= \{S\}, \\
P &= \{S \to s(A, B), A \to a(C, D), B \to b(D) \\
&\quad C \to c(Pcdata), D \to d(B|Pcdata), \\
&\quad Pcdata \to pcdata(\epsilon)\}.
\end{aligned}
$$

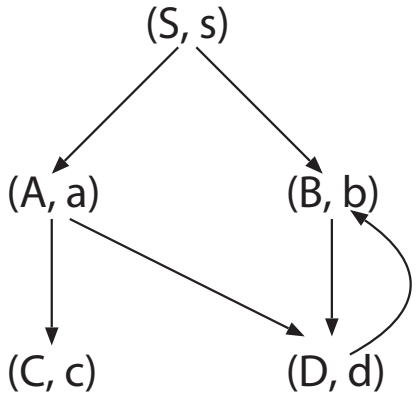


Figure 7.2: An example of production-graph

7.2.2 Xg-Graph

We next construct an xg-graph from a query q and a production graph $G(P)$. Intuitively, each edge on an xg-graph corresponds to an edit operation defined in Chapter 4.

In the following, we first explain an xg-graph for the case of (1) mentioned below (the case of (2) and (3) are omitted because they are similar to the discussions in Section 5.2).

(1) Only child ($\downarrow$) can be used as an axis

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

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

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

Let us consider the xg-graph constructed from a simple query $q = /\downarrow::a/\downarrow::d$ and the production-graph $G(P)$ in Figure 7.2. Since only $\downarrow$ axis is allowed, it suffices to consider location step insertion, location step deletion, and label substitution.

Figure. 7.3 shows an xg-graph $G(q, G(P))$ constructed from q and $G(P)$. As shown in this figure, the xg-graph is constructed from $|q| + 1$ copies of $G(P)$ with their nodes connected by several edges. Note that $(N, n)_0, (N, n)_1, (N, n)_2$ are newly

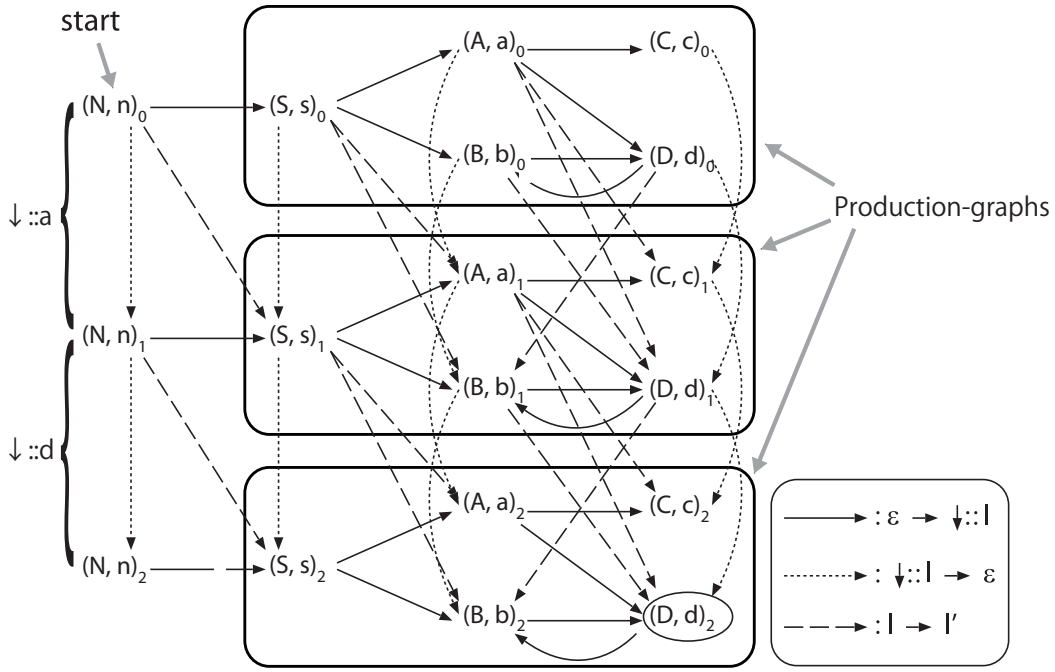


Figure 7.3: An xg-graph $G(q, G(P))$

added nodes, which correspond to the "root node" in the XPath data model. We subscript each node of a production-graph to distinguish the nodes. For example, the node (S, s) over $G(P)$ is denoted $(S, s)_0$ on the topmost production-graph of $G(p, G(P))$, $(S, s)_1$ on the second topmost production-graph, and so on.

As shown in Fig. 7.3, we have the following three kinds of edges in an xg-graph.

- A "horizontal" edge ($\rightarrow$) corresponds to a location step insertion ($\epsilon \rightarrow \downarrow :: l$).
- A "slant" edge ($\dashrightarrow$) corresponds to a label substitution ($l \rightarrow l'$).
- A "vertical" edge ($\cdots\!\!>$) corresponds to a location step deletion ($\downarrow :: l \rightarrow \epsilon$).

More concretely, let us first consider horizontal edge $n_0 \rightarrow (S, s)_0$ in Fig. 7.3. This edge means "moving from the root node to child node s, using no location step of q". This is moving to child node s by location step $\downarrow :: s$ using no location step in q. That is, the edge represents an edit operation $[\epsilon \rightarrow \downarrow :: s]_0$.

Let us next consider slant edge $(S, s)_0 \dashrightarrow (B, b)_1$ in Fig. 7.3. 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 $s_0 \dashrightarrow (B, b)_1$ represents $[a \rightarrow b]_1$.

Finally, consider vertical edge $(B, b)_1 \dashrightarrow (B, b)_2$ in Fig. 7.3. This edge means "staying the same node b by ignoring (deleting) the second location step $\downarrow::\ d$ of q". Thus the edge $b_1 \dashrightarrow (B, b)_2$ represents $[\downarrow::\ d \rightarrow \epsilon]_2$.

In Fig. 7.3, $(N, n)_0$ is called the *start node* and $(D, d)_2$ is called the *accepting node*. Each path from the start node to the accepting node represents a simple query valid against G obtained by correcting q. For example, let us consider a path $p = (N, n)_0 \rightarrow (S, s)_0 \dashrightarrow (A, a)_1 \dashrightarrow (D, d)_2$ in Fig. 7.3. In this path, the first edge $(N, n)_0 \rightarrow (S, s)_0$ represents a location step insertion $[\epsilon \rightarrow \downarrow::\ s]_0$. The second edge $(S, s)_0 \dashrightarrow (A, a)_1$ represents a label substitution $[a \rightarrow a]_1$, that is, the first location step "$\downarrow::\ a$" of p is unchanged. Similarly, the location step "$\downarrow::\ d$" of q is unchanged. Therefore, p represents an XPath expression $/\downarrow::\ s/\downarrow::\ a/\downarrow::\ d$, which is valid against G.

Formal Definition of Xg-graph and Costs of Edges

Let us show the formal definition of xg-graph, and the cost of each edge on the graph.

Let $G = (N, T, S, P)$ be a regular tree grammar, $G(P) = (V, E)$ be a production-graph, $p = /ax[1] :: l[1]/\cdots/ax[m] :: l[m]$ be a simple query. By $G_i(P)$ we mean the graph obtained from $G(P)$ by subscripting each node with i. That is, $V_i = \{(X, a)_i \mid X \rightarrow ar \in P\}$ and $E_i = \{(X, a)_i \rightarrow (X', a')_i \mid X \rightarrow ar \in P\}$.

The *xg-graph* for p and $G(P)$, denoted $G(p, G(P))$, is a directed graph (V', E') defined as follows (Note that n of $(N, n)_0, \cdots, (N, n)_m$ is a label where $n \notin V$).

$$\begin{aligned} V' &= \{(N, n)_0, \cdots, (N, n)_m\} \cup V_0 \cup \cdots \cup V_m \\ E' &= E_{insc} \cup (F_1 \cup \cdots \cup F_m) \end{aligned} \tag{7.1}$$

Here, E_{insc} in (7.1) is the set of edges inserting $\downarrow :: l$ (corresponding to "$\epsilon \rightarrow \downarrow :: l$" in Fig. 7.3) and is defined as follows. Note that E_i is an edge in $G_i(P)$.

$$E_{insc} = \{(N, n)_0 \rightarrow (S, s)_0, \cdots, (N, n)_m \rightarrow (S, s)_m\} \cup (E_0 \cup \cdots \cup E_m)$$

F_i $(1 \leq i \leq m)$ in (7.1) is the set of edges between $G_{i-1}(P)$ and $G_i(P)$, and have some definitions depending on the type of the ith axis on p. In this dissertation, for simplicity we assume that $ax[i] \in \{\downarrow\}$ and $F_i = D_i \cup C_i$. Then D_i and C_i are defined as follows. Note that D_i corresponds an edge "$\downarrow :: l \rightarrow \epsilon$" in Fig. 7.3 and C_i corresponds "$l \rightarrow l'$" in Fig. 7.3, respectively.

$$D_i = \{(N, n)_{i-1} \rightarrow (N, n)_i\} \cup \{l_{i-1} \rightarrow l_i \mid l \in V\} \tag{7.2}$$

$$C_i = \{(N, n)_{i-1} \rightarrow (S, s)_i\} \cup \{l_{i-1} \rightarrow l'_i \mid l \rightarrow l' \in E\} \tag{7.3}$$

We next consider the cost of each edge on an xg-graph. Suppose that following costs correspond edit scripts defined in Chapter 4.

- $\gamma(l \rightarrow l')$: cost of label substituting l to l'
- $\gamma(\epsilon \rightarrow ax :: l)$: cost of inserting a location step $ax :: l$
- $\gamma(ax :: l \rightarrow \epsilon)$: cost of deleting a location step $ax :: l$

According to the above costs, we define the cost $\gamma(e)$ of an edge$e \in E'$ on an xg-graph, as follows.

- The case where $e \in E_{inssc}$: We denote $e = l_i \rightarrow l'_i$. Since this edge represents inserting a location step $\downarrow :: l'$, we define $\gamma(e) = \gamma(\epsilon \rightarrow \downarrow :: l')$.
- The case where $e \in D_i$: We denote $e = l_{i-1} \rightarrow l_i$. Since this edge represents deleting a location step $ax[i] :: l[i]$, we define $\gamma(e) = \gamma(ax[i] :: l[i] \rightarrow \epsilon)$.
- The case where $e \in C_i$: We denote $e = l_{i-1} \rightarrow l'_i$. Since this edge represents substituting $ax[i]$ with $\downarrow$ and substituting $l[i]$ with l', we define

$\gamma(e) = \gamma(ax[i] \rightarrow\downarrow) + \gamma(l[i] \rightarrow l')$.

For example, let us consider the following cost function.

$$\begin{aligned} \gamma(l \rightarrow l') &= \begin{cases} 0 & \text{when } l = l' \\ 1 & \text{otherwise} \end{cases} \\ \gamma(\epsilon \rightarrow ax :: l) &= 1 \\ \gamma(ax :: l \rightarrow \epsilon) &= 1 \end{aligned}$$

Then for the path $(N, n)_0 \rightarrow (S, s)_0 \dashrightarrow (A, a)_1 \dashrightarrow (D, d)_2$, we obtain a cost $\gamma(\epsilon \rightarrow\downarrow:: s) + (\gamma(\downarrow\rightarrow\downarrow) + \gamma(a \rightarrow a)) + (\gamma(\downarrow\rightarrow\downarrow) + \gamma(d \rightarrow d)) = 1 + 0 + 0 = 1$.

7.3 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 regular tree grammar. We first consider the case where a query is simple, then present for queries in XP. Furthermore, we consider the pruning for graphs.

7.3.1 Algorithm for Simple Query

Let $G(P)$ be a regular tree grammar, $q = /ax[1] :: l[1]/ \cdots /ax[m] :: l[m]$ be a simple query, $G(q, G(P)) = (V, E)$ be the xg-graph for q and $G(P)$. Moreover, let $(N, n)_0 \in V$ be the start node and $(X, l[m])_m \in V$ be the accepting node of $G(q, G(P))$.

By the above cost definition, in order to find top-K queries syntactically close to q under G, it suffices to solve the K shortest paths problem over the xg-graph $G(q, G(P))$ between the start node and the accepting node, and output K XPath expressions represented by the obtained paths $p'_1, \cdots, p'_K$. The algorithm is described as follows.

Input: A regular tree grammar $G = (N, T, S, P)$, 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 G.

1. Construct the production graph $G(P)$ of G.
2. Construct the xg-graph $G(q, G(P))$ for q and $G(P)$.
3. Solve the K shortest paths problem on $G(q, G(P))$ between the start node and the accepting node of $G(q, G(P))$.
4. Let $q_1, \cdots, q_K$ be the queries represented by the K paths obtained in step 3. Return $q_1, \cdots, q_K$.

Thus we have the following.

Theorem 7 *Let $G = (N, T, S, P)$ be a regular tree grammar, q be a simple query, and K be a positive integer. Then the above algorithm outputs top-K queries syntactically close to q under G. Moreover, the algorithm runs in $O(K \cdot |q| \cdot |P|^2 \cdot \log(|q| \cdot |P|))$ time.*

7.3.2 Algorithm for Queries in XP

Let $q = /ax[1] :: l[1][exp[1]]/\cdots/ax[m] :: l[m][exp[m]] \in XP$, $sp(q)$ be a query expression obtained from extracting location steps with predicates from XP. that is,

$$sp(q) = /ax[1] :: l[1]/\cdots/ax[m] :: l[m].$$

Similar to FindKPaths, we construct an xg-graph $G(sp(q), G(P))$ first, then solve top-K shortest paths problem.

FindKPathsOnG(G, q, K)

Input: A regular tree grammar $G = (N, T, S, P)$, a query $q = /ax[1] :: l[1][exp[1]]/\cdots/ax[m] :: l[m][exp[m]]$, and a positive integer K.
Output: Top-K queries syntactically close to q under G.

1. Construct the production-graph $G(P)$.
2. Construct the xg-graph $G(sp(q), G(P))$.
3. For each $1 \leq i \leq m$ with $exp[i] \neq \epsilon$, modify $G(sp(q), G(P))$ as follows.
   (a) For each edge $e \in D_i$, let $\gamma(e) \leftarrow \gamma(e) + \gamma(exp[i] \rightarrow \epsilon)$.
   (b) For each node $l_i \in V_i$, do the following (i) – (iii).
      i. Replace l_i with its corresponding gadget (Fig. 6.3).
      ii. Call FindKPathsOnG(G', q', K), where $G' = (N, T, X, P)$, X of node (X, l_i) where $X \rightarrow ar \in P$, and $q' = /l_i/exp[i]$. Let $sc'_1, \cdots, sc'_K$ be the results.
      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} on $G(sp(q), G(D))$ with its corresponding gadget (Fig. 6.3)
   (b) $op_1, \cdots, op_K$ be the K optimum edit operations for Let $ax[m] :: l[m]$.
   (c) $\gamma(e_j) \leftarrow \gamma(op_j)$ for every $1 \leq j \leq K$.
5. Solve the K shortest paths problem on $G(sp(q), G(P))$ modified as above.
6. Let $sc_1, \cdots, sc_K$ be the results obtained above. Return $sc_1(q), \cdots, sc_K(q)$.

The following result can be obtained similarly to Theorems 5 and 6.

Theorem 8 *Let $G = (N, T, S, P)$ be a regular tree grammar, $q \in XP$ be a query, and K be a positive integer. Then* FindKPathsOnG *outputs top-K queries syntactically close to q under G. Moreover,* FindKPathsOnG *runs in $O(K \cdot |q| \cdot |P|^{2+mnl(q)} \cdot \log(|q| \cdot |P|))$.*

7.3.3 Graph Optimization and Pruning

In the above algorithms, pruning a production-graph and an xg-graph is not considered. By optimizing nodes on a production-graph and pruning unnecessary nodes on an xg-graph, we can save space and time.

In this section, we first describe optimizing of a production-graph, then show how to prune an xg-graph.

Optimization of Production-Graph

Since regular tree grammar allow "conflicting" non-terminals, same non-terminals may occur more than once in a production-graph. Therefore, more than one identical query may be obtained when top-K edit scripts are applied to a query.

For example, let us consider a regular tree grammar, where

$$
\begin{aligned}
N &= \{S, A1, A2, B, C, D, Pcdata\}, \\
T &= \{s, a, b, c, d, pcdata\}, \\
S &= \{S\}, \\
P &= \{S \rightarrow s(A1, A2), A1 \rightarrow a(B, C), A2 \rightarrow a(B, D) \\
&\quad B \rightarrow b(Pcdata), C \rightarrow c(Pcdata), \\
&\quad D \rightarrow d(Pcdata), Pcdata \rightarrow pcdata(\epsilon)\}.
\end{aligned}
$$

Then the algorithm constructs the production-graph of G shown in Fig.7.4. Here, the terminals of the paths $(S, s) \rightarrow (A1, a) \rightarrow (B, b)$ and $(S, s) \rightarrow (A2, a) \rightarrow (B, b)$ are same, $/ \downarrow :: a / \downarrow :: b$ may outputted more than once by solving the K shortest paths problem on an xg-graph. To avoid this problem, we *contract* nodes with same terminals as shown in Fig.7.5. In the following, we describe contraction formally.

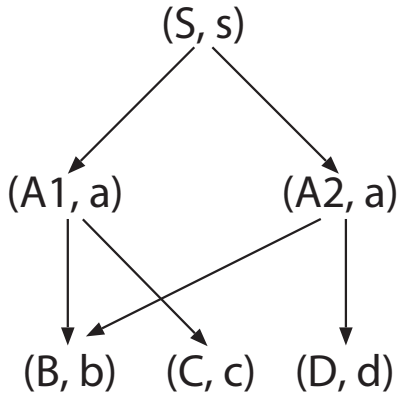


Figure 7.4: A production-graph that non-terminals conflict

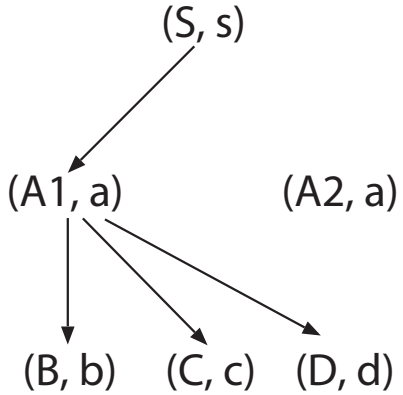


Figure 7.5: A (contracted) production-graph that non-terminals do not conflict

Recall that a production-graph $G(P) = (V, E)$ is a directed graph, where

$$
\begin{aligned}
V &= \{(X, a) \mid X \to ar \in P\}, \\
E &= \{(X, a) \to (X', a') \mid X \to ar \in P,\ X' \text{ occurs in } r,\ X' \neq \text{Pcdata}, \\
&\qquad X' \to a'r' \in P \text{ for some } a' \in T \text{ and some regular expression } r' \text{ over } N\}.
\end{aligned}
$$

If $X_i \neq X_j, a_i = a_j$, there exist nodes (X_i, a_i), (X_j, a_j) reachable from a node in $\{(S, a) \mid S \to ar \in P\}$, and the following four conditions hold, then a production-graph can be constructed by (1) and (2) described bellow.

- (X_j, a_j) is unreachable from (X_i, a_i)

- (X_i, a_i) is unreachable from (X_j, a_j)
- (X_i, a_i) have no (X_i, a_i) as a child node.
- (X_j, a_j) have no (X_j, a_j) as a child node.

1. Let

$$\begin{aligned} E_i &= \{(X_i, a_i) \to (X_i', a_i') \mid \\ &\quad X_i \to a_i r \in P, X_i' \text{ occurs in } r, X_i' \neq Pcdata, \\ &\quad X_i' \to a_i' r' \in P \text{ for some } a_i' \in T \text{ and some regular expression } r' \text{ on } N\}, \\ E_j &= \{(X_j, a_j) \to (X_j', a_j') \mid \\ &\quad X_j \to a_j r \in P, X_j' \text{ occurs in } r, X_j' \neq Pcdata, \\ &\quad X_j' \to a_j' r' \in P \text{ for some } a_j' \in T \text{ and some regular expression } r' \text{ on } N\}. \end{aligned}$$

   For any $(X_j, a_j) \to (X_j', a_j') \in E_j$, if $a_i' \neq a_j'$ for any $(X_i, a_i) \to (X_i', a_i') \in E_i$, then add $(X_i, a_i) \to (X_j', a_j')$ to E and delete $(X_j, a_j) \to (X_j', a_j')$ from E.

2. Let

$$\begin{aligned} E_j &= \{(X_{j-1}, a_{j-1}) \to (X_j, a_j) \mid \\ &\quad X_j \to a_{j-1} r \in P, X_j \text{ occurs in } r, X_j \neq Pcdata, \\ &\quad X_j \to a_j r' \in P \text{ for some } a_j \in T \text{ and some regular expression } r' \text{ on } N\}. \end{aligned}$$

   Delete every edge in E_j from E, and add $(X_{j-1}, a_{j-1}) \to (X_i, a_i)$ to E .

Pruning Xg-Graph

Suppose that G is a local tree grammar. Then we can save space and time for solving K shortest paths problem by deleting nodes unreachable to the accepting node.

Let us consider the size of an xg-graph $G(q, G(P))$ for a local tree grammar $G = (N, T, S, P)$ and a query q. For simplicity, we assume that a production-graph $G(P)$ of G is a complete n-ary tree. Since whether a node can reach the accepting node or not depends on $\rightarrow^+$ and $\leftarrow^+$ axes, we have two cases to be considered.

- The case where q does not include $\rightarrow^+$ nor $\leftarrow^+$ axis : The path that can reach the accepting node is only one from the assumption. Thus, the number of necessary nodes on $G(P)$ is $\log |T|$, therefore the size of the xg-graph is reduced to $O(|q| \cdot \log |T|)$.

- The case where q includes $\rightarrow^+$ or $\leftarrow^+$ axis : Let us assume that the depth of the node that includes $\rightarrow^+$ or $\leftarrow^+$ axis first from the document element is l. From the assumption, the number of necessary nodes on $G(P)$ is $\frac{|T|}{2^l}$, therefore the size of the xg-graph is reduced to $|q| \cdot \frac{|T|}{2^l}$. In this case, since $\rightarrow^+$ or $\leftarrow^+$ axis does not occur in $l = 0$, that is, the document element, we can reduce the size of an xg-graph to $\frac{1}{2}$ or less.

Chapter 8

Experimental Results

In this chapter, we present two experimental results. The first experiment evaluates the "quality" of the output of the algorithm, and the second experiment evaluates the execution time of the algorithm.

The algorithm is implemented in Ruby, and the experiments are performed on an Apple Xserver with the following specifications.

- Mac OS X Server 10.6.8
- Xeon 2.26GHz CPU
- 6GB Memory
- Ruby-1.9.3

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

8.1 Quality of the Output of the Algorithm

For a Schema S and an incorrect query q written by a user, there are a number of queries similar to q under S, 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 such 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 schema used in this experiment is auction.dtd of XMark [59], which is a recursive schema. As for XPath queries, we use XPath queries of XPathMark [24]. 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 implementation of our algorithm. 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". Table. 8.1 shows the above seven correct queries and conditions.

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 so that the query 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 8.1. 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 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.

$$
\begin{aligned}
\gamma(l \to l') &= \text{the normalized string edit distance [47]} \\
&\quad \text{between } l \text{ and } l', \\
\gamma(ax \to ax') &= \begin{cases} 0 & \text{if } ax = ax', \\ 2 & \text{otherwise,} \end{cases} \\
\gamma(\epsilon \to ax :: l) &= 1, \\
\gamma(ax :: l \to \epsilon) &= 2.
\end{aligned}
$$

Fig. 8.1 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 8.1. 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

Table 8.1: XPath queries (correct queries) and conditions

1.	/site/closed_auctions/closed_auction/annotation/description/text/keyword
	Keywords in annotations of closed auctions {child}
2.	//closed_auction//keyword
	Keywords in annotations of closed auctions {descendant}
3.	/site/closed_auctions/closed_auction//keyword
	Keywords in annotations of closed auctions {child and descendant}
4.	/site/closed_auctions/closed_auction[annotation/description/text/keyword]/date
	Closed auctions with an annotation containing a keyword {filter with child}
5.	/site/closed_auctions/closed_auction[descendant::keyword]/date
	Closed auctions with an annotation containing a keyword {filter with descendant}
6.	/site/open_auctions/open_auction/bidder[following-sibling::bidder]
	Bidders except the last one of each open auction {following-sibling}
7.	/site/open_auctions/open_auction/bidder[preceding-sibling::bidder]
	Bidders except the first one of each open auction {preceding-sibling}

of the incorrect query is the following,

```
/closed_auctions/closed_auction/annotation/description//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.

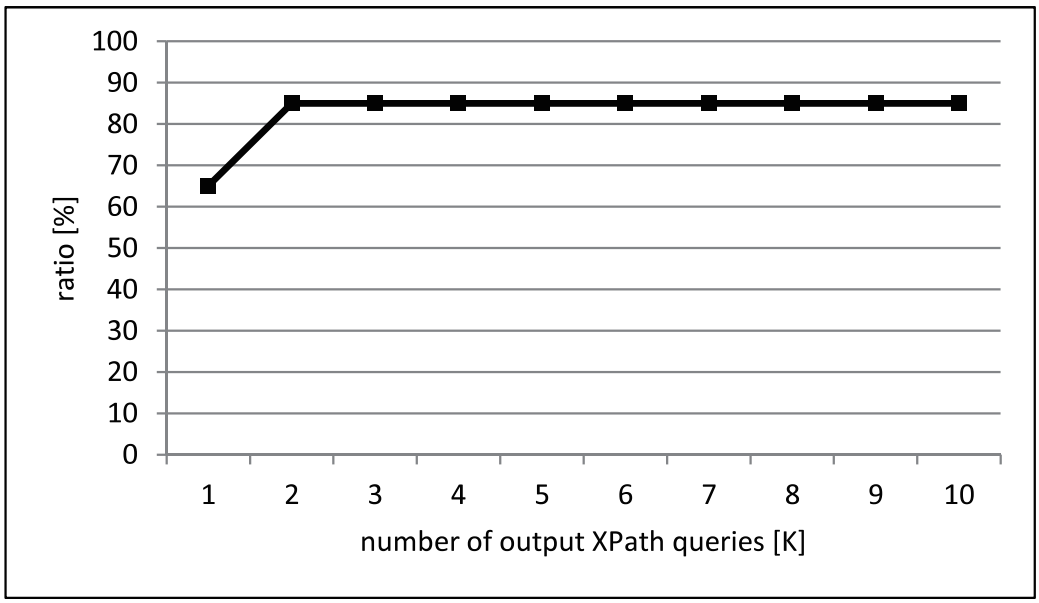


Figure 8.1: Ratios at which the outputs contain correct answers

8.2 Execution Time of the Algorithm

We next evaluate the execution time of the algorithm. In particular, 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 partition the queries shown in Table 8.1 into two sets. First set Q_1contains 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

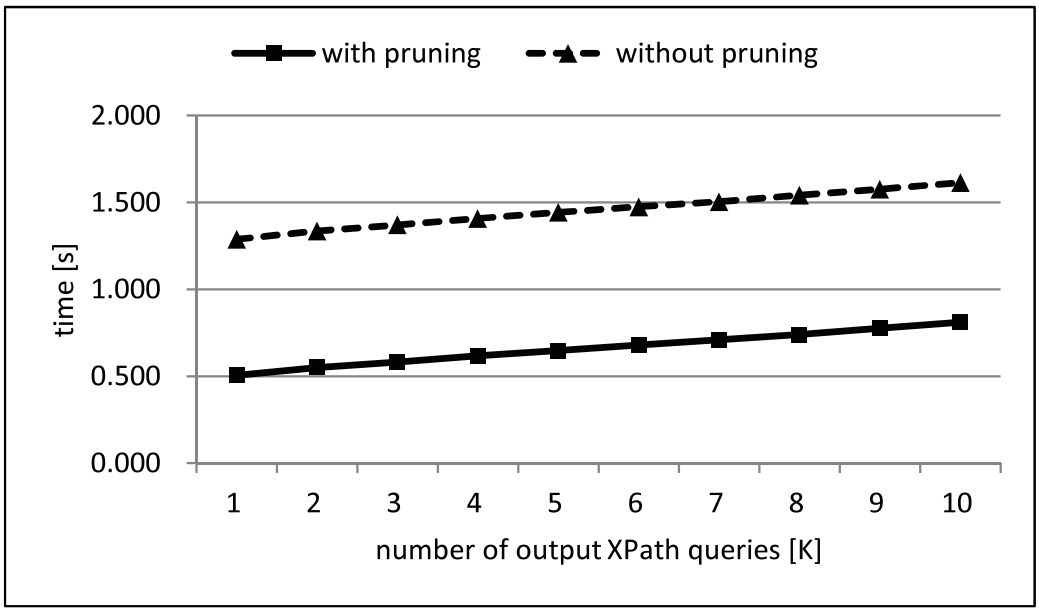


Figure 8.2: Execution time with/without pruning of the algorithm for queries targeting "far" nodes

with the same cost function of the previous experiment and measure its execution time.

Figure 8.2 plots the average execution times for Q_1, and Fig. 8.3 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 execution 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.

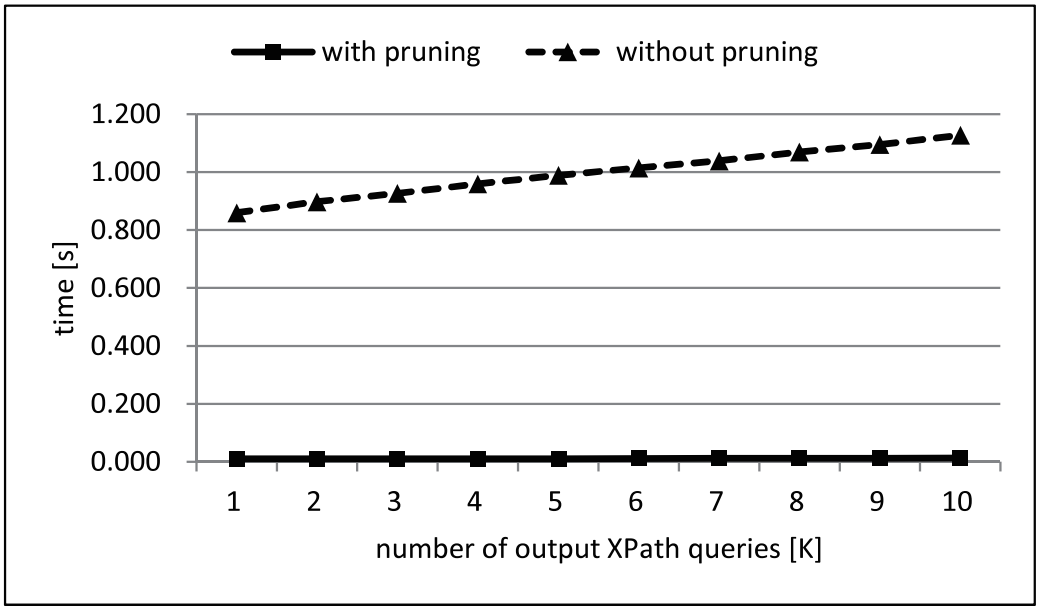


Figure 8.3: Execution time with/without pruning of the algorithm for queries targeting “near” nodes

Table 8.2: 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

Chapter 9

Discussion

This chapter discusses about the XPath query correction problem, mainly from a point of view of computational complexity and efficiency.

9.1 Edit Operation and Intractability

The complexity of the problem depends on the edit operations applied to XPath queries; "core" and "core + extended". The problem is tractable for the former edit operations, while the problem becomes NP-hard for the latter edit operations even if only simple XPath queries are allowed. The main difference between the former and the latter is whether location step exchange is allowed or not. Here, let us consider the reason why allowing location step exchange considerably increases the complexity of the problem. In Theorem 1, the NP-hardness of the problem in the case where location step exchange is allowed is shown by reducing the Hamilton path problem to the XPath query correction problem. The hardness of the Hamilton path problem comes from the fact that for a given graph G, a node ordering of G that brings a Hamilton path is hard to find. The proof of the theorem means that, by allowing location step exchange, the number of possible orderings of location steps may increase exponentially, which implies that finding an optimum valid query becomes considerably hard. Users that do not fully understand

the structure of a schema tend to write invalid queries in which some of the location steps are incorrectly interchanged. In this sense, location step exchange is a useful edit operation. However, one location step exchange can be simulated by a pair of a location step deletion and a location step insertion, although the latter cost does not always coincide with the former cost. In addition, the algorithm proposed in this dissertation presents top-K valid queries to users rather than a single valid query. This suggests that restricting the available set of edit operations to "core" is not too restrictive from a practical point of view.

9.2 Algorithm and Complexity

Let us next consider the time complexity of the algorithm, in terms of the expressive power of schema. As shown in Tables 9.1 and 9.2, the time complexity of the algorithm under DTD is almost equivalent to the complexity under regular tree grammar. According to [51], there are three major classes of schema languages; local tree grammar, single-type tree grammar, and regular tree grammar. Regular tree grammar corresponds to RELAX NG [54], single-type tree grammar corresponds to W3C XML Schema [79, 80], and local tree grammar corresponds to DTD. It is shown that the expressive power of regular tree grammar is strictly larger than that of single-type tree grammar, and the expressive power of single-type tree grammar is strictly larger than that of local tree grammar. Although there is a significant gap between regular tree grammar and DTD in terms of tree grammar, the expressive power of schema hardly affects the time complexity of the algorithm.

Let us consider the time complexity of the algorithm further. Assuming that the set of available edit operations is restricted to "core", the algorithm finds, for a schema S, a (possibly invalid) XPath query q, and a positive integer K, top-K queries syntactically close to q. Firstly, if XPath queries are restricted to be simple, then the algorithm runs in time polynomial of the sizes of S and q. On the

other hand, if XPath queries in XP are allowed, then the algorithm runs in time polynomial of the sizes of S and q in the case where the nest level of predicates of q is bounded by a constant, while the running time becomes exponential if the nest level of predicates of q is unrestricted. To see why the nest level of predicates affects the complexity of the algorithm, let us consider the following query

$$/ls_1[exp_1]/\cdots/ls_n[exp_n],$$

where ls_i is a location step and exp_i is a predicate. Note that if the last label of ls_i is changed to another label, then corrections to predicate exp_i becomes completely different. Therefore, corrections to a predicate exp_i are affected by the corrections to the location step ls_i that holds exp_i. This is the reason why the complexity of the algorithm increases exponentially to the nest level of predicates. It is open whether the problem is tractable or not in the case where arbitrary nest level of predicate is allowed, and to investigate the (in)tractability is an interesting problem.

The results of the experimental evaluations show the efficiency of the algorithm. In Chapter 8, it is shown that the algorithm runs highly efficiently, although the evaluations are done under only XMark auction.dtd. However, auction.dtd used in the evaluations is a relatively large DTD. Actually, auction.dtd is the sixth largest DTD among the 27 real-world DTDs listed in [34]. Moreover, auction.dtd contains cycles. These imply that the algorithm runs efficiently for most of real-world DTDs. However, more experiments should be done under schemas other than auction.dtd, which is left as a future work.

9.3 Boundary of Tractability and Intractability

As shown in Tables 9.1 and 9.2, the difference between “core” and “core + extended” is found to be the major boundary for the complexity of the XPath query correction problem. In the former case, the problem is tractable for many cases,

Table 9.1: The complexity of the XPath query correction problem under DTD

Edit operation	Query class		
	simple	XP	
		$mnl(q) \leq$ constant	general case
core	PTIME (Theorem 4)	PTIME (Theorem 6)	$O(K \cdot \|q\| \cdot \|\Sigma_e\|^{2+mnl(q)} \cdot \log(\|q\| \cdot \|\Sigma_e\|))$ (Theorem 6)
core + extended	NP-hard (Theorems 1 and 2)	NP-hard (Theorems 1 and 2)	

while in the latter case the problem becomes intractable even for simple XPath queries. More specifically, in the former case the problem can be solved efficiently if a query is simple or in XP with bounded nest level of predicates, but the running time of the algorithm grows exponentially if the nest level is not bounded by any constant. On the other hand, the complexity of the problem is shown to be hardly affected by the expressive power of schema.

Finally, there are still some problems that need to be investigated. First, it is open whether the problem is tractable in the case where a query is in XP and the nest level of predicates is not bounded. Second, XP does not allow upward axes such as parent and ancestor-or-self. Investigating the complexity of the problem for broader XPath classes allowing such axes is also left as a future work.

Table 9.2: The complexity of the XPath query correction problem under regular tree grammar

Edit operation	Query class		
	simple	XP	
		$mnl(q) \leq$ constant	general case
core	PTIME (Theorem 7)	PTIME (Theorem 8)	$O(K \cdot \|q\| \cdot \|P\|^{2+mnl(q)} \cdot \log(\|q\| \cdot \|P\|))$ (Theorem 8)
core + extended	NP-hard (Theorems 1 and 2)	NP-hard (Theorems 1 and 2)	

Chapter 10

Conclusion

In this dissertation, we firstly proposed two classes of XPath queries and two classes of edit operations to XPath queries. We considered the intractability of the XPath query correction problem in terms of the classes of edit operations and the classes of XPath query. Then we proposed an algorithm that finds, for a query q, a schema S, and a positive integer K, top-K queries syntactically close to q under S. Experimental results suggested that the algorithm outputs "correct" answers efficiently in many cases.

The results of this dissertation are summarized as follows. First, the extended edit operations clearly influence the complexity of the query correction problem. However, the core edit operations are still useful because the edit operations are complete even without the extended edit operations. Next, the complexity of the query correction problem remains the same under DTD and regular tree grammar. Thus the expressive power of schema does not affect the complexity of this problem. Finally, for the XPath query classes, the XPath queries in XP can be corrected efficiently in many cases if the core edit operations are allowed. However, whether this problem can be solve efficiently or not in the case where any nest level of predicate is allowed is open, and identifying the complexity is left as a future work.

In addition, to apply this research in practical application areas, several ex-

tensions to the algorithms are desired. First, to output useful candidate queries, using not only schema but also some part of data is especially effective for correcting and recommending predicates. For example, an incorrect numerical value in a predicate can be corrected by referring data values in the places indicated by the predicate. Next, cost settings of edit operations directly influence output queries and user experience. Therefore, for optimizing costs of edit operations, some learning mechanism based on user feedback are required to collect users' errors and analyze the tendency of the errors.

Acknowledgement

The author would like to give his sincere thanks, first of all, Associate Professor Nobutaka Suzuki who graciously supported and guided the author. Without him, the author did not aim to become a researcher.

The author is grateful to Professor Atuyuki Morishima and Professor Tetsuji Satoh, both of whom shared their time and knowledge with him. Their advice will continue to live in his future.

The author would like to thank Professors Shigeo Sugimoto and Associate Professor Toshiyuki Amagasa both of whom provided many essential and beneficial comments their reviews of his dissertation.

The author also thank to all the members of nslab, mlab and hitslab. They usually encouraged and gave pleasure to him.

Furthermore, the author thank to his colleagues in Chiba university. Thanks to their support, the author had completed this dissertation.

Finally, the author express his deepest thanks to his parents and his friends for their infinite encouragement and kindness.

Bibliography

[1] Alon, N., Milo, T., Neven, F., Suciu, D. and Vianu, V.: Typechecking XML Views of Relational Databases, *ACM Transactions on Computational Logic*, Vol. 4, No. 3, pp. 315–354 (2003).

[2] Alon, N., Milo, T., Neven, F., Suciu, D. and Vianu, V.: XML with data values: typechecking revisited, *Journal of Computer and System Sciences*, Vol. 66, No. 4, pp. 688–727 (2003).

[3] ALTOVA: XMLSpy. `http://www.altova.com/jp/xmlspy.html`.

[4] Amer-Yahia, S., Cho, S., Lakshmanan, L. V. S. and Srivastava, D.: Minimization of Tree Pattern Queries, *In Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD 2001)*, ACM, pp. 497–508 (2001).

[5] Amer-Yahia, S., Cho, S. and Srivastava, D.: Tree Pattern Relaxation, *In Proceedings of the 8th International Conference on Extending Database Technology (EDBT 2002)*, Springer Berlin Heidelberg, pp. 89–102 (2002).

[6] Amer-Yahia, S., Lakshmanan, L. V. and Pandit, S.: FleXPath: Flexible structure and full-text querying for XML, *In Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD 2004)*, ACM, pp. 83–94 (2004).

[7] Angluin, D.: Learning Regular Sets from Queries and Counterexamples, *Information and Computation*, Vol. 75, No. 2, pp. 87–106 (1987).

[8] Apache Software Foundation: Xalan. `http://xalan.apache.org/`.

[9] Arora, S., Lund, C., Motwani, R., Sudan, M. and Szegedy, M.: Proof Verification and the Hardness of Approximation Problems, *Journal of the ACM*, Vol. 45, No. 3, pp. 501–555 (1998).

[10] Benedikt, M., Fan, W. and Geerts, F.: XPath Satisfiability in the Presence of DTDs, *In the Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2005)*, ACM, pp. 25–36 (2005).

[11] Benedikt, M., Fan, W. and Geerts, F.: XPath Satisfiability in the Presence of DTDs, *Journal of the ACM*, Vol. 55, No. 2, pp. 8:1–8:79 (2008).

[12] Böckenhauer, H.-J. and Bongartz, D.: *Algorithmic Aspects of Bioinformatics (Natural Computing Series)*, Springer-Verlag Berlin Heidelberg (2007).

[13] Brodianskiy, T. and Cohen, S.: Self-correcting Queries for Xml, *In Proceedings of the 16th ACM Conference on Conference on Information and Knowledge Management (CIKM 2007)*, ACM, pp. 11–20 (2007).

[14] Chen, Y., Wang, W., Liu, Z. and Lin, X.: Keyword Search on Structured and Semi-structured Data, *In Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD 2009)*, ACM, pp. 1005–1010 (2009).

[15] Choi, B.: What are real DTDs like?, *In Proceedings of the 5th International Workshop on the Web and Databases (WebDB 2002)*, pp. 43–48 (2002).

[16] Cohen, S. and Brodianskiy, T.: Correcting queries for XML, *In Proceedings of the 16th ACM Conference on Conference on Information and Knowledge Management (CIKM 2007)*, Vol. 34, No. 8, pp. 690–710 (2009).

[17] Demaine, E. D., Mozes, S., Rossman, B. and Weimann, O.: An Optimal Decomposition Algorithm for Tree Edit Distance, *ACM Transactions on Algorithms*, Vol. 6, No. 1, pp. 2:1–2:19 (2009).

[18] Eppstein, D.: Finding the k shortest paths, *SIAM Journal on Computing*, Vol. 28, No. 2, pp. 652–673 (1998).

[19] Fan, J., Li, G. and Zhou, L.: Interactive SQL query suggestion: Making databases user-friendly, *In Proceedings of the IEEE 27th International Conference on Data Engineering (ICDE 2011)*, IEEE Computer Society, pp. 351–362 (2011).

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

[21] Fazzinga, B., Flesca, S. and Pugliese, A.: Retrieving XML Data from Heterogeneous Sources Through Vague Querying, *ACM Transactions on Internet Technology*, Vol. 9, No. 2, pp. 7:1–7:35 (2009).

[22] Feng, D.-F. and Doolittle, R. F.: Progressive sequence alignment as a prerequisite to correct phylogenetic trees, *Journal of Molecular Evolution*, Vol. 25, No. 4, pp. 351 – 60 (1987).

[23] Flesca, S., Furfaro, F. and Masciari, E.: On the Minimization of Xpath Queries, *In Proceedings of the 29th International Conference on Very Large Data Bases (VLDB 2003)*, ACM, pp. 153–164 (2003).

[24] Franceschet, M.: XPathMark: An XPath Benchmark for the XMark Generated Data, *In Proceedings of the 3rd International XML Database Symposium (XSym 2005)*, Springer Berlin Heidelberg, pp. 129–143 (2005).

[25] Gao, X., Xiao, B., Tao, D. and Li, X.: A survey of graph edit distance, *Pattern Analysis and Applications*, Vol. 13, No. 1, pp. 113–129 (2010).

[26] Garey, M. R. and Johnson, D. S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman (1979).

[27] Geerts, F. and Fan, W.: Satisfiability of XPath Queries with Sibling Axes, *In Revised Selected Papers of the 10th International Workshop on Database Programming Languages (DBPL 2005)*, Springer Berlin Heidelberg, pp. 122–137 (2005).

[28] Genevès, P. and Layaïda, N.: A System for the Static Analysis of XPath, *ACM Transactions on Information Systems*, Vol. 24, No. 4, pp. 475–502 (2006).

[29] Higgins, D. G., Thompson, J. D. and Gibson, T. J.: Using CLUSTAL for multiple sequence alignments, *Computer Methods for Macromolecular Sequence Analysis*, Methods in Enzymology, Vol. 266, Academic Press, pp. 383 – 402 (1996).

[30] Horie, K. and Suzuki, N.: Extracting Differences Between Regular Tree Grammars, *In Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC 2013)*, ACM, pp. 859–864 (2013).

[31] Ishihara, Y., Hashimoto, K., Shimizu, S. and Fujiwara, T.: XPath Satisfiability with Downward and Sibling Axes is Tractable Under Most of Real-world DTDs, *In Proceedings of the 12th International Workshop on Web Information and Data Management (WIDM 2012)*, ACM, pp. 11–18 (2012).

[32] Ishihara, Y., Morimoto, T., Shimizu, S., Hashimoto, K. and Fujiwara, T.: A Tractable Subclass of DTDs for XPath Satisfiability with Sibling Axes, *In Proceedings of the 12th International Symposium on Database Programming Languages (DBPL 2009)*, Springer Berlin Heidelberg, pp. 68–83 (2009).

[33] Ishihara, Y., Shimizu, S. and Fujiwara, T.: Extending the Tractability Results on XPath Satisfiability with Sibling Axes, *In Proceedings of the 7th International XML Database Symposium (XSym 2010)*, Springer Berlin Heidelberg, pp. 33–47 (2010).

[34] Ishihara, Y., Suzuki, N., Hashimoto, K., Shimizu, S. and Fujiwara, T.: XPath Satisfiability with Parent Axes or Qualifiers Is Tractable under Many of Real-World DTDs, *In Proceedings of the 14th International Symposium on Database Programming Languages (DBPL 2013)* (2013). `http://arxiv.org/abs/1308.0769`.

[35] Ives, Z. G., Halevy, A. Y. and Weld, D. S.: An XML query engine for network-bound data, *VLDB Journal*, Vol. 11, No. 4, pp. 380–402 (2002).

[36] Karchmer, M., Newman, I., Saks, M. and Wigderson, A.: Non-deterministic Communication Complexity with Few Witnesses, *Journal of Computer and System Sciences*, Vol. 49, No. 2, pp. 247–257 (1994).

[37] Kay, M. H.: SAXON. `http://saxon.sourceforge.net/`.

[38] Leonardi, E., Hoai, T. T., Bhowmick, S. S. and Madria, S.: DTD-Diff: A Change Detection Algorithm for DTDs, *In Proceedings of the 11th International Conference on Database Systems for Advanced Applications (DASFAA 2006)*, Springer Berlin Heidelberg, pp. 817–827 (2006).

[39] Leonardi, E., Hoai, T. T., Bhowmick, S. S. and Madria, S.: DTD-Diff: A change detection algorithm for DTDs, *Data & Knowledge Engineering*, Vol. 61, No. 2, pp. 384 – 402 (2007).

[40] Levenshtein, V. I.: Binary codes capable of correcting deletions, insertions and reversals, *Soviet Physics Doklady*, Vol. 10, pp. 707–710 (1966).

[41] Li, G., Feng, J., Wang, J. and Zhou, L.: Effective keyword search for valuable lcas over XML documents, *In Proceedings of the 16th ACM Conference*

on Conference on Information and Knowledge Management (CIKM 2007), ACM, pp. 31–40 (2007).

[42] Li, Y., Yu, C. and Jagadish, H. V.: Schema-Free XQuery, *In Proceedings of the 30th International Conference on Very Large Data Bases (VLDB 2004)*, ACM, pp. 72–83 (2004).

[43] Li, Y., Yu, C. and Jagadish, H. V.: Enabling Schema-Free XQuery with meaningful query focus, *VLDB Journal*, Vol. 17, pp. 355–377 (2008).

[44] Martens, W. and Neven, F.: Frontiers of Tractability for Typechecking Simple XML Transformations, *In Proceedings of the 23rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2004)*, ACM, pp. 23–34 (2004).

[45] Martens, W. and Neven, F.: On the Complexity of Typechecking Top-down XML Transformations, *Theoretical Computer Science*, Vol. 336, No. 1, pp. 153–180 (2005).

[46] Martins, E.: K-th Shortest Paths Problem. `http://www.mat.uc.pt/~eqvm/OPP/KSPP/KSPP.html`.

[47] Marzal, A. and Vidal, E.: Computation of Normalized Edit Distance and Applications, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 15, pp. 926–932 (1993).

[48] Miklau, G. and Suciu, D.: Containment and Equivalence for a Fragment of XPath, *Journal of the ACM*, Vol. 51, No. 1, pp. 2–45 (2004).

[49] Montazerian, M., Wood, P. T. and Mousavi, S. R.: XPath Query Satisfiability is in PTIME for Real-world DTDs, *In Proceedings of the 5th International XML Database Symposium (XSym 2007)*, Springer Berlin Heidelberg, pp. 17–30 (2007).

[50] Morishima, A., Kitagawa, H. and Matsumoto, A.: A machine learning approach to rapid development of XML mapping queries, *In Proceedings of the 20th IEEE International Conference on Data Engineering (ICDE 2004)*, pp. 276–287 (2004).

[51] Murata, M., Lee, D., Mani, M. and Kawaguchi, K.: Taxonomy of XML Schema Languages Using Formal Language Theory, *ACM Transactions on Internet Technology*, Vol. 5, No. 4, pp. 660–704 (2005).

[52] Neven, F. and Schwentick, T.: XPath Containment in the Presence of Disjunction, DTDs, and Variables, *In Proceedings of the 9th International Conference on Database Theory (ICDT 2003)*, Springer Berlin Heidelberg, pp. 315–329 (2003).

[53] Nguyen, K. and Cao, J.: Exploit Keyword Query Semantics and Structure of Data for Effective XML Keyword Search, *In Proceedings of the 21st Australasian Conference on Database Technologies (ADC 2010)*, Australian Computer Society, Inc., pp. 133–140 (2010).

[54] OASIS: RELAX NG (Clark, J. and Murata, M., Eds.). `https://www.oasis-open.org/committees/relax-ng/spec-20011203.html`.

[55] Pawlik, M. and Augsten, N.: RTED: A Robust Algorithm for the Tree Edit Distance, *Proceedings of the VLDB Endowment*, Vol. 5, No. 4, pp. 334–345 (2011).

[56] Ramanan, P.: Efficient Algorithms for Minimizing Tree Pattern Queries, *In Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD 2002)*, ACM, pp. 299–309 (2002).

[57] Schenkel, R. and Theobald, M.: Feedback-Driven Structural Query Expansion for Ranked Retrieval of XML Data, *In Proceedings of the 10th In-*

ternational Conference on Extending Database Technology (EDBT 2006), Springer Berlin Heidelberg, pp. 331–348 (2006).

[58] Schlieder, T.: Schema-driven evaluation of approximate tree-pattern queries, *In Proceedings of the 8th International Conference on Extending Database Technology (EDBT 2002)*, Springer Berlin Heidelberg, pp. 514–532 (2002).

[59] Schmidt, A., Waas, F., Kersten, M., Carey, M. J., Manolescu, I. and Busse, R.: XMark: A Benchmark for XML Data Managemet, *In Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002)*, VLDB Endowment, pp. 974–085 (2002).

[60] Schulz, U. K. and Mihov, S.: Fast string correction with Levenshtein automata, *International Association for Pattern Recognition*, Vol. 5, No. 1, pp. 67–85 (2002).

[61] Schwentick, T.: XPath Query Containment, *ACM SIGMOD Record*, Vol. 33, No. 1, pp. 101–109 (2004).

[62] Smith, T. and Waterman, M.: Identification of common molecular subsequences, *Journal of Molecular Biology*, Vol. 147, No. 1, pp. 195 – 197 (1981).

[63] SoftTree Technologies: SoftTree SQL Assistant. `http://www.softtreetech.com/isql.htm`.

[64] Sugimura, K., Ishihara, Y. and Fujiwara, T.: Proposal and Evaluation of Polynomial-time Algorithms for Deciding XPath Satisfiability, *IPSJ Journal*, Vol. 57, No. 5, pp. 1477–1488 (2016). (in Japanese with English Abstract).

[65] Suzuki, N.: On Finding an Edit Script between an XML Document and a DTD, *IPSJ Digital Courier*, Vol. 2, pp. 813–825 (2006).

[66] Suzuki, N.: Finding K Optimum Edit Scripts between an XML Document and a RegularTree Grammar, *In Proceedings of the 1st Workshop on Emerging Research Opportunities for Web Data Management (EROW 2007)*, pp. 81–95 (2007).

[67] Suzuki, N.: Satisfiability of Simple Xpath Fragments Under Fixed DTDs, *In Proceedings of the 28th British National Conference on Databases (BNCOD 2011)*, Springer Berlin Heidelberg, pp. 194–208 (2011).

[68] Suzuki, N. and Fukushima, Y.: Satisfiability of Simple Xpath Fragments in the Presence of DTDs, *In Proceedings of the 11th ACM International Workshop on Web Information and Data Management (WIDM 2009)*, ACM, pp. 15–22 (2009).

[69] Suzuki, N., Fukushima, Y. and Ikeda, K.: Satisfiability of Simple XPath Fragments under Duplicate-Free DTDs, *IEICE Transactions on Information and Systems*, Vol. 96, No. 5, pp. 1029–1042 (2013).

[70] Tai, K.-C.: The Tree-to-Tree Correction Problem, *Journal of the ACM*, Vol. 26, No. 3, pp. 422–433 (1979).

[71] Termehchy, A. and Winslett, M.: Effective, Design-independent XML Keyword Search, *In Proceedings of the 18th ACM Conference on Conference on Information and Knowledge Management (CIKM 2009)*, ACM, pp. 107–116 (2009).

[72] Termehchy, A. and Winslett, M.: Using structural information in XML keyword search effectively., *ACM Transactions on Database Systems*, Vol. 36, No. 1, p. 4 (2011).

[73] Thomas H. Cormen, Charles E. Leiserson, R. L. R. and Stein, C.: *Introduction to Algorithms, Third Edition*, The MIT Press (2009).

[74] Ukkonen, E.: On approximate string matching, *Foundations of Computation Theory: Proceedings of the 1983 International FCT-Conference*, Springer Berlin Heidelberg, pp. 487–495 (1983).

[75] Waterman, M. S. and Eggert, M.: A new algorithm for best subsequence alignments with application to tRNA-rRNA comparisons, *Journal of Molecular Biology*, Vol. 197, No. 4, pp. 723 – 728 (1987).

[76] Welty, C.: Correcting user errors in SQL, *International Journal of Man-Machine Studies*, Vol. 22, No. 4, pp. 463 – 477 (1985).

[77] Wood, P. T.: Minimising Simple XPath Expressions, *In Proceedings of the 4th International Workshop on the Web and Databases (WebDB 2001)*, pp. 13–18 (2001).

[78] Wood, P. T.: *Containment for XPath Fragments under DTD Constraints*, pp. 300–314, Springer Berlin Heidelberg (2003).

[79] World Wide Web Consortium: W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures (Gao, S., Sperberg-McQueen, C. M. and Thompson, H. S., Eds.). `https://www.w3.org/TR/xmlschema11-1/`.

[80] World Wide Web Consortium: W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes (Peterson, D., Gao, S., Malhotra, A., Sperberg-McQueen, C. M. and Thompson, H. S., Eds.). `https://www.w3.org/TR/xmlschema11-2/`.

[81] World Wide Web Consortium: XML Path Language (XPath) (Clark, J. and DeRose, S., Eds.). `http://www.w3.org/TR/xpath`.

[82] World Wide Web Consortium: XML Query Language (XQuery) (Boag, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J and Siméon, J., Eds.). `https://www.w3.org/TR/xquery/`.

[83] World Wide Web Consortium: XSL Transformations (XSLT) (Clark, J., Ed.). `http://www.w3.org/TR/xslt`.

[84] Xu, Y. and Papakonstantinou, Y.: Efficient keyword search for smallest LCAs in XML databases, *In Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD 2005)*, ACM, pp. 527–538 (2005).

[85] Zhang, K. and Shasha, D.: Simple Fast Algorithms for the Editing Distance between Trees and Related Problems, *SIAM Journal on Computing*, Vol. 18, No. 6, pp. 1245–1262 (1989).

[86] Zhang, K., Statman, R. and Shasha, D.: On the Editing Distance Between Unordered Labeled Trees, *Information Processing Letters*, Vol. 42, No. 3, pp. 133–139 (1992).

Full List of Publications

Journal Papers (refereed)

1. Suzuki, N., Fukushima, Y. and Ikeda, K.: Satisfiability of Simple XPath Fragments under Duplicate-Free DTDs, *IEICE Transactions on Information and Systems*, Vol. E96-D, No. 5, pp. 1029–1042 (2013).

2. Ikeda, K. and Suzuki, N.: An Algorithm for Finding top-K Valid XPath Queries, *IPSJ Transactions on Databases*, Vol. 7, No. 2, pp. 70–82 (2014).

3. Suzuki, N., Ikeda, K. and Kwon, Y.: An Algorithm for All-Pairs Regular Path Problem on External Memory Graphs, *IEICE Transactions on Information and Systems*, Vol. E99-D, No. 4, pp. 944-958 (2016).

International Conference Papers (refereed)

1. Ikeda, K. and Suzuki, N.: An Algorithm for Finding K Correct XPath Expressions, *In Proceedings of the 3rd International Workshop with Mentors on Databases, Web and Information Management for Young Researchers (iDB Workshop 2011)*, 10p (2011).

2. Ikeda, K. and Suzuki, N.: Finding top-K Correct XPath Queries of User's Incorrect XPath Query, *In Proceedings of the 23rd International Conference on Database and Expert Systems Applications (DEXA 2012)*, pp. 116–130 (2012).

3. Hasegawa, K., Ikeda, K. and Suzuki, N.: An Algorithm for Transforming XPath Expressions According to Schema Evolution, *In Proceedings of the First International Workshop on Document Changes: Modelling Detection, Storage and Visualization (co-located with ACM DocEng 2013)*, 8p (2013).

4. Suzuki, N., Ikeda, K. and Kwon, Y.: An External Memory Algorithm for All-Pairs Regular Path Problem, *In Proceedings of the 26th International Conference on Database and Expert Systems Applications (DEXA 2015)*, pp. 399–414 (2015).

5. Sakurai, E., Morishima, A., Ikeda, K. and N, Suzuki.: Bookshelf Problem: A Human-in-the-Loop Approach for Data Grouping without Complete Information, *In Proceedings of iConference 2016*, 10p (2016).

6. Ikeda, K., Morishima, A., Rahman, H., Roy, S. B., Thirumuruganathan, S., Amer-Yahia, S. and Das, G.: Collaborative Crowdsourcing with Crowd4U, *In Proceedings of the 42nd International Conference on Very Large Data Bases (VLDB 2016)*, 4p (2016)(accepted).