

# An Efficient Algorithm for Tree Pattern Query Minimization under Broad Integrity Constraints

Dunren Che

*Email: dche@cs.siu.edu*

*Department of Computer Science*

*Southern Illinois University*

*Carbondale, IL, 62901, USA*

**Abstract: Purpose** – Tree pattern is at the core of XML queries. The tree patterns in XML queries typically contain redundancies, especially when broad integrity constraints (ICs) are present and considered. Apparently, tree pattern minimization has great significance for efficient XML query processing. Although various minimization schemes/algorithms have been proposed, none of them can exploit broad ICs for thoroughly minimizing the tree patterns in XML queries. The purpose of this research is to develop an innovative minimization scheme and provide a novel implementation algorithm.

**Design/methodology/approach** – Query augmentation/expansion was taken as a necessary first-step by most prior approaches to acquire XML query pattern minimization under the presence of certain ICs. The adopted augmentation/expansion is also the curse for the typical  $O(n^4)$  time-complexity of the proposed algorithms. This article presents an innovative approach called *allying* to effectively circumvent the otherwise necessary augmentation step and to retain the time complexity of the implementation algorithm within the optimal, i.e.,  $O(n^2)$ . Meanwhile, the graph simulation concept is adapted and generalized to a three-tier definition scheme so that broader ICs are incorporated.

**Findings** – The innovative *allying* minimization approach is identified and an effective implementation algorithm named *AlliedMinimize* is developed. This algorithm is both runtime optimal – taking  $O(n^2)$  time – and most powerful in terms of the broadness of constraints it can exploit for XML query pattern minimization. Experimental study confirms the validity of the proposed approach and algorithm.

**Research limitations/implications** – Though the algorithm *AlliedMinimize* is so far the most powerful XML query pattern minimization algorithm, it does not incorporate all potential ICs existing in the context of XML. Effectively integrating this innovative minimization scheme into a full-fledged XML query optimizer remains to be investigated in the future.

**Practical implications** – In practice, *Allying* and *AlliedMinimize* can be used to achieve a kind of quick optimization for XML queries via fast minimization of the tree patterns involved in XML queries under broad ICs.

**Originality/value** – This article presents a novel scheme and an efficient algorithm for XML query pattern minimization under broad ICs.

**Keywords:** XML query minimization, XML query optimization, XML query processing, Tree pattern minimization, XML query, Tree pattern query

**Article Type:** Research paper

## Introduction

XML has emerged as the *de facto* standard for the representation and interchange of data/information especially on the World Wide Web. XML is not only well suited for marked-up documents and information sources but also for structured data (typically with a hierarchy), semistructured data, as well as the conventional relational data. Consequently, more and more data

are being formatted according to the XML standard. As a result, efficiently querying XML data has become a critical issue for many related applications including e-commerce, e-newspapers, and digital libraries, etc.

Traditionally, efficient execution of queries is acquired primarily through effective query optimization. Query optimization typically adopts the cost-based approach (Selinger, 1979) and aims at obtaining the least expensive query plan (chosen from a large number of alternate plans) for each input query. Heuristic knowledge may be additionally exploited in order to reduce the number of candidate plans that need to be examined, and thus to reduce the time spent on query optimization (We have demonstrated such a heuristic-based approach in our prior work (Che, 2006)).

XML data takes the tree structure, and an XML database consists of a forest of trees and thus is also called a *tree database*. In contrast to relational queries, XML queries have two integral dimensions of search criteria: the first dimension specifies the structural pattern of the target data (via a tree pattern) and the second dimension describes conditions about the contents/values of the searched data. Compared with relational data, XML data is relatively semantic-rich in terms of the structural relationships among the involved data elements. XML queries naturally carry this *semantic-rich* nature. Instead of being a challenge, the rich semantics (typically characterized by various constraints) represents an important asset of XML data for efficient query processing (Che, 2006a).

Due to the tree-structured nature of XML data, XML queries typically carry one or more tree patterns as their primary search criterion (including single-node tree patterns). XML queries are therefore called tree pattern queries (TPQs) by earlier researchers such as Amer-Yahia et al (Amer-Yahia, 2001) and Ramanan (Ramanan, 2002). The tree pattern involved in a TPQ is typically non-minimal because of possibly redundant *query nodes* (i.e., the nodes in the tree pattern) that repeatedly specify the same or an implied sub-criterion. Such redundancy can be generally classified as *syntactical redundancy* and *semantic redundancy*. Syntactical redundancy is caused by unnecessary query nodes; semantic redundancy is induced by certain integrity constraints (or ICs for short), which are either explicitly given as in a traditional database or extracted from an existing DTD/XSD (XML Schema Definition). Undoubtedly, TPQ minimization bears immediate consequences to the succeeding steps (including optimization and execution) in the course of XML query processing. In essence, TPQ minimization forms an integral part of XML query optimization. After a TPQ is first minimized, the (additional) succeeding optimization strategies can be conducted more efficiently. On the other hand, TPQ minimization itself can render much performance benefit for the TPQ's evaluation – this benefit is especially meaningful when an XML data repository/system does not have (or employ) a comprehensive query optimization module.

The focus of this article is on TPQ minimization in the presence of *broad* ICs because no prior work has ever been done that can adequately exploit broad ICs for the purpose of TPQ minimization.

Under the presence of ICs, query augmentation/expansion (for incorporating the effects of ICs) has been taken as the necessary first-step toward ultimate identification and removing of the redundant nodes in a TPQ, as done by Amer-Yahia et al (Amer-Yahia, 2001; Amer-Yahia, 2002), Ramanan (Ramanan, 2002), and Chen et al (Chen, 2006b). However, we observe that it is this augmentation step that resulted in the increase of the complexity of these algorithms (Amer-Yahia, 2001; Amer-Yahia, 2002; Ramanan, 2002; Chen, 2006b). Taking the *MinimizeChase* algorithm (Ramanan, 2002) as an example, in order to help identify redundant query nodes, the algorithm produces *Chase(Q)*, the augmented form of *Q*, resulting in the increase of the query's size from  $n$  to  $n^3$ , which in turn accounts for the final time complexity  $O(n^4)$  of the algorithm.

In this article, we introduce a novel, powerful mechanism, called *allying*, in order to facilitate identifying redundant query nodes in a TPQ and to avoid physically augmenting the query. The basic idea of *allying* is the following: First, the constraint graph  $G_C$  (which is a DAG) is derived from the given set  $C$  of ICs. Second, each query node in the input TPQ  $Q$  is “*allied*” with

a corresponding *type node* in the  $G_C$ . Those query nodes with *alliances* thus acquire increased *simulation power* from their alliances so that more potentially simulated nodes may be identified for these query nodes, without the need to physically augmenting the TPQs (A query node simulated by another query node is a potentially redundant node; this principle will be further explained later on). A much broader spectrum of ICs is incorporated by our allying scheme for TQP minimization. In result, we deliver a highly efficient and powerful algorithm for TPQ minimization. To the best of our knowledge, there are no other algorithms that are as efficient as ours and are as powerful as ours in terms of the broadness of ICs being exploited for TPQ minimization.

The remainder of this article is set forth as follows: Section 2 reviews related work. Section 3 introduces the data model and constraints that our scheme will explore. Section 4 presents our novel algorithm *AlliedMinimize* for TPQ minimization in the presence of broad ICs. In Section 5, we first walk through the application of our approach and algorithm to a couple of query examples for illustration purpose, then discuss an interesting case regarding the exploitation of the entrance location constraints, and finally show some experimental results of our algorithm. The article is concluded in Section 6.

## Related Work

Amer-Yahia et al did a piece of pioneer work on TPQ minimization (Amer-Yahia, 2001; Amer-Yahia, 2002). In the absence of any ICs, their proposed algorithm induces a runtime of  $O(n^4)$  with respect to the query size  $n$ ; when only three simple forms of constraints, i.e., required children, required descendants, and the subtype constraints, are considered, their algorithm takes  $O(n^6)$  time (actually, the algorithm does not always work correctly with the subtype constraints per Ramanan (Ramanan, 2002)). Based on the *graph simulation* concept tailored to the tree structures, Ramanan (Ramanan, 2002) proposed a few interesting algorithms that are more efficient than prior algorithms (Amer-Yahia, 2001; Amer-Yahia, 2002). His *TPQMinimize* algorithm takes  $O(n^2)$  time in the absence of any ICs, and his *MinimizeChase* algorithm takes  $O(n^4)$  time and works only with required children/descendants and the subtype constraints. Yet his third algorithm, *MinimizeCTPQ*, presented in the same paper (Ramanan, 2002), claims to be runtime optimal (i.e., taking  $O(n^2)$  time) but works with even lesser types of constraints (i.e., with only required children/descendants). Recently, the author and his colleague proposed a new IC-independent algorithm (Chen, 2006a) that takes  $O(n^2)$  time and a corresponding IC-present algorithm of  $O(n^3)$  time (Chen, 2006b) for TPQ minimization. In this article, the author make a further step forward by introducing a more efficient and more powerful scheme and an algorithm for TPQ minimization.

In addition to the work of Amer-Yahia et al (Amer-Yahia, 2001; Amer-Yahia, 2002) and that of Ramanan (Ramanan, 2002), which are most related to the author's research, the following are some more, generally related works.

Query minimization with or without ICs has been a fundamental problem in query processing and optimization, and was studied extensively under different data models. In essence, tree pattern queries are specialized conjunctive queries tailored to the domain of tree-structured data. The minimization of conjunctive queries based on the relational data model is known to be NP-complete (Florescu, 1998; Garey, 1979). The containment issue and XPath (Clark, 1999) query optimization in the presence of certain constraints were studied by Wood (Wood, 2001; Wood, 2003). Flesca et al (Flesca, 2003) investigated XPath query minimization, and particularly studied the issue with a special case of XPath queries, comprising the child/descendant, branch, and wildcard operators. They proved the so-called *global minimality* property of XPath queries that states that a minimum tree pattern equivalent to a given pattern  $p$  can be found among a sub-pattern of  $p$  and can be obtained by pruning the redundant branches within  $p$ . Based on this result, an algorithm for tree pattern minimization was developed that works in exponential time with regard to the tree pattern size. Lee et al (Lee, 2000) studied a variety of semantic constraints, explicitly or implicitly induced by the DTDs of XML data, while the emphasis was on the preservation of these constraints during transformation of

XML data into relational formats. Wood (Wood, 2000a; Wood, 2000b) studied the equivalence issue of XML queries between different forms of query patterns.

## Data Model, Constraints, and Definitions

In this section, we introduce the data model, the various constraints that our algorithm explores, and numerous important definitions that form the backbone of the approach and algorithms to be presented in the subsequent sections.

In our data model, XML data are represented as ordered trees, called *data trees*; each tree node represents an element (with an associated element type) and each edge in a tree represents an element-subelement relationship. A database built in this way consists of a collection (or a forest) of data trees, and is often referred to as a *tree database* (including XML databases and directory databases, etc). Various constraints may accompany a tree database. Some ICs can be derived from the accompanying DTD/XSD of the stored data if available, and some ICs may be specified separately as in a traditional database (in addition to the database schema) to make the semantics of the stored data complete and precise. In the sequel, both the term *IC* and the single-word term *constraint* may be used interchangeably.

In most of the query languages proposed for XML, including XPath (Clark, 1999) and XQuery (Boag, 2007), data querying is accomplished by binding the query variables to the *data nodes* in the XML tree database. An XML query typically specifies a tree-shaped search pattern (specifying the conditions on the structural part), associated with node predicates (specifying the conditions on the content/value part). XML queries thus are also referred to as tree pattern queries (TPQs) to highlight the importance of the tree structures for XML queries. The nodes in a TPQ are called *query nodes* (in contrast to the *data nodes* in a tree database). A query node is typically labelled using the element's type name chosen from an alphabet,  $\Sigma$ , which denotes the set of all type names allowed for a given database. We assume the common conventions as adopted in (Amer-Yahia, 2001; Ramanan, 2002), i.e., using simple names like  $u, v, w$ , or  $v1, v2, v3$  to identify the query nodes in a TPQ, while the corresponding type of each query node, for example,  $u$ , is denoted by  $\tau(u)$ . For TPQs, multiple query nodes may be associated with the same element type. For convenience, we assume a TPQ  $Q$  has a single output node, denoted by  $out(Q)$  in the text and '\*' in graphic illustrations (nonetheless, our method and the developed algorithm straightforwardly extend to the situation when multiple output nodes are desired). We differentiate two kinds of edges that may appear in a TPQ: *c-edge* (child edge) and *d-edge* (descendant edge). Accordingly, a c-edge connects a query node to a child node, called *c-child*, and a d-edge connects to a descendant node, called *d-child*. A c-edge from node  $u$  to node  $v$  is denoted by  $u \rightarrow v$  in the text and by a single line in graphic illustrations, and a d-edge from node  $u$  to node  $v$  is denoted by  $u \Rightarrow v$  in the text and by a double-line in graphic illustrations. A node  $v$  in a TPQ is said to be a *descendant* of another node  $u$  if there exists either a d-edge from  $u$  to  $v$  or a path from  $u$  to  $v$  comprising more than one edge (either c-edge or d-edge) in the TPQ. A child node is not treated as a proper descendant node in this article. A c-edge in a TPQ directly maps to the (data) edges (indicating a direct element-subelement relation) in a tree database, while a d-edge maps to the (data) paths (indicating an indirect element-subelement relation) in the database, comprising more than one data edge.

An *answer* to a TPQ  $Q$  over a tree database is obtained by an embedding of  $Q$  into the database. An *embedding* of  $Q$  into the database  $db$  is a mapping,  $\beta: Q \rightarrow db$ , which maps the query nodes in  $Q$  to the data nodes in  $db$  that satisfies the following conditions:

1. Preserve node types: For each node  $u \in Q$ ,  $u$  and  $\beta(u)$  must be associated to the same element type.

2. Preserve c-/d-edge relationships: If  $u \rightarrow v$  is in  $Q$ , then  $\beta(v)$  must be a child (i.e., a direct subelement) of  $\beta(u)$  in  $db$ ; and if  $u \Rightarrow v$  is in  $Q$ , then  $\beta(v)$  must be a descendant (i.e., indirect subelement) of  $\beta(u)$  in  $db$ .

An embedding could map several query nodes (of the same type) in a TPQ to the same data node in a database, and vice versa. Answering a query  $Q$  in a given database requires finding all possible embeddings of  $Q$  into the database. The result of a query  $Q$  is a set of data nodes, denoted by  $\beta(out(Q))$  and determined by all embeddings of  $Q$  into the database.

In general, the efficiency of finding the result of a TPQ depends on the size of the query (and of course on the size of the database as well). So, it is extremely important to minimize the TPQ before performing any further processing. As pointed out by Amer-Yahia et al (Amer-Yahia, 2001), a TPQ  $Q$  typically fails to be minimal because of the following two possible reasons:

1.  $Q$  may simply contain some redundant query nodes or redundant branches, and the removal of them does not affect the final result of the query. This kind of redundancy is the so-called *syntactic redundancy*.
2. When some ICs are present and considered, additional redundancies may be identified and can be removed without affecting the query's final result. This kind of redundancy is the so-called *semantic redundancy*.

In the following, we review the important constraints that have been identified for XML and are to be explored by our approach and algorithm.

1. **Required child:** Every data node (or element) of type  $t1$  must have a child node (or subelement) of type  $t2$ . This constraint is denoted by  $t1 \rightarrow t2$ .
2. **Required descendant:** Every node of type  $t1$  must have a descendant node of type  $t2$ . This constraint is denoted by  $t1 \Rightarrow t2$ .
3. **Subtype:** Every node of type  $t1$  must also be of type  $t2$ ; this constraint is denoted by  $t1 \leq t2$ . Obviously, subtype is a reflexive relation, i.e.,  $t \leq t$  holds for every type  $t \in \Sigma$ .
4. **Co-occurrence:** Whenever a node of type  $t1$  occurs, there is always a co-occurring sibling node of type  $t2$ . This constraint is denoted by  $co(t1, t2)$ . The co-occurrence relation is symmetric, i.e., if  $co(t1, t2)$ , then  $co(t2, t1)$ , and vice versa.
5. **Required parent:** Every node of type  $t1$  must have a parent node of type  $t2$ . This constraint is denoted by  $t1 \leftarrow t2$ .
6. **Required ancestor:** Every node of type  $t1$  must have an ancestor node of type  $t2$ . This constraint is denoted by  $t1 \Leftarrow t2$ . We do not accept  $t1 \leftarrow t2$  as a special case of  $t1 \Leftarrow t2$ .
7. **Entrance location** (also called **path constraint**): For a given DTD/XSD, element type  $t$  is said to be an *entrance location* (or *EL* for short) for element type  $t1$  and  $t2$  if in any document (data tree) complying with the DTD/XSD, all paths from every element of type  $t1$  to every element of type  $t2$  (toward the root) pass through an element of type  $t$ . This constraint is denoted by  $el(t, t1, t2)$ .

The above definition for entrance location just specifies a generic situation, i.e., without mentioning whether the intermediate type  $t$  is a parent or ancestor of type  $t1$  and whether it is a c-child or d-child of  $t2$ . When it comes to implementation, these specifics must be made explicit. But in this article, our interest is to present a generic approach and these specifics are omitted for simplicity.

The entrance location concept was first introduced by Böhm et al in (Böhm, 1998). A very interesting, special case of entrance location is when  $t2$  is designated to be the root (node), i.e.,  $el(t, t1, root)$ , or denoted by the short-hand notation  $el(t, t1)$ . What is actually meant by  $el(t, t1)$  is that  $t$  is a required parent or required ancestor of  $t1$ . So, the required parent and required ancestor constraints are in fact a special case of the EL constraints. Required child, required descendant, and subtype

are the three basic types of constraints examined by Amer-Yahia et al (Amer-Yahia 2001; Amer-Yahia 2002) and Ramanan (Ramanan, 2002) for TPQ minimization. The most efficient known algorithm *MinimizeCTPQ* (Ramanan, 2002) works only with the required child and the required descendant constraints. The co-occurrence constraints were discussed by Wood (Wood, 2001; Wood, 2003), where the constraints were referred to as *sibling constraints*. Wood (Wood, 1999) also considered the required parent constraints. Required ancestor constraints are a derived case (via repeated application) of required parents. Our goal in this article is different: we aim for a more powerful minimization scheme for TPQs. Distinguished from all previous work, our approach presented in this article is not only highly efficient but also most powerful in terms of broadness of the constraints (as outlined above) being exploited.

Our work furthers the prior, inspiring work of Ramanan (Ramanan, 2002). In the following, we review the *chase* technique (Ullman, 1989) that was adapted by Ramanan (Ramanan, 2002) for TPQ minimization. We then present a series of important concepts that form the basis of our approach.

Ramanan (Ramanan, 2002) adapted the classical chase technique (Ullman, 1989) for semantically expanding a TPQ as the first step toward minimizing it. As a result, the expanded TPQ incorporates the effects of relevant constraints, which leads to the discovery of the potential redundancy under the effect of these constraints. The procedure, *MinimizeChase*, adopted by Ramanan (Ramanan, 2002) works as follows: First, with a given set  $C$  of constraints (which can only be the required child/descendant and the subtype constraints),  $closure(C)$  is computed by means of a set of 7 expansion rules. Based on  $closure(C)$ , the expanded form of the query,  $chase(Q)$ , is then produced, which successfully incorporates the effects of the relevant constraints in  $C$ . Finally, all redundant nodes from the expanded form,  $chase(Q)$ , are identified and removed.

Our approach retains the 7 expansion rules proposed by Ramanan (Ramanan, 2002). In addition, we introduce 3 new rules for incorporating the effect of the co-occurrence constraints. The new set of totally 10 rules is exploited by our *allying* minimization approach for building a constraint graph (to be detailed later), but not for expanding the input TPQ. We list the 10 expansion rules below, of which the first 7 rules are quoted from (Ramanan, 2002):

1. If  $t1 \rightarrow t2$ , then add  $t1 \Rightarrow t2$
2. If  $t1 \Rightarrow t2$  and  $t2 \Rightarrow t3$ , then add  $t1 \Rightarrow t3$
3. If  $t1 \leq t2$  and  $t2 \leq t3$ , then add  $t1 \leq t3$
4. If  $t1 \leq t2$  and  $t2 \rightarrow t3$ , then add  $t1 \rightarrow t3$
5. If  $t1 \leq t2$  and  $t2 \Rightarrow t3$ , then add  $t1 \Rightarrow t3$
6. If  $t1 \rightarrow t2$  and  $t2 \leq t3$ , then add  $t1 \rightarrow t3$
7. If  $t1 \Rightarrow t2$  and  $t2 \leq t3$ , then add  $t1 \Rightarrow t3$
8. If  $co(t1,t2)$  and  $co(t2,t3)$ , then add  $co(t1,t3)$
9. If  $t1 \rightarrow t2$  and  $co(t2,t3)$ , then add  $t1 \rightarrow t3$
10. If  $t1 \Rightarrow t2$  and  $co(t2,t3)$ , then add  $t1 \Rightarrow t3$

After applying these rules to the constraint set  $C$ , the effects of the required child/descendant, subtype, and co-occurrence constraints are reflected in the closure of  $C$ , denoted by  $C^+$ , all in the form of required children/descendants. As a result, the required child and descendant, subtype, and co-occurrence constraints together form a DAG, referred to as *constraint graph*, denoted by  $G_C = (N_C, E_C)$ , where  $N_C$  is the node set  $\Sigma$  (which is the set of all element types under consideration), and  $E_C$  is the edge set.  $E_C$  may contain three types of edges: the c-edges and d-edges which respectively represent the required children and required descendents, and the subtype edges which can be symbolized by dashed directed lines, each linking to a supper type from a subtype (figuratively, we may say that the c-edges and d-edges are “vertical” and the subtype edges are “horizontal”).

The constraint graph  $G_C$  may not be a connected graph. However,  $G_C$  does not cover the entire constraint closure  $C^+$  because the EL constraints (if any) cannot be incorporated in to the graph. Therefore, a full representation of  $C^+$  should consist of the two parts, leading to the following more formal definition:

**Definition 1 (Constraint Closure)** Let  $C$  be a set of constraints,  $\text{closure}(C)$  is fully represented by  $C^+ = \langle G_C, EL_C \rangle$ , where  $G_C$  denotes the constraint graph  $G_C = (N_C, E_C)$ , and  $EL_C$  represents all EL (entrance location) constraints.

In implementation,  $EL_C$  can be stored in a hash table, of which every type name is mapped to a bucket and each bucket contains the pairs of types that the hashed type is an *entrance location* for them.

In the context of a tree database, ICs are typically specified at the schema level. Usually, for a given query, only certain constraints are relevant. We introduce the following concept to characterize the relevance of constraints to a given query:

**Definition 2 (Constraint Restriction)** Let  $C$  be a set of constraints,  $Q$  be a given TPQ. The restriction of  $C^+$  to  $Q$ , denoted as  $C^{+'} = \langle G'_C, EL'_C \rangle$  retains only those constrains from  $C^+$  that are relevant<sup>1</sup> to  $Q$ , where  $G'_C = (N'_C, E'_C)$  represents the restriction of  $G_C$  to  $Q$ , and  $EL'_C$  represents the restriction of  $EL_C$  to  $Q$ .

The node set  $N'_C$  consists of a subset of element types and their names are taken from  $\Sigma' \subseteq \Sigma$ . For a given query  $Q$  of size  $n$  (i.e., with  $n$  query nodes), an important issue here is how to characterize the size of  $\Sigma'$ . Ramanan (Ramanan, 2002) assumed that  $\Sigma'$  consists of the types common to both  $Q$  and  $C$ , and thus characterized the size of  $\Sigma'$  using the same size parameter  $n$  of  $Q$ . We want to point out that this assumption is wrong or at least inaccurate. For any node  $v \in Q$ , let  $t = \pi(v)$  and  $G'_C(t)$  denotes the sub-graph of  $G'_C$  rooted at  $t$ . Obviously the whole sub-graph  $G'_C(t)$  is relevant to  $Q$  and could be applied for expanding  $Q$ , but  $G'_C(t)$  may contain some type nodes that do not correspond to any of the original query nodes in  $Q$ . Therefore, the size  $m$  of  $G'_C$  is usually different from the size  $n$  of  $Q$ . The parameter  $m$  generally indicates the scale of the constraint set  $C$  but not the actual size of  $C$  (this is why we prefer to use the term “the scale of  $C$ ” instead of “the size of  $C$ ”). However, it makes sense to assume  $m \approx n$  because, on the one hand, multiple query nodes in  $Q$  may map to the same type node in  $G'_C$  and, on the other hand,  $G'_C$  may contain type nodes that do not have an occurrence in  $Q$ . Furthermore, assuming  $m \approx n$  is especially meaningful when comparing the performance of our algorithms with related ones that adopt a single parameter space such as (Amer-Yahia, 2001; Ramanan, 2002; Chen, 2006b).

We now introduce the important definitions that form the backbone of our approach and algorithm for TPQ minimization.

**Definition 3 (Equivalent Queries)** Let  $Q(D)$  denote the result of a query  $Q$  on a database  $D$ . We say query  $Q1$  is contained in query  $Q2$ , denoted by  $Q1 \subseteq Q2$ , if  $Q1(D) \subseteq Q2(D)$  for all databases  $D$ ; furthermore, we say  $Q1$  and  $Q2$  are equivalent, denoted by  $Q1 = Q2$ , if  $Q1 \subseteq Q2$  and  $Q2 \subseteq Q1$ .

**Definition 4 (Redundant Node)** Let  $Q$  be a query, and  $v$  a node in  $Q$ . We say  $v$  is redundant if after deleting the node  $v$ , the resultant query  $Q'$  is equivalent to the original query  $Q$ .

**Definition 5 (Minimal Query)** A query  $Q$  is minimal if no query of smaller size (in terms of the number of involved query nodes) is equivalent to  $Q$ .

---

<sup>1</sup> As an example, suppose  $t1 \rightarrow t2$  is in  $G_C$  but neither  $t1$  nor  $t2$  appears in  $Q$ , then  $t1 \rightarrow t2$  is not relevant to  $Q$  and not retained in  $G'_C$ , the restriction of  $G_C$  to  $Q$ .

The key problem in TPQ minimization is how to efficiently identify all the redundant query nodes in a TPQ. Once it has been done, eliminating redundant nodes from a TPQ is straightforward. Generally, if the effect of a query node in a TPQ is implied (or covered) by another query node in the TPQ, then the first node is redundant and can be removed. The graph simulation concept, as demonstrated by Ramanan (Ramanan, 2002), is a useful mechanism for identifying redundant nodes in TPQs. In the following, we redefine the simulation (relation) concept at three different tiers to facilitate the discovery of redundant nodes under broader ICs. We start with the base definition at tier 1, which is adapted from Ramanan's corresponding definition in (Ramanan, 2002).

**Definition 6 (Literal Simulation)** Let  $Q = (N, E)$  represent a TPQ with a set  $N$  of nodes and a set  $E$  of directed edges; each node  $u \in N$  is associated with its type  $\tau(u)$ . We say that node  $u \in N$  is **l-simulated** (literally simulated) by node  $v \in N$ , denoted by  $u \preceq_l v$  (accordingly,  $v$  is called an **l-simulator** of  $u$ ), whenever all the following conditions hold:

- (1). If  $u$  is the output node, then  $v = u$  (an output node is only simulated by itself and is never redundant.)
- (2). Preserve node types: i.e.,  $\tau(v) \leq \tau(u)$ , which says that the l-simulator  $v$  of  $u$  must have the same or a more specific type associated with it.
- (3). Preserve child edge relationships: if  $u$  has a c-child  $u'$ , then  $v$  has to have a c-child  $v'$  such that  $u' \preceq_l v'$ .
- (4). Preserve descendant edge relationships: if  $u$  has a d-child  $u''$ , then  $v$  has to have a d-child (or a descendant)  $v''$  such that  $u'' \preceq_l v''$ .

The above simulation concept is the graph simulation (relation) concept tailored to the tree structure of the XML data model. The simulation relation is defined among the query nodes in a TPQ; it, in fact, implies a substitution relation between two subtrees rooted at two corresponding query nodes that hold the simulation relation (and elimination of redundancy actually requires eliminating whole redundant subtrees in a TPQ). Our tier-1 definition of the simulation relation (Definition 6) considers only the structural relationships among the nodes in a TPQ (in a literal way). Our definition is an extension of Ramanan's corresponding definition (Ramanan, 2002) by incorporating the implication of the subtype constraints into the definition. Literal simulation forms the base of our three tier simulation scheme. Our tier-2 definition further extends the simulation concept by incorporating the effects of more ICs.

**Definition 7 (Semantic Simulation)** Let  $Q = (N, E)$  be a TPQ with a set  $N$  of nodes and a set  $E$  of directed edges; each node  $u \in N$  is associated with a corresponding type  $\tau(u)$ . We say that node  $u \in N$  is **s-simulated** (semantically simulated) by node  $v \in N$ , denoted as  $u \preceq_s v$  (accordingly,  $v$  is called an **s-simulator** of  $u$ ), whenever the following condition is satisfied:

- (1). If  $u \preceq_l v$ , i.e.,  $v$  is an l-simulator of  $u$ , then  $u \preceq_s v$ , or
- (2). If  $u$  is a leaf and  $co(\tau(v), \tau(u))$  holds (in this case we do not care whether  $v$  is a leaf), or
- (3). If  $u$  is a leaf child of  $w$ ,  $v$  is a d-child of  $w$ , and  $el(\tau(u), (\tau(v), \tau(w)))^2$ .

Obviously, as defined above, with a given TPQ, the s-simulation is a superset of the l-simulation, which means by considering the effects of certain ICs, more simulation relationships can be identified and more redundant nodes will be revealed (and eliminated). S-simulation incorporates two more types of constraints, co-occurrence and entrance location. The incorporation of co-occurrence in this concept is self-evident, and is limited to only the leaf level because non-leaf co-occurring nodes may have children/descendants that hardly bear any simulation relation.

---

<sup>2</sup> When  $u$  is a c-child of  $w$ , the entrance location  $\tau(u)$  must also be a c-child of  $\tau(w)$ ; accordingly, when  $u$  is a d-child of  $w$ ,  $\tau(u)$  must be a d-child of  $\tau(w)$ .

The incorporation of entrance locations into the s-simulation is illustrated in Figure 1. Figure 1(a) shows an input (sub-)query pattern; Assume  $\tau(u)$  is a c-child of  $\tau(w)$  and  $el(\tau(u), \tau(v), \tau(w))$  holds, i.e.,  $\tau(u)$  is a c-child entrance location for  $\tau(v)$  and  $\tau(w)$ . According to the entrance location definition (that implies that we can freely introduce a new query node into a query without changing the query’s result), Figure 1(a) is equivalent to Figure 1(b). Obviously, the node  $u$  on the left branch (Figure 1(b)) is simulated by the node  $u$  on the right branch. Therefore, Figure 1(b) can be equivalently rewritten as Figure 1(c). After removing the introduced entrance location node  $u$  from Figure 1(c), we get the resultant TPQ shown in Figure 1(d). However, when  $\tau(u)$  is a d-child entrance location for  $\tau(v)$  and  $\tau(w)$  and  $u$  is still a c-child of  $w$  (as shown in Figure 1(e)), the simulation relation between the two  $u$  nodes cannot be determined and the above minimization cannot be performed.

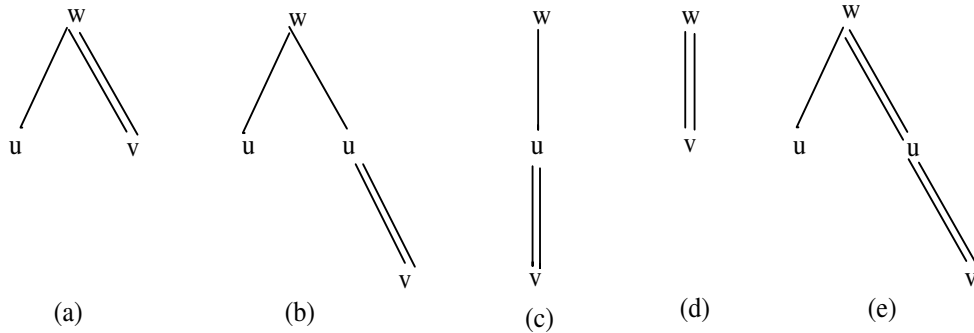


Figure 1. Illustration of the effect of entrance location on TPQ minimization.

As the required parent/ancestor constraints can be regarded as a special case of entrance location constraints, our s-simulation relation thus effectively incorporates the required parent/ancestor constraints into the 3-tier simulation framework.

Now we present the tier-3 definition for simulation – the *augmented simulation*.

**Definition 8 (Augmented Simulation)** Let  $Q = (N, E)$  be a TPQ with a set  $N$  of nodes and a set  $E$  of directed edges (each node  $u \in N$  is associated with a type  $\tau(u)$ ) and  $C$  be a set of ICs (More specifically, let  $C^{++} = \langle G'_C, EL'_C \rangle$  be the restriction of closure  $C^+$  to  $Q$  and  $G'_C = (N'_C, E'_C)$  be the restriction of the constraint graph  $G_C$  to  $Q$ ). We define an **a-simulator** (i.e., augmented simulator) of  $u \in N$  to be either an s-simulator of  $u$  found only in  $Q$  or an s-simulator of  $u$  found only in  $G'_C$ . An augmented simulation is denoted by  $u \preceq_a v$ , which indicates that  $v$  is an a-simulator of the query node  $u$ . Furthermore,  $v$  is called an **actual simulator** of  $u$  if  $u \preceq_a v$  and  $v \in N$ ; otherwise,  $v$  is called a **virtual simulator** of  $u$  if  $u \preceq_a v$  and  $v \in N'_C$ .

Definition 8 embodies the core technique, **allying**, pursued by our approach and algorithm (detailed in the next section). Intuitively, for a node  $u$  in a given TPQ  $Q$ , we want to identify all those query node(s) in  $Q$  that can simulate  $u$  and thus makes  $u$  redundant and removed. Often a node in  $Q$  has the potential of simulating another query node when the effect of certain constraints is present and considered. This potential is usually discovered after the query is augmented by incorporating the effects of the constraints as shown in (Amer-Yahia, 2001; Amer-Yahia, 2002; Ramanan, 2002). Because augmentation is the main reason that causes the runtime increase of these algorithms, we want to avoid physically augmenting a TPQ while the ultimate goal is to minimize it. *Allying* is an effective mechanism that allows a query node  $u$  “allied” with a corresponding type node  $\tau(u)$  in the relevant constraint graph  $G'_C$  and enable  $u$  to borrow extra “simulation power” from  $G'_C(\tau(u))$  for discovering more (potentially redundant) nodes simulated by  $u$ .

Our earlier definition (Definition 4) with regard to redundant nodes is declarative, but not procedural. That is, the definition does not tell us how to identify redundant nodes, which however must be answered when it comes to implementation. In the

following, we redefine the concept of *redundant nodes* from a different perspective and the goal is to facilitate algorithmic implementation.

**Definition 9 (Redundant Node)** *Let  $Q$  be a TPQ, any query node  $u \in Q$  is redundant if there exists an actual simulator  $v$  of  $u$  and  $v$  is a sibling node of  $u$  in  $Q$ .*

Notice that the existence of an actual simulator is a necessary but not a sufficient condition in order to make a query node redundant (A similar observation was also made by Ramanan (Ramanan, 2002)). Thus, in Definition 9, we require an actual simulator to be a sibling of a potentially redundant query node. The redundancies in a TPQ captured by means of simulation (see Definition 9) do not cover all the redundancies that may exist in a TPQ. There are redundancies beyond the simulation mechanism, which we refer to as *non-simulation-based redundancies*. Obviously, the simulation concept successfully captures all syntactic redundancies. In the presence of various ICs (as introduced earlier in this article), the redundancy issue becomes complicated. A “standard” way (followed by both Ramanan (Ramanan, 2002) and Amer-Yahia et al (Amer-Yahia, 2001; Amer-Yahia, 2002) and the author (Chen, 2006b)) is to augment an input TPQ by incorporating the effects of these ICs. However, not all ICs can be incorporated into a TPQ through augmentation. Even if we limit ourselves to only those ICs (i.e., the required child/descendant and subtype constraints) whose effects can be incorporated via augmentation, the simulation mechanism still cannot capture all potential redundancies. As an example, consider the TPQ in Figure 2(a) and assume the constraint set  $C = \{\tau(v4) \leq \tau(v2), \tau(v5) \leq \tau(v3), \tau(v4) \rightarrow \tau(v5)\}$ . Under the constraints  $C$ , the TPQ is already maximal (i.e. it cannot be further augmented, e.g., using the scheme adopted by Amer-Yahia (Amer-Yahia, 2001; Amer-Yahia, 2002) or Ramanan (Ramanan, 2002)). Obviously, node  $v4$  simulates node  $v2$  and at the same time is a sibling of  $v2$ . According to Definition 9,  $v2$  is redundant and the subtree rooted at  $v2$  (including  $v2$ ) shall be removed, resulting in Figure 2(b). However, the redundancy of node  $v5$ , caused by the constraint  $\tau(v4) \rightarrow \tau(v5)$ , remains. A more complex case of non-simulation-based redundancies is caused by the EL constraints as illustrated in Figure 2(c), where the entrance location node  $u$  can be simply removed (according the definition of entrance location), resulting in the equivalent TPQ in Figure 2(d). Our algorithm systematically examines all these cases of redundancies that previous algorithms failed to do.

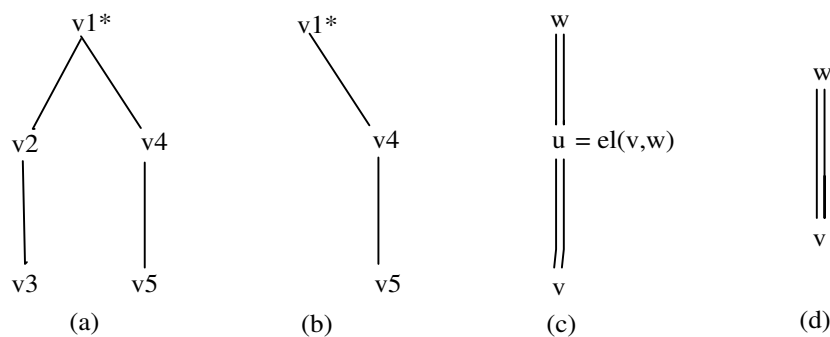


Figure 2. Illustration of non simulation-based redundancy in TPQs.

## Algorithms

It was recognized (Amer-Yahia, 2001; Amer-Yahia, 2002, Ramanan, 2002; Chen, 2006b) that, in the presence of certain constraints, if TPQ augmentation (for incorporating the effect of the constraints) were not carried out first, the TPQ would eventually fail to be minimized. Consequently, TPQ minimization in the presence of these constraints was conducted in three consecutive steps as in (Amer-Yahia, 2001; Amer-Yahia, 2002, Ramanan, 2002; Chen, 2006b): (1) TPQ augmentation –

incorporating the effect of relevant constraints into the query; (2) discovering and eliminating redundant query nodes from the augmented TPQ; (3) cleaning-up the added nodes by the augmentation step. Ramanan demonstrated (Ramanan, 2002) that when only the required child and required descendant constraints are considered, it is possible to identify all the potential redundant nodes in a TPQ by simply “*mimicking*” the effect of these constraints (thus bypassing the otherwise needed physical augmentation on the query). This simple mechanism led to the runtime optimality of his *MinimizeCTPQ* algorithm (Ramanan, 2002). Though highly efficient, this algorithm has very limited practical use because it can only exploit the required child/descendant constraints, while broader ICs are available in the context of XML. To surpass this limitation, we introduce a new, simple, yet powerful technique, called *allying*. In the same spirit, *allying* is also meant to avoid physically augmenting a TPQ  $Q$  via building *alliances* between the query nodes in  $Q$  and relevant type nodes in  $G'_C$ . The *allying* technique is embodied by the definition of the *augmented simulation relation* (See Definition 8). *Allying* substantially exceeds the limitation of *mimicking* (Ramanan, 2002) in that it allows incorporation (without causing physical augmentation) of a much broader spectrum of constraints into a query, including co-occurrence, and required parents/ancestors (as special cases of entrance locations), in addition to the simple required child/descendant constraints (which are the only two kinds of ICs that the *mimicking* technique can handle).

Before we present our minimization algorithm with full details, we point out a flaw of Ramanan’s *MinimizeCTPQ* algorithm (Ramanan, 2002). This algorithm performs a *reduction* step (based on the required child/descendant constraints) before computing the simulation relation and eliminating redundant query nodes determined according to the computed simulation relation. A problem of this arrangement is that a query may eventually fail to be minimized after reduction is performed. We will use an example in the next section to demonstrate this observation. Accordingly, reduction in our minimization approach is arranged as a post-processing step. The post reduction step in our algorithm subsumes the corresponding operations in *MinimizeCTPQ* and additional operations particularly designed for removing the redundant nodes caused by the EL constraints (also covering the required parent/ancestor constraints).

With a given TPQ  $Q$ , consisted of a set  $S$  of query nodes, we introduce the following auxiliary notations to facilitate the implementation and representation of our algorithms:

- ***cpar(S)***: the set of query nodes from  $Q$  that are the *c-parents* of the nodes in the node set  $S$ ; conversely, we say that every node in *cpar(S)* has a *c-child* in  $S$ .
- ***anc(S)***: the set of query nodes in  $Q$  that have a *proper* descendant in the node set  $S$  (recall that a c-child is not counted as a *proper* descendant).
- ***sim(u)***: for a query node  $u$  in  $Q$ , *sim(u)* denotes the set of all the actual simulators of node  $u$  (see Definition 8).
- ***augcpar(S)***: this is an augmented version of *cpar(S)*, consisting of the c-parents found from  $Q$  and also the c-parents found from  $G'_C$  (recall that  $G'_C$  is the constraint graph restricted to  $Q$ ). The c-parents found in  $G'_C$  are referred to as *virtual* c-parents.
- ***auganc(S)***: this is an augmented version of *anc(S)*, consisting of the (proper) ancestors found in  $Q$  and the (proper) ancestors found in  $G'_C$ . The ancestors found from  $G'_C$  are referred to as *virtual* ancestors.
- ***augsim(u)***: this is an augmented version of *sim(u)*, consists of both  $u$ ’s actual simulators (found in  $Q$ ) and  $u$ ’s virtual simulators (found in  $G'_C$ ).
- ***out(u)***: for a query node  $u$  in  $Q$ , *out(u)* tells whether  $u$  is an output node (*cf.* *out(Q)*) that returns the output node of  $Q$ ).

The main algorithm that implements our minimization scheme is called *AlliedMinimize* (see Figure 3), which in turn calls three other (sub-) algorithms for accomplishing the minimization task. The three sub-algorithms are *AlliedSimulation* (see Figure 4), which computes the augmented simulation relations on  $Q$ , *AlliedPruning* (see Figure 5), which identifies and removes

redundant nodes based on the computed simulation relations on  $Q$  (according to Definition 9), and *AlliedReduction*, which removes four cases of non-simulation-based redundant nodes from  $Q$ .

**Algorithm *AlliedMinimize***

**begin**

1. call *AlliedSimulation*( $Q$ ) to compute the augmented simulation relations on  $Q$
2. call *AlliedPruning*( $root(Q)$ ) to minimize  $Q$  by eliminating redundant query nodes/branches from  $Q$
3. call *AlliedReduction*( $Q$ ) to perform reduction on the output of *AlliedPruning*  
 /\* this step removes non-simulation-based redundant query nodes \*/

**end**

Figure 3. The main minimization algorithm *AlliedMinimize*

**Algorithm *AlliedSimulation*( $Q$ )**

**begin**

- 1 **let**  $N$  be a list of the nodes of  $Q$  in some bottom-up order
  - 2 **for** each  $u \in N$  in order **do** {
  - 3   **if**  $out(u)$  **then**  $augsim(u) = \{u\}$
  - 4   **else** {
  - 5     **if**  $u$  is a leaf **then**
  - 6        $augsim(u) = \{v \in N \cup N'_C \mid \tau(v) \leq \tau(u) \text{ is in } C^+\}$
  - 7        $\cup \{v \in N \mid el(u, v, w) \text{ in } C^+ \text{ and } v \text{ is a d-child sibling of } u \text{ in } Q\}$
  - 8     **else**
  - 9        $augsim(u) = \{v \in N \cup N'_C \mid \tau(v) \leq \tau(u), v \in augcpar(augsim(u')) \text{ for each}$
  - 10        c-child  $u'$  of  $u$ , and  $v \in anganc(augsim(u'')) \text{ for each d-child } u'' \text{ of } u\}$
  - 11     **for** every  $v \in augsim(u)$
  - 12       **if**  $v \in N$  and  $\tau(v) \in augsim(u)$  **then**  $v$  “allies” with the type node  $\tau(v)$
  - 13       /\* Algorithmically, the “allies” operation “associates”  $\tau(v)$  with node  $v$ , which in turn is implemented using  
       an associative array. \*/
  - 14     }
  - 15     compute  $augcpar(augsim(u))$
  - 16     compute  $anganc(augsim(u))$
  - 17 }
- end**

Figure 4. The *AlliedSimulation* algorithm

**Algorithm *AlliedPruning*(*u*)****begin**

```

1  for each child v of u do {
2    if v is a c-child of u then
3      if u has another undeleted c-child  $w \in \text{augsim}(v) \cap N$ 
4        then delete v /* deleting the entire subtree rooted at v in Q */
5      else AlliedPruning(v) /* node v is nonredundant */
6    if v is a d-child then
7      if u has another undeleted child  $w \in (\text{augsim}(v) \cup \text{auganc}(\text{augsim}(v))) \cap N$ 
8        then delete v /* deleting the entire subtree rooted at v in Q */
9      else AlliedPruning(v) /* node v is not redundant */
10 }
end

```

Figure 5. The *AlliedPruning* algorithm**Algorithm *AlliedReduction*(*Q*)****begin**

```

1  let M be a list of the nodes of Q in some bottom-up order
2  for each node  $v \in M$  do {
3    if v is a leaf then {
4      if Q contains a c-edge  $u \rightarrow v$  and  $\tau(u) \rightarrow \tau(v)$  is in  $G'_C$  then remove v from Q
5      else if Q contains d-edge  $u \Rightarrow v$  and  $\tau(u) \Rightarrow \tau(v)$  is in  $G'_C$  then remove v from Q
6    }
7  }
8  for each node  $u \in M$  (in the reverse order as at line 1) do {
9    for each child v of u do {
10     if  $\neg \text{out}(u)$  and v is the unique child of u and  $el(u, v)$  then remove u
11     if  $\neg \text{out}(v)$  and v has a unique child w and  $el(v, w, u)$  then bypass v
12       /* Here the “bypass” operation retains the connectivity from u to w after removing the intermediate node v */
13 }
end

```

Figure 6. The *AlliedReduction* algorithm

The main algorithm, *AlliedMinimize* (Figure 3) carries out three steps of operations, and each step is performed by invoking a corresponding sub-algorithm. The *AlliedSimulation* sub-algorithm computes the augmented simulation (see Definition 8) on an input TPQ *Q*. The *AlliedPruning* sub-algorithm identifies and eliminates (simulation-based) redundant query nodes from *Q* according to Definition 9. The *AlliedReduction* sub-algorithm removes four cases of non simulation-based redundant nodes, resulted as the immediate consequence of the required child/descendant and required parent/ancestor constraints (a special case of the EL constraint).

In general, *AlliedSimulation* is an algorithmic version of Definition 8, while *AlliedPruning* is straightforwardly based on Definition 9. The correctness of both sub-algorithms comes from these two definitions, respectively. However, in order to comprehend the details of *AlliedMinimize*, understanding the usage of the auxiliary function *auganc(S)* in *MinimizeCTPQ* (Ramanan, 2002) is helpful. A common feature in *AlliedMinimize* and *MinimizeCTPQ* (Ramanan, 2002) is that both avoid physical augmentation on an input TPQ to achieve the optimal runtime performance. For each node  $u$  in query  $Q$ , our sub-algorithm *AlliedSimulation* in turn computes *augsim(u)*, *augcpar(augsim(u))*, and *auganc(augsim(u))*. These three auxiliary functions (representing three important supportive mechanisms) help computing the augmented simulation relation (Definition 8) on  $Q$ , which includes, for each query node  $u$ , all actual simulators (in  $Q$ ) and all virtual simulators (in  $G'_C$ ) for the node  $u$ . *AlliedSimulation* implements a more general concept of simulation – augmented simulation (Definition 8). Our *AlliedSimulation* is more powerful than Ramanan’s *CTPQSimulation* (the simulation sub-algorithm used in *MinimizeCTPQ* (Ramanan, 2002)), and the power of *AlliedSimulation* is rendered by the supportive mechanisms *augsim(u)*, *augcpar(augsim(u))*, and *auganc(augsim(u))*. Algorithmically, the “allies” operation in *AlliedSimulation* (Figure 4, line 12) simply sets up a link from every possible query node  $u$  in  $Q$  to a corresponding type node  $\tau(u)$  in  $G'_C$ . The effect is the same as a virtually augmented TPQ is obtained, from which the augmented simulation on the query  $Q$  is computed, which then is used to determine redundant query nodes. In our implementation (Figure 4), the *allying* mechanism is embodied via the use of the three auxiliary functions: *augsim(u)*, *augcpar(augsim(u))*, and *auganc(augsim(u))*.

With a given TPQ  $Q$ , multiple nodes from the query node set  $N$  may get allied with the same type node in  $G'_C$ , and every type node in  $G'_C$  is distinct – representing a unique element type. While computing *augsim(u)*, *augcpar(augsim(u))*, and *auganc(augsim(u))*, with each allied pair of a query node (in  $Q$ ) and a type node (in  $G'_C$ ), it is sufficient to retain just the query node in the results if it is decided to be a proper member. We will use examples to readdress this issue in the next section.

After *AlliedSimulation* finishes computing the augmented simulation relation on  $Q$  (i.e., for each query node  $u$  in  $Q$ , *augsim(u)* is computed), the sub-algorithm *AlliedPruning* recursively (starting from the root) eliminates all simulation-based redundant nodes from  $Q$  (according to Definition 9). When examining simulation-based redundant nodes, the algorithm has to check whether there is an undeleted actual simulator  $v \in Q$  that is also a sibling of the query node being examined.

The most essential part of the main algorithm *AlliedMinimize* is the call for the *AlliedSimulation* sub-algorithm, which incorporates the effect of a given constraint set  $C$  into the input query  $Q$  via the *allying* technique (instead of physical augmentation). The *MinimizeCTPQ* algorithm (Ramanan, 2002) was designed also to avoid physically augmenting a TPQ through “mimicking” the effect of certain constraints, which are limited to only required children and descendants. Obviously, *MinimizeCTPQ* has very limited practical use because of this limitation. Our algorithm explores much broader ICs: besides the required child/descendant constraints, the subtype constraints is smoothly incorporated through the computation of *augsim(u)* (at line 6 and line 9 in Figure 4, respectively); the EL constraints are also explored (at line 7 in Figure 4).

*AlliedSimulation* (Figure 4) is an algorithmic version (implementation) of our third tier definition for simulations (Definition 8); it computes the augmented simulation relation on a TPQ with resort to the three important auxiliary functions: *augsim(u)*, *augcpar(augsim(u))*, and *auganc(augsim(u))*. *AlliedPruning* (Figure 5) is an algorithmic version (an implementation) of Definition 9; it identifies (based on the a-simulation relation computed by *AlliedSimulation*) and eliminates all simulation-based redundant nodes from the input query  $Q$ . In the sub-algorithm *AlliedReduction* (Figure 6), four cases of non simulation-based redundancies are checked and eliminated. To further demonstrate the correctness of *AlliedMinimize*, we will provide illustrative examples in the next section, where we also show the necessity of arranging the reduction step *after* elimination of all identified simulation-based redundancies (this in fact disproves the arrangement that Ramanan made with his *MinimizeCTPQ* algorithm (Ramanan, 2002) – i.e., performing reduction *before* identifying and removing simulation-based redundancies).

The above discussion leads to the following result (Theorem 1):

**Theorem 1.** *The AlliedMinimize algorithm correctly computes the minimized form of a TPQ in the presence of broad integrity constraints, which may include required children, required descendants, subtypes, entrance locations, required parents, and required ancestors (as a special case of the entrance location constraints).*

**Proof.** As the main algorithm *AlliedMinimize* consists of three steps, the theorem can be proved by separately proving each step. Step 1 computes the augmented simulation relations on an input query  $Q$  by calling the sub-algorithm *AlliedSimulation*. The correctness of it comes from the validity of our three-tier definitions for simulations. Step 2 prunes simulation-based redundant nodes by calling the sub-algorithm *AlliedPruning*, which trivially implements Definition 9, and the correctness is self-evident. In principle, step 3 does not affect the correctness of the main algorithm *AlliedMinimize*, but only the thoroughness of the final minimization result; the correctness of this step is rather self-explaining. ■

We now examine the time complexity of *AlliedMinimize*. Assume the input TPQ  $Q$  is of size  $n$ , i.e., with  $n$  nodes in the query pattern tree. For each query node  $u$ , algorithm *AlliedSimulation* (Figure 4) in order computes  $aug_{sim}(u)$ ,  $aug_{cpar}(aug_{sim}(u))$ , and  $aug_{anc}(aug_{sim}(u))$ . Primarily, this algorithm consists of a *for* loop (line 2 to line 17), of which line 3 to line 10 compute  $aug_{sim}(u)$ . If  $u$  is an output node, computing  $aug_{sim}(u)$  takes  $O(1)$  time. If  $u$  is a leaf node, computing  $aug_{sim}(u)$  consists of two steps: at line 6, the algorithm examines candidates from  $N \cup N'_C$  (let  $|N'_C| = m$ ), and the time taken is upper bounded by  $O(|N \cup N'_C|) = O(n+m)$ ; the time spent at line 7 is proportional to the number of the siblings of  $u$ , and is upper bounded by  $O(n * |siblings(u)|)$  (notice that retrieving the entrance location information for node  $u$  takes constant time  $O(1)$  assuming a hash table is used to store the EL information). So the total time needed for computing  $aug_{sim}(u)$  when  $u$  is a leaf node is  $O(n+m) + O(n * |siblings(u)|) = O(n + m + n * |siblings(u)|)$ . If  $u$  is an internal node, determining whether  $v \in aug_{sim}(u)$  is proportional to the number of children of  $u$ , and is upper bounded by  $O(|N \cup N'_C| * |children(u)|) = O((n+m) * |children(u)|)$ . At line 11 and 12, each actual simulator “allies” with a virtual simulator, taking  $O(1)$  time, and the total time spent on allying is  $O(n)$ . Now consider computing  $aug_{cpar}(aug_{sim}(u))$  (line 15) and  $aug_{anc}(aug_{sim}(u))$  (line 16). Both check candidates from  $N \cup N'_C$  bottom-up (in the order of decreasing depth – i.e., the distance to the root), and can be computed in  $O(n+m)$  time. Putting all together and using  $|children(u)|$  to substitute for  $|siblings(u)|$  (because the number of one’s siblings is approximately the same as the number of children of one’s parent – to be accurate, the difference is 1), we get the time spent by a single pass of the *for* loop (line 3 to line 17):  $O(2(m+n) + (m+2n) * |children(u)| + 2)$ ; hence, the total time spent by the *for* loop is  $O(\sum_{v \in N} (2(m+n) + (m+2n) * |children(u)| + 2)) = O(4n^2 + 3mn + 2n)$ , which is approximately  $O(n^2)$  if assuming  $m \approx n$  (this assumption is based on the observation that  $G'_C$  usually has a comparable size with  $Q$ ). Finally, because the augmented simulation relation itself (Definition 8) can be of size  $\Theta(n * (n+m)) \approx \Theta(n^2)$ , the sub-algorithm *AlliedSimulation* is runtime optimal (this part of time dominates the total time of the entire *AlliedMinimize* algorithm as to be made clear below).

Our study indicates that it is the physical augmentation performed on an input TPQ, for example, by the algorithm *MinimizeChase* (Ramanan, 2002), that makes the size of the input TPQ substantially increased (from  $n$  to  $n^3$ ), which leads to the  $O(n^4)$  total time complexity of the algorithm. Accomplishing the *allying* technique, our *AlliedSimulation* algorithm successfully retains the TPQ within the space of its original size  $n$ , which effectively minimizes the subsequent effort needed for identifying redundant query nodes in the TPQ.

Next, we look at the recursive sub-algorithm *AlliedPruning*. For each child  $v$  of  $u$ , *AlliedPruning* takes  $O(|children(u)|)$  time to check if the node  $v$  is redundant. This algorithm is recursively applied to every query node in the input query  $Q$ , starting from the root. So, the total time needed is  $O(\sum_{v \in N} |children(u)|) = O(n)$ .

Lastly, we consider the sub-algorithm *AlliedReduction*. This sub-algorithm can be divided as two main parts. The first *for* loop (from line 2 to line 7) performs the “classic” reduction step introduced by Ramanan in (Ramanan, 2002) for his runtime-optimal algorithm *MinimizeCTPQ*. The first loop removes all non simulation-based redundancies caused by required children/descendants, which can only happen on leaves; the time spent is  $O(n)$ . The second *for* loop (from line 8 to line 13) traverses the TPQ and removes non simulation-based redundancies caused by required parents/ancestors (special cases of the EL constraints). The time spent is also  $O(n)$ .

Obviously, the time spent by *AlliedSimulation* dominates the total time of the main algorithm *AlliedMinimize*. We draw the following conclusion (Theorem 2) from the above discussion with regard to time complexity of *AlliedMinimize*.

**Theorem 2.** *The AlliedMinimize algorithm has optimal runtime of  $O(4n^2+3mn)$  with regard to the query size  $n$  and the scale  $m$  of the constraint set  $C$ ; when assuming  $m \approx n$  (for simplicity) the time complexity becomes  $O(n^2)$ .*

**Proof.** A formal proof is trivially derived from the above discussion. ■

Now we highlight a few key points in *AlliedMinimize*. Graph simulation, though useful for identifying redundant nodes in a TPQ, has practical limitations – it cannot help for identifying non-simulation-based redundancies. There are four cases of non simulation-based redundancies identified and handled in our algorithm: the two cases caused by the required children and required descendants are handled by the first *for* loop in *AlliedReduction* (see Figure 6); and the other two cases caused by required parents/ancestors (special cases of the EL constraints) are handled by the second *for* loop in *AlliedReduction*. The entrance location concept represents an important (generic) type of constraints (covering required parents/ancestors as special cases). This concept is explored at three places in our algorithm: at line 7 of *AlliedSimulation*, it is explored for identifying simulation-based redundancy; at line 10 and 11, it is used for eliminating the two cases of non simulation-based redundancies caused by required parents/ancestors.

Our *AlliedMinimize* is most related to the *MinimizeCTPQ* algorithm proposed by Ramanan (Ramanan, 2002). The two algorithms share an important, common feature – both being designed to circumvent physical augmentation on input queries for the sake of improved performance. However, our *AlliedMinimize* substantially exceeds *MinimizeCTPQ* in the broadness of ICs that can be exploited for TPQ minimization. *AlliedMinimize* distinguishes itself from *MinimizeCTPQ* in the following aspects: (1) a new conceptual mechanism, *allying*, is implemented with *AlliedMinimize*, which is more powerful than the simply *mimicking* idea adopted by *MinimizeCTPQ* (Ramanan, 2002); (2) *AlliedMinimize* is based on a much generalized concept of simulation – augmented simulation (Definition 8), which forms the backbone of the powerful *AlliedMinimize* algorithm; (3) the basic implementation mechanisms in *AlliedMinimize* are extended from the original *auganc()* (Ramanan, 2002) to *augsim()* and *augpar()*, in addition to *auganc()*.

## Examples, Discussion, and Implementation

In this section, we walk through the application of *AlliedMinimize* to a couple of examples for illustration purpose, and at the same time initiate an interesting discussion about the runtime optimality of *AlliedMinimize*. Finally we present some experiment results.

### Examples

**Example 1.** Query  $Q1$  is shown in Figure 7(a); each query node  $v_i$  ( $i=1, 2, \dots, 6$ ) is associated with a corresponding type  $\tau(v_i) = t_i$ ; the given constraint set is  $C = \{t1 \rightarrow t4, t4 \Rightarrow t3, t3 \Rightarrow t5, t4 \Rightarrow t6, t4 \leq t2\}$ . The restricted constraint graph  $G'_C$  is shown in Figure 7(b).

The core of *AlliedMinimize* is the *AlliedSimulation* sub-algorithm that step-by-step computes the *augsim* function on each query node utilizing two other functions, *augcpar* and *anganc*; the *AlliedPruning* sub-algorithm then identifies and eliminates redundant query nodes based on the computed a-simulations on  $Q$  (embodied by the *augsim* function computed for each query node). Applying this algorithm to  $Q1$ , we get  $augsim(v5) = \{v5, t5\}$ ,  $augcpar(augsim(v5)) = \{ \}$ ,  $auganc(augsim(v5)) = \{v3, v2, v1, t3, v4, t1\}$  (notice that in this example as  $v4$  is allied with  $t4$  so we only need to retain  $v4$  in  $auganc(augsim(v5))$ ),  $augsim(v3) = \{v3, t3\}$ ,  $augcpar(augsim(v3)) = \{ \}$ ,  $auganc(augsim(v3)) = \{v2, v1, v4\}$ ,  $augsim(v6) = \{v6, t6\}$ ,  $augcpar(augsim(v6)) = \{ \}$ ,  $auganc(augsim(v6)) = \{v2, v1, v4\}$ ,  $augsim(v2) = \{v2, v4\}$ . According to Definition 9,  $v2$  has an undeleted, sibling, actual simulator  $v4$  in  $Q1$ , so the subtree rooted at  $v2$  is redundant and removed, which results in the TPQ shown in Figure 7(d). After the reduction step,  $v4$  is removed and the final result is shown in Figure 7(e), which is a single node TPQ, comprising one  $v1$ .

The above example also serves as a counterexample to Ramanan’s claim (Ramanan, 2002) with regard to the reduction step. He stated that the reduction performed before minimization is harmless and thus should be arranged as the first step. With the above example  $Q1$ , his *MinimizeCTPQ* would reduce the query to  $R(Q1)$  as shown in Figure 7(c) (where  $v5$  and  $v4$  are removed due to the constraints  $t3 \Rightarrow t5$  and  $t1 \rightarrow t4$ ). Obviously  $R(Q1)$  is not minimal (comparing with Figure 7(e)) but cannot be minimized any further. This scenario demonstrates that the reduction step can only be safely applied after the simulation relation are computed and the redundant nodes are removed. Therefore, we conclude that his well-known, runtime-optimal algorithm *MinimizeCTPQ* (Ramanan, 2002) is incorrect in this regard.

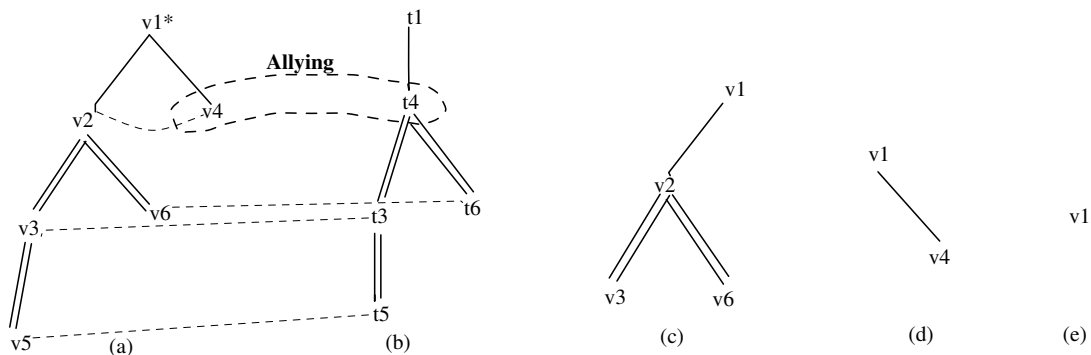


Figure 7. Illustration of Example 1 (dashed lines denote simulation relation)

**Example 2.** Query  $Q_2$  is shown in Figure 8(a); every query node  $v_i$  ( $i=1, 2, \dots, 5$ ) is associated with an element type (annotated on the right of the node). Let the constraint set be  $C = \{SciCollege \rightarrow SciDept, SciDept \rightarrow Lab, SciCollege \leq College, SciDept \leq Dept, el(Univ, SciCollege)\}$ .

We walk through the application of *AlliedMinimize* to  $Q_2$  and observe the following intermediate results:  $augsim(v_4) = \{v_4, Lab\}$ ,  $augpar(augsim(v_4)) = \{v_3, SciDept\}$ ,  $auganc(augsim(v_4)) = \{v_2, v_1, v_5\}$  (noticing that  $v_5$  is “allied” with *SciCollege*, so *SciCollege* is not duplicately retained in  $auganc(augsim(v_4))$ ),  $augsim(v_3) = \{v_3, SciDept\}$ ,  $augpar(augsim(v_3)) = \{v_2, v_5\}$ ,  $auganc(augsim(v_3)) = \{v_1\}$ ,  $augsim(v_2) = \{v_2, v_5\}$ . Since node  $v_2$  has an undeleted, sibling, actual simulator  $v_5$ , the subtree rooted at  $v_2$  is redundant and removed, resulting in the TPQ shown in Figure 8(c). Finally, because of the  $el(Univ, SciCollege)$  constraint (i.e., *Univ* is a required parent of *SciCollege*) and  $\neg out(Univ)$  (i.e., *Univ* is not an output node), *AlliedReduction* ultimately removes node  $v_1$  from  $Q_2$  and produces the final, single-node TPQ in Figure 8(d).

■

Example  $Q_2$  is adapted from (Ramanan, 2002). The same query with the same set of constraints (except for the  $el(Univ, SciCollege)$  constraint) was ever used by Ramanan (Ramanan, 2002) as an example to show the unsolvable-ness of TPQ minimization when the subtype constraint is present. In the above discussion, we purposely adopted this very same query to contrastingly show the superiority of our *AlliedMinimize* algorithm.

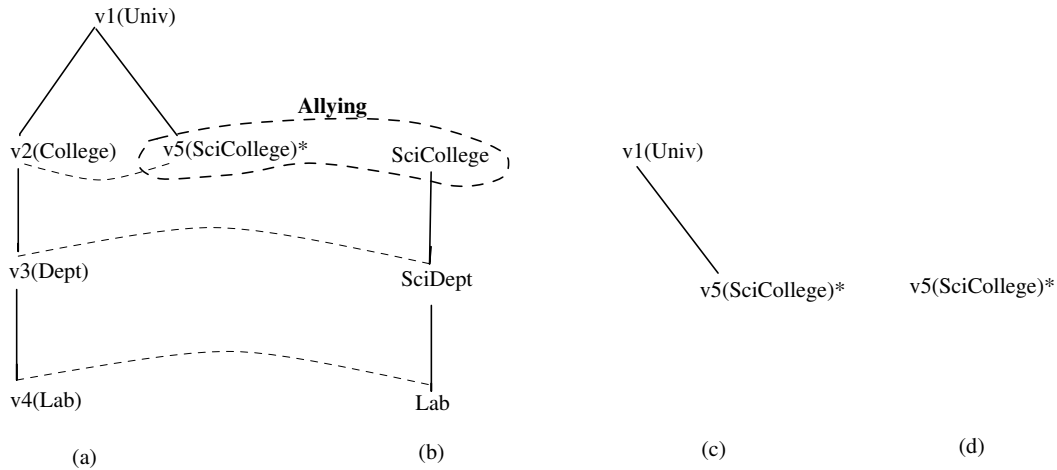


Figure 8. Illustration of example 2

### Discussion

As a novel algorithm for TPQ minimization, our *AlliedMinimize* claims not only runtime-optimal but also the most powerful algorithm proposed so far in terms of the variety of ICs that can be exploited. Even so, there are still cases regarding the EL constraints that *AlliedMinimize* does not incorporate into the uniform framework of its minimization approach. For example, Definition 7 only considers the effect of EL constraints at leaf level while it is possible to explore the usage of EL at higher levels in a TPQ. To retain the runtime optimality, we stopped to further explore the EL constraints in *AlliedMinimize*.

Nevertheless, *AlliedMinimize* is the most powerful algorithm for TPQ minimization that examines a much broad spectrum of ICs to achieve thorough TPQ minimization.

As a more specific example, consider the TPQ shown in Figure 9(a) with the given constraint set  $C = \{el(\tau(v4), \tau(v3), \tau(v1)), \tau(v3) \Rightarrow \tau(v6), \tau(v4) \leq \tau(v2), \tau(v6) \leq \tau(v5)\}$ . Considering the implication of the EL constraint and the required descendant constraint in  $C$ , the TPQ in Figure 9(a) can be equivalently transformed to Figure 9(b). Then, we can reason that the subtree rooted at node  $v4$  simulates the subtree rooted at node  $v2$ , which makes the latter (i.e., the left subtree in Figure 9(b)) redundant and removed, resulting in the TPQ in Figure 9(c). After removing the two added nodes  $v4$  and  $v6$ , the minimal form is the TPQ shown in Figure 9(d). This example demonstrates a case that the combined effect of an EL constraint with other (kinds of) constraints (a required descendant and two subtype constraints in this example) can be further exploited to achieve better minimization results. Our current *AlliedMinimize* algorithm does go farther. We choose to settle at the current status for retaining the highly-desired runtime optimality of the algorithm.

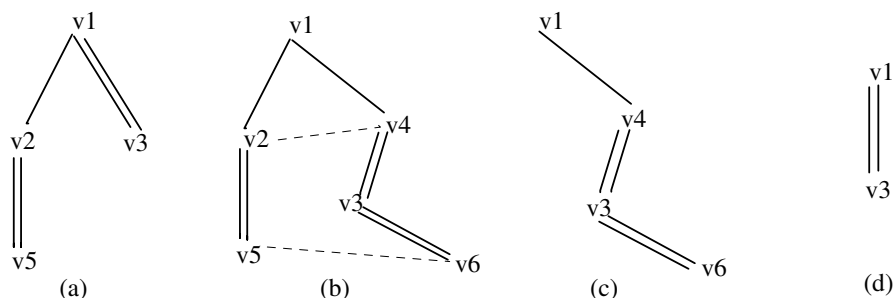


Figure 9. Illustration of limited exploitation of entrance locations

## Experiments

For experimental study, we implemented our *AlliedMinimize* algorithm along with the closely related one, *MinimizeChase* (Ramanan, 2002). We did not implement *MinimizeCTPQ* because *MinimizeCTPQ* is actually incorrect (as we pointed out in the early part of this section) and incapable of handling the important *subtype* constraint. In our test-bed, *AlliedMinimize* and *MinimizeChase* are both implemented in Java (with Java 1.4 compiler) under Windows XP operating system running on a Intel Pentium 4 processor of 1.4 GHZ with 1 GB RAM.

The prior analytical study indicates that the runtime of *AlliedMinimize* is dependent on the query size  $n$  and the constraint scale  $m$ , and is upper-bounded by  $mn$  or  $n^2$  if assuming  $m \approx n$ . From Ramanan's analysis, we know the runtime of *MinimizeChase* is upper bounded by  $n^4$ . The experimental data we obtained further confirms the above analytical results. While doing this experiment, we utilized a synthesized dataset which consists of a randomly generated set of trees; the total size of the dataset is about 50 MB; we applied a pool of test queries with randomly generated fan-outs and query structures (i.e., left-deep, right-deep, or even) and a pool of randomly generated constraints (including required children, required descendants, subtypes, co-occurrences, and EL constraints).

As *MinimizeChase* accepts only the required child and descendant and subtype constraints, for equal-footing comparison, we partitioned our constraint pool and stored the constraints into two separate sets. The first set accommodates the constraints that

are explored by both algorithms, and the second set holds the constraints that are only explored by our *AlliedMinimize*. Various hashing structures were used to provide quick access to these constraints.

### Performance result of *AlliedMinimize*

We recorded the total time spent by *AlliedMinimize* on each query selected from the query pool with a growing query size starting from 10 and with a growing set of constraints starting from 0. The performance data is shown in Table I, where the execution time is the average elapsed time over 500 times of repeated execution of the same query with the same set of constraints.

Query size (→) Constraints (↓)	10	20	30	40	50
<b>0 cons</b>	0.112	0.241	0.512	0.822	1.082
<b>10 cons</b>	0.128	0.272	0.546	0.882	1.262
<b>20 cons</b>	0.141	0.324	0.634	1.124	1.432
<b>30 cons</b>	0.152	0.376	0.722	1.148	1.582
<b>40 cons</b>	0.163	0.436	0.816	1.282	1.694
<b>50 cons</b>	0.184	0.504	0.924	1.442	1.894

Table I. Performance data of *AlliedMinimize*: average run time with increasing query size and increasing constraint set (unit: ms)

The data in Table I is plotted in Figure 10 and 11 for better visual effect. From Figure 10, we can see that when the size of a query is small (less than 20, for example) the growing of the constraint set does not have an obvious affect on the overall performance of the *AlliedMinimize* algorithm. This is understood because for relatively small queries, when the constraint set grows, many added constraints are actually irrelevant to the query and do not contribute to the increase of the query's execution time. It is also noticed that with relatively large queries, the increment of the constraint set leads to more constraints being examined by *AlliedMinimize*, which accounts for the slightly faster increasing of the execution time of *AlliedMinimize*.

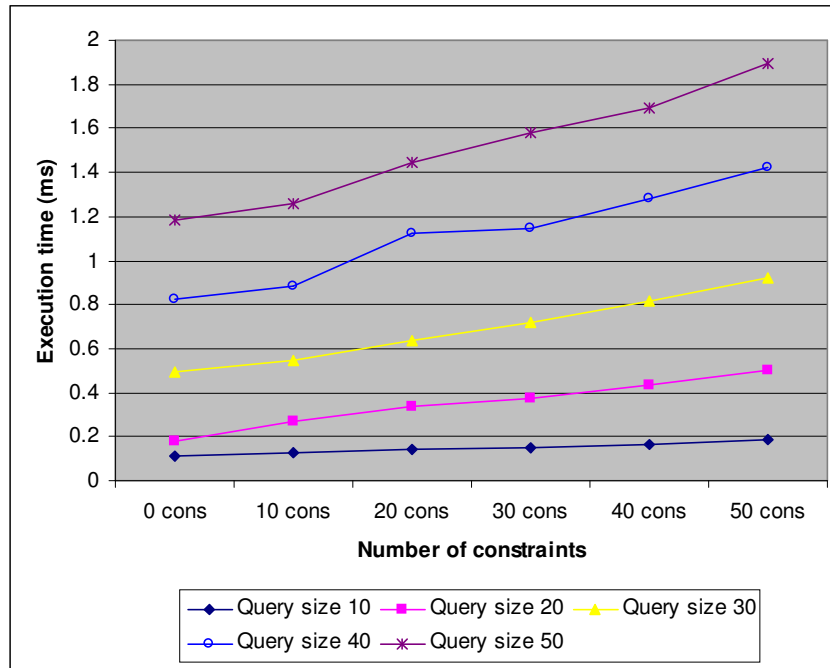


Figure 10. *AlliedMinimize*: performance vs. constraints (with fixed query sizes)

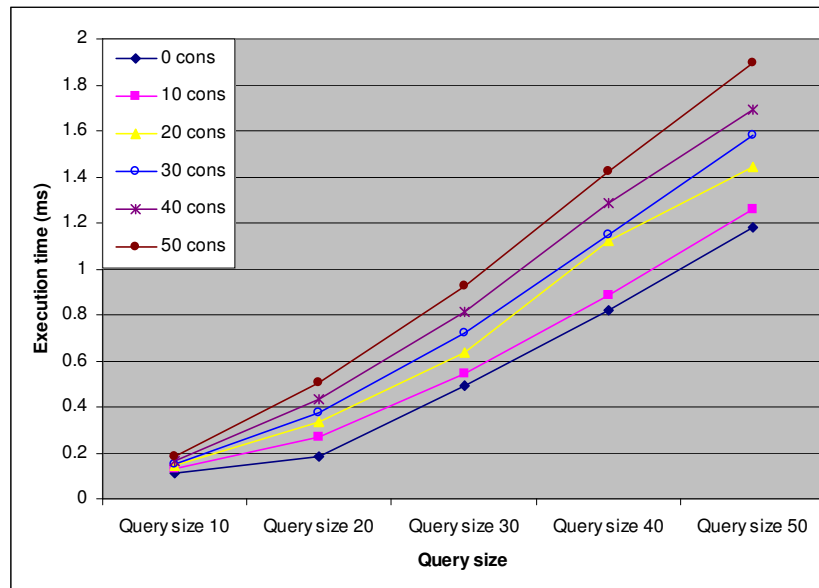


Figure 11. *AlliedMinimize*: performance vs. query sizes (with fixed constraint scale)

### Comparison with *MinimizeChase*

Our work is most related to the work of Amer-Yahia et al (Amer-Yahia, 2001) and that of Ramanan (Ramanan, 2002). These two pieces of related work both deal with at most three different types of constraints: required child, required descendant, and subtype constraints. Because Ramanan’s algorithms are superior to the corresponding algorithms of Amer-Yahia et al, we only choose to compare our algorithm with Ramanan’s representative algorithm *MinimizeChase* in this study. Table II shows the collected performance data of *AlliedMinimize* and *MinimizeChase* when the test-bed is configured with a fixed set of 40 constraints. As expected, *AlliedMinimize* consistently outperforms *MinimizeChase*. The advantage of *AlliedMinimize* over *MinimizeChase* becomes more obvious when the experimental data is transformed into plots (see Figure 12). We notice that as the query size increases, the execution time of *AlliedMinimize* increases linearly, while that of *MinimizeChase* increases much faster especially when the queries becomes larger.

Query size	10	20	30	40	50
MinimizeChase	0.283	0.596	0.998	2.282	3.995
AlliedMinimize	0.163	0.436	0.816	1.282	1.694

Table II. Performance data: *AlliedMinimize* vs. *MinimizeChase* (unit: ms)

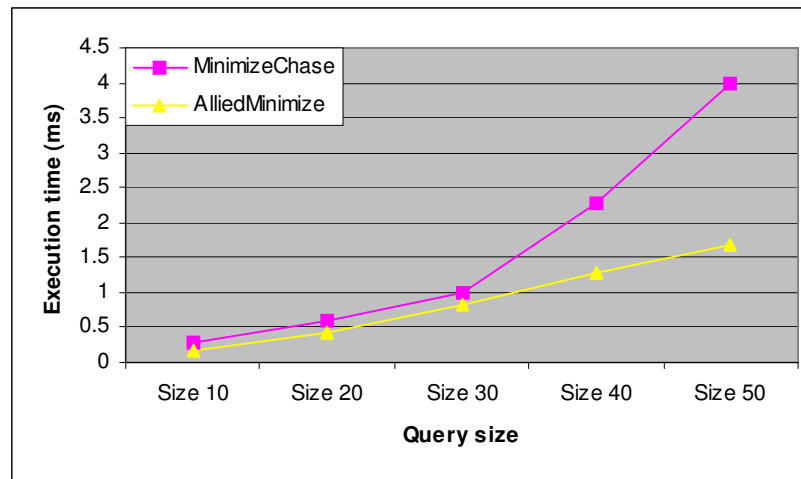


Figure 12. Performance comparison: *AlliedMinimize* vs. *MinimizeChase*

### Conclusion

In this article, we presented a novel, runtime-optimal algorithm for TPQ minimization, called *AlliedMinimize*, based on an innovative technique, called *allying*. The *AlliedMinimize* algorithm is not only highly efficient (runtime optimal), but also very powerful – it can handle a much broader spectrum of XML constraints, including the required child/descendant, subtype, co-occurrence, and entrance location constraints, and the required parent and ancestor constraints (special cases of the entrance

location constraints). In contrast, the well-known algorithms proposed in (Amer-Yahia, 2001; Amer-Yahia, 2002, Ramanan, 2002) all can only deal with three simple types of constraints at the most: the required child, required descendant, and the subtype constraints. *AlliedMinimize* is so far, to the best of our knowledge, the most efficient (runtime optimal) and the most powerful algorithm (in terms of the broadness of constraints that it can explore).

TPQ has great significance for the optimization and accelerated execution of XML queries. As part of future work, we plan to integrate the approach and the algorithm presented in this article into our comprehensive framework for XML query processing and optimization, which includes XML query pattern minimization, logical XML query optimization, physical XML query optimization (cost-based), and efficient evaluation of optimized XML queries.

## Acknowledgements

The author acknowledges the contribution made by several graduate students at SIUC with the experimental study reported in this article. The author would also like to thank the anonymous reviewers for their constructive view comments – that were found helpful in forming the final version of this article.

## References

- Amer-Yahia, S., Cho, S., Lakshmanan, L. V. S., Srivastava, D., (2001), Minimization of Tree Pattern Queries, In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Santa Barbara, CA, USA, pp. 497-508.
- Amer-Yahia, S., Cho, S., Lakshmanan, L. V. S., Srivastava, D., (2002), Tree Pattern Query Minimization, The VLDB Journal, 11(4), pp. 315-331.
- Böhm, K., Aberer, K., Özsu, T. M., Gayer, K., (1998), Query Optimization for Structured Documents Based on Knowledge on the Document Type Definition, In: Proceedings of IEEE International Forum on Research and Technology Advances in Digital Libraries, Santa Barbara, California, USA, pp. 196-205.
- Boag, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J., Siméon, J., (2007), XQuery 1.0: An XML Query Language, World Wide Web Consortium Recommendation, available at <http://www.w3.org/TR/2007/REC-xquery-20070123/>.
- Chakravathy, S., Grant, J., Minker, J., (1988), Foundations of Semantic Query Optimization for Deductive Databases, In: Minker (Ed.), Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann, pp. 243 - 273.
- Che, D., Aberer, K., Özsu, M. T., (2006), Query Optimization in XML Structured-Document Databases, The VLDB Journal, 15(3), pp. 263-289.
- Chen, Y., Che, D., (2006a), Efficient Processing of XML Tree Pattern Queries, Journal of Advanced Computational Intelligence and Intelligent Informatics, pp. 738-743.
- Chen, Y., Che, D., (2006b), Minimization of XML Tree Pattern Queries in the Presence of Constraints, Journal of Advanced Computational Intelligence and Intelligent Informatics, pp. 744-751.
- Clark, J., DeRose, S. (Ed.), (1999), XML Path Language (XPath), World Wide Web Consortium Recommendation, available at <http://www.w3.org/TR/xpath>.
- Fan, W., Simeon, J., (2000), Integrity Constraints for XML, In: Proceedings of ACM PODS Conference, Dallas, Texas, USA, pp. 23-34.
- Flesca, S., Furfaro, F., Masciari, E., (2003), On the Minimization of Xpath Queries, In: Proceedings of the 29th VLDB Conference, Berlin, Germany, pp. 153-164.
- Florescu, D., Levy, A., Suciu, D., (1988), Query Containment for Conjunctive Queries with Regular Expressions, In: Proceedings of the 17th ACM Symposium on Principles of Database Systems, Seattle, WA, USA, pp. 139-148.
- Garey, M. R., Johnson, D. S., (1979), Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman & Co., NY.
- Lee, D., Chu, W.W., (2000), Constraints-preserving Transformation from XML Document Type Definition to Relational Schema, In: Proceedings of the 19th International Conference on Conceptual Modeling (ER'2000), Salt Lake City, Utah, USA, pp. 323-338.
- McHugh, J., Widom, J., (1999), Query Optimization for XML, In: Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, UK, pp. 315-326.
- Miklau, G., Suciu, D., (2002), Containment and Equivalence for an XPath Fragment, In: Proceedings of 21st ACM Symposium on Principles of Database Systems, Madison, Wisconsin, USA pp. 65-76.

- Ramanan, P., (2002), Efficient Algorithms for Minimizing Tree Pattern Queries, In: Proceedings of ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, ACM Press, pp. 299-309.
- Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., Price, T., (1979), Access Path Selection in a Relational Database Management System, In: Proceedings of SIGMOD Conference, Boston, Massachusetts, USA, pp. 23-34.
- Ullman, J. D., (1989), Principles of Database and Knowledge Base Systems, volumes I and II, Computer Science Press, Rockville, Maryland.
- Wood, P. T., (1999), Optimizing Web Queries Using Document Type Definitions, In: Proceedings of 2nd ACM CIKM International Workshop on We Information and Data Management, Kansas City, Missouri, USA, pp. 28-32.
- Wood, P. T., (2000a), On the Equivalence of XML Patterns, In: Proceedings of 1st International Conference on Computational Logic, London, UK, pp. 1152-1166.
- Wood, P. T., (2000b), Rewriting XQL Queries on XML Repositories, In: Proceedings of 17th British National Conference on Databases, Exeter, UK, pp. 209-226.
- Wood, P. T., (2001), Minimizing Simple XPath Expressions, In: Proceedings of the Fourth International Workshop on the Web and Databases, Santa Barbara, CA, USA, pp. 13-18.
- Wood, P. T., (2003), Containment for XPath Fragments under DTD Constraints, In: Proceedings of 9th International Conference of Database Theory, Siena, Italy, pp. 300-314.

**Dunren Che** is an assistant professor at the Department of Computer Science, Southern Illinois University Carbondale (SIUC). Before joining SIUC, he worked as a post-doc/researcher in several world-class institutes for numerous years. He received his PhD from the Beijing University of Aeronautics and Astronautics of China in 1994. He has now produced nearly 70 research papers. His research has recently been focused on XML database technology, particularly XML query processing and optimization. More information about his teaching, research, and publications is online: <http://www.cs.siu.edu/~dche/>