Querying Large Collections of Semistructured Data

by

Shahab Kamali

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2013

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

An increasing amount of data is published as semistructured documents formatted with presentational markup. Examples include data objects such as mathematical expressions encoded with MathML or web pages encoded with XHTML. Our intention is to improve the state of the art in retrieving, manipulating, or mining such data.

We focus first on mathematics retrieval, which is appealing in various domains, such as education, digital libraries, engineering, patent documents, and medical sciences. Capturing the similarity of mathematical expressions also greatly enhances document classification in such domains. Unlike text retrieval, where keywords carry enough semantics to distinguish text documents and rank them, math symbols do not contain much semantic information on their own. Unfortunately, considering the structure of mathematical expressions to calculate relevance scores of documents results in ranking algorithms that are computationally more expensive than the typical ranking algorithms employed for text documents. As a result, current math retrieval systems either limit themselves to exact matches, or they ignore the structure completely; they sacrifice either recall or precision for efficiency.

We propose instead an efficient end-to-end math retrieval system based on a structural similarity ranking algorithm. We describe novel optimization techniques to reduce the index size and the query processing time. Thus, with the proposed optimizations, mathematical contents can be fully exploited to rank documents in response to mathematical queries. We demonstrate the effectiveness and the efficiency of our solution experimentally, using a special-purpose testbed that we developed for evaluating math retrieval systems. We finally extend our retrieval system to accommodate rich queries that consist of combinations of math expressions and textual keywords.

As a second focal point, we address the problem of recognizing structural repetitions in typical web documents. Most web pages use presentational markup standards, in which the tags control the formatting of documents rather than semantically describing their contents. Hence, their structures typically contain more irregularities than descriptive (data-oriented) markup languages. Even though applications would greatly benefit from a grammar inference algorithm that captures structure to make it explicit, the existing algorithms for XML schema inference, which target data-oriented markup, are ineffective in inferring grammars for web documents with presentational markup.

There is currently no general-purpose grammar inference framework that can handle irregularities commonly found in web documents and that can operate with only a few examples. Although inferring grammars for individual web pages has been partially addressed

by data extraction tools, the existing solutions rely on simplifying assumptions that limit their application. Hence, we describe a principled approach to the problem by defining a class of grammars that can be inferred from very small sample sets and can capture the structure of most web documents. The effectiveness of this approach, together with a comparison against various classes of grammars including DTDs and XSDs, is demonstrated through extensive experiments on web documents. We finally use the proposed grammar inference framework to extend our math retrieval system and to optimize it further.

Acknowledgements

During my work on the thesis, I had the distinct honor of being supervised by Frank Wm. Tompa whose vast knowledge and insight in the field together with his nice personality made it a wonderful experience. I would also like to thank my committee members, Ihab Ilyas, J. Ian Munro, Davood Rafiei, and Mark Smucker, for taking the time to read and critique my thesis and for their constructive comments.

Most of all, I want to thank my family from the bottom of my heart: my parents, my brother Shahin, and my wife Erfaneh. You have all always stood by me and believed in me, I would not be where I am today if it was not for your support.

Finally, I would like to thank the many friends I have met during my studies. Without you this could probably be finished earlier, but surely less enjoyably.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Semistructured markup is commonly used to encode online documents or objects such as math expressions or web tables within them. A variety of applications, from information extraction tools to complex query answering and specialized search systems, query such data, mine patterns in them, or extract information from them.

Descriptive and presentational markup are two major categories of markup languages. In descriptive markup tags are used to label parts of a document semantically, while in presentational markup tags are mostly used to specify how they should be visually rendered. The popularity of XML as the standard for expressing semistructured data has motivated many researchers to address many challenging problems. However, the majority of such research is focused on descriptive XML. Despite such a tremendous effort, there are still interesting problems to be solved, which is the aim of this research.

Mathematical expressions are objects with complex structures and rather few distinct terms, and considering their structures to further process them is inevitable. Hence, math retrieval is a perfect example of comparing and mining structures to process and retrieve large collections of semistructured data. Hence, we mainly focus on math retrieval as a representative example. However, most algorithms presented in this thesis can be extended to other structured or semistrucutred objects.

1.1 Mathematics Retrieval

Pages with mathematical contents are commonly published. Mathematics retrieval provides ways to query such pages through their mathematical expressions. The huge number

of potential users and the lack of powerful solutions provide an opportunity for an effective mathematics retrieval system to be developed. Moreover, it can be extended to support management and retrieval of other semistructured data.

Just as other documents with XML markup, mathematical expressions are encoded with respect to their content, i.e. mathematical meaning, or their presentation, i.e. appearance. The majority of published mathematical expressions are represented by their appearance, and the lack of content information forces a retrieval system to rely mostly on the presentation of expressions. Presentational markup for math expressions does not provide much semantic information about the symbols. Instead, only their appearance and relative arrangements are encoded.

Retrieving documents based on their mathematical content is currently very limited. Content-based mathematics retrieval systems (to be described in depth in Section 3.1) are limited to resources that encode the semantics of mathematical expressions, which are not very popular. The few mathematics retrieval systems that rely on the presentation of mathematical expressions either can find exact matches only, or they use a similarity ranking that usually returns a large number of irrelevant results. Moreover, a query language that is simple and easy to use, but powerful enough to help the user in expressing her needs is missing. As a result, the existing mathematics retrieval systems are not very helpful in addressing users' needs.

The lack of a practical definition for similarity between mathematical expressions, and the inadequacy of searching for exact matches only, makes the problem more difficult to answer. Conventional text retrieval systems are not tuned for mathematical expressions. Moreover, there is no clear definition for similarity between mathematical expressions, and merely searching for exact matches often results in missing useful information. Hence, relying on math symbols to compare math expressions is not adequate, and we should consider their rather complex structures to make an accurate comparison.

In Chapter 3 we assume the query is a single expression, and we consider the basic problem of how to capture the relevance of math expressions . We review the related work and describe the existing systems for retrieving mathematical expressions. We also discuss various possible approaches to this problem, and propose retrieval algorithms for the most promising ones. More specifically, we introduce *structural similarity search* and *pattern search*. We finally compare them in terms of their efficiency and accuracy of results. The results have been presented at various conferences [59, 60, 62].

1.2 Optimizing Math Retrieval

Considering the structures of math expressions when comparing them significantly improves retrieval accuracy. However, comparing expressions this way is computationally expensive, which makes the retrieval a time-consuming task, especially when the number of expressions is large. This is an important problem that should be addressed to achieve a practical search system. In Chapter 4 we address this problem by proposing optimization techniques for the search algorithms described in Chapter 3.

Optimizing Structural Similarity Search

To reduce the number of comparisons between expressions, we propose an optimization based on calculating an upper bound on the similarity of expressions. This allows defining early termination conditions to select the top-k results and stop the search algorithm early.

Processing semistructured data objects in a breadth-first manner, such as finding the edit distance between two trees, is the basis of many query answering and data extraction systems. Solutions to such operations typically use dynamic programming: the problem is solved for smaller substructures and the result is stored to be used for solving the problem for larger structures. If an application requires to perform such operation on many semistructured objects, e.g. finding the edit distance between a query and all expressions in the repository, then one possibility is to do it on each object separately. However, if objects share common substructures, the same operation is calculated for such substructures repeatedly. Therefore, if objects in the repository share many common substructures, an indexing scheme that allows reusing the result for such substructures may significantly reduce the query processing time.

Our empirical studies show that many subtrees appear frequently in a repository of mathematical expressions[1]. Therefore, storing a subtree once only, and allowing other subtrees to point to it, reduces the size of the index. This also optimizes the query processing by allowing the engine to reuse partial results.

While comparing expressions, we are often interested in the result only if their similarity is greater than a certain value. In many cases, considering a bound while filling a dynamic programming matrix saves us from filling regions of the matrix whose distance values are

[1]The frequency of subtrees obeys Zipf's law. We anticipate that the frequency of subtrees for some other repositories of semistructured objects such as XHTML documents and natural language parse trees also follows Zipf's law.

greater than the bound. We propose an algorithm to efficiently calculate the similarity accordingly.

We finally show that after applying the proposed optimizations, the search result remains the same while the query processing is significantly faster and comparable to baselines that perform simple text search or database lookup.

Some of the proposed algorithms have been published in conference proceedings [60] and [63].

Optimizing Pattern Search

The pattern search algorithm (to be described in more detail in Section 3.4) requires parsing math expressions with respect to a pattern query. To optimize this approach, we identify substructurues in the query, which allows efficiently filtering expressions that do not comply to the specified pattern. This significantly reduces the number of candidates to be parsed. Details of this approach are presented in Section 4.4.

1.3 Rich Queries

Processing queries that consist of a single math expression is a basic problem that demonstrates how math expressions should be compared. In the first chapters we propose algorithms to process such queries. However, a *rich query* consists of one or more keywords in addition to one or more math expressions. Extending our solution to support rich queries is useful in many situations, hence we discuss it in Chapter 5.

The heterogeneity of data in a rich query imposes new challenges. First, various relevance scores are associated to objects depending on their types, e.g. matching mathematical expressions differs from matching keywords. Second, given a rich query, finding relevant documents requires combining the result of such individual matches. It requires appropriate optimization algorithms and indexing techniques to accommodate large collections of web pages.

In Section 5.4 we propose a scoring scheme that considers partial scores and also the difference between the text versus the math parts (and hence their associated scores) to calculate the relevance. We also propose an efficient algorithm to select the top-k documents with respect to this scoring scheme (Section 5.3). We finally demonstrate the performance of the proposed algorithms by empirically evaluating them in terms of accuracy and speed (Section 5.5).

In Section 5.3.3 we describe an approach to extend the pattern search algorithm to handle similarity constraints. Our proposed algorithm involves transforming a pattern with similarity constraints to a rich query and ranking documents accordingly. Similar to adding the full-text search capability to relational database systems, this greatly increases the expressive power of pattern queries.

1.4 Grammar Inference

Recognizing the structure of tree-structured documents can improve the performance of applications that extract, manipulate, or retrieve such information. This knowledge is represented in the form of a grammar (e.g. XML Schema or DTD).

For example, most data-extraction algorithms need to learn the structure of XHTML pages in order to detect data objects within them. A major source of data extraction failure comes from documents with complex structures, e.g. where data patterns containing optional and duplicate fields repeat arbitrarily often within a web page. Alternatively, a mathematics retrieval system needs to infer the structure of mathematical expressions in order to recognize matrices, polynomials, etc. Although the existence of a grammar can greatly improve many such systems, there has not been much effort to develop a general schema inference framework. Current proposals for such systems mostly rely on domain-specific heuristics, that cannot be effectively applied to other (inherently similar) scenarios. Moreover such approaches make simplifying assumption about the structure of presentational XML documents that affects their robustness significantly.

Therefore, we propose a grammar induction framework for such semistructured data objects in Chapter 6. More specifically, we introduce a grammar to describe semistructured data that is more expressive than DTDs or XML Schemas. Using only one sample document, we rely on detecting instances of repeating patterns to infer grammars. We do not restrict a grammar based on domain-specific assumptions, so our solution is general enough to be applied to various forms of semistructured data. We also review the related work, and discuss the advantages and disadvantages of each approach. Our proposed framework and also the inference algorithms where presented at a workshop in 2011 [61].

1.5 Extending Structural Similarity

The existence of a grammar inference algorithm for math expressions provides an opportunity to handle more complex queries for math retrieval and to optimize processing such

queries. In Chapter 7 we discuss such opportunities. In Section 7.1 we propose an approach based on inferring grammars of math expressions to extend structural similarity search. This allows us to retrieve expressions that are relevant to the query but seem to be too far from it if their grammars are not considered.

1.6 Processing Patterns With Repetitions

Our proposed optimization for processing pattern queries is based on fast filtering of math expressions that do not match the query (Section 4.4). Recognizing repetitions within a math expression provides further optimization opportunities. If a pattern query contains a repeating pattern, and the repetitions in math expressions are captured during the indexing time, we can improve the filtering criteria. In Section 7.2 we explain this approach in more details.

1.7 Thesis Outline

The rest of this thesis is organized as follows. We present definitions for basic concepts that are referred to frequently in this thesis in Chapter 2 . We describe various approaches for comparing math expressions and contrast them in Chapter 3. We then propose optimization techniques to improve the efficiency of the proposed algorithms in Chapter 4. We next consider extending the proposed approaches to process rich queries in Chapter 5. In Chapter 6 we describe a framework to infer grammars for semistructured data from limited samples, and in Chapter 7 we explain algorithms to improve math retrieval based on this framework.

Chapter 2

Definitions

In this chapter we present definitions for some concepts that will be frequently referred to in the rest of this thesis.

2.1 Basic Tree Concepts

A labeled ordered tree is a directed graph with the following properties:

- The graph is acyclic and connected.
- A *label* is assigned to each node. The label of node N is represented by $N.label$.
- There is only one node with no in-going edge called the *root*. We represent the root of a tree T as $T.root$.
- There is exactly one path from the root to each node.
- There is exactly one in-going edge to each node that is not the root. This edge connects the node to its *parent*. The node is the *child* of its parent.
- There is an order among the children of a node. We represent the children of a node N as $N.children$.
- Nodes with no out-going edges are called *leaves*.

- *Descendants* of a node N are the children of N and nodes whose parents are among the descendants of N.
- *Ancestors* of a node N are the nodes along the path from the root to N excluding N.
- The *size* of a tree is the number of its nodes. We represent the size of tree T as $size(T)$.
- A *subtree* of a node N is a descendant node M or N itself together with all its descendants and edges that connect them (also called the subtree that corresponds to M or M's subtree). It is a *proper subtree* of N if $M \neq N$.

In the remainder of this thesis, by a tree we mean a labelled ordered tree with the above properties.

A *forest* is an ordered sequence of trees (also called a *hedge*). Deleting the root of a tree results in a forest that consists of its children subtrees with the same order. Note that a single tree and the empty sequence of trees are also forests.

2.2 XML Documents

The Extensible Markup Language (XML) is designed to structure, transport and store data [19]. In various domains, XML standards are defined to provide a consensus on how data is encoded to be published or transported. Examples include XHTML and MathML [24]. A data unit encoded with XML is also called an *XML document*.

XML Tree Representation: Documents with XML markup can be naturally expressed as ordered labelled trees. A tree T is represented by $T = (V, E)$, where V represents the set of vertices and E represents the set of edges of T. A label $N.label$ is assigned to each node N, and Σ is the set of all possible labels. Two examples are shown in Figure 2.1.

A signature is a compact, convenient surrogate that is used to identify a tree. For our experiments, the signature of a tree T is computed by a conventional hash function applied to its XML string S:

$$sig(T) = S[0] * 31^{(z-1)} + S[1] * 31^{(z-2)} + \cdots + S[z-1] \tag{2.1}$$

where $S[i]$ is the i^{th} character in S and $z = |S|$.

In our experiments with the above hash function collisions are very rare, but for larger datasets appropriate collision resolution algorithms should be considered.

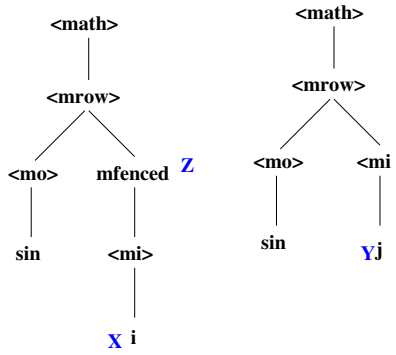


Figure 2.1: Two trees representing $sin(i)$ (left) and $sin\ j$ (right) in Presentation MathML.

2.2.1 Tree Edit Distance

Comparing two labelled ordered trees is a problem that frequently appears in diverse domains such as XML databases, computational biology, computer vision, and compiler optimization [35]. One of the very common similarity measures between two rooted ordered trees is the *tree edit distance* [17].

Consider two ordered labelled trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ and two nodes $N_1 \in V_1 \cup \{P_\phi\}$ and $N_2 \in V_2 \cup \{P_\phi\}$ where P_ϕ is a special node with special label ϵ that is not associated with any other node. An *edit operation* is a function represented by $N_1 \to N_2$ where N_1 and N_2 are not both P_ϕ. The edit operation is a deletion if N_2 is P_ϕ, it is an insertion if N_1 is P_ϕ, and a rename if N_1 and N_2 do not have the same labels. (Deleting a node N replaces the subtree rooted at N by the immediate subtrees of node N; insertion is the inverse of deletion.)

A *cost* represented by the function ω is associated with every edit operation. For example, the cost function might reflect the design goal that renaming a variable is less costly than renaming a math operator. A *transformation* from T_1 to T_2 is a sequence of edit operations that transforms T_1 to T_2. The cost of a transformation is the sum of the costs of its edit operations. The *edit distance* between T_1 and T_2 is the minimum cost of all possible transformations from T_1 to T_2.

Consider two non-empty forests F_1 and F_2 such that either F_1 or F_2 contains at least

two trees. The following recursive formula can be used to calculate the edit distance [117]:

$$dist(F_1, F_2) = min \begin{cases} dist(F_1 - u, F_2) + \omega(u \to \epsilon), \\ dist(F_1, F_2 - v) + \omega(\epsilon \to v), \\ dist(F_1 - T_u, F_2 - T_v) + dist(T_u, T_v) \end{cases}$$

$$dist(T_u, T_v) = min \begin{cases} dist(T_u - u, T_v) + \omega(u \to \epsilon), \\ dist(T_u, T_v - v) + \omega(\epsilon \to v), \\ dist(T_u - u, T_v - v) + \omega(u \to v) \end{cases} \tag{2.2}$$

where T_u and T_v are the first (leftmost) trees in F_1 and F_2 respectively, u and v are the roots of T_u and T_v respectively, and $F - n$ represents the forest produced by deleting root n from the leftmost tree in forest F. The edit distance between a forest F and the empty forest is the cost of iteratively deleting (inserting) all the nodes in F. This formulation implies that a dynamic programming algorithm can efficiently find the edit distance between two trees T_1 and T_2 by building a distance matrix.

Algorithm 1 $dist(T_1, T_2)$

1: **Input:** Two trees T_1 and T_2
2: **Output:** $dist(T_1, T_2)$: The edit distance between T_1 and T_2
3: Let SubTreeMatrix be a $(|T_1|) * (|T_2|)$ matrix
4: Set all entries of $SubTreeMatrix$ to **nil**
5: **return** dist$(T_1, T_2, SubTreeMatrix)$ (see Algorithm 2)

There are various algorithms for selecting appropriate pairs of nodes to optimize the calculation of edit distance. Zhang et al. [118] propose an algorithm with $O(n^2)$ space complexity that runs in $O(n^4)$ where $|T_1|, |T_2| = O(n)$. This algorithm runs efficiently in $O(n^2 log^2 n)$ for trees with depths of $O(logn)$. Demin et al. [35] propose another algorithm with $O(n^2)$ space complexity that runs in $O(n^3)$. Although the worst case complexity is enhanced compared to Zhang's algorithm (which is $O(n^4)$ when trees are not balanced), it happens more frequently in practice. Pawlik and Augsten propose RTED, an efficient algorithm with $O(n^2)$ space complexity and $O(n^3)$ time complexity [93]. RTED chooses the most efficient pair of nodes to minimize the number of intermediate problems. Their approach is based on limiting nodes from one of the trees with regard to specific root-leaf paths. They use an auxiliary data structure (with size $O(n^2)$) to keep track of the number of subproblems produced if a path is considered. This algorithm guarantees to choose the path that minimizes the number of subproblems.

Algorithm 2 $dist(F_1, F_2, SubTreeMatrix)$

Input: Two forests F_1 and F_2 and a $(|F_1|+1)*(|F_2|+1)$ matrix SubTreeMatrix
Output: $dist(F_1, F_2)$: The edit distance between F_1 and F_2
Let $ForestMatrix$ be a $(|F_1|+1)*(|F_2|+1)$ matrix.
$ForestMatrix[0][0] = 0$
for i = 1 to $|F_1|$ **do**
  $ForestMatrix[i][0] = ForestMatrix[i-1][0] + deleteCost(F_1[i])$
end for
for i = 1 to $|F_2|$ **do**
  $ForestMatrix[0][i] = ForestMatrix[0][i-1] + insertCost(F_2[i])$
end for
for i = 1 to $|F_1$ **do**
  for j = 1 to $|F_1$ **do**
    $v_1 = ForestMatrix[i-1][j] + deleteCost(F_1[i])$
    $v_2 = ForestMatrix[i], [j-1] + insertCost(F_2[j])$
    if F_1 and F_2 contain only one tree **and** $i = |F_1|$ and $j = |F_2|$ **then**
      $v_3 = ForestMatrix[i-1][j-1] + renameCost(F_1[i], F_2[j])$
    else
      if $SubTreeMatrix[offset(F_1[i])][offset(F_2[j])] =$ **nil then**
        $SubTreeMatrix[offset(F_1[i])][offset(F_2[j])] = dist(T_{F_1[i]}, T_{F_2[j]})$
      end if
      $v_3 = ForestMatrix[i - |T_{F_1[i]}|][j - |T_{F_2[j]}|]$
        $+SubTreeMatrix[offset(F_1[i])][offset(F_2[j])]$
    end if
    $ForestMatrix[i][j] = min\{v_1, v_2, v_3\}$.
  end for
end for
return $ForestMatrix[|F_1|][|F_2|]$

To calculate the edit distance of two trees with respect to Equation 2.2, we consider Algorithms 1 and 2 which are based on Zhang's algorithm [118]. Given a forest F, $F[i]$ is the i^{th} node in the forest with respect to a post-order traversal. For a node n, *offset(n)* is its post-order rank in the original tree. $SubTreeMatrix$ contains the edit distance of pairs of subtrees of the original tree. For example $SubTreeMatrix[i][j]$ contains the edit distance of the corresponding subtrees for the i^{th} node of T_1 and the j^{th} node of T_2 according to their post-order rank.

2.3 Mathematical Expressions

Definition 1 (Mathematical Expressions)**.** *A mathematical expression (or math expression) is a finite combination of symbols that is formed according to some context-dependent rules. Symbols can designate numbers (constants), variables, operations, functions, and other mathematical entities.*

2.3.1 Mathematical Markup Language (MathML)

The approaches to encode and represent a mathematical expression can be divided into two main groups:

1. Content-based: Semantics of symbols and their interactions are encoded. *Content MathML* [24] and *OpenMath* [21] belong to this group.

2. Presentation-based: Expressions are encoded with respect to their appearance. Examples include images of expressions, *Presentation MathML* [24], and LaTeX [45].

MathML is an XML application that is increasingly used to describe mathematical notation. It is part of the W3C recommendation that provides tools to capture both the structure (Presentation MathML) and content (Content MathML). MathML allows mathematics to be served, received, and processed on the Web, as HTML has enabled this functionality for text. Presentation MathML is increasingly used to publish mathematics information, and many web browsers support it. There are various tools to translate mathematical expressions from other languages, including LaTeX, into Presentation MathML. Moreover, Presentation MathML expressions can be processed by parsers and other applications for XML documents. Hence, we assume mathematical expressions are encoded with Presentation MathML unless otherwise specified.

```
<apply>
     <times/>
     <cn>
          2
     </cn>
     <apply>
          <plus/>
          <ci>
               x
          </ci>
          <apply>
               <times/>
               <cn>
                    3
               </cn>
               <ci>
                    y
               </ci>
          </apply>
     </apply>
</apply>
```

```
<mrow>
     <mn>
          2
     </mn>
     <mfenced>
          <mi>
               x
          </mi>
          <mo>
               +
          </mo>
          <mn>
               3
          </mn>
          <mi>
               y
          </mi>
     </mfenced>
</mrow>
```

Figure 2.2: Content MathML (left) vs. Presentation MathML (right) for $2(x + 3y)$

Example 1. *Consider* $2(x + 3y)$ *as a simple expression. The Content MathML encoding for this expression is shown in Figure 6.1 (left), and the Presentation MathML is shown in Figure 6.1 (right). Presentation MathML contains some surface semantic information. For example,* <mn> *and* <mi> *indicate that* 2 *and* 3 *are numbers and* x *and* y *are variables, respectively. However, the multiplication operator is represented by the* <times> *tag in content markup, but it is invisible and hence not shown in presentation markup. On the other hand, parentheses are not encoded in content markup because they do not carry semantic information. The plus operator is represented by the* <plus> *tag in content markup, and its operands (*x *and* $3y$*) are also clearly specified. Using presentation markup, the "+" symbol is shown where it appears in the expression, and even though it is marked as an operator, its operands are not explicitly indicated.*

A text document, such as a web page, that contains a mathematical expression is a *document with mathematical content.*

Chapter 3

Math Retrieval

Many collections of documents contain mathematical expressions. Examples include technical and educational web sites, digital libraries, and other document repositories such as patent collections. The example in Figure 3.1 shows a Wikipedia page describing a Physics concept. Much of the information in this page is presented in the form of mathematical expressions. Currently, due to the lack of an effective mathematics retrieval system, such rich mathematical knowledge is not fully exploited when searching for such documents.

Querying with mathematical expressions, and consequently retrieving relevant documents based on their mathematical content, is not straightforward:

- Mathematical expressions are objects with complex structures and rather few distinct symbols and terms. The symbols and terms alone are usually inadequate to distinguish among mathematical expressions. For example $\sum_{i=1}^{n} i$ and $\sum_{j=1}^{m} j$ are quite similar, but 2^n and n^2 are quite dissimilar even though they share the same symbols.
- Relevant mathematical expressions might include small variations in their structures or symbols. For example, $1+\sum_{i=1}^{n} i^2$ and $\sum_{j=1}^{n} j^k$ might both be useful matches to a query. On the other hand, there is no canonical form for many mathematical expressions.
- Each mathematical expression has two sides: *i*) its appearance (or *presentation*) and *ii*) its mathematical meaning (often termed its *content*). The majority of the published mathematical expressions are encoded with respect to their appearance, and most instances do not preserve much semantic information.

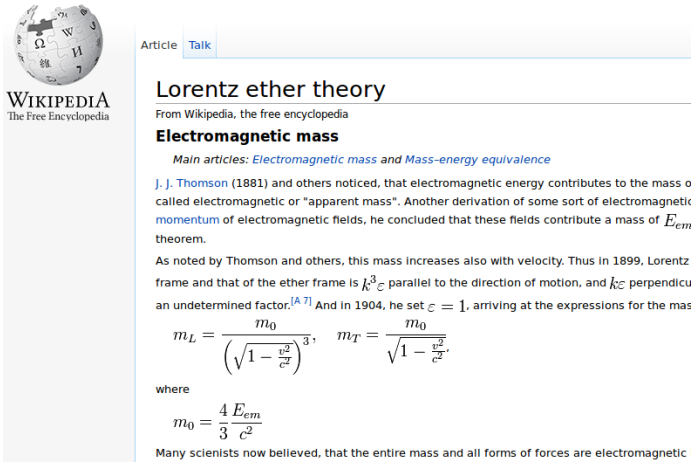


Figure 3.1: An example of a web page with mathematical content.

As is true for other retrieval systems, a mathematics search engine should be evaluated based on its usefulness, that is, how well it can satisfy users' needs. What makes mathematics retrieval distinct is the difficulty of judging which mathematical expressions are relevant and which are not. For example, a user who is interested in $\sin^2(x)$ might also be interested in $\cos^2(x)$ but not in $\sin^3(x)$. We know of no consensus for similarity of mathematical expressions in general. On the other hand, if we were to limit the search to exact matches only, many relevant expressions will be missed, and the user might need to issue too many queries to find a useful answer. For example if the user is looking for $\sum_{i=1}^{10} \frac{1}{(i+1)^2}$, then pages that contain $\sum_{j=1}^{10} \frac{1}{(j+1)^2}$, $\sum_{i=1}^{n} \frac{1}{(i+1)^2}$, and $\sum_{x=1}^{n} \frac{1}{x^2}$ probably also address her needs.

Mathematics retrieval is still at an early stage. Unfortunately, content-based mathematics retrieval systems [37, 67] are limited to resources that encode the semantics of mathematical expressions, and they do not perform well with presentation markup. The lack of content information within web pages forces a retrieval system to rely mostly on the presentation of expressions, and it is often hard to judge whether a similar-looking expression is relevant to a query.

Some systems rely on the presentation of mathematical expressions [7, 47, 86, 100, 110, 111], but they either find exact matches only or they use models that ignore the whole or parts of the structure and usually return many irrelevant results. They do not define how to measure the relevance of matched mathematical expressions, and there has not been much effort to evaluate such systems in terms of the usefulness of search results.

In this chapter we focus on the problem of matching mathematical expressions, and hence we assume that a query consists of a single expression. We typically use simple expressions in our explanations for illustrative purposes, but we do not limit the sophistication or complexity of the expressions that are to be retrieved or used as queries. Systematically addressing this problem is a prerequisite for developing systems that handle more complex queries, such as ones that consist of multiple expressions or a combination of expressions and keywords.

Because mathematical expressions are often distinguished by their structure rather than relying merely on the symbols they include, we describe two search paradigms that incorporate structure:

1. Structural similarity: The similarity of two expressions is defined as a function of their structures and the symbols they share. The similarity is used as an indication of how relevant a document containing an expression is when given another expression as a query. We propose an algorithm based on tree edit distance to calculate the similarity of two expressions. Documents are ranked with respect to the similarity of their contained mathematical expressions to the query.

2. Pattern match: As an alternative approach, a query can be represented as a pattern or template. The added expressivity of such a query language provides the user with tools to specify more details, which allows more accurate and complete results in return. On the other hand, the richness and variety of mathematical concepts implies that the query language is potentially difficult to learn and use.

If the two mentioned approaches perform equally well, the simpler query language is probably preferred; the extra cost of forming a query with the expressive query language is justified only when this expressive power results in higher-quality answers. We discuss the advantages and disadvantages of each approach, and we report on an extensive empirical study to evaluate them in terms of their ability to predict the relevance of pages containing mathematical expressions. We also describe other alternative algorithms (e.g. keyword search only, etc.) and compare them against the proposed algorithms.

The contributions of this chapter are as follows:

- We categorize existing approaches to match mathematical expressions, concentrating on two paradigms that consider the structure of expressions.
- We propose a representative system for each search paradigm.

- We evaluate and compare the described approaches through detailed user studies in real scenarios.

This is the first attempt to describe and evaluate possible solutions in a principled way [62]. Understanding the effectiveness of approaches to matching mathematical expressions is necessary for evaluating the further development of any mathematics retrieval algorithm. Hence, we believe the result of this study is an important step towards building a useful mathematics retrieval system.

In this chapter we focus on the quality of results when matching mathematical expressions. Indexing and other optimization techniques to reduce query processing time or index size [63] is discussed in the next chapter. In this chapter, we also do not address the problem of evaluating mathematical expressions, which is the goal of systems such as Wolfram Alpha [1], or Bing Math [58].

3.1 Related Work

3.1.1 Encoding Mathematical Expressions

Interchange Formats

Interchange formats are encoding schemes with the capability to create, convert, and manage data.

MathML: Mathematical Markup Language (MathML) [24] is a W3C recommended XML application for describing mathematical notations. Since MathML is an application of XML, its syntax is governed by XML syntax rules. A mathematical expression has two different, though related, aspects: the visual appearance and the mathematical content. Based on this observation, MathML defines two standard markup frameworks for representing mathematical expressions: presentation markup and content markup. In Presentation MathML, the two dimensional visual appearance of an expression is encoded. Content markup deals with the functional structure of mathematical expressions.

OpenMath: OpenMath [23] is a standard for representing mathematical expressions, allowing them to be exchanged, stored in databases, or published on the World Wide Web. OpenMath was proposed to provide a standard way of communication between mathematical applications, so unlike MathML, OpenMath is solely concerned with the content of mathematical expressions and not its presentation form.

Typesetting formats

Typesetting formats, such as LaTeX, are the encoding schemes used by typesetting systems to express mathematical expression. They are not able to encode the meaning of an expression, but instead they offer a set of symbols and provide ways to put them together in an expression.

As discussed earlier, in the remainder of this thesis we assume mathematical expressions are encoded with presentation MathML.

3.1.2 Math Retrieval Algorithms

Exact Match

Some algorithms assume expressions are available only in images, and they try to match a given query by calculating the similarity of images [113, 114]. In the best case, the performance of such algorithms is similar to *ExactMatch* algorithms, which allow for very limited variation among the expressions returned. We describe and evaluate ExactMatch algorithms further in Section 3.5.

TexSN [110] is a textual language that can be used to normalize mathematical expressions into canonical forms. After that a search is performed to find mathematical expressions that exactly match a (normalized) query. MathQL [47] and MML Query [7] propose very detailed and formal query languages through which mathematical expressions are presented as sets of symbols. To perform a search, sets of expressions containing specific symbols are selected and then intersected using relational database operations. Einwohner and Fateman [37] propose a data structure and an algorithm for searching integral tables. In this approach, pre-calculated integrals are stored in a table. A requested integral matches an entry in the table if its integrand agrees with that of the table entry up to a choice of parameters, e.g. $\frac{1}{x^2+1}$ matches $\frac{1}{x^2+a}$. We characterize all of these approaches as *NormalizedExactMatch* algorithms, which we describe and evaluate further in Section 3.5.

As shown in Section 3.5, ExactMatch and NormalizedExactMatch perform poorly in retrieving web pages with mathematical content.

Substructure match

Sojka and Liska [100] propose an algorithm that first tokenizes expressions, where a token is a subtree of the expression. Each token is next normalized with respect to various rules

(e.g. variables names are removed, number values are removed, or both), and multiple normalized copies are preserved. The resulting collection of tokens is then indexed with a text search engine. A query is similarly normalized (but not tokenized) and then matched against the index.

Similarly, Egomath [84] tranforms math expressions into tokens (that represent subexpressions), and uses a text search system to index and query them. Regardless of the tokenization details, some structure information is missed by transforming an expression into bags of tokens, which affects the accuracy of results as shown later in this paper.

MathWebSearch [67] is a semantic-based search engine for mathematical expressions. The query language is an extension to OpenMath, with some added tags and attributes, e.g. *mq:not, mq:and, mq:or*. Mathematical expressions are interpreted as prefix terms and are stored in a tree data structure called a substitution tree, where common prefixes are shared. A search is performed by traversing the tree. MathWebSearch can only process and index expressions encoded with Content MathML and OpenMath; presentation-based encoding is not well suited for use by this system.

Schellenberg et al. [98] propose extending substitution trees to expressions with LaTeX encoding. Such approaches support exact matching of expressions well, but they support partial matching only when expressions share a common part at the top of the tree. Kamali and Tompa [59] propose to allow the common parts of two expressions to appear anywhere in the trees.

We characterize such algorithms as *SubexprExactMatch* algorithms, and we show in Section 3.5 that their performance remains relatively poor.

Structure Similarity

Pillay and Zanibbi [94] propose an algorithm based on tree edit distance for combining the results of different math recognition algorithms. The goal of this approach is to enhance such algorithms to recognize hand-written expressions. Algorithms for retrieving general XML documents based on tree-edit distance have been proposed [70], and these could be adapted to match XML-encoded mathematical expressions. However, these approaches have not been thoroughly investigated for retrieving mathematical expressions. We propose an algorithm in this *SimSearch* class in Section 3.3 and show in Section 3.5 that it has a much better performance than other approaches such as exact match.

An alternative for matching based on structural similarity is to express a query in the form of a template, much as QBE does for querying relational data [125]. We describe how

templates may be used to specify precisely where variability is permitted. We review our previous work on *PatternMatch* algorithm [60] in more detail in Section 3.4 and evaluate its performance in Section 3.5.

Keyword Similarity

The maturity of keyword search algorithms has motivated some researchers to use them for mathematics retrieval [111, 114]. Such approaches typically represent a mathematics expression as a bag of words, where each word represents a mathematics symbol or function. Youssef [111] proposes an algorithm based on the vector space model to rank mathematical expressions in response to a given query. To try to accommodate the specific nature of mathematics, alternative weighting schemes are considered instead of term frequency and inverse document frequency. Nguyen et al. [90] propose another algorithm that considers a semantic encoding (Content MathML) of expressions. Each expression is represented by several textual tags, after which standard keyword search algorithms are used to search mathematical expressions. This allows supporting queries that contain both keywords and mathematical expressions and using existing IR optimizations. As shown in Section 3.5, ignoring the structure significantly affects the performance.

MathFind [86] is another math-aware search engine that encodes mathematical expressions as text objects. In MathFind, a layer is added to a typical text search engine to analyze expressions in MathML and decomposes each expression into a sequence of text encoded math fragments that are analogous to words in text documents. No further details about this search engine and its performance are published.

3.1.3 Question Answering

Some systems such as Wolfram Alpha [1], or Bing Math [58] answer mathematical expressions instead of returning pages that contain such expressions. For example they calculate the answer to $\sum_{i=1}^{4} i$ with 10 which is the result of this summation. This is a different problem that is out of the scope of this paper.

3.1.4 Querying Semistructured Data

In order to issue structured queries over XML data using XPath [9] or XQuery [18], information about the corresponding structure and metadata is required. Often such information

is not available, or data sources have different structures. Therefore, some languages are proposed to relax the need for complete specification of a tree pattern [64, 95]. For example Placek et al. [95] propose a heuristic approach for checking the containment of partially specified tree pattern queries. Alternatively, it is proposed to enable full keyword search for XML in a way that the results resemble that of a structured query. A problem that arises is to identify a meaningful unit for the search result. Note that this is not a problem in structured search because the returned information is specified precisely in structured queries. There are several proposals to determine which part of an XML tree should be returned for a given keyword query. Many consider the lowest common ancestor of the nodes that contain the queried keywords [50, 31, 108, 53, 16]. There are other attempts to determine the returned information more accurately [78]. FleXPath [5] is a system that integrates XPath querying with full-text search in a flexible way. In this approach, a framework for query relaxation is defined formally. If all keywords are not found in the location that is specified in the query, they are looked up in neighboring locations, and if found, the result will be returned with a lower rank (compared to the case where all keywords are found in the specified location). Yahia et al. [4] propose a framework for efficient XML search with complex text predicates (e.g. find all elements in XML trees that contains terms A and B such that A appears after B and they are within a window of 10 words). This is specially a challenging problem when A and B are in different subtrees, or there are more than one occurrence of A or B. This approach is based on translating queries to an algebra, called XFT, which supports rewriting optimization and efficient score computation. In summary, , approaches that consider keyword search over XML documents assume the keywords that appear in tags are enough for distinguishing between different substructures. As we will show later, this is not the case for mathematical expressions.

Whereas the database community looks at the problem of incorporating text in structured queries, the IR community looks at the alternative problem of ranking XML components according to a keyword query. Often, the goal of such proposals is to reduce the granularity of keyword search, which is traditionally at the level of documents, to finer levels (e.g. sections of an article). Bremer and Gertz [20] propose a framework for integrating document and data retrieval for XML. In this approach, document fragments are selected according to patterns specified in the query. Then, they are ranked according to query keywords using IR techniques. Lui et al. [77] propose a configurable indexing and ranking algorithm for XML documents. In this approach, an indexing algorithm, called *Ctree*, is proposed to group equivalent nodes in an XML tree into one node (e.g. all authors' names are put in one node). Also a vector based ranking algorithm is proposed that calculates the weight of terms using their frequency and location in the XML tree. XSearch [31] is a search engine for XML documents that returns related fragments of an XML document

and ranks them using IR techniques.

In an object-oriented database [10], data units are represented as objects, where an object is an instance of a class. A class might be derived from other classes and inherit some of their properties. Comparing to relational databases, object-oriented databases provide a more natural and efficient way for storing and querying complex objects. Various indexing techniques are proposed for efficient processing of frequent queries [11]. In all cases, it is assumed that the structure of a class, the inheritance relationship among classes, and the query are known when building an index.

Multimedia retrieval may also be seen as a related problem to math retrieval. Proposals for multimedia retrieval mostly aim to build a unified framework for storing and querying multimedia objects. MediaLand [107] is a multimedia data management system that processes multimedia queries based on a seamless integration of various search approaches. For example it allows a user to search for image files in her personal computer the same way she searches for video files. Zwol and Apers [105] propose another multimedia retrieval algorithm that assigns to each multimedia document a set of attributes describing its contents and performs a keyword search to retrieve objects. It is assumed that the annotations are adequate to distinguish between different objects.

Tree edit distance has been previously considered for comparing XML trees [92, 34]. Such approaches are originally proposed for version control or clustering XML documents. Further research in this direction is mainly focused on modifying tree edit distance to handle special cases [103]. Such approaches do not consider domain-specific tuning (or normalizing) for document retrieval, defining and evaluating a useful similarity measure, and handling multiple XML objects in a document. In fact, it is orthogonal to a domain-specific (e.g. math) search system by providing it with alternative tree distance measures to enhance its results.

3.1.5 Evaluating Math Retrieval Systems

Very few studies consider the problem of evaluating math retrieval systems in terms of satisfying user needs. This is partly due to the lack of a consensus on the definition of the relevance of math expressions, and partly due to the lack of a clear understanding of users' needs.

Zhao et al. [121] report on the interviews of a small group of potential users to ascertain their needs. They conclude that users prefer to use keywords that describe an expression to search for it rather than specifying the expression (e.g. "binomial coefficient" instead of $\binom{n}{k}$). As we mentioned earlier, in many cases an expression is not described with keywords

or the user is not aware of such keywords. Moreover, with math expressions more details can be specified (e.g. $\binom{n^2}{n}$). Finally, user-friendly interfaces for entering math expressions, such as pen-based devices, were not widely available at the time of the interview.

Some search algorithms that compare images of expressions evaluate their systems in terms of success rate [114, 112]. In such cases, the success rate mostly captures the correctness of recognizing math expressions rather than their relevance. In other words, if an expression or subexpression is returned as the search result, and they exactly match, it is counted as a successful search.

To date, mathematics retrieval systems that perform approximate matching and then rank expressions based on their similarity to a query have not been analyzed in terms of their effectiveness: there are no experimental results comparing their ability to find relevant matches. However, the lack of their popularity may be a sign that in many situations they do not perform well.

3.2 Problem Formulation

Here we present a general definition for the search problem and the query language. Details of the query language and the way a match is defined are specific to a mathematics retrieval system. We describe several possible approaches in the following sections.

Query: The aim of a query is to describe a mathematical expression. Hence, a query is either a mathematics expression, or it specifies a pattern that describes one or more expressions.

Search problem: Given a query, the search problem is to find a list of relevant documents with mathematical content. A document is relevant if it contains an expression that is relevant to the query.

The query and all mathematical expressions are encoded with Presentation MathML. Because forming queries directly with Presentation MathML is difficult, input devices such as pen-based interfaces and tablets [79, 99] or more widely-known languages such as LaTeX could be used instead to enter a query. Automatic tools can then be applied to translate queries to Presentation MathML. Hence, regardless of the user interface, we assume the query is eventually represented in the form of Presentation MathML. Thus, this approach is appropriate for the majority of the available mathematics information on the web.

3.2.1 Discussion

In the case of text retrieval, syntactic variants of query terms can be matched through stemmers and semantic variants can be matched through ontologies. These and similar tools can improve the results of search systems. Similarly, for mathematics retrieval using mathematical equivalence rules and transforming expressions to canonical forms accordingly (e.g. "$ab + ac$" and "$a(b + c)$") can improve search results. Nevertheless, such approaches are orthogonal to our algorithms and out of the scope of this paper.

Extending the query language to cover more complex cases can increase the usefulness of a search system. For example, allowing a query to consist of multiple mathematical expressions or a combination of mathematical expressions and text keywords can increase its expressive power. Including a (symbolic) mathematics engine to calculate the answer to a mathematical query can also be used to address some users' needs. However, in this paper our primary goal is to study the usefulness of the basic search paradigms and to compare them. While such extensions are potentially useful, the effectiveness of the basic search primitives should be proved first. Hence, in this paper we only focus on the basic search paradigms. In the next chapters, we will propose extended query languages by leveraging our findings.

3.3 Similarity Search

The usual search problem is to find a ranked list of relevant documents according to a similarity function. The similarity function for mathematics considers only the mathematical content of a document, where each potentially relevant document contains at least one mathematics expression that matches the query. After the user inputs a query through a user interface, it is translated into Presentation MathML to be processed by the ranking algorithm. The result consists of a list of ranked documents sorted with respect to the similarity of their mathematical content to the query. Figure 3.2 shows the flow of data in this approach.

A general sketch for similarity search is presented in Algorithm 3. The definition of similarity between two mathematical expressions (Line 6) is a key concept that significantly affects such systems.

Just like other information retrieval systems, semantic similarity ranking is generally very useful, as it can better capture the intention of a user. Thus, the limited semantic information that is available (i.e., whether a symbol is a number, a variable, or an operator)

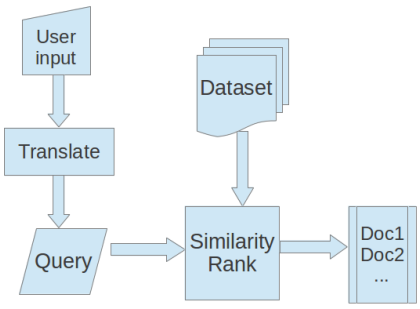


Figure 3.2: The flow of data in a mathematics retrieval system based on similarity ranking.

should also be considered to calculate similarity in order to broaden the set of potentially matching expressions.

Unfortunately, a ranking function based on more expressive semantic similarity requires that the query and the expressions be semantically encoded using a markup language such as OpenMath or Content MathML. Hence, it requires more effort from the user to form a query semantically and also requires that content markup be used to publish mathematical expressions. As stated earlier, this is generally unavailable for retrieval from the web.

Algorithm 3 Similarity Search

Input: Query q and collection D of documents.
Output: A list of documents ranked with respect to their similarity to q.
Define list L that is initially empty
for each document $d \in D$ **do**
  for each math expression E in d **do**
    Calculate the similarity of E and q and store the result
  end for
  Calculate the similarity of d and q and store the result in L
end for
Sort documents in L with respect to the calculated similarities in descending order
return L

3.3.1 A Similarity Ranking Algorithm

We now propose a similarity function that is based on tree edit distance [17], and define the similarity of a document to a math query accordingly. More specifically, we propose

appropriate similarity functions to be used in Lines 6 and 8 of Algorithm 3.

Consider two ordered labelled trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ and two nodes $N_1 \in V_1 \cup \{P_\phi\}$ and $N_2 \in V_2 \cup \{P_\phi\}$ where P_ϕ is a special node with a special label ϵ. Also assume Σ is the set of all possible labels and does not include ϵ. An edit operation is a function represented by $(N_1 \rightarrow N_2)$ where $(N_1.label, N_2.label) \in (\Sigma \cup \epsilon) \times (\Sigma \cup \epsilon) - \{(\epsilon, \epsilon)\}$. The operation is a relabelling if $N_1.label,\ N_2.label \neq \epsilon$. It is a deletion if N_1 is not the root of T_1 and $N_2.label = \epsilon$, where deleting N_1 makes the children of N_1 become the children of the parent of N_1 in place of node N_1. Finally, the operation is an insertion if $N_1.label = \epsilon$, where insertion is the mirror image of deletion. A transformation τ from T_1 to T_2 is a sequence of edit operations that transforms T_1 to T_2. To each edit operation $N_1 \rightarrow N_2$ we assign a cost $\omega(N_1 \rightarrow N_2)$. The cost of a transformation is the sum of the costs of its edit operations. The edit distance of T_1 and T_2 is defined as follows:

$$dist(T_1, T_2) = min\{cost(\tau) | \tau(T_1) = T_2\} \tag{3.1}$$

We customize the cost of an edit operation $N_1 \rightarrow N_2$ for mathematical expressions as follows:

1. If $N_1.label = N_2.label$ then $\omega(N_1 \rightarrow N_2) = 0$.
2. If N_1, N_2 are leaf nodes and $N_1.label \neq N_2.label$ and $parent(N_1).label = parent(N_2).label$ then $\omega(N_1 \rightarrow N_2) = C_{PL}(parent(N_1).label,\ N_1.label,\ N_2.label)$.
3. If N_1, N_2 are leaf nodes and $N_1.label \neq N_2.label$ and $parent(N_1).label \neq parent(N_2).label$ then $\omega(N_1 \rightarrow N_2) = C_L(N_1.label,\ N_2.label)$.
4. If N_1, N_2 are not both leaf nodes and $N_1.label \neq N_2.label$ then $cost(N_1 \rightarrow N_2) = C_I(N_1.label,\ N_2.label)$.

In the above definition, C_I and C_L, and C_{PL} are static functions that assign values to an edit operation. Their values for various inputs are shown in Table 3.1. In this table, "<mi>", "<mn>", and "<mo>" represent variables, numbers, and operators respectively; α, β, and γ are constants whose values are set based on the following observations about math expressions (Some math retrieval systems normalize math expressions based on similar observations [84].) Typically, renaming variables affects the semantics less than changing math operators. Similarly, renaming a variable should be less costly than changing a variable to a number, and renaming non-leaf nodes should be more costly that renaming leaf nodes. Therefore, we set $\alpha \leq \beta \leq \gamma$. Finding optimum values for the parameters and also further tuning the costs of edit operations is a direction of our future work.

Value of C	Condition
$C_{PL}(\text{“} < mi > \text{”}, x, y) = \alpha$	$x \neq y$
$C_{PL}(\text{“} < mn > \text{”}, x, y) = \alpha$	$x \neq y$
$C_{PL}(\text{“} < mo > \text{”}, x, y) = \alpha$	$x \neq y$ and $\{x, y\} \in \{+, -\}$
$C_L(\epsilon, x) = \beta$	
$C_L(x, \epsilon) = \beta$	
$C_L(x, y) = 2\beta$	$x \neq y$
$C_I(\epsilon, x) = \gamma$	
$C_I(x, \epsilon) = \gamma$	
$C_I(x, y) = 2\gamma$	$x \neq y$

Table 3.1: Examples of various cost values assigned to edit operations

Example 2. *Consider nodes X and Y in Figure 2.1. $X \rightarrow Y$ is a relabelling. The label of their parents, <mi>, states that they are variables. According to Table 3.1, $C_{PL}(\text{“}$ <mi> ”, “i”, “j”$) = \alpha$. Hence, $\omega(X \rightarrow Y) = \alpha$. Also, $Z \rightarrow P_\phi$ is a deletion and $\omega(Z \rightarrow P_\phi) = \gamma$. The edit distance between the two trees is equal to $\alpha + \gamma$.*

Consider two mathematical expressions E_1 and E_2 represented by trees T_1 and T_2. The similarity of the two expressions is calculated as follows:

$$sim(E_1, E_2) = 1 - \frac{dist(T_1, T_2)}{|T_1| + |T_2|} \tag{3.2}$$

where $|T|$ is the number of nodes in tree T.

There are many algorithms for calculating the edit distance between two trees. We use RTED [93] to calculate the tree edit distance.

Assume document d contains mathematical expressions $E_1 \ldots E_n$. The rank of d for a query Q is calculated with the following formula:

$$docRank(d, Q) = max\{sim(E_i, Q) | E_i \in d\} \tag{3.3}$$

that is, a document's score is equal to the similarity of the most similar expression in that document.

3.4 Pattern Search

An alternative to similarity ranking is to specify a template as the query and return expressions that match it as the search result [60]. This allows flexible matching of expressions

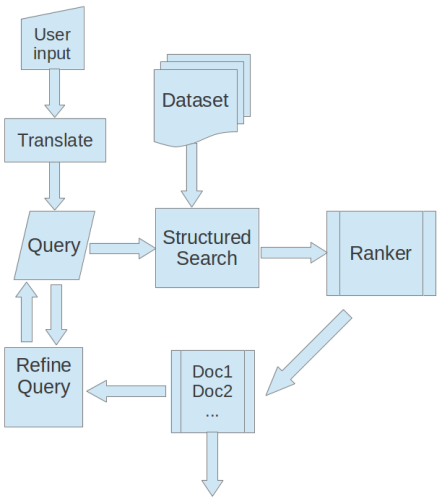


Figure 3.3: The flow of data in a mathematics retrieval system based on pattern matching.

but in a controlled way (as distinct from the similarity ranking where the user has less control on approximate matching of expressions). For example, the user can specify that she is looking for $[E]^n$ where n is a number and $[E]$ is an expression that contains $\sin(x)$. This capability is not supported by any exact matching algorithm, and the similarity search may not rank relevant expressions high enough. The approach is analogous to querying a collection of strings by specifying a partial context-free grammar, and returning strings that can be parsed by the grammar. Similarly, a template can be defined using wildcards as non-terminals, and regular expressions to describe their relationships. For example, $\sqrt{[V]}$, where $[V]$ is a wildcard that matches any variable, can be used to search for expressions that consist of the square root of a variable.

According to this paradigm, the user has more power to direct the search; hence the results are expected to be more relevant. However, the variety of wildcards and operations that are available to specify a template may result in a complex query language, which requires more effort from the user.

To find a relevant document, the user starts with a pattern to be searched. It may be necessary to tune the query, as the initial pattern may not correctly model the expression the user is looking for or it may be too general or too specific. After the results are shown, she tunes the pattern until she finds a relevant answer. A diagram of the data flow for this search paradigm is shown in Figure 3.3. Algorithm 4 presents a general sketch for this search paradigm.

In some cases, combining similarity ranking and structured search is useful. For example, assume a user is looking for expressions that contain the square root of an expression E such that E is similar to $sin(x)$. In this case, $\sqrt{sin(x)}$ is a better match than $\sqrt{sin(\frac{x}{2}+1)}$, and while the latter still complies with the pattern, $\sqrt{sin(x)}+1$ is not match. This search paradigm has been adopted in other contexts, for example where relational data is ranked with respect to an IR ranking algorithm. Examples include full-text search capability of relational databases [51], XQuery with full-text capability [3] or other keyword search features on structured data [52]. Adding this capability to a search system can increase its flexibility to capture relevant results. On the other hand, the effectiveness of a similarity ranking algorithm and the correctness of results must still be investigated.

In conclusion, this search paradigm is suitable for specialized search where the user can be expected to put more effort to form a query and get better results in return.

Algorithm 4 Structured Search

```
1: Input: Query q and collection D of documents.
2: Output: A ranked list of documents that match the query.
3: Define list L that is initially empty
4: for each document d ∈ D do
5:    for each math expression E in d do
6:       if E matches q then
7:          put d in L
8:       end if
9:    end for
10: end for
11: Sort documents in L with respect to ranking criteria
12: return L
```

3.4.1 A Model Query Language

In this section we propose a query language for pattern search and an algorithm for matching and looking up a query.

A query is expressed as a pattern consisting of a mathematical expression augmented with *wild cards*, optional parts, and constraints in the form of *where clauses*. A query matches an expression (Algorithm 4-Line 6) as follows. A wild card represents a slot that will match any subtree of the appropriate type, where $[V_i]$ matches any variable, $[N_i]$ matches any number, $[O_i]$ matches any operator, and $[E_i]$ matches any expression. A wild

card's index i is an optional natural number such that if two or more wild cards share the same type and index, they must match identical subtrees. Wild cards with no index are unconstrained.

Example 3. *The query* $x^{[N_1]} - y^{[N_1]}$ *matches* $x^2 - y^2$ *and* $x^5 - y^5$ *but not* $x^2 - y^3$*, whereas either of the queries* $x^{[N_1]} - y^{[N_2]}$ *or* $x^{[N]} - y^{[N]}$ *matches all three.*

Optional parts are enclosed by braces and they may appear in some matching expressions. Similar to optionals, other regular expression operators such as disjunctive or repetition operators may also be defined.

Example 4. $x^2\{+[N]\}$ *matches* x^2 *and* $x^2 + 1$ *but not* $x^2 + y$ *or* $x^2 - 1$.

Constraints can be specified for wild cards in a query using a "where" clause, as follows:

- Number wild cards can be constrained to a specific range or to a domain, which can be specified using a context-free grammar.
- Variable wild cards can be constrained to a restricted set of possible names.
- Operator wild cards can be constrained to a restricted set of operators.
- Expression wild cards can be constrained to contain a given subexpression, which can in turn include further wild cards and constraints.

Example 5.

- *Query "*$[E]^2[O1]3$ **where** $O \in \{+, -\}$*" matches* $x^2 + 3$ *and* $(x + 1)^2 - 3$ *but not* $x^2 \times 3$.
- *Query "*$x^{[N1]}$ **where** $1 \leq N1 \leq 5$*" matches* x^2 *but not* x^9 *or* x^{-1}.
- *Query "*$[E1] - 2$ **where** $[E1]$ *contains* x^2*" matches* $x^2 - 2$ *and* $\log(x^2 + 3y) - 2$ *but not* $x - 2$ *or* $y^2 - 2$.
- *Query "*$[E1]$ **where** $E1$ *contains* $\log_2([V])$*" matches all expressions that include a base 2 logarithm of a variable.*
- *Query "*$\sqrt{[E_1]}$ **where** E_1 *similar to* $\sin(x)$*" matches both* $\sqrt{sin(x)}$ *and* $\sqrt{sin(x+1)}$ *(ranking the first expression higher) but not* $\sqrt{\sin(x)} + 1$.

In our experiments we assume a pattern does not contain a similarity constraint. Otherwise, pattern search would be a generalized form of the similarity search approach, which makes it hard to compare them. Moreover, ranking documents with respect to a pattern query that contains multiple similarity constraints is a complex problem that should be addressed after the more basic problem of capturing the similarity of two math expressions (discussed in this paper) is addressed. This problem is a direction of our future work.

Pattern Matching Algorithm

In this section we describe how a query is matched against a math expression. As mentioned, similar to an expression, a query is represented as a MathML tree, with some extra tags that represent wild cards and regular expression operators such as optionals. We represent wildcards by a special node with tag "<wild>". We also mark regular expression operators with special tags or flags. For instance we mark an optional subtree with a special flag that is stored in its root. An example is shown in Figure 3.4.

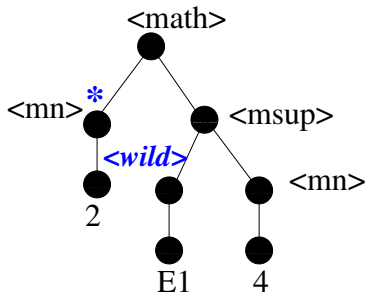


Figure 3.4: A modified query tree representing $\{2\}[E1]^4$.

In the rest of this section we assume a pattern does not contain a similarity constraint (e.g. $[E]^2$ where $[E]$ is similar to $\sin(x)$). Hence, whereas the algorithm described in Section 3.3.1 assigns a score to a document that represents how well it matches the query, in this approach a document either matches a query or it does not. We will later extend this approach to handle the similarity constraint in Section 5.3.3.

To match a query against an expression, we first compare their roots tags. If the root of the query is not a wildcard, and its tag matches the root of the expression, we parse the children of the expression with respect to the sequence of the children of the query root and the regular expression operators. We recursively match the subtrees that correspond to pairs of nodes matched by the parser.

If the root of Q is a wild card, we evaluate the match as follows. First if the wild card has an index, e.g. $E2$ or $V3$, we need to determine whether it has already been bound to a subtree because of a previous match having been made when matching another part of the query. If an expression, E', is already bound to the wildcard represented by Q, then Q matches E only if E and E' are equal, i.e. have the same signatures. If no expression has previously been bound to Q, we need to compare the types of values at the roots. For example if Q is a number wild card and E is not a number (its root's label is not "<mn>") then the result is false. Similarly, variable, operator, and expression wild cards must match variables, operators, and expressions, respectively. Otherwise, if there are no constraints on the wild card, we return true.

Assume Q is an operator wild card with the constraint that it should belong to a specific set of operators, S. It matches E only if $E.root$ is "<mo>" and the label of its child is in S. Similarly, if Q is a number or variable wild card, we check E against the constraint. If Q is an expression wild card and there is a constraint that E should contain Q', we match all subtrees of E against Q' and return true as soon as a match is found; otherwise, if no match is found, we return false. Matching an expression containment constraint is detailed in Algorithm 5.

Algorithm 5 $submatch(Q', E)$

```
1: Input: Query, Q', and Expression, E.
2: Output: true if Q' matches a subtree in E and false otherwise
3: if match(Q', E) then
4:    return true
5: end if
6: for i := 1 to C_E do
7:    if submatch(Q, E[i]) then
8:       return true
9:    end if
10: end for
11: return false
```

3.4.2 Query Processing

A query is processed by trying to match it against the stored expressions by parsing them with respect to the query. Each document that contains a match is included in the search result.

Algorithm 6 $match(Q, E)$

```
1: Input: Expression, E, and Query, Q.
2: Output: true if Q and E match and false otherwise
3: opt = number of optional subtrees in Q.children.
4: if Q is a wild card then
5:   if Q is bound to subtree E' then
6:     return true if E and E' are equal and false otherwise
7:   end if
8:   if the type of wild card matches E.root then
9:     if all constraints on the wild card are satisfied then
10:      if Q has an index then
11:        Bind E to the wild card designated by Q
12:      end if
13:      return true
14:    else
15:      return false
16:    end if
17:  else
18:    return false
19:  end if
20: end if
21: if E and Q are the same then
22:   return true
23: end if
24: if E.root.label ≠ Q.root.label then
25:   return false
26: end if
27: Look up Q in the cache of E.
28: if Q is found then
29:   return the cached match result
30: else
31:   return parse(Q[1..C_Q], E[1..C_E])
32: end if
```

While similarity ranking is in fact an information retrieval approach to the problem, pattern search resembles a database look-up. Therefore, the result of this search paradigm is a list of documents with expressions that match the query. To rank documents in the list (Algorithm 4-Line 11), a ranking criterion should be considered. In our implementation we sort results with respect to the sizes of the matched expressions in increasing order.

3.5 Experiments

In this section we present the results of our empirical evaluation of the described approaches.The goals of this evaluation are:

- To investigate how well an algorithm satisfies users' needs when searching for mathematical expressions.
- To compare diverse approaches to searching for mathematical content.
- To investigate how much effort is needed to form a suitable query.

3.5.1 Alternative Algorithms

In our experiments we consider the following specific algorithms:

- *TextSearch:* The query and expressions are treated as bags of words (nodes' labels in their XML trees). A standard text search algorithm is used for ranking documents according to a given query[1].
- *ExactMatch:* An expression is reported as a search result only if it matches a given query exactly. Results are ranked with respect to the alphabetic order of the name of their corresponding documents.
- *NormalizedExactMatch:* Some normalization is performed on the query and on the stored expressions: in particular, we ignore specific numbers, variables, and operators by removing all leaf nodes. The normalized expressions are searched and ranked according to the ExactMatch algorithm.

[1]We used Apache Lucene in our implementation.

- *SubexprExactMatch:* An expression is returned as a search result if at least one of its subexpressions exactly matches the query. Results are ranked by increasing sizes of their DOM trees.
- *NormalizedSubExactMatch:* Normalization is done on the query and on the stored expressions as for NormalizedExactMatch, and an expression is returned as a search result if one of its normalized subexpressions matches the normalized query.
- *MIaS*: As described in Section 3.1, subtrees are normalized and transformed into tokens and a text search engine is used to index and retrieve them [100].
- *SimSearch:* Expressions are matched against a query according to the algorithm described in Section 3.3. The costs of edit operations (α, β, and γ in Table 3.1) are all equal to one.
- *PatternSearch:* Expressions are matched against a pattern according to the algorithm described in Section 3.4. Like SubexprExactMatch, results are ranked with respect to the sizes of their DOM trees.

Note that among the above algorithms, the results of ExactMatch are subsets of the results of TextSearch, NormalizedExactMatch, and SubexprExactMatch.

3.5.2 Experiment Setup

Data Collection

For our experiments we use a collection of web pages with mathematical content. We collected pages from the Wikipedia and DLMF (Digital Library of Mathematics Functions) websites. Wikipedia pages contain images of expressions annotated with equivalent LaTeX encodings of the expressions. We extracted such annotations and translated them into Presentation MatchML using Tralics [46]. DLMF pages use Presentation MathML to represent mathematical expressions. Statistics summarizing this dataset are presented in Table 3.2.

Queries

To evaluate the described algorithms we prepared two sets of queries as follows.

- *Interview:* We invited a wide range of students and researchers to participate in our study. They were asked to try our system and search for mathematical expressions of potential interest to them in practical situations. They could also provide us with their feedback about the quality of results after each search. The total number of participants is 52, but as we will explain later, some queries for which the user did not provide relevance feedback were rejected. Because human participants were involved in this study, clearance was obtained through the Office of Research Ethics at the University of Waterloo.

- *Mathematics forum:* People often use mathematics forums in order to ask questions or discuss math-related topics. Many discussion threads can be described with a query that consists of a single math expression. Usually, by reading the rest of the thread and responses, the exact intention of the user is clear. This allows us to manually judge if a given expression, together with the page that contains it, can answer the information need of the user who started the thread. We manually read such discussions and gathered a collection of queries.

The research investigator created the precise query formulations for PatternSearch. Thus the experimental results reflect search environments in which queries are formed reasonably well by an experienced user.

Table 3.3 summarizes statistics about the queries, where the number of nodes in the query tree is used to represent query size. For the sake of reproducibility and as a basis for further evaluation of various search paradigms, both the dataset and the complete set of queries can be obtained from the authors upon request[2]. Some sample queries from each dataset are shown in Table 3.4.

[2]The query collections with examples of matching documents are publicly available at *http://mathretrieval.uwaterloo.ca/queries.xhtml.*

	Wikipedia	DLMF	Total
Number of pages	44,368	1,550	45,918
Number of expressions	611,210	252,148	863,358
Average size of expressions	28.3	17.6	25.2
Maximum size of expressions	578	223	578

Table 3.2: Dataset statistics

Methodology

For each query we use each algorithm to search the dataset, and only consider the top 10 results.

The way we collected queries ensures that a user's information needs are clear, which allows us to judge if a match is actually relevant or not. Searches for which we do not have a user's relevance feedback (i.e., Forum queries) require that we manually judge results. Hence, for Forum queries we consider discussion threads that clearly describe an information need with no ambiguity. For example a discussion thread might start with this question: "prove that $F_n^2 - F_{n-1}F_{n-2} = (-1)^{n-1}$ where F_n is the n^{th} Fibonacci number". A search result page is considered relevant if it satisfies the information need that is inferred from the thread. If a page contains data that can be clearly used to answer the query, we judge it as relevant. Note that a page may contain an exact match to a query, but it still does not answer the information need, hence we assume it is irrelevant (e.g. if a page contains the same expression as in the previous example, but F_n is not a Fibonacci number, or it does not contain any information that helps to prove it.).

In order to ensure the judgment is not biased, for the forum queries we asked two graduate students to judge the relevance of the results to the query based on their understandings from the content of the corresponding discussion thread. We exclude queries for which they disagree on the relevance of results. For the interview queries the user is explicitly asked to rate the results by pushing *like* or *dislike* buttons for each search result. To ensure that the user did not randomly click on such buttons, for some queries we showed results that are obviously far from the query. We excluded cases where the user rated all results including the random ones as relevant.

As mentioned earlier, for PatternSearch the query may be refined repeatedly unless appropriate results are returned or the query is refined a certain number of times and the user gives up (Figure 3.3). Hence, unless otherwise specified, the results for PatternSearch are presented with respect to the final refined query. For other algorithms however, refining a query is often not necessary or effective, and the results are shown for the original query.

Evaluation measures

NFR: A search fails if fewer than 10 results are returned (including nothing returned), and none of them is relevant. Non-Failure-Rate (NFR) is the number of searches that do not fail divided by the total number of searches:

$$NFR = \frac{|\{q \in Q | searching\ q\ does\ not\ fail\}|}{|Q|} \tag{3.4}$$

MRR: The rank of the first correct answer is a representative metric for the success of a mathematics search. Hence, for each search we consider the *Reciprocal Rank* (RR), that is, the inverse of the rank of the first relevant answer. For example if the first correct answer is ranked second, the reciprocal rank of the search is $\frac{1}{2}$. The *Mean Reciprocal Rank* (MRR) is the average reciprocal rank for all queries:

$$MRR = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{C(q)} \tag{3.5}$$

where Q is the collection of queries, and $C(q)$ is the rank of the first relevant answer for query q.

If 10 results are returned but no relevant document is among them, we optimistically assume that the search did not fail and that the rank of the first relevant document is 11. This approach distinguishes the situation where it is known that an answer cannot be found (failure) from the one in which an answer is not returned within the top 10 results but might have been returned in a longer list. If a search fails, we do not include it for calculating MRR.

Rewrite-Rate: It often happens that a search is not successful, and a user must rewrite the query to find a relevant result. For each search algorithm, starting from an initial query, we count how many times a query was rewritten to obtain a relevant answer among the top 10 results. This information is obtained from the search logs. For forum queries, we assume that the user gives up after five tries, and the search fails or no relevant result is found. The average number of rewrites for all queries is the *rewrite rate* of an algorithm.

Other measures such as *Mean Average Precision* (MAP) could alternatively be considered. However, for our data and query collections, MRR seems to be a better choice. In most cases, there are a few (often one) relevant documents for the query. Hence, MRR can better reflect the accuracy of algorithms. Recall that some of the baselines (e.g. pattern search and substructure search) deploy database operators to perform a search, while some other ones (e.g. structural similarity and keyword search) use IR techniques. Hence, because such approaches are inherently different, it is important to consider measures that fairly compare them. MRR and NFR together provide an indicative measure of the accuracy of such algorithms.

	Interview	Math Forum	Total
Number of queries	45	53	98
Average size of queries	14.2	23.8	19.4

Table 3.3: Query statistics

Interview	Math Forum
$\sum_{x=1}^{\infty} \frac{1}{x}$	$(p \vee q) \wedge (p \to r) \wedge (q \to r) \to r$
$y' = -x \sin\theta + y \cos\theta$	$\arctan(x) = x - x^3/3 + x^5/5 - x^7/7 + x^9/9 \ldots$
$\sum_{i=1}^{n} \frac{n!}{i!} 2^i$	$(a^2 + b^2 + c^2)^2 = 2(a^4 + b^4 + c^4)$
$(a-b)^2 = a^2 + b^2 - 2ab$	$T(n) = T(n-1) + T(n-2)$
$\exp \sum_t \lambda f(t, Q, t) \frac{1}{z_\lambda(Q)}$	$\log(x+3) + \log x$
$\frac{x}{1+\sin x}$	$\int_{-\infty}^{\infty} \exp(-r^2) dr$
$\int \frac{\sin x}{x} dx$	$\sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!}$

Table 3.4: Example queries

3.5.3 Evaluation Results

Correctness

The NFR and MRR for each algorithm are presented in Tables 3.5 and 3.6 for the Forum and Interview queries, respectively. As the results suggest, PatternSearch and SimSearch have high NFR and also high MRRs. PatternSearch has a higher MRR because irrelevant expressions are less likely to match a carefully formed pattern. On the other hand, SimSearch has a slightly higher NFR because in some cases even an experienced user may not be able to guess the pattern that will yield a correct answer. Furthermore, the next section shows that a template pattern may need to be modified several times to capture a relevant result.

Because MRR is only calculated when the search does not fail, ExactMatch has a high (in fact, perfect) MRR. However, in most cases, there is no expression that exactly matches the query, and hence no result is produced and this algorithm fails. SubexprExactMatch has a slightly better NFR, but it is still too low to satisfy many users' needs. This implies that often there are no relevant expressions or subexpressions that exactly match the query. However, in instances where a matching expression or subexpression exists, it is ranked highly by ExactMatch and SubexprExactMatch. Normalizing expressions, as done in NormalizedExactMatch and NormalizedSubExactMatch, further increases the NFR, but it also increases the chances that irrelevant expressions are matched. Because such algo-

Algorithm	NFR	MRR	p-value
SimSearch	100%	0.74	-
PatternSearch	90%	0.86	0.171
MiaS	94%	0.46	*0.002*
TextSearch	100%	0.19	*0.000*
ExactMatch	13%	1	0.186
NormalizedExactMatch	34%	0.46	*0.007*
SubexprExactMatch	18%	0.94	0.173
NormalizedSubExactMatch	41%	0.43	*0.004*

Table 3.5: Algorithms' performance for Forum Queries.

rithms do not offer an effective ranking algorithm, in many cases the most relevant results are not among the top 10 results.

Note that although the MRR of SimSearch is lower than ExactMatch and SubexprExactMatch, it has a much higher NFR as it produces some results for all queries. If we only consider queries for which there is an exact match (so ExactMatch produces at least one answer), SimSearch has an MRR that is not significantly different from that of ExactMatch. The reason is that in such cases, the structural similarity for documents that contain exact matches is 1, and such documents are ranked at the top of the results.

TextSearch has a very low MRR. Because this algorithm ignores the structure, it often does not rank a correct answer highly enough against many irrelevant expressions with similar MathML tags and symbols but different structures.

Note that although we reformulate queries only for pattern search, the structural similarity search produces results that are comparable with the results of well-formulated pattern queries. ExactMatch or NormalizedExactMatch are essentially pattern search with poorly formed queries. As shown, such algorithms produce poor results.

To show the statistical significance of the results, we use a Student's t-test on the reciprocal ranks of the queries. For each algorithm, we test whether there is a statistical difference between the reciprocal ranks of its produced results and that of SimSearch. We consider a one-tailed t-test for paired samples (i.e. only non-failed searches are considered). As the data in Tables 5.4 and 3.6 suggest, there are significant differences (at the 0.05 significance level) between the results of SimSearch and all algorithms except PatternSearch, ExactMatch, and SubexprExactMatch (for Forum queries). The reason is that in cases that such algorithms do not fail, SimSearch ranks relevant results equally well.

The F1 measure is used to create a single score that balances precision and recall for

Algorithm	NFR	MRR	p-value
SimSearch	100%	0.78	-
PatternSearch	76%	0.96	0.084
MiaS	90%	0.63	*0.014*
TextSearch	100%	0.23	*0.000*
ExactMatch	15%	1	0.211
NormalizedExactMatch	50%	0.58	*0.005*
SubexprExactMatch	30%	0.62	*0.044*
NormalizedSubExactMatch	60%	0.44	*0.009*

Table 3.6: Algorithms' performance for Interview Queries.

traditional retrieval tasks. In a similar way, we can calculate the harmonic mean of our NFR and MRR measures. The results are presented in Table 3.7. As the results suggest, the overall performance of SimSearch and PatternSearch is better than the other algorithms.

Query Rewriting

To compare how much effort is required from the user to perform a search, we look at PatternSearch in terms of its rewrite rates. Assume a user is looking for $\sum_{i=1}^{10} 2^i$, and suppose the only relevant match to this query is $\sum_{i=1}^{n} a^i$. The edit distance between the corresponding DOM trees is relatively low, and thus this answer is ranked highly by the SimSearch algorithm. For PatternSearch, however, if the user forms a query with no wild cards, it performs similarly to ExactMatch, and the correct answer is not found. The following is a plausible sequence of query refinements before an answer is found:

Algorithm	Forum	Interview
SimSearch	0.85	0.88
PatternSearch	0.88	0.85
MiaS	0.62	0.74
TextSearch	0.32	0.37
ExactMatch	0.23	0.26
NormalizedExactMatch	0.39	0.54
SubexprExactMatch	0.30	0.40
NormalizedSubExactMatch	0.42	0.51

Table 3.7: Harmonic means of NFR and MRR scores for Forum and Interview Queries.

1 -	$\sum_{i=1}^{10} 2^i$	No match!
2 -	$\sum_{[V1]=1}^{10} 2^{[V1]}$	No match!
3 -	$\sum_{[V1]=1}^{[N1]} 2^{[V1]}$	No match!
4 -	$\sum_{[V1]=1}^{[N1]} [N2]^{[V1]}$	No match!
5 -	$\sum_{[V1]=1}^{[N1]} [V2]^{[V1]}$	A match is found!

While the rewrite rate of SimSearch is always 1 (as no query rewriting is required), our application of PatternSearch required an average rewrite rate of 2.2 and 1.45 for the Forum and Interview queries respectively. As the results suggest, when using PatternSearch, each query may well be refined to obtain relevant results, and hence the user must invest more effort to find relevant documents.

Summary

In summary, simply viewing mathematics expressions as if they were conventional document fragments, as represented by TextSearch, or not allowing variations in matched expressions or subexpressions, as represented by ExactMatch and SubexprExactMatch, leads to extremely poor search results. On the other hand, SimSearch and PatternSearch perform very well: much better than the other algorithms that ignore the structure or perform exact matching only. PatternSearch may perform slightly better than SimSearch,

but the user will likely need to spend more time to tune a query pattern when using this algorithm. Reassuringly, these results are consistent across the two sources of queries.

So in conclusion, structural similarity search seems to be the best way for general users to search for mathematical expressions, but we hypothesize that pattern search may be the preferred approach for experienced users in specific domains.

3.6 Chapter Conclusions

Given a mathematics expression, finding pages with relevant mathematical content is an important problem that is the basis of many mathematics retrieval systems. Correctly predicting the relevance of mathematical expressions is a core problem that should be addressed in order to develop useful retrieval systems.

We characterized several possible approaches to this problem, and we elaborated two working systems that exploit the structure of mathematical expressions for approximate match: structural similarity search and pattern matching. We empirically showed that these two search paradigms outperform other search techniques, including the ones that perform exact matching of (normalized) expressions or subexpressions and the one that performs keyword search. We also showed that it takes more effort from the user to form queries when doing pattern search as compared to similarity search, but when relevant matches are found they are ranked somewhat higher. So in conclusion, structural similarity search seems to be the best way for general users to search for mathematical expressions, but we hypothesize that pattern search may be the preferred approach for experienced users in specific domains.

In this chapter we focused on the usability of answers and how well a search system can find relevant documents for a given query. Others may wish to re-evaluate these results using more controlled methods for assessing relevance. The study should next be extended in an ongoing effort to include new approaches as they are developed. Optimizing the proposed search techniques in terms of query processing time and index size is discussed in Chapter 4. Based on the results of this chapter, in the next chapters we propose more complex query languages to accommodate queries that consist of multiple mathematical expressions supplemented by textual keywords that might match other parts of relevant documents, or pattern queries with one or more similarity constraints.

NTCIR is an international initiative to create a public and shared infrastructure to facilitate research in Math IR. It aims to provide a test collection and a set of math tasks.

As a part of our future research, we plan to use this data (which is not yet available) to further evaluate the discussed algorithms.

Chapter 4

Optimizing Math Retrieval

In Chapter 3 we described algorithms for retrieving documents with respect to their mathematical expressions. We concluded that the most effective techniques for capturing the relevance of math expressions are: *i*) structural similarity search and *ii*) pattern search.

To be useful, besides the correctness of results (i.e. their relevance to the query), the query processing time must be kept reasonably low. However, this is difficult to achieve because calculating structural similarity of expressions is computationally expensive, and many potential expressions must be considered in response to each query. Hence, efficiently processing a query is a challenging problem that we address in this chapter.

4.1 Literature Review

Many applications depend on efficient processing of top-k queries. Because of its importance, many algorithms are proposed to optimize the processing of such queries. Top-k selection algorithms typically assume data is given in the form of one or more lists, and they aim to find the top-k items with the highest aggregate score. They normally define a stop condition, that if satisfied early, allows returning the results without processing the remaining items while guaranteeing the correct results are returned. The various approaches differ in how they access the data. Some algorithms assume both sorted and random access to data. Examples include Threshold Algorithm (TA) [38], Combined Algorithm [38], Quick-Combine[48], and KLEE [82]. Some other algorithms, including Upper and Pick [80], Mpro [26], topX [104], and Rank-Join [54], limit the random access and consider controlled random probes only. They assume at least one list supports sorted access, and random

accesses are scheduled to be performed if necessary. Upper and Pick algorithm is designed for web-accessible sources that differ in how they allow access to their data. It controls the random access by selecting best candidates based on their score upper bounds from the sorted lists.

To process pipelined queries, the random access to data is often limited or infeasible. Hence, some algorithms such as NRA [38], J^* [88], and Stream-Combine [49] are proposed that require only sorted access to the data. Algorithms that do not allow random access to data, cannot report the exact scores of objects. They instead calculate bounds on their exact scores. As we will discuss further, our intermediate lists allow both random and sorted access to data. Our algorithm also allows producing the results without fully processing all data objects.

Some top-k selection algorithms also propose further optimizations by modifying the order of polling the intermediate lists (if more than one list exists). Algorithms such as TA and TopX select lists in a round-robin manner. Li et al. [73] propose an approach based on prioritizing intermediate lists. This allows processing a query by accessing some lists more frequently, or even completely ignoring some lists if they do not contribute to the final ranking. Alternatively, to address this scheduling problem, IO-Top-k [8] proposes strategies for sorted access and a cost model for random access to data. The combined cost of random access and sorted access is considered, and algorithms to reduce this cost are proposed. Probabilistic models are used to estimate the probability that a document fits in the top-k documents based on the distribution of data in the intermediate lists. It is shown that the scheduling problem is hard, therefore it is assumed the number of intermediate lists is small.

In Section 4.3, we propose an upper bound on the similarity score of documents that can be calculated efficiently using standard keyword search. This allows forming a sorted list of documents and selecting the top-k documents. In Section 5.3 we consider the more general case where multiple intermediate lists are formed. We are able to use any top-k selection algorithm with sorted and random access to data.

Zezula et al. [115] propose a metric space approach for optimizing similarity search. A metric space must satisfy properties such as non-negativity, symmetry, identity, and triangle inequality. They propose indexing algorithms that optimizes range queries for various types of similarity search algorithms assuming the metric space conditions are satisfied. We note that tree edit distance also satisfies the metric space model. Hence this approach can be used to further optimize our algorithm. Selecting an appropriate range for a given query, and combining this approach with the optimizations that we will describe in this chapter is a direction for future work.

To process a structured query, naively traversing all paths in an XML tree is inefficient. A general indexing paradigm for semistructured data is to create a structural summary in the form of a labelled directed graph that preserves all paths in the original tree while having fewer nodes and edges. DataGuide [43] is the first attempt to create such structural summary for semistructured data. 1-Index [83] and A(k)-index [66] use structural similarity to partition the nodes of a tree into classes of bisimilar nodes and form summary graphs accordingly. To answer more complex queries that contain multiple paths and require backward traversing of some paths, The Forward and Backward (F&B) index is proposed [65]. To optimize the size of the index, some heuristics such as removing nodes with special tags and restricting the tree depth are considered. Despite using these heuristics, the size of the index in some cases is still very large in the above indexing schemes. FIX [119] is an indexing scheme that breaks a large document into twig patterns, which allows high pruning power and reduces the search space. Another family of indexing techniques is based on transforming a tree into sequences of numbers or strings and processing queries by performing operations on such sequences [40, 39, 36, 96]. Almost all of the above indexing schemes assume a query is either a single path, or it is a twig, which is unordered and is significantly smaller than the data tree. However, in some contexts such as mathematics retrieval, queries are ordered trees with rather large and complex structures. The size of a query is similar to the size of its matching expression trees. Moreover, such indexes are not optimized for similarity search.

4.2 Optimizing Structural Similarity Search

As described, a search algorithm based on the structural similarity of math expressions would be time consuming because it requires calculating the edit distances of many pairs of trees, which is computationally expensive. A naive approach is to calculate the similarity score of every document and return the top k documents as the search result. However, this naive approach depicted in Algorithm 3 performs some unnecessary computations and can be optimized as follows:

1. Calculating the similarity of the query and an expression requires finding the edit distance between their corresponding XML trees which is computationally expensive. On the other hand, it is not necessary to calculate the similarity of expressions that can be quickly seen to be too far from the query.

2. Many expressions are repeated in a collection of math expressions, and many share large overlapping sub-expressions. Hence, memoizing some partial results and reusing

them saves us from repeatedly recalculating scores.

3. Finding the exact value of the edit distance between two expressions is not necessary if it becomes apparent during the calculation of this value that it is not among the top-k.

The next three sections address these observations. Throughout this chapter, we use the definition of tree edit distance and costs given is Section 3.3.1.

4.3 Early Termination

In this section we propose a top-k selection algorithm that reduces query processing time by avoiding some unnecessary computations. More specifically, we define an upper limit on the similarity of two mathematical expressions that can be calculated efficiently, and we define a stopping condition with respect to this upper limit. Given a query, instead of calculating the similarity for all expressions, we stop when this condition is satisfied. For ease of explanation, to calculate the edit distance, we will assume that the costs of all delete and insert operations are 1 and the cost of rename is 2.

For a tree T, we designate the set of labels in T as $\tau(T) = \{\lambda(N)|N \in T\}$. For two trees, T_1 and T_2, we define *τ-difference* and *τ-intersection* as follows:

$$(T_1 -_\tau T_2) = \{N \in T_1 | \lambda(N) \notin \tau(T_2)\} \tag{4.1}$$

$$T_1 \cap_\tau T_2 = (\{N|N \in T_1\} - (T_1 -_\tau T_2)) \cup (\{N|N \in T_2\} - (T_2 -_\tau T_1)) \tag{4.2}$$

Note that both τ-difference and τ-intersection are defined over sets of nodes, not sets of labels. As a result,

$$|T_1 \cap_\tau T_2| = |T_1| - |T_1 -_\tau T_2| + |T_2| - |T_2 -_\tau T_1| \tag{4.3}$$

Consider expression E and query Q. We first calculate an upper bound on the value of $sim(E, Q)$. If the label of a node N in T_E, the XML tree of E, does not appear in T_Q, the XML tree of Q, their edit distance is at least equal to $1 + dist(T_E - N, T_Q)$ where $T_E - N$ is the tree that results from deleting N from T_E. A similar argument can be made for nodes in T_Q whose labels do not appear in T_E. Hence, the following lower bound on the edit distance of E and Q can be defined:
$dist(T_E, T_Q) \geq |T_E -_\tau T_Q| + |T_Q -_\tau T_E|$

from which an upper bound on the similarity of the two expressions is calculated using (3.2) and (4.3):

$$sim(E,Q) \leq 1 - \frac{|T_E -_\tau T_Q| + |T_Q -_\tau T_E|}{|T_E| + |T_Q|} = \frac{|T_E \cap_\tau T_Q|}{|T_E| + |T_Q|} \tag{4.4}$$

and the upper bound for the relevance of a document d to Q is calculated using (3.3):

$$docRank(d,Q) \leq upperRank(d,Q) = \max_{E_i \in d} \frac{|T_{E_i} \cap_\tau T_Q|}{|T_{E_i}| + |T_Q|} \tag{4.5}$$

We employ a keyword search algorithm to calculate $upperRank(d,Q)$ as follows. We build an inverted index on node labels, treating each expression as a bag of words. A document is a collection of such expressions (bags of words). In general the keyword search algorithm can be modified by assigning custom weights to terms to handle arbitrary edit costs.

To find the most relevant expressions, we maintain a priority queue of length k ("the top-k list"), which is initially empty. First, we rank documents with respect to $upperRank(d,Q)$ and define a cursor that iterates through the ranked documents. At each step, we calculate the relevance score of the current document with respect to the query, using (3.3), and update the top-k list if there are fewer than k elements on the list or this value is greater than the relevance score of the lowest document in the top-k list. We stop the algorithm if the cursor reaches the end of the ranked input list, or if k documents have been selected and $upperRank(d_{next},Q)$ (where d_{next} is the next document after the cursor) is smaller than the score of the lowest document in the top-k list. A summary of these steps is presented in Algorithm 7.

Algorithm 7 produces the same results as Algorithm 3 but it reduces the query processing time by avoiding some unnecessary computations. In Section 4.5 we show that this optimization significantly reduces the query processing time.

4.3.1 Compact Index

In this section we propose an indexing algorithm that *i*) reduces the space requirement and *ii*) speeds up the query processing. Our indexing algorithm is based on the observation that often many subexpressions appear repeatedly in a collection of math expressions.

Consider a collection of trees $C = \{T_1, \ldots, T_n\}$. Let $G \in_{sub} C$ denote that G is a subtree of T_i for some $T_i \in C$. The total number of subtree instances in C is equal to $|T_1| + \cdots + |T_n|$.

Algorithm 7 Similarity Search with Early Termination

1: **Input:** Query Q and collection D of documents.
2: **Output:** A ranked list of top k documents.
3: Treat Q as a bag of words and perform a keyword search to rank documents with respect to $upperRank(d, Q)$.
4: Define a cursor C pointing to the top of the ranked result.
5: Define an empty priority queue $TopK$.
6: **while** true **do**
7: $d_C \leftarrow$ the document referenced by C.
8: **if** d_C is null **or** $upperRank(d_C, Q) < \min_{d \in TopK} docRank(d, Q)$ **and** $|TopK| = k$ **then**
9: break
10: **end if**
11: Calculate $docRank(d_C, Q)$.
12: **if** $|Topk| < k$ **or** $docRank(d_C, Q) > \min_{d \in TopK} docRank(d, Q)$ **then**
13: Insert d_C in $TopK$.
14: **if** $|TopK| > k$ **then**
15: Remove document with smallest score from $TopK$.
16: **end if**
17: **end if**
18: $C \leftarrow C.next$
19: **end while**
20: **return** $TopK$

If two subtrees G_1 and G_2 represent equivalent subexpressions, where equivalence is defined as being exactly the same [1], we write $G_1 \sim G_2$. This relation partitions $\{G|G \in_{sub} C\}$ into equivalence classes. Given an arbitrary tree T, its *frequency in* C is the size of the matching equivalence class in C:

$$freq(T, C) = |\{G|G \in_{sub} C \wedge G \sim T\}| \tag{4.6}$$

We omit the second argument C when it is clear from context.

Given a collection of math expressions, we observe that many subtrees appear repeatedly in various expressions' XML trees. To confirm this, we ran experiments on a collection of more than 863,000 math expressions. Details of this collection are presented in Section 4.5.1, and the experimental confirmation is included in Section 4.5.3.

The basis of our indexing algorithm is to store each subexpression once only and to allow matching subtrees to point to them. This significantly decreases the size of the index, and as we will explain later, it also effectively speeds up the retrieval algorithm. The approach can also be combined with other optimization techniques, such as the one proposed in Section 4.3, to further decrease query processing time.

We assign a signature to each subtree such that matching subtrees have the same signatures and subtrees that do not match the same expression have different signatures. Any hash function that calculates a long bit pattern from the structure and node labels and any collision resolution method can be used for this purpose.

Our index is a table, indexed by signatures, whose entries represent unique MathML subtrees (both complete trees and proper subtrees). Each entry contains the label of the root and a list of pointers to table entries corresponding to the list of the children of the root. A data structure called *exp-info* is assigned to each expression that represents a complete tree in order to store information about documents that contain it. Each entry also contains some other information, such as the frequency of the corresponding tree in the collection.

Initially, the index is empty. We add expression trees one by one to the index. To add a tree T we first calculate its signature to index into the table. If there is a match, we return a pointer to the corresponding entry in the table. We also update the *exp-info* of T if it is a complete tree. If T is not found, we add a new entry to the table for that index, storing information such as the root's label, etc. Then, we recursively insert subtrees that correspond to the children of the root of T in the index, and insert a list of the pointers to

[1] Alternatively, equivalence can be defined as being exactly the same after some normalization which is a direction of our future work.

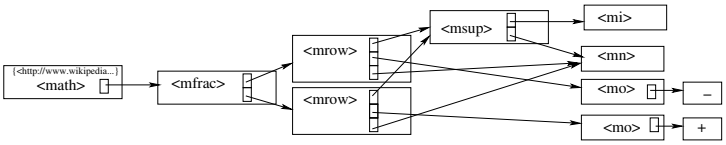


Figure 4.1: The index after $\frac{x^2-1}{x^2+1}$ is added.

their corresponding entries in the entry of T. This algorithm guarantees that each tree is inserted once only, even if it repeats. Figure 4.1 shows a fragment of the index after $\frac{x^2-1}{x^2+1}$ is added.

4.3.2 Memoization with Unconstrained Memory

Calculating the edit distance between two trees involves calculating the edit distance between many of their corresponding subtrees. Dynamic programming ensures that each pair of subtrees is compared no more than once within a single invocation of $sim(E_i, Q)$, but building the distance matrix involves calculating the similarity between each pair of subtrees, one from E_i and one from Q. As noted in the previous section, many subexpressions are shared among the mathematical expressions found in a typical document collection; building the distance matrix to compute the similarity of a query to each stored expression independently does not capitalize on earlier computations. We can reduce computation time significantly by memoizing some intermediate results for later reuse.

When calculating the edit distance between two trees, we store the result in an auxiliary data structure that we call a *distance cache*. More specifically, the cache stores triples of the form $[T_e, T_q, dist(T_e, T_q)]$ where T_e is a subtree of the expression, T_q is a subtree of the query, and $dist(T_e, T_q)$ is the edit distance between T_e and T_q. Effectively we are saving the distances computed by the dynamic programming algorithm (Equation 2.2) across similarity calls.

We implement the cache as a hash table where the key consists of the two signatures for T_e and T_q. Hence, the complexity of inserting and searching for a triple is $O(1)$. If D represents the set of all document-level expressions whose distances to Q are calculated through invocations to *docRank* in Algorithm 7, $S = \{G | G \in_{sub} D\}$, and n is the number of equivalence classes in S, the space required to store the distance cache is $O(n|Q|)$.

Each time we require the edit distance between two trees, we use the value in the cache if it is there. Otherwise we calculate the distance and store the result together with the signatures of the two subtrees in the cache.

Algorithm 8 Index Insertion $Add(E, I, d)$

1: **Input:** MathML tree for expression E, index I, and document d containing E (if E is a complete tree).
2: **Action:** Adds E to I
3: $sig \leftarrow signature(E)$
4: Search for sig in I
5: **if** sig is found **then**
6: $ent \leftarrow$ entry for sig
7: **if** E is a complete tree **then**
8: Update the exp-info of ent with d
9: **end if**
10: **return** a pointer to ent
11: **else**
12: $newEnt \leftarrow createEntry(I, sig)$
13: Insert the root label in $newEnt$
14: **if** E is a complete tree **then**
15: Update the exp-info in $newEnt$ with d
16: **end if**
17: childSeq $\leftarrow$ an empty list
18: **for** i = 1 to |children of E| **do**
19: childSeq[i] = Add(Child[i],I, null)
20: **end for**
21: Insert childSeq[i] in $newEnt$
22: **return** a pointer to $newEnt$
23: **end if**

As described in Section 2.2.1, the complexity of state-of-the-art algorithms for calculating the tree edit distance is $O(n^3)$.

Theorem 1. *Consider two labelled order trees T_1 and T_2. If the distance between each pair (T_u, T_v) ($u \in T_1$ and $v \in T_2$) of subtrees is found in the cache, the complexity of calculating their edit distance is $O(|T_1||T_2|)$.*

Proof. According to Equation 2.2, if $dist(T_u, T_v)$ is already calculated for each $u \in T_1$ and $v \in T_2$, then the complexity of the dynamic programming algorithm is equal to the size of the matrix to be filled out, which is $|T_1||T_2|$. □

Hence, if the distance of many pairs of subtrees is found in the cache, there is a noticeable gain in the query processing time. With respect to our results as explained in Section 4.5.3, many subtrees repeat frequently in a collection of math expressions, hence chances that the distance between two subtrees is found in the cache is rather high.

Memiozation with Memory Constraint

If the available memory is limited or there are too many expressions, we may not be able to store all pairs of distances as just described. However, calculating the edit distance between small trees may be sufficiently fast that there is no benefit gained by using the cache, and storing such pairs significantly increases the size of the cache. Furthermore, storing the results for rare subtrees may not be worthwhile, as the stored results may not be reused often enough to realize the benefit of using the cache.

Cost Model

The benefit of memoizing the edit distance between two trees comes from the savings in processing time if the result is found in the cache instead of being calculated for the distance matrix. Following this line of reasoning, we augment the caching criteria described above to choose which distances should be stored and which should not. We calculate the benefit of storing the triple $[T_e, T_q, dist(T_e, T_q)]$ as $benefit(T_e, T_q) = calcCost(T_e, T_q) - cacheCost(T_e, T_q)$, where $calcCost(T_e, T_q)$ and $cacheCost(T_e, T_q)$ are the costs of calculating the edit distance and looking up a value in the cache respectively.

We also wish to account for the number of times we will be able to realize the savings by reusing the value from the cache. Therefore, to each pair (T_e, T_q), we assign a weight $weight(T_e, T_q)$ that reflects the frequency of occurrence of that pair. We suggest how to compute the weights below.

Problem Statement

Consider a set of tree pairs $P = \{(T_e^1, T_q^1), \ldots, (T_e^n, T_q^n)\}$ and a space constraint that allows $\mathcal{C}$ triples to be cached. Our task is then to select a set of subtree pairs
$\mathcal{H}^* = \arg\max_{\mathcal{H}} \sum_{(T_e^i, T_q^i) \in \mathcal{H}} \mathit{weight}(T_e^i, T_q^i)\ \mathit{benefit}(T_e^i, T_q^i)$
such that
$|\mathcal{H}^*| \leq \mathcal{C}$.

On-the-fly Cache

If we are given the set P, the problem is easily solved by choosing the $\mathcal{C}$ triples having the highest values for $\mathit{weight}(T_e^i, T_q^i)\ \mathit{benefit}(T_e^i, T_q^i)$. However, Algorithm 7 maintains a sorted list of expressions, and starting from the head of the list calculates the similarity of each expression to Q. Thus, we cannot predict exactly which pair of subtrees will be compared before the algorithm stops.

We need to assign the weight for a pair of subtrees that reflects the number of times that pair will be needed for filling a dynamic programming matrix during the remainder of the execution of Algorithm 7. Consider the following motivating example:

Example 6. *Assume $\mathit{freq}(T_e, D) = 100$, and $\mathit{freq}(T_q, \{T_Q\}) = 1$. The similarity between the expressions represented by T_e and T_q will be calculated at most 100 times by Algorithm 7. While processing the query, if the edit distance function has already been called to fill 99 distance matrices for this pair, it will be called at most once more for the rest of the query processing. Caching the edit distance between T_e and T_q at this point is not likely to be as cost-effective as caching the distance for another pair of trees if those trees might still be compared 10 more times during query processing.*

We want to assign a weight to each pair that reflects this declining benefit. However, we cannot afford to store frequencies for every pair of subtrees (otherwise we could store the distances instead). Therefore, we estimate the frequencies based on the frequencies for each subtree independently.

Note that T_e matches $\mathit{freq}(T_e, D)$ subtrees of the expressions in the collection and requires up to $|T_Q|$ entries to be made in the distance matrix during dynamic programming. We augment the index described above by adding fields freq_D and freq_{cur} to each node to store the frequency of that subexpression in the document collection together with a variant of that frequency, both initialized to be equal to $\mathit{freq}(T_e, D)$ for the node corresponding to T_e. Whenever we require a value for $\mathit{dist}(T_e, T_q)$, we calculate its score as the weighted

benefit based on expected re-use as $score(T_e, T_q) = freq_{cur}(T_e)\, freq(T_q, \{T_Q\})\, benefit(T_e, T_q)$ where $freq(T_q, \{T_Q\})$ is the number of subtrees in the XML tree of Q that match T_q. We also save the score in the cache along with the distance, and update $freq_{cur}(T_e)$ with the value $freq_{cur}(T_e) - \frac{1}{|T_Q|}$ to reflect the maximum number of times T_e might still be required in a distance computation. We then set $score(T_e, T_q)$ to the value of the weight at the time $dist(T_e, T_q)$ was most recently evaluated.

Algorithm 9 details how the scores for each pair of trees is calculated and used to manage a limited cache. A priority queue maintains the most promising $\mathcal{M}$ pairs in the cache as similarity search progresses. Thus the cache stores quadruples $[s_e, s_q, dist(T_e, T_q), score(T_e, T_q)]$ where s_e and s_q are the signatures for T_e and T_q respectively. Because $score(x, y)$ increases monotonically with $freq_{cur}(x)$ and $freq(y)$ and because trees cannot repeat more frequently than any of their subtrees, if $dist(T_e, T_q)$ is stored in the cache for some subtree T_e stored in the document collection and some subtree T_q of the query, then $dist(T'_e, T'_q)$ is also stored for all $T'_e \in_{sub} T_e$ and $T'_q \in_{sub} T_q$, as long as $benefit(T'_e, T'_q)$ is sufficiently high.

Algorithm 9 Calculating Edit Distance with a Limited Cache

Input: Two trees T_e and T_q, $|T_Q|$ (the number of nodes in the query tree), and cache $\mathcal{M}$ storing quadruples.
Output: $dist(T_e, T_q)$ (with side-effects on $\mathcal{M}$ and $freq_{cur}(T_e)$)
Form pair $p = (s_e, s_q)$ that consists of the signatures of T_e and T_q.
$freq_{cur}(T_e) \leftarrow freq_{cur}(T_e) - \frac{1}{|Q|}$.
$v \leftarrow freq_{cur}(T_e) * freq(T_q) * benefit(T_e, T_q)$ (the score for this pair).
if p is found in $\mathcal{M}$ **then**
  $dist \leftarrow dist(T_e, T_q)$ associated with p in $\mathcal{M}$
  Replace the matched quadruple in $\mathcal{M}$ by $(s_e, s_q, dist, v)$.
else
  $dist \leftarrow$ compute $dist(T_e, T_q)$ using the distance matrix and cache for subproblems.
  $m \leftarrow min\{score(m) | m \in \mathcal{M}\}$
  if $m < v$ **then**
    if $|\mathcal{M}| = \mathcal{C}$ **then**
      Remove the entry with minimum score from $\mathcal{M}$.
    end if
    Insert $(s_e, s_q, dist, v)$ into $\mathcal{M}$.
  end if
end if
return $dist$

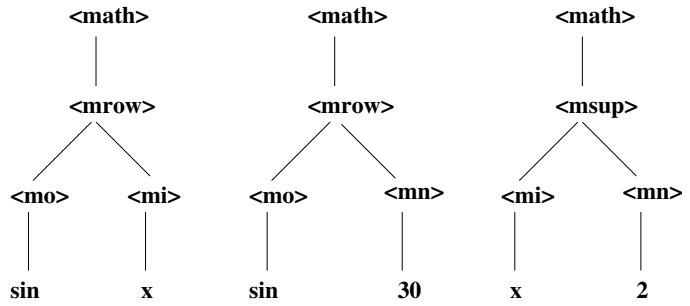


Figure 4.2: The XML tree for *left)* $\sin\ x$, *center)* $\sin 30$ and *right)* x^2.

4.3.3 Bounded Edit Distance Calculation

In Section 2.2.1 we described the edit distance between two ordered trees, and explained an algorithm to calculate it (Algorithm 1). We also described an algorithm to find the top-k relevant documents earlier in this section (Algorithm 7). We notice that we can optimize Algorithm 1 if called by Algorithm 7 through a call to the *docRank* function. More specifically, the exact value of the edit distance between the query and an expression is not needed if we can show this distance is too high for the calculated similarity to be among the top-k. In this section we describe this optimization, and in Section 4.5 we show that it reduces the total query processing time by an order of magnitude.

Assume k documents are already in the top-k list, and Algorithm 7 proceeds. With respect to this algorithm, the exact value for the score of a document d is only useful if: $docRank(d, Q) \geq \min_{c \in TopK} docRank(c, Q) = minScore$. Hence, the score of an expression E in d is only useful if:

$$1 - \frac{dist(E, Q)}{|E| + |Q|} \geq minScore \Rightarrow dist(E, Q) \leq (1 - minScore).(|E| + |Q|) = uBound(E) \tag{4.7}$$

Example 7. *Consider query* $Q = \sin x$ *and* $E = x^2$ *(Figure 4.2). Assume the lowest match in the top-k is for expression* $E' = \sin\ 30$. $sim(E', Q) = 0.83$, *and* $|Q| = |E| = 6$. *Hence* $dist(E, Q)$ *must be at most* $(1 - 0.83)(6 + 6) = 2.04$ *to be useful. In other words, if the edit distance between* E *and* Q *exceeds* 2.04*, we do not need to calculate the exact value for* $dist(E, Q)$.

With respect to Equation 2.2, the edit distance between two trees is calculated recursively from the edit distance of their subproblems. Following this equation, the value of

each intermediate result (e.g. $dist(F_1 - u, F_2) + \omega(u \to \epsilon)$ and $dist(T_u - u, T_v) + \omega(u \to \epsilon)$) is only useful when it is not greater than $uBound(E)$. Therefore, at each step when filling up the dynamic programming matrix (Algorithm 2-Line 34), we replace the value by a large number ∞ if it is greater than $uBound(E)$. We also make a recursive call to the *dist* function (Line 27) only if $ForestMatrix[i - |T_{F_1[i]}|][j - |T_{F_2[j]}|] \leq uBound(E)$.

Consider the dynamic programming matrix in Algorithm 2 (*ForestMatrix*). Assume all values in a row in *ForestMatrix* are ∞:

$$\begin{pmatrix} \cdots & \cdots & \cdots & \cdots & \cdots \\ & & \vdots & & \\ \infty & \cdots & \infty & \cdots & \infty \\ & & \vdots & & \\ \cdots & \cdots & \cdots & \cdots & \cdots \end{pmatrix}$$

We call such a row an *infinity-row*.

Theorem 2. *Assume the m^{th} row of the ForestMatrix is an infinity row ($1 \leq m < |F_2|$). The $(m+1)^{th}$ row is also an infinity row.*

Proof. Assuming $ForestMatrix[i][m] = \infty$ for $0 \leq i \leq |F_1|$, we show that $ForestMatrix[i][m+1] = \infty$ for $0 \leq i \leq |F_1|$.

$ForestMatrix[0][m+1] = ForestMatrix[0][m] + insertCost(F_2[m+1]) \geq ForestMatrix[0][m] = \infty$. Hence, $ForestMatrix[0][m+1] = \infty$.

Assuming $dist(F_1, F_2) < \min\{dist(F_1 - u, F_2) + \omega(u \to \epsilon), dist(F_1, F_2 - v) + \omega(\epsilon \to v)\}$, we prove that:
$dist(F_1 - T_u, F_2 - T_v) + dist(T_u, T_v) \geq dist(F_1 - u, F_2 - v)$.
We first show that $dist(T_u, T_v) \geq dist(T_u - u, T_v - v)$. Assume $dist(T_u, T_v) < dist(T_u - u, T_v - v)$. Therefore, from Equation 2.2, $dist(T_u, T_v) = \min\{dist(T_u - u, T_v) + \omega(u \to \epsilon), dist(T_u, T_v - v) + \omega(\epsilon \to v)\}$ and $dist(F_1, F_2) = dist(F_1 - T_u, F_2 - T_v) + \min\{dist(T_1 - u, T_2) + \omega(u \to \epsilon), dist(T_1, T_2 - v) + \omega(\epsilon \to v)\} \geq \min\{dist(F_1 - u, F_2) + \omega(u \to \epsilon), dist(F_1, F_2 - v) + \omega(\epsilon \to v)\}$ which is a contradiction.

Therefore, $dist(F_1, F_2) = dist(F_1 - T_u, F_2 - T_v) + dist(T_u, T_v) \geq dist(F_1 - T_u, F_2 - T_v) + dist(T_u - u, T_v - v) \geq dist(F_1 - u, F_2 - v)$.

$dist(F_1, F_2) = \min\{dist(F_1 - u, F_2) + \omega(u \to \epsilon), dist(F_1, F_2 - v) + \omega(\epsilon \to v), dist(F_1 - T_u, F_2 - T_v) + dist(T_u, T_v)\} \geq \min\{dist(F_1 - u, F_2) + \omega(u \to \epsilon), dist(F_1, F_2 - v) + \omega(\epsilon \to$

$v), dist(F_1{-}u, F_2{-}v)\}$. Hence, $ForestMatrix[i][m{+}1] \geq \min\{ForestMatrix[i{-}1][m{+}1]+ deleteCost(F_1[i]), ForestMatrix[i][m]{+}insertCost(F_2[m{+}1]), ForestMatrix[i{-}1][m]\} \geq \min\{ForestMatrix[i-1][m+1], \infty, \infty\} = ForestMatrix[i-1][m]$. Hence, because $ForestMatrix[0, m{+}1] = \infty$, we can prove by induction that $ForestMatrix[i][m{+}1] = \infty$ for $0 \leq i \leq |F_1|$.

□

Theorem 3. *Any value below an infinity-row is ∞.*

Proof. Following Theorem 2, we can easily prove by induction that $ForestMatrix[i][j] = \infty$ for any $j > m$. □

According to Theorem 3, any value below an infinity-row is ∞. Hence, we can stop filling the rest of the matrix as soon as we detect an infinity-row .

Example 8. *The original $ForestMatrix$ for E and Q (Example 7) is:*

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 3 & 2 & 3 & 4 \\ 3 & 4 & 5 & 4 & 3 & 4 & 5 \\ 4 & 5 & 6 & 5 & 4 & 5 & 6 \\ 5 & 6 & 7 & 6 & 5 & 6 & 7 \\ 6 & 7 & 8 & 7 & 6 & 7 & 6 \end{pmatrix}$$

Replacing the values that are greater than 2 with ∞ results in this matrix:

$$\begin{pmatrix} 0 & 1 & 2 & \infty & \infty & \infty & \infty \\ 1 & 2 & \infty & 2 & \infty & \infty & \infty \\ 2 & \infty & \infty & \infty & 2 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

The fourth row in this matrix is an infinity-row. Hence, we stop the algorithm after this row is filled.

Algorithm 10 $dist(F_1, F_2, SubTreeMatrix)$

Input: Two forests F_1 and F_2 and a $(|F_1|+1)*(|F_2|+1)$ matrix SubTreeMatrix, and $uBound$ which is the highest value for the distance.
Output: $dist(F_1, F_2)$: The edit distance between F_1 and F_2
Let ForestMatrix be a $(|F_1|+1)*(|F_2|+1)$ matrix.
ForestMatrix[0][0] = 0
for i = 1 to $|F_1|$ **do**
    $ForestMatrix[i][0] = ForestMatrix[i-1][0] + deleteCost(F_1[i])$
    if $ForestMatrix[i][0] > uBound$ **then**
        $ForestMatrix[i][0] = \infty$
    end if
end for
for i = 1 to $|F_2|$ **do**
    $ForestMatrix[0][i] = ForestMatrix[0][i-1] + insertCost(F_2[i])$
    if $ForestMatrix[0][i] > uBound$ **then**
        $ForestMatrix[0][i] = \infty$
    end if
end for
for i = 1 to $|F_1$ **do**
    $isInfinitRow \leftarrow true$
    for j = 1 to $|F_1$ **do**
        $v_1 = ForestMatrix[i-1][j] + deleteCost(F_1[i])$
        $v_2 = ForestMatrix[i], [j-1] + insertCost(F_2[j])$
        if F_1 and F_2 contain only one tree **and** $i = |F_1|$ and $j = |F_2|$ **then**
            $v_3 = ForestMatrix[i-1][j-1] + renameCost(F_1[i], F_2[j])$
        else
            if $ForestMatrix[i - |T_{F_1[i]}|][j - |T_{F_2[j]}|] < \infty$ **then**
                if $SubTreeMatrix[offset(F_1[i])][offset(F_2[j])] =$ **nil then**
                $SubTreeMatrix[offset(F_1[i])][offset(F_2[j])] = dist(T_{F_1[i]}, T_{F_2[j]})$
                end if
                $v_3 = ForestMatrix[i - |T_{F_1[i]}|][j - |T_{F_2[j]}|]$
                $+SubTreeMatrix[offset(F_1[i])][offset(F_2[j])]$
            else
                $v_3 = \infty$
            end if
        end if
        $ForestMatrix[i][j] = min\{v_1, v_2, v_3\}$.
        if $ForestMatrix[i][j] > uBound$ **then**
            $ForestMatrix[i][j] = \infty$
        else
            $isInfinitRow \leftarrow false$
        end if
    end for
    if $isInfinitRow$ **then**
        return ∞
    end if
end for
return ForestMatrix[$|F_1|$][$|F_2|$]

To apply the above optimization, we modify Algorithm 2 for calculating the edit distance as presented in Algorithm 10.

Note that as the algorithm progresses and the value of the bound decreases, an infinity row is often found at earlier stages.

4.4 Optimizing Pattern Search

In Section 3.4 we described a pattern matching strategy for math retrieval. In this section we propose an efficient algorithm to process queries represented in the form of a pattern. Our algorithm produces the same result as the pattern search algorithm described in Section 3.4.1 with a much lower query processing time.

4.4.1 Transforming Expressions

The aim of transforming an expression is to modify it so it is independent of details such as number values or variable and operator names. This will allow us to efficiently look up candidate expressions with respect to patterns that contain wildcards.

For an expression E, we create an enhanced expression, $\mathcal{H}(E)$, as follows. Leaves represent literal values such as numbers, variables, and operators (Figure 4.3-A). Hence, we remove all leaves of the corresponding tree of E. We also remove attributes that are not mathematically significant such as font sizes or white space. After these two steps, we obtain a tree that is independent of the specified details (Figure 4.3-B).

4.4.2 Building the Index

To build the index, we first create a pseudo-document for each expression in the collection as follows.

Consider an expression E in document d. We transform E as described in the previous section. For each node N in the transformed expression, we calculate the signature of the subtree rooted at N (Equation 2.1). We also consider the path from the root of E to N, and calculate its signature. The signature of a path is calculated similar to a tree (treating the path as a tree that consists of a single path).

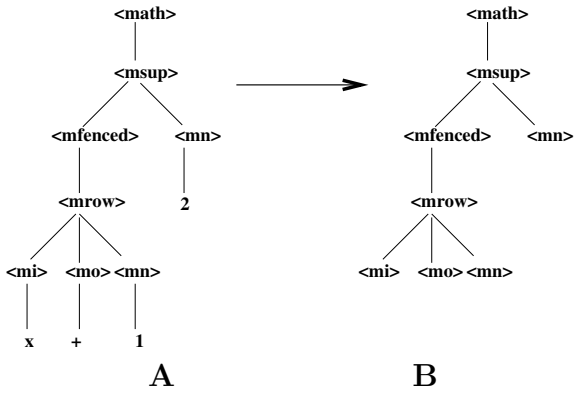


Figure 4.3: **A**) The original expression tree for $(x+1)^2$. **B**) The transformed expression.

The pseudo-document for E consists of a set of terms, where each term is the signature of a subtree or a path in its enhanced form with no duplicates.The header of the pseudo-document also contains a pointer to E, and a pointer to the page that contains E.

Similar to text documents, we build an inverted index on the terms. Using this index, we can efficiently retrieve expressions with respect to their contained terms using standard text-retrieval techniques. A summary of the steps is presented in Algorithm 11.

4.4.3 Processing a Pattern Query

Recall that a pattern query is a math expression with some further information represented in the form of wild cards and regular expression operators such as repetition or optional operators. Our optimization is based on using the index to efficiently filter expressions that do not match the query because they do not contain specific parts. This results in a set of potential matches (candidates), that will be processed further to check if they actually match the query. In the remainder of this section we elaborate on this idea.

Given a pattern query, we first transform it to obtain its enhanced form as explained in the previous section by removing leaves of the tree. An example is shown in Figure 4.4. This results in a tree that may contain wild cards or regular expression operators such as disjunctive, optional, and repetition operators.

A node is a *pseudo node* if it is represents a wild card or it is associated with a regular expression operator such as a disjunction or repetition operator, otherwise it is a *constant node*. A subtree is *maximal-constant tree* if:

Algorithm 11 Building the Index For Optimum Pattern Query Processing

```
1: Input: collection C of expressions.
2: Output: an index to facilitate processing pattern queries.
3: Let ind be an empty inverted index
4: for each expression E ∈ C do
5:    Let pd be an empty pseudo-document
6:    pd.page ← pointer to the page that contains E
7:    pd.expression ← pointer to E
8:    E′ ← transform(E)
9:    for each node N of E′ do
10:      sSig ← the signature of the subtree rooted at N.
11:      if pd does not contain sSig then
12:         Add sSig to pd
13:      end if
14:      pSig ← the signature of the path from E′.root to N.
15:      if pd does not contain pSig then
16:         Add pSig to pd
17:      end if
18:   end for
19:   Add pd to ind
20: end for
```

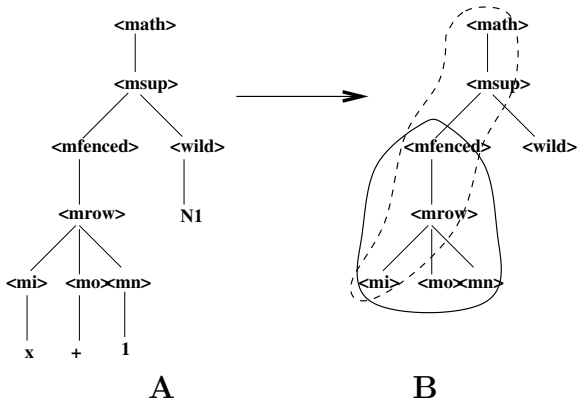


Figure 4.4: **A**) The original tree for pattern $(x+1)^{[N1]}$. **B**) The transformed pattern.

1. It consists of constant nodes only.
2. None of the immediate subtrees of the root's ancestors is constant.
3. None of its ancestors is a pseudo node.

An example of a maximal constant subtree is circled with solid line in Figure 4.4-B

A path in the tree is a *maximal-constant path* if:

1. Starts from the root.
2. Consists of constant nodes only.
3. No constant path with a longer length exists that contains all its nodes (i.e. it cannot be extended).

From the last property we can conclude that a maximal-constant path ends with a leaf node or the parent of a pseudo node. An example of a maximal-constant path is circled with dashed line in Figure 4.4-B.

We next form a *token query* that consists of a collection of tokens. Each token is the signature of a maximal-constant path, or a maximal-constant subtree. For each such path or subtree, we calculate the signature and add it to the token query if it is not added previously (to avoid duplicates).

After the token query is formed, we use the index to retrieve expressions that contain such tokens using a standard keyword search algorithm. Each retrieved expression is a candidate that should be processed further to check if it matches the pattern query.

Algorithm 12 *optimizedPatternSearch*(Q)

```
1: Input: Query, Q.
2: Output: a list of documents containing expressions that match Q.
3: Modify the query
4: T ← the tree representing the modified query
5: E ← an empty set of tokens.
6: for each maximal-constant subtree M of T do
7:    sig ← the signature of M
8:    Add sig to E
9: end for
10: for each maximal-constant path P in N do
11:    sig ← the signature of P
12:    Add sig to E
13: end for
14: candidateExprs ← textSearch(E)
15: res ← an empty list of documents
16: for entry ent in candidateExprs do
17:    E ← the expression stored in ent
18:    if match(E, Q) [Algorithm 6] then
19:       Add the document associated to ent to res
20:    end if
21: end for
22: return res
```

	Wikipedia	DLMF	Combined
Number of pages	44,368	1,550	45,918
Number of expressions	611,210	252,148	863,358
Average expression size	28.3	17.6	25.2
Maximum expression size	578	223	578

Table 4.1: Dataset statistics

Example 9. *Assume the query is* $(x+1)^{[N1]}$ *(Figure 4.4-A). The modified query is shown in Figure 4.4-B. The modified tree contains only one maximal-constant subtree (the subtree rooted at the node with tag "<mfenced>"). There are three maximal-constant paths: from the root (with tag "<math>") to nodes with tags "<mi>", "<mo>", and "<mn>". Hence, the token query contains four tokens: the signature of the subtree, and the three paths. The pseudo-document for expression* $E = (x+1)^2$ *(Figure 4.3) contains all the tokens, and hence* E *is returned as a candidate expression to be matched against the query.*

Hence, after a list of candidate expressions are obtained, we use Algorithm 6 to match each one against the query. We return documents that contain matching expressions as the search results.

4.5 Experiments

In this section we empirically evaluate the proposed optimization techniques.

4.5.1 Experiment Setup

Data Collection

For our experiments we use a collection of web pages with mathematical content. We collected pages from the Wikipedia and DLMF (Digital Library of Mathematics Functions) websites. Wikipedia pages contain images of expressions annotated with equivalent LaTeX encodings of the expressions. We extracted such annotations and translated them into Presentation MatchML using Tralics [46]. DLMF pages use Presentation MathML to represent mathematical expressions. Statistics summarizing this dataset are presented in Table 4.1.

Data and Query Collections

In our experiments we use the collection of expressions from Wikipedia and DLMF websites as described in Section 4.5.1. We also consider the Interview and the Forum query collections (Section 4.5.1).

Evaluation Measures

We evaluate the proposed algorithms using the following measures:

Query Processing Time: The time in milliseconds from when a query is submitted until the results are returned. A query is encoded with Presentation MathML and if the user interface allows other formats, the time taken to translate it is ignored. Also the network delay and the time to render results are not included. Each query is executed five times and the average time is used as its query processing time. For a collection of queries, we measure the query processing time of each and report the *average query processing time*.

Alternative Algorithms.

We further refine SimSearch to cover the following algorithms that reflect the proposed optimization techniques:

- *Unoptimized*: Each expression is stored independently. The relevance score is calculated for any expression sharing at least one tag with the query.
- *ET*: The early termination algorithm described in Sect. 4.3. Each expression is stored independently. As described, an inverted index is used to calculate upper bounds on the scores of each document, which increases the index size.
- *Compact*: Similar to unoptimized a query is processed by comparing the relevance of each document that contains an expression with at least one node whose tag appears in the query. Each subtree is stored once only to reduce the index size as described in Sect. 4.3.1.
- *Compact-ET-NMC*: The early termination algorithm with a compact index, and no memory constraint as described in Sect. 4.3.2.
- *Compact-ET-RandMC*: Similar to Compact-ET-MC, but entries are chosen at random for being assigned space in the cache.

- *Compact-ET-MC*: The early termination algorithm with a compact index and a constraint on the memory that is available during the query processing (Sect. 4.3.2). The results are presented for specific amounts of available memory separately (e.g. if the memory constraint allows storing 1000 cache entries, we use the label Compact-ET-MC-1000). We consider three values for the memory constraint: 5000, 10000, and 50000 entries.

- *Compact-ET-NMC-ETTED*: Compact-ET-NMC algorithm with modified tree edit distance calculation (Algorithm 10).

4.5.2 Methodology

Each alternative approach consists of an indexing scheme and a query processing algorithm. The performance of each approach is measured by the average query processing time. Hence, for each approach we first import the expressions in the collection and build an index. For that approach we then process queries from the query collection and calculate the described evaluation measures.

4.5.3 Index Size

In this section we investigate the repetition rate of subtrees in the collection of expressions and show how it affects the index size.

The average number of repetitions of subtrees with sizes in specific ranges is listed in Table 4.2. The average repetitions of trees whose sizes are in the range of $[1-k]$ for various values of k is shown as a graph. As the results suggest, most subtrees repeat at least a few times. Not surprisingly, for smaller subtrees the rate of repetition is higher.

Next, we compare the compact index to an index that stores each expression independently. As shown in Table 4.3, the size of the compact index (in terms of the number of nodes stored) is significantly smaller than that of the regular index.

4.5.4 Query Processing Time

Figure 4.5 shows that the early termination algorithm significantly reduces the query processing time — by a factor of 44. Using the compact index and memoizing partial results also reduces the query processing time by an additional factor of 1.5, to about .8 seconds

Size	Average repetition
1-5	325.0
6-10	10.5
11-15	3.2
16-20	2.1
21-25	1.7
26-30	1.5
> 30	1.3

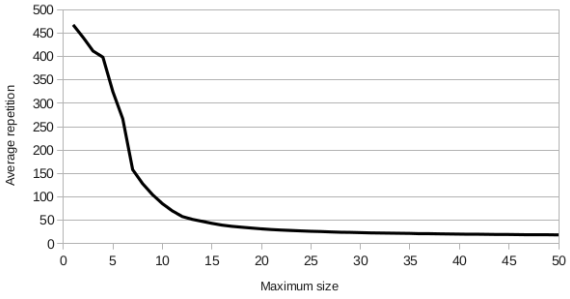


Table 4.2: Repetitions of subtrees.

	Number Of Stored Nodes
Original index	19,775,322
Compact index	1,284,701

Table 4.3: Subtree repetitions in experimental dataset and resulting index sizes.

per query on average (Note that accuracy is not affected by employing any of the optimization techniques.). Figures 4.5 and 4.6 compare the proposed approach against alternative approaches. The alternative algorithms use straightforward text search or database lookup algorithms, which result in query processing times that are two to four times faster, but at the expense of very poor accuracy. To date, these approaches have been preferred to a more elaborate similarity search, largely because the latter was deemed to be too slow to be practical. However, *Compact-ET-NMC*, which applies both early termination and memoization, has practical processing speeds and far better accuracy.

The effect of the available memory on the query processing time is investigated in Fig. 4.5.4. For higher values of the space budget, the query processing time is very similar to that of *Compact-ET-NMC*, which assumes there is no constraint on the available memory. Even for smaller values of the constraint (e.g. when we can memoize at most 5,000 intermediate results), there is a notable improvement over the performance of *ET*.

The figure also compares the performance when the available space is managed with respect to the described algorithm and when distances for pairs of trees are chosen to be cached at random. For a small space budget, caching randomly chosen pairs has little advantage over the ET algorithm, which does not use a cache. For greater values of the space budget the performance is improved compared to ET, but not as much as when caching is applied more strategically. For example, the performance of *Compact-ET-MC-50000* is very close to that of *Compact-ET-NMC*, which assumes unlimited memory is

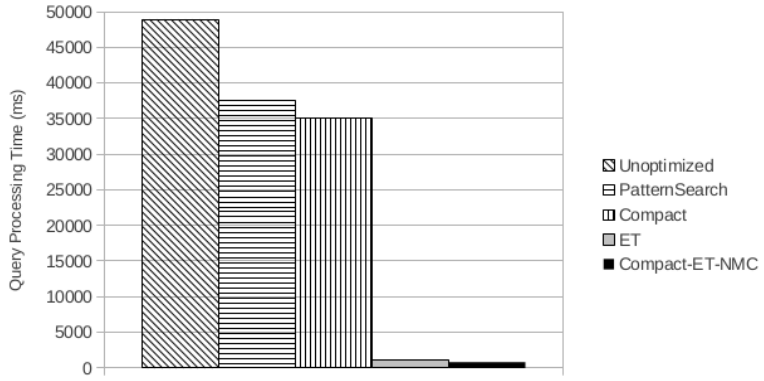


Figure 4.5: The query processing time of alternative algorithms.

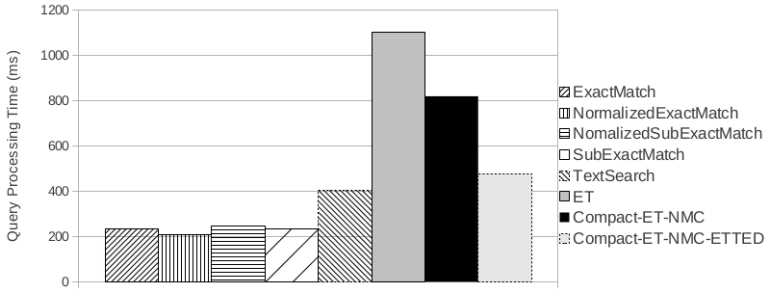


Figure 4.6: The query processing time of alternative algorithms.

available, and *Compact-ET-MC-5000* performs similarly fast as *Compact-ET-RandMC-50000* while using only a tenth of the space budget. This validates the proposed method for choosing which pairs to cache.

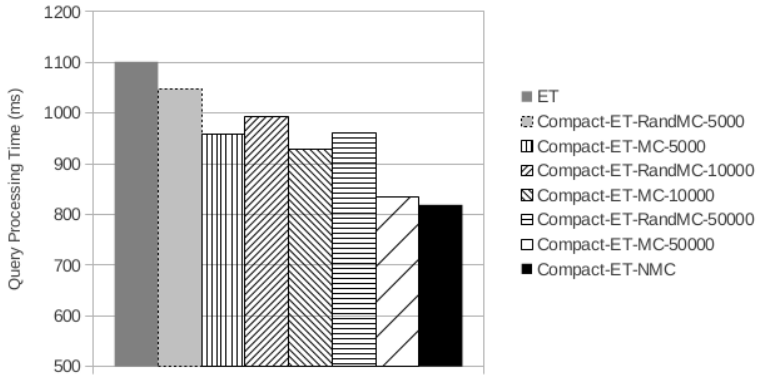


Figure 4.7: The query processing time for various space budgets and cache strategies.

Chapter 5

Rich Queries

In the previous chapters we explained how a query that represents a single math expression can be processed. We now turn to handling richer queries that contain multiple math expressions or a combination of keywords and expressions which is the aim of this chapter. Supporting queries that consist of keywords and math expressions significantly improves the retrieval of rich documents, and we refer to such queries as *rich queries*. An example of a rich query is presented below:

Example 10. *An appropriate query to retrieve pages about a probability distribution that contain a specific expression describing its mass function is:* $Q =$ *"*$\binom{n}{k}p^k(1-p)^{n-k}$ *probability distribution". This query consists of an expression and two keywords.*

The information contained in the math parts of rich documents, such as those shown in Figure 5.1, allows supporting queries in the form of math expressions to retrieve such documents in a more precise and expressive way. However, queries that consist of math expressions only fail to access the additional information embedded in the text. On the other hand, as shown in Chapter 3, by performing keyword search only, the structured information is not fully exploited, or are exploited in an undesired fashion. In summary, the advantages of supporting rich queries are as follows:

- Allowing keywords with math expressions helps overcoming *polysemy* (i.e. when two expressions with similar appearance have different meanings).
- More expressive queries help to narrow down a search when many documents contain specific expressions or terms.

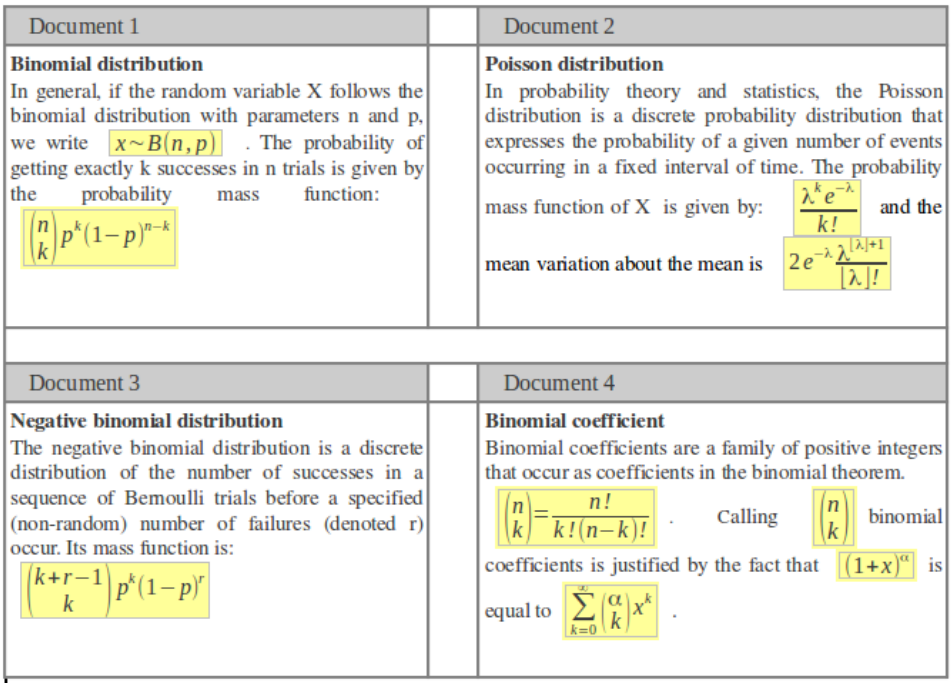


Figure 5.1: Four sample documents. Math expressions are highlighted.

- While a user may prefer to use keywords in cases where an expression can be described by keywords, she may also include math expressions to find more accurate results.

Example 11. *Assume a user is looking for documents that discuss how the intersection of two curves, $y = x^2 + 1$ and $y = 2x$, is calculated. A query that consists of only one math expression ($y = x^2 + 1$ or $y = 2x$) causes many irrelevant documents to be retrieved. On the other hand, a rich query such as $\{y = x^2 + 1,\ y = 2x,$ "intersection"$\}$ can better express her needs.*

In the rest of this chapter, we first assume a rich query consists of several math expressions only. We next consider rich queries that consist of math expressions and keywords. Finally, we consider the related problem of processing structured queries with multiple similarity constraints. In all cases, the search result is a ranked list of the k most similar documents to the query. The similarity of a document to a query will be defined in the next sections, where we also describe algorithms to rank documents efficiently.

5.1 Retrieving Objects with Text Search

In many approaches that consider retrieving pages by the objects they contain, it is assumed that the object is annotated with textual tags. Such approaches usually differ on how objects are annotated.

Zhou et al. [124] propose algorithms for exploiting social annotations for retrieving documents. They assume each document is annotated by a user, and queries and annotations have a plain text format. Given a query and a document, the ranking algorithm estimates the query language model[1] and compares it with the language model of the document, using a risk minimization technique to obtain its score. Yang et al. [109] study the problem of modelling and ranking accumulative social annotations. They assume a document is annotated by various users over time, and propose an algorithm to calculate its score. A ranking algorithm for retrieving socially annotated data based on interpolated n-grams is proposed by Sarkas et al. [97]. The aim of this algorithm is to retrieve objects such as images or videos that are described by keyword tags. Queries and tags are assumed to be bags of words.

The above algorithms assume the annotations of a document, similar to its content, have a plain text format. Hence they cannot be applied to process rich queries because math expressions should be compared differently from textual terms. Moreover, as we previously showed, and will discuss further in Section 5.5, ignoring the structure of math expressions (i.e. annotating them with the tags of their XML trees) results in a very poor accuracy.

Nie et al. [91] propose an algorithm for web object retrieval. They consider the problems of extracting object information from web pages, and exploiting such information for retrieving objects with keyword queries. Each object is described using structured data that is extracted from text with some confidence level. After the structured data is extracted, the text is not used for retrieving objects, and structured queries are considered instead. This approach is not applicable to the problem we consider in this chapter because math objects are already available and they are distinct from the text.

[1] A language model is a probability distribution of data units, e.g. words, that captures the statistical regularities of the language.

5.2 Rich Queries With Multiple Math Expressions

In this section we consider the problem of processing queries consisting of several math expressions. Note that in Chapter 3, a query is assumed to designate one math expression only, which is a special case of the query language we consider in this section.

Assume query Q consists of n math expressions: $Q = \{Q_1^M, Q_2^M, \ldots, Q_n^M\}$. Recall that according to Equation 3.3, for each $Q_j^M \in Q$:

$$docRank(d, Q_j^M) = max\{sim(E_i, Q_j^M) | E_i \in d\} \quad .$$

Assume we model the query or a document as a vector where each dimension corresponds to an expression in the query. For each expression of the query assume the corresponding value is 1 in the query vector[2], and it is $docRank(d, Q_i)$ in the document vector which is formed at query time. After that, we calculate the similarity of a document to a query based on cosine similarity as widely used in information retrieval [6] as follows:

$$docRank(d, Q) = \frac{\sum_{Q_i \in Q} docRank(d, Q_i)}{|Q|\ |d|} \tag{5.1}$$

In the above equation, $|d|$ is the number of expressions in d.

In Section 5.5, we empirically show that the above similarity measure results in an effective ranking scheme.

5.3 Efficiently Processing A Rich Math Query

In this section we describe an algorithm to process a query and rank documents based on the described scoring function (Equation 5.5). We assume a query consists of several math expressions.

To select the most relevant documents, a trivial solution is to calculate the scores for all documents in the corpus and then to choose the documents with the highest scores. Obviously, this approach does not scale well when the size of the corpus is large. Moreover, processing each component of the query independently results in a number of potentially long lists that must be joined to obtain the final scores. The existence of many such intermediate lists makes this a time consuming task.

[2] Assuming there is no prior preference on the query components, otherwise a weight could be considered.

To process a query efficiently, we first form intermediate lists for each component of the query. Each such list contains a partial score that can be calculated efficiently. After such lists are formed, we select the top-k results efficiently by defining appropriate stop-conditions and list selection criteria. An overview of these steps is shown in Figure 5.2.

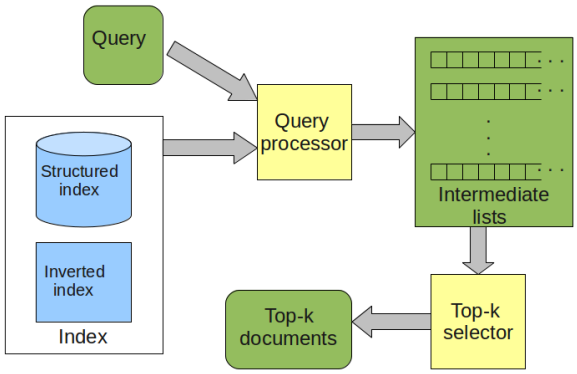


Figure 5.2: The flow of data during the query processing.

In the remainder of this section we propose efficient algorithms to create intermediate lists. We next describe a top-k selection algorithm to retrieve the most relevant documents.

5.3.1 Intermediate Lists And Values

Given a query, the similarity score for a document is the combination of its score for each query expression (Equation 5.5). Hence, to process a query, we form intermediate lists as follows. Given a query $Q = \{Q_1^M \dots Q_n^M\}$ (where Q_i^M is the i^{th} math expression of the query), we form a sorted list for each Q_i^M. We denote the list that corresponds to Q_i^M with L_i^M. Together with each document, we store a score and sort the list with respect to this score.

The score of documents in L_i^M should be $docRank(Q_i^M, d_M)$. However, as argued in Section 4.3, calculating this score for every document is not efficient. Instead, we calculate an upper bound on the value of $docRank(Q_i^M, d_M)$ and use it to sort documents in L_i^M. Similar to Equation 4.5, we calculate the upper limit on the value of $docRank(Q_i^M, d_M)$ as follows:

$$docRank(Q_i^M, d_M) \leq docRank_{UL}(d_M, Q_i^M) = \max_{E_i \in d_M} \frac{|T_{E_i} \cap_\tau T_{Q_i^M}|}{|T_{E_i}| + |T_{Q_i^M}|} \tag{5.2}$$

Recall that T_{E_i} and $T_{Q_i^M}$ in the above equation represent the XML trees of expressions E_i and Q_i^M respectively.

In summary, L_i^M stores $d.id$ and $docRank_{UL}(d_M, Q_i^M)$ (Equation 5.2) for each document d if d_M contains at least one tag from Q_i^M. This list is sorted with respect to $docRank_{UL}(d_M, Q_i^M)$.

To form the described intermediate lists efficiently, we create an inverted index (I_M) on the tags of mathematical expressions. As described in Section 4.3, I_M allows efficient ranking of documents with respect to $docRank_{UL}$ (Equation 5.2) by performing a keyword search.

5.3.2 Selecting Relevant Documents Using Intermediate Lists

Given a query, after the intermediate lists are formed, the search result, which is a ranked list of top-k documents, is found using the information in these lists. Because the intermediate lists are sorted in decreasing order of the similarity scores, and the similarity function in Equation 5.2 is increasing for each partial score, a top-k selection algorithm guarantees to return the most similar documents for a query.

As stated earlier, a naive solution is to join all lists, obtain the relevance score of each document, and finally sort the results. This algorithm is not efficient, because we only need the top-k relevant documents, and calculating the score of all documents requires unnecessary computations. Hence, we instead use a top-k selection algorithm as follows. In Section 4.1, we described various top-k selection algorithms that can be used to find the top-k results with respect to their relevance score. We use the Threshold Algorithm [38] with further optimizations in this section.

We maintain a list of documents that at each step contains the k most relevant documents processed so far. At each round, we pick an intermediate list (the order of picking lists is discussed later in this section). Next, the document at the top of the chosen list is removed, its score is calculated, and the top-k list is updated if necessary. We define a threshold ϕ and update its value at each step. The threshold is the relevance score of an imaginary document, such that the score of each of its components is the maximum in the corresponding list. This maximum score is the score of the document at the top of the list if it is not empty (it is zero otherwise). The algorithm stops when all intermediate lists are exhausted or when a *stop condition* is satisfied. That is, when the score of the lowest document in the top-k list is greater than the threshold.

The order of picking intermediate lists impacts how fast the stop condition is satisfied. In general picking the following lists more often makes the algorithm stop earlier [55]:

- Shorter lists.
- Lists whose stored probabilities have a higher variance.

Therefore, to each intermediate list L, we assign an importance grade $\mathcal{G}(L)$:

$$\mathcal{G}(L) = \frac{1}{\sqrt{|L|}}\sigma(L) \tag{5.3}$$

where $\sigma(L)$ is the standard deviation of the corresponding scores of documents in L. After all scores are calculated, at each round we assign a number of tokens to each list that is proportional to $\frac{\mathcal{G}(L)}{\sum \mathcal{G}(L_i)}$. At each round we pick a list with respect to the number of tokens assigned to it. At the end of a round, we update $\mathcal{G}$ for each list.

A summary of these steps is presented in Algorithm 13.

5.3.3 Pattern Queries With Similarity Constraint

We now describe a specific application of processing rich queries described earlier in this section. We introduced an algorithm to process pattern queries in Section 3.4.1 where we assumed a pattern does not contain a similarity constraint. Supporting similarity greatly increases the expressive power of the approach by adding all the benefits of similarity search to it. Recall that a similarity constraint is associated to an expression wildcard as follows:

Definition 2. *Consider an expression wildcard* $[E]$*. A similarity constraint for* $[E]$ *is specified as " where* $[E] \sim E'$ *" such that* E' *is a math expression.*

Because a pattern may include an arbitrary number of expression wildcards, there may also be more than one similarity constraint.

Definition 3. *A similarity constraint associated with* $[E]$ *is* ambiguous*, if* Q *can be matched against a given expression* X *such that either of two subexpressions of* X *can be matched to* $[E]$*.*

For instance, if a similarity constraint is associated to an expression wildcard that appears with another expression wildcard as disjuncts, it is ambiguous, but this is not the only such condition.

Example 12. *Consider query* $Q=$" $[E_1]?[E_2][E_3]?$ *where* $[E_2] \sim \sin(x)$. *Matching this query against expression* $E= \sin^2(y)\cos(y)$ *results in two possibilities:* $[E_2]$ *is bound to* $\sin^2(y)$ *or* $[E_2]$ *is bound to* $\cos(y)$. *The decision is made with respect to the value of their similarities.*

Processing ambiguous constraints is computationally expensive, and such cases are rather rare. Hence, we disallow query with ambiguous constraints.

To process a pattern query with similarity constraints, we first remove all such constraints. This results in a more general query that does not contain any similarity constraints. We process the refined query as described in Section 4.4.3. The result of this step is a list of expressions that match the query exactly, but with some constraints underspecified.

Next, we rank documents associated with the list of the matched expressions with respect to the similarity constraints in the query. We form a rich query R_Q that consists of a collection of expressions that correspond to the similarity constraints. We process this query as described previously in this section with the following modifications. Instead of the collection of all expressions, we only consider the expressions resulting from the first step. For each constraint C_i, we consider the collection X_i of subexpressions bound to its associated wildcard. While processing R_Q, we form intermediate list Q_i^M (Subsection 5.3.1) only from expressions in X_i. While forming the intermediate lists, in order to look up subtrees bound to the similarity constraints by their tag words, we should also consider extending the index. More specifically, we extend I_M (Section 5.3.1) by adding all subtrees of each expression to the inverted index. The rest of this algorithm remains unchanged.

We obtain the final result by ranking documents that contain expressions resulting from the first step with respect to R_Q. The above steps are summarized in Algorithm 14, and the approach is evaluated experimentally in Section 5.5.

5.4 Rich Queries With Math Expressions and Keywords

Effective retrieval of documents with respect to rich queries that contain math expressions and also keywords requires holistic calculation of the similarity of each part. Some desired properties of the similarity score include the following: (*i*) the score of each part should be calculated differently as they have different formats and semantics; (*ii*) the score also should reflect full matches versus partial matches; and finally (*iii*) relevance scores of the

math part and the text part should be combined in a meaningful way to obtain the score of the whole document.

Assume a rich query consists of a collection of math expressions and a collection of terms. More specifically, a query Q consists of two sets: $Q = Q_M \cup Q_T$, where $Q_M = \{Q_1^M, \ldots, Q_n^M\}$ is a collection of math expressions, and $Q_T = \{Q_1^T, \ldots, Q_m^T\}$ is a collection of terms.

Following standard practice, we calculate the similarity of Q_T to d based on cosine similarity [6] as follows. For each term Q_i^T, $docRank(Q_i^T, d_T)$ is calculated based on term frequency (i.e. the number of times a term appears in a document), inverse document frequency (i.e. the number of documents that contain a term) and whether d_T contains Q_i^T or not. More specifically, given term t, document d, and a collection of documents D, we consider the following equations:
$tf(t,d) = \sqrt{frequency\ of\ t\ in\ d}$
$idf(t) = \log \frac{|D|}{|\{d \in D \mid t \in d\}|}$

$$docRank(Q_T, d_T) = \sum_{Q_i^T \in Q_T} tf(Q_i^T, d_T).idf(Q_i^T) \tag{5.4}$$

Finally, the similarity of a document to the query is calculated as follows:

$$docRank(Q, d) = \omega\ docRank(Q_T, d_T) + (1 - \omega) docRank(Q_M, d_M) \tag{5.5}$$

$docRank(Q_M, d_M)$ is calculated as described in Section 5.2. ω $(0 \leq \omega \leq 1)$ specifies how much the text part versus the math part of the query affects the similarity. For ease of explanation we assume both parts are equally important ($\omega = 0.5$) in the rest of this chapter. As a part of our future work, we propose to investigate the effect of different values of ω on the quality of results.

According to the above equation, the value of docRank for the text and the math expressions are linearly combined. Using the theory of language models, we can show that $docRank(Q_T, d_T)$ and $docRank(Q_M, d_M)$ are in fact estimations of the probabilities that the language models that generate the text and math parts of the query also generate the text and math parts of the document (details are out of the scope of this thesis). This justifies the combination of the partial scores according to Equation 5.5.

We can efficiently calculate the similarity score of math expressions in a similar way as described in Algorithm 13. The only modification that is necessary is to add a list of documents sorted by the value of $docRank(Q_T, d_T)$, and calculating final score of each document with respect to Equation 5.5.

5.5 Experiments

In this section we present the results of the empirical evaluation of the proposed algorithms. The purpose of this study is to compare the performance of the algorithms in terms of the usefulness of results and query processing speed.

5.5.1 Alternative Algorithms

We evaluate and compare the following algorithms for this study:

RS3: Our proposed algorithm (Rich Structural Similarity Search) for processing rich queries with math expressions and keywords (Algorithm 13).

RichTextSearch: Similar to TextSearch described in Section 3.5.1, we treat each math expression as a collection of nodes' labels in their XML trees. Together with other expressions and the text of a document (or query), they form a collection of terms. A standard keyword search algorithm is used to rank documents with respect to the query and select the top-k matches [3].

MEM: Exactly match math expressions. Documents that do not contain at least one exact match to each math expression in the query are filtered out. The remaining documents are ranked with respect to the query keywords.

RichMIaS: According to MIaS algorithm (described in Section 3.1), math expressions are transformed into collections of tokens. Such tokens together with the text of a document, form a collection of terms. A rich query is similarly transformed into a collection of terms. A standard text search engine is used to index both the text and math expressions, and to rank them according to a rich query.

5.5.2 Data And Query Collections

For this study, we use the collection of documents described in Section 4.5.1. However, for this experiment, we also preserve the text of each document.

To collect a set of representative queries, similar to Section 4.5.1, we use math forums. By manually reading the discussions in a thread we gathered a collection of queries. After

[3] We used Apache Lucene in our implementation.

Forum Query Collection	
Number of queries	20
Average number of math expressions per query	1.4
Average keyword per query	1.2
Average size of query math expressions	19

Table 5.1: Rich math queries statistics.

reading such discussions we often understand the intention of the user, which allows us to judge if a document is relevant to the query or not. For this experiment we form a query as a combination of math expressions and keywords. We only consider queries that contain at least two components, and one or more math expressions. Statistics about the query collection are presented in Table 5.1.

5.5.3 Evaluation Measures

We evaluate the discussed algorithms in terms of the correctness (relevance of results to the query), and query processing time. For the correctness of results, we consider NFR and MRR metrics described in Subsection 3.5.2. We consider the average query processing time (Subsection 4.5.1) to compare algorithms in terms of their efficiency.

5.5.4 Evaluation Methodology

To evaluate the algorithms in terms of the correctness of results, we run them on the collection of queries. Recall that a query corresponds to a discussion thread in a math forum, and the content of the thread allows us to understand the intention of the query. Hence, in each case we manually inspect the results and calculate the Reciprocal Rank (RR). We finally calculate the NFR, and the MRR for each algorithm.

To measure the query processing time, we do not consider the network delay, the time to convert query's math expressions into Presentation MathML, or to render the results.

5.5.5 Evaluation Results

The result of evaluating the correctness of the described algorithms is presented in Table 5.4. As the results suggest, RS3 returns some results for all queries (NFR = 100%).

Algorithm	NFR	MRR
RS3	100%	0.81
RichTextSearch	100%	0.14
RichMIaS	100%	0.64
MEM	18%	0.95

Table 5.2: Algorithms' performance for Rich Queries.

	NFR	MRR
PatternSim	76%	0.72
PatternSim with similarity constraint	89%	0.70

Table 5.3: Algorithms' performance for Rich Queries.

This algorithm ranks a relevant page fairly high in most cases (MRR = 0.81). RichTextSearch also returns some results for all queries, but it performs poorly in ranking a relevant document high (MRR = 0.14). Surprisingly, this algorithm performs worse than TextSearch (Section 3.5.1). This is because in this algorithm the tags of all math expressions are considered together with the rest of the document as a single collection of terms. Hence, matching expressions based on their tags and ignoring their structures results in an even worse performance for this algorithm. MEM does not return any result in many cases (NFR = 18%) but it ranks a correct result highly when it finds a match. RichMIaS performs better than the other baselines, but because it loses some structure information, it performs worse than RS3. In summary, RS3 outperforms the other algorithms in terms of accuracy.

We next consider the set of pattern queries extracted from math forums (Subsection 3.5.2), and modify some by adding similarity constraints to them based on the discussions in a thread. We compare the performance of PatternSearch with similarity constraints on the modified queries with that of the original pattern search with no similarity constraint. Note that we do not consider the queries that are not modified for this comparison. As the results suggest, similarity search increases the NFR, which implies that for more queries we return some results. This is because with similarity constraints expressions can be matched more flexibly. The MRR stays relatively high.

Finally, we consider the query processing times of the described algorithms. As the results suggest, RS3 and PatternSim have a somewhat higher query processing time than the other algorithms, but they perform significantly better in terms of the accuracy of results.

Algorithm	Average Query Processing Time (MS)
RS3	1077
RichTextSearch	573
RichMIaS	721
MEM	682
PatternSim	949

Table 5.4: Algorithms' performance for Rich Queries.

5.6 Conclusions and Further Work

We proposed algorithms to efficiently process queries that consist of several math expressions and, possibly, textual keywords. We also proposed optimization techniques and showed how such approaches can be used to process pattern queries with similarity constraints.

The proposed algorithms can be applied to other types of data. For example, pages that consist of text and sets of attribute-value pairs are popular on the web (e.g. a page about an electronic product that contains a review in plain text format, and specifications of the product in attribute-value format). Retrieving such documents with respect to both parts is a problem with useful applications that is relevant to the problem we discussed int this chapter.

To combine the scores of the text and the math parts of a document, we used Equation 5.5 and assumed ω is set to 0.5. Tuning this parameter to enhance the results should be further investigated.

Finally, further work on indexing and optimization algorithms is necessary in order to handle rich queries on larger data sets while keeping the query processing time low and even competitive to that of keyword search.

Algorithm 13 Selecting top-k results for rich math queries

Input: Query Q, document collection D.
for each $Q_i^M \in Q_M$ **do**
  Perform a keyword search using I_M
  $L_i^M \leftarrow$ a ranked list of documents with respect to the upper bound on $docRank_{UL}(d_M, Q_i^M)$
end for
allLists = $\{L_1^M, \ldots L_{|Q_M|}^M\}$
Set threshold ϕ = the score of an imaginary document with highest value in each list.
Set topK list to be empty
while true **do**
  $L \leftarrow$ next list in allLists such that $L.tokens > 0$
  if L is null (no list with positive number of tokens exists) **then**
    Update $\mathcal{G}(L_i)$s and assign tokens
  else
    $L.tokens \leftarrow L.tokens - 1$
    $d \leftarrow$ top document in L
    Remove d from all lists
    Calculate $docRank(Q, d)$
    if $docRank(Q, d) > min\{docRank(Q, d_i) | d_i \in topK\}$ **then**
      Insert d in topK.
      Remove document with lowest score if $|topK| > k$
    end if
    Update ϕ
  end if
  if all lists are empty, or $\phi < min\{docRank(Q, d_i) | d_i \in topK\}$ **then**
    Return topK
  end if
end while

Algorithm 14 Processing pattern queries with similarity constraints.

1: Input: Pattern query, Q.
2: Output: A list of documents that contain matching expressions to Q ranked with respect to the similarity constraints.
3: $Q' \leftarrow$ modify Q by removing all similarity constraints.
4: Run $optimizedPatternSearch(Q)$, and store information about bound expression wildcards.
5: $exprs \leftarrow$ expressions that match Q'
6: $R_Q \leftarrow$ a rich query that consists of the expressions in the similarity constraints of Q
7: Rank expressions in $exprs$ with respect to R_Q
8: **return** ranked list of documents containing expressions in $exprs$

Chapter 6

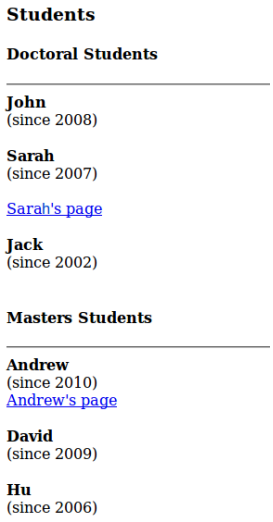
Grammar Inference

The techniques we have investigated so far rely on matching the structure and content of a query to the structure and content of expressions appearing in documents. However, some of the proposed algorithms can be improved by comparing the grammar describing the query against expressions appearing in the document or against their grammars. Furthermore, by considering grammars, we will see in the next chapter that we can also extend the query language. An approach based on recognizing grammars can also be used as the basis of an information extraction system.

A considerable amount of information is stored in documents encoded with a markup language. Extracting, manipulating, and retrieving such information requires a knowledge about the structure of the documents and the way data is arranged in them. Such knowledge is represented in the form of a grammar (e.g. XML Schema or DTDs). Similarly, web page wrappers describe the structure of a group of web pages. Although required by many applications, the grammatical information is unknown in many scenarios.

Recall that markup languages can be classified into descriptive and presentational. Descriptive markup encodes data records and is mostly used to exchange data between applications or to store it in a database, and presentational markup is mostly used to format documents in order to publish them. HTML or XHTML pages and objects within a page such as mathematical expressions encoded with (presentation) MathML [24] are examples of widely used presentational markup languages for publishing data on the Web. An example of presentational XML is shown in Figure 6.1, where students are listed in the body of an XHTML document.

Example 13. *Assume a set of university professors' personal web pages is collected, and we wish to extract information about supervised students. Each page is created by a different*




```
<h3>Students</h3>
<br/>
<h4>Doctoral Students</h4>
<hr/>
<b>John</b>
<div> (since 2008)</div>
<br/>
<b>Sarah</b>
<div> (since 2007)</div>
<a href="uwaterloo.ca/~sarah">
        Sarah's page
</a>
<b>Jack</b>
<div> (since 2002) </div>
<h4>Masters Students</h4>
<hr/>
<b>Andrew</b>
<div>(since 2010)</div>
<a href="waterloo.ca/~andy">
  Andrew's page
</a>
<br/>
<b>David</b>
<div>(since 2009)</div>
<br/>
   ...
```

```
<students>
  <studentlist level="doctoral">
    <student>
      <name>John</Name>
      <year>2008</year>
    </student>
    <student>
      <name>Sarah</Name>
      <year>2007</year>
      <homepage>
        uwaterloo.ca/~sarah
      </homepage>
    </student>
    <student>
      <name>Jack</Name>
      <year>2002</year>
    </student>
  </studentlist>
<studentlist level="masters">
    <student>
      <name>Andrew</Name>
      <year>2010</year>
      <homepage>
        waterloo.ca/~andy
      </homepage>
    </student>
     ...
```

Figure 6.1: Presentational XML (left) vs. descriptive XML (right)

professor and has a potentially unique structure in the collection. Also, many professors are not familiar with HTML and hence may use HTML tags only for their visual effect, resulting in complex structures (e.g. Figure 6.1) . Currently, because there are very few examples corresponding to each grammar, recognizing lists such as these cannot be done without intensive manual involvement. Similar examples exist for other types of presentational XML documents, such as mathematical expressions.

To date, schema induction has mostly been studied in the context of descriptive XML, aiming to infer appropriate DTDs or XSDs for a set of sample XML documents [41, 13, 12], and in the case of presentational XML to induce wrappers for a set of web pages [72, 76, 33, 123]. They typically require huge sample sets to infer a grammar correctly. That is, they assume a large set of instances generated from the same grammar is given, and the

problem is to infer the unknown grammar accordingly. Descriptive markup is often used to encode massive amounts of information to be stored in a database. Similarly, many large websites use a single template to publish data in web pages. Hence, providing a learning algorithm with large sample sets is feasible in such cases. However, in many scenarios only a few, possibly only one, samples are available, hence these approaches fail to infer a grammar correctly.

A great amount of data is published with presentational markup, and many existing and emerging applications need to analyze such data before they can provide extra services:

- A math retrieval system benefits from identifying the grammar of expressions to enhance the retrieval accuracy or to optimize it for specific queries (see Chapter 7).
- Commercial search engines and question answering systems (e.g. Google and Bing's Calculators), need to identify specific queries (query classifiers) and recognize their structures to further process them [60, 101].
- Rendering engines restructure web pages that are initially designed to be shown on traditional devices such as desktop computers, in order to optimize them for increasingly popular devices such as smart-phones and tablets [27].
- Web tables are widely used to enhance web search and are sources of other valuable information [22] (e.g. Google Fusion Tables [44]). Currently, only tables that are created using HTML table tags can be identified [22].

The above applications are often provided with only one sample. Assuming only one sample is given, the problem is to infer its grammar. In general, grammars overlap and a given sample may be generated by many grammars. Based on our observation of the existing Web objects, we define a class of grammars that minimizes such overlaps and allows inferring a grammar that is likely to have been used to generate a given sample (Web object) correctly.

Despite the previous work, we do not assume all samples are generated by the same grammar. Instead, we assume that most grammars that generate real-world samples satisfy some simple constraints. We show that limiting ourselves to suitably constrained grammars reduces false inferences (i.e. grammars that generate a sample but differ from the unknown grammar that was actually used to generate it). We formally define a suitable set of constraints in this chapter and propose algorithms to infer grammars that satisfy them. In order to infer correct grammars, the defined constraints should realistically reflect the characterization of the samples. This is especially a challenging problem if the sample documents use presentational markup.

Example 14. *Consider the presentational XML document in Figure 6.1. There is no explicit delimiter that separates the students and their attributes from each other and from the rest of the document. Moreover, some fields appear often and in various locations within a listing (e.g.* `<br/>` *delimits disparate entities), and the same tags might represent different entities. For example in the same figure* `<div>` *tags are used for both year and biography. Figure 6.1 also illustrates an alternative representative XML for the student listings. With descriptive markup, tags represent the semantics of their contents. Therefore, the same tags represent similar data and the structure is cleaner. For example, the* `<name>` *tag appears exactly once for each student because each student has exactly one name.*

Despite its wide applications, there has not been a principled approach to the described problem. Data extraction systems define some constraints on the grammars that allow a grammar to be identified for a given instance. However, such constraints are too restrictive and result in inferring incorrect grammars in many cases. For example some approaches consider only single occurrence grammars [75, 123, 2], some do not allow optional and disjunctive expressions within repetitive patterns [74, 106, 116], and some do not allow nested repeating patterns [25, 74, 116]. These restrictions are described further in Section 6.3.

An operator that greatly influences the cardinality and expressive power of a grammar is the repetition operator (e.g. A^*). In practice, such repetitions often provide important information about the data (e.g. list of data records in web pages). A repetition consists of a grammar (nested within the unknown grammar) that repeats. Hence, each instance of the unknown grammar contains several instances of the nested one. Therefore, even if one sample is provided, it potentially contains several samples from the nested grammars. For example, Figure 6.1 displays a single page with many instances of a student record. If we are able to identify such (smaller) samples, we can potentially infer the nested grammars, and through them, the unknown overall grammar. We achieve this by defining constraints on the repetition operator that allows identifying nested instances correctly.

Our goal is to identify an unknown grammar that contains repetition when only one instance is given. We define a class of grammars, more expressive than DTDs and XSDs, that captures repetitions without unnecessary over-generalization of the language. We propose an algorithm that finds instances of repeating patterns within a given instance based on multiple sequence alignment to infer a grammar. Because the algorithm is particularly suited to recognize the structure in individual pages using presentation markup, we name it the Presentational Markup Grammar Inference algorithm, or PMGI. We argue that the chance that an appropriate grammar is inferred is the highest among the grammars that generate the sample. Through empirical studies, we demonstrate the effectiveness of our algorithm on documents with presentational markup.

Because PMGI works well when only one sample is provided, it is suitable for applications dealing with documents with presentational markup, although it can also be applied to the easier problem of inferring grammars for descriptive XML documents. If more than one sample is provided, our algorithm can be used to infer a grammar for each one. Combining the resulting grammars to obtain a more general grammar is a straightforward application of the same algorithm.

In summary, the contributions of this chapter are as follows:

- We define a class of regular tree grammars, called *k-normalized regular tree grammars* (k-NRTGs), that can model typical presentational XML found on the Web. The set of languages generated by k-NRTGs is a superset of the languages of DTDs and XSDs. In other words, our grammar is more expressive than DTDs and XSDs in capturing the structure of presentational XML documents.
- We propose an inference algorithm and prove that, even if only a few samples are provided, the correct grammar is inferred with high probability.
- We categorize and compare the class of grammars identified by the existing algorithms against k-NRTGs in terms of how well they can describe XML documents on the Web.

We finally show that while the existing approaches fail to recognize a grammar that correctly describes a presentation XML document when a small sample size is given, our algorithm correctly recognizes the grammar with high probability [61].

6.1 Definitions

6.1.1 Tree Grammars

A regular tree grammar describes a "language" of parse trees in terms of regular expressions, just as a conventional grammar uses regular expressions on the right sides of its productions to describe a language of strings.

Definition 4. *A regular tree grammar is a 4-tuple $H = (N, T, S, P)$ where N is a finite set of non-terminals, T is a finite set of terminals, S is a set of start symbols ($S \subseteq N$), and P is a set of production rules of the form $X \to a$ or $X \to a[E]$ where X is a non-terminal, a is a terminal, and E is a regular expression over N [87].*

Terminals correspond to labels for a tree's nodes, and they cannot be null. Each non-terminal X, with production rule $X \to a[E]$, generates a set of trees, where a represents the label of each root, and E is their content model. Different non-terminals may not generate the same sets of trees. We allow the following operators within a regular expression: *concatenation*, *disjunction*, and *bound repetition*. Concatenation of E_1 and E_2 is represented as E_1E_2 and their disjunction is represented as $E_1|E_2$. *Bound repetition* is represented as $E^{[l,u]}$, where E is a regular expression that is constrained to repeat at least l times and at most u times, and $0 < l \leq u$. For example $E^{[1,3]}$ means E should occur at least once and at most three times. The bound $[1, \infty]$ indicates that the regular expression E can repeat arbitrarily often. For convenience, we define $E^k = E^{[k,k]}$, $E? = E|\epsilon$ (i.e., optional, where ϵ denotes the empty tree), $E^+ = E^{[1,\infty]}$, and $E^* = E^+|\epsilon$. The language of a regular expression E is the set of all strings generated by E and is denoted by $L(E)$. Two regular expressions are equivalent if they generate the same language. An *atomic* regular expression is a non-terminal or an expression of the form E^+ or $(E_1|E_2|...|E_k)$ where $k > 1$ and each E_i is an atomic regular expression (i.e. a limited concatenation of expressions is non-atomic). A regular expression E is in disjunctive form if $E = (E_1|E_2|...|E_k)$, $k > 1$, and $L(E_i) \neq L(E_j)$ if $i \neq j$. (Note that by definition $E?$ and E^* are disjunctive regular expressions when $E \neq \epsilon$.)

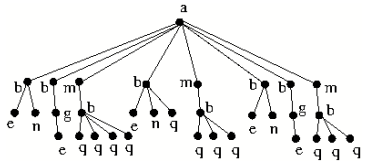


Figure 6.2: A labeled ordered tree.

As an example consider the tree in Figure 6.2, which can be generated by the following regular tree grammar:

$$
\begin{array}{lll}
H = \{N, T, S, P\}, & P = & \{E \to e, N \to n, Q \to q, \\
T = \{a, b, e, g, m, n, q\} & & G \to g[E], J \to b[G], \\
N = \{E, N, Q, G, I, J, K, L, M\} & & I \to b[ENQ?], K \to b[Q^+], \\
S = \{L\} & & M \to m[K], L \to a[(IJ?M)^+]\}
\end{array}
$$

Note that I, J, and K share a common root symbol. This grammar cannot be described by a DTD or XSD.

6.1.2 k-Normalized Regular Expressions

In the rest of this section we define a class of regular tree grammars that can be efficiently inferred from small sets of samples. Whereas a DTD describes *valid tagged text* in a document and XML Schema describes *valid elements and attributes* in a corresponding Infoset model, a regular tree grammar (the basis of RELAX NG) describes *valid parse trees* [32, 87].

Our k-normalized regular tree grammars relax some restrictions that DTDs and XSDs impose and hence can better model documents with presentational markup. The way such grammars are defined allows learning them from small sample sets.

We wish to assign weights to subtrees to capture differences in their significance in forming a tree. Hence we compare subtrees by size (i.e., the number of nodes they contain). We assign weights to each non-terminal N in the grammar to represent the maximum size of a tree that can be generated by N or a constant W if this size is unbounded or it exceeds W. We augment this by assigning weights to regular expressions as follows:

- if E is a nonterminal N where $N \to n[E']$,
  $\omega(E) = min\{W, 1 + \omega(E')\}$
- $\omega(E_1E_2...E_k) = min\{W, \sum \omega(E_i)\}$
- $\omega(E^{[l,u]}) = min\{W, u.\omega(E)\}$
- $\omega(E_1|E_2|...|E_k) = max\{\omega(E_i)\}$

Alternative monotonic weighting schemes could be adopted instead, provided they assign the same (positive) weights to equivalent expressions.

Definition 5. *For an atomic regular expression A, let $\kappa(A) = 1$ if either A is disjunctive or $\epsilon \in L(A)$ and $\kappa(A) = 0$ otherwise, and let α be a constant such that $0 \leq \alpha < 1$. A regular expression of the form $R = A_1A_2 \ldots A_n$, where A_is are atomic regular expressions, is* repeatable *if $V(R) = \frac{\sum \kappa(A_i)\omega(A_i)}{\sum \omega(A_i)} \leq \alpha$. E is a* repetitive pattern *if $E = (R)^{[l,u]}$ and $u > 1$. E is a valid repetitive pattern if R is valid and repeatable, where a regular expression is* valid *if it contains only valid repetitive patterns or no repetitive pattern at all.*

For example if $E = (A?B(C|D))^+$, $\alpha = 0.5$, and $\omega(A) = \omega(B) = \omega(C) = \omega(D) = 5$, then $V(E) = \frac{\omega(A?)+\omega(C|D)}{\omega(A?)+\omega(B)+\omega(C|D)} = \frac{10}{15} > \alpha$, so E is not a valid repetitive pattern. This reflects the fact that the production includes "too few" terms that must appear in every

repetition. On the other hand, if A and $(C|D)$ are "small," say $\omega(A) = \omega(C) = \omega(D) = 1$, then $V(E) = \frac{2}{7}$ which is less than α, and therefore E would be a valid repetitive pattern: the "large" B appears in every repetition. Without such a restriction, any sample sequence can be generated by a repetitive pattern. Assume the sequence of the children of a node in the instance tree is $c_1c_2...c_n$, and non-terminal C_i generates c_i, then this sequence can be generated by $(C_1|C_2|...|C_n)^+$, which is a repetitive pattern but not a valid repetitive pattern if $\alpha < 1$. If $\alpha = 0$ then no disjunctive or optional components are allowed in a repetitive pattern and if $\alpha = 1$ then any regular expression is repeatable.

Definition 6. *Let $U = U_1U_2...U_m$ and $V = V_1V_2...V_n$ be two regular expressions in which every U_i and V_i is atomic. The* alignment *of U and V is a maximal ordered sequence $< (U_{i_1}, V_{j_1}), (U_{i_2}, V_{j_2}), ..., (U_{i_k}, V_{j_k}) >$ of pairs such that U_{i_x} and V_{j_x} are equivalent and for every two pairs (U_{i_x}, V_{j_x}) and (U_{i_y}, V_{j_y}), $i_x < i_y$ iff $j_x < j_y$. Define $\widetilde{U_{i_x}} = U_{i_x+1}...U_{i_{x+1}-1}$ (or ϵ if $i_{x+1} = i_x + 1$), and $\widetilde{U_{i_0}} = U_1...U_{i_1-1}$, and define $\widetilde{(V_{j_x})}$ similarly. Finally define $A_x = U_{i_x}$ and $A'_x = (\widetilde{(U_{i_x})}|\widetilde{(V_{j_x})})$. The* merge operator $\bigoplus$ *is defined to be the combination function that takes two regular expressions and returns a "covering" regular expression as follows: $U \bigoplus V = A'_0A_1A'_1...A_kA'_k$.*

For example $AB \bigoplus BC = A?BC?$ and $(ABCE) \bigoplus (B(C|D)E) = (A?B(C|D)E)$. In Section 6.2.2 we will describe various sequence alignment algorithms. Sequence alignment is a well-studied problem with applications in various fields such as bio-informatics and natural language processing [85].

Definition 7. *Consider a regular expression of the form $E = U^{[l_1,u_1]}CV^{[l_2,u_2]}$ where $(U \bigoplus V)C?$ is repeatable. The* compression function Γ *returns a regular expression as follows: $\Gamma(E) = ((U \bigoplus V)C?)^{[l_1+l_2,u_1+u_2]}$. A regular expression is* compressed *if no compression can be applied to any of its subexpressions, and in that case we define $\Gamma(E) = E$ for notational convenience. Note that $L(E) \subseteq L(\Gamma(E))$.*

For example, assume that $\omega(A) = \omega(B) = 3$, $\omega(C) = 1$, and $\alpha = 0.5$. According to the above definition, $\Gamma(ABAB) = (AB)^2$, $\Gamma(ABC(AB)^3) = (ABC?)^4$, $\Gamma((ABC)^{[1,3]}(AB)^2) = (ABC?)^{[3,5]}$, and $\Gamma((AB)^+C(AB)^2) = ((ABC?)^{[3,\infty]}$.
$ABABBCBC$ and $(AB)^2BCBC$ are not compressed, but $(AB)^2(BC)^2$ is compressed. If $\omega(A) = \omega(C) = 1$ and $\omega(B) = 3$, then $(AB)^2(BC)^2$ is not compressed because $AB \bigoplus BC$ is repeatable and $(A?BC?)^4$ is valid and compressed. According to the choice of subexpressions, more than one compressed form might exist for a regular expression, e.g. AB^2C and $(A?BC?)^2$ are both compressed forms of $ABBC$ (if $(A?BC?)$ is repeatable).

Definition 8. *For a constant k, a* k-normalized regular expression *(k-*NRE*) is a compressed regular expression where all bound repetition operators are of the form $[1, \infty]$ or $[m, n]$ where $m \leq n \leq k$. Regular tree grammars with k-normalized regular expressions are called* k-normalized regular tree grammars *(k-*NRTG*).*

For example $(AB)^3(CD)^2$ and $(AB)^{[1,3]}(CD)^2$ are compressed regular expressions, but they are not k-NREs for $k < 3$, $(AB)^+(CD)^2$ is a 2-NRE, and $(AB)^+(CD)^+$ is a 1-NRE. To ensure k-normality, we define the compression function, Γ_k, identically to Γ but replacing $[l, u]$ in the resulting expression by $[1, \infty]$ if $u > k$. Our experiments show that 2-NRTGs can capture the structure of most presentational XML published on the Web.

We now briefly explain the concept of language identification in the limit introduced by Gold [42].

Definition 9. *A* language learner *is an algorithm that infers the name of an unknown language. The learner is provided with a training sequence $(i_1, i_2, \dots)$ of information units describing the language. A training sequence can be in the form of text (i.e., each i_t is a string from the language) or an informant (i.e., each i_t states whether or not a specific string is in the language). We associate a set of* allowable training sequences *with each language. Given a specific training sequence, at time t the learner is provided with i_t, and it guesses the name of the unknown language. The language is* identifiable in the limit *if after some finite time the guesses are all the same and are correct. A class of languages is* identifiable in the limit *if there is an effective learner (i.e. an algorithm) that can identify in the limit any language in the class given any allowable training sequence for that language. A language is* finitely identifiable *if there is a finite time t when the learner can tell that it is making correct guesses. (Note that unlike finite identifiability, identifiability in the limit does not require the learner to know when its guesses are correct.)*

According to this definition, a text consists only of positive examples while an informant can also contain negative examples. For instance, consider these regular expressions: $R_1 = L((abc)^+)$ and $R_2 = L((abc?)^+)$. A text training set for the first regular expression can only contain strings such as *abc* or *abcabc*, whereas an informant training set can contain information such as "*abcabc* is in R_1" and "*ababc* is not in R_1". Since $R_1 \subset R_2$, if a learner is given only positive samples from R_1, it cannot distinguish R_1 from R_2. Gold proves that the class of regular expression languages are not identifiable in the limit from positive examples. However, if the probability of any string in $L(E)$ appearing as a text is at least $\delta > 0$, the more samples from R_1 are observed with no samples from $R_2 - R_1$, the higher the chance that R_1 is the correct guess. Also, languages with limited cardinality are identifiable in the limit from positive examples. A *characteristic set* of strings for L is a

subset C of L such that for any training sample set S if $C \subseteq S$ then the learner identifies L in the limit.

The problem of inferring k-NRTGs from one instance is equivalent to the problem of identifying an unknown language in the limit, but with the extra requirement that the training set is finitely bounded and the bound is small. If the sample set does not contain a characteristic set of strings for the unknown language, then it is impossible to identify it, but we propose an approach that infers it with high probability.

6.2 The Inference Algorithm

In this section we explain an inference algorithm when only one sample is provided. We start by describing an algorithm for inferring a k-normalized regular expression (k-NRE) that generates a given instance. Then we describe a clustering algorithm for subtrees in order to transform a sequence of subtrees into a sequence of cluster ids. We next infer a k-NRE that generates this sequence. Inferring a k-NRTG consists of inferring k-NREs for the sequence of the subtrees of each node in a depth-first manner and combining the k-NREs of nodes whose corresponding subtrees belong to the same cluster.

6.2.1 Inferring k-NREs

An instance of a regular expression E is an arbitrary string taken from $L(E)$. We represent non-terminals with capital letters and their instances with lower case letters.

Theorem 4. *Consider a valid repetitive regular expression $R = E^+$ where $E = E_1 E_2 \dots E_k$ and each E_i is an atomic regular expression that does not itself contain a repetitive pattern. Assuming E repeats at least twice, an instance of R, r, can be partitioned into instances of E. Moreover, E is identifiable in the limit, and if the set of its instances contains a characteristic set of E, R and E can be identified.*

Proof. Any instance of R is a sequence of instances of E. Because none of the E_is contain a repetitive pattern, $L(E)$ has a limited cardinality (i.e. $|L(E)|$ is bounded), and so it can be finitely identified from positive examples. We show that the inference of E implies the inference of R.

Each E_i is an atomic regular expression that does not contain a repetitive regular expression. Thus, it is either a non-terminal, or it is disjunctive. Let $W_d = \sum \kappa(E_i)\omega(E_i)$ (the sum of the weights of the disjunctive components) and $W_n = \sum(1-\kappa(E_i))\omega(E_i)$ (the sum of the weights of the non-terminals). R is a valid repetitive regular expression, so E is repeatable. Therefore $V(E) = \frac{W_d}{W_d+W_n} \leq \alpha$, so there are no disjunctive terms if $\alpha = 0$ and $W_d\frac{1-\alpha}{\alpha} \leq W_n$ if $0 < \alpha < 1$, which implies that $W_n > 0$. Thus some non-terminals must appear in every instance of E. Assume M is the set of such *mandatory* non-terminals, and let D be the set of all non-terminals in R. $M \subseteq D$, so $0 < |M| \leq |D|$. Assume E_m is the leftmost E_i that belongs to M (i.e. m is the smallest i such that $E_i \in M$). E_m is not in disjunctive form and it is atomic, therefore it consists of a single non-terminal. Assume E' is a permutations of the elements of E:
$E' = E_m, E_{m+1}, \ldots, E_k, E_1, \ldots, E_{m-1}$.
Because E is repeatable, E' is also repeatable. We first partition the given sample into instances of E'.

Recall that r is an instance of $R = E^+$. Let $r = r_1 \ldots r_{|r|}$ where each r_i is a symbol that represents an instance of a non-terminal. Assume $r_\tau = s$ is an instance of E_m (the leftmost E_i that belongs to M). Hence, s must be the first symbol in every instance of E'. For symbol s, we define an s-based subsequence to be a subsequence $r_i \ldots r_j$ of r such that:

$$
\begin{aligned}
&r_i = s \\
&1 \leq j - i + 1 \leq \tfrac{|r|}{2-\alpha} \\
&j = |r| \ or \ r_{j+1} = s
\end{aligned}
$$

The number of instances of E in r is at least 2 and at most $|r|$, and it is repeatable, therefore as calculated in the above equation, $1 \leq |E| \leq \frac{|r|}{2-\alpha}$.

Let $Q = \{q_1 \ldots q_{|Q|}\}$ be a set of s-based subsequences that cover $r_\tau \ldots r_{|r|} r_1 \ldots r_{\tau-1}$. Assume there is no overlap between them. We perform a multiple sequence alignment to find the largest common subsequence of the strings in Q. Assume q_{max} is the longest q_i, and $com(Q)$ is the length of the longest common subsequence of q_is. If $\frac{c_{max}-com(Q)}{com(Q)} \leq \alpha$, then Q is a possible partition of r according to E'. A partition of r according to E can be acquired by permuting the subsequences in Q. We repeat the above algorithm for $s = r_\tau$ with $\tau = 1$ to $\frac{\alpha|r|}{2-\alpha}$, and find all possible partitions according to the above algorithm. Each partition results in a possible grammar for E. Intuitively, if there are more samples from a language, comparing to fewer samples from another language, the first sample set is more indicative, assuming the cardinality of both languages are similar. Therefore, if there is more than one partition possible, we choose the one with the maximum repetitions (Q with maximum

size, $|Q|$). If there is more than one such partition with maximum repetitions, we choose the one with the smallest number of unaligned (i.e. optional) symbols. A summary of the above steps is presented in Algorithm 15.

After such *anchoring* non-terminals are recognized, the subexpressions that are in disjunctive form can be determined, and hence E and R are recognized. We describe details in the next section. □

Algorithm 15 Partition

```
{Goal: partition an instance of R = E+ into instances of E}
Input: r, an instance of R.
Output: A partition of r into instances of E.
res = Φ
for i = 1 to |r|/(2−α) do
   s = r_i
   Find all s-based* subsequences of r.
   for all Q = a set of s-based non-overlapping subsequences that covers
   r_i ... r_|r| r_1 ... r_{i−1} do
      Perform multiple sequence alignment on Q.
      if (c_max − com(Q))/com(Q) > α then
         Discard Q
         continue
      end if
      if |Q| > |res| then
         res = Q
      else if |Q| = |res| and |com(Q) > |com(res)| then
         res = Q
      else if |Q| = |res| and |com(Q) = |com(res)| then
         Report both, or disambiguate
      end if
   end for
end for
return res
* An s-based subsequence of r is r_i ... r_j such that r_i = s, 1 ≤ j − i + 1 ≤ |r|/(2−α), and j = |r| or r_{j+1} = s.
```

For a given regular expression E with limited cardinality and sequence S of strings in $L(E)$, let $P(E|S)$ be the probability that E is the correct regular expression given S and

$P(S|E)$ be the probability of generating S given E. Assuming a uniform distribution over strings in $L(E)$ (i.e. all strings in $L(E)$ are generated with equal probability), $P(S|E) = \prod_{s \in S} P(s|E) = \prod^{|S|} \frac{1}{|L(E)|} = \frac{1}{|L(E)|^{|S|}}$. By Bayes' theorem: $P(E|S) = P(S|E)\frac{P(E)}{P(S)} = \frac{P(E)}{P(S)|L(E)|^{|S|}}$. Therefore the k-NRE with the smallest language that can generate S has the highest probability to be the unknown grammar. Hence, because R should be compressed, if $r = s_1 s_2 ... s_n$ and if E_{s_i} is the regular expression chosen to describe s_i, $E_{s_1} \bigoplus E_{s_2} \cdots \bigoplus E_{s_n}$ has the highest probability of being the correct k-NRE for E. If $n > k$, then we choose $R = E^+$; otherwise, $R = E^n$ is more likely to be the correct expression for generating this instance.

In some cases where E contains boundary disjunctive forms (i.e. E_1 or E_k are disjunctive), ambiguities in recognizing instances of E might occur. For example, $r = abababa$ can be partitioned as ab, ab, ab, a, which results in inferring $E = (AB?)^+$, or as a, ba, ba, ba, which results in $E = (B?A)^+$. In these situations, our algorithm generates all possible partitions, and heuristics can subsequently be applied to resolve such ambiguities.

6.2.2 Sequence Alignment

Multiple sequence alignment is the problem of aligning a collection of sequences of symbols. Efficient algorithms that use dynamic programming to align two sequences exist [89], but the general problem of finding the optimal alignment for n sequences is NP-complete. However, because of its wide range of applications, many approximation algorithm are proposed [85]. Progressive multiple sequence alignment is a technique that incrementally builds up an alignment by combining pairwise alignments starting from the two most similar sequences. An alignment is built in two stages: first a guide tree is formed that stores pairwise similarity information of the sequences, and then sequences are aligned incrementally according to the guide tree [85]. This method does not guarantee to find the optimum alignment, but it performs well in most biological applications where hundreds to thousands of sequences are required to be aligned and their lengths is in the range of thousands symbols.

According to our experiments, the number of sequences and also their length rarely exceed 50, and therefore the alignment problem is much simpler in our case. Moreover, the result of progressive methods is close to the optimal when sequences are similar. We are interested only in the alignment of instances of a repetitive pattern, and by definition, such instances should be similar enough if the pattern is valid. Hence, the error from using these methods are negligible in practice. There are various implementations of progressive

sequence alignment, and we use Clustal [28], which is commonly used in the bioinformatics community.

6.2.3 Inferring a Repetition-Free Grammar

After the partitioning step, we must infer the grammar of the repetitive pattern. So the problem is to infer a grammar with limited cardinality (because it does not contain a repeating pattern), where the number of instances is the size of the partition. Such grammar should look like $D_0M_1D_1 \dots M_nD'_n$ where each D_i is an expression in disjunctive form (D_is can be empty strings), and each M_i is a mandatory non-terminal. The cost-based sequence alignment allows us to arrange the instances as a grid, where each cell contains a symbol or is empty. Each sequence corresponds to a row, and non-empty symbols in the a given column are the same. Columns that correspond to mandatory non-terminals do not contain an empty symbol in any row:

a	-	-	d	e	-
a	b	c	d	-	f
a	b	c	d	e	-

Assume $A_1 \dots A_k$ are the sequence of symbols whose corresponding columns are between M_i and M_{i+1}. $A_1? \dots A_k?$ obviously generates the sample set, but it might be an over-generalization. For example, in the above alignment $(BC)?$ and $E|F$ also generate the sample set and they are subsets of $B?C?$ and $E?F?$. We argued earlier that the most precise grammar according to a given sample set has the highest chance to be the correct grammar. Intuitively, a regular expression is precise if it does not cover too many expressions that are not among the samples. To find a precise grammar, we form an automaton with a state representing each column, using standard approaches to form an automaton according to the sample set and to find the regular expression with the smallest language that generates the sample set. Because the language is repetition-free, and the sequences are already aligned, this approach results a concise regular expression and avoids the generation of lengthy regular expressions. In the above example, our approach results in $A(BC)?D(E|F)$, and the corresponding repetitive pattern is $(A(BC)?D(E|F))^+$.

6.2.4 Identifying Instances of Repetitive Patterns

Next, we wish to detect an instance of a repetitive pattern within a larger sample string, e.g. instance *abab* of $(AB)^*$ in *cdababef*. For this purpose, we try the above algorithm

on all consecutive subsequences of the sample. For optimization purposes, we preprocess each subsequence for early detection of the ones that are not repetitive. We do not further process a subsequence U if:

$$\sum_{s_i occurs\ once\ in\ U} \omega(s_i) > \alpha \sum_{distinct\ s_i} \omega(s_i).$$

That is, if the total weight of symbols that appear once only in a subsequenceis is greater than α times the total weight of its distinct symbols, the subsequence cannot be repetitive by definition. In practice, this pre-processing phase greatly reduces the number of times we run our partitioning algorithm.

Algorithm 16 Identify

1: {**Goal:** Find repetitive patterns within a given sample sequence, U}
2: **Input:** A sequence U.
3: Let T be an empty set
4: **for** $i = 1$ to $|U|$ **do**
5: **for** $j = i + 1$ to $|U|$ **do**
6: **if** $\frac{\sum_{s_i\ occurs\ once\ in\ [U_i \ldots U_j]} \omega(s_i)}{\sum_{distinc\ s_i\ in\ [U_i \ldots U-j]} \omega(s_i)} > \alpha$ **then**
7: **Continue**
8: **end if**
9: Run Partition on $[U_i \ldots U_j]$
10: Add the partition information to T
11: **end for**
12: **end for**
13: Disambiguate pairs of overlapping partitions.
14: **return** T

Using Algorithm 16 we can find instances of repetitive patterns within a sample up to the ambiguity caused by boundary disjunctive forms. If two partitions P_1 and P_2 overlap, similar to the disambiguation algorithm presented earlier, we choose the one with the maximum repetitions, and if there is more than one such partition with maximum repetitions, we choose the one with the smallest number of symbols in disjunctive form. Further ties are broken by choosing the partition whose corresponding instance is shorter. If there is still a tie, we report both possibilities. For example consider $E_1 = (ABC)^+(C?DE)^+$ and $E_2 = (ABC?)^+(CDE)^+$. If the repetitions of the first pattern in a given sample is larger (e.g. $S = ABCABCABCABCDECDE$), the probability that E_1 generates it is higher: $P(E_1|S) > P(E_2|S)$. However, for some samples our disambiguation algorithm still results in a tie and these probabilities are equal so we report both patterns, e.g.

$S = ABCABCDECDE$. Many such ambiguities in finding an instance of R are caused by boundary disjunctive forms and can be resolved based on probabilities of generating strings. For example, assume a sample string is $s = abababc$. Two possibilities for E are $X_1 = (ABC?)$ and $X_2 = (AB)$. In the former case *abababc* is an occurrence of R and in the latter case *ababab* is an occurrence of R. Assume that R includes n repetitions of E (n is found as a consequence of finding the set of mandatory non-terminals). If the probability that c appears in an instance of X_1 is 0.5, the probability that it appears only in the last instance given n instances is $(0.5)^n$. Hence for that example $P((ab)^n c|X_1, n) = 0.5^n$ and $P((ab)^n|X_2, n) = 1$. Assuming no *a priori* bias (i.e., $P(X_1) = P(X_2)$), with a high probability X_2 represents E. Note, however, that if $\omega(C)$ is large enough, then there is no ambiguity because $(ABC?)$ is not repeatable and therefore is not an admissible solution. Alternatively, if $\omega(C)$ is small and c appears elsewhere within S, e.g. $s = abcababc$, then again there is no ambiguity: this time $X_1 = (ABC?)^+$ must be correct. Following a similar approach we can resolve most ambiguities caused by boundary disjunctive items.

Given an input string S, if $\mathcal{R}$ is the set of so far identified repetitive patterns in S, for each $R_i \in \mathcal{R}$ we create a new temporary non-terminal N_i and replace the instance of R_i in S with n_i and repeat the above steps. The new non-terminals should be chosen in a way that allows merging compatible patterns if necessary. When no more repetitive patterns are found, we replace the newly introduced non-terminals with their corresponding regular expressions to generate the final result. This allows the inference of expressions with nested repetitive patterns such as $((AB)^+(CD?E)^+)^+$.

The above steps explain how a k-NRE can be inferred from a single input string. If we are given multiple input strings, we infer a regular expression for each and combine the results using $\bigoplus$.

Theorem 5. *For every given sample, there is at least one k-NRE that generates it.*

Proof. The above algorithm obviously generates at least one k-NRE for a given sample set. The number of such grammars can be more if there are ambiguities that cannot be resolved. □

6.2.5 Inferring *k*-NRTGs

Recall that a document with presentational markup can be represented by a tree. Given an instance tree, each subtree is generated by a non-terminal in an unknown k-NRTG, and the sequence of children for its root is generated by the k-NRE for its content model. Therefore, to infer a grammar for a sample tree, we should infer a k-NRE for the sequence of the

children of each node. Our algorithm traverses the tree in post-order so that productions for the children of a node are generated before the root node is considered.

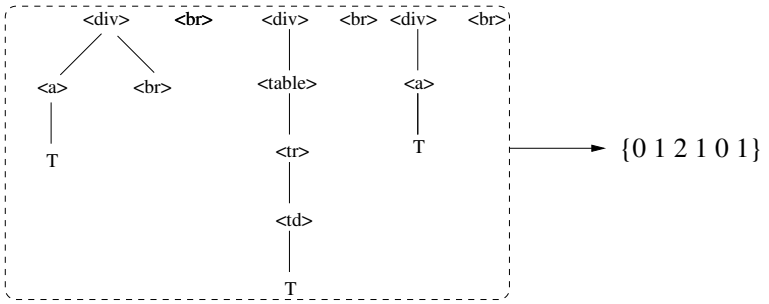


Figure 6.3: The sequence of subtrees is mapped to a sequence of cluster ids.

Consider an instance tree T, a node N in T, and the sequence S of the subtrees rooted at the children of N, $S = c_1 \dots c_k$. Because of the post-order processing, k-NRTGs for each c_i have already been recognized. We use *hierarchical agglomerative clustering* [56], which is based on the similarity between trees, to partition the c_is into clusters such that each cluster contains similar subtrees that are assumed to be generated by a single production in the underlying tree regular grammar. Thus all subtrees in a cluster must have the same root label, however subtrees with the same root labels are not necessarily in the same cluster. There are various ways to calculate tree similarity, such as comparing the inferred k-NRTGs for the c_is using variations of *tree edit distance* [17]. Although tree edit distance usually provides a good measure of similarity, its calculation is costly (its complexity is $O(mn)$ where m and n are the sizes of the two trees). Alternatively, similarity can be defined based on heuristics such as comparing the label and degree of the roots, the sizes, and the depths of trees. Thus an appropriate similarity function should be chosen based on the application requirements.

We assign a unique id to each cluster, which allows us to map $c_1 \dots c_k$ to a sequence I of cluster ids. We combine the k-NREs for all trees in a cluster using an extension of the $\bigoplus$ operator, and replace their content models with the result. We also calculate their weights as described in Section 6.1. Then, we use the algorithm from Section 6.2.1, to infer a regular expression for I. Lastly, we create a new non-terminal, with the label of N as its root label and the inferred k-NRE as its content model. A summary of the above steps is presented in Algorithm 17.

An an example, consider the page in Figure 6.1. A tree representing the presentational markup (XHMTL) for this page is shown in Figure 6.4, and the tree of its inferred grammar is shown in Figure 6.5.

Algorithm 17 PMGI

1: {**Goal:** Infer a K-NRTG for a given sample tree.}
2: **Input:** A sample tree T.
3: **for** each child c in $T.root.children$ **do**
4: PMGI(c)
5: **end for**
6: Cluster $T.root.children$
7: Transform the sequence of the children of $T.root$ into a sequence U of cluster-ids
8: res = U
9: Identify(U)
10: Replace each instance of a repeating pattern in res with its inferred grammar
11: Replace each symbol in res with its corresponding non-terminal.
12: **return** res

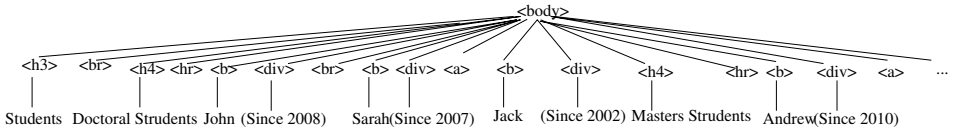


Figure 6.4: The tree representing the XHTML page of Figure 6.1.

6.3 Related Work

Descriptive markup is typically used to encode data objects to be stored in a database. Any subset of the documents in the database can be used as a sample set for an inference algorithm. As a result, a rather large number of sample documents is available to a learning algorithm to infer a grammar for descriptive XMLs. In fact, most grammar induction algorithms for descriptive markup extensively rely on large sample sets [12, 13], and their results for small sample sets is quite poor. XTRACT [41] is a system for inferring a DTD that is balanced in terms of conciseness (i.e. being short in length) and preciseness (i.e. not covering too many expressions that are not among the samples) for a given database of XML documents with descriptive markup. The heuristics that generate regular expressions are rather simple, and they do not generate complex expressions such as the ones that contain optional expressions nested within Kleene stars. Finally, experiments on real data show that XTRACT generates large, long-winded and difficult to understand regular expressions [13]. Bex et al. [13] propose an algorithm for inferring concise DTDs from a

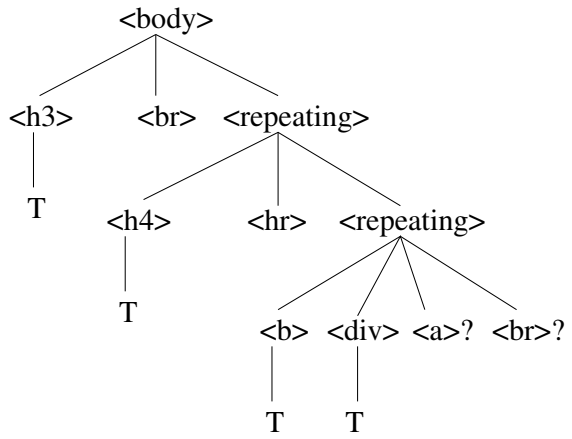


Figure 6.5: The tree representing the inferred grammar of Figure 6.4.

given set of descriptive XML documents. They identify *single occurrence regular expressions*, a class of deterministic regular expressions that allows each non-terminal to appear once only.
Limiting a regular expression to contain each nonterminal once is too restrictive, hence the authors define a class of deterministic regular expressions called *k-occurrence regular expression* that allows each nonterminal to appear at most k times within a regular expression [12, 14]. This algorithm is computationally expensive, hence only small values for the size of k and the alphabet size are considered, and it is rather slow even for such small values. The performance of this algorithm highly depends on the size of the sample set, and requires large sample sizes to perform well (e.g. thousands of samples for medium-size grammars). Moreover, the success rate of the algorithm when the alphabet size is rather large (more than 10), or when the nonterminals repeat often (1.8 or higher on average) within the regular expression is low. It fails when one nonterminal appears many times (more than 4) within a regular expression, which is common in cases such as HTML data. Therefore, this algorithm it is not suited well for applications like extracting data from HTML pages that might require inferring non-deterministic and complex regular expressions in a timely manner.

Unlike inferring DTDs, to infer an XML schema from XML documents the problem of defining complex types should also be addressed. Bex et al. [15] and Chidlovskii [29] propose that nonterminals be assigned types that depend on their contexts (either ancestry or content models), and as a consequence, if a nonterminal has two children with the same root labels, the same types are assigned to those children. Obviously this is not true in some contexts, such as HTML documents, where for example two nodes with label `<div>`

might represent two completely different types.

In summary, DTDs and XSDs impose some limitations on documents which makes them insufficient to model the structure of documents with presentational markup. Consequently, the algorithms for inferring DTDs and XSDs make simplifying assumptions about the structure of documents that makes them less effective for learning grammars for presentational XML documents. Also such algorithms rely on large sample sets that is typically not available in the case of presentational markup.

Because of the way tags are used in documents with presentational markup, they typically have more complex structures than the ones with descriptive markup. DTDs and XSDs are aimed for descriptive XMLs, and hence they are free to impose some restrictions to simplify parsing. For example competing non-terminals (i.e. non-terminals that have the same terminals in their production rules, e.g. $A \rightarrow tR_1$ and $B \rightarrow tR_2$) cannot occur anywhere in a DTD, and XSDs do not allow competing non-terminals within a production rule [87]. As explained in the previous section, this is clearly inadequate for presentational XML. Fortunately most applications that need grammars describing the structure of presentational documents do not require the form to be either DTDs or XSDs.

Web page wrappers [123] are programs that extract semi-structured information from collections of web pages generated from a common template. The problem of wrapper generation is to construct wrappers using a training set. The generated wrappers will be later used to extract data from pages that have similar layouts to the ones in the training set. Wrapper generation algorithms mostly assume some manual tuning by a user. Automatic wrapper generation is also known as wrapper induction [68, 69, 120]. This problem is inherently different from the problem of extracting data from a single page. Wrapper induction algorithms assume a large training set [30, 72, 122]. Moreover, they typically use methods that we explained earlier in this section to learn the structure of a single page, and then combine such structures to infer a grammar for the whole collection [68, 76, 120, 122, 123]. As a result, they inherit similar shortcomings.

Unlike data-centric pages using descriptive markup, pages created with presentation markup are often idiosyncratic: many sample documents are often not generated from the same source using the same grammar. For example, web pages from different websites have different structures, and even pages from the same website might be published with variations in their structures. Hence, providing a large sample set of presentational markup documents with the same grammar is difficult and even infeasible in some cases. In practice, in many data extraction scenatios, the number of samples is very limited [33, 74, 75], and in some cases it is assumed that only one sample is provided [74, 75].

Many information extraction algorithms need to recognize data objects that appear

more than once in a web page. Therefore, most of the algorithms summarized in this section try to detect repetitive patterns in an HTML document as a part of their data extraction process. Some algorithms assume a list of *data records* are represented in the form of tables (without necessarily using HTML table tags), and hence they call a list of data records a *data table*. Liu et al. propose MDR [74] and DEPTA [116], very similar methods for extracting structured data from web pages. It is assumed that a web page contains a list of at least two data records, represented as a Document Object Model (DOM) tree. It is assumed that a repetitive pattern does not contain optional subexpressions. For example if the repeating pattern is of the form $(AC?B)^*$, these algorithms cannot correctly detect it. Also they traverse a tree in a preorder fashion, and if they detect a repetitive pattern among the children of a node, they stop without continuing down to the children. Therefore, they do not detect nested repeating patterns.

NET [75] is another algorithm that is proposed to detect and extract data records from web pages containing a list of at least two data records. It infers only single occurrence regular expressions, and thus languages such as $(ABCA)^*$ are not detected correctly. In fact, if a non-terminal appears more than once in a sequence, NET considers it as repeating,and thus *abcda* is detected as $(AB?C?D?)^*$. As a further restriction, it does not allow the first non-terminal in a repetitive expression to be optional, e.g. $(A?BC)^*$ is not recognizable. In other work, the authors propose a similar algorithm [57] that detects repetitive patterns using a non-deterministic finite automaton, but they assume none of the optionals in a repeating pattern are missing in its first occurrence. Similar to NET, they also assume non-terminals appear at most once in a repetitive pattern.

Alvarez et al. [2] propose an algorithm for extracting a list of data records from a given web page. Their algorithm consists of methods for locating the main list of records in a page, partitioning the list into records, and extracting data from each record. They assume a data record consists of a consecutive sequence of subtrees. They propose an algorithm for clustering subtrees, and assume the first or the last subtree are not optional and do not repeat within the record. They also assume the subtrees that appear before or after the list are not similar to any of the subtrees that form data records in the list and do not belong to the same cluster as any of them. Finally, they do not handle nested regular expressions, and their clustering algorithm does not work when XML tags contain nested repeating subtrees.

Other data extraction systems [71, 81, 102, 106, 25] do not infer a grammar. They instead rely on heuristics to detect data records. For example they assume each data record contains a link to a detail page [71], or they ignore the tree structure of pages [106, 25]. In summary such approaches fail when a document contains nested repetitive patterns [74, 116, 25], a repetitive pattern contains expressions in disjunctive form [74, 116, 106], or

the unknown grammar violates the single occurrence constraint [81, 75]. These limitations significantly affect their performance, and they have not been deployed in other applications dealing with presentational markup, such as rendering engines and table extractors.

6.4 Experimental Results

In this section we empirically evaluate our approach and compare it against various alternatives.

- We categorize grammars that describe presentation XML documents by their imposed restrictions (e.g. whether optional patterns are allowed or not).
- Using real data sets collected from the Web, we compare the classes of grammars in terms of their ability to correctly describe a sample presentation XML document in practice.
- We compare the existing approaches with respect to how well they can correctly recognize samples generated from each class of grammars.
- We evaluate representative algorithms for each class of grammars using real and generated data sets, and compare them in terms of the percentage of samples that they correctly recognize.

6.4.1 Experiment Setup

Alternative grammars: To compare approaches, we consider the class of grammars that each approach infers and compare them. There may be various implementations to infer a grammar for each class, but even an ideal solution would fail when the unknown grammar does not belong to the class. We consider the following classes of grammars:

- *PMGI:* The class of grammars described in Definition 8.
- *A1:* This class of grammars allows only single occurrence regular expressions. The first non-terminal in a repetitive expression is not allowed to be optional. This class of grammars in adopted in various state-of-the-art data extraction algorithms such as NET [75].

- *A2:* A repetitive pattern is not allowed to contain optional or disjunctive patterns. MDR [74] and DEPTA [116] are restricted to this class of grammars.
- *A3:* Competing non-terminals are not allowed. DTDs belong to this class of grammars.
- *A4:* Competing non-terminals are not allowed within the production rule of a non-terminal. XSDs belong to this class of grammars.

For our experiments, we implemented recognizers for classes A1 and A2 based on NET and DEPTA, respectively. Results for classes A3 and A4 are presented as if the experiments were run with perfect recognizers (i.e., failures are recorded only if the correct grammar is not in the appropriate class).

Data collections: For our evaluations we consider the following sample sets:

- *XHTML pages:* A collection of 100 test pages from various domains such as news, personal web pages, online shops, etc. For each page we manually developed a grammar that describes the structure of that page. We assume incomplete HTML tags are fixed, and there is no error in the HTML code of pages.
- *Presentational MathML*: We collected two sets of mathematical expressions encoded with LaTeX from Wikipedia, and translated them into MathML. The first set includes 50 expressions that contain matrices, and the second set includes 50 expressions that contain series and polynomials. We normalized the expressions by removing values of numbers and names of variables, e.g. $x+1$ is normalized to "$\{variable\} + \{number\}$".
- *Generated strings:* A set of 90 regular expressions, manually created to represent the forms of regular expressions that we have frequently found in the structure of web documents. The regular expressions are categorized by various parameters of complexity. Each category contains 10 arbitrary regular expressions with similar complexity, and we randomly generate 10 distinct instances for each.

Some statistics about each dataset are presented in Table 6.1. The second row shows the average number of instances of repetitive patterns within the provided samples.

6.4.2 Experiment Measures

To compare the grammars and algorithms that recognize them, we consider the following measures. *Accuracy:*

	XHTMLs	MathMLs	Strings
Size	100	100	900
Average repetition	16	5	7

Table 6.1: Dataset statistics

Optionals	Disjunction	Duplicates		Multiple repetition				Result			
		Within same repeating pattern		Common nonterminal		Nested					
		Yes	No	Yes	No			Correct	Subset	Superset	Other
×	×	×	×	×	×	×	PMGI	100%	0%	0%	0%
							A1	100%	0%	0%	0%
							A2	100%	0%	0%	0%
√	×	×	×	×	×	×	PMGI	81%	12%	0%	7%
							A1	69%	7%	0%	24%
							A2	0%	100%	0%	0%
×	√	×	×	×	×	×	PMGI	74%	6%	10%	10%
							A1	0%	6%	18%	76%
							A2	0%	100%	0%	0%
×	×	×	√	×	×	×	PMGI	71%	0%	29%	0%
							A1	0%	0%	79%	21%
							A2	100%	0%	0%	0%
×	×	√	×	×	×	×	PMGI	74%	0%	26%	0%
							A1	0%	0%	80%	20%
							A2	100%	0%	0%	0%
×	×	×	×	×	√	×	PMGI	100%	0%	0%	0%
							A1	0%	0%	0%	100%
							A2	100%	0%	0%	0%
×	×	×	√	√	×	×	PMGI	84%	8%	8%	0%
							A1	0%	0%	62%	38%
							A2	100%	0%	0%	0%
×	×	×	×	×	×	√	PMGI	89%	0%	10%	1%
							A1	20%	0%	40%	40%
							A2	0%	100%	0%	0%
√	×	×	√	√	×	×	PMGI	83%	5%	0%	12%
							A1	0%	0%	0%	100%
							A2	0%	100%	0%	0%

Table 6.2: Accuracy for each category of regular expressions.

6.4.3 Parameter Tuning

According to Definition 5, the value of α has a great impact on the grammars that PMGI can infer. Too low values of α result in many false negatives (i.e. patterns that are repetitive but are not detected as repetitive) and hence a low precision. On the other hand, high values of α result in many false positives (i.e. patterns that are not repetitive but are detected as repetitive) and hence a low recall. To find a value for α that results in a balance between false negatives and false positives, we investigate the performance of PMGI on a data sets of Presentation MathML and XHTML documents when the value of α varies from 0 to 1. As Figure 6.6 implies, the performance for very small values of α is relatively poor because in these cases disjunctive forms are not allowed, or are very restricted, and hence the repetitive patterns that require disjunctive forms are not detected correctly. On the other hand, when α is very close to 1, disjunctive forms are less restricted, and some patterns that are not repetitive are mistakenly recognized as being repetitive patterns. PMGI performs best when α is between 0.2 and 0.6, and thus we set the value of α to 0.4 for our experiments.

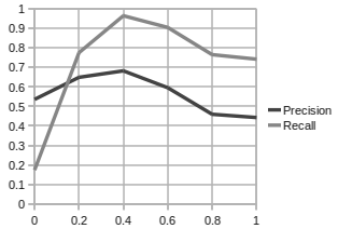


Figure 6.6: Effect of α on the precision and recall of PMGI.

6.4.4 Inferring Grammars

For this experiment we use the *generated strings* dataset. For PMGI the weights of all non-terminals are assumed to be equal. For each sample in the dataset and each class of grammars, we infer a grammar that satisfies the constraints of the class. We calculate the average number of inferred expressions that are equivalent to the original one. If an inferred regular expression is not equivalent to the original regular expression, it might be a subset or a superset of it. It may be that an inferred regular expression is a subset of the original one because the instance is not representative. For example, for original regular expression $(ABC?)^+$ and instance $ABABAB$, the inferred regular expression is $(AB)^+$, which is a subset of the original one. But in some cases, especially with A2, the

	Correct	Subset	Superset	None
PMGI	89%	5%	2.5%	3.5%
A1	69%	4.3%	10.5%	16.2%
A2	52.3%	21.2%	0%	26.5%
A3	31%	-	-	-
A4	44%	-	-	-

Table 6.3: Accuracy of the various classes of grammars on the XHTML dataset.

inferred regular expression is too close to the instance, and so the result is a subset of the original regular expression. For example, for $(ABC?)^+$ with sample $ABCABABC$, A2 infers $ABC(AB)^+C$, which is also a subset of the original expression (Note that this is not compressed so it is not a k-NRE). As the results suggest, for regular expressions with simple structures all classes perform well, but for more complex structures, PMGI outperforms the other alternatives.

Next, we consider the *XHTML* dataset, and similar to the previous experiment, we infer a grammar for each page in the dataset and each class of grammars. For PMGI, the weight of each non-terminal is calculated as explained in Section 6.1.

From each web page we chose some prototypical sections (mostly with repeating patterns) and manually built a grammar for each section. The reason that we considered smaller granularity than a whole page to evaluate the algorithms is that XHTML pages are typically very large, and if an algorithm fails to correctly infer the correct structure of any part, we will conclude that the algorithm fails to infer the structure of the page as a whole. Moreover, in applications depending on data extraction, it often suffices if the algorithm can correctly infer structures for specific sections, namely the data-centric parts of a page. We chose mostly sections that contain repetitive patterns because such sections tend to have more complex structures and they form the basis for recognizing "data records".

The results are calculated as the percentage of grammars that are correctly inferred for each set of web pages, and they are presented in Table 6.3. As the results suggest, A1 grammars tend to be over-generalizations of the original grammars while A2 grammars tend to be under-generalized. A3 does not allow competing non-terminals and A4 does not allow them within a production rule. Hence, they perform poorly for this dataset as many pages contain competing non-terminals (e.g. two different sections of a page with `<div>` tag, or two `<td>` columns with different structures). PMGI outperforms the other alternatives by at least 30%.

	None	Optionals	Repeated non-terminals	Nested
None	100%	92%	100%	100%
Optionals	-	-	77%	71%
Repeated non-terminals	-	-	-	73%

Table 6.4: The performance of PMGI on collections of pages with specific structural complexities.

We further categorize the web pages according to the features in their grammars and present the result for PMGI for each category separately. The result is shown in Table 6.4.

Finally, to demonstrate the performance of PMGI in inferring grammars of other types of presentational XML documents, we applied it on the *Presentational MathML* dataset. The results are presented in Table 6.6, and some examples of mathematical expressions that PMGI can correctly infer are shown in Table 6.7, where N and V are non-terminals that generate numbers and variables. Some polynomial expressions tend to have very complex structures that cannot be expressed with a k-NRTG, hence the performance is relatively low for polynomials.

In conclusion, the constraints defined by alternative classes of grammars are too restrictive, so that in many cases the correct grammar that generates an instance does not belong to such classes.

		Source	Repetitive pattern	Number of	PMGI	A2	A1
$T \rightarrow text$	1	buzzle.com	$L \rightarrow < li > [T]$ $U \rightarrow < ul > [L^*]$ $(T?A)^* T \underbrace{(B(RT)?RIUIRTR)^*}\ TAT$	5	$\surd$	$\times$	$\times$
$A \rightarrow < a > [T]$	2	imdb.com	$\underbrace{(ATI?)^*}$	26	$\surd$	$\times$	$\surd$
$B \rightarrow < b > [T]$	3	imdb.com	$\underbrace{(ATR)^*}$	2	$\surd$	$\surd$	$\surd$
$R \rightarrow < br >$ $I \rightarrow < i > [T]$	4	imdb.com	$L \rightarrow < li > [A]$ $U \rightarrow < ul > L^*$ $\underbrace{(AU)^*}$	2	$\surd$	$\times$	$\surd$
	5	buzzle.com	$L \rightarrow < li > [T]$ $U \rightarrow < ul > [L^*]$ $M \rightarrow < li > [T(AT)?]$ $O \rightarrow < ul > [M^*]$ $N \rightarrow < ol > [L^*]$ $(T?A)^* TR \underbrace{(BR(TATR)?BOBUBNR)^*}\ TAT$	3	$\surd$	$\times$	$\times$
	6	amazon.com	$L \rightarrow < li > [(AT?)^*]$ $\underbrace{L^*}$	2	$\times\ \alpha = 0.4$ $\surd\ \alpha \geq 0.5$	$\times$	$\surd$
	7	tribute.ca	$S \rightarrow < span > [ATA]$ $G \rightarrow < strong > [T]$, $P \rightarrow < span > [G]$ $D \rightarrow < div > [T]$ $P \rightarrow < div > [SPT(DTDR?)^*]$ $\underbrace{(PR?)^*}$	2	$\surd$	$\times$	$\times$
	8	amazon.ca	$H \rightarrow < h6 > [T]$ $\underbrace{(HA^*)^*}$	8	$\surd$	$\times$	$\surd$

Table 6.5: Examples of inferred grammars for expressions from various web pages.

Matrices	91 %
Polynomials	74%
Series	83%

Table 6.6: Accuracy of PMGI on MathML dataset.

Expression	Regular tree grammar
$p_n = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!}$	$M \rightarrow < mrow > [NO]$ $U \rightarrow < msub > [IN]$ $F \rightarrow < frac > [NM]$ $P \rightarrow < mo > +$ $M \rightarrow < mrow > [UPN \underbrace{(PF)^*}]$
$2x^2 + 3y^2 + 2z^2 = 1$	$E \rightarrow < msup > [IN]$ $T \rightarrow +$ $P \rightarrow < mo > [T]$ $L \rightarrow =$ $Q \rightarrow < mo > [L]$ $M \rightarrow < mrow > [\underbrace{(NEP?)^*} QN]$
$\begin{bmatrix} \lambda_1 & \gamma_1 & \beta_1 \\ \lambda_2 & \gamma_2 & \beta_2 \\ \lambda_3 & \gamma_3 & \beta_3 \end{bmatrix}$	$I \rightarrow < msub > [VN]$ $D \rightarrow < mtd > [I]$ $R \rightarrow < mtr > [D^*]$ $M \rightarrow < mfenced > [\underbrace{R^*}]$

Table 6.7: Examples of grammars for mathematical expressions.

Chapter 7

Grammar Inference For Math Retrieval

In the previous chapter we described a framework to infer grammars for semi-structured objects. Now we describe how a grammar inference algorithm can be used to improve the results of a structural similarity algorithm. We also describe an algorithm to optimize processing pattern queries with repetition.

7.1 Extending Structural Similarity

In this section, we consider cases where the query or a document's matching expression contain a pattern that may repeat arbitrarily. As an example consider the query "$x + x^2 + x^3 + x^4 + x^5$", and the expression "$x + x^2 + \cdots + x^n$". Although they are similar, their similarity may not be correctly captured with the algorithms described in Chapter 3. A grammar that describes both of them is $x + [x^{[N]}]^+$, and obviously their similarity is better captured if the grammars are inferred correctly. In the rest of this chapter we address the following problems:

- Given a math expression, infer a grammar that describes it. The inference algorithm should be extended to consider math-specific symbols such as "...".
- Assuming correct grammars are inferred, extend the retrieval algorithms described in Chapter 3 to consider such grammars.

7.1.1 Repetition In Math Expressions

Symbols such as "..." and ":" that denote repetitions appear frequently in math expressions. As an example consider the following expression that appeared earlier in this thesis:

$$\begin{pmatrix} \cdots & \cdots & \cdots & \cdots & \cdots \\ & & \vdots & & \\ \infty & \cdots & \infty & \cdots & \infty \\ & & \vdots & & \\ \cdots & \cdots & \cdots & \cdots & \cdots \end{pmatrix}$$

We call such notations *dots* symbols. Dots symbols replace parts of an expression that are repetitions of the preceding or succeeding subexpressions, introduced to make it more concise. For example using dots we can represent $x^2 + x^3 + x^4 + x^5 + x^6$ with this shorter expression: $x^2 + \cdots + x^6$. However, with the techniques presented so far we cannot correctly infer a grammar that describes the former expression from the latter. We observe the following properties:

- At least one instance of the subexpression that repeats precedes a dots operator. E.g. x^2 in $x^2 + \cdots + x^6$.
- A subexpression that succeeds a dots operator is often also an instance of the repeating subexpression. E.g. x^6 in $x^2 + \cdots + x^6$. However, no subexpression repeats after the dots operator in $x^2 + x^3 + \ldots$.
- Often at least two instances of the repeating subexpression appear near the dots operator. E.g. $x^2 + \cdots + x^6$ and $x^2 + x^3 + \ldots$.
- The operator that connects the instances of the repeating subexpression usually precedes the dots operator. It succeeds the dots operator if an instance of the repeating subexpression also succeeds the dots operator. E.g. $+$ in $x^2 + \cdots + x^6$.

We infer a grammar for expressions with dots operators by starting with the sequence of the siblings of the node representing the dots operator (Figure 7.3). Hence, following Algorithm 17, we cluster the sibling subtrees and transform the sequence to a sequence of cluster ids. Because the dots operator is not part of the repeating pattern, we remove its corresponding cluster id from the sequence. We next run Algorithm 16 to find the repetitive patterns. According to the above observation, in most cases at least two occurrences of

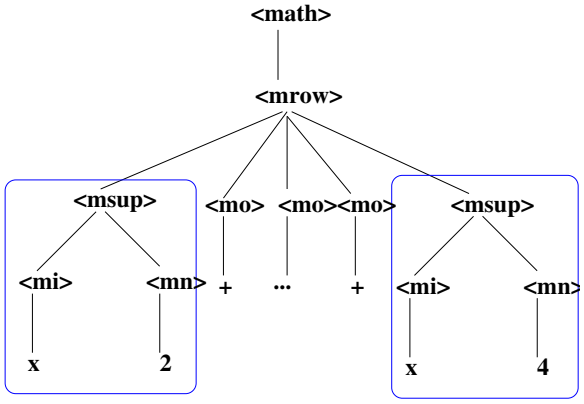


Figure 7.1: Presentation MathML for $x^2 + \cdots + x^6$.

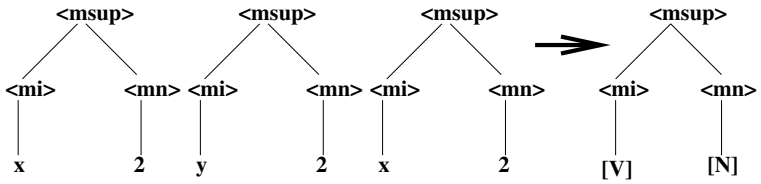


Figure 7.2: Combining subtrees to obtain a template.

the repeating pattern exist. Hence, Algorithm 16 is able to find such occurrences. Also we observe the dots operator is most likely preceded by the operator that separates the occurrences of the repeating pattern (which we call the connecting operator, e.g. + in Figure 7.3), hence we modify Algorithm 16 to consider this. More specifically, we limit the loop at Line 5 of Algorithm 16 by limiting s (Line 6) to be the corresponding cluster id for the connecting operator.

Note that if an expression does not contain a dots operator, we can still apply Algorithm 17 to infer its grammar.

While merging subtrees of the same cluster to obtain a template, we perform the following normalizations. To merge number or variable nodes with different labels, instead of adding one node for each label in a conjunctive form, we create a new node whose label is a wild card for numbers. For example to merge subtrees for x^2, x^3, and x^4 instead of creating a template such as $x^{2|3|4}$ we use a number wildcard (e.g. $x^{[N]}$), and similarly for variables. Finally, we remove symbols indicating optional or disjunctive subtrees (e.g. '*' symbol). Examples are shown in Figures 7.2 and 7.3.

In the next section we describe an approach to extend math retrieval by considering

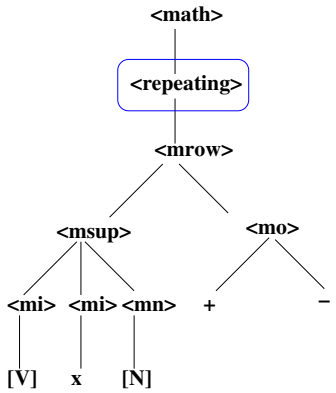


Figure 7.3: Inferred Grammar for $ax^2 - bx^3 + ax^4 - bx^5 + \ldots$.

grammars of math expressions to compare them.

7.1.2 Using Grammars For Math Retrieval

Assume grammars for all expressions with repetitive patterns are inferred. In the remainder of this section we describe how this information can be deployed to increase the recall of a math retrieval algorithm based on structural similarity search. For the ease of explanation assume the query consists of a single math expression. Also assume the search problem is to find the top-k documents.

Consider an expression E in the collection that belongs to document d. We infer a grammar that describes E and if it contains a repetitive pattern, we add the inferred grammar (after the described normalizations, e.g. Figure 7.3) as a new math expression to d. Hence, d contains both E and its inferred grammar. For the rest of this chapter we assume all documents are modified this way. Note that a document that does not contain a math expression with a repetitive pattern is the same as its modified form. We also modify a query similarly.

We similarly process a modified query that contains a repetitive pattern. Note that if the query consists of a single math expression with a repetitive pattern, the modified query contains the original expression and its inferred grammar.

In Chapter 5 we discussed how a rich query with multiple components is processed. We similarly process a modified query. The advantage of this approach is that documents containing expressions with similar grammars to those in the query are still retrieved, while documents containing closer matches are ranked higher. For example if the query is

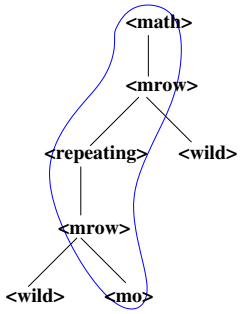


Figure 7.4: The tree representing $\{[N]+\}\{2,\}[N]$.

$x^2 + \cdots + x^n$, a document that contains $x^2 + x^3 + x^4$ is retrieved, but another document that contains a closer match such as $y^2 + \cdots + y^k$ is ranked higher.

7.2 Further Optimizing Pattern Search

In Section 4.4 we described an algorithm to optimize the processing of pattern queries. The described algorithm is based on fast filtering of irrelevant expressions and parsing the resulting candidate expressions with the query to find matches. This often reduces the query processing time significantly, but in some cases where the query consists of a repeating pattern, even after filtering expressions, too many candidates may still remain to be matched against the query.

Example 15. *Assume the query is* $\{[N]+\}\{2,\}[N]$ *that is, a number and the plus operator repeat at least twice followed by a number. It matches* $1+2$*,* $1+3+9$*, etc. The normalized tree for this query is shown in Figure 7.4. The only maximal-constant subtree consists of a single node with tag <mo> which appears in many expressions. Also the only maximal-constant path is <math><mrow> which also appears in many expressions.*

Therefore, processing such queries requires matching the query against many expressions which is a time-consuming task.

To optimize the processing of such queries, we build an index as follows. The index structure is the same as described in Section 4.4. We similarly, transform each expression and add it to the index. For each expression we infer a grammar (Section 4.4). If the grammar contains a repeating pattern, we also add the inferred grammar to the index similar to an ordinary math expression (by transforming it first, and then creating a pseudo-document for it, e.g. Figure 7.5).

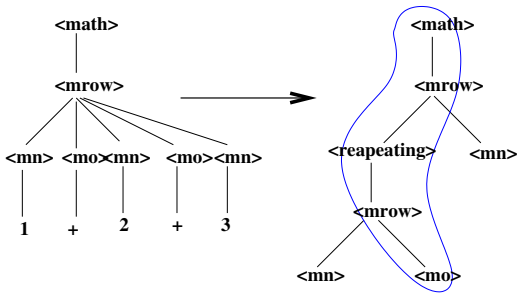


Figure 7.5: Inferring a grammar and transforming it for $1 + 2 + 3$.

To process a query we modify the definition of constant nodes to also include the ones that represent a repetition ("<repeating>" tag). Hence, paths or subtrees that contain such nodes can be maximal-constant. For example the paths marked in Figures 7.4 and 7.5 are maximal-constant.

The modified index allows retrieval of expressions that contain such paths. In most cases, the number of such candidate expressions is much lower as compared to the number in the original optimization algorithm.

7.3 Experiments

In this section we present the result of the empirical evaluation of the proposed algorithms. We first investigate how modifying the structural similarity search with respect to the inferred grammars enhances the accuracy of results. Next, we compare the query processing time of the pattern search algorithm when it is optimized for queries with repetitive patterns.

7.3.1 Alternative Algorithms

For the accuracy of results, we consider the proposed algorithm (represented by ReptSim-Search) and also the alternative algorithms described in Section 3.5.1.

7.3.2 Data And Query Collections

For this study, we use the collection of expressions described in Section 4.5.1. We enhance documents by inferring grammars and adding the ones that contain repetitive patterns to

them.

Forum Query Collection	
Number of queries	10
Average size of query math expressions	34

Table 7.1: Math queries with repetition.

We gather a collection of queries with repetitive patterns from a math forum in a similar way as described in Section 4.5.1. We manually read discussion threads and formed representative queries for each thread. For this experiment, we only consider queries with repetitive patterns. We also form pattern queries as previously described. Statistics about the query collection are presented in Table 7.1.

7.3.3 Evaluation Measures

We compare the algorithms in terms of their accuracy. We consider NFR and MRR metrics as described in Subsection 3.5.2.

7.3.4 Evaluation Results

The NFR and MRR of the described algorithms is presented in Table 7.2. The NFR of ReptSimSearch and SimSearch is 1 which implies that both algorithms return some results for all queries. However, the MRR of SimSearch is rather low, but the MRR of ReptSimSearch is higher. This is because for some queries, there is no close match but there are expressions whose grammars match the grammar of the query. TextSearch also returns some results for all queries but it has a poor accuracy. MiaS produces a result for most queries, but its accuracy is still low. The other algorithms do not produce any results for most queries except when there is an exact match (or normalized exact match). In conclusion, ReptSimSearch performs better than the other algorithms on queries with repetitive patterns. Note that if a query does not contain a repetitive pattern, this algorithm produces the same results as SimSearch.

Next, we compare the query processing time of PatternSearch when a query contains repetitive patterns after the proposed modifications. We denote the modified algorithm as ModPatternSearch. The average query processing time of PatternSearch is $3718ms$, while ModPatternSearch has an average query processing time of $1478ms$, which is less than half as long. Our results confirm that when a pattern repeats and the maximal-constant

Algorithm	NFR	MRR
ReptSimSearch	100%	0.68
SimSearch	100%	0.46
PatternSearch	40%	1
MiaS	80%	0.42
TextSearch	100%	0.27
ExactMatch	20%	1
NormalizedExactMatch	40%	1
SubexprExactMatch	20%	1
NormalizedSubExactMatch	40%	1

Table 7.2: Algorithms' performance for queries with repetitive patterns.

paths or subtrees in the query are not enough to filter out non-matching expressions, the algorithm performs much better if we use repeating patterns to filter out more expressions.

Chapter 8

Conclusions

Semistructured objects and XML documents are widely used to publish data on the Web or in digital libraries. Processing such objects is the basis of many services such as special-purpose search systems. Such pages are usually encoded with presentation markup that mainly aims to describe how such documents are visualized rather than how the underlying data is organized. This makes it more difficult to process such objects.

Mathematical expressions are representative of XML documents with fairly complex structures and rather few symbols. They appear in many online documents and retrieving such documents with respect to their mathematical content is appealing in various domains. Hence, we focused on math retrieval as a representative example and proposed algorithms that often can be extended to other types of objects with presentational XML encoding.

In Chapter 3, we considered various similarity measures to compare math expressions and proposed corresponding retrieval algorithms. Through extensive empirical evaluations on real data and realistic scenarios, we showed that math retrieval based on structural similarity and pattern search outperform other approaches. We also noted that forming queries for structural similarity search is easier, while pattern search is more accurate. Hence the former approach is preferred for typical users.

Considering structures to compare math expressions is computationally expensive. Also, the number of expressions that should be compared for each search is quite large, which makes applications of such approaches limited. Therefore, in Chapter 4 we proposed optimization techniques that significantly reduce the query processing time without affecting the accuracy of results.

We initially assumed a query consists of a single math expression. In Chapter 5 we

relaxed this assumption and proposed algorithms to handle rich queries that are combinations of one or more math expressions and textual keywords.

We also considered the problem of inferring a grammar. We assumed only one sample with presentational markup is provided and supposed the sample contains a pattern that repeats. In Chapter 6 we proposed a framework to infer grammars for such samples and relaxed some restrictions imposed by the existing algorithms that limit the their application. Using this framework, in Chapter 7 we proposed techniques to enhance the result of our math retrieval system for queries that contain math expressions with repeating patterns.

Because we assume math expressions are encoded with respect to their appearance, considering math equivalences to compare them is a very difficult problem. However, if we can infer the content of math expressions from their presentation, we can better compare them and significantly enhance the retrieval results. Hence, one direction for future research is to consider math equivalences for expressions with presentational encoding and to extend the retrieval algorithm accordingly.

Further optimizing the proposed algorithms for web-scale datasets is another direction for future work. While we showed that our algorithms work well in terms of query processing time for a specific domain or digital library, a web-scale dataset that is larger by an order of magnitude or more requires further optimizations.

Bibliography

[1] www.wolframalpha.com.

[2] M. Álvarez, A. Pan, J. Raposo, F. Bellas, and F. Cacheda. Extracting lists of data records from semi-structured web pages. *Data Knowl. Eng.*, 64(2):491–509, 2008.

[3] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram. TeXQuery: a full-text search extension to XQuery. In *The International World Wide Web Conference (WWW)*, pages 583–594, 2004.

[4] S. Amer-Yahia, E. Curtmola, and A. Deutsch. Flexible and efficient XML search with complex full-text predicates. In *ACM International Conference on Management of Data (SIGMOD)*, pages 575–586, 2006.

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

[6] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[7] G. Bancerek. Information retrieval and rendering with MML Query. In *Mathematical Knowledge Management (MKM) Conference*, pages 266–279, 2006.

[8] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. Io-top-k: Index-access optimized top-k query processing. In *International Conference on Very Large Data Bases (VLDB)*, pages 475–486, 2006.

[9] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernndez, M. Kay, J. Robie, and J. Simon. XML Path Language (XPath) 2.0. 2007.

[10] E. Bertino and G. Guerrini. *Object-Oriented Databases.* Wiley Encyclopedia of Computer Science and Engineering, 2008.

[11] E. Bertino, B. C. Ooi, R. Sacks-Davis, K.-L. Tan, J. Zobel, B. Shidlovsky, and B. Catania. *Indexing Techniques for Advanced Database Systems.* Kluwer, 1997.

[12] G. J. Bex, W. Gelade, F. Neven, and S. Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. *ACM Transactions on the Web*, 4(4):14:1–14:32, 2010.

[13] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise DTDs from XML data. In *International Conference on Very Large Data Bases (VLDB)*, pages 115–126, 2006.

[14] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren. Inference of concise regular expressions and DTDs. *ACM Transactions on Database Systems*, 35(2), 2010.

[15] G. J. Bex, F. Neven, and S. Vansummeren. Inferring XML Schema definitions from XML data. In *International Conference on Very Large Data Bases (VLDB)*, pages 998–1009, 2007.

[16] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *International Conference on Data Engineering (ICDE)*, pages 431–440, 2002.

[17] P. Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.

[18] S. Boag, D. Chamberlin, M. F. Fernndez, D. Florescu, J. Robie, and J. Simon. XQuery 1.0: An XML Query Language. 2007.

[19] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Fifth Edition).* W3C Recommendation, 2008.

[20] J.-M. Bremer and M. Gertz. Integrating document and data retrieval based on XML. *The VLDB Journal*, 15:53–83, 2006.

[21] S. Buswell, O. Caprotti, D. P. Carlisle, M. C. Dewar, M. Gaëtano, and M. Kohlhase, editors. *The OpenMath Standard, Version 2.0.* The OpenMath Esprit Consortium, 2004.

[22] M. J. Cafarella, A. Y. Halevy, D. Z. Wang, E. Wu, and Y. Zhang. Webtables: exploring the power of tables on the web. *International Conference on Very Large Data Bases (PVLDB)*, 1(1):538–549, 2008.

[23] D. Carlisle. OpenMath, MathML, and XSL. *SIGSAM Conference on Symbolic and Algebraic Manipulation*, 34(2):6–11, 2000.

[24] D. Carlisle, P. Ion, and R. Miner. *Mathematical Markup Language (MathML) Version 3.0*. W3C Recommendation, 2010.

[25] C.-H. Chang and S.-C. Lui. IEPAD: Information extraction based on pattern discovery. In *The International World Wide Web Conference (WWW)*, pages 681–688.

[26] K. C.-C. Chang and S. won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *ACM International Conference on Management of Data (SIGMOD)*, pages 346–357, 2002.

[27] Y. Chen, W.-Y. Ma, and H. Zhang. Detecting web page structure for adaptive viewing on small form factor devices. In *The International World Wide Web Conference (WWW)*, pages 225–233, 2003.

[28] R. Chenna, H. Sugawara, T. Koike, R. Lopez, T. J. Gibson, D. G. Higgins, and J. D. Thompson. Multiple sequence alignment with the Clustal series of programs. *Nucleic Acids Res*, 31:3497–3500, 2003.

[29] B. Chidlovskii. Schema extraction from XML data: A grammatical inference approach. In *International Workshop on Knowledge Representation meets Databases*, 2001. 10 pp.

[30] B. Chidlovskii, B. Roustant, and M. Brette. Documentum eci self-repairing wrappers: performance analysis. In *ACM International Conference on Management of Data (SIGMOD)*, pages 708–717, 2006.

[31] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A semantic search engine for XML. In *International Conference on Very Large Data Bases (VLDB)*, pages 45–56, 2003.

[32] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 2007. Available on: `http://www.grappa.univ-lille3.fr/tata`.

[33] V. Crescenzi, G. Mecca, and P. Merialdo. RoadRunner: Towards automatic data extraction from large web sites. In *International Conference on Very Large Data Bases (VLDB)*, pages 109–118, 2001.

[34] T. Dalamagas, T. Cheng, K.-J. Winkel, and T. K. Sellis. A methodology for clustering xml documents by structure. *Inf. Syst.*, 31(3):187–228, 2006.

[35] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. *ACM Transactions on Algorithms*, 6(1), 2009.

[36] P. F. Dietz. Maintaining order in a linked list. In *STOC*, pages 122–127, 1982.

[37] T. H. Einwohner and R. J. Fateman. Searching techniques for integral tables. In *The International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 133–139, 1995.

[38] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences - Elsevier*, 66(4):614–656, 2003.

[39] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and searching XML data via two zips. In *The International World Wide Web Conference (WWW)*, pages 751–760, 2006.

[40] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *The Journal of the ACM (JACM)*, 57(1), 2009.

[41] M. N. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: Learning document type descriptors from XML document collections. *Data Mining and Knowledge Discovery*, 7(1):23–56, 2003.

[42] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[43] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *International Conference on Very Large Data Bases (VLDB)*, pages 436–445, 1997.

[44] H. Gonzalez, A. Y. Halevy, C. S. Jensen, A. Langen, J. Madhavan, R. Shapley, W. Shen, and J. Goldberg-Kidon. Google fusion tables: web-centered data management and collaboration. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1061–1066, 2010.

[45] G. Grätzer. *Math into LaTeX*. Birkhauser, 3rd edition, 2000.

[46] J. Grimm. *Tralics, A LaTeX to XML Translator*. INRIA, 2008.

[47] F. Guidi and I. Schena. A query language for a metadata framework about mathematical resources. In *MKM*, pages 105–118, 2003.

[48] U. Güntzer, W.-T. Balke, and W. Kießling. Optimizing multi-feature queries for image databases. In *International Conference on Very Large Data Bases (VLDB)*, pages 419–428, 2000.

[49] U. Güntzer, W.-T. Balke, and W. Kießling. Towards efficient multi-feature queries in heterogeneous environments. In *International Symposium on Information Technology (ITCC)*, pages 622–628, 2001.

[50] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *ACM International Conference on Management of Data (SIGMOD)*, pages 16–27, 2003.

[51] J. R. Hamilton and T. K. Nayak. Microsoft SQL Server full-text search. *IEEE Data Engineering Bulletin*, 24(4):7–10, 2001.

[52] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *International Conference on Very Large Data Bases (VLDB)*, pages 850–861, 2003.

[53] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava. Keyword proximity search in XML trees. *IEEE Transactions on Knowledge and Data Engineering*, 18(4):525–539, 2006.

[54] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB Journal*, 13(3):207–221, 2004.

[55] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, 40(4), 2008.

[56] A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[57] N. Jindal and B. Liu. A generalized tree matching algorithm considering nested lists for web data extraction. In *SIAM Conference on Data Mining (SDM)*, pages 930–941, 2010.

[58] S. Kamali, J. Apacible, and Y. Hosseinkashi. Answering math queries with search engines. In *The International World Wide Web Conference (WWW)*, pages 43–52, 2012.

[59] S. Kamali and F. W. Tompa. Improving mathematics retrieval. In *MKM/Calculemus/DML*, pages 37–48, 2009.

[60] S. Kamali and F. W. Tompa. A new mathematics retrieval system. In *Conference on Information and Knowledge Management (CIKM)*, pages 1413–1416, 2010.

[61] S. Kamali and F. W. Tompa. Grammar inference for web documents. In *International Workshop on the Web and Databases (WebDB)*, 2011.

[62] S. Kamali and F. W. Tompa. Retrieving documents with mathematical content. In *ACM Special Interest Group on Information Retrieval (SIGIR)*, 2013.

[63] S. Kamali and F. W. Tompa. Structural similarity search for mathematics retrieval. In *MKM/Calculemus/DML*, pages 246–262, 2013.

[64] Y. Kanza and Y. Sagiv. Flexible queries over semistructured data. In *PODS*, pages 40–51, 2001.

[65] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *ACM International Conference on Management of Data (SIGMOD)*, pages 133–144, 2002.

[66] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *International Conference on Data Engineering (ICDE)*, pages 129–140, 2002.

[67] M. Kohlhase and I. A. Sucan. A search engine for mathematical formulae. In *Artificial Intelligence and Symbolic Computation (AISC)*, pages 241–253. Springer, 2006.

[68] N. Kushmerick. *Wrapper induction for information extraction*. PhD thesis, Dept of Computer Science and Engineering, Univ of Washington. Technical Report UW-CSE-97-11-04, 1997.

[69] N. Kushmerick. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence*, 118(1–2):15–68, 2000.

[70] C. Laitang, M. Boughanem, and K. Pinel-Sauvagnat. XML information retrieval through tree edit distance and structural summaries. In *Asia Information Retrieval Societies Conference (AIRS)*, pages 73–83, 2011.

[71] K. Lerman, L. Getoor, S. Minton, and C. Knoblock. Using the structure of web sites for automatic segmentation of tables. In *ACM International Conference on Management of Data (SIGMOD)*, pages 119–130, New York, NY, USA, 2004.

[72] K. Lerman, S. N. Minton, and C. A. Knoblock. Wrapper maintenance: a machine learning approach. *Journal of Artificial Intelligence Research (JAIR)*, 18(1):149–181, 2003.

[73] C. Li, K. C.-C. Chang, and I. F. Ilyas. Supporting ad-hoc ranking aggregates. In *ACM International Conference on Management of Data (SIGMOD)*, pages 61–72, 2006.

[74] B. Liu, R. Grossman, and Y. Zhai. Mining data records in web pages. In *ACM Conference on Knowledge Discovery and Data Mining (KDD)*, pages 601–606, 2003.

[75] B. Liu and Y. Zhai. NET – a system for extracting web data from flat and nested data records. In *Web Information Systems Engineering (WISE)*, pages 487–495, 2005.

[76] L. Liu, C. Pu, and W. Han. XWRAP: An XML-enabled wrapper construction system for web information sources. In *International Conference on Data Engineering (ICDE)*, pages 611–621, 2000.

[77] S. Liu, Q. Zou, and W. W. Chu. Configurable indexing and ranking for XML information retrieval. In *ACM Special Interest Group on Information Retrieval (SIGIR)*, pages 88–95, 2004.

[78] Z. Liu and Y. Chen. Identifying meaningful return information for XML keyword search. In *ACM International Conference on Management of Data (SIGMOD)*, pages 329–340, 2007.

[79] S. Maclean and G. Labahn. A new approach for recognizing handwritten mathematics using relational grammars and fuzzy sets. *International Journal on Document Analysis and Recognition (IJDAR)*, pages 1–25, 2012.

[80] A. Marian, N. Bruno, and L. Gravano. Evaluating top-k queries over web-accessible databases. *ACM Transactions on Database Systems*, 29(2):319–362, 2004.

[81] G. Miao, J. Tatemura, W.-P. Hsiung, A. Sawires, and L. E. Moser. Extracting data records from the web using tag path clustering. In *The International World Wide Web Conference (WWW)*, pages 981–990, 2009.

[82] S. Michel, P. Triantafillou, and G. Weikum. Klee: A framework for distributed top-k query algorithms. In *International Conference on Very Large Data Bases (VLDB)*, pages 637–648, 2005.

[83] T. Milo and D. Suciu. Index structures for path expressions. In *The International Conference on Database Theory (ICDT)*, pages 277–295, 1999.

[84] J. Mišutka and L. Galamboš. System description: Egomath2 as a tool for mathematical searching on wikipedia.org. In *Calculemus/MKM*, pages 307–309, 2011.

[85] D. M. Mount. *Bioinformatics - sequence and genome analysis (2. ed.)*. Cold Spring Harbor Laboratory Press, 2004.

[86] R. Munavalli and R. Miner. Mathfind: a math-aware search engine. In *ACM Special Interest Group on Information Retrieval (SIGIR)*, pages 735–735, 2006.

[87] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, 5(4):660–704, 2005.

[88] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *International Conference on Very Large Data Bases (VLDB)*, pages 281–290, 2001.

[89] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[90] T. T. Nguyen, K. Chang, and S. C. Hui. A math-aware search engine for math question answering system. In *Conference on Information and Knowledge Management (CIKM)*, pages 724–733, 2012.

[91] Z. Nie, Y. Ma, S. Shi, J.-R. Wen, and W.-Y. Ma. Web object retrieval. In *The International World Wide Web Conference (WWW)*, pages 81–90, 2007.

[92] A. Nierman and H. V. Jagadish. Evaluating structural similarity in xml documents. In *International Workshop on the Web and Databases (WebDB)*, pages 61–66, 2002.

[93] M. Pawlik and N. Augsten. Rted: A robust algorithm for the tree edit distance. *International Conference on Very Large Data Bases (PVLDB)*, 5(4):334–345, 2011.

[94] A. Pillay and R. Zanibbi. Intelligent combination of structural analysis algorithms: Application to mathematical expression recognition. In *International Workshop on Pen-Based Mathematical Computation (PenMath)*, 2009.

[95] P. Placek, D. Theodoratos, S. Souldatos, T. Dalamagas, and T. Sellis. A heuristic approach for checking containment of generalized tree-pattern queries. In *Conference on Information and Knowledge Management (CIKM)*, pages 551–560, 2008.

[96] P. Rao and B. Moon. PRIX: Indexing and querying XML using prüfer sequences. In *International Conference on Data Engineering (ICDE)*, pages 288–300, 2004.

[97] N. Sarkas, G. Das, and N. Koudas. Improved search for socially annotated data. *International Conference on Very Large Data Bases (PVLDB)*, 2(1):778–789, 2009.

[98] T. Schellenberg, B. Yuan, and R. Zanibbi. Layout-based substitution tree indexing and retrieval for mathematical expressions. In *Document Recognition and Retrieval Conference (DRR)*, 2012.

[99] E. S. Smirnova and S. M. Watt. Communicating mathematics via pen-based interfaces. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 9–18, 2008.

[100] P. Sojka and M. Líska. The art of mathematics retrieval. In *ACM Symposium on Document Engineering*, pages 57–60, 2011.

[101] B. Sun, P. Mitra, C. L. Giles, and K. T. Mueller. Identifying, indexing, and ranking chemical formulae and chemical names in digital documents. *ACM Transactions on Information Systems*, 29(2):12, 2011.

[102] M. Takahashi. Generalizations of regular sets and their application to a study of context-free languages. *Information and Control*, 27(1):1 – 36, 1975.

[103] J. Tekli, R. Chbeir, and K. Yétongnon. Efficient xml structural similarity detection using sub-tree commonalities. In *Brazilian Symposium on Databases (SBBD)*, pages 116–130, 2007.

[104] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *International Conference on Very Large Data Bases (VLDB)*, pages 648–659, 2004.

[105] R. van Zwol and P. M. G. Apers. The webspace method: on the integration of database technology with multimedia retrieval. In *Conference on Information and Knowledge Management (CIKM)*, pages 438–445, 2000.

[106] J. Wang and F. H. Lochovsky. Data extraction and label assignment for web databases. In *The International World Wide Web Conference (WWW)*, pages 187–196, 2003.

[107] J.-R. Wen, Q. Li, W.-Y. Ma, and H. Zhang. A multi-paradigm querying approach for a generic multimedia database management system. *SIGMOD Record*, 32(1):26–34, 2003.

[108] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest LCAs in XML databases. In *ACM International Conference on Management of Data (SIGMOD)*, pages 537–538, 2005.

[109] L. Yang, S. Xu, S. Bao, D. Han, Z. Su, and Y. Yu. A study of information retrieval on accumulative social descriptions using the generation features. In *Conference on Information and Knowledge Management (CIKM)*, pages 721–730, 2009.

[110] A. Youssef. Search of mathematical contents: Issues and methods. In *International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE)*, pages 100–105, 2005.

[111] A. Youssef. Methods of relevance ranking and hit-content generation in math search. In *Calculemus/MKM*, pages 393–406, 2007.

[112] R. Zanibbi and D. Blostein. Recognition and retrieval of mathematical expressions. *International Journal on Document Analysis and Recognition (IJDAR)*, 15(4):331–357, 2012.

[113] R. Zanibbi and L. Yu. Math spotting: Retrieving math in technical documents using handwritten query images. In *International Conference on Document Analysis and Recognition (ICDAR)*, pages 446–451, 2011.

[114] R. Zanibbi and B. Yuan. Keyword and image-based retrieval of mathematical expressions. In *Document Recognition and Retrieval Conference (DRR)*, pages 1–10, 2011.

[115] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search - The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Kluwer, 2006.

[116] Y. Zhai and B. Liu. Structured data extraction from the web based on partial tree alignment. *IEEE Transactions on Knowledge and Data Engineering*, 18:1614–1628, 2006.

[117] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing (SICOMP)*, 18(6):1245–1262, 1989.

[118] K. Zhang, R. Statman, and D. Shasha. On the editing distance between unordered labeled trees. *Inf. Process. Lett.*, 42(3):133–139, 1992.

[119] N. Zhang, M. T. Özsu, I. F. Ilyas, and A. Aboulnaga. FIX: Feature-based indexing technique for XML documents. In *International Conference on Very Large Data Bases (VLDB)*, pages 259–270, 2006.

[120] H. Zhao, W. Meng, Z. Wu, V. Raghavan, and C. Yu. Fully automatic wrapper generation for search engines. In *The International World Wide Web Conference (WWW)*, pages 66–75, 2005.

[121] J. Zhao, M.-Y. Kan, and Y. L. Theng. Math information retrieval: user requirements and prototype implementation. In *ACM/IEEE-CS joint conference on Digital libraries*, pages 187–196, 2008.

[122] S. Zheng, R. Song, J.-R. Wen, and C. L. Giles. Efficient record-level wrapper induction. In *CIKM*, pages 47–56, 2009.

[123] S. Zheng, R. Song, J.-R. Wen, and D. Wu. Joint optimization of wrapper generation and template detection. In *ACM Conference on Knowledge Discovery and Data Mining (KDD)*, pages 894–902, 2007.

[124] D. Zhou, J. Bian, S. Zheng, H. Zha, and C. L. Giles. Exploring social annotations for information retrieval. In *The International World Wide Web Conference (WWW)*, pages 715–724, 2008.

[125] M. M. Zloof. Query-by-Example: the invocation and definition of tables and forms. In *International Conference on Very Large Data Bases (VLDB)*, pages 1–24, 1975.