Structural Similarity Search for Mathematics Retrieval*

Shahab Kamali and Frank Wm. Tompa

David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada
{skamali,fwtompa}@cs.uwaterloo.ca

Abstract. Retrieving documents by querying their mathematical content directly can be useful in various domains, including education, engineering, patent research, physics, and medical sciences. As distinct from text retrieval, however, mathematical symbols in isolation do not contain much semantic information, and the structure of an expression must be considered as well. Unfortunately, considering the structure to calculate the 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 optimizations techniques to reduce the index size and the query processing time, and we experimentally validate our system in terms of correctness and efficiency. Thus, with the proposed optimizations, mathematical contents can be fully exploited to rank documents in response to mathematical queries.

1 Introduction

Documents with mathematical expressions are extensively published in technical and educational web sites, digital libraries, and other document repositories such as patent collections. Retrieving such documents with respect to their math content is a challenging problem. 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, a search for documents that include the expression $\int x\sqrt{x^2+a^2}\,dx$ is not likely satisfied by a document that includes $\sqrt{x+2}\int 2ax\,dx$. Moreover, relevant mathematical expressions might include small variations in their structures or symbols. For example, a document including $1+\sum_{i=1}^{n} i^k$ might well be useful in

* Financial assistance for the research was provided by the Natural Sciences and Engineering Research Council of Canada, Mprime, and the University of Waterloo.

J. Carette et al. (Eds.): CICM 2013, LNAI 7961, pp. 246–262, 2013.

response to a query to find documents including $\sum_{j=1}^{n} j^2$. Hence exact matching of mathematical expressions is not a sufficiently powerful search strategy.

The majority of the published mathematical expressions are encoded with respect to their appearance (*presentation*), and most instances do not preserve much semantic information. Content-based mathematics retrieval systems [3,9] are limited to resources that encode the semantics of mathematical expressions, and they do not perform well with expressions encoded using presentation markup. Other systems search based on the presentation of mathematical expressions [2,5,13,18,19], but they either find exact matches only or they use a "bag of symbols" model that often returns many irrelevant results.

Because mathematical expressions are often distinguished by their structure, we should not rely merely on the symbols they include but instead consider a search paradigm that incorporates mathematical structure as well. More specifically, the similarity of two expressions, defined as a function of their structures and the symbols they share [6], can be used as an indication of the relevance of documents when a math expression is given as a query. 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 paper.

The rest of this paper is organized as follows. In Sect. 2 we explain the query language and the search problem. We next describe related work. We describe a structural similarity search algorithm in Sect. 4, and we propose optimization techniques for this algorithm in Sect. 5 and 6. We finally present an evaluation of our algorithm and conclude the paper.

2 The Framework

A mathematical expression is a finite combination of symbols that is formed according to some context-dependent rules. Symbols can designate numbers (constants), variables, operators, functions, and other mathematical entities.

A text document, such as a web page, that contains a mathematical expression is a *document with mathematical content*. We assume that a query is a mathematics expression. Given a query, the search problem is to find the top-k relevant documents, where documents are ranked with respect to the similarity of their mathematical expressions to the query.

Presentation MathML is part of the W3C recommendation that is increasingly used to publish mathematics information on the web, and many web browsers support it. There are various tools to translate mathematical expressions from other languages, including LaTeX, into Presentation MathML. Thus we can assume that stored expressions are encoded in this form when they are indexed. Because forming queries directly with Presentation MathML is difficult, however, input devices such as pen-based interfaces and tablets [11,16] or more

widely-known languages such as LaTeX might be preferred for entering a query. Nevertheless, automatic tools can also be applied to translate queries to Presentation MathML, and therefore, regardless of the user interface, we can assume a query is also represented using this encoding. In summary, it is appropriate to assume that Presentation MathML is employed when querying mathematics information on the web.

3 Related Work

Currently, there are a few alternative approaches to math retrieval. In one approach, expressions that match the query exactly are considered as relevant. Examples include algorithms that are based on comparing images of expressions [20,21] (they calculate the similarity of images, which allows for very limited variation among the expressions returned) or using very detailed and formal query languages that enable database operations to match expressions [2,5]. We characterize such algorithms as *ExactMatch* algorithms in this paper. Some other algorithms perform some normalizations on the query and also on the expressions before exactly matching them [18]. As shown below, ExactMatch and NormalizedExactMatch perform poorly when searching for mathematical content.

As a variant, some algorithms consider retrieving expressions that share substructures with the query [3,6,9,15]. These algorithms do not consider ranking the results when many partial matches exist. We characterize all such algorithms as *SubexprExactMatch* algorithms and note that *normalized subexpression exact match* algorithms are also feasible.

Another approach to math retrieval is to transform an expression into a collection of tokens where each token represents a math symbol or a substructure [13,14,17,19,12]. 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 below.

Algorithms for retrieving general XML documents based on tree-edit distance have been proposed [10], and these could be adapted to match XML-encoded mathematical expressions. An alternative for matching based on structural similarity is to express a query in the form of a template, specifying precisely where variability is

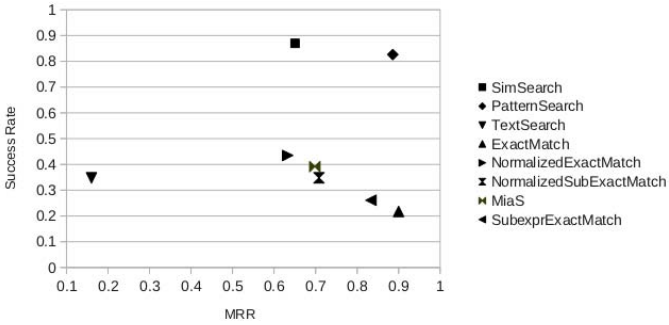


Fig. 1. Mean reciprocal rank versus success rate of each algorithm for Forum queries

permitted and where exact matching is required [7]. This *PatternSearch* approach requires more effort and skill on the part of the user who formulates queries.

Elsewhere [8] we have compared these various approaches in terms of their ability to retrieve documents that contain mathematical expressions that match a query. Some of the results are summarized in Fig. 1, which is explained in more detail in Sect. 7. For the present, we merely observe that similarity search (SimSearch) and PatternSearch outperform the other approaches by a wide margin in terms of accuracy, and forming queries with SimSearch is much easier than with PatternSearch. The goal of this paper is to demonstrate that similarity search can also be sufficiently fast to be used in practice.

4 Structural Similarity Search

Text with XML markup such as Presentation MathML can be naturally expressed as ordered labelled trees, also called Document Object Model (DOM) trees. A DOM tree T is represented by $T = (V, E)$, where V represents the set of nodes and E represents the set of edges of T. A label $\lambda(n)$ is assigned to each node $n \in V$. In this paper we refer to a math expression and its corresponding DOM tree interchangeably.

We define similarity in terms of "tree edit distance" as follows. 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 label ϵ. 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, ω might reflect the design goal that renaming a variable is less costly than renaming a math operator. For ease of explanation, however, we will assume that the costs of all delete and insert operations are 1 and the cost of rename is 2. 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.

A *forest* is an ordered sequence of trees. For example deleting the root of a tree results in a forest that consists of its immediate subtrees. Note that a single tree and the empty sequence of trees are also forests. With these definitions, the following recursive formula can be used to calculate edit distance [22]:

$$
\begin{aligned}
dist(F_1, F_2) &= min \begin{cases} dist(F_1 - u, F_2) + \omega(u \to \epsilon), \\ dist(F_1, F_2 - v) + \omega(\epsilon \to v), \\ dist(F_1 - T_u, F_2 - T_v) + dist(T_u, T_v) \end{cases} \\
dist(T_u, T_v) &= min \begin{cases} dist(T_u - u, T_v) + \omega(u \to \epsilon), \\ dist(T_u, T_v - v) + \omega(\epsilon \to v), \\ dist(T_u - u, T_v - v) + \omega(u \to v) \end{cases}
\end{aligned}
\tag{1}
$$

where F_1 and F_2 are two non-empty forests such that either F_1 or F_2 contains at least two trees, 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.

We calculate the structural similarity of two mathematical expressions E_1 and E_2 represented by trees T_1 and T_2 as follows:

$$sim(E_1, E_2) = 1 - \frac{dist(T_1, T_2)}{|T_1| + |T_2|} \tag{2}$$

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

Assume document d contains mathematical expressions $E_1 \dots E_n$. The rank of d for a query Q is calculated as the maximum similarity of expressions in d:

$$docRank(d, Q) = \max_{E_i \in d} sim(E_i, Q) \tag{3}$$

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 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 DOM 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.

The next two sections address these observations.

5 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.

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:

$$(T_1 -_\tau T_2) = \{N \in T_1 | \lambda(N) \notin \tau(T_2)\} \tag{4}$$

$$T_1 \cap_\tau T_2 = (\{N|N \in T_1\} - (T_1 -_\tau T_2)) \cup (\{N|N \in T_2\} - (T_2 -_\tau T_1)) \quad (5)$$

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

$$|T_1 \cap_\tau T_2| = |T_1| - |T_1 -_\tau T_2| + |T_2| - |T_2 -_\tau T_1| \quad (6)$$

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 DOM tree of E, does not appear in T_Q, the DOM 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 (2) and (6):

$$sim(E, Q) \leq 1 - \frac{|T_E -_\tau T_Q| + |T_Q -_\tau T_E|}{|T_E| + |T_Q|} = \frac{|T_E \cap_\tau T_Q|}{|T_E| + |T_Q|} \quad (7)$$

and the upper bound for the relevance of a document d to Q is calculated using (3):

$$docRank(d, Q) \leq upperRank(d, Q) = \max_{E_i \in d} \frac{|T_{E_i} \cap_\tau T_Q|}{|T_{E_i}| + |T_Q|} \quad (8)$$

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"), as presented in Algorithm 1. This algorithm produces the same results as the naive algorithm, but it reduces the query processing time by avoiding some unnecessary computations. In Sect. 7 we show that this optimization significantly reduces the query processing time.

6 Compact Index and Distance Cache

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 $|T_1| + \cdots + |T_n|$. If two subtrees G_1 and G_2 represent equivalent subexpressions, 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:

$$freq(T, C) = |\{G|G \in_{sub} C \wedge G \sim T\}| \quad (9)$$

Algorithm 1. 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 ← the document referenced by C.
8:    if d_C is null or upperRank(d_C, Q) < min_{d∈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∈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 ← C.next
19: end while
20: return TopK
```

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' DOM trees. To confirm this, we ran experiments on a collection of more than 863,000 math expressions. Details of this collection are presented in Sect. 7.1, and the experimental confirmation is included in Sect. 7.3.

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 Sect. 5, 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 their corresponding entries in the entry of T. This algorithm guarantees that each tree is inserted once only, even if it repeats. Figure 2 shows a fragment of the index after $\frac{x^2-1}{x^2+1}$ is added.

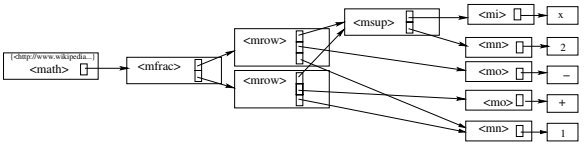


Fig. 2. The index after $\frac{x^2-1}{x^2+1}$ is added

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 (1) 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 1, $S = \{G | G \in_{sub} D\}$, 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.

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.

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.

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 $\mathcal{H}^* = \arg\max_{\mathcal{H}} \sum_{(T_e^i, T_q^i) \in \mathcal{H}} weight(T_e^i, T_q^i)\, benefit(T_e^i, T_q^i)$ such that $|\mathcal{H}| \leq \mathcal{C}$.

If we are given the set P, the problem is easily solved by choosing the $\mathcal{C}$ triples having the highest values for $weight(T_e^i, T_q^i)\, benefit(T_e^i, T_q^i)$. However, Algorithm 1 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 1. Consider the following motivating example:

Example 1. Assume $freq(T_e, D) = 100$, and $freq(T_q, \{T_Q\}) = 1$. The similarity between the expressions represented by T_e and T_q will be calculated at most 100 times by Algorithm 1. 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 $freq(T_e, D)$ 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 $freq(T_e, D)$ for the node corresponding to T_e. Whenever we require a value for $dist(T_e, T_q)$, 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 DOM 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.

Algorithm 2 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 2. 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
  $dist \leftarrow$ computed $dist(T_e, T_q)$ 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 $dist$

In the next section we show that the proposed optimization techniques significantly reduce the query processing time in practice.

7 Experiments

In this section we investigate the performance of the proposed algorithms.

7.1 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 the annotations and translated them into Presentation MathML using Tralics [4]. DLMF pages use Presentation MathML to represent mathematical expressions. Statistics summarizing this dataset are presented in Table 1A.

Table 1. Experimental dataset and query statistics

A	Wikipedia	DLMF	Combined
Num. pages	44,368	1,550	45,918
Num. exprs.	611,210	252,148	863,358
Avg. expr. size	28.3	17.6	25.2
Max. expr. size	578	223	578

B	Interview	Forum	Combined
Num. queries	45	53	98
Avg. query size	14.2	23.8	19.4

Query Collection. 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.
- *Mathematics forum:* People often use mathematics forums in order to ask a questions or discuss math-related topics. Many threads start by a user asking a question in the form of a single mathematics expression. Usually, by reading the rest of a 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.

We only consider queries with at least one match in our dataset. Table 1B summarizes statistics about the queries, where the number of nodes in the query tree is used to represent query size.

Evaluation Measures. We evaluate the proposed algorithms using the following measures:

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. The *Mean Reciprocal Rank* (MRR) is the average reciprocal rank for all queries:

$$MRR = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{\mathcal{R}(q)} \tag{10}$$

where Q is the collection of queries, and $\mathcal{R}(q)$ is the rank of the first relevant answer for query q.

Success-Rate: If at least one of the top 20 search results returned by an algorithm for a given query is relevant, we classify the search as successful. Alternatively, if the first relevant answer is not among the top 20 results, or if no relevant result is returned at all, the search is classified as unsuccessful. The success rate is the number of successful searches divided by the total number of searches:

$$Success\ Rate = \frac{|\{q \in Q | q\ is\ successfully\ searched\}|}{|Q|} \tag{11}$$

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. For a collection of queries, we measure the query processing time of each and report the *average query processing time*.

Alternative Algorithms. We evaluate the described algorithms by comparing their performance against the following alternative algorithms:

- *TextSearch:* The query and expressions are treated as bags of words. A standard text search algorithm is used for ranking expressions 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 replace numbers and variables with generic labels N and V, respectively. The normalized expressions are searched and ranked according to the ExactMatch algorithm.
- *SubexprExactMatch:* An expression is returned as a search result if one of its subexpressions exactly matches the query. Results are ranked by increasing sizes of their DOM trees and ties are broken using the alphabetic order of the name of their corresponding documents.
- *NormalizedSubExactMatch:* Normalization is performed 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*: Expressions are matched using the algorithm proposed by Sojka and Liska [17]: An expression is first tokenized, where a token is a subtree of the expression. Each token is next normalized with respect to various rules (e.g. number values are removed, or variables are removed, or both), and multiple normalized copies are preserved. The result, which is a collection of tokens, is indexed with a text search engine. Each query is similarly normalized (but not tokenized) and then matched against the index.

[1] We used Apache Lucene [1] in our implementation.

- *PatternSearch:* Expressions are matched against a query template as described by Kamali and Tompa [7]. Like SubexprExactMatch, results are ranked with respect to the sizes of their DOM trees.
- *SimSearch:* Expressions are matched against a query according to the algorithm described in Sect. 4.

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. 5. 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. 6.
- *Compact-ET-NMC*: The early termination algorithm with a compact index, and no memory constraint as described in Sect. 6.
- *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. 6). 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-RandMC*: Similar to Compact-ET-MC, but entries are chosen at random for being assigned space in the cache.

7.2 Correctness

Fuller descriptions of the algorithms and correctness results for the experiments are reported elsewhere [8]. For completeness, we summarize the correctness results here.

The success rate against MRR for each algorithm is plotted in Fig. 1 for the Forum queries and in Fig. 3 for the Interview queries. As both figures show, PatternSearch and SimSearch have high success rates and also high MRRs. Interestingly, PatternSearch has a higher MRR because irrelevant expressions are less likely to match a carefully formed pattern, whereas SimSeach has a slightly higher success rate because in some cases even an experienced user may not be able to guess the pattern that will yield a correct answer.

In summary, SimSearch and PatternSearch perform much better than the other approaches in terms of the correctness of results. However, because forming queries for SimSearch is easier, it is generally preferred over PatternSearch.

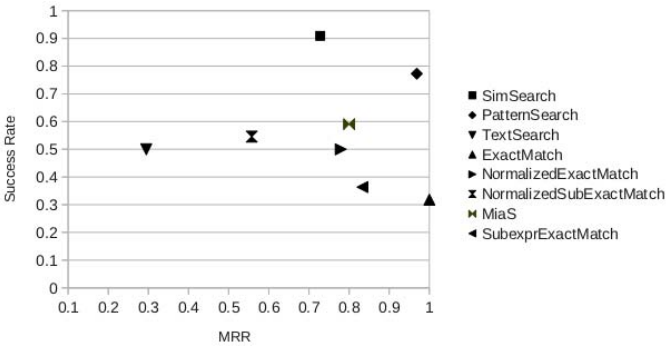


Fig. 3. MRR versus success rate of each algorithm for Interview queries

7.3 Index Size

The average number of repetitions of subtrees with sizes in specific ranges is listed in Table 2A. 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. For our experiments, an expression's signature is computed by a conventional hash function applied to its XML string S: $S[0] * 31^{(z-1)} + S[1] * 31^{(z-2)} + \cdots + S[z-1]$ where $S[i]$ is the i^{th} character in S and $z = |S|$. As shown in Table 2C, the size of the compact index (in terms of the number of nodes stored) is significantly smaller than that of the regular index.

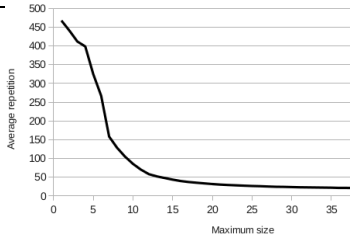
7.4 Query Processing Time

Figure 4A 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

Table 2. Subtree repetitions in experimental dataset and resulting index sizes

Size	Avg. 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

A



B

	Num. Nodes
Original index	19,775,322
Compact index	1,284,701

C

factor of 1.5, to about .8 seconds per query on average. (Note that accuracy is not affected by employing any of the optimization techniques.) Figure 4B compares 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.

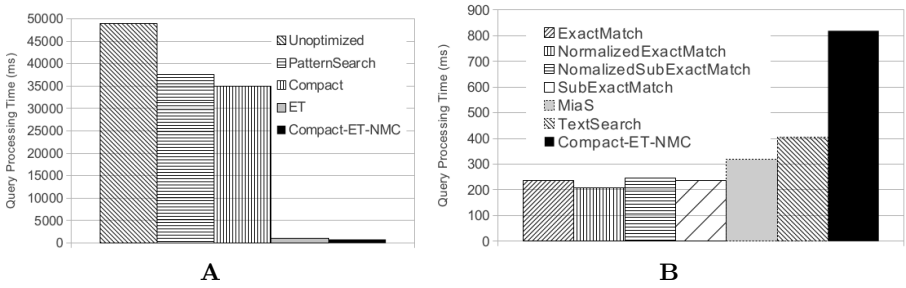


Fig. 4. The query processing time of alternative algorithms

The effect of the available memory on the query processing time is investigated in Fig. 5. 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.

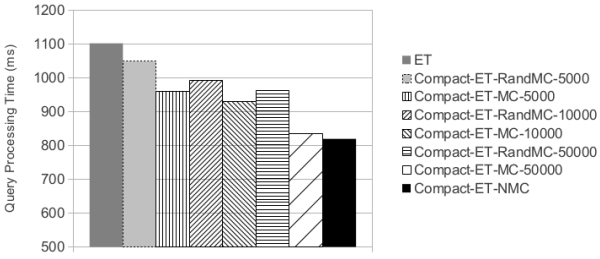


Fig. 5. The query processing time for various space budgets and cache strategies

For example, the performance of *Compact-ET-MC-50000* is very close to that of *Compact-ET-NMC*, which assumes unlimited memory is 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.

8 Conclusion

Mathematics retrieval is still in an early stage of development. We have shown that in order to correctly capture the relevance of math expressions, their structures must be considered. Tree edit distance, which is a standard technique to compare structures, is computationally expensive, but optimization techniques can reduce query processing time significantly. Through extensive experiments, we showed that our algorithm significantly outperforms baseline algorithms in terms of the accuracy of results while performing comparably in terms of query processing time even when memory is limited. Additional improvements should still be explored, however, to close the remaining performance gap.

References

1. `lucene.apache.org`
2. Bancerek, G.: Information retrieval and rendering with MML Query. In: Borwein, J.M., Farmer, W.M. (eds.) MKM 2006. LNCS (LNAI), vol. 4108, pp. 266–279. Springer, Heidelberg (2006)
3. Einwohner, T.H., Fateman, R.J.: Searching techniques for integral tables. In: ISSAC, pp. 133–139 (1995)
4. Grimm, J.: Tralics, A LaTeX to *XML* Translator. INRIA (2008)
5. Guidi, F., Schena, I.: A query language for a metadata framework about mathematical resources. In: Asperti, A., Buchberger, B., Davenport, J.H. (eds.) MKM 2003. LNCS, vol. 2594, pp. 105–118. Springer, Heidelberg (2003)
6. Kamali, S., Tompa, F.W.: Improving mathematics retrieval. In: DML, pp. 37–48 (2009)
7. Kamali, S., Tompa, F.W.: A new mathematics retrieval system. In: CIKM, pp. 1413–1416 (2010)
8. Kamali, S., Tompa, F.W.: Retrieving documents with mathematical content. In: SIGIR (2013)
9. Kohlhase, M., Sucan, I.: A search engine for mathematical formulae. In: Calmet, J., Ida, T., Wang, D. (eds.) AISC 2006. LNCS (LNAI), vol. 4120, pp. 241–253. Springer, Heidelberg (2006)
10. Laitang, C., Boughanem, M., Pinel-Sauvagnat, K.: XML information retrieval through tree edit distance and structural summaries. In: Salem, M.V.M., Shaalan, K., Oroumchian, F., Shakery, A., Khelalfa, H. (eds.) AIRS 2011. LNCS, vol. 7097, pp. 73–83. Springer, Heidelberg (2011)
11. Maclean, S., Labahn, G.: A new approach for recognizing handwritten mathematics using relational grammars and fuzzy sets. In: IJDAR, pp. 1–25 (2012)

12. Mišutka, J., Galamboš, L.: System description: Egomath2 as a tool for mathematical searching on wikipedia.org. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) Calculemus/MKM 2011. LNCS, vol. 6824, pp. 307–309. Springer, Heidelberg (2011)
13. Munavalli, R., Miner, R.: Mathfind: a math-aware search engine. In: SIGIR, pp. 735–735 (2006)
14. Nguyen, T.T., Chang, K., Hui, S.C.: A math-aware search engine for math question answering system. In: CIKM, pp. 724–733 (2012)
15. Schellenberg, T., Yuan, B., Zanibbi, R.: Layout-based substitution tree indexing and retrieval for mathematical expressions. In: DRR (2012)
16. Smirnova, E.S., Watt, S.M.: Communicating mathematics via pen-based interfaces. In: SYNASC, pp. 9–18 (2008)
17. Sojka, P., Líska, M.: The art of mathematics retrieval. In: ACM Symposium on Document Engineering, pp. 57–60 (2011)
18. Youssef, A.: Search of mathematical contents: Issues and methods. In: IASSE, pp. 100–105 (2005)
19. Youssef, A.S.: Methods of relevance ranking and hit-content generation in math search. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) MKM/CALCULEMUS 2007. LNCS (LNAI), vol. 4573, pp. 393–406. Springer, Heidelberg (2007)
20. Zanibbi, R., Yu, L.: Math spotting: Retrieving math in technical documents using handwritten query images. In: ICDAR, pp. 446–451 (2011)
21. Zanibbi, R., Yuan, B.: Keyword and image-based retrieval of mathematical expressions. In: DRR, pp. 1–10 (2011)
22. Zhang, K., Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. SIAM J. Comput. 18(6), 1245–1262 (1989)