PAPER Special Section on Data Engineering and Information Management

Efficient Algorithm for Math Formula Semantic Search

Shunsuke OHASHI[†], **Giovanni Yoko KRISTIANTO**[†a)], **Goran TOPIĆ**[††], ***Nonmembers***, ***and*** **Akiko AIZAWA**[†,††b)], ***Member***

SUMMARY Mathematical formulae play an important role in many scientific domains. Regardless of the importance of mathematical formula search, conventional keyword-based retrieval methods are not sufficient for searching mathematical formulae, which are structured as trees. The increasing number as well as the structural complexity of mathematical formulae in scientific articles lead to the necessity for large-scale structure-aware formula search techniques. In this paper, we formulate three types of measures that represent distinctive features of semantic similarity of math formulae, and develop efficient hash-based algorithms for the approximate calculation. Our experiments using NTCIR-11 Math-2 Task dataset, a large-scale test collection for math information retrieval with about 60-million formulae, show that the proposed method improves the search precision while also keeps the scalability and runtime efficiency high.

key words: *tree hashing, MathML, mathematical formula search, information retrieval*

1. Introduction

Mathematical formulae play an important role in many scientific domains. As a growing number of mathematical formulae becomes accessible on the Web, the necessity for mathematical information retrieval also increases. However, conventional web search engines are designed for natural language text and cannot properly handle mathematical formulae. Math formulae on the Web are commonly represented using MathML markup language which is an XML application specified by W3C. Existing LaTeX formulae can be converted into MathML using publicly available tools such as LaTeXML [1].

Since each MathML formula is represented as an XML tree structure, conventional methods for tree structure indexing, such as pq-gram [2], are applicable. However, the standard methods do not take into account features specific to mathematical formulae. For example, Cauchy–Schwarz inequality $|u \cdot v| \leq ||u||||v||$ has two variables u and v. We can also describe this inequality as $|x \cdot y| \leq ||x||||y||$; while variable names are different, the meaning of both representations is the same. However, most existing tree similarity search algorithms cannot capture this equivalence, because their similarity function is sensitive to any changes in subordinate nodes (including variable renaming). On the other hand, in certain formulae, some variable names implicitly convey the semantics of the variable. For example, variables used in physics often represent specific physical quantities. In $F = ma$, F, m, and a stand for force, mass, and acceleration, respectively. When F, m and a are subsituted with V, R and I, the resulting equation, $V = RI$ represents Ohm's law, completely different physical property. Therefore, math retrieval systems should also consider these variable-sensitive cases in their search operations.

When we read mathematical formulae, often we care most about the high-level structure, so that the nodes near the root of the tree are the same, while possibly ignoring substitutions that happen near to leaf level [3]. For example, the definition of spectral radius of a matrix, taken from NTCIR11-Math dataset [4], is $\rho(A) = \lim_{n\to\infty} \|A^n\|^{1/n}$, which can be viewed as $\rho(*) = \lim_{n\to\infty} \| *^n \|^{1/n}$, where $'*'$ can be an arbitrary formula itself. In order to deal with these cases, we need to index substructures of mathematical formulae that capture functions and their arguments structures.

The field of mathematical information retrieval is still in its early stage in analyzing users' information need. The complexity of similarity calculation in math formula search, which is basically similar tree-structure search, becomes higher for more complicated, and expectedly more effective similar measures. Therefore, effectiveness and efficiency of similarity calculation cannot be completely separated. Our basic assumption in this paper is that we first need efficient algorithms in order to further investigate the syntax-semantic similarity of math formulae. In this paper, we introduce three types of similarity measures to reflect math-specific users' information needs we mentioned above: *variable name*, *expression structure*, and *alpha-equivalence* measures. We also propose efficient algorithms to calculate the proposed measures. These algorithms perform similarity calculation in almost linear time to the input size. In our experiments, we show the efficiency and search accuracy of our method using NTCIR-11 Math-2 Task dataset, a large-scale test collection for math information retrieval with about 60-million formulae expressed in MathML.

Manuscript received July 1, 2015.
Manuscript revised November 9, 2015.
Manuscript publicized January 14, 2016.

[†]The authors are with the Department of Computer Science, The University of Tokyo, Tokyo, 113–0033 Japan.
[††]The authors are with the National Institute of Informatics, Tokyo, 101–8430 Japan.
a) E-mail: giovanni@nii.ac.jp
b) E-mail: aizawa@nii.ac.jp

DOI: 10.1587/transinf.2015DAP0023

2. Related Work

2.1 Mathematical Formula Search

Recent development of math search systems involves several techniques to process mathematical expressions prior to indexing step [5], [6]. These techniques [6] include segmentation, normalization (e.g. repairing broken MathML-representation of math expressions and removing MathML attributes), enrichment (e.g. automatic inference of metadata from context analysis), and approximation. Based on [5], the approximation technique may involve using thesaurus (i.e. adding synonyms for symbols to a query), canonical orderings, enumerating variables, replacing symbols with their types, and simplification. Furthermore, for indexing purpose, current math search systems apply structured-based indexing via substitution trees, reduction to full-text searches, reduction to SQL, or reduction to XML-based searches [5], [6].

There were several implementations of math search system published in NTCIR-11 Math-2 Task [4]. FSE [7] proposed 5 similarity measure factors and evaluated those factors individually to prove its relevance to similarity. They considered the taxonomy of functions and operators, data-type hierarchical level for mathematical expressions, the depth of matching position, query coverage, and the different importance between expression and formula (they regarded formula is more important than expression) as similarity measure factors. ICST [8] introduced semantic enrichment of mathematical formulae and used two types of formula tokenizer which is similar to our proposal. IFISB [9] extracted operators, constants, identifiers, and operators as a formula representation and nouns and verbs from the natural language sentences which are around the formula as the context representation. Those representations were indexed using Elasticsearch [10]. KWARC [11] presented the MathWebSearch system. It supported full-text search using Elasticsearch [10] and formula query using trie-like tree index, which is similar to the substitution tree indexing [12]. MCAT [13] decomposed the Presentation MathML tree into several paths from a root to a leaf. They expanded the keyword query using the dependency graph in which vertices represents each formula in the document. MIRMU [14] developed the search engine called Math Indexer and Searcher (MIaS) [15]. This MIaS system applied ordering, tokenization, variable unification, and constant unification to each math expression. This team achieved the best MAP score in NTCIR-11 Math task. RIT [16] handled the mathematical formula using Symbol Layout Tree, which represents the layout relations between symbols in the formula. They captured the structured of math expressions by generating symbol pair tuples from this layout tree. TUW-IMP [17] combined standard tokenizer of Lucene [18] for text search and the tokenizer which extracts all literal and sliced subtree in the formula tree.

According to Kamali et al. [19], mathematical formulae search can be categorized into two types: one based on exact match and the other on approximate match. The later case can be further grouped into three approaches: substructure match, structure similarity, and keyword similarity.

The substructure match supports both exact and partial matching. For a given query, a mathematical expression is retrieved by this approach if one of its subexpressions exactly matches the query. In the structure similarity approach, the SimSearch method proposed by [19] utilized a tree-edit distance based method to calculate the similarity of two mathematical expressions. An alternative for matching based on structural similarity is the PatternSearch algorithm [20], which specifies a template as the query and returned expressions that match the template as the search result. On the other hand, the keyword similarity approach considers each mathematical expression as a bag of symbols and functions, and completely ignores the structure.

Kamali et al. [19] also compared the performance and showed that SimSearch and PatternSearch perform better than other approaches. Comparing these two, PatternSearch performs slightly better than SimSearch, but requires longer execution time to process wildcard queries. Based on this, they concluded that structural similarity is the best approach for general cases, and pattern matching approach is advantageous when a user is experienced with a target domain. In this paper, we will focus on structural similarity.

2.2 Tree Structure Similarity

In this section, we briefly describe several studies about tree similarity measures. Tree kernel method [21] first generates a feature vector for each tree structure, where each element of the vector refers to the existence of a particular subtree component. Then, tree similarity is defined as the inner product of these vectors. An efficient dynamic programming based algorithm is applied for the calculation. Tree kernel is used in several fields including natural language processing [22] and bioinformatics [23].

The Tree Edit Distance (TED) [24] is an extension of the edit distance for strings [25] and is widely used as a tree similarity measure. In the same way as the edit distance for strings, TED allows three operations: insertion, deletion, and substitution of subtrees. The TED between two trees is defined as the minimum possible number of edit operations to transform. TEDs can be flexibly adapted to different domains by adjusting the costs for editing operations. However, the calculation of TED requires $O(n^3)$ where n is the number of nodes in the tree [26], and the complexity makes it infeasible directly apply TED to large-scale math formula search.

In order to overcome the computation cost problem of TED, several approximation methods have been formulated. Yang et al. [27] proposed a binary branch distance where a triple, consists of a node, the leftmost child, and the right sibling of the node, is first generated for each node on the tree. Then, the generated triples are converted into a vector representation of the tree whose L_1 distance gives the lower

bound to the TED. The time complexity is $O(|T_1| + |T_2|)$, where $|T_i|$ denotes the number of nodes in the tree T_i. Augsten et al. proposed pq-gram [2] where substructures of an input tree are extracted and used as a feature set. The Jaccard similarity of the feature set gives the lower bound of the weight-modified TED. The time complexity of feature set generation and Jaccard similarity calculation is $O(n)$ and $O(n \log n)$, respectively.

MinHash [28] is a randomized algorithm to speed up the Jaccard similarity calculation by preprocessing. MinHash reduces both the size of the memory and the time complexity of the similarity calculation. Yuan et al. [29] applied MinHash to XML document retrieval, which can be easily extended to general tree similarity search.

3. Proposed Method

3.1 Overview

Our formulation in this paper is based on a simple search model where a system retrieves similar math formulae to a given math formula query and returns a ranked list based on the similarity score. We assume that the math formulae are represented using MathML Presentation and/or Content Markup languages. MathML has an annotation tag for variables: *mi* in Presentation MathML, and *ci* in Content MathML. As described herein, we call a node a variable node if the parent node of the node is *ci* or *mi*.

In our framework, all math formulae are first converted to fixed size binary vectors, which are then used for similarity calculation. An overview of the procedure is shown in Fig. 1. The procedure method consists of three steps: First, an XML parser is applied to the input MathML representation. We consider the resulting tree strucuture as a rooted ordered labeled tree, a tree with a root node where each node has an ordered list of its children, and is labeled with a string. Second, subtree structures are extracted using three types of algorithms we define in Sect. 3.2. Third, randomly generated N minhash functions are applied to the obtained subtree set and the returned values are stored in an inverted index to speed up the similarity calculation [28].

3.2 Semantic Similarity Measures for Math Formulae

In this paper, we introduce three types of similarity measures to capture math-specific semantic similarity. These measures are based on substructure matching. In substructure matching, an input tree is converted into a set of substructures. We call this set a *feature set* of a tree. The tree similarity is defined as the set similarity of the feature sets. As is already mentioned in Sect. 3.1, we used Jaccard coefficient for the set similarity.

- Variable Name Measure
  The goal of the variable name measure is to capture the implicit semantics represented by variable names. The feature set consists of all the subtrees of the input tree. All variable names are treated as a label of the corresponding node.
- Function Structure Measure
  The goal of the function structure measure is to capture the semantics of structures or patterns in the formulae. For instance, the structure of $y = x^2$ indicates a quadratic equation, which in this example the left-hand side variable (y) is defined as the square of another variable (x). Thus, based on this measure, both $y = x^2$ and $b = a^2$ have the same semantic. The feature set of this measure consists of all the substructures of a specified depth d rooted on any node of the tree. When $d = 2$, this measure is equivalent to the generalized term used in [3].
- Alpha-Equivalence Measure
  The goal of the alpha-equivalence measure is to provide a metric which is invariant to variable names. Alpha-equivalence is the equivalence relation used in lambda calculus or programming language theory. In math formula, alpha equivalent transformation corresponds to renaming of variables. For an example, this measure can capture the mathematician's intuition that $x = x$ has fundamentally a similar meaning to $y = y$, but not to $x = y$, for any value of x and y. The feature set consists of subtrees of the input tree where all the variables are renamed according to their appearance order.

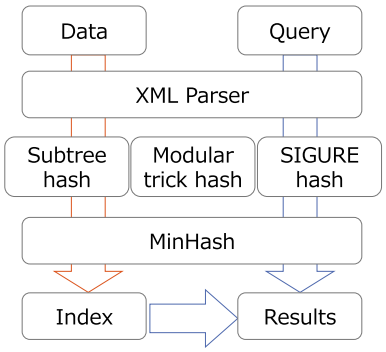


Fig. 1 Overview of the procedure.

Mathematical formula is an abstract representation whose semantics has syntax-semantic duality. MathML markup language is designed to express both the syntax and semantic structures. The later, the semantic structure representation, is closely related to the lambda calculus, one of the well-known formal models of computing. In fact, well-written MathML formulae can be directly executed by Mathematica[†]. The three types of similarity measures considered in our paper, variables, expression structure, and alpha-equivalence, are not ad hoc, but correspond to the variables, function application, and alpha equivalence of the lambda calculus. Since mathematical formulae are widely used in many scientific domains, mathematical formula search is not only for mathematicians. Users often do

[†]https://www.wolfram.com/mathematica/

not have clear distinction between syntax and semantics, and search semantically similar formula using syntactic similarity. Therefore, we used our algorithms to calculate the syntax similarity as well. Although we haven't systematically analyzed mathematicians perception of similarity in this paper, we expect that some of the similarity issues can be handled by the formal semantic model. In the next section, we present three hash functions, *subtree hash*, *modular trick*, and *SIGRE hash*, corresponding to variable name, function structure, and alpha-equivalence measures, respectively.

4. Algorithm

4.1 Subtree Hash

Subtree Hash is a hash function that takes a tree as an input and returns a feature set. The pseudocode for Subtree Hash is shown in Algorithm 1. This algorithm is intuitively a depth-first pre-order traversal to construct hash codes, visiting each node once, and at each node having to combine (already computed) hash codes from children. The input for this algorithm is a root node of a tree, which is represented by n in the pseudocode. The main part of this algorithm is implemented in $STHrec$ function. This function returns not only x, the hash value corresponding to the subtree rooted in n, but also H, the set of hash values which correspond to all the subtrees included in the tree rooted at n (including x). If n is a leaf, $STHrec$ function returns a, the hash value of the label of n, and $\{a\}$, the set which contains a as its only element, as x and H respectively (from l.5 to l.8). If n has any child nodes, the set of child nodes is used as an argument for the recursive call to the $STHrec$ function (from l.12 to l.13). Each returned x is kept in the array X. This array is used to calculate the hash value of the tree rooted at n (l.14, l.16). To compute the hash value, Rolling Hash algorithm [30] is used (from l.18 to l.22). a is used as the parameter for Rolling Hash algorithm. H is the union of all H returned from recursive calls and x, the hash value which corresponds to the tree rooted at n (l.17). Time complexity of this algorithm is $O(N)$, where N denotes the number of nodes included in the input tree. For each node in the tree, $STHrec$ function is called exactly once, and its amortized complexity [31] is $O(1)$. The heaviest process in this function is calling the *RollingHash* function, and its time complexity is $O(L)$, where L denotes the length of *array*. However, the sum of the length of *array* used in whole process of Subtree Hash is equal to $N-1$, because each hash value corresponding to a subtree appears exactly once in *array*, except for the hash value of the whole tree. Therefore, amortized complexity (i.e. the total expense divided by the number of invocations) of *RollingHash* is $O(1)$, and the time complexity of $SubtreeHash$ is $O(N)$.

We give a detailed example of how Subtree Hash works. Let us consider that the input formula is $g(f(x,y),f(y,z))$ and it is converted into the tree shown in Fig. 2, and we assume that $p = 11$, $hash(x) = 5$, $hash(y) = 6$, $hash(f) = 3$, and $hash(g) = 4$. For example, let us consider the left subtree rooted in a node with label f. Hash value of this node is calculated as follows: $(3^1 \cdot 5 + 3^0 \cdot 6)\%11 = 10$ using *RollingHash* function. Right side of the node with label f is processed in the same way, resulting in 3. As for the root node, the hash value is calculated as $(4^1 \cdot 10 + 4^0 \cdot 3)\%11 = 10$. Note that the hash value is different if the original subtree is different, therefore this hash value represents a subtree.

Algorithm 1 SubtreeHash(n)

```
1:  function SubtreeHash(n)
2:      x, H ← STHrec(n)
3:      return H

4:  function STHrec(n)
5:      C ← {c | c is child of n}
6:      a ← hash(label(n))%p
7:      if |C| = 0 then
8:          return (a, {a})
9:      else
10:         X ← newVector()
11:         H ← {}
12:         for c in C do
13:             (x_sub, H_sub) ← STHrec(c)
14:             X.append(x_sub)
15:             H ← H ∪ H_sub
16:         x ← RollingHash(a, X)
17:         return (x, H ∪ {x})

18: function RollingHash(a, array)
19:     x ← 0
20:     for h in array do
21:         x ← (x * a + h)%p
22:     return x
```

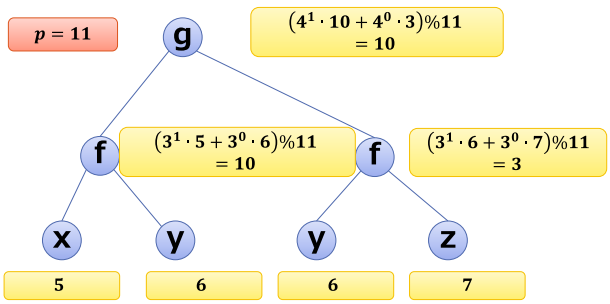


Fig. 2 Example of subtree hash.

4.2 Modular Trick

Modular Trick is a tree similarity metric designed to capture the similarity of high level hierarchy, as we mentioned as function structure measure in Sect. 3.

This algorithm is obtained by slightly modifying the algorithm of Subtree Hash. Its pseudocode is shown in Algorithm 2.

In this algorithm, only *RollingHash* function part is modified from Subtree Hash. The modified lines are l.21 and l.22. A new parameter b is added to the algorithm. The $MTrec$ function is identical to the $STHrec$ function defined

Algorithm 2 ModularTrick(n)

```
1: function ModularTrick(n)
2:     x, H ← MTrec(n)
3:     return H

4: function MTrec(n)
5:     C ← {c | c is child of n}
6:     a ← hash(label(n))%p
7:     if |C| = 0 then
8:         return (a, {a})
9:     else
10:        X ← newVector()
11:        H ← {}
12:        for c in C do
13:            (x_sub, H_sub) ← MTrec(c)
14:            X.append(x_sub)
15:            H ← H ∪ H_sub
16:        x ← RollingHash(a, X)
17:        return (x, H ∪ {x})

18: function RollingHash(a, array)
19:     x ← 0
20:     for h in array do
21:         x ← (x ∗ (a|1) + h)%p
22:     return (x ∗ b + a)%p
```

in Algorithm 1, except that it calls the modified *RollingHash* function. Using this algorithm, we can extract the subtree which consists of a node and its descendants with distance smaller than or equal to $\log_p b$ if we specify parameters b and p appropriately.

This theorem is the key for this algorithm.

Theorem 1: Let $MTHash(t)$ be the first element of $MTrec(t)$. If $b^s\%p = 0$, $MTHash(t)$ is invariant under modification to any descendant d such that the distance between d and t is greater than or equal to s.

Theorem 1 expresses that we can hash a subtree to a desired depth s by choosing b and p such that $b^s\%p = 0$ holds. We can prove this theorem by analyzing how a descendant d affects the value $MTHash(t)$. We can describe $MTHash(t)$ mathematically as follows.

$$MTHash(t) = (b\sum_{i=1}^{|C|}(a|1)^{i-1}h_i + a)\%p \quad (1)$$

By recursively applying this formula to h_i that is an ancestor of the node d, we obtain this equation, where $dist(t,d)$ is the distance between t and d.

$$\begin{aligned} MTHash(t) &= b(\ldots + b(\ldots(\ldots + MTHash(d))) \\ &\quad \ldots + a)\%p \\ &= (\ldots + b^{dist(t,d)}MTHash(t))\%p \end{aligned}$$

If $b^{dist(t,d)}\%p = 0$, $MTHash(d)$ and d are eliminated, and have no effect on $MTHash(t)$. QED.

As mentioned above and discussed in detail in Sect. Appendix, p and the Rolling Hash parameter b should optimally be co-prime. Given the above, this algorithm works well with the following parameter values:

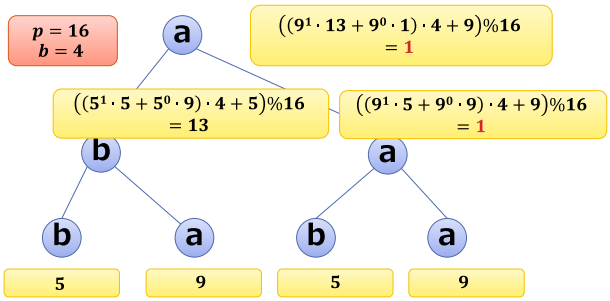


Fig. 3 Example of modular trick.

$$\begin{aligned} p &= 2^{64} \\ b &= 2^{32}, 2^{22}, 2^{16}, \ldots, 1, 0 \end{aligned}$$

Different b result in different subtree pruning depths. For example, when b is set to 2^{22}, the subtrees will be hashed to the depth of 3 (i.e. nodes at depth greater than $\log_{2^{22}} 2^{64} = 2.90909\ldots$ would be pruned away). The parameter of Rolling Hash in Algorithm 2 is forced to be odd in l.21, using the | operator (bit-OR). Therefore, p and the parameter are always co-prime when using $p = 2^{64}$.

This algorithm is a generalization of "generalized term" proposed in [3] and Subtree Hash. It is equivalent to generalized term or Subtree Hash when b is set to 2^{32} or 1, respectively.

We give a detailed example of how Modular Trick works. Let us consider that the input formula is $a(b(b,a),a(b,a))$ and it is converted into the tree shown in Fig. 3, and we assume that $b = 2^2$, $p = 2^4$, $hash(a) = 9$ and $hash(b) = 5$. For example, let us consider the left subtree under the root, rooted in the node labeled b. The hash value of this node is calculated as follows: $((5^1 \cdot 5 + 5^0 \cdot 9) \cdot 4 + 5)\%16 = 13$ using modified *RollingHash* function. The right subtree rooted in a is processed in same way, resulting in 1. As for the root node, its hash value is calculated as $((9^1 \cdot 13 + 9^0 \cdot 1) \cdot 4 + 9)\%16$. Note that the hash value of the root node and the a node in the second level are equal, since no descendants at depth $log_{2^2}2^4 = 2$ or deeper contribute to the hash.

4.3 SIGURE Hash

SIGURE Hash[†] is the algorithm for the alpha-equivalence similarity measure. The pseudocode is shown in Algorithm 3, 4. We also provided an example of SIGURE Hash calculation in Fig. 4.

This algorithm splits the hash value into two parts, V and X. V manages the variable names and their order of appearance. X represents the hash value in polynomial form, consisting of a vector of polynomial coefficients X_v for each variable v in V, and a constant number c. When X and V are fixed, the hash value is determined uniquely by evaluating

[†]Structural Information Greedy Unify REcursive Hash. Pronounced the same way as the Japanese word 時雨 (shigure, drizzling rain)

Algorithm 3 SIGUREHash(n)

Require: n : root node of the tree

1: **function** SIGUREHash(n)
2: $(X, V, H) \leftarrow SHrec(n)$
3: **return** H
4: **function** SHrec(n)
5: $V \leftarrow newHashMap()$
6: $X_v \leftarrow newHashMap()$
7: **if** $isVariableNode(n)$ **then**
8: $V[label(n)] \leftarrow length(V)$
9: $X_v[label(n)] \leftarrow 1$
10: $X \leftarrow (X_v, 0)$
11: **return** $(X, V, \{Eval(X, V)\})$
12: $C \leftarrow \{c \mid c \text{ is child of } n\}$
13: $a \leftarrow hash(label(n))$
14: **if** $|C| = 0$ **then**
15: $X \leftarrow (X_v, a)$
16: **return** $(X, V, \{Eval(X, V)\})$
17: **else**
18: $X \leftarrow (X_v, 0)$
19: $H \leftarrow \{\}$
20: **for** c in C **do** ▷ Traverse from left child.
21: $(X_{sub}, V_{sub}, H_{sub}) \leftarrow SHrec(c)$
22: $X, V \leftarrow Merge(X, V, X_{sub}, V_{sub}, a)$
23: $H \leftarrow H \cup H_{sub}$
24: **return** $(X, V, H \cup \{Eval(X, V)\})$

Algorithm 4 Auxiliary function used in SIGUREHash

1: **function** Eval(X, V)
2: $(X_v, c) \leftarrow X$
3: $h \leftarrow c$
4: **for** v in $key(V)$ **do**
5: $h \leftarrow h + X_v[v] * hash(V[v])$
6: **return** h
7: **function** Merge($X, V, X_{sub}, V_{sub}, a$)
8: $X_{v,ret} \leftarrow newHashMap()$
9: $V_{ret} \leftarrow newHashMap()$
10: $X_v, c \leftarrow X$
11: $X_{v,sub}, c_{sub} \leftarrow X_{sub}$
12: **for** v in $key(V)$ **do** ▷ Traverse from the key which has the smallest value.
13: $V_{ret}[v] \leftarrow length(V_{ret})$
14: $X_{v,ret}[v] \leftarrow X_v[v] * a$
15: **for** v in $key(V_{sub})$ **do** ▷ Traverse from the key which has the smallest value.
16: **if** v not in $key(V)$ **then**
17: $V_{ret}[v] \leftarrow length(V_{ret})$
18: $X_{v,ret}[v] \leftarrow 0$
19: $X_{v,ret}[v] \leftarrow X_{v,ret}[v] + X_{v,sub}[v]$
20: $c_{ret} \leftarrow c * a + c_{sub}$
21: **return** $((X_{v,ret}, c_{ret}), V_{ret})$

the polynomial using variable values from V. This calculation is done in the *Eval* function (l.5 in Algorithm 4).

The main part of this algorithm is implemented in the *SHrec* function. This function returns not only H, set of hash values which corresponds to all the subtrees included in the tree rooted at n (including itself), but also X and V as explained above. If the input n is a leaf node containing a variable, V will have a only one element, $label(n)$ as key and 0 as value, which means that label occurs 0-th (0 index) in the subtree, and X_v will also have only one element, which means the polynomial consists of a single term $1 \cdot hash(0)$ (from l.7 to l.11 in Algorithm 3). If n is not variable node but is leaf node, V and X_v are empty, and the constant a is set as the hash value of the label of the node n (from l.14 to l.17 in Algorithm 3). If n has child nodes, the hash value is calculated in the same way as Subtree Hash. However, the recursive call does not return a hash value, but the (X, V) tuple; therefore, we have to modify the Rolling Hash process. The modified process is implemented as the *Merge* function. This function corresponds to the operation of the single step in Rolling Hash. It takes the hash of the label of n, and two (X, V) tuples. The first tuple, (X, V), is the accumulated result of already traversed subtrees, while the second one, $(X_{sub}.V_{sub})$ is the result of the subtree which is being merged. Merging X is rather simple, because we just treat it as a set of polynomial coefficients (l.14 and l.19 in Algorithm 4). However, in order to merge V, we have to consider the traversal order. All variables which appear in V are earlier in preorder than the variables in V_{sub} because *SHrec* function traverses the tree in preorder and V is generated earlier than V_{sub}. Therefore, what we have to do to merge V is to add the variables which occur in V_{sub} but do not occur in V to V while preserving their order of appearance in V_{sub} (l.13 and l.17 in Algorithm 4). As you can see in Algorithm 4, there are only two types of the operation applied to a hash value, multiplication by constant a and addition. This means the hash value polynomial is always first-order, and therefore $|V| <= |X|$ always holds. The time complexity of this algorithm is $O(NK)$ where N is the size of the input tree and K is the number of unique variable names in the tree. Both *Eval* and *Merge* function are called at most N times. The time complexity of *Eval* and *Merge* are both $O(K)$, because of those for-loops. Therefore entire time complexity is $O(N)K$. K could be $O(N)$ in artificial input and it means time complexity could be $O(N^2)$. However, this algorithm works on almost linear time to the size of input because K is much smaller than N in real-data.

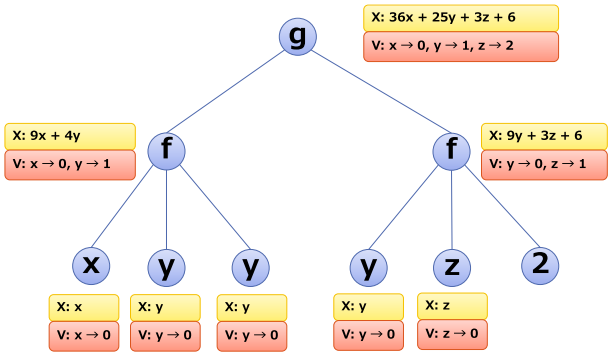


Fig. 4 Example of SIGURE hash.

We give a detailed example of how SIGURE Hash works. Let us consider that the input formula is $g(f(x, y, y), f(y, z, 2))$ and it is converted into the tree shown in Fig. 4 with all leaf nodes marked as variable nodes, and we assume that $hash(f) = 3$, $hash(g) = 4$ and $hash(2) = 6$. For example, let us consider the left subtree, rooted in the

node labeled f. The hash polynomial of this node is calculated as $3^2x + 3^1y + 3^0y = 9x + 4y$ using the *Merge* function, with the variable order being $V = \{x \to 0, y \to 1\}$. The right subtree rooted in another node labeled f is processed as follows: $3^2y + 3^1z + 3^0 \cdot 6 = 9y + 3z + 6$ ($V = \{y \to 0, z \to 1\}$). As for the top node, the hash polynomial is calculated as $4^1(9x + 4y) + 4^0(9y + 3z + 6) = 36x + 25y + 3z + 6$, with V merging to $\{x \to 0, y \to 1, z \to 2\}$. The actual hash value, which does not contain variables, is calculated by substituting the hash of the values from V into the polynomial.

5. Experiments

5.1 Experimental Setup

The experiments conducted in this paper used a dataset released by NTCIR-11 Math-2 [4]. The dataset contains 105,120 scientific papers that are segmented into 8,301,578 search units with about 60 million formulae. The size of the document collection is about 180 GB. The dataset also contains 50 queries, each of which consists of mathematical formulae and natural language keywords. For each query, 50 relevance judgment results are also provided.

In this experiment, we investigated the combination of the three algorithms proposed in the previous section: Subtree Hash, Modular Trick, and SIGURE Hash. We used $b = 2^{32}, 2^{22}, 2^{16}$ for Modular Trick. As a baseline, we considered the pq-gram similarity method [2] where $p = q = 2$. All search methods are implemented using Python 2.7 and used the same XML parser and MinHash module. The number of MinHash function is set to be 30. The computation resource used was four Intel Xeon E7-4870 @2.40GHz cores and 1TB RAM. The query-formula similarity calculation in all search methods was parallelized into 32 processes.

5.2 Retrieval Performance

In the experiment, top 1,000 retrieval units were first obtained for each algorithm for each given query. Next, the retrieval performance was calculated considering only the retrieval units with relevance judgment. We used the Preicision-at-5 (P'@5), Precision-at-10 (P'@10), and Mean Average Precision (MAP') at maximum depth of 50 as the metrics. These metrics are widely used for evaluating information retrieval systems [32].

Table 1 shows the result. Although the performance of Subtree Hash, Modular Trick, and SIGURE Hash did not exceed the one of pq-gram, the combined method outperforms pq-gram in all metrics, i.e., P'@5, P'@10, and MAP'. We will later analyze the characteristics of formulae retrieved by each algorithm.

Among the Modular Trick runs, Modular Trick with $b = 2^{32}$, i.e. subtree depth is 2, achieved the best P'@5, P'@10, and MAP'. By definition, Subtree Hash is equivalent to Modular Trick with $b = 2^0 (= 1)$, and as the depth of subtree becomes larger, the performance of Modular Trick converges to the performance of Subtree Hash.

Table 1 Comparison of retrieval performance.

Algorithm	*P'@5*	*P'@10*	*MAP'*
Subtree	0.3320	0.2220	0.1670
Mod. Trick ($b = 2^{32}$)	0.3560	0.2680	0.2253
Mod. Trick ($b = 2^{22}$)	0.3160	0.2260	0.1992
Mod. Trick ($b = 2^{16}$)	0.3480	0.2300	0.1764
SIGURE	0.2680	0.1960	0.1515
Combined	0.3560	0.2800*	0.1936
pq-gram	0.3080	0.2140	0.1714

*statistically significant ($p < 0.05$ in two-tailed t-test) compared to pq-gram.

5.3 Query-Wise Comparison

In this section, we analyze the performance of our algorithms over several queries to find their strengths and weaknesses. Table 2 illustrates the precision comparison of our algorithms over 6 queries. The first two queries are the queries in which Subtree Hash has better MAP' than SIGURE Hash. The next two queries are the queries in which Modular Trick has better MAP' than Subtree Hash. The remaining two queries are the queries in which SIGURE Hash has better MAP' than Subtree Hash.

Let us consider the first two query formulae, in which Subtree Hash performs better than SIGURE Hash. The first formula is found in physics article, and the second in mathematics. The variable names ϕ and $\widehat{\mathrm{CH}}$ are often used to represent specific concepts, i.e. quantum field and arithmetic Chow group, respectively. As a consequence, the meaning of each formula strongly depends on these variable names. Substituting these variables with arbitratry variables will alter the meaning of the formulae. From the query performance, we can see that Subtree Hash can capture the meanings of the variable names very well.

Next, let us examine the results of Modular Trick algorithm in the next two queries. In these queries, Modular Trick significantly outperforms the other two algorithms. Variable names in those queries are almost meaningless. Furthermore, the relations between variables are not described in those formulae. However, their high level structure has significant meanings. For example, the expression which represents the definition of the spectral radius on matrices has the high level structure that represents ρ as a function defined by the limiting value. It is difficult for Subtree Hash or SIGURE Hash to capture this type of semantic.

In the last two query formulae in which SIGURE Hash delivers the best MAP', each variable name does not represent any specific entity. However, the relation of variables is essential for these formulae. For example, let us consider the formula that appears in the definition of Cauchy Schwarz inequality. It is difficult for Subtree Hash to retrieve this formula, because the relevant formulae may have the variable names u and v changed. Modular Trick also encounter the same difficulty, because the top level structure of this formula (i.e. two subexpressions connected with "greater than or equal to" symbol) is a quite common structure. On the

Table 2 Examples of query-wise MAP' performance.

Query Formula	*Keywords*	*Subtree*	*Mod. Trick ($b = 2^{32}$)*	*SIGURE*
ϕ^4	Quantum, Field, Theory	0.8218	0.5983	0.0208
$\widehat{\mathrm{CH}}^p(A) \cong Y$	arithmetic, Chow, group	0.4354	0.0500	0.1039
$\rho(A) = \lim_{n\to\infty} \|A^n\|^{1/n}$	spectral radius, matrix	0.5714	0.7857	0.6429
$x^n + y^n = z^n, x, y, z, n \in \mathbb{N}$	Diophantine equations	0.0000	0.8750	0.0000
$\|u \cdot v\| \leq \|\|u\|\|\|v\|\|$	Cauchy, Schwarz	0.2500	0.0000	0.4167
$\|x + y\|_p \leq \|x\|_p + \|y\|_p$	minkowski, inequality	0.3077	0.3372	0.5307

other hand, SIGURE Hash can retrieve it because it drops the variable name noise. As we have shown above, each of three algorithms we developed has its own strengths and weaknesses.

5.4 Processing Time

In the experiments using NTCIR-11 Math-2 dataset, the index construction time was 25,223 seconds for the proposed method (i.e. the combination of Subtree Hash, Modular Trick, and SIGURE Hash) and 112,388 seconds for the pq-gram. The difference reflects that for an input tree with N nodes, Subtree Hash, Modular Trick, and SIGURE Hash generate a feature set with N elements while the pq-gram generates a feature set with $(p + q)N$ elements. As for the average query response time, the combined method and pq-gram took 6.91 and 3.68 seconds, respectively. The combined method took longer time because it has larger index size and consequently longer index scan time. Our method produced many terms that represent short subexpressions, such as notation "x" (which quite often appears in math formulae). Therefore, if there is a query which contains term(s) matching such short subexpressions, our method have to retrieve and score a very long document list that contains these subexpressions. On the other hand, pq-gram does not suffer from this issue, since its produced terms vary by the surrounding subexpressions.

5.5 Retrieval Performance Using Wikipedia Dataset

Up to this point, we used NTCIR-11 Math-2 Main Task dataset for evaluation. It is large enough to prove the scalability of our method, but it is not fair to use this dataset to compare our method to other participants in the main task. Our proposed method focuses on the formula search while this main task allowed participants to exploit the keyword query, which drastically improved their accuracy. In this section, we report the evaluation result of our system in the NTCIR-11 Math-2 Wikipedia subtask [33]. This subtask generates formula queries in three steps: choosing seed formula randomly from the dataset, injecting the query variables into the formula, and finally generating the XML topics using LaTeXML [1]. In total, there are 100 queries generated in this subtask. In addition, there are two types of evaluation conducted: page-centric evaluation, which regards a hit as correct if the seeding page was found, and formula-centric evaluation, where a hit is correct if a formula with exactly the same TEX input was found. Furthermore, there are two metrics for each evaluation type: the percentage of seed formulae found (recall over all hit, top-∞) and the mean reciprocal rank (MRR). Table 3 shows the result of all participants in Wikipedia subtask. In this task, our team and TUW were the best performer with regard to the number of found seeds. However, the MRR of our system was considered moderate. In page-centric evaluation, both team had 97% recall, but TUW achieved slightly higher MRR (82%) than our team (76%). In formula-centric evaluation, our team delivered 94% recall with $MRR = 82\%$, while TUW gave 93% recall with $MRR = 88\%$. After the NTCIR-11, the updated Tangent system [34] from RIT improved its performance. It achieved perfect recall in page-centric evaluation, but still gave under 90% recall in formula-centric.

Table 3 Comparison of retrieval performance using Wikipedia dataset.

Team Name	*Page Centric*		*Formula Centric*	
	Recall	*MRR*	*Recall*	*MRR*
NII (our team)	97	76	94	82
TUB	91	73	87	68
KWARC	75	82	-	-
RHMS	48	2	-	-
RIT(Tangent-2)	88	80	78	86
MIaS	65	76	63	81
TUW	97	82	93	88
Post NTCIR-11				
RIT (Tangent-3)	100	83	89	85

5.6 Discussion

In our current implementation, we combined the proposed three algorithms simply by taking the join of the Minhash values. As a consequence, all feature elements are considered equally important. However, the discrimination power may differ much: For example, x is less important than $\frac{P(Y|X)P(X)}{P(Y)}$ and should be weighted less in Subtree Hash. For this purpose, we can use the conventional term weighting methods used in full text search. Also, improving the unification ability will increase the recall of SIGURE Hash. In our current formulation, SIGURE Hash can handle only variable unification. Since it does not support subexpression unification, $(x_1 - y_1)^2 + (x_2 - y_2)^2 = d^2$ cannot be retrieved when $a^2 + b^2 = c^2$ is given as the query. Since such term to variable substitution is commonly observed in scientific literature, significant portion of possibly similar formulae may be overlooked.

6. Conclusion

In this paper, we first defined three different similarity measures for math formula search and developed an efficient semantic search method with time complexity almost linear to the input size. In the evaluation using NTCIR-11 Math-2 dataset, our algorithm outperformed pq-gram, a commonly used state-of-the-art tree similarity search algorithm. Further analysis of individual query result showed that the proposed similarity measures capture different characteristics of the semantic similarities.

Future work includes enhancing the current method with weighted indexing and subexpression unification to improve the search performance. Another obvious direction is integrating the proposed method with conventional keyword-based search to exploit natural language keywords in the queries. Exploring their potential ability in other tasks of tree similarity search will be another possible future work as well.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Number 24300062.

References

[1] N.I. of Standards and Technology, "Latexml: A latex to xml converter." http://dlmf.nist.gov/LaTeXML/
[2] N. Augsten, M. Böhlen, and J. Gamper, "The pq-gram distance between ordered labeled trees," ACM Transactions on Database Systems (TODS), vol.35, no.1, pp.1–36, 2010.
[3] X. Lin, L. Gao, X. Hu, Z. Tang, Y. Xiao, and X. Liu, "A mathematics retrieval system for formulae in layout presentations," Proc. 37th international ACM SIGIR conference on Research and development in information retrieval, pp.697–706, ACM, 2014.
[4] A. Aizawa, M. Kohlhase, I. Ounis, and M. Schubotz, "Ntcir-11 math-2 task overview," Proc. NTCIR-11 Math-2 task Workshop Meeting, pp.88–98, 2014.
[5] R. Zanibbi and D. Blostein, "Recognition and retrieval of mathematical expressions," International Journal on Document Analysis and Recognition, vol.15, no.4, pp.331–357, 2012.
[6] F. Guidi and C.S. Coen, "A survey on retrieval of mathematical knowledge," Proc. 8th Conference on Intelligent Computer Mathematics, 2015.
[7] M. Schubotz, A. Youssef, V. Markl, H.S. Cohl, and J.J. Li, "Evaluation of similarity-measure factors for formulae based on the ntcir-11 math task," Proc. 11th NTCIR Conference on Evaluation of Information Access Technologies, pp.108–113, 2014.
[8] L. Gao, Y. Wang, L. Hao, and Z. Tang, "Icst math retrieval system for ntcir-11 math-2 task," Proc. 11th NTCIR Conference on Evaluation of Information Access Technologies, pp.99–102, 2014.
[9] J.M.G. Pinto, S. Barthel, and W.T. Balke, "Qualibeta at the ntcir-11 math 2 task: An attempt to query math collections," Proc. 11th NTCIR Conference on Evaluation of Information Access Technologies, pp.103–107, 2014.
[10] "Elasticsearch." http://www.elasticsearch.org/
[11] R. Hambasan, M. Kohlhase, and C. Prodescu, "Mathwebsearch at ntcir-11," Proc. 11th NTCIR Conference on Evaluation of Information Access Technologies, pp.114–119, 2014.
[12] P. Graf, "Substitution tree indexing," Rewriting Techniques and Applications, pp.117–131, Springer, 1995.
[13] G.Y. Kristianto, G. Topić, F. Ho, and A. Aizawa, "The mcat math retrieval system for ntcir-11 math track," Proc. 11th NTCIR Conference on Evaluation of Information Access Technologies, pp.120–126, 2014.
[14] M. Ružicka, P. Sojka, and M. Líška, "Math indexer and searcher under the hood: History and development of a winning strategy," Proc. 11th NTCIR Conference on Evaluation of Information Access Technologies, pp.127–134, 2014.
[15] P. Sojka and M. Líška, "The art of mathematics retrieval," Proc. 11th ACM symposium on Document engineering, pp.57–60, ACM, 2011.
[16] N. Pattaniyil and R. Zanibbi, "Combining tf-idf text retrieval with an inverted index over symbol pairs in math expressions: The tangent math search engine at ntcir 2014," Proc. 11th NTCIR Conference on Evaluation of Information Access Technologies, pp.135–142, 2014.
[17] A. Lipani, L. Andersson, F. Piroi, M. Lupu, and A. Hanbury, "Tuw-imp at the ntcir-11 math-2," Proc. 11th NTCIR Conference on Evaluation of Information Access Technologies, pp.143–146, 2014.
[18] "Apache lucene." http://lucene.apache.org/
[19] S. Kamali and F.W. Tompa, "Retrieving documents with mathematical content," Proc. 36th international ACM SIGIR conference on Research and development in information retrieval, pp.353–362, ACM, 2013.
[20] S. Kamali and F.W. Tompa, "A new mathematics retrieval system," Proc. 2010 ACM CIKM, pp.1413—1416, ACM, 2010.
[21] M. Collins and N. Duffy, "Convolution kernels for natural language," Advances in Neural Information Processing Systems, pp.625–632, 2001.
[22] A. Culotta and J. Sorensen, "Dependency tree kernels for relation extraction," Proc. 42nd Annual Meeting on Association for Computational Linguistics, p.423, Association for Computational Linguistics, 2004.
[23] J.-P. Vert, "A tree kernel to analyse phylogenetic profiles," Bioinformatics, vol.18, no.suppl 1, pp.S276–S284, 2002.
[24] K.-C. Tai, "The tree-to-tree correction problem," Journal of the ACM (JACM), vol.26, no.3, pp.422–433, 1979.
[25] V.I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," Soviet Physics Doklady, p.707, 1966.
[26] M. Pawlik and N. Augsten, "RTED: a robust algorithm for the tree edit distance," Proc. VLDB Endowment, vol.5, no.4, pp.334–345, 2011.
[27] R. Yang, P. Kalnis, and A.K. Tung, "Similarity evaluation on tree-structured data," Proc. 2005 ACM SIGMOD international conference on Management of data, pp.754–765, ACM, 2005.
[28] A.Z. Broder, "On the resemblance and containment of documents," Compression and Complexity of Sequences 1997. Proceedings, pp.21–29, IEEE, 1997.
[29] P. Yuan, C. Sha, X. Wang, B. Yang, A. Zhou, and S. Yang, "Xml structural similarity search using mapreduce," in Web-Age Information Management, pp.169–181, Springer, 2010.
[30] R.M. Karp and M.O. Rabin, "Efficient randomized pattern-matching algorithms," IBM Journal of Research and Development, vol.31, no.2, pp.249–260, 1987.
[31] R.E. Tarjan, "Amortized computational complexity," SIAM Journal on Algebraic Discrete Methods, vol.6, no.2, pp.306–318, 1985.
[32] R.A. Baeza-Yates and B. Ribeiro-Neto, Modern information retrieval, ACM press New York, 1999.
[33] M. Schubotz, A. Youssef, V. Markl, and H.S. Cohl, "Challenges of mathematical information retrieval in the NTCIR-11 math wikipedia task," Proc. 38th international ACM SIGIR conference on Research and development in information retrieval, pp.951–954, 2015.
[34] R. Zanibbi, K. Davila, A. Kane, and F.W. Tompa, "The tangent search engine: Improved similarity metrics and scalability for math formula search," ArXiv e-prints, CoRR, vol.abs/1507.06235, 2015.

Appendix: Analysis of Subtree Hash

As Subtree Hash is a hash function, there exists a risk of collision, where two different subtrees are hashed into the same value. We prove that hash value of Subtree Hash distributes uniformly random. Thus, the risk of collision is minimized. We give some definitions and assumptions needed in the proof.

Definition 1: Let TreeHash(n) be the first element of STHrec(n).

TreeHash function returns hashed value which corresponds to the subtree rooted at the given node.

Assumption 1: The hash function used in Subtree Hash (l.7 in Algorithm 1) returns uniformly random value if the input (*label*(*n*) in Algorithm 1) is uniformly random string.

This assumption means that hash function used in Algorithm 1 is good enough uniformity. The probability of minimum collision follows from the uniformity.

The goal of this section is Theorem 2 and it is shown below.

Theorem 2: if all labels in the tree distribute uniformly random, the value TreeHash(n) also distributes uniformly random for any p which is coprime with any possible a (l.7 in Algorithm 1).

This theorem states that Subtree Hash has a good property, that is the probability of hash collision is minimized if an appropriate p is used.

We use mathematical induction on the height of input tree to prove Theorem 2. Base case of mathematical induction is simple, because it follows directly from Assumption 1. Some lemmas are prepared for the proof of inductive step of the mathematical induction.

Lemma 1: Let X be a discrete uniform distribution on $[0, p-1]$. Let $a \in \mathbb{N}$ be coprime with p. $aX \,\%\, p$ is also a discrete uniform distribution on $[0, p-1]$

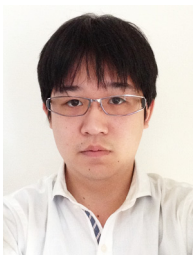
Lemma 2: Let X, Y be a discrete uniform distributions on $[0, p-1]$. $(X+Y)\%p$ is also a discrete uniform distribution on $[0, p-1]$.

Now we have all tools needed to prove the inductive step. By definition of Algorithm A· 1, Eq. (A· 1) holds. In this equation, t represents the root node of the tree whose height is $n > 1$, $|C|$ denotes the number of children of t, and $h_i = TreeHash(c_i)$ where c_i is i-th children of t.

$$TreeHash(t) = \sum_{i=1}^{|C|} a^{i-1} h_i \qquad (\text{A}\cdot 1)$$

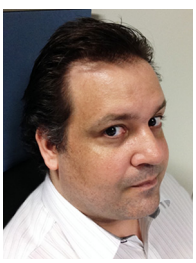
By the assumption of inductive step of mathematical induction, h_i distributes uniformly random. We can prove that $a^{i-1}h_i$ also distributes uniformly random using Lemma 1 Finally, we can prove that TreeHash(t) distributes uniformly random using Lemma 2, and this is what we want to prove.

Thus, we can prove Theorem 2. As we mentioned, this theorem assures that the hash collision probability is minimized where the input tree is random.

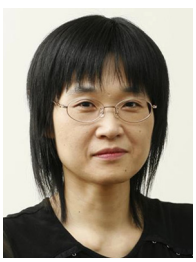


Shunsuke Ohashi has received a master's degree from the Department of Compute Science at the University of Tokyo in 2015.

Giovanni Yoko Kristianto received his bachelor's degree from the Department of Electrical Engineering at the Universitas Gadjah Mada, Indonesia in 2009 and obtained his master's degree from the Department of Computer Science at the The University of Tokyo in 2014. He is now a PhD student in the Department of Computer Science at The University of Tokyo. His research interests include machine learning, information retrieval, and knowledge and information extraction.



Goran Topić has received a master's degree from the Graduate School of Interdisciplinary Information Studies at the University of Tokyo.



Akiko Aizawa graduated from the Department of Electronics at the University of Tokyo in 1985 and completed her doctoral studies in electrical engineering in 1990. She was a visiting researcher at the University of Illinois at Urbana-Champaign from 1990 to 1992. She is currently a professor in Digital Content and Media Sciences Research Division at National Institute of Informatics. She is also an adjunct professor at the Graduate School of Information Science and Technology at the University of Tokyo. Her research interests include statistical text processing, linguistic resources construction, and corpus-based knowledge acquisition.