A NOVEL SIMILARITY-SEARCH METHOD FOR MATHEMATICAL CONTENT IN LaTeX MARKUP AND ITS IMPLEMENTATION

by

Wei Zhong

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering

Summer 2015

A NOVEL SIMILARITY-SEARCH METHOD FOR MATHEMATICAL CONTENT IN LaTeX MARKUP AND ITS IMPLEMENTATION

by

Wei Zhong

Approved: _____

Hui Fang, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____

Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _____

Babatunde A. Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: _____

James G. Richards, Ph.D.
Vice Provost for Graduate and Professional Education

ACKNOWLEDGMENTS

Thank you to my family for their support from every perspective through out my graduate academic education. Thank you to my advisor Hui Fang who offers me the opportunity to develop my idea further and supports me in many other ways. I am also grateful to all InfoLab members for their kind help. And thanks to those not previously mentioned, who have influenced me or helped along the way.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Mathematical content are widely contained by digital document, but major search engines fail to offer a way to search those structural content effectively, because traditional IR methods are deficient to capture some important aspects of math language. In this paper, we propose a similarity-search method for LaTeX math expressions trying to provide a new idea to better search math content. Our approach uses an intermediate tree representation to capture structural information of math expression, and based on a previous idea, we index math expressions by tree leaf-root paths. A search method to limit search set for possible subexpression isomorphism is provided. We rank search results by a few intuitive similarity metres from both structural and symbolic points of view. We also build our own proof-of-concept prototype search engine to demonstrate these ideas, and thus are able to present some evaluation results through this paper. Experiment shows these proposed measurements can advance effectiveness with respect to our baseline search method.

Chapter 1

INTRODUCTION

Apart from general text content, structured information is also widely contained by digital document. Among these, a lot of mathematical content are represented, they are primarily using LaTeX, which is a rich structural markup language. Information Retrieval on those structured data in mathematics language is not as well-studied or exhaustively covered as that is in general text IR research. To better search mathematical content can be significantly meaningful in terms of extending our border on information retrieval.

However, the structured sense of mathematical language, as well as many its semantic properties (see section 1.2), makes general text retrieval models deficient to provide good search results. This is because some fundamental differences between mathematical language and general text. Through this paper, we have made our efforts to tackle some of the problems we are having to search mathematical language. Some of the ideas used in this paper that deals with "tree structured" data can be generalized and potentially applied to other fields of structured data retrieval.

1.1 Math IR Domains

Mathematical information involves a wide spectrum of topics, [1] gives a comprehensive review on mathematical IR researches. We are of cause not focusing on every aspect in mathematical information retrieval. It is good to clarify our concentration in this paper here by first listing a set of concentrations in the related research area and define our target field of study.

Listed here, are considered three major topics for mathematical information retrieval:

1. Boolean or Similarity Search
2. Math Detection and Recognition
3. Evaluation, Derivation and Calculation

Boolean/Similarity search finds or ranks mathematical expressions against a query. They define the criteria of matching expressions or dimensions of similarity between two expressions. This is analogy to the boolean or similarity search of general text search engines, except the query and document may contain mathematical expressions. Some search engines deal with only formula (e.g. SearchOnMath [1], Uniquation [2] and Tangent [3]) and some math-aware search engines (e.g. WolframAlpha [4] and Zentralblatt math from MathWebSearch [5]) are able to search query with mixed text and mathematical formula. These search engines can be useful in many ways, for example, student may utilize it to know which identity can be applied to a formulae in order to give a proof of that formulae. This is the area where we focus in this paper.

Digital mathematical content document can also be in an image format (e.g. a handwritten formula), topics on retrieving information in these image requires detection and recognition of their visual features (texture, outline, shape etc.).

Also, because the nature of mathematical language, a query (e.g. an algebra expression) can potentially derived into alternate forms, or calculated and evaluated into a value. These potentially require a system to handle symbolic or numeric calculation, or even a good knowledge of derivation rules implied by different mathematical

[1] http://searchonmath.com

[2] http://uniquation.com/en

[3] http://saskatoon.cs.rit.edu/tangent/random

[4] https://www.wolframalpha.com/

[5] http://search.mathweb.org/zbl/

expression. Those numeric search engines (e.g. computational engine Symbolab [6] and WolframAlpha) can help evaluate mathematical expressions and reveal the deeper information contained from those expressions.

Besides the three concentrations mentioned, there are many other topics. Knowledge mining, for example, will need deeper level of understanding on math language. A typical goal of this topic is to give a solution or answer based on all the document information retrieved. e.g. "Find an article related to the *Four Color Theorem*" [2].

These topics somehow overlap in some cases, for example, some derivation can be used to better assess the similarity between math formulae, e.g. $\frac{a+b}{c}$ and $\frac{a}{c}+\frac{b}{c}$ should be considered as relevant because their equivalent form of representation. Similarly, mathematical knowledge is required to understand the same meaning (thus high similarity) between $\binom{n}{1}$ and C_n^1. So boolean or similarity search possibly involves a level of understanding on mathematical language. However, we are not going to include these problems into our research domain, instead, this paper addresses mathematical expression similarity from only structural and symbolic perspectives.

1.2 Issues in Measuring Similarity

Unlike general text content, mathematical language, by its nature, has many differences from other textual documents, there are a number of new problems in measuring mathematical expression similarity. Even without caring about the possible derivations and high level knowledge inference, there are still a set of new problems for measuring mathematical similarity.

Firstly, the differences between mathematical expressions should be captured in a cooperative manner in order to measure similarity, because only respecting symbolic information is not sufficient in mathematical language. To illustrate this point, we know that $ax+(b+c)$ in most cases is not equivalent to $(a+b)x+c$ although they have the same set of symbols, because the different structure represents a different semantic meaning. And the order of tokens in math expression can be commutative in some

[6] Symbolab Web Search: http://www.symbolab.com

cases but not always, for example, commutative property in math makes $a+b = b+c$ for addition operation, but on the other hand $\frac{a}{b}$ is most likely not equivalent to $\frac{b}{a}$. These characteristics make many general text search methods (e.g. *bag of words* model, *tf-idf* weighting) inadequate. Moreover, symbols can be used interchangeably to represent the same meaning, e.g. $a^2 + b^2 = c^2$ and $x^2 + y^2 = z^2$. However, interchanging symbols does not always preserve mathematical semantics, changes of symbols in expression preserve more syntactic similarity when changes are made by substitution, e.g. Given query $x(1 + x)$, expression $a(1 + a)$ are considered more relevant than $a(1 + b)$.

Secondly, how we evaluate structural similarity between expressions is a question. A complete query may structurally be a part of a document, or only some parts of a query match somewhere in a document expression. In cases when a set of matches occur within some measure of "distance", we may consider them to contribute similarity as a whole, but when matches occur "far away" for a query expression, then under the semantic implication of mathematics, they probably will not contribute the similarity degree in any way. We need metrics to score these similarity under certain rules for relevance assessments.

Lastly, when trying to capture more semantic information from expressions, we can improve our measurement on similarity but it may introduce more ambiguity. For example, semantically incorrect math markups in document, e.g. using "sin" in LaTeX markup instead of macro "\sin", will make it difficult to tell whether it is a product of three variables or a *sine* function if we want to capture their semantical meaning in such a depth. And depending on what level of semantical meaning we want to capture, ambiguity cases can be different. Consider $f(2x + 1)$, if we want to know if f is a function rather than a variable, the only possibility is looking for its contexts, but we can nevertheless always think of it as a product without losing the possibility to search similar expression like $f(1 + 2y)$, the same way goes reciprocal a^{-1} and inverse function f^{-1}. Most often, even if no semantic ambiguity occurs, efforts are needed to capture some semantical meanings. e.g. In $\sin 2\pi$, it is not easy to figure out, without a knowledge on trigonometric function convention, that 2π is the subordinative of sine

function.

1.3 Contribution Summary

This paper has proposed new ideas to search math expressions in a limited search set and evaluate both structural and symbolic similarity of two two math expressions. Based on an existing idea of using leaf-root path (which will be described at section 2.2) to search mathematical expressions, we have further developed it to index structural information for easy discover of structurally similar document expressions. To summarize, the research presented contributes the following points to the field of mathematical information retrieval.

- The method based on operation tree representation of mathematical expression to index information by sub-paths and search math query through simultaneously merging sub-paths. This also enables the ability to prune search set and potentially provide ways to parallelize search process.
- Formally defined sub-expression relation between two mathematical expressions. Based on this definition, we have observed some interesting properties, and depending on this, we have proposed a new search method which can limit the indexed expressions being possible sub-expression of query to a subset of index.
- An algorithm to "check" defined sub-expression relation between two mathematical expressions.
- An algorithm to evaluate symbolic similarity between two mathematical expressions with consideration of symbol α-equality.
- Propose a ranking schema to combine three factors (matching ratio, matching depth, and symbolic similarity score) as a tuple to score overall similarity between a query and a document math expression.
- A parser to directly parse LaTeX math mode content into in-memory operation tree representation, omitting semantic-irrelevant content.
- A set of tools that implement our idea with source code avaiable online: https://bitbucket.org/t-k-/cowpie. And a newly-assembled corpus from StackExchange to provide a LaTeX (instead of MathML) content dataset for public assessment and evaluation.

Chapter 2

RELATED WORK

Boolean or similarity search for mathematical content is not a new topic, conference in this topic is getting increasingly research attention and the proposed systems have progressed considerably [3]. A variety of approaches have already emerged in an early timeline [4], but we can nevertheless categorize them into a few different ideas. [5, 6, 7] use the same way to classify them into text-based and tree-based (structure-based), here we decide to follow the same classification method and give an overview on those different ideas.

2.1 Text-Based Methods

Many researchers are utilizing existing models to deal with mathematical search, and use texted-based approaches to capture structural information on top of matured text search engine and tools (such as *Apache Lucene*).

DLMF project from NIST [8] uses "flattening process" to convert math to textualized terms, and normalize them into *sorted parse tree normal form* which creates an unique form for all possible orders of nodes among associative and commutative operators. Then further introduces serialization and scoping to stack terms [9], trying to capture structure information by using text-IR based systems that supports phrase search. Similar idea is also used by [10], expressions are also augmented for various possible representations, variables are further replaced and normalized, but they are using postfix notation, allows to search subexpressions without knowing the operator between them. MIaS system [11, 12, 13], like the methods above, are also trying to reorder commutative operations, normalize variables and constance numbers into unified

symbols, doing augmentation in a similar fashion. It indexes expressions and subexpressions from all depth levels. The system is able to discriminate and assign different weight based on their generalization level, and place emphasis on that a match in a complex expression is assigned higher weight [13].

Augmentation usually consists a storage demand for combination of both symbols (e.g. a and b) and unified items (*id, const*) in different levels, in order to capture both symbolic information and structure information. Thus implies complex expressions with many commutative operators will cost inevitably larger storage space, the benefits of capturing expression variances will be overshadowed.

Although named as structured-based approach, [14] is using *longest common subsequence* algorithm to capture structure information (in an unified *preprocessed string* and a *level string*). The method takes $O(n^2)$ complexity for comparing a pair of formulae, and no index method is proposed. Therefore is not feasible to efficiently apply to a large collection. The Mathdex search engine [15], from another perspective, uses query likelihood approach [16] to estimate how likely the document will generate the query expression. Math GO! [17] is another system advances some transitional method to better search math content. It tries to find all the symbols and map formula pattern to pattern name keywords (like *matrix* or *root*), and proposes to replace term frequency by co-occurrence of a term with other terms.

2.2 Structure-Based Methods

What text-based methods share in common is they are converting math language symbols to linear tokens, the intrinsic defect when using a bag of words to replace structured information will make conversion process lose considerable information or cause storage-inefficiency. In order to cope with the problems from text-based approach, structure-based methods generally generate intermediate tree structure, and use these information to index or search.

Term indexing

Whelp [18] and MathWebSearch (MWS) directed by Kohlhase [19, 20, 21], are one of the notable structure-based ones which are derived from *automatic theorem proving* and *unification theory* [22]. The system of MWS uses *term indexing* [23] in a *substitution tree index* [23] to to minimize access time and storage. Because the subexpression is not easy to search using substitution tree, MWS indexes all sub-terms, but the increased index size remains manageable [19]. However, their index relies heavily on RAM memory, the considerable RAM resource usage (170GB reportedly [21]) makes it have to scale to accommodate 72% papers on arXiv.

Leaf-root path

[24] uses leaf to root XML path in a MathML object to represent math formula. When efficiency is considered, it only indexes the first and deepest path (to indicate how a formula is started and presumably the most characteristic part of a formula); when user wants to obtain the perfect-match result, it indexes all the MathML object leaf-root path. The boolean search is performed by giving all the paths match with those of the search query. [25] further develops with incorporation of previous method using breath-first search, to add sibling nodes information into index and have achieved better effectiveness.

Very similar idea is proposed by [26] and used in [27]. The latter transform MathML to an "apply free" markup from which the leaf-root path are extracted. Leaf-root path is also used to evaluate similarity between two expressions in MathML.

Symbol layout tree

A *symbol layout tree* [28] (SLT) or *presentation tree* [5] describes geometric layouts of mathematical expression. WikiMirs [5] uses two templates to parse LaTeX markup with two typical operator terms: explicit ones ("\frac", "\sqrt", etc.) and implicit ones ("+", "÷", etc.) to form a presentation tree, then extracts original terms and generalized terms from normalized presentation tree, to provide the flexibility of

both fuzzy and exact search. [28] uses symbol layout tree as a kind of substitution tree, each branch in the tree represents a binding chain for variables, and every child node is an instance of its parent for a generalized term. They introduce *baseline size* to help group similar expressions together in their substitution SLT in order to decrease tree branch factor, however, this makes a single substitution variable not able to match multiple terms along the baseline. Also their implementation makes it unable to index certain formula (e.g. a left-side superscript) and have to generate many queries (e.g. all possible format variations and sub-expressions etc.) for a single query at the time of search. Later [6, 29] have developed a *symbol pairs* idea to capture a relative position information between symbol pairs. Due to the many possible combinations of symbol pairs in a complex math expression, the storage cost is intrinsically large. However, the key-value storage style will be suitable for fast lookup (e.g. HASH).

Other structure-based methods

A novel indexing scheme and lookup algorithm is proposed by [30], its index has hash signature for each subtree because they have observed a lot of common subtree structure occur in math formula collection. This idea will result in a slower index growth. Their lookup algorithm supports wildcards, and performs a boolean match test. Although their lookup time is generally below 700ms, the index size where query lookup time is tested is unclear, but presumably no greater than 70,000 expressions. By constructing a DOM tree, [31] extracts semantic keywords, structure descriptions to indicate subordinative relationship in a string format. The similarity is calculated using normalized tf-idf vector (trained by clustering algorithms) by dot product. Although the final index is generated from text, promising results have been achieved. Tree edit distance is adopted by [32], it tries to overcome the bad time complexity of original algorithm by summarizing and using a structure-preserving compromised edit distance algorithm. Although the result shows query processing time is long but it is reduced to average 0.8 seconds by applying with an early termination algorithm along with a distance cache [33].

2.3 Other Related Work

There are a number of articles trying to use image to assess math expression similarity. [34] compares their image-based approach using connected component-based feature vector with a proposed text-based method, reported precision@k values are low, but the potential for this method to be combined with shape representations or other features will potentially improve it and make it valuable for searching mathematical expressions in image format. [35] uses X-Y tree to cut the page in vertical and horizontal directions alternatively, in order to retrieve math symbols from images.

A lattice-based approach [36] builds formal concept based on selected feature sets of each formula. The ranking is calculated by the distances from query in the lattice map when the query is inserted.

2.4 Performance Review

In the review of many related past research in section 2, we find the combination of state-of-art general text search engine (i.e. *Apache Lucene*) with the efforts to augment expressions by permutation and unification to satisfy the needs of mathematical search has achieved a good result in different metrics of evaluation: The text-based system of MIaS over-performs those of structure-based and ranked the first in four out of six measurement in NTCIR-11 conference [37]. However, structure-based method such as symbol pairs proposed in [6, 29] also gets a competitive result [37].

Although text-based methods have achieved relatively better effectiveness, their commonly adoption of augmentation makes it expensive in terms of hardware storage. On the other hand, structure-based method has the merit to best capture semantic information of a typical mathematical expression, thus still has the potential to further improve effectiveness given the fact that the tools and theory behind test-based methods are already mature and have been developed for decades.

Chapter 3

METHODOLOGY

Our method can be seen as an approach built upon the idea of leaf-root path or sub-path [26, 24, 27, 25] from an operation tree [1]. But we have developed this idea further in many ways. Our index is composed from leaf-root paths extracted from an operation tree, we simultaneously search along the way of all leaf-root paths from a given query operation tree, which is essentially traversing "reversed" sub-paths. Searching in this way makes a pruning method possible to limit our search set. We also propose a sub-structure testing algorithm, which are utilizing some observed properties from our indexed tree. Apart from these, we provide a symbolic similarity algorithm to rank α-equivalent expressions higher. The methods in a nutshell is, for a document expression, construct operation tree, break it down into sub-paths, and index those paths. For a query, traversal index tree as the same way of going through the reversed sub-paths of that query operation tree, search the sub-paths and their merged paths in index meanwhile calculate indexed expression similarity to rank each document expression.

3.1 Intuitions

First it is beneficial to document our intuitions on using operation tree as our intermediate representation and our idea to index it in a way of reversed sub-path tree, and also explain in abstract why this way helps reduce index space and boost search speed. We will give an illustrative example to describe these processes further in section 3.4.5.

3.1.1 Commutative Immunity

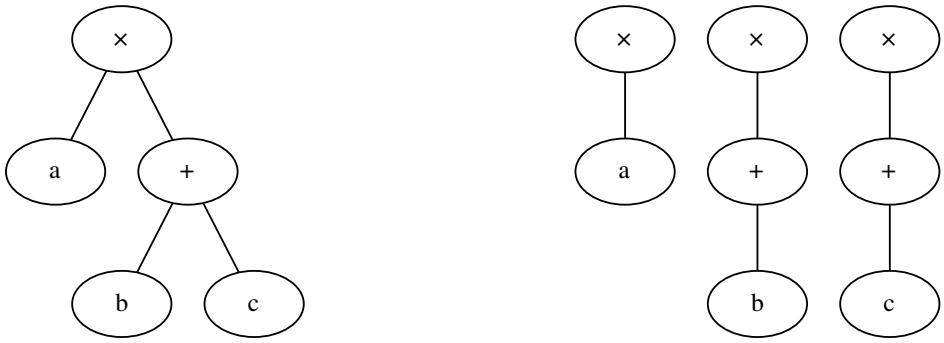
Operators with semantic implication of commutative property (e.g. addition and multiplication) are exhaustively used in mathematical language. The ability to identify the identical equations for any permutation is very essential for a mathematical similarity search engine. Given this as a start point, the leaf-root paths have the advantage to cope this so that we do not need to generate different order of patterns to match formulae with commutative operator. To illustrate this, we know that a leaf-root path from an operation tree (see figure 3.1) is generated through traversing in a bottom-up (or top-down) fashion from a tree, therefore these sub-paths are independent to the relative position of commutative operands. In another word, an operation tree uniquely determines the leaf-node paths decomposed from the tree, no matter how operands are ordered.

3.1.2 Sub-Structure Query Ability

On the other hand, the structure of operation tree also makes it easy to represent sub-expression relation with a formula, because a sub-expression in a formula is usually (depending on the way we construct an operation tree) also a subtree in an operation tree. And by going up from leaves of operation tree, we are essentially traversing to an expression from its subexpression for every level. By making all the leaf-root paths as an index, we can search an expression by going through and beyond the leaf-root paths from its subexpression. This makes operation tree better in terms of searching an expression given a sub-expression of it in query. And it avoids information augmentation on index as some other structure-based methods need (e.g. index all sub-terms of an expression in MWS [19]). Therefore it also helps save storage space.

3.1.3 Index and Search Properties

Additionally, some properties from this method suggest some reduce of space and pruning possibilities in search process. First the sub-paths themselves can be indexed into a tree so that we can search a sub-path by traversing a sub-path tree,



$a(b+c)$ in operation tree Generated leaf-root paths

Figure 3.1: Leaf-root path example

instead of hashing it to find a corresponding value as the symbol-pair search engine (i.e. *Tangent* [6]) does. This allows us to save a lot space as the reverted sub-paths of a large collection will have a great percentage of level sharing a common substring with each other. Also the way to search in a tree structure with a limited branch factor does not lose much efficiency compared to the HASH methods used in *Tangent*, while also offers great storage efficiency. Second, by searching from all the "reverted" sub-paths of a query expression in our proposed index, and apply an intersection on the results from different sub-paths, we will find all the expressions having that query as subexpression (number four observed property from section 3.2.3). And during this search process, upon going further from the query expression root in the "reverted" sub-path, we can merge the next search directories by pruning all the entries that are not shared in common among all the "reverted" search paths. Furthermore, if a relevant indexed formula has been found in a search path level, then other relevant sub-paths will be found at the same level too, thus some implementation strategies can be applied to speed search further (i.e. distributed search to quick search massive).

3.2 Structure Similarity

The basic ideas used in our approach, to test whether a mathematical expression is an sub-structure of another, to prune and constrain search set are the foundation

work in our research. It is desired to give a description in a formal language so that we can deliver these ideas in the most precise way. Some important observations as well as brief justifications are provided for completeness.

3.2.1 Definitions

For the second issue addressed in section 1.2, specifically, to assess the structural similarity. Some formal definitions are proposed from previous study [38] which provides a quantified n-similarity relation to address the similarity degree between math expressions, based on the max-weight common subtree. The subtree, by their definition, includes all descendants from a node, however, we are going to use the subtree definition in graph theory here to describe the sub-structure relation. To be explicit, given a rooted tree T, the connected graph whose edges are also in T is defined as the subtree of T. We are also going to give a definition to describe a "match" between query and a document math formula.

Because we are using some formal descriptions to better illustrate our ideas, there are some notations we need to clarify that are used throughout this paper.

A *path* p is a sequence of numbers given by $p = p_0 p_1 \dots p_n$ where $n \geq 0$, $p_i \in \mathbf{R}$. $\mathbf{P}$ is the set of all paths. A *leaf-root path* is a path from root to a leaf in a tree. Any function $f : \mathbf{R} \to \mathbf{R}$ applied on path p is mapped to a path too: $f(p) = f(p_0)f(p_1)\dots f(p_n)$. And we name a *concatenation* of two paths ${}^1p = p_0 p_1 \dots p_n$ and ${}^2p = p_n p_{n+1} \dots p_m$ where $m \geq n$, to be a new path denoted as ${}^1p \cdot {}^2p = p_0 p_1 \dots p_n p_{n+1} \dots p_m$, and the concatenation of a path p on a path set $S = \{s_1, s_2 \dots s_n\}$ is defined as $S \cdot p = \{s_1 \cdot p,\ s_2 \cdot p\ \dots\ s_n \cdot p\}$. The *longest common prefix* path p^* between two paths p_1 and p_2 is mapped by the function named lcp, which is defined by $p^* = \text{lcp}(p_1, p_2) = \text{lcp}(p_2, p_1)$.

Each node in a formula tree is associated with a label (labels are not required to be distinct here) to represent an unified mathematical operator, variable, constance etc., a class of similar operators can have the same label value (e.g. same label for token $+$, $\oplus$ and $\pm$). Also each leaf node is associated with a symbol value to represent a symbolic instance of that constance or variable (e.g. "123", β, x, y etc.). Besides, a

formula subtree relation is also defined to address the sub-structure relation between two mathematical expressions. Below are our formal definitions.

3.2.1.1 Formula Tree

A *formula tree* is a labeled rooted tree $T = T(V, E, r)$ with root r, where each vertices $v \in V(T)$ is associated with a label (not necessarily unique in the same tree) $\ell_T(v) \in \mathbf{R}$ mapped by label function ℓ_T, and each leaf $l \in V(T)$ is further associated with a symbol $\mathcal{S}_T(l) \in \mathbf{R}$ mapped by symbol function $\mathcal{S}_T$. For convenience, we will write function ℓ and $\mathcal{S}$ as their short names which refer to the tree implied by the context, and we use $\mathcal{S}(p)$ to indicate the symbol of the leaf in a leaf-root path p. In addition, we use ${}^sT$ to denote a subtree in T rooted by vertices $s \in V(T)$, with all its descendants.

3.2.1.2 Formula Subtree

Given formula tree S and T, we say S is a *formula subtree* of T if there exists an injective mapping $\phi : V(S) \to V(T)$ satisfying:

1. $\forall\ (v_1, v_2) \in E(S)$, we have $(\phi(v_1), \phi(v_2)) \in E(T)$;
2. $\forall\ v \in V(S)$, we have $\ell(v) = \ell(\phi(v))$;
3. If $v \in V(S)$ is a leaf vertices in S, then $\phi(v)$ is also a leaf in T.

Such a mapping ϕ is called a *formula subtree isomorphic embedding* (or *formula embedding*) for $S \to T$. If satisfied, we denote $S \preceq_l T$ on Φ, where Φ ($\Phi \neq \emptyset$) is the set of all the possible formula embeddings for $S \to T$.

3.2.1.3 Leaf-Root Path Set

A *leaf-root path set* generated by tree T is a set of all the leaf-root paths from tree T, mapped by a function $g(T)$.

3.2.1.4 Index

An *index* Π is a set of trees such that $\forall\ T \in \Pi$, we have $T \in \mathcal{I}_\Pi(a)$ for any $a \in \ell(g(T))$, we say T is *indexed* in Π and $\mathcal{I}_\Pi$ is called *index look-up function* for index Π.

3.2.2 Search Method

For a collection of document expressions, we will index them by merging all the reverted leaf-root paths from each document formula tree into a large "inverted" index tree, in which each node at path a stores the information of all the indexed formula trees in $\mathcal{I}_\Pi(a)$. Through searching all sub-paths from a query formula tree T_q at the same time, we are able to limit the set of possible formula trees being structurally matching (in formula subtree relation) with T_q, to only a subset of our index. This is illustrated as follows.

Given an index Π and a formula tree T_q, $\forall\ T_d \in \Pi$: If $T_q \preceq_l T_d$ on Φ, then $\exists\ \hat{a} \in \mathbf{P},\ s.t.$

$$T_d \in \bigcap_{a \in L} \mathcal{I}_\Pi(a)$$

where $L = \ell(g(T_q)) \cdot \hat{a}$.

Justification. Denote the root of T_q and T_d as r and s respectively. Let $\hat{p}$ be the path determined by vertices from $t = \phi(r)$ to s in T_d, and ${}^1p,\ {}^2p \ldots {}^np,\ n \geq 1$ be all the leaf-node paths in T_q. Then $\hat{a} = \ell(\hat{p})$, this is because: $L = \ell(\{\,{}^1p,\ {}^2p \ldots {}^np\}) \cdot \hat{a} = \ell(\{\phi({}^1p), \phi({}^2p) \ldots \phi({}^np)\}) \cdot \ell(\hat{p}) = \{\ell(\phi({}^1p) \cdot \hat{p}), \ell(\phi({}^2p) \cdot \hat{p}) \ldots \ell(\phi({}^np) \cdot \hat{p})\}$. According to definition 3.2.1.2 and $t = \phi(r)$, we have $\phi({}^ip) \cdot \hat{p} \in g(T_d),\ 1 \leq i \leq n$. Since $T_d \in \Pi$, T_d is indexed in Π with respect to each of the elements in L, that is to say $\forall\ a \in L,\ T_d \in \mathcal{I}_\Pi(a)$.

To summarize, we search the index by intersecting the indexed formula trees from all the generated leaf-node paths at the same time, then further possible search path $\hat{a}$ is only possible when paths along the generated leaf-node paths in the index have a common prefix. Therefore we can "merge" the paths ahead and prune those paths

not in common. Level by level, we are always able to find the structurally matched formula tree as long as it is indexed in Π.

3.2.3 Substructure Matching

However, query formula tree will not necessarily be formula subtree of all the document (indexed) formula trees in our search set $\bigcap_{a \in L} \mathcal{I}_\Pi(a)$, even if their generated leaf-root paths are identical. One supporting example for this point is shown in figure 3.2. To address this problem, we propose an algorithm described in in figure 3.4, to test the document formula trees in our search set to see if they are in formula subtree relation with query formula tree. This algorithm is inspired from the following observations.

3.2.3.1 Observation #1

For two formula trees which satisfy $T_q \preceq_l T_d$ on Φ, then $\forall\ \phi \in \Phi$, $p \in g(T_q)$, also any vertices v along path p, the following properties are obtained:

$$\deg(v) \leq \deg(\phi(v)) \tag{3.1}$$

$$\ell(p) = \ell(\phi(p)) \tag{3.2}$$

$$|g(T_q)| \leq |g(T_d)| \tag{3.3}$$

Justification. Because $\forall\ w \in V(T_q)\ s.t.\ (v, w) \in E(T_q)$, there exists $(\phi(v), \phi(w)) \in E(T_d)$. And for any (if exists) two different edges $(v, w_1), (v, w_2) \in E(T_q)$, $w_1 \neq w_2 \in V(T_q)$, we know $(\phi(v), \phi(w_1)) \neq (\phi(v), \phi(w_2))$ by definition 3.2.1.2. Therefore any different edge from v is associated with a distinct edge from $\phi(v)$, thus we can get (3.1). Given the fact that every non-empty path p can be decomposed into a series of edges $(p_0, p_1), (p_1, p_2) \ldots (p_{n-1}, p_n),\ n > 0$, property (3.2) is trivial. Because there is exact one path between every two nodes in a tree, the leaf-root path is uniquely determined by a leaf node in a tree. Hence the rationale of (3.3) can be obtained in a similar manner with that of (3.1), expect neighbor edges are replaced by leaf-node paths.

3.2.3.2 Observation #2

Given two formula trees T_q and T_d, if $|g(T_q)| = 1$ and $\ell(g(T_q)) \subseteq \ell(g(T_d))$, then $T_q \preceq_l T_d$.

Justification. Obviously there is only one leaf-root path in T_q because $|g(T_q)| = 1$. Denote the path as $p = p_0 \dots p_n,\ n \geq 0$ where p_n is the leaf. Since $\ell(p) \subseteq \ell(g(T_d))$, we know that there must exist a path $p' = p'_0 \dots p'_n \in g(T_d)$ such that $\ell(p) = \ell(p')$ where p'_n is the leaf of T_d. Then the injective function $\phi : p_i \to p'_i,\ 0 \leq i \leq n$ satisfies all the requirements for T_q as a formula subtree of T_d.

3.2.3.3 Observation #3

For two formula trees T_q and T_d, if $T_q = T(V, E, r) \preceq_l T_d$ on Φ, $\forall a, b \in g(T_q)$ and a mapping $\phi \in \Phi$. Let $T'_d = {}^tT_d$ where $t = \phi(r)$ and $a' = \phi(a)$, $\forall\ b' \in g(T'_d)$, it follows that:

$$b' = \phi(b) \quad \Rightarrow \quad |\mathrm{lcp}(a, b)| = |\mathrm{lcp}(a', b')|$$

Furthermore, $\forall\ c \in g(T_q)\ s.t.\ |\mathrm{lcp}(a, b)| \neq |\mathrm{lcp}(a, c)|$, we have

$$|\mathrm{lcp}(a, b)| = |\mathrm{lcp}(a', b')| \quad \Rightarrow \quad b' \neq \phi(c)$$

Justification. Because $a, b \in g(T_q)$, thus $a_0 = b_0 = r$, and we can also make sure $\mathrm{lcp}(a, b) \geq 1$. Denote the path of $a = a_0 \dots a_n a_{n+1} \dots a_{l-1}$, similarly denote the path of b as $b = b_0 \dots b_n b_{n+1} \dots b_{m-1}$, where the length of each $l, m \geq 1$ and $a_i = b_i$, $0 \leq i \leq n \leq \min(l-1, m-1)$ while $a_{n+1} \neq b_{n+1}$ if $l, m > 1$. On the other hand $a' = \phi(a)$ and $b' \in g({}^tT_d)$, therefore $a'_0 = \phi(a_0) = \phi(r) = t = b'_0$. For the first conclusion, if $b' = \phi(b)$, there are two cases. If either $|a|$ or $|b|$ is equal to one then $|\mathrm{lcp}(a, b)| = |\mathrm{lcp}(a', b')| = 1$; Otherwise if $l, m > 1$, path $a_0 \dots a_n = b_0 \dots b_n$ and $a_{n+1} \neq b_{n+1}$ follow that $\phi(a_0 \dots a_n) = \phi(b_0 \dots b_n)$ and $\phi(a_{n+1}) \neq \phi(b_{n+1})$ by definition. Because edge $(\phi(a_n), \phi(a_{n+1}))$ and $(\phi(b_n), \phi(b_{n+1}))$ are both in $E(T'_d)$, we have $|\mathrm{lcp}(a, b)| = |\mathrm{lcp}(a', b')| = n$. For the second conclusion, we prove by contradiction. Assume $b' = \phi(c)$, by the first conclusion we know $|\mathrm{lcp}(a, c)| = |\mathrm{lcp}(a', b')|$. On the other hand,

because $|\text{lcp}(a,c)| \neq |\text{lcp}(a,b)| = |\text{lcp}(a',b')|$, thus $|\text{lcp}(a,c)| \neq |\text{lcp}(a',b')|$ which is impossible.

3.2.4 Interpretation

The observations above offer some insights on how to test a substructure of a mathematical expression.

First we give some explanations on the definition. A formula subtree relation defined in 3.2.1.2 describes not only a sub-structure relation between two math expressions, it also requires a label similarity and leaf inclusion. Because structure shape (subtree isomorphic) is not the only one factor to determine whether a math formula is a subexpression of another. Given expression in figure 3.1 as an example, $b+c$ and $a \times (b+c)$ are considered "similar" because $b+c$ is structurally an subexpression of $a \times (b+c)$. However, if expressions with different symbols but in similar semantics are given, e.g. $b \oplus c$ or $b \pm c$, they should also be considered as similar to $a \times (b+c)$ because both the operations has the similar semantical meaning as "add". These operations should be labeled in which all the similar operation tokens have the same label value. Also, operation tree representation generally puts operator in the intermediate nodes and operands in the leaves, so it is not common to address a sub-structure without leaves, like "$a \times +$". This is the reason that a structure-similarity relation of two should also contain their leaves.

Now that we have defined our structure similarity rule as whether two trees T_q and T_d can satisfy $T_q \preceq_l T_d$, we break down a formula tree into leaf-root paths p and index the label of each path $\ell(p)$. So if given a "similar" path q, we can further find the previous trees that also have $\ell(q)$ as its labeled path. In section 3.2.3, the first observation gives some constrains to test if two leaf-node paths are similar without knowing the complete tree from which they are generated. However, comparing all the paths from the index one by one would be very inefficient. Section 3.2.2 suggests if we search the index by all the generated leaf-node paths from a tree at the same time, then we may just need to look into an intersected region instead of the whole collection.

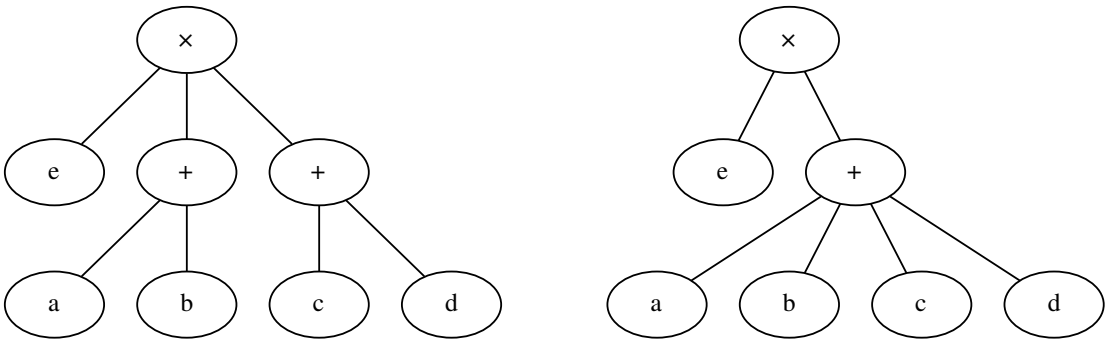


Figure 3.2: Leaf-root paths with different structure

Because every tree indexed ($T_d \in \Pi$) and matched by the query will be found at our defined search set.

However, knowing the matched tree in in a set does not necessarily mean all the tree in the set match with query. Figure 3.2 gives one case where two set of leaf-root paths are identical while the structures from which they are generated are different, and not in any sub-tree relation. If left figure is the query, then we can certainly find the tree in right figure as long as it is indexed, indicated by section 3.2.2. Although leaf-root paths offer some desired properties, whether the trees found through searching sub-paths of a query are also structure isomorphic with the query tree is still unknown.

Observations #2 and #3 in section 3.2.3 offer the way to test structure isomorphic. The former is a sufficient condition to test structure isomorphic, but the tree must first have only one leaf-root path. The latter states two necessary conditions to be a formula subtree of another. This leads to an idea to decompose the tree and divide the problem into subtree matching problems by ruling out impossible matches between leaf-root paths using observation #3, until it is obvious to conclude the structure isomorphic in a sub-problem by using observation #2. These observations inspire the idea to filter expressions with structure isomorphism in our search set $\bigcap_{a\in L}\mathcal{I}_{\Pi}(a)$.

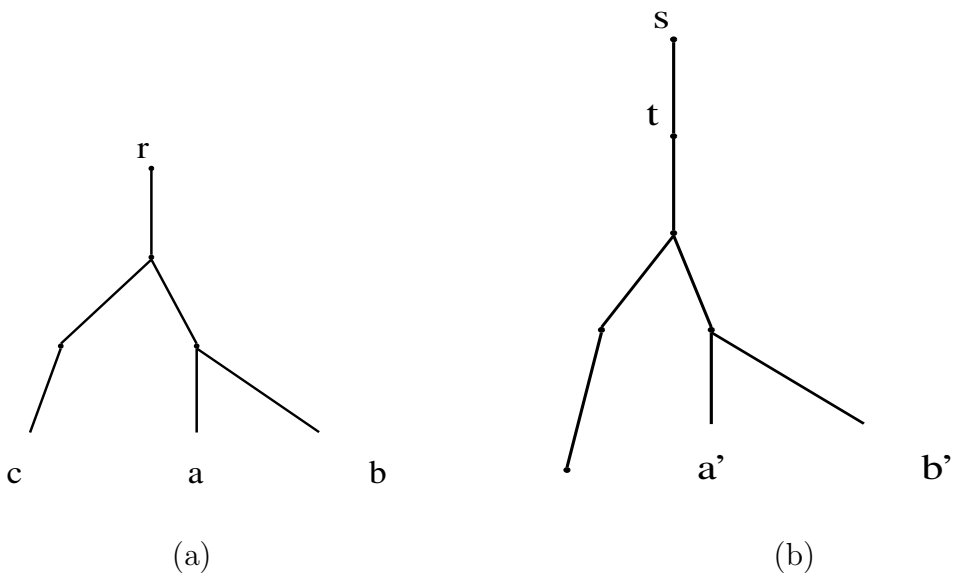


Figure 3.3: Formula subtree matching

3.2.5 The Decompose-and-Match Algorithm

Here we propose and describe an algorithm for formula subtree matching based on the interpretation in section 3.2.4. Figure 3.3 illustrates a general case in which query tree (a) is trying to match a document tree (b).

Initially every leaf-root path in (a) should be associated with a set of leaf-root paths in (b) that are possible (by the constrains of observation #1 in 3.2.3) to be isomorphic, we call this set *candidate set*. Given figure 3.3, the candidate set of path a in (a) probably is $\{a',\ b'\}$ in (b) if the nodes are assigned universally the same label. Then we arbitrarily choose a path in (a) as a reference path (heuristically a *heavy path* [39]), for each of the paths in its candidate set, we choose it as reference path in (b), and suppose we choose a' here. At this time we can apply the two constrains from observation #3 and ruling out some impossible isomorphic paths in candidate set of each path in (a), then divide the problems further. For example, because $|\text{lcp}(a,b)| = |\text{lcp}(a',b')|$, we know b' is still in candidate set of b; while b' is not in candidate of c anymore because $|\text{lcp}(a,b)| \neq |\text{lcp}(a,c)|$. After going through these eliminations for

each leaf-node path (except the reference path a) in (a), we now have two similar subproblems: c as a subtree along with its candidate set, and b as a subtree along with its candidate set. We can apply this algorithm recursively until a very simple subproblem is encountered (e.g. the case in observation #2). During this process, if we find any candidate set to be empty, we stop the subproblem process and change to another reference path or stop the algorithm completely if every possible reference path is tried.

The detailed algorithm is described in figure 3.4. The main procedure is *decomposeAndMatch* where the parameters Q and C are the set of leaf-root paths in query tree and the candidate sets associated with all leaf-root paths respectively. The procedure returns SUCC if a match is found, otherwise FAIL is returned indicating the formula tree in (a) cannot be a formula subtree of that in (b).

3.3 Symbolic Similarity

Besides structural similarity, symbolic similarity is also essential to be considered, because it has some benefits to further differentiate math expression similarity.

Firstly, it is essential to score equations with symbolic matches a higher similarity because they may imply more semantic meaning. For example, $E = mc^2$ is considered more meaningful when exact symbols are used rather than just being structure identical with $y = ax^2$.

Secondly, as illustrated in section 1.2, same mathematical symbols in an expression (i.e. bound variables) usually can only maintain semantical equality if the changes are made by substitutions. (similar to the notion of α-equality [40]). This is an important semantic information that we need to capture in most math expressions that contain bound variables, to capture this kind of equality between expressions certainly involves comparison of symbols.

There is one thing to address here, in many mathematical search systems, a query may be specified with wildcards and thus will match any document with an expression substitution to that wildcard. And a query symbol not specified by wildcard

```
1: procedure removeCandidate(d, Q, C)
2:     for a ∈ Q do
3:         if C_a = ∅ then
4:             return ∅
5:         else
6:             C_a := C_a − {d}
7:     return C
8:
9: procedure match(a, a′, Q, C)
10:    for b ∈ Q do
11:        t := lcp(a, b)
12:        Q_t := Q_t ∪ {b}
13:        P := P ∪ {t}
14:    for t ∈ P do
15:        for b ∈ Q_t do
16:            for b′ ∈ C_b do
17:                if t ≠ lcp(a′, b′) then
18:                    C := removeCandidate(b′, Q_t, C)
19:                    if |C| = 0 then
20:                        return FAIL
21:        if decomposeAndMatch(Q_t, C) = FAIL then
22:            return FAIL
23:    return SUCC
24:
25: procedure decomposeAndMatch(Q, C)
26:    if Q = ∅ then return SUCC
27:    a := OnePathIn(Q)                          ▷ Choose a reference path in Q
28:    Q_new := Q − {a}
29:    for a′ ∈ C_a do
30:        C_new := removeCandidate(a′, Q_new, C)
31:        if C_new = ∅ then return FAIL
32:        if match(a, a′, Q_new, C_new) then return SUCC
33:    return FAIL
```

Figure 3.4: The decompose-and-match algorithm

is expecting an exact symbolic occurrence in document. Yet in our approach, same symbols imply a higher rank, we can get an exact symbolic match if it does exist in our index. On the other hand, we do not have to specify wildcard to match its α-equalities. [11] also doubts the wildcard demand in mathematical search as it is not common to expect an exact symbol occurrence when we query in mathematical language.

3.3.1 Ranking Constrains

As we have discussed, symbolic similarity is essential to be captured, in order to further rank document expressions. Here we propose two constrains to addressed all the considerations, they can be summarized as:

- Given two formula trees $T_q \preceq_l T_d$ on Φ, suppose a leaf $l \in V(T_q)$ is isomorphic to leaf $l' \in V(T_d)$, that is to say, there exists $\phi(l) = l'$, where $\phi \in \Phi$, then if their symbol matches, i.e. $\mathcal{S}(l) = \mathcal{S}(l')$, we score them higher than those do not match symbolically. And the more symbolic matches there are, the higher symbolic relevance degree two expressions would expect to have.
- α-equivalent expressions have more symbolic relevance degree than those are not, and the more *bond variables* (variable with same symbol in an expression) two expressions match at the structurally matching positions, the more symbolically relevant they are considered to be.

In this paper, we use these two constrains as the rule to rank retrieval results in terms of symbolic similarity. And in cases where both constrains can be applied, we prioritize the second constrain. This is because, intuitively, as long as the semantic meaning of two expressions is the same, using different set of symbols does not make a difference. However, bound variable match is more important because an mathematical expression with more than one identical symbols most often implies those symbols represents the same variable.

The constrains and idea above are illustrated by the following example. Let the rank of a document math expression d be $r(d)$, and given query $\sqrt{a}(a-b)$ for instance. It is easy to see under the first constrain:

$$r\big(\sqrt{a}(a-b)\big) > r\big(\sqrt{a}(a-x)\big) > r\big(\sqrt{x}(x-y)\big)$$

The document with the highest rank here is an exact match, with three symbols matching in total. The second document has two symbols matching while the third document has no symbolic match at all.

In the same manner, by the second constrain we have:

$$r\big(\sqrt{x}(x-b)\big) > r\big(\sqrt{x}(y-b)\big)$$

The first document uses bound variable x but it preserves the same semantics compared to the query, except for the symbol of that bond variable is different with that of query. The second one does not have bound variable match, in another word, it uses different symbols (i.e. "$x,\ y$") at positions where query expression uses the same symbols (i.e. "a").

One common pattern the first example follows is all "ranking" documents in the first example have the bound variable match, the only difference is the number of symbol matches. On the other hand, all "ranking" documents in the second example have the same number of symbol matches (only "b" is matched in a symbolic way). So it is easy to follow only one of the two constrain. However, sometimes both constrains can be applied so that conflict may occur and we have to choose only one to follow. For example, given document expression $\sqrt{a}(x-b)$ and $\sqrt{x}(x-b)$, the former has two symbolic matches (i.e. "$a,\ b$") with our query $\sqrt{a}(a-b)$, while it does not have bound variable match. The latter, on the other hand, has bound variable match while only has one symbolic match (i.e. "b"). We nevertheless score the latter higher because it does not lose any semantics.

3.3.2 The Mark-and-Cross Algorithm

Here we propose an algorithm to score symbolic similarity between query and document expressions. To follow all the constrains and issues addressed, intuitively, we first take the bond variable with greatest number of occurrence in query expression, try to match as much as possible with each bond variable from document expression. The *best-matching* bond variable in document expression is chosen to contribute to the

final symbolic relevance score (proportionally to the number of matches in that bond variable), and we exclude its paths from matching query paths in future iterations. In the next iteration, we choose the bond variable with the second number of occurrence in query expression and repeat this process until all the query bond variables are iterated.

The algorithm is described in figure 3.5. It takes three arguments, the set of leaf-root paths D and Q in document expression and query expression respectively, and the candidate sets C associated with all leaf-root paths in query. The bound variables in D is defined by a set $V(D) = \{x \mid \mathcal{S}(x),\ x \in D\}$, which contains all the leaf node symbols from document expression. Procedure *sortBySymbolAndOccur* takes a set of leaf-root paths and returns a list of all the leaf-root paths. The list is sorted by tuple $\big(N_p, \mathcal{S}(p)\big)$ for list element p, where N_p is the number of $\mathcal{S}(p)$ occurred in all the list path symbols. Take the example expression $\left(b \cdot \dfrac{a+b}{a+c} + a\right)$ again, the result list returned by *sortBySymbolAndOccur* is a sequence of leaf-root path of a, a, a, b, b, c respectively.

Each document path a' is associated with a tag $T_{a'}$ which has three possible states: marked, unmarked and crossed. And bond variable $v \in \mathcal{V}(D)$ can be given a score B_v which represents the similarity degree between current evaluating query/document bond variables. The function $\mathrm{sim}(a, a')$ measures the symbolic similarity degree between two leaf-root paths a and a'.

Intuitively, we set the similarity function

$$\mathrm{sim}(a, a') = \begin{cases} 1 & \text{if } \mathcal{S}(a) = \mathcal{S}(a') \\ \\ \alpha < 1 & \text{otherwise} \end{cases}$$

to give more similarity weights to leaf-root paths with exact symbol match. (this function $\mathrm{sim}(a, a')$ is used only when a document path is in candidate set, see figure 3.5).

Let us determine the proper value for α. Consider the conflicting cases stated in this section by using another example here, given query expression $a + \frac{1}{a} + \sqrt{a}$ and

```
1: procedure markAndCross(D, Q, C)
2:     score := 0
3:     if D = ∅ then
4:         return 0
5:     for a′ ∈ D do
6:         T_{a′} := unmark
7:     for v ∈ 𝒱(D) do
8:         B_v := 0
9:     QList := sortByOccurAndSymbol(Q)
10:    for a in QList do
11:        for v ∈ 𝒱(D) do
12:            m := −∞
13:            m_p := ∅
14:            for a′ ∈ C_a ∩ {y | 𝒮(y) = v, y ∈ D} do
15:                if T_{a′} = unmark and sim(a, a′) > m then
16:                    m := sim(a, a′)
17:                    m_p := a′
18:            if m_p ≠ ∅ then
19:                T_{m_p} := mark
20:                B_v := B_v + m
21:            else                                  ▷ Exhausted all candidates
22:                return 0
23:        if 𝒮(a) changed or last iteration of a then
24:            m := −∞
25:            m_v := ∅
26:            for v ∈ 𝒱(D) do
27:                if B_v > m then
28:                    m := B_v
29:                    m_v := v
30:                B_v := 0
31:            score := score + m
32:            for v ∈ 𝒱(D) do
33:                if v = m_v then
34:                    nextState := unmark
35:                else
36:                    nextState := cross
37:                for a′ ∈ C_a ∩ {y | 𝒮(y) = v, y ∈ D} do
38:                    if T_{a′} = mark then
39:                        T_{a′} := nextState
40:    return score
```

Figure 3.5: The mark-and-cross algorithm

document expression $a + \frac{1}{a} + b + \frac{1}{b} + \sqrt{b}$, we consider bond-variable matching

$$\boxed{\text{a}} + \frac{1}{\boxed{\text{a}}} + \sqrt{\boxed{\text{a}}}$$

with

$$a + \frac{1}{a} + \boxed{\text{b}} + \frac{1}{\boxed{\text{b}}} + \sqrt{\boxed{\text{b}}}$$

weighted more than exact symbol matching

$$\boxed{\text{a}} + \frac{1}{\boxed{\text{a}}} + \sqrt{a}$$

with

$$\boxed{\text{a}} + \frac{1}{\boxed{\text{a}}} + b + \frac{1}{b} + \sqrt{b}$$

(expressions surrounded by a box here indicates the matching part)

Because the former matching has more variables involved even if they are not identical symbolic matches compared with its counterpart of the latter. That is to say, given a document bond-variable matching k variables with those in query, we need α to satisfy $k\alpha > (k-1) \times 1 = k - 1$ and $\alpha < 1$. Therefore, in our practice, we set α to a value close to 1, e.g. 0.9.

By sorting the query paths in Q, the algorithm is able to take out paths of the same bond variables in maximum-occurrence-first order from QList. Each query path a tries to match a path a' in each document bond variable v by selecting the unmarked path m_p with maximum $\text{sim}(a, a')$ value, and accumulate this value on B_v which indicates the similarity between currently evaluating query bond variable and the bond variable v in document expression. In addition, mark the tag T_{m_p} associated with the document path m_p. Once a query bond variable has been iterated completely (line 23), let the document bond variable m_v with greatest B_v value m, be the best-matching bond variable in $\mathcal{V}(D)$ with the query bond variable just iterated, then add m to the score. Before iterating a new query bond variable, we will cross all the document paths of variable m_v to indicate they are confirmed been matched, and unmark the tags of those marked paths that are not variable m_v. We continue doing so until all

the query paths are iterated, finally return the score indicating the symbolic similarity between the two expressions.

3.4 Combine The Two

We have already discussed the problem and proposed the algorithms to both test structure isomorphic and measure symbolic similarity between mathematical expressions. The two algorithms, however, do not cooperate in an unified way. To illustrate this point, assume we first use DECOMPOSE-AND-MATCH algorithm and have concluded one is a formula subtree of another, but we only get one possible substructure match. There are very likely other possible positions where this tree can also be formula subtree of another, because the candidate sets are not unique (or $|\Phi| > 1$). Thus we need to exhaust all possibilities and apply MARK-AND-CROSS algorithm to each of them, in order to find the maximum symbolic similarity pair. Analogously, assume we first want to get the symbolic similarity degree, then we are uncertain about the paths we have specified in candidate set are really isomorphic paths to the query path, i.e. we will not guarantee substructure relation when using DECOMPOSE-AND-MATCH algorithm.

One possible way to both test strict structure isomorphism and measure symbolic similarity is to decompose the tree and try to match all the substructures but at the same time heuristically choose reference paths to find the best simbolic match in a greedy way, if no possible substructure can be matched isomorphically, we have to rollback and try other candidates using backtracking. From efficiency perspective, the MARK-AND-CROSS algorithm already has an worst case time complexity of $O\big(|Q|\times|D|\big)$, if we try to find the best match (find the one with not only the most symbolic similarity but also who satisfies structure isomorphism), then it will eventually lead to more time complexity.

3.4.1 Relaxed Structure Match

The complexity introduced to combine the two methods will make our approach infeasible to efficiently deal with large data set. Here we choose to relax our constrain

on strict structure isomorphism. As figure 3.2 has illustrated, we know that the labels of a leaf-node path set being a subset of another does not necessarily mean the tree generating the former path set is a formula subtree of that generating the latter. To further generalize it, we say for any two formula tree T_q and T_d and $\forall\ \hat{a} \in \mathbf{P}$, if $\ell(g(T_q)) \cdot \hat{a} \subseteq \ell(g(T_d))$, it is not sufficient to imply $T_q \preceq_l T_d$. Nevertheless, we think the cases which makes the above statement insufficient are fairly rare in common mathematical content, and the complexity introduced from considering those cases will offset the benefit to strictly identify the structure isomorphism. Therefore, we loose our constrain on structure similarity so that any $T_d \in \bigcap_{a \in L} \mathcal{I}_\Pi(a)$ is considered structurally relevant to query formula tree T_q in section 3.2.2, and we say T_d is *searchable* by T_q in index Π. Optionally, we can apply constrain 3.1 in observation #1 to further eliminate cases such as the one in figure 3.2 so that less query/document expressions which are not in formula subtree relation would be considered relevant. We name this constrain as *fan-number constrain*.

The revised method will collect all the "structurally similar" document formula trees that satisfy the fan-number constrain in the set $\bigcap_{a \in L} \mathcal{I}_\Pi(a)$. Then use MARK-AND-CROSS algorithm to rank their symbolic similarity.

3.4.2 Matching-Depth

On the other hand, we introduce a *matching depth* $d = |\hat{a}|$ into symbolic similarity measurement algorithm, to make it also consider the depth of a structural match. As it is addressed in [11], the deeper level at where two formula matches, the lower similarity weight would it be, since the deeper sub-formulae in in mathematical expression will make it less important to the overall formula. For example, given query formula $\sqrt{a}$, expression $\sqrt{x}$ would score higher than $\sqrt{\sqrt{x}}$ does. To reflect the impact on score, we define *matching depth factor* $f(d)$ to be a function value where f is in negative correlation with matching depth: $f(d) = 1/(1+d)$.

3.4.3 Matching-Ratio

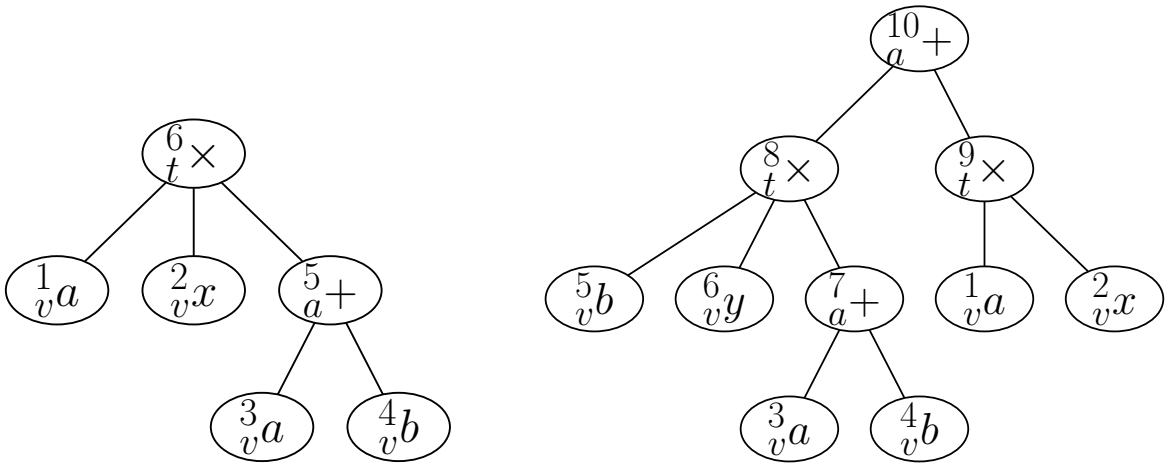
We may also need a way to evaluate how many percentage is a query matching a document expression. According to the property 3.3 in section 3.2.3 (observation #1), for two expression in formula subtree relation, we have $\frac{|g(T_q)|}{|g(T_d)|} \leq 1$, we name the ratio of left-hand as *matching-ratio*, which characterises the structural coverage for a matching query in an expression. Consider the scenario where a query expression is structural isomorphic to two document expressions, and the symbolic similarity between them are the same. However, the different "area" that the query matches to the two document expressions essentially makes them be scored differently. For example, for query $ax+b$, document expression $ax + b$ should precede $x^2 + ax + b$ although the query matches both two document expressions equally in terms of symbol.

3.4.4 Final Ranking Schema

Before matching-depth and matching-ratio is introduced, if a formula subtree relation is inferred between two formula trees, they are considered structurally matching in a boolean manner. There is no similarity degree between a complete match (or a sub-formula) and structurally irrelevant. In another word, structural differences is considered in a way to filter indexed expressions, for further symbolic similarity assessment. The final output ranking is determined by symbolic similarity degree, without considering structural similarity. Matching-depth and matching-ratio are metres introduced to measure the structural similarity, in the end we need to combine them with symbolic similarity to rank search results. Denote symbolic similarity score calculated by MARK-AND-CROSS algorithm to be s, and $f(d)$ being the matching depth factor, r being the matching ratio for a pair of query/document. We will use them together in a tuple $(s, f(d), r)$, to indicate the overall similarity and rank items in search results,

3.4.5 Illustrated by an Example

We illustrate the method introduced in this chapter by a simple example here. Given a query expression $ax(a + b)$ and document expression $ax + (b + a)by$, here we



query expression $ax(a+b)$ document expression $ax+(b+a)by$

Figure 3.6: Example query/document expression representation

show how we search the relevant document using this query and how the relevance score between them is calculated by mark-and-cross algorithm.

The query expression and document expression are represented by operation trees T_q and T_d in figure 3.6. Instead of only denoting the operation symbols at the internal nodes and the variable/constant symbols at the leaf nodes, we use the notation ${}^i_l S$ or ${}^i S$ to denote a node with symbol S labeled by l (i.e. $\ell(S) = l$) with vertex number i. The three different possible labels used here are v, t and a, standing for unified name "variable", "times" and "add" respectively. To be concise and descriptive, we will interchangeably use either ${}^i S$ notation or q_i (or p_i) to represent the leaf-root path in a query (or document) operation tree where the leaf vertex i resides.

Firstly, the generated path sets for T_q and T_d are:

$$g(T_q) = \{1 \cdot 6,\ 2 \cdot 6,\ 3 \cdot 5 \cdot 6,\ 4 \cdot 5 \cdot 6\}$$

$$g(T_d) = \{5 \cdot 8 \cdot 10,\ 6 \cdot 8 \cdot 10,\ 3 \cdot 7 \cdot 8 \cdot 10,\ 4 \cdot 7 \cdot 8 \cdot 10,\ 1 \cdot 9 \cdot 10,\ 2 \cdot 9 \cdot 10\}$$

And the labeled path sets for each of the two are:

$$\ell\left(g(T_q)\right) = \{v \cdot t,\ v \cdot a \cdot t\}$$

$$\ell\left(g(T_d)\right) = \{v \cdot t \cdot a,\ v \cdot a \cdot t \cdot a\}$$

Because $\ell\left(g(T_q)\right) \cdot a = \ell\left(g(T_d)\right)$, which follows

$$\ell(g(T_q)) \cdot \hat{a} \subseteq \ell(g(T_d))$$

where $\hat{a} = a$, we know T_d is searchable by T_q, so there exists a path $\hat{a}$ we can search to append after every labeled path in set $\ell\left(g(T_d)\right)$ so that we will find T_d by intersecting all the formula trees indexed (in index Π) in these paths.

Secondly, by the implications from observation #1 of section 3.2.3, we get the candidate set for each of the path in T_q:

$$C_{q_1} = \{p_5,\ p_6\}$$

$$C_{q_2} = \{p_5,\ p_6\}$$

$$C_{q_3} = \{p_3,\ p_4\}$$

$$C_{q_4} = \{p_3,\ p_4\}$$

In addition, get the list L containing all the query paths in T_q sorted by symbol and its occurrence in all path symbols,

$$\text{QList} = q_1,\ q_3,\ q_4,\ q_2.$$

Lastly, we can calculate the symbolic similarity degree between the two expressions by going through each query path from QList in order and apply mark-and-cross algorithm (use $\alpha = 0.9$). Then the first two iterations which calculates the matching score for bound variable a in T_q can be illustrated by table 3.1.

Each path of query bound variable a is compared with that from document path in its candidate set, and each resulting $\text{sim}(a, a')$ value is accumulated on the corresponding document bound variable B_v. Then the current score is also calculated, by accumulating the max B_v value among all the bound variable v in document expression. The tags associated with document path ${}^4b$ and ${}^5b$ are marked as crossed

current score	0.9			
bound variable	$B_a = 1$	$B_b = 1.8$ (max)		$B_y = 0.9$
document path / **query path**	${}^{3}a$	${}^{4}b$	${}^{5}b$	${}^{6}y$
${}^{1}a$			0.9	0.9
${}^{3}a$	1	0.9		

Table 3.1: First two iterations of example score evaluation

current score	$0.9 + 0.9 = 1.8$			
bound variable	$B_a = 0.9$ (max)	$B_b = 0$		$B_y = 0$
document path / **query path**	${}^{3}a$	${}^{4}b$	${}^{5}b$	${}^{6}y$
${}^{1}a$			0.9	0.9
${}^{3}a$	1	0.9		
${}^{4}b$	0.9			

Table 3.2: 3rd iteration of example score evaluation

current score	$1.8 + 0.9 = 2.7$			
bound variable	$B_a = 0$	$B_b = 0$		$B_y = 0.9$ (max)
document path / **query path**	${}^{3}a$	${}^{4}b$	${}^{5}b$	${}^{6}y$
${}^{1}a$			0.9	0.9
${}^{3}a$	1	0.9		
${}^{4}b$	0.9			
${}^{2}x$				0.9

Table 3.3: 4th iteration of example score evaluation

state, to prevent path ${}^{4}b$ and ${}^{5}b$ from being compared in future iterations. For the next query bond variable b in QList, first zero every B_v value, then calculate its matching score with the document paths in its candidate sets in a similar manner except we have skipped some document paths in candidate sets because they are crossed in the previous iteration. Table 3.2 shows the intermediate results before iterate to the next query bond variable in QList. Finally, we use the same way to process the last query bond variable x. Its query path ${}^{2}x$ is matched with document path ${}^{6}y$ with $\text{sim}({}^{2}x,\ {}^{6}y)$ value equals to 0.9. Then the only non-zero B_v value from document bound variable y will contribute to final symbolic similarity score which in the end is added up to 2.7 (see table 3.3).

Now we have finished our symbolic similarity evaluation between given query expression and document expression. We can also infer that the matching-depth d is $|\hat{a}| = 1$, matching factor $f(d) = \dfrac{1}{1+d} = 0.5$, and the matching-ratio $r = \dfrac{|g(T_q)|}{|g(T_d)|} = \dfrac{4}{6} \approx 0.67$. The final score tuple $(2.7,\ 0.5,\ 0.67)$ is used to determine the similarity degree in general and rank the document expression.

Chapter 4

IMPLEMENTATION AND EVALUATION

Based on the method we propose, we have implemented a search engine as well as a crawler and a parser for LaTeX content. Also, a demo [1] with Web interface is available for demonstration. In this chapter, we will introduce the way we implement our proposed method and use this system to evaluate its effectiveness and efficiency.

4.1 System Overview

Our system is consisted of a crawler, a parser, an indexer, a search program back-end and a Web front-end. The crawler searches and extracts math mode LaTeX from Website that directly returns LaTeX markup (e.g. from math content websites using client-side rendering, e.g. MathJax). The parser parses LaTeX markup and prints the generated operation tree. The indexer uses parser to transform LaTeX markup into operation tree and indexes sub-paths on disk. The search program searches a query in index and returns search results, also exports a library for Web front-end which is a CGI program to interactive with user.

The outline of our system process is, crawl and convert math expressions into an in-memory tree structure, then write the leaf-root path labels along with degree number associated with each nodes to disk as our index. When a user uses our Web front end to search, the Web CGI program calls the search library to search and score similar expressions in index, the search library outputs ranking results and Web CGI program rewrite them in HTML.

[1] demo page: http://infolab.ece.udel.edu:8912/cowpie/

4.2 Implementation

Each component of our search engine is implemented as a separated module, the only way to interact with each module is by calling a library API that another module exposes, or by reading/writing to a plain text file. In this section, we will describe the implementation detail for each isolated module.

4.2.1 Crawler

The crawler is a python script using *BeautifulSoup* to crawl Web pages and extract math mode LaTeX. We have implemented a crawler specifically for the posts on Math Stack Exchange website, crawler can be specified to crawl posts by page or post ID. For a post, the crawler searches the pattern of a LaTeX inline or display text mode using regular expression. When a LaTeX mode has been extracted, it is stripped with line feed and saved as plain text file one text mode per line and one file per post.

4.2.2 Parser

We are tokenizing math content using lexer generator *flex* and have implemented a LALR parser generated from a set of grammar rules in *GNU bison*, specifically for MathJax content in a subset of LaTeX (those related to mathematics). Our parser transforms a math formula into an in-memory operation tree (representing formula tree), as an intermediate step to extract the path labels, path ID, and degree numbers associated, for every leaf-root path from the tree. Our lexer omits all LaTeX control sequences not matching any pattern of our defined tokens, most of them are considered unrelated to math formula semantics (environment statement, color, mbox etc.).

The complete grammar rules we use to parse LaTeX math mode text is listed in Appendix A. The start rule *doc* represents a LaTeX math mode (either inline or display mode). Sub-rules (lower case names in grammar rules) handle mathematical patterns such as addition/subtraction, modular, relation (e.g. equality, greater than, less than), factor, script, brackets etc. Tokens are using upper case names, the *atom* rule represents a series of tokens that are mostly used as an atom unit in mathematical

```
‘--(ADD, ADD, #14, *2, @0)
    ‘--(NEG, NEG, #7, *1, @0)
    |   ‘--[‘b’, VAR, #1, *1, @0]
    ‘--(SQRT, SQRT, #13, *1, @1)
        ‘-(ADD, ADD, #12, *2, @0)
            |--(HANGER, HANGER, #9, *2, @0)
            |   |--(SUP_SCRIPT, SUP_SCRIPT, #8, *1, @0)
            |   |   ‘-[‘2’, NUM, #2, *1, @0]
            |   ‘-[‘b’, VAR, #3, *1, @1]
            ‘-(NEG, NEG, #11, *1, @1)
                ‘-(TIMES, TIMES, #10, *3, @0)
                    |--[‘4’, NUM, #4, *1, @0]
                    |--[‘a’, VAR, #5, *1, @1]
                    ‘-[‘c’, VAR, #6, *1, @2]
```

Figure 4.1: Example parser output

context (e.g. fraction, root, infinity). Other tokens are as part of some grammar rules. The partial list of lexer tokens for our parser are listed in Appendix B, the left field is escaped LaTeX command string, the right field in curly brackets is the token that the left command mapped to. Although enumerating LaTeX math mode commands results in a long lexer list, we get a very simple grammar rules that is enough to handle most of our crawled data.

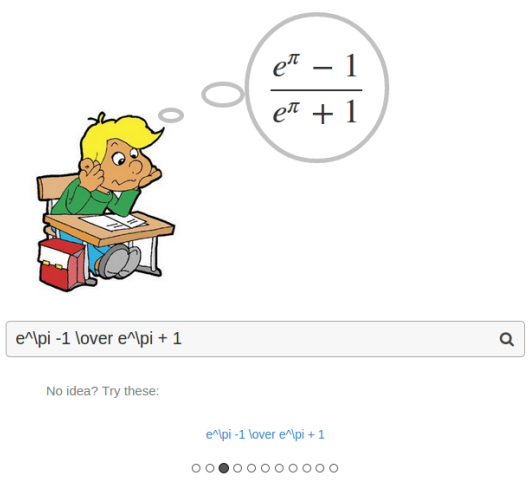
An example of an operation tree plain text output generated from our parser for math expression $-b \pm \sqrt{b^2 - 4ac}$ is shown by figure 4.1. Where every node is associated with five values: symbol value (meaningful only for leaf node), label value, node ID (indicated by “#”), degree of that node (indicated by “*”) and the rank of a node (indicated by “@”) in terms of its father node. The node ID for a leaf will be used to identify the leaf-root path with respect to that leaf node, and the rank of child node will be used to locate non-commutative tokens in a math expression. The indexer later will choose some of these information from nodes along a leaf-root path to index the tree.

4.2.3 Index

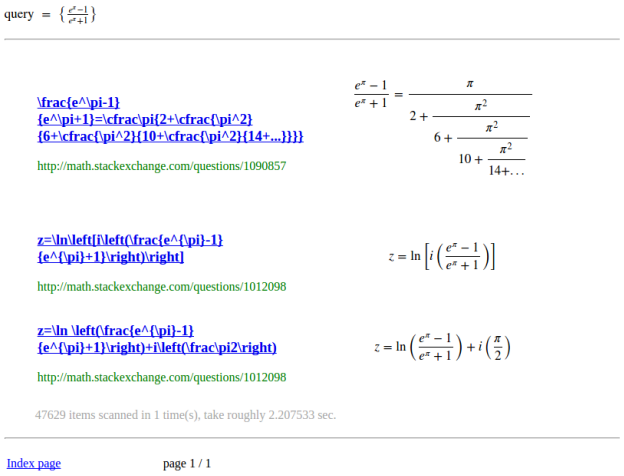
The indexer writes the information extracted from parser onto disk. There are two parts in our index, the first part uses native file system (for the sake of implementation simplicity) to store leaf-root path labels in directories from which our search engine can go level by level. Path ID, degree numbers, and also the formula ID of expression which generates that leaf-root path (we refer these three as *branch word*) are stored in a "posting" file at the directory corresponding to that branch word labels, e.g. the tree in figure 3.1 will result in indexing two directories: ./VAR/TIMES and ./VAR/ADD/TIMES. All the indexed branch words with path labels corresponding to directory ./VAR/TIMES/ are stored in the posting file of that directory, located at ./VAR/TIMES/posting.bin in this case. Branch words in a posting file are ordered by their formula IDs to speed search (so we can merge two posting files with size M and N respectively, then find the branch words with same formula ID in $O(M+N)$ time). The second part of our index is a key-value database (using *Kyoto Cabinet*) to map a formula ID to additional information for that formula (e.g. original markup, number of leaf-root paths $|T_d|$ and the URL on which the formula is crawled).

4.2.4 Search Program

The compromised substructure searching (described in section 3.2.2) is used in our search engine to filter out likely isomorphic expressions in our index. Searching is performed by simultaneously going from all the directories corresponding to the generated leaf-root paths of query formula tree, to all their merged subdirectories. We keep traversing, at each level intersecting the branch words (by their formula IDs) in the posting files from all the searching directories. The intersected formula IDs actually represent the trees in our search set $\bigcap_{a\in L}\mathcal{I}_\Pi(a)$. Every document formula spotted in the search set is considered as a hit, we then apply MARKANDCROSS algorithm to get its symbolic similarity score with query formula (using the branch word information indexed in posting file) . The search program uses our ranking tuple described in section 3.4.4 to indicate the overall similarity and rank items in search results, i.e. to



Search page



Results page

Figure 4.2: Web front-end

decide whether one should be ranked higher than the other, first compare s, if equal, compare $f(d)$ and then r. In addition, we use a min-heap to keep the top-k scored items in our search results (by replacing the lowest scored items if we find a newer hit with higher score), where k is the maximum number of items we keep in search results. We also place an early termination valve on the number of branch words can be searched for one query at a time, so that when exceeding this limit, search engine will stop and return the search results it has up to that time. Because some query can potentially have a very long posting list, doing so would make our searching response time no more than a certain value.

4.2.5 Web Front-End

The Web front-end presents user a Web page (search page) that accepts a query in an input box and returns search results in several Web pages (result pages). User query is sent from browser to the front-end program by HTTP POST method and the start page of all ranked hits is specified by HTTP GET request. The front end CGI program calls search library and wraps search results in HTML list elements, at the

ID	formula	ID	formula
1	$\int_0^\infty dx \int_x^\infty F(x,y)dy = \int_0^\infty dy \int_0^y F(x,y)dx$	2	$X(i\omega)$
3	$x^n + y^n = z^n$	4	$\int_{-\infty}^{\infty} e^{-x^2} dx$
5	$\frac{f(x+h)-f(x)}{h}$	6	$\frac{\sin x}{x}$
7	$ax^2 + bx + c$	8	$\frac{e^x+y}{z}$
9	$O(n \log n)$	10	$H^n(X) = Z^n(X)/B^n(X)$
11	$A_n = \frac{1}{\pi} \int_{-\pi}^{\pi} F(x) \cos(nx)dx$	12	$\lim_{x\to\infty}(1 + \frac{1}{x})^x$
13	$f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \ldots$	14	$f(a) = \frac{1}{2\pi i} \oint_r \frac{f(z)}{z-a}\, \mathrm{d}z$
15	$x^2 + 2xy + y^2 = \lvert x\rvert^2 + 2\lvert x\rvert\lvert y\rvert + \lvert y\rvert^2$	16	$\int_a^b f(x)\, \mathrm{d}x = F(b) - F(a)$
17	$\frac{n!}{r_1!\cdot r_2!\cdots r_k!}$	18	$-b \pm \sqrt{b^2 - 4ac}$
19	$1 + \tan^2\theta = \sec^2\theta$	20	$\bar{u} = (x, y, z)$

Table 4.1: Test query set

same time creates navigation links to help user browse results by page. Figure 4.2 shows a screenshot of the search page and a result page from Web front-end.

4.3 Evaluation

We rely on the search engine described in previous section to evaluate our search methods proposed in this paper. Yet retrieval based on an equation is a relatively new task, previously studied in the context of NTCIR Math Tasks. The criteria to give judges about assessing the level of similarity degree between math expressions is not in a high level of agreement. Nevertheless, we choose to evaluate our system by comparison to a naive baseline, and provide our intuitive guidelines for relevance judgements.

4.3.1 Dataset and Index

Our own dataset is created to evaluate proposed method. We have crawled LaTeX content from the posts of nearly entire (27180 pages of questions) Math Stack Exchange website before March 2015. The data set [2] is plain text files generated by crawler output (LaTeX math mode content per line, one file for each post). Over 8 million expressions of math mode are contained in the data set. The dataset is available through a roughly 60MB *bzip2* compressed file.

Our test query set [3] consists queries mostly from [2] and [6], some of them are excluded here because we are unable to find similar formula in our own dataset. Table 4.1 shows our complete test queries used in our evaluation.

The popular evaluation dataset in this research domain, the NTCIR Math Task collection, is in MathML/XML format, and original LaTeX information is not always preserved in their dataset. Here we choose not to use their dataset because we are parsing LaTeX directly. Although converting MathML/XML formula to LaTeX is possible, we fail to convert all the document in NTCIR dataset correctly (using *pandoc*). Furthermore, supporting wildcard query is a default requirement in NTCIR Math Task, while our approach does not support wildcard (see section 3.3).

After parsing our dataset using the indexer, the resulting index includes an 892 MB key-value database file and roughly 9.4GB directories and posting files.

4.3.2 Relevance Judgement

We have defined four relevance levels, scored from 0 to 4 in our evaluation experiment. This criteria considers both structural similarity and symbolic similarity. Structural similarity is scored by either 0, 1 (mostly similar) or 2 (complete matching); symbolic similarity is scored by 0, 1 (mostly identical symbols for the matching parts) or 2 (identical symbols for the matching parts). The overall level of relevance is simply

[2] raw data: http://infolab.ece.udel.edu:8912/cowpie/raw.tar.bz2

[3] query set: http://infolab.ece.udel.edu:8912/cowpie/test-queries.tex

Relevance Score		Structurally		
		Low similarity	High similarity	Complete match
Symbolically	Low similarity	0	1	2
	High similarity	1	2	3
	Complete match	2	3	4

Table 4.2: Judgement score table

the sum of the two scores. Table 4.2 lists all possible values for the overall relevance score in each case.

4.3.3 Results

We have evaluated two methods here. The baseline method is a boolean search for structural related document using the method in section 3.2.2. We compare the baseline method with the more comprehensive search method proposed in this paper, which further considers symbolic similarity score, matching depth factor and matching ratio as described in section 3.4.

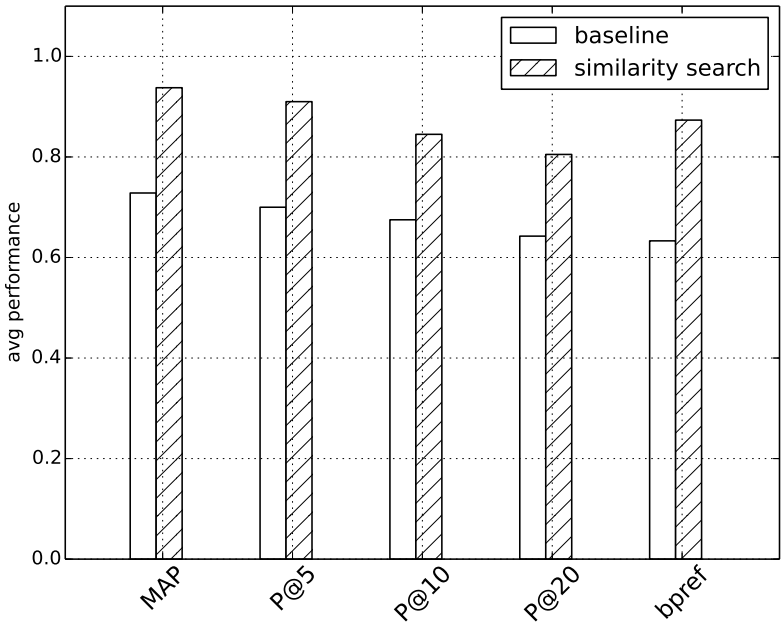
Table 4.3 shows the distribution of hits and relevance judgement score of top 20 results (or less, if the number of hits are fewer than 20) for each test query listed in table 4.1. The relevance scores directly reflect the effectiveness of our methods. figure 4.3 has visualized the relevance level of our evaluation. On the other hand, we also have done a performance evaluation on the average query look up time for each test query. The efficiency performance comparison of these two methods is shown in figure 4.4 (The query look-up time is the time consumed for searching directories and posting files, this is the majority of time consumption for the query look-up in our implementation).

Results show that our symbolic similarity measurement, if used, can boost search effectiveness in all the five metres evaluated, and consumes a reasonable extra time on top of the baseline method.

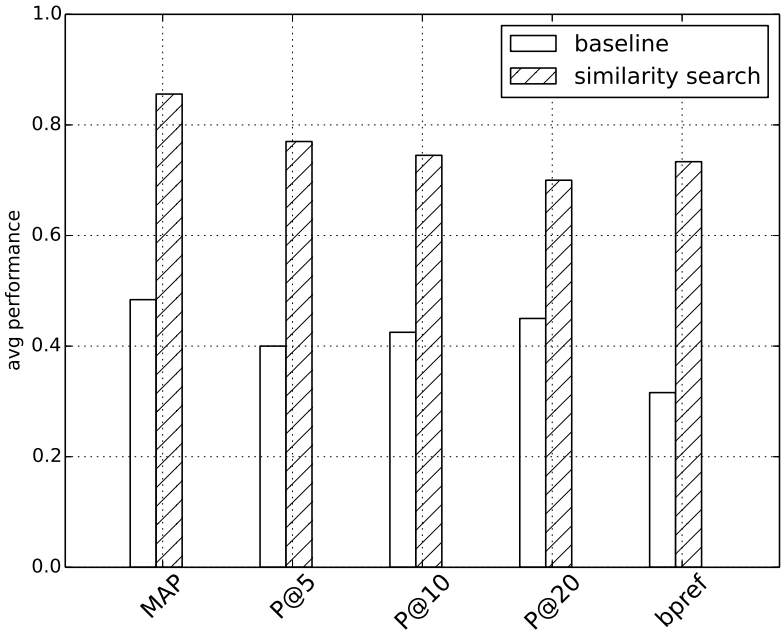
Query	Relevance Score					Judged
	0	1	2	3	4	
1	15	2	2	0	1	20
2	20	0	0	0	0	20
3	15	4	1	0	0	20
4	0	0	0	6	14	20
5	0	0	0	8	12	20
6	0	2	5	2	11	20
7	1	4	3	3	9	20
8	4	2	13	1	0	20
9	17	2	1	0	0	20
10	5	1	1	1	0	8
11	0	0	4	11	5	20
12	0	0	1	16	3	20
13	0	0	0	1	6	7
14	0	0	4	13	3	20
15	14	3	1	1	1	20
16	0	2	5	8	5	20
17	0	0	0	15	5	20
18	8	6	2	2	2	20
19	0	0	5	13	2	20
20	19	1	0	0	0	20

Query	Relevance Score					Judged
	0	1	2	3	4	
1	13	2	2	1	1	19
2	15	1	0	3	1	20
3	0	0	0	0	20	20
4	0	0	0	1	19	20
5	0	0	0	0	20	20
6	1	0	0	0	19	20
7	0	0	0	0	20	20
8	4	3	12	1	0	20
9	0	0	0	0	20	20
10	4	1	1	1	0	7
11	0	0	4	11	5	20
12	0	0	0	9	11	20
13	0	0	0	1	6	7
14	0	0	9	6	5	20
15	14	2	1	1	2	20
16	0	0	0	0	20	20
17	0	0	0	14	6	20
18	0	0	2	0	18	20
19	0	0	0	0	20	20
20	0	0	2	7	11	20

Table 4.3: Relevance score distribution



Partial relevance (relevance score ≥ 1)



Relevance (relevance score ≥ 3)

Figure 4.3: Effectiveness performance

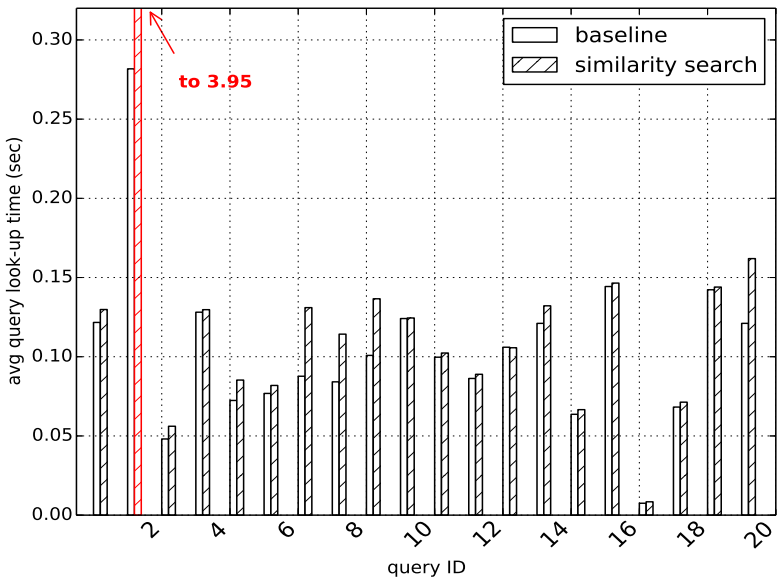


Figure 4.4: Efficiency performance

Chapter 5

CONCLUSION AND FUTURE WORK

We try to advance the search method for math expression using sub-paths. This is a structure-based approach (as described in section 2.2) to search and assess mathematical expression similarity. Without using vector-space model as opposed to its presence on text-based approach, our method has tried to further develop structure-based method and offer some new insights for future research, and the evaluation results already indicate the feasibility of our methods. One of our hypothesis is structure-based method better captures mathematical expression semantics. We believe there is large potential to further refine our novel similarity-search method as our presented attempt to improve mathematical-similarity search has demonstrated some promising aspects of our method, and lured some new ideas to improve our method.

5.1 Summary and Conclusion

Our method tries to measure math-expression similarity from the perspective of their structures and operand symbols. We search structurally relevant expressions in a subset of index and our evaluation results have demonstrated the overall cost-effectiveness of adding symbolic similarity-search method on proposed structural similarity-search. This paper's main contribution includes, first gives definitions and algorithms for evaluating the similarity between two math expressions to allow a math-aware search system to rank candidate expressions against a query expression; second gives a novel search method to reduce the search set of structurally similar expressions given a query. By implementing those ideas and conducting experiments on our dataset, we have offered insights into a new way to perform matching of structured content in general. Given the newness of the domain, we have showed an appealing

new direction to handle math expression search from an angel of both symbolic and structural similarity.

5.2 Current Problems

We expect our index to be less redundant because our method does not need augmentation to capture the same semantical meanings between different alternative forms of a math expression, while it is observed that our index size is still relatively large (9.4GB) compared to the size of input raw data (around 60MB). There are presumably two reasons for this: Firstly we are implementing our index based on file system directories for the ease of implementation, the overhead for constructing directories in file system cannot be overlooked. Secondly, the 60MB raw data are all math formula in LaTeX, in contrast to a general text search engine, whose raw data usually contains the entire document text, our raw data is comparatively more data-intensive, and carries more information than general text. Moreover, we do not use any kind of compression in our index. Another observation is, the absolute value of our query lookup time is not ideal, especially the first-time querying takes much longer time (because cache miss) than the time of subsequent same queries. Furthermore, queries with simple decomposed sub-paths take relative much longer time to search (e.g. query number 2 takes almost 4 seconds in average).

5.3 Future Work

In the next stage, trying to reduce the overhead for searching the index can be a good way to fix our bottleneck on both improving storage space efficiency and speeding up query look-up process. We may try to index the data into a single large file (e.g. implement a B-tree like file structure to store sub-paths) instead of small files spread over directories, to lower the system overhead in storage and take advantage of cache locality. It maybe also a good idea to try compressing the index and saving less information on disk for a branch word to further reduce the storage demand for our index. Nevertheless, Considering the fact that our system is newly created, we

have already met the goal to test our new ideas on concerned domain, despite the less usability compared to already mature existing tools for general text search.

Also, the experiments we have is not based on popular NTCIR dataset in this research domain, because of the problem we mentioned in section 4.3.1. This leaves a comparison of our method with mainstream math information retrieval methods to be desired. In the future, we may try to create a new parser for MathML content to generate operation tree defined in this paper, so that we are able to give experiments on some standard dataset of math IR, with larger data size and more testing queries, in order to provide more comparable results on evaluation.

Other than the problems addressed above, it is also desired to integrate general text search ability into our math-only search method. Additionally, the manner we use to break math formula into branch words and to index them through posting list makes it easy to parallelize and distribute the searching process, which means there is a large potential for future efforts to improve the efficiency of this method.

REFERENCES

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

[2] Topics for the ntcir-10 math task full-text search queries. http://ntcir-math.nii.ac.jp/wp-content/blogs.dir/13/files/2014/02/NTCIR10-math-topics.pdf. Accessed: 2015-03-31.

[3] Akiko Aizawa, Michael Kohlhase, and Iadh Ounis. Ntcir-11 math-2 task overview. *The 11th NTCIR Conference*, 2014.

[4] Jozef Misutka. Mathematical search engine. Master's thesis, Charles University in Prague, May 2013.

[5] Xuan Hu, Liangcai Gao, Xiaoyan Lin, Zhi Tang, Xiaofan Lin, and Josef B. Baker. Wikimirs: A mathematical information retrieval system for wikipedia. *Proceedings of the 13th ACM/IEEE-CS joint conference on Digital libraries. Pages 11-20*, 2013.

[6] David Stalnaker and Richard Zanibbi. Math expression retrieval using an inverted index over symbol pairs in math expressions: The tangent math search engine at ntcir 2014. *Proc. SPIE 9402, Document Recognition and Retrieval XXII, 940207*, 2015.

[7] Qun Zhang and Abdou Youssef. An approach to math-similarity search. *Intelligent Computer Mathematics. International Conference, CICM*, 2014.

[8] Miller B. and Youssef A. Technical aspects of the digital library of mathematical functions. *Annals of Mathematics and Artificial Intelligence 38(1-3), 121136*, 2003.

[9] Youssef A. Information search and retrieval of mathematical contents: Issues and methods. *The ISCA 14th Intl Conf. on Intelligent and Adaptive Systems and Software Engineering (IASSE 2005)*, 2005.

[10] Jozef Miutka and Leo Galambo. Extending full text search engine for mathematical content. *Towards Digital Mathematics Library.*, 2008.

[11] Petr Sojka and Martin Lka. Indexing and searching mathematics in digital libraries. *Intelligent Computer Mathematics*, 6824:228–243, 2011.

[12] Petr Sojka and Martin Lka. The art of mathematics retrieval. *ACM Conference on Document Engineering, DocEng 2011*, 2011.

[13] Martin Lka. Evaluation of mathematics retrieval. Master's thesis, Masarykova University, 2013.

[14] P. Pavan Kumar, Arun Agarwal, and Chakravarthy Bhagvati. A structure based approach for mathematical expression retrieval. *Multi-disciplinary Trends in Artificial Intelligence*, 7694:23–34, 2012.

[15] Robert Miner and Rajesh Munavalli. *An Approach to Mathematical Search Through Query Formulation and Data Normalization*. Springer Berlin Heidelberg, 2007.

[16] Christopher D. Manning, Prabhakar Paghavan, and Hinrich Schutze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[17] Muhammad Adeel, Hui Siu Cheung, and Ar Hayat Khiyal. Math go! prototype of a content based mathematical formula search engine, 2008.

[18] Andrea Asperti, Ferruccio Guidi, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. A content based mathematical search engine: whelp. In *In: Post-proceedings of the Types 2004 International Conference, Vol. 3839 of LNCS*, pages 17–32. Springer-Verlag, 2004.

[19] Michael Kohlhase and Ioan A. Sucan. A search engine for mathematical formulae. In *Proc. of Artificial Intelligence and Symbolic Computation, number 4120 in LNAI*, pages 241–253. Springer, 2006.

[20] Michael Kohlhase. Mathwebsearch 0.4 a semantic search engine for mathematics.

[21] Michael Kohlhase. Mathwebsearch 0.5: Scaling an open formula search engine.

[22] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (3rd Edition)*. Prentice Hall, December 2009.

[23] Peter Graf. *Term Indexing*. Springer Verlag, 1996.

[24] Yoshinori Hijikata, Hideki Hashimoto, and Shogo Nishida. An investigation of index formats for the search of mathml objects. In *Web Intelligence/IAT Workshops*, pages 244–248. IEEE, 2007.

[25] Yoshinori Hijikata, Hideki Hashimoto, and Shogo Nishida. Search mathematical formulas by mathematical formulas. *Human Interface and the Management of Information. Designing Information, Symposium on Human Interface*, pages 404–411, 2009.

[26] Hiroshi Ichikawa, Taiichi Hashimoto, Takenobu Tokunaga, and Hozumi Tanaka. New methods of retrieve sentences based on syntactic similarity. *IPSJ SIG Technical Reports, DBS-136, FI-79*, pages 39–46, 2005.

[27] Yokoi Keisuke and Aizawa Akiko. An approach to similarity search for mathematical expressions using mathml. *Towards a Digital Mathematics Library. Grand Bend, Ontario, Canada*, pages 27–35, 2009.

[28] Thomas Schellenberg, Bo Yuan, and Richard Zanibbi. Layout-based substitution tree indexing and retrieval for mathematical expressions. *Proc. SPIE 8297, Document Recognition and Retrieval XIX, 82970I*, 2012.

[29] David Stalnaker and Richard Zanibbi. Math expression retrieval using an inverted index over symbol pairs. *Proc. SPIE 9402, Document Recognition and Retrieval XXII, 940207*, 2015.

[30] Shahab Kamali and Frank Wm. Tompa. A new mathematics retrieval system. *CIKM*, 2010.

[31] Kai Ma, Siu Cheung Hui, and Kuiyu Chang. Feature extraction and clustering-based retrieval for mathematical formulas. In *Software Engineering and Data Mining (SEDM), 2010 2nd International Conference on*, pages 372–377, June 2010.

[32] Cyril Laitang, Mohand Boughanem, and Karen Pinel-Sauvagnat. Xml information retrieval through tree edit distance and structural summaries. In *Information Retrieval Technology*, volume 7097 of *Lecture Notes in Computer Science*, pages 73–83. Springer Berlin Heidelberg, 2011.

[33] Shahab Kamali and FrankWm. Tompa. Structural similarity search for mathematics retrieval. In *Intelligent Computer Mathematics*, volume 7961 of *Lecture Notes in Computer Science*, pages 246–262. Springer Berlin Heidelberg, 2013.

[34] Richard Zanibbi and Bo Yuan. Keyword and image-based retrieval for mathematical expressions. *Multi-disciplinary Trends in Artificial Intelligence. 6th International Workshop, MIWAI 2012.*, pages 23–34, 2011.

[35] Li Yu and Richard Zanibbi. Math spotting: Retrieving math in technical documents using handwritten query images. *Document Analysis and Recognition (ICDAR)*, pages 446 – 451, 2009.

[36] T. Nguyen, S. Hui, and K. Chang. A lattice-based approach for mathematical search using formal concept analysis. *Expert Systems with Applications*, 2012.

[37] Akiko Aizawa, Michael Kohlhase, Iadh Ounis, and Moritz Schubotz. Ntcir-11 math-2 task overview. *Proc. of the 11th NTCIR Conference, Tokyo, Japan*, 2014.

[38] Kamali Shahab and Tompa Frank Wm. Improving mathematics retrieval. *Towards a Digital Mathematics Library. Grand Bend, Ontario, Canada*, pages 37–48, 2009.

[39] PhilipN. Klein. Computing the edit-distance between unrooted ordered trees. volume 1461 of *Lecture Notes in Computer Science*, pages 91–102. Springer Berlin Heidelberg, 1998.

[40] J. Roger Hindley. *Introduction to Combinators and [Lambda]-Calculus*. Cambridge University Press, 1986.

Appendix A

GRAMMAR RULES OF PARSER

```
doc : tex '\n' ;

rel : atom | _REL ;

tex : NULL | term
| tex '+' term | tex '+'
| tex '-' term | tex '-'
| tex _REL tex
| tex _SEP_CLASS tex
| tex _ABOVE tex
| tex _OVER tex
| tex _MODULAR tex
| tex _CHOOSE tex
| tex _STACKREL atom rel tex
| tex _SET_REL atom rel tex
| tex _X_ARROW atom tex
| tex _X_ARROW '[' tex ']' atom tex ;

term : factor | term factor
| term _TIMES factor | term _DIV factor ;

factor : pack | factor '!' | factor script ;

pack : atom
| '(' tex ')' | '(' tex ']'
| '[' tex ')' | '[' tex ']'
| _LEFT_CEIL tex _RIGHT_CEIL
| _LEFT_FLOOR tex _RIGHT_FLOOR ;

script
: '_' atom | '^' atom
| '_' atom '^' atom
| '^' atom '_' atom ;
```

```
atom : NUM | ZERO | ONE | VAR
| FUN_CLASS | SUM_CLASS
| '{' tex '}'
| FRAC__ | _FRAC atom atom
| COMBIN__ | _COMBIN atom atom
| _VECT atom
| _SQRT atom | _SQRT '[' tex ']' atom | _ROOT atom _OF atom
| _PRIME_VAR
| _DOTS | _PARTIAL | _PI | _INFTY
| _EMPTY | _ANGLE | _PERP | _CIRC
| _PERCENT | _VERT ;
```

Appendix B

LEXER TOKENS OF PARSER

```
\times                                  { _TIMES }
\otimes                                 { _TIMES }
\ltimes                                 { _TIMES }
\rtimes                                 { _TIMES }
\cdot                                   { _TIMES }
\odot                                   { _TIMES }
\ast                                    { _TIMES }

\over                                    { _OVER }
/* above */
\above                                  { _ABOVE }
/* div */
/                                         { _DIV }
\div                                      { _DIV }
\divideontimes                            { _DIV }
/* frac */
\frac                                    { _FRAC }
\dfrac                                   { _FRAC }
\cfrac                                   { _FRAC }
\tfrac                                   { _FRAC }

\frac[ ]*[0-9][0-9]                      { FRAC_ }
\dfrac[ ]*[0-9][0-9]                     { FRAC_ }
\cfrac[ ]*[0-9][0-9]                     { FRAC_ }
\tfrac[ ]*[0-9][0-9]                     { FRAC_ }

\left[ ]*\\lceil                    { _LEFT_CEIL; }
\left[ ]*\\lfloor                   { _LEFT_FLOOR }
\right[ ]*\\rceil                   { _RIGHT_CEIL; }
\right[ ]*\\rfloor                  { _RIGHT_FLOOR }
/* shorter ceil and floor */
\lceil                              { _LEFT_CEIL }
\lfloor                             { _LEFT_FLOOR }
\rceil                              { _RIGHT_CEIL }
```

```
\rfloor                                 { _RIGHT_FLOOR }

"|"                                       { _VERT }
"\\|"                                     { _VERT }
\vert                                    { _VERT }
\Vert                                    { _VERT }
\[Aa]rrowvert                            { _VERT }
\bracevert                               { _VERT }
\r[vV]ert                                { _VERT }
\l[vV]ert                                { _VERT }
\mid                                     { _VERT }

\choose                                 { _CHOOSE }
\brack                                  { _CHOOSE }

\dbinom[ ]*[0-9][0-9]                   { COMBIN__ }
\tbinom[ ]*[0-9][0-9]                   { COMBIN__ }
\binom[ ]*[0-9][0-9]                    { COMBIN__ }
\dbinom                                 { _COMBIN }
\tbinom                                 { _COMBIN }
\binom                                  { _COMBIN }

 /* modular operation */

\\pmod                                  { _MODULAR }
\\bmod                                  { _MODULAR }
\\mod                                   { _MODULAR }

 /* prime */

"'"                                    { _PRIME_SUP }
\\prime                                { _PRIME_VAR }

 /* sqrt and root */

\\sqrt                                    { _SQRT }
\\root                                    { _ROOT }
\\of                                      { _OF;  }

 /* vector */

\\vec                                     { _VECT }
\\overleftarrow                           { _VECT }
```

```
\\overrightarrow                          { _VECT }

 /* partial */

\\partial                               { _PARTIAL }
\\nabla                                 { _PARTIAL }

 /* PI */

\\pi                                      { _PI }

 /* infty */

\\infty                                 { _INFTY }

 /* empty */

\\empty                                 { _EMPTY }
\\emptyset                              { _EMPTY }
\\varnothing                            { _EMPTY }

 /* angle */

\\triangle                              { _ANGLE }
\\triangledown                          { _ANGLE }
\\angle                                 { _ANGLE }
\\vartriangle                           { _ANGLE }
\\vartriangleleft                       { _ANGLE }
\\vartriangleright                      { _ANGLE }
\\triangleleft                          { _ANGLE }
\\triangleright                         { _ANGLE }
\\measuredangle                         { _ANGLE }
\\sphericalangle                        { _ANGLE }

 /* perpendicular */

\\perp                                   { _PERP }
\\bot                                    { _PERP }

 /* circle */

\\circ                                   { _CIRC }
```

```
 /* percentage */

\\%                                     { _PERCENT }

 /* dots */

\\dots                                    { _DOTS }
\.\.\.                                    { _DOTS }
\\ldots                                   { _DOTS }
\\vdots                                   { _DOTS }
\\cdots                                   { _DOTS }
\\ddots                                   { _DOTS }
\\ddot                                    { _DOTS }
\\dddot                                   { _DOTS }
\\ddddot                                  { _DOTS }
\\dotsb                                   { _DOTS }
\\dotsc                                   { _DOTS }
\\dotsi                                   { _DOTS }
\\dotsm                                   { _DOTS }
\\dotso                                   { _DOTS }
\\iddots                                  { _DOTS }

 /* equality class */

=                                          { _REL }
:=                                         { _REL }
"<"                                        { _REL }
">"                                        { _REL }
\\Cap                                      { _REL }
\\Cup                                      { _REL }
\\Join                                     { _REL }
\\Prec                                     { _REL }
\\Subset                                   { _REL }
\\Supset                                   { _REL }
\\[dD]oteq                                 { _REL }
\\and                                      { _REL }
\\approx                                   { _REL }
\\approxeq                                 { _REL }
\\asymp                                    { _REL }
\\backepsilon                              { _REL }
\\backsim                                  { _REL }
\\backsimeq                                { _REL }
\\bowtie                                   { _REL }
```

```
\\cap                                   { _REL }
\\circeq                                { _REL }
\\cong                                  { _REL }
\\cup                                   { _REL }
\\curlyeqprec                           { _REL }
\\curlyeqsucc                           { _REL }
\\curlyvee                              { _REL }
\\curlywedge                            { _REL }
\\dashv                                 { _REL }
\\dot=                                  { _REL }
\\eqslantgtr                            { _REL }
\\eqslantless                           { _REL }
\\equiv                                 { _REL }
\\ge                                    { _REL }
\\geq                                   { _REL }
\\geqq                                  { _REL }
\\geqslant                              { _REL }
\\gg                                    { _REL }
\\gnapprox                              { _REL }
\\gnsim                                 { _REL }
\\gt                                    { _REL }
\\gtrapprox                             { _REL }
\\gtrdot                                { _REL }
\\gtreqless                             { _REL }
\\gtreqqless                            { _REL }
\\gtrless                               { _REL }
\\gtrsim                                { _REL }
\\in                                    { _REL }
\\land                                  { _REL }
\\le                                    { _REL }
\\leadsto                               { _REL }
\\leq                                   { _REL }
\\leqq                                  { _REL }
\\leqslant                              { _REL }
\\lessapprox                            { _REL }
\\lessdot                               { _REL }
\\lesssim                               { _REL }
\\ll                                    { _REL }
\\lnapprox                              { _REL }
\\lneq                                  { _REL }
\\lneqq                                 { _REL }
\\lor                                   { _REL }
\\lt                                    { _REL }
```

```
\\lvertneqq                                { _REL }
\\ncong                                    { _REL }
\\ne                                       { _REL }
\\neq                                      { _REL }
\\ngeq                                     { _REL }
\\ngeqq                                    { _REL }
\\ngeqslant                                { _REL }
\\ni                                       { _REL }
\\nleq                                     { _REL }
\\nleqq                                    { _REL }
\\nleqslant                                { _REL }
\\nless                                    { _REL }
\\not("="|"\equiv"|"\in")                  { _REL }
\\nparallel                                { _REL }
\\nprec                                    { _REL }
\\npreceq                                  { _REL }
\\nsim                                     { _REL }
\\nsubseteq                                { _REL }
\\nsucc                                    { _REL }
\\nsucceq                                  { _REL }
\\nsupseteq                                { _REL }
\\owns                                     { _REL }
\\parallel                                 { _REL }
\\prec                                     { _REL }
\\preceq                                   { _REL }
\\propto                                   { _REL }
\\sim                                      { _REL }
\\simeq                                    { _REL }
\\sqcap                                    { _REL }
\\sqcup                                    { _REL }
\\sqsubset                                 { _REL }
\\sqsubseteq                               { _REL }
\\sqsupset                                 { _REL }
\\sqsupseteq                               { _REL }
\\subset                                   { _REL }
\\subseteq                                 { _REL }
\\subseteqq                                { _REL }
\\subsetneq                                { _REL }
\\subsetneqq                               { _REL }
\\succ                                     { _REL }
\\succapprox                               { _REL }
\\succcurlyeq                              { _REL }
\\succeq                                   { _REL }
```

```
\\succnapprox                                   { _REL }
\\succneqq                                      { _REL }
\\succnsim                                      { _REL }
\\succsim                                       { _REL }
\\supset                                        { _REL }
\\supseteq                                      { _REL }
\\supseteqq                                     { _REL }
\\supsetneq                                     { _REL }
\\supsetneqq                                    { _REL }
\\thickapprox                                   { _REL }
\\thicksim                                      { _REL }
\\trianglelefteq                                { _REL }
\\triangleq                                     { _REL }
\\triangleq                                     { _REL }
\\unlhd                                         { _REL }
\\unrhd                                         { _REL }
\\varsubsetneq                                  { _REL }
\\varsubsetneqq                                 { _REL }
\\varsupsetneq                                  { _REL }
\\varsupsetneqq                                 { _REL }
\\vee                                           { _REL }
\\veebar                                        { _REL }
\\wedge                                         { _REL }

 /* stack above operations */

\\xleftarrow                                    { _X_ARROW  }
\\xrightarrow                                   { _X_ARROW  }
\\stackrel                                      { _STACKREL }
\\buildrel                                      { _BUILDREL }
\\overset                                       { _SET_REL  }
\\underset                                      { _SET_REL  }

 /* seperation class */

 /* table related */
\\\\                                           { _SEP_CLASS }
\\cr                                           { _SEP_CLASS }
\\newline                                      { _SEP_CLASS }
\\:                                            { _SEP_CLASS }
\\>                                            { _SEP_CLASS }
\\enspace                                      { _SEP_CLASS }
 /* -- */
```

```
,                                   { _SEP_CLASS }
;                                   { _SEP_CLASS }
\colon                              { _SEP_CLASS }
":"                                 { _SEP_CLASS }
\\\&                                { _SEP_CLASS }
\\And                               { _SEP_CLASS }
\\Downarrow                         { _SEP_CLASS }
\\Leftarrow                         { _SEP_CLASS }
\\Leftrightarrow                    { _SEP_CLASS }
\\Lleftarrow                        { _SEP_CLASS }
\\Longleftarrow                     { _SEP_CLASS }
\\Longleftrightarrow                { _SEP_CLASS }
\\Longrightarrow                    { _SEP_CLASS }
\\Lsh                               { _SEP_CLASS }
\\Rightarrow                        { _SEP_CLASS }
\\Rrightarrow                       { _SEP_CLASS }
\\Rsh                               { _SEP_CLASS }
\\Uparrow                           { _SEP_CLASS }
\\Updownarrow                       { _SEP_CLASS }
\\atop                              { _SEP_CLASS }
\\between                           { _SEP_CLASS }
\\circlearrowleft                   { _SEP_CLASS }
\\circlearrowright                  { _SEP_CLASS }
\\curvearrowleft                    { _SEP_CLASS }
\\curvearrowright                   { _SEP_CLASS }
\\downarrow                         { _SEP_CLASS }
\\downdownarrows                    { _SEP_CLASS }
\\downharpoonleft                   { _SEP_CLASS }
\\downharpoonright                  { _SEP_CLASS }
\\eqcirc                            { _SEP_CLASS }
\\eqsim                             { _SEP_CLASS }
\\exists                            { _SEP_CLASS }
\\forall                            { _SEP_CLASS }
\\frown                             { _SEP_CLASS }
\\gets                              { _SEP_CLASS }
\\ggg                               { _SEP_CLASS }
\\gggtr                             { _SEP_CLASS }
\\gneq                              { _SEP_CLASS }
\\gneqq                             { _SEP_CLASS }
\\gvertneqq                         { _SEP_CLASS }
\\hookleftarrow                     { _SEP_CLASS }
\\hookrightarrow                    { _SEP_CLASS }
\\iff                               { _SEP_CLASS }
```

```
\\impliedby                    { _SEP_CLASS }
\\implies                      { _SEP_CLASS }
\\leftarrow                    { _SEP_CLASS }
\\leftarrowtail                { _SEP_CLASS }
\\leftharpoondown              { _SEP_CLASS }
\\leftharpoonup                { _SEP_CLASS }
\\leftleftarrows               { _SEP_CLASS }
\\leftrightarrow               { _SEP_CLASS }
\\leftrightarrows              { _SEP_CLASS }
\\leftrightharpoons            { _SEP_CLASS }
\\leftrightsquigarrow          { _SEP_CLASS }
\\longleftarrow                { _SEP_CLASS }
\\longleftrightarrow           { _SEP_CLASS }
\\longmapsto                   { _SEP_CLASS }
\\longrightarrow               { _SEP_CLASS }
\\looparrowleft                { _SEP_CLASS }
\\looparrowright               { _SEP_CLASS }
\\mapsto                       { _SEP_CLASS }
\\multimap                     { _SEP_CLASS }
\\nLeftarrow                   { _SEP_CLASS }
\\nLeftrightarrow              { _SEP_CLASS }
\\nRightarrow                  { _SEP_CLASS }
\\nearrow                      { _SEP_CLASS }
\\nexists                      { _SEP_CLASS }
\\ngtr                         { _SEP_CLASS }
\\nleftarrow                   { _SEP_CLASS }
\\nleftrightarrow              { _SEP_CLASS }
\\nmid                         { _SEP_CLASS }
\\nrightarrow                  { _SEP_CLASS }
\\nwarrow                      { _SEP_CLASS }
\\precapprox                   { _SEP_CLASS }
\\preccurlyeq                  { _SEP_CLASS }
\\precnapprox                  { _SEP_CLASS }
\\precneqq                     { _SEP_CLASS }
\\precnsim                     { _SEP_CLASS }
\\precsim                      { _SEP_CLASS }
\\qquad                        { _SEP_CLASS }
\\quad                         { _SEP_CLASS }
\\rightarrow                   { _SEP_CLASS }
\\rightarrowtail               { _SEP_CLASS }
\\rightharpoondown             { _SEP_CLASS }
\\rightharpoonup               { _SEP_CLASS }
\\rightleftarrows              { _SEP_CLASS }
```

```
\\rightleftharpoons                    { _SEP_CLASS }
\\rightrightarrows                     { _SEP_CLASS }
\\rightsquigarrow                      { _SEP_CLASS }
\\searrow                              { _SEP_CLASS }
\\smallfrown                           { _SEP_CLASS }
\\smallsmile                           { _SEP_CLASS }
\\swarrow                              { _SEP_CLASS }
\\to                                   { _SEP_CLASS }
\\uparrow                              { _SEP_CLASS }
\\updownarrow                          { _SEP_CLASS }
\\upharpoonleft                        { _SEP_CLASS }
\\upharpoonright                       { _SEP_CLASS }
\\upuparrows                           { _SEP_CLASS }

 /* variables */
\\Alpha                                { VAR }
\\Beta                                 { VAR }
\\Chi                                  { VAR }
\\Delta                                { VAR }
\\Epsilon                              { VAR }
\\Eta                                  { VAR }
\\Gamma                                { VAR }
\\Iota                                 { VAR }
\\Kappa                                { VAR }
\\Lambda                               { VAR }
\\Mu                                   { VAR }
\\Nu                                   { VAR }
\\Omega                                { VAR }
\\Omicron                              { VAR }
\\Phi                                  { VAR }
\\Pi                                   { VAR }
\\Psi                                  { VAR }
\\Re                                   { VAR }
\\Rho                                  { VAR }
\\Sigma                                { VAR }
\\Tau                                  { VAR }
\\Theta                                { VAR }
\\Upsilon                              { VAR }
\\VarLambda                            { VAR }
\\VarOmega                             { VAR }
\\Xi                                   { VAR }
\\Zeta                                 { VAR }
\\aleph                                { VAR }
```

```
\\alpha                                 { VAR }
\\amalg                                 { VAR }
\\beta                                  { VAR }
\\beth                                  { VAR }
\\chi                                   { VAR }
\\delta                                 { VAR }
\\ell                                   { VAR }
\\epsilon                               { VAR }
\\eta                                   { VAR }
\\eth                                   { VAR }
\\gamma                                 { VAR }
\\imath                                 { VAR }
\\iota                                  { VAR }
\\jmath                                 { VAR }
\\kappa                                 { VAR }
\\lambda                                { VAR }
\\mho                                   { VAR }
\\mu                                    { VAR }
\\nu                                    { VAR }
\\omega                                 { VAR }
\\omicron                               { VAR }
\\phi                                   { VAR }
\\psi                                   { VAR }
\\rho                                   { VAR }
\\sigma                                 { VAR }
\\tau                                   { VAR }
\\theta                                 { VAR }
\\top                                   { VAR }
\\upsilon                               { VAR }
\\varDelta                              { VAR }
\\varGamma                              { VAR }
\\varPhi                                { VAR }
\\varPi                                 { VAR }
\\varPsi                                { VAR }
\\varSigma                              { VAR }
\\varTheta                              { VAR }
\\varUpsilon                            { VAR }
\\varXi                                 { VAR }
\\varepsilon                            { VAR }
\\varkappa                              { VAR }
\\varphi                                { VAR }
\\varpi                                 { VAR }
\\varpropto                             { VAR }
```

```
\\varrho                                { VAR }
\\varsigma                              { VAR }
\\vartheta                              { VAR }
\\wr                                    { VAR }
\\xi                                    { VAR }
\\zeta                                  { VAR }

 /* functions, auto-generated */

\\Pr                                  { FUN_CLASS }
\\arccos                              { FUN_CLASS }
\\arcsin                              { FUN_CLASS }
\\arctan                              { FUN_CLASS }
\\cos                                 { FUN_CLASS }
\\cosh                                { FUN_CLASS }
\\cot                                 { FUN_CLASS }
\\coth                                { FUN_CLASS }
\\csc                                 { FUN_CLASS }
\\deg                                 { FUN_CLASS }
\\det                                 { FUN_CLASS }
\\dim                                 { FUN_CLASS }
\\exp                                 { FUN_CLASS }
\\gcd                                 { FUN_CLASS }
\\hom                                 { FUN_CLASS }
\\ker                                 { FUN_CLASS }
\\lg                                  { FUN_CLASS }
\\ln                                  { FUN_CLASS }
\\log                                 { FUN_CLASS }
\\max                                 { FUN_CLASS }
\\min                                 { FUN_CLASS }
\\sec                                 { FUN_CLASS }
\\sin                                 { FUN_CLASS }
\\sinh                                { FUN_CLASS }
\\tan                                 { FUN_CLASS }
\\tanh                                { FUN_CLASS }
 /* not LaTeX standard commands */
\\sgn                                 { FUN_CLASS }
\\signum                              { FUN_CLASS }
\\sign                                { FUN_CLASS }

 /* sum class, auto-generated */

\\arg                                 { SUM_CLASS }
```

```
\\bigcap                    { SUM_CLASS }
\\bigcup                    { SUM_CLASS }
\\bigcirc                   { SUM_CLASS }
\\bigodot                   { SUM_CLASS }
\\bigoplus                  { SUM_CLASS }
\\bigotimes                 { SUM_CLASS }
\\bigsqcup                  { SUM_CLASS }
\\bigtriangledown           { SUM_CLASS }
\\bigtriangleup             { SUM_CLASS }
\\biguplus                  { SUM_CLASS }
\\bigvee                    { SUM_CLASS }
\\bigwedge                  { SUM_CLASS }
\\coprod                    { SUM_CLASS }
\\idotsint                  { SUM_CLASS }
\\int                       { SUM_CLASS }
\\iint                      { SUM_CLASS }
\\iiint                     { SUM_CLASS }
\\iiiint                    { SUM_CLASS }
\\intop                     { SUM_CLASS }
\\inf                       { SUM_CLASS }
\\injlim                    { SUM_CLASS }
\\smallint                  { SUM_CLASS }
\\sup                       { SUM_CLASS }
\\varinjlim                 { SUM_CLASS }
\\varprojlim                { SUM_CLASS }
\\lim                       { SUM_CLASS }
\\liminf                    { SUM_CLASS }
\\limsup                    { SUM_CLASS }
\\varliminf                 { SUM_CLASS }
\\varlimsup                 { SUM_CLASS }
\\oint                      { SUM_CLASS }
\\prod                      { SUM_CLASS }
\\projlim                   { SUM_CLASS }
\\sum                       { SUM_CLASS }
```