Layout-Based Substitution Tree Indexing and Retrieval for Mathematical Expressions

APPROVED BY

SUPERVISING COMMITTEE:

Richard Zanibbi, Chair

Bo Yuan, Reader

Paul Tymann, Observer

Layout-Based Substitution Tree Indexing and Retrieval for Mathematical Expressions

by

Matthew Thomas Schellenberg

THESIS

Presented to the Faculty of the Golisano College of Computer and Information Sciences

Rochester Institute of Technology

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science, Computer Science

Rochester Institute of Technology

November 2011

Acknowledgments

Thanks to my committee members for their insightful guidance and extraordinary patience. Thanks to my experiment participants for their timely assistance. And thanks to everyone who supported me throughout this remarkable journey.

Abstract

Layout-Based Substitution Tree Indexing and Retrieval for Mathematical Expressions

Matthew Thomas Schellenberg
Rochester Institute of Technology, 2011

Supervisors: Richard Zanibbi

We introduce a new system for layout-based indexing and retrieval of mathematical expressions using substitution trees. Substitution trees can efficiently store and find hierarchically-structured data based on similarity. Previously Kolhase and Sucan applied substitution trees to indexing mathematical expressions in operator tree representation (Content MathML) and query-by-expression retrieval. In this investigation, we use substitution trees to index mathematical expressions in symbol layout tree representation (LaTeX) to group expressions based on the similarity of their symbols, symbol layout, sub-expressions and size.

We describe our novel substitution tree indexing and retrieval algorithms and our many significant contributions to the behavior of these algorithms, including: allowing substitution trees to index and retrieve layout-based mathematical expressions instead of predicates; introducing a bias in the insertion function that helps group expressions in the index based on similarity in baseline size; modifying the search function to find expressions that are not identical yet still structurally similar to a search query; and ranking search results based on their similarity in symbols and symbol layout to the search query.

We provide an experiment testing our system against the term frequency-inverse document frequency (TF-IDF) keyword-based system of Zanibbi and Yuan and demonstrate that: in many cases, the two systems are comparable; our system excelled at finding expressions identical to the search query and expressions containing relevant sub-expressions; and our system experiences some limitations due to the insertion bias and the presence of LaTeX formatting in expressions. Future work includes: designing a different insertion bias that improves the quality of search results; modifying the behavior of the search and ranking functions; and extending the scope of the system so that it can index websites or non-LaTeX expressions (such as MathML or images).

Overall, we present a promising first attempt at layout-based substitution tree indexing and retrieval for mathematical expressions.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

While information retrieval is a well-explored field of research, mathematical information retrieval (MIR) is far less developed. We present a new MIR system that uses substitution trees [7, 8] to index mathematical expressions from a database of LaTeX documents and retrieve relevant search results. The substitution tree data structure, originally created for automated theorem proving, has not been used for MIR with layout-based expressions such as those written in LaTeX. This novel approach to MIR is comparable to existing MIR systems and promises to be even more effective upon further refinement.

Conventional search engines remain ineffective in finding mathematical expressions due, in part, to their inability to correctly handle special mathematical notation and symbols [18]. Some have tried building search engines focusing on MIR to specifically address these issues, with varying success [27]. Notably, Kohlhase and Sucan [14] made a search engine using substitution trees indexing Content MathML (which represents mathematical semantics – see operator trees in [27]) and they only provide a partial description of their specific implementation of the algorithm. We chose to index LaTeX because it is a much more widely used and available source for mathematical information, though our system could be modified to index other encodings. Their system, in contrast to ours, is intended for partial queries (where some of the terms are unknown) and does not emphasize symbols and symbol layout (allowing a varied number and ordering of terms in matching expressions through the use of special attribute tags). Zanibbi and Yuan have created a keyword-based MIR system using Lucene that they show to be effective [29]. It uses a vector-space approach to store LaTeX expressions and a term frequency-inverse document frequency (TF-IDF) model to rank search results. We will be using their system in our tests for comparison and, unlike Kohlhase and Sucan, provide an experiment rating the quality of search results returned by the two systems.

1.1 Problem Description

The goal for this thesis is to design novel indexing and matching algorithms using substitution trees [7, 8] for a mathematical information retrieval system. We aim to improve retrieval effectiveness over existing search engines: the overall relevance of the documents that are returned from a LaTeX database as results of a particular search query given in LaTeX notation. We will use Zanibbi and Yuan's keyword-based system for comparison [29], examining different ways that information retrieval algorithms represent expressions for indexing purposes and explore how their current vector-space indexing model can be enhanced or changed. Specifically, we are going to investigate methods of finding other expressions in the matching process relevant to the search query by identifying sub-expressions that are present in both the document and the search query and by detecting if the query or its sub-expressions are equivalent to other expressions through structural similarity. **My hypothesis is that, by using a substitution tree indexing model, the relevance of search results for queries of LaTeX mathematical expressions will be improved over a keyword-based vector-space indexing model of Zanibbi and Yuan**: since the substitution tree's unique structure depicts sub-expressions as sub-trees, it should allow for easier searching of similar sub-expressions than the linear vector-space model by comparing the sub-trees of the search query and the contents of each indexed expression.

Mathematical information retrieval is important for the same reason that all information retrieval is important [27, 30]. Instead of first considering why someone would search for a mathematical expression, instead consider why someone would search for a word or phrase on a search engine like Google. The answer lies in a need for information (in the form of webpages and documents) relevant to the search query. Since so much information is available through digitalized documents and the Internet, parsing through that information to find what is relevant to a specific word, phrase or expression is very important. A person could be looking for not only definitions, exercises, theorems and other information directly containing the expression, but also relationships between the expression and other expressions and information (for example, linking $a^2 + b^2 = c^2$ to "Pythagorean Theorem" and $d = \sqrt{x^2 + y^2}$). A search could also find relevant expressions and information through partial matches on the query and through similarities to the query in symbols and symbol layout (for example, linking $a^2 + b^2 = c^2$ to $a^2 + b^2$ and $x^2 + y^2 = z^2$). Through searching, a user can normally learn a great deal about any given subject; thus the need for a good mathematical information retrieval system is to promote learning itself – which, in turn, will lead to future mathematical and technological innovation.

Many search engines specifically constructed for handling mathematical expressions already exist, including Math WebSearch [14], ActiveMath [16], Mathdex [19], Whelp [2], and Springer LaTeX Search. However, each has limitations. Zanibbi and Yuan have begun working on a robust search engine for mathematical expressions in an attempt to overcome some of these limitations [29, 27]. One such limitation is the difficulty in providing a proper user interface for entering mathematical expressions as search queries. Another common problem includes limited sources of information, restrictions on data based on file type or formatting and the need to manually edit documents in order to highlight indexing terms. They work to alleviate this problem by using a database of LaTeX documents which provides a standard notation for writing mathematical expressions and a significantly large collection of available documents (e.g. through arXiv.org). Most importantly, existing search engines are sometimes brittle due to ineffective information archival schemes due to poor choice in indexing methods. This can be improved with the development of an effective indexing technique for documents with mathematical information, and is what this thesis will examine.

1.2 Importance of Research

Mathematical information retrieval (MIR) is still a young area of research. This thesis hopes not only to build upon prior work done in MIR, but also more thoroughly examine the substitution tree indexing model. As large quantities of information become accessible to more and more people around the world, the ability to effectively retrieve information pertinent to the user will become imperative. Furthermore, this thesis will explore aspects of information retrieval that are essential to all work related to the field: namely, the issue of determining and understanding the relevancy of search results, and how to evaluate that relevance (making what is normally a subjective task into a quantitative evaluation for the purposes of scientific research and future comparison). Finally, important ideas for future research in MIR will be presented.

1.3 Assumptions and Limitations

The scope of our work is to create a new search engine for mathematical expressions (which we also call a “mathematical information retrieval (MIR) system”) using substitution trees. This search engine will index expressions found in a database of LaTeX docuemnts (as opposed to, for example, scraping the Internet) and will therefore be limited to the expressions found in that database. Since our focus is on using substitution tree indexing and retrieval, the components of our search engine requiring user interaction will not be designed for widespread commercial use; for example, queries are written in LaTeX syntax. Additionally, our search engine will not be streamlined to ensure that it is as time or memory-efficient as possible; our research instead focuses on a proof-of-concept study of retrieval results. Since we ignored expressions containing more than 100 symbols and assume that each letter is an individual symbol, our search engine does not work well with theorems or any expressions that contain many words or sentences. Our search engine is not capable of making mathematically semantic equivalences (unlike Kohlhase and Sucan [14]); we look for relevant search results based on similarity in symbols and symbol layout. This also means that the ranking of our search results is based on their similarity of symbol and symbol layout to the search query. The search engine makes some assumptions for its behavior during the retrieval process – such as that variable symbols hold no special meaning and can be replaced with one another to provide relevant (though not identical) search results ($x = y + 1$ is the same as $a = b + 1$) – that we believe is desirable for the majority of users; however, since the relevancy of search results is subjective, we can't guarantee that this behavior is well-suited for all users.

1.4 Our Contributions

The research presented in this paper offers the following contributions to the field of mathematical information retrieval:

- The modified substitution tree for layout-based mathematical expressions and the adaptation of the symbol layout tree encoding to LaTeX syntax.
- The novel substitution tree-based indexing algorithm for layout-based mathematical expressions incorporating a bias in the insertion algorithm that helps group expressions

in the index based on similarity in baseline size.

- The novel substitution tree-based retrieval algorithm for layout-based mathematical expressions which retrieves expressions that are not identical yet still structurally similar to a search query and a ranking algorithm that sorts search results based on the similarity of an expression's symbols and symbol layout to the search query.

- An experiment comparing our substitution tree MIR system to a leading MIR system and a with results that show the two systems to be comparable overall but suggest that our system can be improved in future implementations.

We have also provided our source code available online here:

http://www.cs.rit.edu/ dprl

1.5 Thesis Outline

The chapters contained within this paper present the full extent of our research:

Chapter 2 – Background: Previous work in MIR that is related to our research, an introduction to substitution trees including formal definitions, and a brief summary of the original indexing and retrieval algorithms.

Chapter 3 – Methodology: Description of the substitution tree indexing and retrieval algorithms (and all relevant functions), including formal definitions, pseudocode, textual and visual examples and our specific changes to the original algorithms.

Chapter 4 – Experiment and Results: Our experiment comparing two MIR systems, our presentation of its results, and our discussion analyzing those results.

Chapter 5 – Conclusion and Future Work: Our conclusions and ideas for future work based upon our research.

Chapter 6 – Appendix: Tables containing more detailed information concerning the results of our experiment.

Chapter 2

Background

This chapter discusses previous work in the field of mathematical information retrieval including other MIR systems. It also introduces and defines substitution trees and other relevant terms as well as summarizing the substitution tree indexing and retrieval algorithms.

2.1 Related Work

Information retrieval is a thoroughly explored area of artificial intelligence [10, 22]. Search engines such as Google are able to quickly retrieve a plethora of relevant information from documents and web pages on the World Wide Web based on a simple text query given by the user. However, these conventional retrieval algorithms are ineffective when search queries contain mathematical expressions [18]: expressions are hard to represent as queries through a simple textual search interface due to the use of unusual symbols or notations; different mathematical fields may have different meanings for certain symbols or notations [11]; important mathematical symbols such as operators, decimals, parentheses and other punctuation are often ignored or misinterpreted; extracting mathematical symbols from sources in a search database can be difficult because of the different ways expressions are represented (PDF, images, LaTeX, and MathML, to name a few); and, partially due to this fact, indexing sources containing mathematical information is challenging. Therefore creating a specialized search engine for mathematical information, whose search queries are mathematical expressions and the indexes of the search database are generated considering mathematical language and notation, is preferred over adapting current conventional textual search engines but could complement such engines.

Information retrieval itself has been previously well defined. Hiemstra describes three parts of the information retrieval process: representing document content (indexing), representation of the user's information need (search query formulation), and the comparison of the two representations (matching) [10]. This thesis will address the indexing and matching processes in order to find a better way of indexing the documents in the database and match relevant documents with the search query. The problem is not only constructing these two algorithms, but ensuring that they work together to achieve maximum potential. Hiemstra discusses various approaches to indexing for information retrieval systems, including the vector-space model which is currently used in Zanibbi and Yuan's search engine [29]. In this approach, each indexed document is represented by a vector which contains each term deemed important by the indexing algorithm. The search query is similarly represented by its own vector whose components are associated with the given search terms. Matching is done through a scoring algorithm which compares the components of the search vector with each of the document vectors individually. This method is effective because it is inexpensive and able to identify documents with matching terms. However, the simplicity of the model also means it is linear in the vector used to represent an item (meaning that it can only make matches based on shared symbols) and thus cannot identify relevant documents if they do not contain a matching term.

Historically, the vector-space model has suffered from a key challenge of assigning appropriate weights to vector components [10]. This was due to difficulties implementing an effective the standard term frequency-inverse document frequency (TF-IDF) weighting algorithm used by most textual search engines. Zanibbi and Yuan modified this strategy by indexing on individual expressions instead of on entire documents. This term frequency-inverse formula frequency (TF-IFF) method weighs the importance of different terms in all expressions against the importance of terms within a particular expression and produces more precise retrieval results. A similar strategy treating each expression as a separate document will be enacted in this thesis.

Zhao et al. specifically discuss the design of information retrieval systems and their ability to understand mathematical information in such a system [30]. They present three types of search engines: those which are "math-unaware," treating mathematical notation merely as text and ignoring its importances; those which are "syntactically math-aware," recognizing the syntactical structure of mathematical expression and matching terms at the syntactic level; and those which are "semantically math-aware," evaluating not only the syntactical structure of mathematical expressions but also the semantic meanings of those expressions. By definition, semantically math-aware systems are capable of manipulating expressions and resolving mathematical equivalence between them, including in cases where they are syntactically different but semantically identical; however, this semantic understanding of expressions would seem to occur on varying levels depending on the extent to which the system could match equivalent expressions (more on equivalence in the next section).

Graf introduces the substitution tree indexing method [7, 8], originally developed to perform term substitution for automated theorm proving. A substitution tree represents any set of idempotent substitutions. The nodes of the tree are substitutions, and each branch in the tree represents a binding chain for variables. The substitutions of a branch from the root down to a leaf node are combined to yield an instance of the root node's substitution. Retrieval is based on a representation of substitutions as lists of variable-term pairs and a backtracking algorithm that traverses through the tree searching for appropriate substitutions. Overall, substitution trees have low memory requirements (since it stores substitutions and not complete terms) and provide high search speed compared to other tree indexing methods [7, 8]. Substitution tree indexing is itself a combination of the pre-existing discrimination tree and abstraction tree indexing methods, offering the advantages of both small size and better indexing performance.

2.2 Other MIR Systems

Researchers have been working on a variety of other MIR systems for over a decade; however, many are now old or deprecated.

LaTeX Search, found at latexsearch.com, is a service provided by Springer (source: www.springer.com). According to the site, the system allows users to search "over 1 million LaTeX equations embedded in tens of thousands of articles from various disciplines." Entering a search query produces two separate lists of results – one for exact matches and another for "similar" matches – and individual results display the LaTeX source code for the matching expression, its converted image, and links to the source document.

ActiveMath is a learning environment created by Libbrecht and Melis [16] that contains semantically represented mathematical content. It indexes definitions, examples, exercises, and other mathematical content from documents in the OpenMath library by encoding them as sequences of semantic tokens using a custom syntax and storing them in a Lucene index. The system retrieves content by converting the search query into a sequence of semantic tokens (like it does during indexing) and comparing those tokens to entries found in the index.

Whelp is a content based mathematical search engine introduced by Asperti et al. [2] that allows users to search through a library of 40,000 theorems from the Coq proof assistant.

Theorems are indexed using a "logic-independent metadata model" by relating objects in the theorem based on their position (where possible positions include hypothesis, conclusion and proof, among others). Queries are entered using a custom syntax similar to TEXwhich is then converted into metadata and compared to the entries in the index. Search results are a list of references to the matching theorems' locations in the Coq library.

The Mathdex search engine, built by Miner and Munavalli [19], is able to convert all types of mathematical content available on the Web into XHTML+MathML in order to create an extensive index of mathematical expressions. This system uses weighted mathematical n-grams to emphasize similarity of symbols and sub-expressions between expressions.

Miller and Youssef's NIST Digital Library of Mathematical Functions (DLMF) Project uses LATEXML to translate LATEX into XHTML and MathML for expression indexing [18]. They augmented an existing textual search engine for expression retrieval by implementing a keyword-based expression representation and developing a custom mathematical query language resembling TEX. They also added a thesaurus in order to enumerate terms that are related (such as semantic equivalences or names for famous expressions) which they claim "will continue to evolve."

For our investigation, two of these systems are especially important: Math WebSearch, which uses substitution trees to index Content MathML; and a system designed by Zanibbi and Yuan [29] to index LATEX.

2.2.1 Math WebSearch

Math WebSearch, developed by Kohlhase and Sucan [14], is the first MIR system to index expressions written in Content MatML (contained in MathML and OpenMath) using substitution trees. They only provide a partial description of their code, including rough pseudocode for their insertion function. Search queries are entered as Content MathML and can include generic terms that take advantage of the unique structure of the substitution tree in order to match a larger number of expressions. We seek to build upon their work with substitution tree indexing by applying it to layout-based mathematical equations in LaTeX documents.

2.2.2 Keyword-Based Retrieval using Lucene

Zanibbi and Yuan's keyword-based Lucene MIR system [29] uses a vector-space model to index expressions from a database of LaTeX documents and ranks search results based on a TF-IDF (term frequency-inverse document frequency) relevance measure. Due to its effectiveness, its use of layout-based expressions and its different indexing and retrieval models, we will be testing our novel substitution tree MIR system against this system in our experiment (described in Chapter 4).

2.3 Graf's Substitution Trees

Graf first introduced the substitution tree as a tool for automated theorem proving [7, 8]. It provides an efficient and intuitive way for storing predicates and grouping them based on similarity. These predicates are composed of terms – variables, functions (terms that also contain a set of terms) and constants (functions whose set of terms is empty) – and the similarity of two predicates is based upon the terms they share and the arrangement of those terms.

In a substitution tree, each node represents a predicate; the leaf nodes represent specific predicates that have been inserted into the tree, while the non-leaf nodes represent predicates containing one or more generalized variables, known as *substitution variables*.

Each child node specializes its parent by replacing one or more of these substitution variables with specific terms; these replacements are called *substitutions* and are contained within the node. Thus the predicates in the substitution tree go from more abstract to more specific through substitutions. Following a branch of the tree from root to leaf will produce a predicate represented by the leaf node through the combined substitutions of every node along the branch. This also means that every child node is an *instance* of its parent, requiring only a substitution (or set of substitutions) to produce the child's predicate from the parent's, and every parent node is a *generalization* of each of its children. The substitution tree represents all of the inserted predicates simultaneously and each predicate can be produced by following a specific branch.

2.4 Our Substitution Trees

Our implementation uses mathematical expressions instead of predicates; expressions themselves are essentially sets of terms, where each term represents a symbol in the expression. This makes our conversion much easier: each node in our substitution trees represents an expression instead of a predicate, but otherwise their behavior is much the same. Substitution variables still replace terms, and substitutions are still made from substitution variables to terms; terms simply represent a wider variety of symbols that appear in mathematical expressions, such as operators ($+$, $*$, etc.).

We use another type of tree, called a *symbol layout tree* (or SLT), to represent the expressions within the nodes of the substitution tree. An SLT is an encoding for a mathematical expression that is constructed based on the spatial relationship of the symbols in that expression. Each node of the SLT contains a *term* that represents a variable, constant, symbol or function in an expression. Each node has at least four branches dependent on

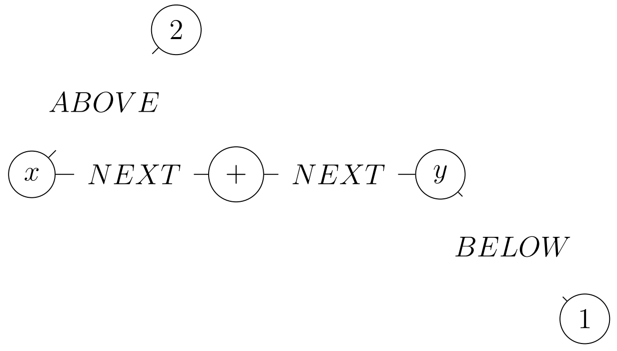

Figure 2.1: The symbol layout tree for $x^2 + y_1$. The “above” branch is also used for terms that are superscript to another term (a caret in LaTeX), and the “below” branch for terms that are subscript (an underscore).

the node’s term: one representing the term(s) positioned spatially to the right of the term, one representing the term(s) positioned spatially above or superscript to the term, one representing the term(s) positioned spatially below or subscript to the term, and one or more representing the sets of terms that are argument(s) to the term, if the term is a function: in LaTeX functions (e.g., `\frac`), these are positioned within sets of brackets { }.

Due to the similarity of our representation of mathematical expressions to the predicates used in Graf’s original implementation, the overall behavior for the indexing and retrieval algorithms remains the same and is described in this section. Changes we have made for our specific implementation of Graf’s definitions for substitution trees as well as explicit descriptions of our code are detailed in Chapter 3.

2.4.1 Definitions

A *symbol layout tree* is described by a 5-tuple $(t, A, B, \{X_1, ..., X_n\}, N)$ where t is a term and A, B, $X_1, ..., X_n$ and N are SLTs. A is the SLT positioned spatially *above* or *superscript to* t, B is the SLT positioned spatially *below* or *subscript to* t, $X_1, ..., X_n$ are the SLTs that are *arguments* to t, and N is the SLT positioned spatially to the right of t (the *next* node). We assume that a LaTeX term cannot have another term both above and superscript to it, or both below and subscript to it. No node's term may equal $\emptyset$ (the empty set) but any node itself may equal $\emptyset$. We use the notation (t) as shorthand for SLTs that only contain a term, and (t, N) for SLTs that only contain a term and a *next* branch: $(t) = (t, \emptyset, \emptyset, \emptyset, \emptyset), (t, N) = (t, \emptyset, \emptyset, \emptyset, N)$. We use the terms "SLT" and "expression" interchangeably. For example, $x^2 + 5 * \sqrt{y_1} = (x, (2), \emptyset, \emptyset, (+, (5, (*, (\backslash sqrt, \emptyset, \emptyset,$ $(y, \emptyset, (1), \emptyset, \emptyset), \emptyset)))))$.

The *sub-expressions* of an expression are all of the SLTs contained within Above, Below and Argument branches, as well as those within parentheses, brackets and braces. Sub-expressions themselves are also expressions and can contain sub-expressions of their own. For example, the sub-expressions of $x_{n-1}^{x*(y+1)}$ are $n - 1$, $x * (y + 1)$ and $y + 1$.

A *substitution* $\sigma = \{S_1 \rightarrow T_1, ..., S_n \rightarrow T_n\}, \forall i\ S_i \in \mathbf{V}$ represents a replacement of each SLT S_i with the corresponding SLT T_i. $\mathbf{V}$ is the set of *variable symbols*: single-node SLTs which will hereafter be represented by $\boxed{\text{N}}$, where N is an integer. We will use the term "substitution" to refer to both a set of substitutions (σ) and a single substitution ($S \rightarrow T$). When a substitution σ is *applied* to an SLT X, each instance of $S_i \in \sigma$ in X is replaced with $T_i\ \forall i$. For example, if $\sigma = \{\boxed{1} \rightarrow x^2 = (x, (2), \emptyset, \emptyset, \emptyset),\ \boxed{2} \rightarrow 5 = (5)\}$ and $X = \boxed{1} + \boxed{2} = (\boxed{1}, (+, (\boxed{2})))$, $apply(\sigma, X) = X\sigma = (x, (2), \emptyset, \emptyset, (+, (5))) = x^2 + 5$.

The *domain* of a substitution is defined as $\mathsf{DOM}(\sigma) := \{x \in \mathbf{V} | x\sigma \neq x\}$. The *codomain* of a substitution is defined as $\mathsf{COD}(\sigma) := \{x\sigma | x \in \mathsf{DOM}(\sigma)\}$. The *image* of a substitution is defined as $\mathsf{IM}(\sigma) := \mathsf{VAR}(\mathsf{COD}(\sigma))$ where $\mathsf{VAR}(T)$ is the set of all substitution variables occurring in the SLT T. Thus the image of a substitution is the set of all substitution variables occurring in the codomain of that substitution. For example, if $\sigma = \{\boxed{1} \rightarrow$

$5, \boxed{2} \rightarrow \boxed{3} + \boxed{4}\}$, then $image(\sigma) = \{\boxed{3}, \boxed{4}\}$. A *restriction* $\sigma|_U$ where $U \subseteq V$ means that $\mathsf{DOM}(\sigma|_U) \subseteq U$.

Two substitutions $\sigma = \{S_1 \rightarrow T_1, ..., S_n \rightarrow T_n\}$ and $\tau = \{X_1 \rightarrow Y_1, ..., X_n \rightarrow Y_n\}$ can be *composed*, or added together, using the following formula:

$$compose(\sigma, \tau) = \sigma\tau = \{S_1 \rightarrow T_1\tau, ..., S_n \rightarrow T_n\tau\} \cup \{X_i \rightarrow Y_i | X_i \in \mathsf{DOM}(\tau) \backslash \mathsf{DOM}(\sigma)\}$$

For example, if $\sigma = \{z \rightarrow f(x)\}$ and $\tau = \{x \rightarrow a, y \rightarrow c\}$, $\sigma\tau = \{z \rightarrow f(a), x \rightarrow a, y \rightarrow c\}$. Notice that the order in which two substitutions are composed makes a difference because the substitutions in σ are applied to τ but not visa-versa.

A *substitution tree* represents a set of substitutions where each node is a substitution and each branch is a set of idempotent substitutions. (A substitution $\sigma = \{S_1 \rightarrow T_1, ..., S_n \rightarrow T_n\}$ is idempotent if $\forall i \ S_i$ does not appear in T_i). Each branch in the tree represents a binding chain for variables, and, for any node, the composition of all substitutions in the branch from the root to that node produce an expression that is an instance of the root node's substitution. Thus any node itself can be said to contain an expression which is found by composing the substitutions from the root to that node. Any mathematical expression may be inserted into a substitution tree where it becomes represented by a new node if it's not already in the tree. This insertion may also change existing nodes in order to create the most generalized set of substitutions for all expressions contained within the tree. The order in which the expressions are inserted impacts the layout of the substitution tree. A substitution tree node is described by a tuple (τ, Σ) where τ is the substitution represented by the node and Σ is the set of child substitution tree nodes. Since every node represents an expression, we sometimes use "node" as shorthand for "the expression that the node represents."

$$\begin{tikzcd}
& & \boxed{0} \to \boxed{1} - \boxed{2} \arrow[dl, dash] \arrow[dr, dash] & \\
& \boxed{2} \to 5 \arrow[dl, dash] \arrow[dr, dash] & & \boxed{1} \to 5,\ \boxed{2} \to \boxed{*_1} \\
\boxed{1} \to 3 & & \boxed{1} \to \boxed{*_1} &
\end{tikzcd}$$

Figure 2.2: A substitution tree representing the expressions $x - 5$, $3 - 5$ and $5 - x$. These expressions are normalized to $\boxed{*_1} - 5$, $3 - 5$, and $5 - \boxed{*_1}$ respectively. Indicator variables can be substituted for any single mathematical variable; in this case, $\boxed{*_1} \to x$. $\boxed{0}$ is used as the root substitution variable. An expression can be reconstructed by composing the substitutions along a branch. For example, the expression $x - 5$ is reconstructed using the following set of substitutions: $\boxed{0} \to \boxed{1} - \boxed{2}$, $\boxed{2} \to 5$, $\boxed{1} \to \boxed{*_1}$ where $\boxed{*_1} \to x$.

2.4.2 Indexing Expressions using Substitution Trees

Indexing mathematical expressions using substitution trees involves inserting all of the expressions, one at a time, into a single substitution tree. This constructs a substitution tree (the *index*) that represents each expression.

Before a new expression is inserted into a substitution tree, all variables in the expression (hereafter known as *mathematical variables*) must be *normalized* by renaming them as *indicator variables*, denoted by $\boxed{*_i}$. An indicator variable is a type of substitution variable that can only be substituted for a mathematical variable. Any indicator variable that is not already bound can be substituted for any mathematical variable; so $\boxed{*_i} = a = \ldots = z$. Variables are normalized in the order they appear in the input expression from left to right: for example, $x + y$ and $y + x$ are both normalized to $\boxed{*_1} + \boxed{*_2}$. This normalization allows a single node to represent multiple expressions, and this overlap helps both indexing (since fewer nodes makes the tree smaller and insertion faster) and retrieval (since expressions with such similarity should be relevant search results; after all, x can equal y in the right context).

Inserting an expression into a substitution tree relies on the property that every child

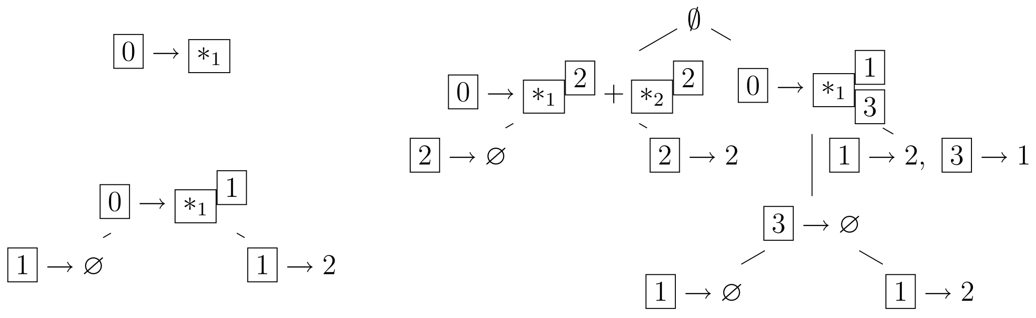

Figure 2.3: *Top Left:* The expression x is inserted into an empty substitution tree. The tree now contains one node with a substitution from the root substitution variable $\boxed{0}$ to the normalized expression. This substitution will not only match x but also any single mathematical variable. *Bottom Left:* Now x^2 is inserted into the same substitution tree. Since no match could be found, it creates a generalization – a new node replacing the old root and its children. The two new children nodes represent more specifics instance of this new, generalized substitution. *Right:* The same substitution tree after the expressions $x + y$, $x^2 + y^2$ and x_1^2 are added. A null substitution $\emptyset$ is sometimes produced if a generalized substitution is empty. A substitution to $\varnothing$ represents a replacement of that substitution variable with nothing.

node is an instance of its parent and every parent node is a generalization of all its children. The insertion algorithm takes a normalized expression e and first looks for a *match* among the index root and its children. Two expressions match if one is an instance of another. If any of the root's children are a match, the algorithm is called recursively using that child as the new root; if the root alone is a match, a new node representing e is created and added to the root's children. If none of the nodes are a match, then the algorithm finds the *most specific common generalization* (MSCG) of e and the root, where the MSCG is a substitution containing the most specific (the closest match) of all of the possible generalizations of the two expressions. This generalization replaces the root node and takes two children of its own: an instance of the root (and its original children) and an instance of e (a leaf node that represent e). The MSCG algorithm is described in great detail in the next chapter.

2.4.3 Retrieving Expressions from Substitution Trees

Retrieving relevant mathematical expressions from substitution trees benefits from the properties of the tree and the behavior of insertion. Finding exact matches to a *search query* is straightforward because, if the expression exists in the index, it can be found by simply following the correct branch (the branch that contains nodes that match the search query). Expressions similar to the search query can be found while following the matching branch due to the nature of how they were inserted. Since a new expression is inserted into a branch that shares a common match or MSCG, that expression will share the most similarities with its parent and siblings. These similarities are based on the expressions' shared symbols and symbol layout (the arrangement of those symbols). Thus the parents and siblings of a node that matches the search query will also be relevant search results.

The search algorithm works by seeing if the search query is a match to the index root. If so, the algorithm is called recursively on each of the root's children; if not, the search fails. If the root is both a match and a leaf, the expression it represents is added to the list of results.

An important note concerning the retrieval algorithm is that we in no way search based on semantic relevancy. The search function cannot consider equivalences or mathematical properties (such as $1+2 = 2+1$), keyword associations (such as the names of famous expressions) or other semantic knowledge.

Chapter 3

Methodology

The previous chapter explained the overall behavior of substitution tree indexing and retrieval as it appeared in Graf's original mathematical definitions for substitution trees [7, 8]. This chapter introduces the necessary additions and modifications we've made to our own substitution tree implementation in order to accommodate our innovative research in indexing and retrieving mathematical expressions. We present these novel changes throughout the chapter as we describe in great detail the different data structures and functions that form our algorithms.

3.1 Summary of Algorithm Modifications

Some of the changes we've made are especially significant: the use of mathematical expressions, the insertion bias, and the overall search behavior.

3.1.1 Predicates and Mathematical Expressions

As previously stated, we index whole mathematical expressions instead of predicates. Expressions are more complex than predicates due to the spatial differences of their symbols – superscript, subscript, above, below, or contained within a function. Additionally, expressions are sets of terms rather than either being single terms or functions (which, while similar, behave differently than sets of terms). We represent expressions in the index using symbol layout trees (as discussed in the previous chapter), which is a layout-based encoding for expressions. SLTs can be considered specialized 5-element predicates that can be nested to form different branches of the tree depending on the symbol layout. Thanks to the beauty of recursion, we can represent expressions using SLTs and still take advantage of much of Graf's substitution tree implementation. However, this change to SLTs required modifying many of Graf's functions for our own implementation – in particular, tracking the corresponding branches of SLTs. Therefore we are specializing – rather than generalizing – Graf's representation and implementation.

Another change from Graf's implementation is that our substitution variables can have Above, Below, and Argument branches of its own, which can themselves also be substitution variables. This was necessary because of our switch to SLTs and gives our substitutions the ability to match a wider array of expressions through extended dimensionality.

3.1.2 The Insertion Bias

The most significant addition we made to the indexing algorithm was to introduce an insertion bias. A bias is needed because otherwise most of the expressions group together in a single branch from the index root. This is probably due to our transition from predicates to expressions and the sheer size of our index. This behavior is not desired because it does not take full advantage of the substitution tree's structure and would make retrieval inefficient (often forcing the algorithm to run an exhaustive search of the index). The bias we have chosen is the size of the *baseline*, which is the set of SLT nodes starting from the root of an expression and continuing along the Next branch of each node. The baseline size is not affected by the number of non-empty Above, Below or Argument branches that extend from the baseline nodes, nor their depth: thus x and $x^{x^y_z}$ both have a baseline size of 1, and $x + y$ and $\cos x + \frac{1}{2}$ both have a baseline size of 3. An SLT node and its corresponding term are said to lie along the baseline if it is part of this set. This bias is introduced in the `match`

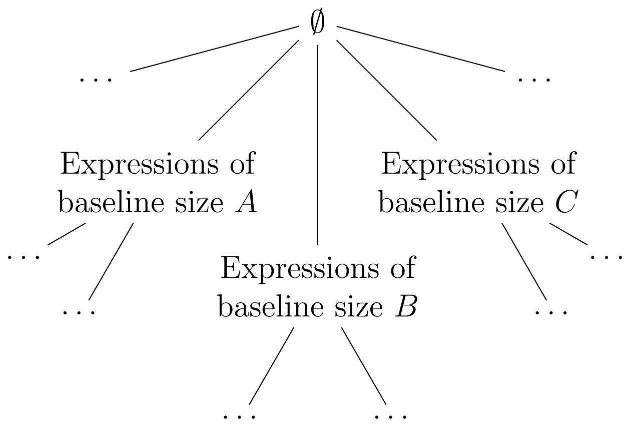
∅
Expressions of baseline size A
Expressions of baseline size B
Expressions of baseline size C
...

Figure 3.1: Essentially, the sub-trees of the substitution tree index's root are separated by the size of an expression's baseline due to the baseline insertion bias.

and `MSCG` functions and causes each of the index root's sub-trees to represent expressions of a different baseline size. This specific bias is desirable because it helps group similar expressions together (based on their size) which is useful for both indexing and retrieval (since large expressions are less relevant to small search queries, and visa-versa).

While a single substitution variable can't match multiple terms along the baseline due to this bias, it can still match multiple terms that are above, below, or argument to the terms along the baseline (in other words, non-baseline sub-expressions). For example, the expression $\cos(\boxed{1})$ can match $\cos(x+1)$ using the matcher $\boxed{1} \rightarrow x+1$, but the expression $\boxed{1}$ cannot match $x+1$ because it lies along the baseline. Additionally, a substitution variable that does not lie along the baseline can match the empty substitution $\varnothing$, allowing us to match expressions like x^2 with x because $x = x^{\varnothing}$. This behavior is an inherent part of the algorithm and is only superseded for nodes along the Next branch due to the baseline bias we have implemented.

3.1.3 Finding Relevant Results through Multiple Searches

Graf's search function takes an expression (the search query) and returns the expressions in the index that are identical to the query. However, for our system, we want to find both expressions that are identical to the query and also expressions that aren't identical but still relevant to the query. To solve this problem, for each search query entered by the user, we make multiple searches using the query and variations on the query (including the query's sub-expressions [14]) generated by the retrieval algorithm. Kohlhase and Sucan's system do not search using additional queries but do include attributes to individual terms in the query that allow them to match generic terms or an unspecified number or ordering of terms. Our variations are described later in this chapter.

3.2 The Data Structures

SLTs are represented by a C struct containing an array of characters for the SLT's term, three pointers to other SLT structs for its Above, Below and Next branches (any of which can be $\emptyset$ if that branch does not exist), and a vector of pointers to SLT structs for its zero or more Argument branches. Substitutions are represented by C++ maps from character pointers to SLT struct pointers. With a substitution $\sigma = \{S_1 \rightarrow T_1, \ldots, S_n \rightarrow T_n\}$, this map corresponds to each substitution $S_i \rightarrow T_i$ where the character pointer is the substitution variable (S_i) and the SLT struct pointer is the expression (T_i). Substitution trees are represented by a C struct containing a unique id (used for saving the tree to file), a substitution map for the node's set of substitutions, a vector of substitution tree pointers for the node's children, and two vectors – one of SLT struct pointers, another of character pointers – for the node's final expressions (as SLT structs) and the LaTeX document within which those expressions are contained. These last two vectors are always the same size, and are always empty for non-leaf nodes and non-empty for leaf nodes. This information is saved when an expression is first inserted into the substitution tree and is used when returning search results.

Many of the functions in our system rely on either iterating recursively through nodes in a tree (symbol layout or substitution) or through the elements in a substitution map. Also, all functions that don't return an integer are call-by-reference, so when we say that a data structure is "returned" we really mean that it was given as a function argument and thereafter modified. Functions that can succeed or fail return an integer (0 or 1) to designate their outcome.

3.3 Implementing Basic Functions

All substitutions have basic inherent mathematical properties as described in the previous chapter, including taking the *domain* or *image* of a substitution, *applying* a substitution to an expression, and *composing* two substitutions. These functions are necessary for the indexing and retrieval algorithms and their code implementations are described below.

The `domain` function takes a substitution map (a C++ map representing a substitution). It puts each of the map's keys into a vector and returns that vector.

The `image` function is given a substitution map. For each element in the map, if that element's term is a substitution variable, the function puts that term into a vector. Then it recurs on each of the element's branches (calling itself on the Above node, Below node, Next node, and each Argument node). The same vector is used for each element and each branch and is returned by the function once all elements have been evaluated. The final vector contains all of the substitution variables found in the codomain of the given substitution map (the SLTs in each value of the map's key-value pairs).

The `apply` function takes an SLT struct and a substitution map and returns a new SLT struct. Starting at the SLT's root (the "current node"), it looks for a key in the substitution map that matches the current node's term. If a matching key exists, we know that the term is a substitution variable and that it has a known mapping in the input substitution; therefore the function replaces the old term (in the current node) with the new term (the term contained within the key's corresponding SLT). The new term's Next branch is inserted into the SLT in front of the old term's Next branch, and the new term's Above, Below and Argument branches, if non-empty, are copied over as well, completely replacing those of the old term. This replacement can be done safely because the system is designed to guarantee that a substitution variable with an Above, Below or Argument branch will not be replaced by a term containing those branches (for example, the application of the substitution $\boxed{1} \rightarrow x^3$ to the expression $\boxed{1}^2$ will never exist in a substitution tree). The function then recurs on each of the current node's branches in order to apply the substitution to the whole expression.

The `compose` function is given two substitution maps $\sigma = \{S_1 \rightarrow T_1, \ldots, S_m \rightarrow T_m\}$ and $\tau = \{Q_1 \rightarrow R_1, \ldots, Q_n \rightarrow R_n\}$ and returns their composition as a new substitution map. First, the function applies τ to each element in σ: $T_i\tau\ \forall i, 0 \leq i \leq m$. The resulting expression is added to the new substitution map with its corresponding key S_i. Then each substitution $Q_i \rightarrow R_i$ where $Q_i \in \mathsf{DOM}(\tau)\backslash\mathsf{DOM}(\sigma)$ (for substitution variables Q_i distinct from $S_1, \ldots, S_m$) is added to the new substitution map as well (an example composition is given in the previous chapter).

3.4 The Indexing Algorithm

The indexing process involves first taking a collection of LaTeX documents and extracting their mathematical expressions into individual files through Zanibbi and Yuan's modified version of the `latex2html` utility [29]. Each file is then converted from LaTeX syntax to SLT syntax by constructing a parse tree of the LaTeX grammar and transforming the parse tree into a linearized representation of SLTs using a custom TXL grammar [5]. These SLT files are then inserted into the substitution tree index one after another. The order of insertion is based on the name of the document and the order of appearance in that document. Once the index has been created it is saved to a text file which allows the index to be recreated when the program is run again.

It is important to note that, for the extraction process, mathematical expressions written in "display mode" using double dollar signs ($$ ... $$) are treated exactly the same as identical expressions written using single dollar signs ($ ... $) even though they might be rendered with slightly different spacing. For example, generating the SLT for $\sum_{i=1}^{n} i$ always puts the $i = 1$ in the Below branch even though it is shown as superscript to the summation sign when using single dollar signs and below the summation sign when using double dollar signs. Additionally, LaTeX functions that show arguments as spatially above or below (such as `\frac`) are treated like all other LaTeX functions. For example, generating the SLT for $\frac{1}{2}$ would create an SLT node containing `\frac` with empty Above, Below and Next branches and two Argument branches containing 1 and 2 respectively.

This algorithm has a few shortcomings. First, it cannot consider semantic similarities when inserting an expression, so, for example, $1 + 2$ will not be matched to $2 + 1$. Also, the comparison of expressions during matching does not find the largest common subsequence of the two expressions, but instead always starts comparing from the root of the SLT; therefore

$2 + 1$ and $3 + 2 + 1$ will not be matched either. However, these expressions will likely still be returned as relevant results during the retrieval process as described below. (Remember that two expressions only match if one is an instance of another, and otherwise will not be matched even if the two expressions are similar.)

The indexing algorithm contains five major parts: normalizing mathematical variables in expressions, looking for matching expressions, finding the most specific common generalization of expressions, selecting branches in the tree for insertion, and inserting expressions into the tree. The rest of this section describes each of these parts and their corresponding functions.

3.4.1 Normalization of Mathematical Variables in Expressions

The arguments for the normalization function are an integer used to create new indicator variables and two pointers to the root of an SLT: one will remain pointing to the root, and the other will be advanced as the function iterates through the SLT (the "current node"). It checks if the term contained in the current node needs to be normalized. If so, it creates a new indicator variable using the input integer (and increasing it by one for future use) to replace that term and iterates through the entire expression (from the root to the leaf of each of its branches) replacing each instance of the term with the indicator variable. After this check is completed, the function calls itself recursively on each of the current node's branches to ensure that the expression is fully normalized.

For normalization, we assume that mathematical variables are only represented by single alphabetical characters ($a, \ldots, z$) or greek characters ($\alpha, \ldots, \omega$ and $A, \ldots, \Omega$). We also assume an implied multiplication of consecutive alphabetical characters, so $abc = a \times b \times c = \boxed{*_1} \times \boxed{*_2} \times \boxed{*_3}$. Finally, we assume that all functions are preceded by a backslash, in adherence to LaTeX syntax, and are treated as a single term; thus $\backslash cos\{x\}$ will be treated as a function while $cos\{x\}$ will be treated as $c \times o \times s \times (x)$.

3.4.2 Looking for Matching Expressions

Comparing two expressions to determine if they have similarities in symbols or symbol layout is done through matching. Matching occurs through Graf's two functions, $\mathcal{G}$ and $\mathcal{V}$. Both take two substitution maps, τ and ρ, and try to find a matcher substitution σ to show that ρ is an instance of τ. The difference between the two functions is that $\mathcal{G}$ allows indicator and substitution variables (the difference between indicator and substitution variables is described in the previous chapter) to be substituted for mathematical variables (necessary for retrieval) while $\mathcal{V}$ only allows non-indicator substitution variables to be substituted, treating indicator variables as constants (used in the indexing algorithm). Both $\mathcal{G}$ and $\mathcal{V}$ iterate through the domain of τ (the keys in the substitution map), compute $((X_i\tau)\rho)$ and $(X_i\rho)$ and give these resulting SLTs to the match function.

$$\mathcal{G}(\tau,\rho) := \{\sigma | \forall X_i \in \mathsf{DOM}(\tau), X_i\tau\rho\sigma = X_i\rho\} \tag{3.1}$$

$$\mathcal{V}(\tau,\rho) := \{\sigma | \sigma \in \mathcal{G}(\tau,\rho) \wedge \mathsf{DOM}(\sigma) \cap \mathbf{V}^* = \emptyset\} \tag{3.2}$$

($\mathbf{V}^* \subset \mathbf{V}$ is the set of all indicator variables.)

The match function takes two SLT structs, S and T, an integer for whether to match indicator variables, and an integer for whether the current SLT nodes lie along the baseline (so, when first called, $baseline = 1$), and returns a matcher substitution σ for the two SLTs if successful. If both SLTs are $\emptyset$, the match completes successfully (the base case). If either SLT is empty ($S = (\varnothing)$ or $T = (\varnothing)$), the function replaces it with $\emptyset$. If only one of the

SLTs is $\emptyset$ and *baseline* $= 1$, the match fails because of the insertion bias (the baselines of the two SLTs are different sizes). If the terms contained within the current nodes of S and T are the same, recursively call the function on each of their corresponding branches (so, using S's Above node and T's Above node, etc.) and return success only if all those calls are successful.

If *match indicator variables* $= 1$ and S's term is an indicator variable v and T's term is a mathematical variable t, check if $v\sigma = v$ or $v\sigma = t$: if either are true, add $v \rightarrow t$ to σ and recursively call the function on each branch as if the terms were the same; if neither are true, the match fails because an indicator variable cannot be substituted for two different terms simultaneously.

If the term in S's current node is a substitution variable v, check if $v\sigma = v$ or $v\sigma T$ (where, if $T = \emptyset$, set $T = (\varnothing)$). If neither are true, the match fails like before. If either are true, create a temporary SLT $U = T$. Then consider each pair of the corresponding branches of S and T: for each pair, if both branches exist (the nodes are not $\emptyset$), erase the branch in U and recursively call the function on those branches. If the match fails for any of these recursive calls, or the number of Argument branches for S and T are not equal, return failure. Additionally, if S has a Next branch but T does not, the match fails because S's next node doesn't have anything to match. Finally, if T has a Next branch and S does not and *baseline* $= 1$, the match fails because a substitution variable cannot match multiple nodes along the baseline. If it passes all these cases successfully, the match succeeds. If S's term was not a substitution variable, the match fails.

The pseudo-code for the `match` function is shown in Figures 3.2 (Part 1), 3.3 (Part 2), and 3.4 (Part 3). An example of the `match` function in action can be seen in Table 3.1.

```
MATCH(SLT S, SLT T, substitution_map sigma, baseline, MIV) {
  if(S and T are NULL) {
    return success
  }
  % Insertion bias:  baseline size is not equal
  if(baseline = true and (S is NULL or T is NULL)) {
    return failure
  }
  if(S's term is EMPTY and baseline = false) {
    return match(NULL, T, sigma, baseline, MIV)
  }
  if(T's term is EMPTY and baseline = false) {
    return match(S, NULL, sigma, baseline, MIV)
  }
  % Possible match if the terms are the same or if a substitution for an
  % indicator variable can be made
  if(S's term = T's term or (S's term is an indicator variable and
      T's term is a mathematical variable and MIV = true)) {
    if(S's term is an indicator variable) {
      % Add the new substitution to sigma if possible
      if(sigma does not contain S's term as a key) {
        create a copy of T called U
        erase all of U's branches
        add (S's term, U) to sigma
      }
      % Fail if an incompatible substitution already exists
      else if(sigma contains S's term as a key and
          get_key(sigma, S's term) != T's term) {
        return failure
      }
    }
[CONTINUED ON THE NEXT PAGE]
```

Figure 3.2: Part 1 of the pseudo-code for the `match` function. MIV represents whether the function matches indicator variables. An SLT is "EMPTY" if its term is ∅ and it has no branches.

[CONTINUED FROM THE PREVIOUS PAGE]

```
    if(S's number of Argument branches != T's number of Argument branches) {
      return failure
    }
    % Match corresponding branches and fail if any match fails
    for(each Argument branch s_i in S and each Argument branch t_i in T) {
      if(match(s_i, t_i, sigma, false, MIV) fails) {
        return failure
      }
    }
    if(match(S's Above node, T's Above node, sigma, false, MIV) and
        match(S's Below node, T's Below node, sigma, false, MIV) and
        match(S's Next node, T's Next node, sigma, baseline, MIV)
        are all successful) {
      return success
    }
    return failure
  }
  % Possible match if a substitution can be made
  if(S's term is a non-indicator substitution variable) {
    if(sigma does not contain S's term as a key) {
      if(T is NULL) {
        add (S's term, EMPTY) to sigma
        return success
      }
      create a copy of T called U
      % Erase branches that will be matched individually
      for(each element Z in the set [Above, Below, Next]) {
        if(both S and T have a Z branch) {
          erase U's Z branch
        }
      }
      for(each Argument branch s_i in S and each Argument branch t_i in T) {
        erase Argument branch u_i in U
      }
```

[CONTINUED ON THE NEXT PAGE]

Figure 3.3: Part 2 of the pseudo-code for the `match` function.

```
[CONTINUED FROM THE PREVIOUS PAGE]
      add (S's term, U) to sigma
      ret = success
      % Match corresponding branches and fail if any match fail
      for(each element Z in the set [Above, Below, Next]) {
        if(ret = success and S has a Z branch and (T has a Z branch or
            S's Z node's term is a substitution variable)) {
          ret = match(S's Z node, T's Z node, sigma, false, MIV)
        }
      }
      for(each Argument branch s_i in S and each Argument branch t_i in T) {
        if(match(s_i, t_i, sigma, false, MIV) fails) {
          ret = failure
          break
        }
      }
      % Insertion bias
      if((S has a Next branch and T does not) or (baseline = true and
          T has a Next branch and S does not)) {
        ret = failure
      }
      if(ret = success) {
        return success
      }
      remove (S's term, U) from sigma
      return failure
    }
    else if(S maps to EMPTY in sigma and T is NULL or EMPTY) {
      return success
    }
  }
  return failure
}
[END]
```

Figure 3.4: Part 3 of the pseudo-code for the `match` function.

Table 3.1: Example of the `match` function called by $\mathcal{G}$ where $S = \boxed{1}^3 + \boxed{*_1}_1$ and $T = \boxed{1}^3 + x_2$. Note that if the function had been called by $\mathcal{V}$ instead, it would have failed on Step 19.

Step	S	T	Result	Next
1	$\boxed{1}$	$\boxed{1}$	Both terms match.	Check Above branch.
2	3	3	Both terms match.	Check Above branch.
3	$\emptyset$	$\emptyset$	No Above branches.	Backtrace.
4	3	3		Check Below branch.
5	$\emptyset$	$\emptyset$	No Below branches.	Backtrace.
6	3	3		Check Next branch.
7	$\emptyset$	$\emptyset$	No Next branches.	Backtrace.
8	3	3		Check Argument branches.
9	$\emptyset$	$\emptyset$	No Argument branches.	Backtrace.
10	3	3	All branches match.	Match is successful. Backtrace.
11	$\boxed{1}$	$\boxed{1}$		Check Below branch.
12	$\emptyset$	$\emptyset$	No Below branches.	Backtrace.
13	$\boxed{1}$	$\boxed{1}$		Check Next branch.
14	$+$	$+$	Both terms match.	Check Above branch.
15	$\emptyset$	$\emptyset$	No Above branches.	Backtrace.
16	$+$	$+$		Check Below branch.
17	$\emptyset$	$\emptyset$	No Below branches.	Backtrace.
18	$+$	$+$		Check Next branch.
19	$\boxed{*_1}$	x	Possible substitution: $\boxed{*_1} \to x$. Check if $\boxed{*_1} \in \sigma$: $\sigma = \emptyset$.	Add $\boxed{*_1} \to x$ to σ. Check Above branch.
20	$\emptyset$	$\emptyset$	No Above branches.	Backtrace.
21	$\boxed{*_1}$	x		Check Below branch.
22	1	2	Terms do not match.	Match failed. Backtrace.
23	$\boxed{*_1}$	x	Below branch match failed.	Match failed. Backtrace.
24	$+$	$+$	Next branch match failed.	Match failed. Backtrace.
25	$\boxed{1}$	$\boxed{1}$	Next branch match failed.	Match failed. Return 0.

3.4.3 Finding the Most Common Specific Generalization for Substitutions

Finding the MSCG for two substitutions involves looking at both of the substitutions and determining how they are similar to one another. The parts that are similar form the generalization of the two substitutions while the parts that are different form two individual instances of that generalization which, when composed with the generalization, produce the original substitutions. The similar parts include not only identical parts but also terms that share a common MSCG themselves. Thus the MSCG algorithm is made of two separate functions: one that is called by the insertion function which finds the MSCG of two substitutions ($\texttt{MSCG}_{\texttt{Sub}}$); and one that is called by the $\texttt{MSCG}_{\texttt{Sub}}$ function which finds the MSCG of two SLTs, if possible ($\texttt{MSCG}_{\texttt{SLT}}$).

The MSCG algorithm takes two substitution maps τ and ρ and returns a substitution map μ representing the generalization and two substitution maps σ_1 and σ_2 representing two instances of μ that produce τ and ρ respectively. It iterates through each substitution $S_i \to T_i$ in $\tau = \{S_1 \to T_1, \ldots, S_n \to T_n\}$ and compares it to ρ in order to determine if the substitution is similar enough to generalize (and added to μ) or too different and must be individualized (modifying σ_1 and σ_2 appropriately). It does this through calling $\texttt{MSCG}_{\texttt{Sub}}$. Once all substitutions in τ have been considered, the function is finished; the substitutions that remain in ρ which need to be added to σ_2 are done so in the `insert` function as described below. The algorithm is defined mathematically in Figure 3.5.

$\texttt{MSCG}_{\texttt{Sub}}$ can handle the substitution $S_i \to T_i$ in four different ways depending on the content of ρ which Graf names BIND, FREEZE, MIX and DIVIDE. If ρ does not contain the substitution variable S_i then the substitution is unique to τ and is added to σ_1 through BIND (Definition 3.4). If ρ contains an identical substitution to $S_i \to T_i$ then it is added to μ with FREEZE (Definition 3.5). MIX (Definition 3.6) is used when ρ contains a conflicting

$$\begin{aligned}
\texttt{MSCG}(\tau,\rho) :=& \texttt{MSCG}_{\texttt{Sub}}(\{S \to T\}, \rho, \emptyset, \emptyset, \emptyset) \ \forall \ \{S \to T\} \in \tau && (3.3)\\
\texttt{MSCG}_{\texttt{Sub}}(\{S \to T\}, \rho, \mu, \sigma_1, \sigma_2) :=& (\mu, \sigma_1 \cup \{S \to T\}, \sigma_2) \text{ if } S \not\ni \mathsf{DOM}(\rho) && (3.4)\\
\texttt{MSCG}_{\texttt{Sub}}(\{S \to T\}, \rho, \mu, \sigma_1, \sigma_2) :=& (\mu \cup \{S \to T\}, \sigma_1, \sigma_2) \text{ if } S\rho = T && (3.5)\\
\texttt{MSCG}_{\texttt{Sub}}(\{S \to T\}, \rho, \mu, \sigma_1, \sigma_2) :=& (\mu \cup \{S \to M\}, \sigma_1', \sigma_2') \text{ if } S\rho = U \wedge \\
& \texttt{MSCG}_{\texttt{SLT}}(T, U, \sigma_1, \sigma_2, true) = (M, \sigma_1', \sigma_2') && (3.6)\\
\texttt{MSCG}_{\texttt{Sub}}(\{S \to T\}, \rho, \mu, \sigma_1, \sigma_2) :=& (\mu, \sigma_1 \cup \{S \to T\}, \sigma_2 \cup \{S \to S\rho\}) \text{ otherwise.} && (3.7)
\end{aligned}$$

Figure 3.5: The definitions for the MSCG algorithm and $\texttt{MSCG}_{\texttt{Sub}}$ function. MSCG returns a 3-tuple containing the substitution μ which is the generalization of the two substitutions τ and ρ, a substitution σ_1 which, when composed with μ, represents τ, and a substitution σ_2 which, when composed with μ, represents ρ. Definition 3.4 is BIND, definition 3.5 is FREEZE, definition 3.6 is MIX, and definition 3.7 is DIVIDE.

substitution but $\texttt{MSCG}_{\texttt{SLT}}$ can produce a generalization M of the two expressions (see below). The new substitution $S_i \to M$ is added to μ and σ_1 and σ_2 are modified as necessary. Finally, DIVIDE (Definition 3.7) will split two conflicting substitutions that cannot be generalized into σ_1 and σ_2, making no change to μ. Therefore $\texttt{MSCG}_{\texttt{Sub}}$ can always be successful through DIVIDE. In some circumstances, DIVIDE is the case chosen for all substitutions in τ, leaving $\mu = \emptyset$. Figure 3.5 presents the formal definition for the MSCG algorithm and Figure 3.6 displays an example of the algorithm.

MIX must also do some checking to confirm that $S_i \to M$ is not a "redundant" substitution. Specifically, it checks for cases when a single substitution variable is replaced with another single substitution variable (for example, $\boxed{1} \to \boxed{2}$). Since this kind of substitution can continue infinitely, it must be prevented. In such cases, $S_i \to M$ is not added to μ and all instances of M in σ_1 and σ_2 are replaced with S_i. Further redundancy-checking is performed in the `select` function as described below.

The $\texttt{MSCG}_{\texttt{SLT}}$ function is given two SLT structs S and T, the two substitution maps

$$\begin{aligned}
\tau &= \{\boxed{1} \rightarrow 1, \boxed{2} \rightarrow +, \boxed{3} \rightarrow \boxed{5}^{2} + 9, \boxed{4} \rightarrow 4\} \\
\rho &= \{\boxed{2} \rightarrow +, \boxed{3} \rightarrow 3 * 9, \boxed{4} \rightarrow 10\} \\
\mu &= \{\boxed{2} \rightarrow +, \boxed{3} \rightarrow \boxed{5}^{\boxed{6}}\boxed{7}9\} \\
\sigma_1 &= \{\boxed{1} \rightarrow 1, \boxed{4} \rightarrow 4, \boxed{6} \rightarrow 2, \boxed{7} \rightarrow +\} \\
\sigma_2 &= \{\boxed{4} \rightarrow 10, \boxed{5} \rightarrow 3, \boxed{6} \rightarrow \varnothing, \boxed{7} \rightarrow *\}
\end{aligned}$$

Figure 3.6: Example outcome of the MSCG algorithm. τ and ρ are sets of substitutions that could be seen in the middle of the insertion process. Note that they are not expressions by themselves; as substitutions, they could be applied to many different expressions, such as $5 * (\boxed{3}\ \boxed{2}\ \boxed{1})^{\boxed{4}+1}$ or $\boxed{1}^{2}\ \boxed{2}\ \boxed{3} + \cos(\boxed{4} + \boxed{9})$. $\texttt{MSCG}_{\texttt{Sub}}$ uses BIND on $\boxed{1}$, FREEZE on $\boxed{2}$, MIX on $\boxed{3}$ and DIVIDE on $\boxed{4}$. The call to $\texttt{MSCG}_{\texttt{SLT}}$ for $\boxed{3}$ (comparing the conflicting expressions in τ and ρ: $\boxed{5}^{2} + 9$ and $3 * 9$) is enumerated as a step-by-step procedure in Table 3.2.

σ_1 and σ_2 used in $\texttt{MSCG}_{\texttt{Sub}}$, and whether S and T lie along the baseline (b). It returns the generalized SLT M and the two substitution maps σ_1' and σ_2' which are modified to accommodate M. Starting at the root of the two SLTs, the function goes through comparing each pair of corresponding nodes along all of their branches (Above, Below, Next and Argument) through recursion to ensure that the entirety of both expressions are generalized. M is built during the recursion, one node at a time, with each node corresponding to a node in S or T. For each SLT node $S = (s, A_1, B_1, \{X_1, \ldots, X_m\}, N_1)$ and $T = (t, A_2, B_2, \{Y_1, \ldots, Y_n\}, N_2)$, the function compares s and t. If $s = t$ then no more work is necessary; the corresponding node in M is set to the term and the function continues by comparing S and T's branches. If a common substitution for s and t already exists in σ_1 and σ_2 such that $\{v \rightarrow s\} \in \sigma_1$ and $\{v \rightarrow t\} \in \sigma_2$ then the node in M is set to v and the function continues. If s is a substitution variable then $s \rightarrow t$ is added to σ_2, the node in M is set to s and the function continues. Finally, if s and t cannot be generalized in any other way, a new substitution variable v is created, $v \to s$ is added to σ_1, $v \to t$ is added to σ_2, the current node in M is set to v and the function continues.

$\texttt{MSCG}_{\texttt{SLT}}$ fails if, at any point, S and T lie along the baseline and $S = \emptyset \oplus T = \emptyset$ (where $\oplus$ is XOR). This means that the baseline size of the two expressions is not equal and, because of the baseline insertion bias, no generalization can be made. Assuming that it doesn't fail in any of its recursive calls, the function succeeds in producing a generalization for the expressions S and T.

Technically, two different LaTeX functions can be generalized as long as they have the same number of arguments and the MSCG doesn't fail otherwise. For example, $\cos x$ (`\cos{x}`) and $\hat{1}$ (`\hat{1}`) can be generalized to $\boxed{1}\{\boxed{2}\}$, but $\cos x$ and $\frac{1}{2}$ (`\frac{1}{2}`) can't because the former has one argument and the latter has two.

3.4.4 Selecting the Best Branch for Insertion

The `select` function chooses the child of the current substitution tree node that is the most similar to an expression. It is given a substitution tree struct (τ, Σ) (the "current node" in the index, where τ is a substitution contained in that node and Σ is its set of child nodes), a substitution map ρ representing the SLT T to be inserted ($\rho = \{\boxed{0} \to T\}$), and an SLT struct P containing the path from the root of the index to the current node. It returns a substitution tree struct from Σ that is the chosen branch if successful, or nothing otherwise. The function first iterates through each element in Σ and sees if any of them match ρ; if so, the element becomes the chosen branch and select succeeds. If none of τ's children are a match, the function must look for any possible generalizations for ρ among its children. It iterates through each element in Σ again, this time seeing if any of them are the MSCG for ρ; if so, the function must check that the current node is not the index root and that the

Table 3.2: Example of the $\texttt{MSCG}_{\texttt{SLT}}$ function where $S = \boxed{5}^2 + 9$ and $T = 3 * 9$. The created MSCG is $M = \boxed{5}^{\boxed{6}}\boxed{7}9$. Steps 2 and 14 show that σ_1 already contains the substitution $\boxed{1} \to 1$; this was added before $\texttt{MSCG}_{\texttt{SLT}}$ was called (consider the BIND on $\boxed{1}$ in Figure 3.6). While the substitution is not relevant in this example, it is still technically a part of σ_1 and could be relevant if S or T had contained $\boxed{1}$.

Step	S	T	b	Result	Next
1	$\boxed{5}$	3	1	Possible substitution: $\boxed{5} \to 3$. Check if $\boxed{5} \in \sigma_2$: $\sigma_2 = \emptyset$.	Add $\boxed{5} \to 3$ to σ_2. Check Above branch.
2	2	$\varnothing$	0	Check for substitutions $\boxed{\text{i}} \to 2 \in \sigma_1$ and $\boxed{\text{i}} \to \varnothing \in \sigma_2$: $\sigma_1 = \{\boxed{1} \to 1\}$, $\sigma_2 = \{\boxed{5} \to 3\}$.	Add $\boxed{\text{i}} \to 2$ to σ_1 and $\boxed{\text{i}} \to \varnothing$ to σ_2 where $i = 6$. Check Above branch.
3	$\emptyset$	$\emptyset$	0	No Above branches.	Backtrace.
4	2	$\varnothing$	0		Check Below branch.
5	$\emptyset$	$\emptyset$	0	No Below branches.	Backtrace.
6	2	$\varnothing$	0		Check Next branch.
7	$\emptyset$	$\emptyset$	0	No Next branches.	Backtrace.
8	2	$\varnothing$	0		Check Argument branches.
9	$\emptyset$	$\emptyset$	0	No Argument branches.	Backtrace.
10	2	$\varnothing$	0	All branches successful.	Backtrace.
11	$\boxed{5}$	3	1		Check Below branch.
12	$\emptyset$	$\emptyset$	0	No Below branches.	Backtrace.
13	$\boxed{5}$	3	1		Check Next branch.
14	$+$	$*$	1	Check for substitutions $\boxed{\text{i}} \to + \in \sigma_1$ and $\boxed{\text{i}} \to * \in \sigma_2$: $\sigma_1 = \{\boxed{1} \to 1, \boxed{6} \to 2\}$, $\sigma_2 = \{\boxed{5} \to 3, \boxed{6} \to \varnothing\}$.	Add $\boxed{\text{i}} \to +$ to σ_1 and $\boxed{\text{i}} \to *$ to σ_2 where $i = 7$. Check Above branch.
15	$\emptyset$	$\emptyset$	0	Terms match.	Backtrace
16	$+$	$*$	1		Check Below branch.
17	$\emptyset$	$\emptyset$	0	No Below branches.	Backtrace.
18	$+$	$*$	1		Check Next branch.
19	9	9	1	Terms match; do nothing.	Check Above branch.
20	$\emptyset$	$\emptyset$	0	No Above branches.	Backtrace.
...	...	...	...	...	...
33	$\boxed{5}$	3	1	All branches successful.	MSCG successful. Return 1.

$$\begin{aligned}
\texttt{select}((\tau,\Sigma),\rho,P) :=& \{(\tau',\Sigma') \mid (\tau',\Sigma') \in \Sigma \wedge \exists\sigma \in \mathcal{V}(\tau',\rho)\} && (3.8)\\
\texttt{select}((\tau,\Sigma),\rho,P) :=& \{(\tau',\Sigma') \mid (\tau',\Sigma') \in \Sigma \wedge \texttt{MSCG}^*(\tau',\rho) = (\mu,\sigma_1,\sigma_2)\ \wedge \\
& \neg\texttt{index\_root}(P) \wedge \neg\texttt{redundant}(P,P\mu) \wedge P\tau' \neq P\mu\} && (3.9)\\
\texttt{select}((\tau,\Sigma),\rho,P) :=& \emptyset \text{ otherwise.} && (3.10)
\end{aligned}$$

Figure 3.7: The definition for the `select` function. $\texttt{MSCG}^*$ means that the call to MSCG does not allow the function to use its DIVIDE case.

new path to this possible generalization produced by MSCG is not "redundant" (see below). If these checks succeed, the element from Σ becomes the chosen branch and select succeeds; if none of τ's children is a generalization for ρ that passes the two checks, then select fails. The formal definition for the `select` function is shown in Figure 3.7, and an example can be found in Figure 3.8.

It is important to note that, when select calls MSCG, it does not allow the function to use its DIVIDE case (Definition 3.7). Since a generalization can be created from any two substitutions through DIVIDE, it is not a good indicator for finding the node that is most similar to ρ. Also, if the insertion function must DIVIDE in order to add ρ to the index, it will do so with τ and not with one of τ's children. This is why the call to MSCG is denoted in Definition 3.9 with an asterisk.

The `redundant` function is used to guarantee that generalizations proposed by the MSCG in `select` (S) are less specific than the expression represented by ρ (T). This is a constraint that we have added to our implementation of the algorithm; it was not included in Graf's original definitions but was necessary in order for the `select` function to behave properly. This check for redundancy is different from that which already exists in the MSCG algorithm (see MIX as described above) because it considers the previous substitution in its entirety as opposed to individual parts of it. Since S would be added to the index as a parent

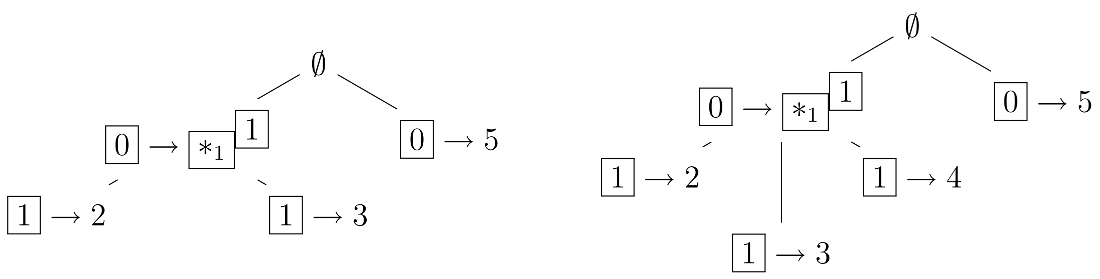

Figure 3.8: *Left:* We want to insert x^4 into a substitution tree which contains x^2, x^3 and 5. Starting at the root node $(\emptyset, \{\{\boxed{0} \rightarrow \boxed{*_1}^{\boxed{1}}\}, \{\boxed{0} \rightarrow 5\}\})$, the `select` function can choose between either of the root's two children – in which case the insertion algorithm will recur using the selected node as the new root – or fail, forcing the new expression to be added as a new child of the root. It determines that $\boxed{0} \rightarrow \boxed{*_1}^{\boxed{1}}$ is the best child node for recursion because $\boxed{*_1}^{\boxed{1}}$ is a generalization of x^4. For the next node $(\{\boxed{0} \rightarrow \boxed{*_1}^{\boxed{1}}\}, \{\boxed{1} \rightarrow 2\}, \{\boxed{1} \rightarrow 3\}\})$, the `select` function fails because it can neither find a generalization of x^4 in the two child nodes nor create a new one, and thus a generalization must be created using $\{\boxed{0} \rightarrow \boxed{*_1}^{\boxed{1}}\}$ through the MSCG algorithm. *Right:* The substitution tree after insertion.

for T, the function must ensure that T is an instance of S since following a branch in the substitution tree is supposed to produce an expression that is more specific with each passing node. Sometimes MSCG will offer an invalid generalization, forcing T to replace a term in S with a substitution variable (for example, $S = \boxed{1} + 5, T = \boxed{1} + \boxed{2}$), and such generalizations must be avoided. Thus the `redundant` function tells the `select` function which proposals from the MSCG function to ignore. The only exception to this test is when the current node the root substitution variable $\boxed{0}$ in order to create the initial set of sub-trees from the root. The formal definition for the `redundant` function is shown in Figure 3.9.

3.4.5 The Insertion Function

The `insert` function remains close to Graf's original implementation as summarized in Section 2.4. It is given a file containing a mathematical expression that has been converted

$$\begin{aligned}
\texttt{redundant}(\emptyset, T) &:= \text{true if } T = (v, A, B, \{X_1, \ldots, X_m\}, N) \wedge v \in \mathbf{V} && (3.11)\\
\texttt{redundant}(S, T) &:= \text{true if } S = (t, A_1, B_1, \{X_1, \ldots, X_n\}, N_1) \wedge t \notin \mathbf{V} \wedge \\
&\quad T = (v, A_2, B_2, \{Y_1, \ldots, Y_n\}, N_2) \wedge v \in \mathbf{V} && (3.12)\\
\texttt{redundant}(S, T) &:= \text{true if } S = (t, A_1, B_1, \{X_1, \ldots, X_n\}, N_1) \wedge && (3.13)\\
&\quad T = (t, A_2, B_2, \{Y_1, \ldots, Y_n\}, N_2) \wedge \\
&\quad \Big(\texttt{redundant}(A_1, A_2) \vee \texttt{redundant}(B_1, B_2) \vee \texttt{redundant}(N_1, N_2) \vee \\
&\quad \texttt{redundant}(X_i, Y_i)\ \forall\ i, 0 \le i \le k \text{ where } k = \texttt{max}(m, n)\Big) && (3.14)\\
\texttt{redundant}(S, T) &:= \text{false otherwise.} && (3.15)
\end{aligned}$$

Figure 3.9: The definition for the `redundant` function.

from LaTeX sytax to SLT syntax as described at the beginning of this section (an example can be found in Figure 3.10). The function reads the content of this file into a new SLT struct E which is then normalized and used to create a new substitution map $\rho = \{\boxed{0} \rightarrow \texttt{normalize}(E)\}$. It finds the current path $P = \boxed{0}\tau$ where (τ, Σ) is the root of the index (or $P = \boxed{0}$ if the root is null). Most of the work is done through the helper function ins which takes the index root (a substitution tree struct), ρ, the domain of ρ as a vector of character pointers (the set of "open variables" OV), and P and returns a new substitution tree struct to replace the index root. We also added E and the name of the input file as additional arguments to the function used solely in the creation of the new substitution tree leaf node that contains ρ in order to store the original expression information for later retrieval. Figure 3.11 shows the formal definition for the `insert` function.

The `ins` function first checks if the index root is null, and, if so, returns $(\rho, \emptyset)$. Then it looks for a match σ between τ and ρ using $\mathcal{V}$. If a match is found and $\Sigma = \emptyset$ then τ is an exact match to ρ and no new node must be added to the index. If a match is found and $\Sigma \neq \emptyset$ then the `select` function is called to determine if any of the nodes in Σ can produce

```
LaTeX:                          SLT:

x_{1}^{2*x}+\frac{1}{2}         x
                                :: REL _
                                :: ARG 1
                                     1
                                ::
                                :: REL ^
                                :: ARG 1
                                     2
                                     *
                                     x
                                ::
                                +
                                :: FN \frac
                                :: ARG 2
                                     1
                                ::
                                :: ARG 1
                                     2
                                ::
```

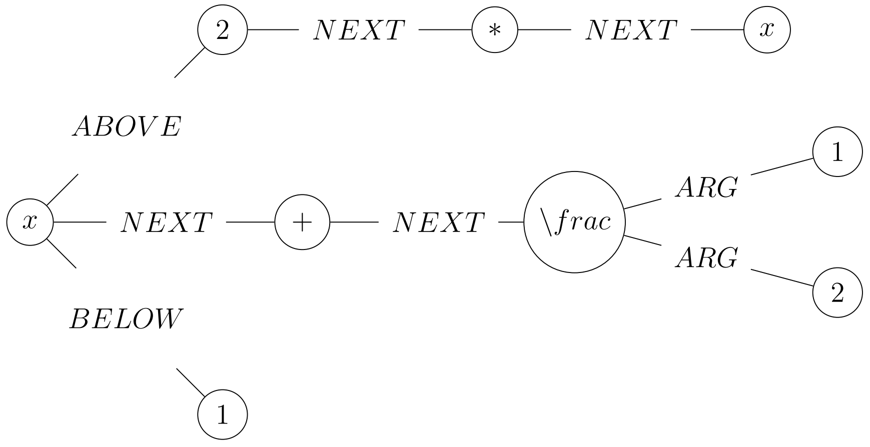

Figure 3.10: Representing an expression in sample file from traditional LaTeX syntax (top left) to the SLT syntax used by our system for input files (top right) and then into a visual SLT representation (bottom).

a match or suitable generalization for ρ. It is given the composition of ρ and σ instead of just ρ so it can be correctly compared to the elements in Σ. If `select` succeeds it returns the child chosen for recursion, (τ', Σ'). The old path P is applied to τ' to generate the new path, OV is updated with new substitution variables found in τ, and (τ', Σ') is removed from Σ. Then the result of the recursive call to ins is added to Σ (in effect replacing the old child) and this updated substitution tree node is returned. If `select` fails then the only node that matches ρ in (τ, Σ) is τ itself, so $\rho\sigma$ (without the substitution variables that are already present in τ) is added to Σ as a new child for τ. If a match is not found, the function must create a generalization for τ and ρ through the MSCG function. The generalization μ replaces τ in the substitution tree node and two new children replace Σ: one containing σ_1 and having Σ as its set of children; the other containing σ_2 combined with ρ (without the substitution variables that were already present in τ) and having no children. This replacement substitution tree node is returned and the function exits. Thus a generalization can always be created if none of the nodes in the branch is a match to the newly inserted expression.

3.5 The Retrieval Algorithm

Our innovations for Graf's retrieval algorithm include using the `search` function not only on the search query (which will only return exact matches), but also on each of the query's sub-expressions, and several *variations* of the query and its sub-expressions that are generated by the retrieval algorithm. These variations are described later in this section; however, it is important to note that all of these variations are expressions that contain substitution variables themselves in order to match different expressions in the index. Since now both sides of our matching comparison (the search query and the expressions in the index) can potentially contain substitution variables, we must now introduce unification.

$$\begin{aligned}
\texttt{insert}((\tau,\Sigma),E) :=& \texttt{ins}((\tau,\Sigma),\boxed{0}\rightarrow\texttt{normalize}(E),\boxed{0},\boxed{0}\tau) && (3.16)\\
\texttt{ins}(\emptyset,\rho,\mathsf{OV},P) :=& (\rho|_{\mathsf{OV}},\emptyset) && (3.17)\\
\texttt{ins}((\tau,\emptyset),\rho,\mathsf{OV},P) :=& (\tau,\emptyset) \text{ if } \exists\sigma\in\mathcal{V}(\tau,\rho) && (3.18)\\
\texttt{ins}((\tau,\Sigma\cup(\tau',\Sigma')),\rho,\mathsf{OV},P) :=& (\tau,\Sigma\cup\texttt{ins}((\tau',\Sigma'),\rho\sigma,\mathsf{IM}(\tau)\cup\mathsf{OV}\backslash\mathsf{DOM}(\tau),P\tau'))\\
& \text{if } \exists\sigma\in\mathcal{V}(\tau,\rho)\wedge(\tau',\Sigma')=\texttt{select}((\tau,\Sigma\cup(\tau',\Sigma')))\\
& && (3.19)\\
\texttt{ins}((\tau,\Sigma),\rho,\mathsf{OV},P) :=& (\tau,\Sigma\cup(\rho\sigma|_{\mathsf{IM}(\tau)\cup\mathsf{OV}\backslash\mathsf{DOM}(\tau)},\emptyset))\\
& \text{if } \exists\sigma\in\mathcal{V}(\tau,\rho)\wedge\emptyset=\texttt{select}((\tau,\Sigma),\rho\sigma,P) && (3.20)\\
\texttt{ins}((\tau,\Sigma),\rho,\mathsf{OV},P) :=& (\mu,\{(\sigma_1,\Sigma),(\sigma_2\cup\rho|_{\mathsf{OV}\backslash\mathsf{DOM}(\tau)},\emptyset)\})\\
& \text{if } \nexists\sigma\in\mathcal{V}(\tau,\rho)\wedge\texttt{MSCG}(\tau,\rho)=(\mu,\sigma_1,\sigma_2) && (3.21)
\end{aligned}$$

Figure 3.11: The definition for the `insert` and `ins` functions. All notation was defined in the previous chapter.

3.5.1 Unification: Matching on Both Sides

The retrieval algorithm uses $\mathcal{G}$ (in order to match identifier variables in the index) to look for matches between a search query and expressions in the index. However, the `match` function is one-sided, making substitutions for only terms in one of its input SLTs (the expressions in the index). To be able to match two expressions where both may contain substitution variables, such as is the case when searching for variations of the search query, the `unify` function must be used instead. This function is similar to the `match` function except that, for its two input SLTs S and T, it makes the same checks for T as it does for S (see the description for the match function in Section 3.4.2 for further details). The unification counterpart for $\mathcal{G}$ is $\mathcal{U}$ which behaves exactly the same way except that it calls the `unify` function instead of the `match` function.

$$\mathcal{U}(\tau,\rho) :=\{\sigma|\forall X_i\in\mathsf{DOM}(\tau),X_i\tau\rho\sigma=X_i\rho\sigma\wedge\sigma\text{ is most general}\} \qquad (3.22)$$

Compared to $\mathcal{G}$ (definition 3.1), $\mathcal{U}$ (definition 3.22) contains σ on both sides of its equality because σ is a unifier instead of simply a matcher.

3.5.2 Ranking Search Results

The search results are ranked by comparing each individual result to the original search query, even if the result was found using a sub-expression or generalization. Comparisons for the ranking algorithm is based on a hybrid of the *bipartite expression representation* [28] and the well-known *bag-of-words* model. Comparing expressions using the bipartite representation makes a list of neighbor relationships for each of the two expressions. Each element in these lists is a 5-tuple (s, n, r, p, b), where s is the symbol, n is a symbol neighboring s, r is the relationship between s and n (above, below, argument or next), p is the position of s along the baseline, and b is a number representing the baseline that gets changed for sub-expressions. For example, $x^{x+1} = \{(x, x, above, 1, 1), (x, +, above, 1, 1), (x, 1, above, 1, 1), (x, +, next, 1, 2), (x, 1, next, 1, 2), (+, 1, next, 2, 2)\}$. The bag-of-words approach also makes a list for each expression, but the elements of this list are merely each of the symbols that appear in the corresponding expression. For example, $x^{x+1} = \{x, x, +, 1\}$.

Both comparison functions calculate a rank between 0 and 1 by finding the set similarity [11] on the number of matches and partial matches between the two lists. Partial matches are found as a percentage matching of tuple elements. Elements themselves can either match fully if they are equal or partially if their terms share a common type (variable, constant, operator, function), so y is a closer match to x than 1. Partial matches are chosen by a greedy matching algorithm. Using the set similarity forces both the number of successful matches and the length of each expression to factor in to the rank. It also causes identical matches to always be ranked at 100%.

The entire ranking algorithm is described in more detail in Section 3.5.6.

3.5.3 Creating Extended Variations of the Search Query

We make multiple variations of the search query by *extending* it in different ways. An expression is extended by adding one or more unused substitution variables to each side of the expression; the extension amount is set by the user and every possible variation is created and used in the retrieval. For example, for the expression $x+1$ and extension amount 2, the variations include $\boxed{1}\,x+1$, $x+1\,\boxed{1}$, $\boxed{1}\,\boxed{2}\,x+1$, $\boxed{1}\,x+1\,\boxed{2}$, and $x+1\,\boxed{1}\,\boxed{2}$. This is important in order to find similar expressions that are longer than the search query, such as $x+1+2$, because the baseline size bias implemented in the indexing algorithm would otherwise cause these expressions to be ignored by the retrieval algorithm. For example, the expression $\boxed{1}\,x+1$ matches $2x+1$ which is similar (and thus relevant) to the original search query $x+1$.

3.5.4 A Detailed Description of the Retrieval Algorithm

The retrieval algorithm is given the name of an SLT file, and the maximum amount the query should be extended. The algorithm produces a vector of search results whose elements are populated by the multiple separate searches that are performed for the query. Each search result is its own C struct containing an SLT struct for the relevant expression, a character pointer for the name of the document in which that expression is located, and a double for the similarity of the result (as a percentage) used in ranking.

The algorithm then calls the search function on the query itself, storing the search results that are returned in the search result vector. Next, it searches for all extended variations of the search query. Then it searches for all sub-expressions of the search query, including all of their nested sub-expressions: for example, the sub-expressions of the query

$x^{x*(y+1)} + y_{n-1}$ are $x*(y+1)$, $y+1$, and $n-1$. Single term sub-expressions are ignored because they skew the results due to their frequency (like searching for "the" in Google): for example, we do not search for any sub-expressions of the query $x^x + y_1$. The algorithm also creates *generalized sub-expression term* variations for each sub-expression. These variations replace the node leading to a sub-expression with a substitution variable and strip away all other terms: for example, the generalized sub-expression term variation for $x^{x+1} + 1$ is $\boxed{1}^{x+1}$. This variation allows the algorithm to find even more expressions containing similar sub-expressions. Finally, the algorithm searches for all variations of all sub-expressions (including nested sub-expressions) and produces the final search result vector that is sorted by rank and then displayed to the user.

The `search` function takes the root of the index (τ, Σ) as a substitution tree struct, the original search query Q and the current search query E as the substitution maps $\xi = \{\boxed{0} \to Q\}$ and $\rho = \{\boxed{0} \to E\}$ respectively, the current path P from the index root to τ as an SLT struct, and the retrieval function $\mathcal{X}$ to be used in this particular search as a function pointer. The retrieval functions include $\mathcal{G}$, $\mathcal{V}$ and $\mathcal{U}$ (although $\mathcal{V}$ is not used). The current search query for the first invocation of the search function is the same as the original search query; however, both are needed to keep track of the search query given by the user when subsequent searches (such as those on sub-expressions) are made.

The function first calls $\mathcal{X}$ on $\{\boxed{0} \to P\}$ and ρ. If $\mathcal{X}$ successfully returns a substitution map (the matcher or unifier), then τ is a match to the query. If $\Sigma = \emptyset$ (the node is a leaf and thus represents a specific mathematical expression) then τ's expression (or expressions if identical expressions from multiple files were inserted into the index) is ranked and added to the search result vector. If $\Sigma \neq \emptyset$ (the node has children and thus is a generalized expression containing one or more substitution variables) then search recurs on each element in Σ. This ensures that only exact matches are included in the results.

While this implementation of the retrieval algorithm worked fairly well, it overlooked a few very relevant expressions during our preliminary test runs. Due to time constraints, we were unfortunately forced to add a major behavioral change to the algorithm that, while improving the quality of search results, decreases time and memory performance significantly. This change was, in addition to searching the index for the query, sub-expressions of the query, and extended variations of the query and its sub-expressions, to also search for variations of the query's *completely generalized baseline*. A more elegant solution to our problem of search result quality can be found in Section 3.5.5.

We create the completely generalized baseline by replacing each term on the baseline of the original search query with a substitution variable. For example, $x^2 + 1$ becomes $\boxed{1}\,\boxed{2}\,\boxed{3}$. We ignore the query's Above, Below and Argument branches during this creation because a substitution variable that lies along the baseline can match terms with or without these branches ($\boxed{1}$ can match any expression with a baseline size of 1, including x, x^2 or $x^{x_z^y}$). We also create completely generalized baselines for the extended variations of the query (so, continuing our example, we would also search for $\boxed{1}\,\boxed{2}\,\boxed{3}\,\boxed{4}$ and $\boxed{1}\,\boxed{2}\,\boxed{3}\,\boxed{4}\,\boxed{5}$, assuming the default extension amount of 2).

Notice that the use of the completely generalized baseline causes many of the previous searches to become unnecessary, because, for example, the results returned by a search for $x^2 + 1$ will always be a subset of the results returned by a search for $\boxed{1}\,\boxed{2}\,\boxed{3}$. In practice, this means that all expressions in the index with the same baseline size as the query (as well as slightly larger sizes, due to the extended queries) are added to the search result vector. While our ranking algorithm ensures that irrelevant results do not appear among the top search results, this unfortunate addition essentially produces a semi-exhaustive search of the

Table 3.3: Examples of the queries generated by the retrieval algorithm using the original search query $x^{x+1}+5$ and the default extension amount (2). The algorithm searches for each of these variations and adds all matches to a vector of search results. The finished vector is sorted by the ranking algorithm based on the matching expression's similarity to the original search query. Notice that the last three queries (starting with $\boxed{1}\ \boxed{2}\ \boxed{3}$) are variations of the completely generalized baseline; using these expressions as a search queries makes many of our other queries (including the original query) unnecessary because they match any expressions with a baseline size of 3, 4 or 5.

Search Queries Generated from $x^{x+1}+5$

Original Query	$x^{x+1}+5$	Extended Sub-Exp.	$x+1\ \boxed{1}\ \boxed{2}$
Extended Original	$\boxed{1}\ x^{x+1}+5$	Generalized Sub-Exp. Term	$\boxed{1}^{x+1}$
Extended Original	$x^{x+1}+5\ \boxed{1}$	Extended Gen. Sub-Exp. Term	$\boxed{1}\ \boxed{2}^{x+1}$
Extended Original	$\boxed{1}\ x^{x+1}+5\ \boxed{2}$	Extended Gen. Sub-Exp. Term	$\boxed{1}^{x+1}\ \boxed{2}$
Extended Original	$\boxed{1}\ \boxed{2}\ x^{x+1}+5$	Extended Gen. Sub-Exp. Term	$\boxed{1}\ \boxed{2}^{x+1}\ \boxed{3}$
Extended Original	$x^{x+1}+5\ \boxed{1}\ \boxed{2}$	Extended Gen. Sub-Exp. Term	$\boxed{1}\ \boxed{2}\ \boxed{3}^{x+1}$
Sub-Expression	$x+1$	Extended Gen. Sub-Exp. Term	$\boxed{1}^{x+1}\ \boxed{2}\ \boxed{3}$
Extended Sub-Exp.	$\boxed{1}\ x+1$	Completely Gen. Baseline	$\boxed{1}\ \boxed{2}\ \boxed{3}$
Extended Sub-Exp.	$x+1\ \boxed{1}$	Completely Gen. Baseline	$\boxed{1}\ \boxed{2}\ \boxed{3}\ \boxed{4}$
Extended Sub-Exp.	$\boxed{1}\ x+1\ \boxed{2}$	Completely Gen. Baseline	$\boxed{1}\ \boxed{2}\ \boxed{3}\ \boxed{4}\ \boxed{5}$
Extended Sub-Exp.	$\boxed{1}\ \boxed{2}\ x+1$		

index and does not fully utilize the generality of Graf's search function.

3.5.5 The Retrieval Algorithm: A More Elegant Solution

Upon post-experiment reflection, we have designed a more elegant solution to improve the quality of our search results without resorting to the semi-exhaustive search enacted by our use of the completely generalized baseline. Our solution is to create *generalized* variations of the search query and its sub-expressions. These variations are similar to the generalized sub-expressions but contain the whole expression (as opposed to just one term) and affect all terms in the expression (as opposed to just terms containing sub-expressions) and are thus more useful. The retrieval algorithm treats these new variations the same way that it treats the extended variations of the query: by creating and searching for each variation individually and adding the results produced by each search to the final search result vector. This would replace our use of the completely generalized baseline, although that could be re-introduced to the algorithm if such behavior was desired.

An expression is generalized by replacing one or more of its terms with substitution variables. The retrieval algorithm would search on a complete set of all possible variations, from replacing just a single term in the expression to replacing all but one of the terms (replacing all of the terms would result in the completely generalized baseline, which is not what we want). An example of a complete set of all possible generalized variations of a search query can be seen in Table 3.4.

3.5.6 A Detailed Description of the Ranking Algorithm

Ranking is done through the `rank` function as shown in Definition 3.23. We use the name "identity" to refer to the lists created by the functions in $\mathcal{F}$ (the bag-of-words and the bipartite representation). An expression's identity is stored in a C struct containing each

Table 3.4: Examples of the generalized variations of the search query $x^{x+1}+5$.

Generalized Forms of $x^{x+1}+5$

$\boxed{1}^{x+1}+5$	$\boxed{1}^{x+1}\ \boxed{2}\ 5$	$\boxed{1}^{x+1}+\boxed{2}$	$\boxed{1}^{\boxed{2}}+5$	$\boxed{1}^{x+1}\ \boxed{2}\ \boxed{3}$	$\boxed{1}^{\boxed{2}}\ \boxed{3}\ 5$
$\boxed{1}^{\boxed{2}}+\boxed{3}$	$x^{\boxed{1}}+5$	$x^{\boxed{1}}\ \boxed{2}\ 5$	$x^{\boxed{1}}+\boxed{2}$	$x^{\boxed{1}}\ \boxed{2}\ \boxed{3}$	$x^{x+1}\ \boxed{1}\ 5$
$x^{x+1}\ \boxed{1}\ \boxed{2}$	$x^{x+1}+\boxed{1}$	$\boxed{1}+5$	$\boxed{1}\ \boxed{2}\ 5$	$\boxed{1}+\boxed{2}$	

element of the identity 5-tuple (s, n, r, p, b): character pointers represent the symbol s and neighbor n while integers represent the relation r (an enumeration), position p and baseline b. For bag-of-words identities, only s is considered; the other fields are ignored. Both identities are created through the `get_identity` function.

The `get_identity` function takes an SLT struct $(t, A, B, \{X_i, \ldots, X_n\}, N)$, current position integer p and current baseline integer b as arguments and returns two vectors of identity structs – one for the bag-of-words and another for the bipartite representation. First it adds (t) to the bag-of-words identity vector. Then it adds (t, t', r, p, b) to the bipartite identity vector for the terms in each node in A, B, $X_1, \ldots, X_n$ and N, where $r = 1$ for terms in the Above branch, $r = 2$ for terms in the Below branch, $r = 3$ for terms in the Next branch, $r = 4$ for terms in any of the Argument branches. Then the function calls itself recursively on N with $p = p + 1$ and b, and on A and B with p and $b = b + 1$.

The `rank_identity` function ranks the identity vectors for two corresponding equations once they have been created. It each combination of elements in the vectors X and Y (which are both either bags-of-words or bipartite representations) and calculates the value for each possible combination of $x \in X$ and $y \in Y$. These values and references to x and y

$$\texttt{rank}(e_1, e_2) = \frac{1}{|\mathcal{F}|} \sum_{f \in \mathcal{F}} \frac{2(\texttt{rank\_identity}(f(e_1), f(e_2)))}{|f(e_1)| + |f(e_2)|} \tag{3.23}$$

Figure 3.12: The ranking algorithm for two expressions e_1 and e_2 which returns a value between 0 and 1 ranking the expressions' similarity to one another, similar to the Tanimoto set similarity metric. $\mathcal{F} = \{\texttt{bag\_of\_words}, \texttt{bipartite\_representation}\}$ contains the identity functions which each take an expression and return the corresponding set of identities for that expression. While these identity functions are combined in our code implementation (see `get_identity`), they are represented in this equation separately to show that it is flexible enough to allow more identity functions to be added in the future.

are stored in a C struct and added to a vector which, once all possible combinations have been evaluated, is sorted based on value. The function then iterates through the this vector and calculates the total ranking using the highest possible value whose corresponding values x and y have not yet been used (a greedy match).

The value for two identities $(s_i, n_i, r_i, p_i, b_i)$ and $(s_k, n_k, r_k, p_k, b_k)$ is calculated by considering each pair in the tuple individually. For r, p, and b, the value is increased by 1 if they are the same or 0 if they are different. For s and n, the value is increased by 1 if they are the same symbol, 0.25 if they are the same symbol type – they're both mathematical variables (x and y), operators ($+$ and $*$), numbers (1 and 2), or functions ($\backslash cos$ and $\backslash frac$) – or 0 if they are different symbols and different symbol types. We chose 0.25 as the intermittent value because we tested 0.5 and felt that it caused the rank to be too high. This total value is divided by 5 to compute a value between 0 and 1. For a bag-of-words identity, only s is compared and the total value is not divided by 5.

```
rank_identity(identity_vector X, identity_vector Y,
              double (*compare_identity)(identity, identity)) {

  for(each element x_i in X) {
    for(each element y_j in Y) {
      value = compare_identity(x_i, y_j)
      % Insert "value" into the vector "values" at index (i, j)
      values(i, j) = value
    }
  }
  % Sort such that the highest value is first (note that this does not
  % change the indexes that correspond to the values)
  sort_descending(values)
  total = 0.0
  for(each element v_k in values from k=0 to k=size(values)) {
    used = false
    % If either identity element has already been matched, skip this value
    for(each element u_k in used_vectors) {
      if(getX(v_k) = getX(u_k) or getY(v_k) = getY(u_k)) {
        used = true
        break
      }
    }
    if(used = false) {
      total += 2 * v_k
      add v_k to used_values
    }
    % If all of the identity elements have been matched, end the loop
    if(size(used_values) > min(size(X), size(Y))) {
      break
    }
  }
  return total / (size(X) + size(Y))
}
```

Figure 3.13: The pseudo-code for the `rank_identity` function. The `compare_identity` function is given as an argument and calculates the value for comparing two individual identities, returning that value as a double. It can be customized depending on the type of identity it must consider (bag-of-words or bipartite representation).

3.6 Summary

In summary, we have implemented Graf's substitution tree insertion and search functions for use in our own indexing and retrieval system for mathematical expressions. We have made some small but necessary changes to the functions while retaining their overall behavior in order to accommodate our transition from indexing predicates to expressions encoded in a layout-based format to preserve the spatial differences of their symbols. We have included a bias in the insertion function in order to populate the substitution tree with expressions effectively. We have modified the retrieval algorithm to conduct searches on the query, sub-expressions of the query and other variations of the query in order to return all relevant search results contained in the index. Finally, we have added a ranking algorithm to sort search results based on the similarity of their symbols and the layout of those symbols to the search query.

In the next chapter we perform an experiment to compare our novel substitution tree MIR system to the mathematical search engine developed by Zanibbi and Yuan [29]. We also present a discussion of these results in which we make observations about our system.

Chapter 4

Results and Discussion

This chapter presents the details of our experiment comparing our substitution tree MIR system to Zanibbi and Yuan's keyword-based MIR system [29]. We then analyze and discuss the results of our experiment.

4.1 Experiment and Results

We used the *precision-at-k* performance metric (which measures the relevance of the top k results retrieved during a search [27]) to test the top $k = 20$ results of 10 test search queries. Our search queries and document database were the same that Zanibbi and Yuan used in their experiment [29]. We evaluated our results through an online survey that was completed by 10 college upperclassmen in the Computer Science, Math and Engineering fields (students with an advanced knowledge of mathematics). The survey contained 20 questions randomly ordered for each participant to prevent effects arising from presenting retrieval results in a fixed order. Each of the 10 search queries appeared twice: once with the top 20 results using Zanibbi and Yuan's Lucene keyword-based system, and again with the top 20 results using our substitution tree system. All expressions were shown as pictures created using `latex2html`. Each question asked the participant to individually judge the 20 search results on their similarity to the query "in terms of both the similarity in symbols between the query and candidate expressions and in their spatial arrangement." Each search result was rated as either "not similar at all," "somewhat similar" or "very similar/identical."

Table 4.1: Precision-at-k ($k = 20$) for 10 search queries (collection: 24,479 expressions from 50 LaTeX documents). Ten participants rated the results as "not similar at all" (0), "somewhat similar" (0.5) or "very similar/identical" (1.0). The mean ratings for the top 5 results and top 20 results for Zanibbi and Yuan's Lucene keyword-based system [29] and our substitution tree system are shown below as percentages.

		Top 5 Results		Top 20 Results	
Query	Expression	Lucene	Sub. Tree	Lucene	Sub. Tree
1	$d,$	80.0	40.0	78.3	29.8
2	$L_1 \times L_2 \times L_3$	43.0	43.0	18.3	17.3
3	$\{v \in W_0^{1,p}(D) :\ v \geq \psi \text{ a.e. in } D\}$	49.0	37.0	25.8	15.8
4	e_{n+1}	32.0	95.0	38.0	65.0
5	$\mathcal{U}'$	50.0	47.0	22.5	18.8
6	$\prod_{y \in \Sigma} k(y) \to \prod_{y \in \Sigma'} k(y)$	40.0	34.0	13.3	9.0
7	$\mathbf{x}^{\mathbf{u}_1}$	61.0	38.0	26.3	12.5
8	$D^n = CS^{n-1}$	38.8	35.8	11.5	13.0
9	$\Omega_{\tau a}$	69.0	78.0	31.0	29.8
10	$\Omega = \frac{h^2}{m}(k_1^2 + k_2^2) + v_0 + v_{2k_2},$	49.0	37.0	14.3	10.5
	Mean (μ)	51.1	48.4	27.9	22.2
	Standard Deviation (σ)	14.9	20.8	19.6	16.7

This rating was converted to the numerical scale (0, 0.5, 1.0) and averaged for each result to calculate the mean rating. The mean rating of the top 5 and top 20 results for each query was averaged to produce the final results that appear in Table 4.1.

We indexed 24,479 expressions from 50 LaTeX documents into a 12 GB file in about 6 minutes using a server with 2 Intel Xeon X5670 2.93 GHz processors with 24 cores and a total of 95 GB of RAM. Our system contained fewer expressions than Zanibbi and Yuan's because it ignores expressions containing only one symbol (which appear far too frequently to be useful) or over 100 symbols (which take up far too much space). Retrieval for nine of the queries took between 2 seconds for query 1 to 2.5 minutes for query 10 (retrieval for query 3 took 12 minutes), largely dependent on the size of the search query and its sub-expressions (all queries except for 3, 6 and 10, took 22 seconds or less), and returned between 4992 search results for query 2 to 13266 search results for query 6. Queries with more sub-expressions seemed to return more results due to the extra searches made using those sub-expressions.

As seen in the table, the substitution tree system performs comparably to the Lucene system in many cases, but can also perform much better or much worse. However, the top 1 result was rated "identical" for all 10 search queries using the substitution tree system, but for only 7 using the Lucene system (queries 1, 4, and 7 did not list the identical match first, but did list it in their top 20). Table 4.2 compares the top 20 results from each system for query 4; tables for the other queries are located in the Appendix (Chapter 6). The Appendix also includes tables showing the specific LaTeX syntax for each expression and the distribution of ratings given by the participants of our experiment for the top 20 results of each query.

Table 4.2: Top 20 Results for Query 4: e_{n+1}

Result	Lucene	Sub. Tree	Sub. Tree Rank
1	$e_{n+1}, e_{n+2}\}$	e_{n+1}	100.0
2	$e_{n+1},\ e_{n+2}\},$	e_{n+1}	100.0
3	$e_{n+2} + \alpha e_1 + e_2$	$e_{n+1},$	90.6
4	$\nu_1 + \nu_2 + \nu_3 = 2$	$e_{n+1}]$	90.6
5	$\nu_1 + \nu_2 + \nu_3 = d$	$e_{n+1}\}$	90.6
6	$e_{n+1},$	e_{n+2}	86.9
7	e_{n+1}	e_{n+2}	86.9
8	e_{n+1}	$\underline{f_{n+1}}$	86.9
9	$e_{n+1},$	X^{n+1}	81.9
10	$\underline{f_{n+1}}$	$\frac{1}{a}e_{n+1}$	80.4
11	$= 2v_{43} + v_{46} - \alpha_{13}\alpha_7 v_{58},$	$\frac{1}{\sqrt{\alpha}}e_{n+1}$	80.4
12	$e_1, e_2, \cdots, e_{n+1}$	$\mathscr{O}_{n+1}$	79.0
13	$V_{43}^{\mathfrak{f}} = \langle \alpha_{13}v_{30} + v_{31}, v_{32}, v_{33} + \alpha_7\alpha_{13}v_{42}, v_{34}, \ldots, v_{41}, v_{58}\rangle,$	$e_{r+1},$	78.8
14	$e_{r+1}, \cdots, e_{n+2}$	$e_{n+2},$	78.8
15	$b_{ij}^1 e_1 + b_{ij}^2 e_2 = [e_1, e_2, \cdots, \hat{e_i}, \cdots, \hat{e_j}, \cdots, e_{n+1}, e_{n+2}]$	$e_{n+2}\}$	78.8
16	$\mathfrak{n} = \mathfrak{g}_{\alpha_1} \oplus \mathfrak{g}_{\alpha_1+\alpha_2} \oplus \mathfrak{g}_{2\alpha_1+\alpha_2} \oplus \mathfrak{g}_{3\alpha_1+\alpha_2} \oplus \mathfrak{g}_{3\alpha_1+2\alpha_2} \cong \mathbb{C}^5,$	$e_{n+2}\}$	78.8
17	$\mathfrak{n} = \mathfrak{g}_{\alpha_1} \oplus \mathfrak{g}_{\alpha_1+\alpha_2} \oplus \mathfrak{g}_{2\alpha_1+\alpha_2} \oplus \mathfrak{g}_{3\alpha_1+\alpha_2} \oplus \mathfrak{g}_{3\alpha_1+2\alpha_2} \cong \mathbb{R}^5,$	$e_{n+2}]$	78.8
18	$f_i = x_1^{\nu_1+1}f_{i,1} + x_2^{\nu_2+1}f_{i,2} + x_3^{\nu_3}f_{i,3} \quad (1 \le i \le 3),$	$e_{n+2},$	78.8
19	$(1)'' \begin{cases} [e_2, \cdots, e_{n+1}] = e_1, \\ [e_1, e_3, \cdots, e_{n+1}] = e_2, \\ [e_1, e_2, \hat{e}_3, \cdots, e_{n+1}] = e_3, \\ [e_1, e_4, \cdots, e_{n+2}] = b_{2,3}^1 e_1, \\ [e_2, e_4, \cdots, e_{n+2}] = b_{2,3}^1 e_2, \\ [e_3, e_4, \cdots, e_{n+2}] = b_{2,3}^1 e_3. \end{cases}$	$e_{n+1},$	76.0
20	$= 2v_{23} + v_{25} - \alpha_1 v_{31} + \alpha_1\alpha_{11}v_{42} + (\alpha_1\alpha_{12} + \alpha_{11}\alpha_2 - \alpha_{13}\alpha_3)v_{58},$	$F^{n+1}.$	74.2

4.2 Discussion

We can observe the main strength of our substitution tree system through query 4 where it vastly outperforms its Lucene counterpart both in the mean ratings obtained from our experiment and the quality of its top 20 search results (as shown in Table 4.2). Our system finds many relevant results through sub-expression matching which is apparent in the consistency of results containing $n+1$ and $n+2$ (sometimes even as a superscript rather than a subscript). It also ranks results identical to query 4 before non-identical results, unlike the Lucene system which ranks six non-identical results higher than the identical results. This may exemplify the detriment of the TF-IDF ranking method: since TF-IDF is heavily influenced by documents in the index, it can rank relevant (and identical) results lower simply because they occur more often. While this might be a good idea for text information retrieval, it does not seem as effective for mathematical expressions, especially since a document referencing the same expression multiple times often increases its importance.

The mean ratings for both the top 5 results and top 20 results of queries 2, 5, 6, and 8 are comparable between the two systems. The Lucene system produced much better results for queries 3 and 10, suggesting that our system has difficulty with larger expressions. Both systems would perform better if the database contained a larger set of more closely related documents (as opposed to a set of 50 random documents in the broad field of physics). Plus, since the queries were selected randomly from the database, some of them are probably not good examples of likely queries that would be entered by real users.

One problem with our experiment is that our system doesn't add single symbol expressions to its index while the Lucene system does. This is why the Lucene system does so much better in our test on query 1: 17 of its top 20 results are the expression d which most of our participants deemed "very similar/identical" to the query (d,). Since our system did not index those expressions, the results were skewed in favor of the Lucene system. However, our rationale for not indexing single symbol expressions – because they appear too frequently to be useful search results – still seems legitimate; besides, searching for an expression like d, is like searching Google for the word “the.” Furthermore, while indexing single symbol expressions would not create many new nodes in our index (due to our use of indicator variables), each of those nodes would contain references to a substantial number of documents (since most LaTeX documents contain single symbol expressions).

Results from our system found through sub-expression matching seem to be more prominent for smaller queries (such as query 4). This is probably due to the behavior of our ranking algorithm which determines the rank of a search result by comparing that result to the original search query. For example, the expression $n + 1$ would be ranked much higher when compared to the query e_{n+1} than to the query $e_{n+1} + 1$ simply because it has greater symbol and size disparities with the latter, but it seems like $n + 1$ is just as relevant to both queries. Thus, instead of ranking sub-expression matches by comparing them to the original search query, our system should compare sub-expression matches to the original sub-expression (and then have the rank weighted somehow so that sub-expressions are not ranked at 100% when they are not identical to the original search query). Additionally, the ranking algorithm should consider the frequency of a sub-expression in a search query so that sub-expression matches are ranked higher if the sub-expression appears more than once in the original search query (for example, compare the expression $n+1$ to the queries $e_{n+1}+1$ and $e_{n+1} + f_{n+1}$).

4.2.1 The Effects of LaTeX Formatting

LaTeX formatting has a great effect on our system as shown in the top 20 results of queries 5, 7 and 9 and their comparatively low mean ratings. Query 5 begins with `\mathcal`, query 7 begins with `\mathbf`, and query 9 begins with \,. Our system emphasizes these terms (perhaps because they appear at the start of the queries) during retrieval. While our system is successful in this aspect – all of the top 20 results for the three queries begin with `\mathcal`, `\mathbf`, and \, respectfully – it affected the experiment because the participants were looking for shared variables rather than shared formatting and therefore rated these queries' results lower. This could have been a flaw in our experiment since we only showed our participants images of the expressions and told them to rate results based on their similarity to the query in symbols and symbol layout, not formatting.

This all could have been avoided if our system simply ignored these formatting terms. For example, the result ranked fourth by our system for query 9 (as seen in Table 6.12) looks identical to query 9 yet is still ranked lower than non-identical results. However, we can see why when we look at the original LaTeX for the two expressions:

```
         Query:  \,\Omega_{\tau a}
 Fourth Result:  \,\Omega_{\tau a}\,
```

Since the \, adds two extra terms to the expression, our system ranks it as only 73.2% similar to a query to which it should be identical. Thus it might be prudent for our system to ignore such terms when indexing expressions because formatting has such a significant effect on our system's retrieval and ranking algorithms while having little to no effect on the visual and mathematical similarity of two expressions. On the other hand, if a user enters a formatting term in a search query, that user likely wants to see results which share the same formatting; this seems rather unlikely, though.

4.2.2 The Shortcomings of the Insertion Bias

Our addition of a bias on the baseline size of an expression during its insertion into the index also had a major and noticeable impact on our system's search results. Most of the top 20 results produced by our system are visually similar in size to the query, while the results produced by the Lucene system vary dramatically. Since the searches for our system used the default extension amount of 2, their results were restricted to a baseline size equal to the query or up to two greater than the query unless they were matched on a search of one of the query's sub-expressions. This hurt our results because they seem to miss some relevant results that the Lucene system finds – including results 5 and 10 for query 2, results 4-10 for query 3 and results 4 and 5 for query 10 (see Tables 6.5, 6.6 and 6.13) – simply because they have larger baseline sizes.

The obvious solution would be to have our system conduct its searches using a larger extension amount. However, what's the right amount in order to retrieve all relevant results from the index while maintaining low time costs? For our system to find the same result for query 3 that the Lucene system found and ranked as its 9th result, we would need to set the extension amount to 18; compared to the time costs we already experienced for our searches at extension amount 2, the increase would be ridiculous. The main problem is that our retrieval algorithm cannot fully compensate for the effects of the insertion bias on indexing. While we chose this bias in part because the disparity in size between a search query and search result has a significant impact on that result's relevancy (results often become less relevant as they become larger because of all the superfluous symbols), we see from our experiment that this is not always the case, especially when the query itself (or an expression very similar to the query) is contained within a larger expression (as seen in the Lucene system's results for query 3).

Note that this problem has to do with how expressions are retrieved from the index (and how they are indexed in the first place), not how expressions are ranked. We have full confidence in the effectiveness of our ranking algorithm (besides the modifications with ranking sub-expression matches as noted above) – we just need to be able to find relevant results from the index before we can rank them.

4.3 Summary

Our experiment demonstrates that our novel substitution tree MIR system is comparable to Zanibbi and Yuan's keyword-based vector-space Lucene MIR system. As exemplified in our system's perfect performance in the top 1 search results, our system's ranking algorithm, which is based only on structural similarity, seems to work better than the Lucene system's ranking algorithm, which is based in part on an expression's frequency in the index. Our system's retrieval algorithm successfully finds results using the sub-expressions of search queries, providing further strength to the quality of our search results. Unfortunately, our system is negatively affected by two major factors: the presence of LaTeX formatting in our search queries and the expressions in our index, and the impact of the insertion bias on the results that our retrieval algorithm is able to find. However, these are problems that could be addressed and fixed in future implementations of our substitution tree MIR system.

Overall, while our experiment neither supports nor refutes our hypothesis that a substitution tree indexing model would improve the relevance of search results over Zanibbi and Yuan's vector-space model, it does support our belief that further refinement of our system is warranted.

Chapter 5

Conclusion and Future Work

Our hypothesis was that a substitution tree indexing model would improve the relevance of search results over Zanibbi and Yuan's vector-space model. While our experiment neither supported nor refuted this hypothesis, we believe that further refinement of our novel substitution tree MIR system using what we have learned through our experiment would provide validation. Our implementation presents a promising first attempt at layout-based substitution tree indexing and retrieval: it is comparable to another leading MIR system and effective at finding relevant search results, especially results identical to the search query and results relevant to the query's sub-expressions.

Our contributions include: the design and implementation of substitution trees for layout-based mathematical expressions (Chapter 2); the design and implementation of a substitution tree indexing algorithm for layout-based mathematical expressions, including the introduction of an insertion bias (Chapter 3); the design and implementation of a substitution tree retrieval algorithm for layout-based mathematical expressions, including behavior that retrieves expressions from the tree that are not identical yet still relevant (similar) to a search query (Chapter 3); an experiment comparing our substitution tree MIR system to a leading MIR system (Chapter 4); and a discussion analyzing the results of that experiment and explaining specific ideas of how our system can be improved in future research based on the findings from our experiment (Chapters 4 and 5).

Overall, we believe that we have provided a good foundation for future research in substitution tree indexing and retrieval for mathematical expressions.

5.1 Conclusions and Future Work on the Insertion Bias

Our insertion bias on the size of an expression's baseline causes a significant negative impact on the quality of search results our system produces. This is because we can't determine how relevant an expression is to a search query simply based on its size. Since our indexing algorithm segregates expressions by their size, our retrieval algorithm can miss expressions that should be returned as relevant search results. While size does play a role, we must examine the other factors that lead the human mind to draw similarities between two expressions. Consider the following expression taken from [10]:

$$\text{score}(\vec{d},\vec{q}) = \frac{\sum_{k=1}^{m} d_k \cdot q_k}{\sqrt{\sum_{k=1}^{m}(d_k)^2} \cdot \sqrt{\sum_{k=1}^{m}(q_k)^2}}$$

If asked to simplify this expression through substitutions, most of us would probably follow the same progression:

$$a = \frac{\sum b \cdot c}{\sqrt{\sum b^2} \cdot \sqrt{\sum c^2}} \rightarrow a = \frac{b}{\sqrt{c} \cdot \sqrt{d}} \rightarrow a = \frac{b}{c \cdot d} \rightarrow a = \frac{b}{c} \rightarrow a = b$$

Where we decide to end depends on how much emphasis we put on the different symbols in the expression: the summation signs, the square roots, the multiplication in the denominator, and the fraction. Perhaps it's easy to get past that first step because of the similarity between $\sum_{k=1}^{m} d_k \cdot q_k$, $\sum_{k=1}^{m}(d_k)^2$ and $\sum_{k=1}^{m}(q_k)^2$ or the importance of the summation signs to the overall expression. Perhaps it's easy to substitute out the square roots because they're simply a function on the variables c and d. We could rewrite this progression in another way:

$$\boxed{1} = \frac{\boxed{2}}{\boxed{3}} \rightarrow \boxed{1} = \frac{\boxed{2}}{\boxed{4}\cdot\boxed{5}} \rightarrow \boxed{1} = \frac{\boxed{2}}{\sqrt{\boxed{6}}\cdot\sqrt{\boxed{7}}} \rightarrow \boxed{1} = \frac{\sum\boxed{8}\cdot\boxed{9}}{\sqrt{\sum\boxed{8}^2}\cdot\sqrt{\sum\boxed{9}^2}}$$

Notice that this looks like the branch of a substitution tree with specific substitutions made along the way. We'd expect all expressions with the form $\boxed{1} = \frac{\boxed{2}}{\boxed{3}}$ to be inserted somewhere in this branch, creating new sub-branches as necessary, because that would make sense to us. However, our current indexing algorithm wouldn't allow that to happen due to the insertion bias: it would group $x = \frac{y}{z}$ and $x + 5 = \frac{y}{z}$ in completely separate branches due to their difference in baseline size (since it wouldn't permit the substitution $\boxed{1} \rightarrow x + 5$).

So the real challenge is to design an insertion bias that will group expressions based on similarity in a way that makes sense, be it in terms of expression size and symbol layout, an expression's corresponding operator syntax, an expression's sub-expressions (possibly even with a redefinition of sub-expression, such as the set of substrings of an expression), or some other method. This will be the most important part of future research on indexing with substitution trees, and the success of a new insertion bias will determine the success of the system as a whole. Maybe instead of focusing on the size of an expression's baseline we simply need to identify certain functions and operators on which to bias insertion; maybe it's something more than that. The solution lies in understanding and replicating how our mind naturally simplifies mathematical expressions.

5.2 Future Work on Substitution Tree Indexing and Retrieval

Any reconstructions of our system should incorporate generalized variations of search queries in the retrieval algorithm. See Section 3.5.5 for a detailed description. Such reconstructions could also experiment with using other search query variations to produce more relevant search results.

There are many other opportunities for future research concerning our system. First, the index could be expanded from relying on an extensive document database to scraping the World Wide Web to construct an even larger and more diverse index. The scraper could start with the entire arXiv library of $\LaTeX$ documents (from which our document database was constructed). Regularly updating the index would ensure that it contains expressions from the most current technical, mathematical and scientific papers. Second, a user interface could be created that allows users to enter search queries more easily. Since currently queries must be entered in $\LaTeX$ syntax through a terminal prompt, our system is not very user-friendly, especially to potential users who don't have advanced mathematical knowledge. This user interface could also include additional options on how the retrieval and ranking algorithms should behave: for example, a user might want to emphasize search results that match specific symbols and ignore results that do not contain those symbols. Third, our system could be streamlined by an expert low-level programmer to be more time and memory efficient.

Unlike the system created by Kohlhase and Sucan [14], we do not add sub-expressions to our substitution tree index. This would dramatically increase the size of our index, and the added benefit to the retrieval algorithm would largely overlap with its current behavior of searching for the sub-expressions of a query along with the query itself. However, this might be worth exploring for other cases which would otherwise be overlooked. For example, if we search for the expression x^{x+1} in an index containing the expression $x * (x + 1)$, the latter would not be returned as a relevant result unless the user set the extension amount to 4 (so that the expression matches our extended sub-expression search query $\boxed{1}\ \boxed{2}\ \boxed{3}\ x + 1\ \boxed{4}$). If sub-expressions had been added to the index, then the expression $x + 1$ would have been inserted as a sub-expression of $x * (x + 1)$ and would match any search for $x + 1$. Thus this could be a useful addition in the future.

Some of Kohlhase and Sucan's other innovations could also be useful for incorporation into our system. For example, their system is designed to accept generalized search queries (already containing a substitution variable) for when a user only remember parts of an expression. This feature could easily be implemented in our retrieval algorithm.

Another important problem with our system becomes apparent when considering the `apply` function described in the previous section: making substitutions for SLTs with Next branches (a baseline size larger than one) is done incorrectly in situations where the substitution variable has an Above, Below or Argument branch. For example, applying the substitution $\boxed{1} \rightarrow x + 1$ to the expression $x^{\boxed{1}^{2}}$ will produce x^{x+1^2} instead of $x^{(x+1)^2}$. Note that, with our baseline insertion bias, this situation will never happen with terms in the baseline itself. The most efficient way of fixing this mistake would be to completely overhaul the system by changing the fundamental structure of SLTs to allow them to contain another SLT instead of a term, thereby making it easier to substitute nested expressions correctly in such circumstances.

While we believe that our current ranking algorithm (based on a combination of the bag-of-words and bipartite representation methods) is effective, there is always room for improvement. Fortunately, the abstraction present in our `rank_identity` function allows the implementation of additional identity functions that would cause the rank to be weighted based on other factors. For example, creating an identity function that prefers matching neighboring symbols over matching symbols spread across an expression could improve the quality of search results.

Finally, our normalization of mathematical variable names could be extended for other symbol types such as constants or operators. Each of these symbols would be normalized to their own corresponding type of indicator variable so an indicator variable that replaces a mathematical variable cannot match an indicator variable that replaces another symbol type. This would make it easier for the retrieval algorithm to match similar expressions, and the ranking function would ensure that expressions with symbols that match the query will still be ranked higher than those without matching symbols. For example, the query $x+1$ seems more similar to $x+2$ than to $x+x$ because, since the non-matching symbol is a constant, expressions which replace that symbol with another constant seem more relevant. The same is true for $x+y$ and $x-y$ because of the mathematical similarity of plus and minus.

Chapter 6

Appendix

The tables provided in this chapter show the results of our experiment in greater detail including both the side-by-side image comparison of and the ratings distributions given by participants for the top 20 results for each of our search queries using the two MIR systems.

Table 6.1: The mean ratings (including standard deviation, abbreviated as SD) for the top 5 and top 20 results for Zanibbi and Yuan's Lucene keyword-based system [29] and our substitution tree system are shown below as percentages. The mean and standard deviation of each column is shown in the bottom two rows.

	Top 5				Top 20			
	Lucene		Sub. Tree		Lucene		Sub. Tree	
Query	Mean	SD	Mean	SD	Mean	SD	Mean	SD
1	80.0	0.0	40.0	33.5	78.3	11.3	29.8	16.7
2	43.0	37.4	43.0	39.0	18.3	24.7	17.3	23.8
3	49.0	36.7	37.0	43.4	25.8	27.2	15.8	23.8
4	32.0	32.5	95.0	5.0	38.0	38.0	65.0	25.6
5	50.0	34.8	47.0	30.3	22.5	25.8	18.8	23.4
6	40.0	35.5	34.0	41.7	13.3	23.9	9.0	24.3
7	61.0	21.6	38.0	48.0	26.3	29.4	12.5	27.5
8	38.8	50.0	35.8	37.8	11.5	28.2	13.0	22.2
9	69.0	34.9	78.0	21.7	31.0	32.3	29.8	37.0
10	49.0	38.3	37.0	47.1	14.3	27.4	10.5	26.9
Mean (μ)	51.1	32.2	48.4	34.8	27.9	26.8	22.2	25.1
SD (σ)	14.9	13.2	20.8	13.2	19.6	6.8	16.7	5.1

Table 6.2: The mean ratings (including standard deviation, abbreviated as SD) for the top 5 and top 20 results for Zanibbi and Yuan's Lucene keyword-based system [29] and our substitution tree system are shown below as percentages. As opposed to Table 6, the means and standard deviations in this table were calculated by ignoring results rated as "not similar" (0). The "#" columns show the number of results rated as "somewhat similar" (0.5) or "very similar/identical" (1.0) – the number of "unweighted matches." The mean and standard deviation of each column is shown in the bottom two rows.

	Top 5						Top 20					
	Lucene			Sub. Tree			Lucene			Sub. Tree		
Query	Mean	SD	#	Mean	SD	#	Mean	SD	#	Mean	SD	#
1	100.0	0.0	5	100.0	-	1	92.5	18.3	20	58.3	20.4	6
2	66.7	28.9	3	66.7	28.9	3	62.5	25.0	4	66.7	28.9	3
3	62.5	25.0	4	75.0	35.4	2	55.6	16.7	9	75.0	35.4	2
4	50.0	0.0	2	100.0	0.0	5	72.2	26.4	9	66.7	24.3	18
5	66.7	28.9	3	75.0	35.4	2	60.0	22.4	5	75.0	35.4	2
6	62.5	25.0	4	75.0	35.4	2	62.5	25.0	4	75.0	35.4	2
7	60.0	22.4	5	100.0	0.0	2	55.6	16.7	9	100.0	0.0	2
8	100.0	0.0	2	75.0	35.4	2	100.0	0.0	2	75.0	35.4	2
9	75.0	28.9	4	70.0	27.4	5	62.5	23.2	8	61.1	22.1	9
10	100.0	0.0	2	75.0	35.4	2	100.0	0.0	2	75.0	35.4	2
Mean (μ)	74.3	15.9	3.4	81.2	25.9	2.6	72.3	17.4	7.2	72.8	27.3	4.8
SD (σ)	18.8	13.9	1.2	13.3	15.0	1.4	18.1	9.8	5.3	11.5	11.4	5.2

Table 6.3: The search queries used in our experiment, shown how they are written in LaTeX syntax and how they are rendered in LaTeX.

Query	LaTeX Rendering	LaTeX Syntax
1	$d,$	`d,`
2	$L_1\times L_2\times L_3$	`L_1\times L_2\times L_3`
3	$\{v\in W^{1,p}_{0}(D):\ v\geq \psi\ \mbox{a.e.}\ \mbox{in}\ D\}$	`\{v\in W^{1,p}_{0}(D): \ v\geq \psi \ \mbox{a.e.} \ \mbox{in} \ D\}`
4	e_{n+1}	`e_{n+1}`
5	${\mathcal U}'$	`{\mathcal U}'`
6	$\displaystyle \prod_{y\in \Sigma} k(y)\to \prod_{y\in \Sigma'} k(y)$	`\displaystyle \prod_{y\in \Sigma} k(y)\to \prod_{y\in \Sigma'} k(y)`
7	$\mathbf{x}^{\mathbf{u}_1}$	`\mathbf{x}^{\mathbf{u}_1}`
8	$D^n=CS^{n-1}$	`D^n=CS^{n-1}`
9	$\,\Omega_{\tau a}$	`\,\Omega_{\tau a}`
10	$\displaystyle \Omega=\frac{h^2}{m}(k_1^2+k_2^2)+v_0+v_{2k_2},$	`\displaystyle \Omega=\frac{h^2}{m} (k_1^2+k_2^2)+v_0+v_{2k_2},`

Table 6.4: Top 20 Results for Query 1: $d,$

Result	Lucene	Sub. Tree	Sub. Tree Rank
1	d	$d,$	100.0
2	d	$X,$	73.8
3	d	$S,$	73.8
4	d	$I,$	73.8
5	d	$\rho,$	73.8
6	d	$s,$	73.8
7	d	$A,$	73.8
8	d	$E,$	73.8
9	d	$V,$	73.8
10	d	$G,$	73.8
11	d	$x,$	73.8
12	d	$V,$	73.8
13	dn	$n,$	73.8
14	d	$K,$	73.8
15	d	$\Omega,$	73.8
16	$d,$	$g,$	73.8
17	$d.$	$W,$	73.8
18	d	$T,$	73.8
19	d	$A,$	73.8
20	d	$s,$	73.8

Table 6.5: Top 20 Results for Query 2: $L_1 \times L_2 \times L_3$

Result	Lucene	Sub. Tree	Sub. Tree Rank
1	$L_1 \times L_2 \times L_3$	$L_1 \times L_2 \times L_3$	100.0
2	$E_1 \times E_2 \times E_3$	$E_1 \times E_2 \times E_3$	77.3
3	$\underline{GL_3}$	$G = G_1 \times G_2$	60.2
4	$L_1 \gg L_2$	q_1, q_2, q_3	58.6
5	$\boldsymbol{L} = (L_1, L_2, L_3)$	$\sigma_1, \sigma_2, \sigma_3$	58.6
6	L_2	$\underline{f_1, f_2, f_3}$	58.6
7	L_2	e_1, e_2, e_3	58.6
8	L_2	t_1, t_2, t_3	58.6
9	L_1	E_1, E_2, E_3	58.6
10	(L_1, L_2, L_2)	$\underline{j_1, j_2, j_3}$	58.6
11	$k_2 = (0, n_2/L_2, n_3/L_2)$	i_1, i_2, i_3	58.6
12	$L_1 = \pi^{-1}(U_1) \cong U_1 \times \mathbb{C}_{t_1}$	$\underline{e_1, e_2, e_3}$	58.6
13	$L_2 = \pi^{-1}(U_2) \cong U_2 \times \mathbb{C}_{t_2}$	$\underline{a_1, a_2, a_3}$	58.6
14	$E_0 \otimes_K L = E$	$\underline{a_1 < a_2 = a_3}$	58.6
15	Λ_1	$\underline{e_1, e_2, f_3}$	58.6
16	$\underline{U^{perp} subsetU.}$	$\underline{e_2, e_3, f_1}$	56.9
17	$\Phi = \{\alpha_1, \alpha_2\} \subset \Lambda = \{\alpha_1, \alpha_2, \alpha_3\}$	$L^1(I \times \Omega)$	56.8
18	L_p	$t_1 t_2 t_3 = c$	56.7
19	y_L	$F_2 = L_2 \cdot o$	56.4
20	$\underline{L_0}$	$D_1 \supset D_2 \supset D_3 \cdots$	55.8

Table 6.6: Top 20 Results for Query 3: $\{v \in W_0^{1,p}(D) :\ v \geq \psi \text{ a.e. in } D\}$

Result	Lucene	Sub. Tree	Sub. Tree Rank
1	$\{v \in W_0^{1,p}(D) :\ v \geq \psi \text{ a.e. in } D\}$	$\{v \in W_0^{1,p}(D) :\ v \geq \psi \text{ a.e. in } D\}$	100.0
2	$K_\epsilon = \{v \in W_0^{1,p}(D) : v \geq 0 \text{ a. e. on } T_\epsilon\}$	$K_\epsilon = \{v \in W_0^{1,p}(D) : v \geq 0 \text{ a. e. on } T_\epsilon\}$	78.7
3	$\mathcal{F}_0(u^0) = \inf_{v \in W_0^{1,p}(D)} \mathcal{F}_0(v).$	$\underline{\{(j/p)_0(overline 1_0), a\} inpW.}$	64.3
4	$\{\phi \in C_0^1(D) :\ \phi_- \in C_0^1(D) \text{ and } \phi \geq \psi \text{ a.e. in } D\}$	$X = \{\, x \in X^* \ :\ \beta(x) < 0 \,\}.$	63.1
5	$\min\{\int_D \frac{1}{p}\lvert\nabla u\rvert^p dx - \int_D fudx : u \in W_0^{1,p}(D),\ u \geq 0 \text{ a. e. in } T_\epsilon(\omega)\}$	$\{(x_1, \ldots, x_n) \in \mathbb{R}^n \mid \sum x_i^2 = 1, x_i \geq 0\ \forall i\}$	61.7
6	$\min\{\int_{\mathbf{R}^n} \frac{1}{p}\lvert\nabla v\rvert^p - fvdx :\ u \in W_0^{1,p}(D),\ u \geq 0 \text{ a. e. in } T_\epsilon(\omega)\}$	$\{(z,p) \in \mathbb{C} \times \tilde{S} : 0 < \operatorname{Im} z\}$	60.3
7	$\min\{\int_D \frac{1}{p}\lvert\nabla v\rvert^p - fvdx :\ v \in W_0^{1,p}(D) \text{ and }\ v \geq \psi^\epsilon\}$	$\mathcal{S}_d := S(V_d)^{\mathfrak{u}_2} = \{v \in S(V_d) \vert D_1(v) = 0\},$	60.1
8	$\min\{\int_{\mathbf{R}^n} \frac{1}{p}\lvert\nabla v\rvert^p - fvdx :\ v \in W_0^{1,p}(D),\ v \geq \psi^\epsilon \text{ a. e. in } D$	$\underline{xinOmega_T = \{xinV^n \vert (x,x) = T\}.}$	60.1
9	$\min\{\int_D \frac{1}{p}\lvert\nabla v\rvert^p + \frac{1}{p}\alpha_0 v_-^p - fvdx :\ v \in W_0^{1,p}(D) \text{ and } v \geq \psi \text{ a. e. in } D\}$	$W_1^\perp = \{\mathbf{v} = (v_1, \ldots, v_n) \in (\mathbb{R}^k)^n \mid \sum v_i = 0\}$	60.1
10	$\min\{\int_D \frac{1}{p}\lvert\nabla v\rvert^p + \frac{1}{p}\alpha_0 v_-^p - fvdx :\ v \in W_0^{1,p}(D) \text{ and } v \geq \psi \text{ a. e. in } D\}.\quad \square$	$X = \{\, x \in X^* \ :\ \alpha(x) < 0 \,\}$	60.1
11	$w_\delta^\epsilon(x,\omega) = \inf\{v(x) :\ \triangle_p v \leq \alpha_0 + \delta \text{ in } D_\epsilon,\ v \geq 1 \text{ on } T_\epsilon \text{ and } v = 0 \text{ on } \partial D\}$	$X = \{\, x \in X^* \ :\ \beta(x) < 0 \,\}$	60.1
12	$w_\theta^\epsilon \in W_0^{1,p}(D)$	$\{(x,y) \in X \times Y \mid x = * \text{ or } y = *\}$	59.4
13	$v_{\alpha_0,D}^\epsilon(x,\omega) \geq h_k^\epsilon(x,\omega) - o(1), \text{ a. e. } x \in B_{\frac{\epsilon}{2}}(\epsilon k) \text{ and a. s. } \omega \in \Omega$	$\{(v_1, \ldots, v_n) \in ([-1,1]^k)^n \mid \sum v_i^1 = 0\}$	59.0
14	$\min\{\int_D \frac{1}{p}\lvert\nabla v\rvert^p + \frac{1}{p}\alpha_0 v_-^p - fvdx :\ \forall\ v \in W_0^{1,p}(D)\}\quad \square$	$v_{\alpha',A}^\epsilon(x,\omega) \leq v_{\alpha,A}^\epsilon(x,\omega),\ a.\ e.\ x \in A.$	58.5
15	$\min\{\int_D \frac{1}{p}\lvert\nabla v\rvert^p + \frac{1}{p}\alpha_0 v_-^p - fvdx :\ \forall\ v \in W_0^{1,p}(D)\}$	$H = \left\{ \begin{pmatrix} x & \alpha y \\ y & x \end{pmatrix} : x^2 - \alpha y^2 = 1 \right\}.$	58.4
16	$\overline{v}_{\alpha_0+\delta,D}^\epsilon = \min(v_{\alpha_0+\delta,D}^\epsilon,\ 1) \textit{ converges to } 0 \textit{ in } L^p(D)$	$X = \{\, x \in X^* \ :\ -C < \alpha(x) < 0 \,\}$	58.2
17	$w^\epsilon(x,\omega) = \inf\{v(x) :\ \triangle_p v \leq \alpha_0 \text{ in } D_\epsilon,\ v \geq 1 \text{ on } T_\epsilon \text{ and } v = 0 \text{ on } \partial D\}$	$\lambda_i \in k^*\,,\ n_i \in \{\,0\,,\,1\,\}$	58.1
18	$\begin{aligned} \int_D \lvert\nabla w^\epsilon\rvert^p dx &\leq (\int_D \lvert\nabla w^\epsilon\rvert^p)^{\frac{p-1}{p}} \cdot (\int_D \lvert\nabla h^\epsilon\rvert^p dx)^{\frac{1}{p}} \\ &+ \alpha_0 \int_D (h^\epsilon - w^\epsilon) dx \\ &\leq \frac{p-1}{p} \int_D \lvert\nabla w^\epsilon\rvert^p dx + \frac{1}{p} \int_D \lvert\nabla h^\epsilon\rvert^p dx + C\alpha_0 \end{aligned}$	$\Gamma_g(n) = \{M \in \Gamma_g \mid M \equiv \mathbf{1}_{2g} (\operatorname{mod} n)\}.$	57.9
19	$v_{\alpha,B_1}^\epsilon \geq \frac{\beta}{c(n,p)} \lvert x - x_0\rvert^{\frac{p}{p-1}} - \frac{\beta}{c(n,p)} > 0 \text{ in } B_1(x_0).$	$\{(i,j,s) : i \leq n, j \leq m, s = 0,1\}$	57.8
20	$F' \geq 0, F' \in C_0^\infty\big((\tfrac{1}{2}, \tfrac{3}{4})\big)$	$\deg(v) = \lvert\{w \in V_0 \mid \{v,w\} \in E_o\}\rvert$	57.6

Table 6.7: Top 20 Results for Query 5: $\mathcal{U}'$

Result	Lucene	Sub. Tree	Sub. Tree Rank
1	$\mathcal{U}'$	$\mathcal{U}'$	100.0
2	$\mathcal{U}' = \{U \in \mathcal{U} \mid U \subseteq Y' \cap \Sigma\}$	$\mathcal{F}'$	82.5
3	$Y_{\mathcal{U}} \cong Y'_{\mathcal{U}}$	$\mathcal{G}'$	82.5
4	$\mathcal{U}$	$\mathcal{V}'$	82.5
5	$\mathcal{F}'$	$\mathcal{M}'$	82.5
6	$\mathcal{G}'$	$\mathcal{T}'$	82.5
7	$\mathcal{V}'$	$n\rho'$	63.3
8	$\mathcal{M}'$	$\mathcal{T}',$	62.1
9	$\mathcal{X}'$	ϕ''	59.2
10	$\mathcal{F}'_{\mathcal{U}} := \pi'^{*}\mathcal{F}'$	$\mathcal{T},$	59.2
11	$\mathcal{U} \supset \mathcal{L}$	$\mathcal{P}/A$	57.9
12	$\mathcal{U} \supseteq \mathcal{L}$	$\mathcal{X}/R$	57.9
13	$\mathcal{X}_{\mathcal{U}}$	H''	57.5
14	$\pi' : X_{\mathcal{U}} \to X_{\eta}$	$\lambda \in U$	55.8
15	$H^{i}(X_{\mathcal{U}}, \pi'^{*}\mathcal{F}) \cong H^{i}(X_{\eta}, \mathcal{F}) \otimes k(\mathcal{U})$	$U = \emptyset$	55.8
16	$\mathcal{U}' = \{S \in \beta(\Sigma) \mid S \cap \Sigma \in \mathcal{U}\}$	$\mathcal{C}_{\Omega} =$	55.7
17	$H^{i}(X_{\mathcal{U}}, \pi'^{*}\mathcal{F})$	$\tilde{f}'_{0}$	55.7
18	$\prod H^{i}(X_{y}, \mathcal{F}' \vert_{X_{y}})/\mathcal{U}$	$K \subset U$	54.2
19	$X_{\bullet,\mathcal{U}} \to X_{\mathcal{U}} \supset Z_{\mathcal{U}}$	$0''$	53.3
20	$S - T \in \mathcal{U}$	$\emptyset \in \mathcal{U}$	51.9

Table 6.8: Results 1-10 for Query 6: $\prod_{y\in\Sigma} k(y) \to \prod_{y\in\Sigma'} k(y)$

Result	Lucene	Sub. Tree	Sub. Tree Rank
1	$\prod_{y\in\Sigma} k(y) \to \prod_{y\in\Sigma'} k(y)$	$\prod_{y\in\Sigma} k(y) \to \prod_{y\in\Sigma'} k(y)$	100.0
2	$\beta(\Sigma) = \mathrm{Spec} \prod_{y\in\Sigma} k(y)$	$\beta(\Sigma) = \mathrm{Spec} \prod_{y\in\Sigma} k(y)$	69.4
3	$\{x, y_{1,i}, \ldots, y_{\kappa,i}, x'\} \in E(\Gamma(X, Y_j, p_1))$	$L = (\prod_{m\in\Sigma} k(m))/\mathcal{U}$	63.5
4	$\mathfrak{S}_{w(\lambda)}(x;y) = \prod_{(i,j)\in\lambda} (x_i - y_j).$	$\mu'_y(z) = \mu_y + O(r'_y)$	59.9
5	$\prod_{(i,j)\in\lambda} (x_i - y_j).$	$d_o k(\xi) = \mathrm{Ad}(k)\xi$	59.6
6	$\kappa_3(r'_y, \delta) \in \mathbb{R}$	$y = (p_e(y), p_{e'}(y))$	59.2
7	$X_{\mathcal{U}} = Y_{\mathcal{U}} \times_{\beta(Y)} \bigvee X_y = \bigvee_{y\in\Sigma} X_y/\mathcal{U}$	$x = \phi^{k'kp}(x) = \phi^p(x)$	58.4
8	$G(x,y) = H(x,y) + \sum_{i=1}^{k} \int_\Sigma dvol(z)\Gamma_i(x,z)H(z,y) + F_{k+1}(x,y).$	$x = \phi^{k'kp}(x) = \phi^p(x)$	58.4
9	$\mathfrak{S}_{w_0}(x;y) = \prod_{i+j\leq n} (x_i - y_j).$	$\overline{\mathrm{SP}}^n(\Sigma X) \simeq \Sigma\mathrm{Sym}^{*n}(X)$	57.7
10	$\mathrm{Sym}^{*n}(X) := \coprod_{k=1}^{n} \Delta_{k-1} \times_{\mathfrak{S}_k} X^k/_\sim$	$P\mathrm{Diag}(\bar{y})P^T = \mathrm{Diag}(\hat{y})$	57.5

Table 6.9: Results 11-20 for Query 6: $\prod_{y\in\Sigma} k(y) \to \prod_{y\in\Sigma'} k(y)$

Result	Lucene	Sub. Tree	Sub. Tree Rank
11	**Example 11.6.** As indicated in corollary 11.4, $\sigma H_*(\mathcal{B}_n(S^2);\mathbb{F}_2)$ consist of elements of filtration degree n in $\mathbb{F}_2[\iota_{(3,1)}, f_{(5,2)}, f_{(9,4)}, \dots, f_{(2^i+1,2^i)}, \dots]$. When $n=2$, there are only two generators: $f_{(5,2)}$ and $\iota^2_{(3,1)}$ (of bidegree $(6,2)$) so that $\tilde{H}_*(\mathcal{B}_2(S^2);\mathbb{F}_2) = \begin{cases}\mathbb{F}_2, *=4\\ \mathbb{F}_2, *=5\end{cases}$ and is zero otherwise. For p odd, $\tilde{H}_*(\mathcal{B}_2(S^2);\mathbb{F}_p)$ is trivial (i.e no non trivial class in filtration degree 2 in this case since $\iota^2_{(3,1)} = 0$ with odd primes) and this is consistent with $\mathcal{B}_2(S^2) \simeq \Sigma^3\mathbb{R}P^2$ as shown encore in (1).	$(s,y) \mapsto (cs,y)$	57.4
12	$\mathfrak{S}_{s_{n-i}\,w}(x;y)$	$(k',M') \neq (k,M)$	57.3
13	$\mu_y = K_1\kappa_3(r'_y,\delta)^{-1}\|B_{d'}(x)\cap\Omega_c\|^{-1}\nu(x)$	$\frac{1}{nm}s(y) = s(\frac{1}{n}u).$	56.9
14	**Example 10.3.** $\mathcal{B}_2(S^k) \simeq \Sigma^{k+1}\mathbb{R}P^k$ by (1) and hence $H_{2(k+1)-1}(\mathcal{B}_2(S^k)) = H_k(\mathbb{R}P^k)$ and this is indeed $\mathbb{Z}$ or 0 according to whether k is odd or even. Similarly and by lemma 8.2, the top class in dimension 5 of the space of chords of a closed Riemann surface is trivial as expected $H_5(\mathcal{B}_2(C_g)) \cong H_6((S^4)^{\vee(2g^2+g)} \vee (S^5)^{\vee 2g} \vee \Sigma^4\mathbb{R}P^2) = H_6(\Sigma^4\mathbb{R}P^2) = 0$	$\Sigma(\mathrm{Sym}^{*n}X) \cong \overline{\mathrm{SP}}^n(\Sigma X)$	56.7
15	$\pi : \mathrm{Sym}^{*n}X \longrightarrow T_n(X) = \coprod_{k=1}^{n} \Delta_{k-1} \times_{\mathfrak{S}_k} X^k/_\sim$	$\chi = \Sigma_{x\in\Omega}\gamma_x(\cdot)\chi(\cdot)$	56.6
16	$\Delta_y F(x,y) = \Gamma_k(x,y) - (\int_\Sigma dvol)^{-1}$	$\mathcal{A}(\Theta) > \mathcal{A}(\Theta')$	56.6
17	$\overline{\mathrm{SP}}^n\Sigma X = (\Sigma X)^{(n)} \simeq S^n \wedge_{\mathfrak{S}_n} X^{(n)} \simeq S^1 \wedge (S^{n-1} \wedge_{\mathfrak{S}_n} X^{(n)}) \simeq \Sigma\mathrm{Sym}^{*n}X$	$\Phi(m) := \sum_{k=m}^{\infty} \phi(k)$	56.5
18	**Corollary 6.2.** *Let Y be a pointed left $\mathfrak{S}_n$-space, and suppose $\mathfrak{S}_n$ acts on $S^n = (S^1)^{\wedge n}$ by permuting factors. Then $S^n \wedge_{\mathfrak{S}_n} Y \simeq \Sigma(S^{n-1} \wedge_{\mathfrak{S}_n} Y)$ and $\mathfrak{S}_n$ acts on $S^{n-1} \subset \mathbb{R}^n$ by permuting coordinates. In particular we recover the (weaker) equivalence* $\overline{SP}^n\Sigma X \simeq \Sigma Sym^{*n}(X)$	$K' \cong k'((t))$	56.5
19	$B_{r'_y(1+\delta)}(y) \setminus B_{r'_y}(y)$	$K'/k((x))$	56.2
20	$\omega_\infty(\mathfrak{a})$	$\nabla^*\nabla + 1 : C^\infty(\Sigma) \to C^\infty(\Sigma)$	56.0

Table 6.10: Top 20 Results for Query 7: $\mathbf{x}^{\mathbf{u}_1}$

Result	Lucene	Sub. Tree	Sub. Tree Rank
1	$\mathbf{x}^{\mathbf{u}_1}(1-\mathbf{x}^{\mathbf{v}_1-\mathbf{u}_1})$	$\mathbf{x}^{\mathbf{u}_1}$	100.0
2	$\mathbf{x}^{\mathbf{v}_1}(\mathbf{x}^{\mathbf{u}_1-\mathbf{v}_1}-1)$	$\mathbf{x}^{\mathbf{v}_1}$	98.1
3	$f_1=\mathbf{x}^{\mathbf{u}_1}-\mathbf{x}^{\mathbf{v}_1}, f_2=\mathbf{x}^{\mathbf{u}_2}-\mathbf{x}^{\mathbf{v}_2},\dots$	$\mathbf{H}_1$	67.5
4	$\mathbf{x}^{\mathbf{u}_1}$	$\mathbf{u}_1-\mathbf{u}_2$	65.2
5	$\mathbf{x}^{\mathbf{v}_1}$	$\mathbf{v}_1-\mathbf{u}_2$	65.2
6	$1-\mathbf{x}^{\mathbf{v}_1-\mathbf{u}_1}$	$\mathbf{u}_1-\mathbf{v}_1$	63.9
7	$\mathbf{x}^{\mathbf{u}_1-\mathbf{v}_1}-1$	$\mathbf{v}_1-\mathbf{v}_2$	63.9
8	$f_1=\mathbf{x}^{\mathbf{u}_1}-\mathbf{x}^{\mathbf{v}_1}$	$\mathbf{u}_1-\mathbf{v}_2$	63.9
9	$V=\begin{bmatrix}\mathbf{1}_g & 0\\ C & \mathbf{1}_g\end{bmatrix},\quad {}^tC=C,\quad C\in\mathbf{M}_g(\mathbb{Z}).$	$\sigma_{\mathbf{u}_2}$	60.7
10	$U=\begin{bmatrix}\mathbf{1}_g & B\\ 0 & \mathbf{1}_g\end{bmatrix},\quad {}^tB=B,\quad B\in\mathbf{M}_g(\mathbb{Z}).$	$\mathbf{C}^{\tilde{n}}$	59.3
11	$(\mathbf{x}^{\mathbf{u}_1},\mathbf{x}^{\mathbf{v}_1})$	$\pi_{\mathbf{f}_0}$	58.6
12	$S=R[(\mathbf{x}^{\mathbf{u}_1-\mathbf{v}_1})^{\pm1},(\mathbf{x}^{\mathbf{u}_2-\mathbf{v}_2})^{\pm1},\dots],$	$\sigma_{\mathbf{v}_2}$	58.6
13	$g_1^-,g_1^+\in C_0^\infty(\mathbf{R})$	$G_{\mathbf{u}_2}$	55.8
14	$\frac{d}{dt}q^-=2\mathrm{Re}\left\langle\gamma^u,\frac{d}{dt}\gamma^u\right\rangle_{\mathbf{C}^{n^u}}-2\mathrm{Re}\left\langle\gamma^s,\frac{d}{dt}\gamma^s\right\rangle_{<tex2html_{c}omment_{m}ark>45\mathbf{C}^{n^s}}\geq\delta^-t^{-1}q^+$	$\mathfrak{q}_{\Phi_1}$	54.3
15	$P=\begin{bmatrix}A & B\\ 0 & {}^tA^{-1}\end{bmatrix},\quad A.{}^tB=B.{}^tA,\quad A\in\mathrm{GL}_g(\mathbb{Z}),\quad B\in\mathbf{M}_g(\mathbb{Z}).$	$\lvert\mathcal{G}_1\rvert$	53.8
16	${}^tA.D-{}^tC.B=\mathbf{1}_g$	$G_{\mathbf{v}_2}$	53.7
17	$A.{}^tD-B.{}^tC=\mathbf{1}_g$	$\mathbf{f}_0^u$	53.7
18	$U=U'^2=\begin{bmatrix}\mathbf{1}_3 & 2B\\ 0 & \mathbf{1}_3\end{bmatrix},$	$(\mathbf{H}_1)$	51.3
19	$g_1,\tilde{g}_1,g_2,\tilde{g}_2\in C_0^\infty(\mathbf{R}^n)$	$(\mathbf{H}_1)$	51.3
20	$g_1\in C_0^\infty(\mathbf{R})$	$\mathbf{R}^2\times\mathbf{R}^2$	51.3

Table 6.11: Top 20 Results for Query 8: $D^n = CS^{n-1}$

Result	Lucene	Sub. Tree	Sub. Tree Rank
1	$D^n = CS^{n-1}$	$D^n = CS^{n-1}$	100.0
2	$D^n = CS^{n-1} = S^{n-1} \times I/\sim$	(D^n, S^{n-1})	73.8
3	$v^{-1} = w\,u^{-1}$	$S^{2n-1} = \partial D^{2n}$	70.8
4	$M^{-1} = \begin{pmatrix} {}^tD & -{}^tB \\ -{}^tC & {}^tA \end{pmatrix}$	$S^{n-1}/\mathfrak{S}_n$	70.2
5	$\sum_{j=0}^{n-1} r^j \geq r^n$	$C = n - 1$	68.2
6	$S^{2n-1} = \partial D^{2n}$	$y_{n+1} = ry_n$	67.4
7	$D = f^{-1}E$	S^{n-1}/K	66.5
8	$\Delta_q^n = \prod_{i=1}^{n}(E - q^{i-1}I),$	$Sp_1 Sp_{n-1}$	66.4
9	$c^{14}w_i c^{-14} = c^5 w_i c^{-5}$	$H^+ \subset S^{n-1}$	66.3
10	$\varphi^K = [I^* m^K]^{p'-1}\pi^{1-p'}.$	$T = S^{-1}US$	66.1
11	$10^{-2}\rho' = \rho = r^A$	$D = D^{n+1}$	65.6
12	$((D^k)^n, \partial(D^k)^n) \quad \text{where} \quad \partial(D^k)^n := \bigcup (D^k)^i \times S^{k-1} \times (D^k)^{i-1}$	$V\|S^{n-1}$	65.4
13	$c^5 wc^{-5} = w$	$\Psi = C\psi C^{-1}$	64.9
14	$c^7 wc^{-7} = w$	$D = f^{-1}E$	64.9
15	$= c^1 t_1 c^{-1}.$	$w \in S_{n-1} \subset S_n$	64.9
16	$= c^3 t_1 c^{-3},$	$S = TU^{-1}$	64.6
17	$= c^7 t_1 c^{-7}.$	$S^{n-1} * X$	64.2
18	$= c^2 t_1 c^{-2},$	$v_n = C^{-1}(w_n)$	64.1
19	$= c^4 t_1 c^{-4}.$	ΣS^{n-1}	63.9
20	$= c^3 t_1 c^{-3}.$	$I^n \times (D^{k-1})^n$	63.9

Table 6.12: Top 20 Results for Query 9: $\Omega_{\tau a}$

Result	Lucene	Sub. Tree	Sub. Tree Rank
1	$\Omega_{\tau a}$	$\Omega_{\tau a}$	100.0
2	$\Omega_{\tau a}$	Ω_a	84.4
3	$\tau_k : \Omega \mapsto \Omega$	Ω_a	84.4
4	Ω_a	$\Omega_{\tau a}$	73.2
5	Ω_a	Ω	70.8
6	Ω_a	a_Q	70.1
7	Ω_G	Ω_a	68.6
8	M_Ω	Ω_a	68.6
9	A_Ω	a	67.5
10	Ω_b	Ω	61.9
11	$\Omega_{R'}$	U_j	61.8
12	$\{\Psi_x\}_{x\in\Omega_I}$	Ω_b	60.9
13	$A_\tau = A_{\mathbf{\Omega}(\tau)}, \quad \mathbf{\Omega}(\tau) = [\tau\ \mathbf{1}_g], \quad \tau = \boldsymbol{\tau}(\Omega) = \Omega_2^{-1}\Omega_1,$	$a \in \mathcal{F}$	60.3
14	$\mathcal{F} := \{\, a \in \mathcal{C} \ :\ \Omega_a \text{ is taut} \,\}$	$\tilde{S}$	60.3
15	$\tau = \Omega_2^{-1}\Omega_1 \in \mathbb{H}_g$	$\mathcal{F}$	60.3
16	$\tau = \tau(\Omega) = \Omega_2^{-1}\Omega_1 = \begin{bmatrix} \tau_1 & 0 & 0 \\ 0 & \tau_2 & 0 \\ 0 & 0 & \tau_3 \end{bmatrix} \in \mathbb{H}_3.$	$\mathcal{C}$	60.3
17	$\Omega.M = [\Omega_1\ \Omega_2] \begin{bmatrix} A & B \\ C & D \end{bmatrix} = [\Omega_1 A + \Omega_2 C\ \Omega_1 B + \Omega_2 D].$	$\tilde{X}$	60.3
18	$\Omega_{\mathcal{P}}$	$\tilde{\alpha}$	60.3
19	$\Omega_{\mathcal{M}}$	$\mathbb{C}^*$	60.1
20	$\Omega \in \mathcal{R}_g$	$\mathbb{C}^2$	60.1

Table 6.13: Results 1-10 for Query 10: $\Omega = \dfrac{h^2}{m}(k_1^2 + k_2^2) + v_0 + v_{2k_2},$

Result	Lucene	Sub. Tree	Sub. Tree Rank
1	$\Omega = \frac{h^2}{m}(k_1^2 + k_2^2) + v_0 + v_{2k_2},$	$\Omega = \frac{h^2}{m}(k_1^2 + k_2^2) + v_0 + v_{2k_2},$	100.0
2	$\Omega = \frac{\hbar^2}{m}(k_1^2 + k_2^2) + v_{2k_2} - v_0$	$\Omega = \frac{\hbar^2}{m}(k_1^2 + k_2^2) + v_{2k_2} - v_0$	87.3
3	$\varphi_{k_2,l} = -\frac{b_l}{2} + \frac{1}{2}\sqrt{b_l^2 - 1}, \qquad l^2 > k_2^2,$ $\varphi_{k_2,l} = -\frac{b_l}{2} - \frac{1}{2}\sqrt{b_l^2 - 1}, \qquad l^2 < k_2^2,$ $\varphi_{k_2,k_2} = \frac{1}{2}, \qquad \varphi_{k_2,l} = \varphi_{k_2,-l},$ $b_l = \frac{h^2/m(l^2 - k_2^2) - (v_0 + v_{2k_2})}{v_{l-k_2} + v_{l+k_2}}, \qquad b_l = b_{-l}.$	$f(q_1^2 + q_2^2, p_1^2 + p_2^2) = 0,$	69.2
4	$B_l = \frac{\hbar^2}{2m}(l^2 - k_2^2) + (v_{l-k_2} + v_{l+k_2})\varphi_{k_2,l} - \frac{v_{2k_2}}{2}.$	$D_2^s(v) \neq 0, D_2^{s+1}(v) = 0.$	64.1
5	$B_l = \frac{\hbar^2}{2m}(l^2 - k_2^2) + i(v_{l+k_2} - v_{l-k_2})\varphi_{k_2,l} - \frac{v_{2k_2}}{2}.$	$Z(A_2) = Fe_1 + \cdots + Fe_{n+1},$	63.1
6	$V_l^+ = \frac{v_{l+k_2} + v_{2k_2}}{2}, \qquad V_l^- = \frac{v_{l-k_2} + v_0}{2}.$	$(t_1y_1 + \cdots + t_ky_k + s_0x_0, t)$	63.0
7	$= \psi(r)r^{k/2+m}\frac{\delta_k^m(f\\|\alpha_\infty^{-1})(x, \omega_\infty(x))}{\Omega_\infty^{k+2m}}$	$\widetilde{\lambda} = \lambda + \frac{\hbar^2}{m}k_1(k_2 + l)$	62.9
8	$\sum_{j=2}^{N} D_{1j}\varphi_1(e_1)(z_1 - 1)z_j + \sum_{k,j=2;k\neq j}^{N} D_{kj}\varphi_1(e_1)z_kz_j + \frac{1}{2!}\sum_{j=2}^{N} D_{jj}\varphi_1(e_1)z_j^2 + \cdots.$	$C = (q_1 = q_2 = 0, p_1^2 + p_2^2 = 1)$	62.5
9	$\lambda_{1,l} = -\frac{\hbar^2}{m}k_1l + \sqrt{\left(\frac{\hbar^2l^2}{2m} + v_l\right)^2 - v_l^2}.$	$\nu_k(J_{R_{\nu_k}/R}) = m_1 + m_2 + m_3 - 1 + k$	61.5
10	$\underline{0 leq a_1 leq a_2 leq a_3}$	$2d + 2/q = \frac{d}{4}(d^2 + d + 8)$	61.0

Table 6.14: Results 11-20 for Query 10: $\Omega = \frac{h^2}{m}(k_1^2 + k_2^2) + v_0 + v_{2k_2}$,

Result	Lucene	Sub. Tree	Sub. Tree Rank
11	$\Omega_k \partial_{x_j} \Omega_k = \frac{1}{2}\partial_{x_j}(\lvert\Omega\rvert^2) = 0$	$H_0(\eta, h_0) = A_1\eta + A_2 h_0,$	61.0
12	$E_{k_1,k_2} = N\left(\frac{\hbar^2(k_1^2+k_2^2)}{2m} + \frac{V_0}{2}\right).$	$\zeta = \zeta_{12},\ 2 \mid (m+1),$	60.9
13	$= \frac{1}{L_1 L_2^2}\sum_l \varphi_{k_2,l} e^{ik_1(x+y)} e^{il(x-y)},$	$g_1 \oplus g_2 := (g_1 + g_2) \wedge u,$	60.8
14	$0 = \sum_{x_0 \in \mathfrak{M}} \int_{B_R(0)} (u^2 - um)(x_0 + \frac{y}{\sqrt{\alpha}})\, dy + O(\int_{\mathbb{R}^N \setminus B_R(0)} e^{-c_1 \lvert y\rvert^2}\, dy) + O(\alpha^{\frac{N}{2}} e^{-\gamma\alpha}).$	$\alpha_1 + 2\alpha_2 + \cdots + d\,\alpha_d = \frac{d\,n - k}{2}$	59.9
15	$H_N(\mu_N S_t^N \mid \nu_\alpha)\ +\ N^2 \int_0^t \mathfrak{D}_N(f_s^N)\, ds\ \leq\ H_N(\mu_N \mid \nu_\alpha)\ ,$	$= (m+1)(2m+1),$	59.7
16	$= \psi(r) r^{k/2+m} \frac{d^m(f \Vert \alpha_\infty^{-1})(x, \omega_p(x))}{\Omega_p^{k+2m}}$	$\zeta = \zeta_{20}, \zeta_4,\ 2 \mid (m+1),$	59.2
17	$\lambda_{k_1,k_2,l} = -2ak_1(k_2+l)$ $\pm\left(\frac{1}{2}(a(l^2-k_2^2)+V_l-V_l^+)^2 + \frac{1}{2}(a(l_1^2-k_2^2)+V_l-V_l^+)^2 + V_l^{+2} - V_l^2\right.$ $\left.\pm\frac{1}{2}(a(l_1^2+l^2-2k_2^2)+2V_l-2V_l^+)\sqrt{a^2(l_1^2-l^2)^2+4V_l^{+2}}\right)^{1/2},$	$\zeta = \zeta_{24},\ 4 \mid (m+1),$	58.9
18	$\varphi_{k_2,l}^0 = \frac{1}{2V_0}\left(\frac{\hbar^2}{2m}(k_2^2 - l^2) + V_0 \pm \sqrt{\left(\frac{\hbar^2}{2m}(k_2^2-l^2)+V_0\right)^2 - V_0^2}\right),$	$Q_n(t) = 1 + t + t^2 + \ldots + t^{n-1},$	58.9
19	$\begin{aligned}\eta_i^{(2)} &= D_{f_i}\phi_i^*\phi_0^{-1}(\gamma_2 D_{f_i}^* h_0) \\ &= (\sigma_i(\nabla)+l_i)\gamma_2(\sigma_i^*(\nabla)+l_i^*)h_0 \\ &= \sigma_i(\nabla)\gamma_2\sigma_i^*(\nabla)h_0 + k_1\nabla(\gamma_2 h_0) + k_2 h_0 \\ &= k_0\nabla(\gamma_2 \otimes \nabla h_0) + k_1\nabla(\gamma_2 h_0) + k_2 h_0\end{aligned}$	$1 + 20\frac{m}{2} + 25\binom{m/2}{2} = \frac{(5m+4)(5m+2)}{8}$	58.9
20	$+\sum_{i=i_0+1}^{i_1-1}\sum_{j=1}^{m}\left(-a_i e_1 + a_i e_1 + a_i \frac{2^{-j}}{1-2^{-m}} e_i\right)$	$1 + 20\frac{m}{2} + 25\binom{m/2}{2} = \frac{(5m+4)(5m+2)}{8}$	58.9

Table 6.15: The statistical mode (the most frequent) distribution of the ratings given by participants in our experiment for the top 5 and top 20 search results of Zanibbi and Yuan's Lucene system and our substitution tree system. Entries represent: (number of "not similar" ratings, number of "somewhat similar" ratings, number of "very similar/identical" ratings).

	Lucene		Substitution Tree	
Query	Top 5	Top 20	Top 5	Top 20
1	(0,0,5)	(0,3,17)	(4,0,1)	(14,5,1)
2	(2,2,1)	(16,3,1)	(2,2,1)	(17,2,1)
3	(1,3,1)	(11,8,1)	(3,1,1)	(18,1,1)
4	(3,2,0)	(11,5,4)	(0,0,5)	(2,12,6)
5	(2,2,1)	(15,4,1)	(3,1,1)	(18,1,1)
6	(1,3,1)	(16,3,1)	(3,1,1)	(18,1,1)
7	(0,4,1)	(11,8,1)	(3,0,2)	(18,0,2)
8	(3,0,2)	(18,0,2)	(3,1,1)	(18,1,1)
9	(1,2,2)	(12,6,2)	(0,3,2)	(11,7,2)
10	(3,0,2)	(18,0,2)	(3,1,1)	(18,1,1)

Table 6.16: The distribution of ratings given by participants in our experiment for the top 20 search results of queries 1-5. Each entry contains the ratings given by Zanibbi and Yuan's Lucene system and our substitution tree system, in that order, separated by a comma. Entries represent: (number of "not similar" ratings, number of "somewhat similar" ratings, number of "very similar/identical" ratings).

Result	Query 1	Query 2	Query 3	Query 4	Query 5
1	(0,4,6),(0,0,10)	(0,0,10),(0,0,10)	(0,0,10),(0,0,10)	(0,7,3),(0,0,10)	(0,0,10),(0,0,10)
2	(0,4,6),(5,5,0)	(0,8,2),(0,7,3)	(0,6,4),(0,7,3)	(0,7,3),(0,0,10)	(1,5,4),(5,4,1)
3	(0,4,6),(5,5,0)	(7,3,0),(4,6,0)	(9,1,0),(9,1,0)	(5,4,1),(0,1,9)	(7,3,0),(5,4,1)
4	(0,4,6),(5,5,0)	(8,2,0),(8,2,0)	(3,7,0),(8,2,0)	(10,0,0),(0,2,8)	(1,8,1),(2,7,1)
5	(0,4,6),(5,5,0)	(4,6,0),(8,2,0)	(3,7,0),(9,1,0)	(10,0,0),(0,2,8)	(6,4,0),(5,4,1)
6	(0,4,6),(4,6,0)	(6,4,0),(8,2,0)	(3,7,0),(9,1,0)	(0,1,9),(0,6,4)	(6,4,0),(5,4,1)
7	(0,4,6),(5,5,0)	(6,4,0),(7,3,0)	(4,5,1),(9,1,0)	(0,0,10),(0,6,4)	(2,8,0),(7,3,0)
8	(0,4,6),(5,5,0)	(6,4,0),(8,2,0)	(1,8,1),(8,2,0)	(0,0,10),(2,7,1)	(6,4,0),(5,4,1)
9	(0,4,6),(5,5,0)	(6,4,0),(7,3,0)	(2,7,1),(8,2,0)	(0,1,9),(7,3,0)	(6,4,0),(7,3,0)
10	(0,4,6),(5,5,0)	(2,8,0),(8,2,0)	(2,7,1),(8,2,0)	(3,5,2),(1,8,1)	(8,2,0),(9,1,0)
11	(0,4,6),(4,6,0)	(9,1,0),(8,2,0)	(9,1,0),(8,2,0)	(10,0,0),(1,8,1)	(7,3,0),(10,0,0)
12	(0,4,6),(5,5,0)	(9,1,0),(8,2,0)	(6,4,0),(9,1,0)	(3,4,3),(4,5,1)	(7,3,0),(10,0,0)
13	(0,3,7),(4,6,0)	(9,1,0),(8,2,0)	(10,0,0),(9,1,0)	(10,0,0),(0,7,3)	(9,1,0),(7,3,0)
14	(0,4,6),(5,5,0)	(10,0,0),(9,1,0)	(7,3,0),(7,3,0)	(3,4,3),(0,6,4)	(10,0,0),(9,1,0)
15	(0,4,6),(5,5,0)	(10,0,0),(8,2,0)	(7,3,0),(8,2,0)	(6,2,2),(1,6,3)	(10,0,0),(9,1,0)
16	(0,5,5),(4,6,0)	(10,0,0),(8,2,0)	(10,0,0),(8,2,0)	(10,0,0),(1,6,3)	(3,5,2),(10,0,0)
17	(0,5,5),(5,5,0)	(10,0,0),(10,0,0)	(10,0,0),(9,1,0)	(10,0,0),(1,6,3)	(10,0,0),(10,0,0)
18	(0,4,6),(5,5,0)	(8,2,0),(9,1,0)	(9,1,0),(9,1,0)	(9,1,0),(1,5,4)	(9,1,0),(8,2,0)
19	(0,4,6),(6,4,0)	(10,0,0),(10,0,0)	(10,0,0),(8,2,0)	(5,4,1),(0,1,9)	(10,0,0),(9,1,0)
20	(0,4,6),(4,6,0)	(9,1,0),(8,2,0)	(10,0,0),(7,3,0)	(9,0,0),(7,3,0)	(9,1,0),(9,1,0)

Table 6.17: The distribution of ratings given by participants in our experiment for the top 20 search results of queries 6-10. Each entry contains the ratings given by Zanibbi and Yuan's Lucene system and our substitution tree system, in that order, separated by a comma. Entries represent: (number of "not similar" ratings, number of "somewhat similar" ratings, number of "very similar/identical" ratings).

Result	Query 6	Query 7	Query 8	Query 9	Query 10
1	(0,0,10),(0,0,10)	(2,6,2),(0,0,10)	(0,0,10),(0,0,10)	(0,0,10),(0,0,10)	(0,0,10),(0,0,10)
2	(3,7,0),(1,8,1)	(2,7,1),(0,4,6)	(0,3,7),(3,7,0)	(0,0,10),(0,6,4)	(0,4,6),(0,5,5)
3	(9,1,0),(7,3,0)	(3,5,2),(9,1,0)	(9,1,0),(6,4,0)	(7,3,0),(0,6,4)	(7,3,0),(8,2,0)
4	(4,6,0),(9,1,0)	(0,1,9),(9,1,0)	(10,0,0),(8,2,0)	(1,5,4),(0,0,10)	(5,5,0),(10,0,0)
5	(4,6,0),(10,0,0)	(0,6,4),(10,0,0)	(10,0,0),(8,2,0)	(1,5,4),(0,10,0)	(5,5,0),(10,0,0)
6	(10,0,0),(10,0,0)	(3,7,0),(9,1,0)	(6,4,0),(10,0,0)	(1,5,4),(10,0,0)	(10,0,0),(10,0,0)
7	(10,0,0),(10,0,0)	(1,9,0),(10,0,0)	(9,1,0),(9,1,0)	(3,7,0),(0,7,3)	(10,0,0),(9,1,0)
8	(10,0,0),(10,0,0)	(1,7,2),(9,1,0)	(10,0,0),(9,1,0)	(9,1,0),(0,7,3)	(9,1,0),(9,1,0)
9	(5,5,0),(10,0,0)	(9,1,0),(10,0,0)	(10,0,0),(9,1,0)	(9,1,0),(10,0,0)	(7,3,0),(8,2,0)
10	(6,4,0),(10,0,0)	(9,1,0),(10,0,0)	(10,0,0),(9,1,0)	(2,7,1),(3,7,0)	(10,0,0),(9,1,0)
11	(10,0,0),(9,1,0)	(1,7,2),(10,0,0)	(10,0,0),(6,4,0)	(3,7,0),(10,0,0)	(10,0,0),(10,0,0)
12	(10,0,0),(10,0,0)	(6,4,0),(10,0,0)	(9,1,0),(9,1,0)	(10,0,0),(2,8,0)	(8,2,0),(10,0,0)
13	(10,0,0),(9,1,0)	(10,0,0),(10,0,0)	(9,1,0),(9,1,0)	(10,0,0),(10,0,0)	(10,0,0),(10,0,0)
14	(10,0,0),(10,0,0)	(10,0,0),(10,0,0)	(9,1,0),(8,2,0)	(8,2,0),(10,0,0)	(10,0,0),(10,0,0)
15	(7,3,0),(10,0,0)	(10,0,0),(10,0,0)	(10,0,0),(9,1,0)	(9,1,0),(10,0,0)	(10,0,0),(10,0,0)
16	(10,0,0),(10,0,0)	(10,0,0),(9,1,0)	(10,0,0),(9,1,0)	(8,2,0),(10,0,0)	(10,0,0),(10,0,0)
17	(10,0,0),(10,0,0)	(10,0,0),(9,1,0)	(10,0,0),(10,0,0)	(8,2,0),(10,0,0)	(10,0,0),(10,0,0)
18	(10,0,0),(10,0,0)	(10,0,0),(10,0,0)	(10,0,0),(8,2,0)	(5,5,0),(10,0,0)	(8,2,0),(10,0,0)
19	(9,1,0),(10,0,0)	(10,0,0),(10,0,0)	(10,0,0),(10,0,0)	(5,5,0),(10,0,0)	(10,0,0),(10,0,0)
20	(10,0,0),(10,0,0)	(10,0,0),(10,0,0)	(10,0,0),(10,0,0)	(10,0,0),(10,0,0)	(10,0,0),(10,0,0)

Bibliography

[1] Moody Altamimi and Abdou Youssef. An extensive math query language. In *ISCA International Conference on Software Engineering and Data Engineering (SEDE-2007)*, pages 57–63, Las Vegas, NV, 2007.

[2] Andrea Asperti, Ferruccio Guidi, Claudio Coen, Enrico Tassi, and Stefano Zacchiroli. A content based mathematical search engine: Whelp. In *Proc. Types for Proofs and Programs 2004*, volume 3839 of *Lecture Notes in Computer Science*, pages 17–32, 2006.

[3] D. Bolstein, J. Cordy, and R. Zanibbi. Applying compiler techniques to diagram recognition. In *Proc. International Conf. Pattern Recognition*, volume 3, pages 123–126, 2002.

[4] Pia Borlund. User-centered evaluation of information retrieval systems. In *Information Retrieval: Searching in the 21st Century*, pages 21–37. Wiley, West Sussex, UK, 2009.

[5] J.R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[6] David Doermann. The indexing and retrieval of document images: A survey. *Journal on Computer Vision and Image Understanding*, 70:287–298, June 1998.

[7] Peter Graf. Substitution tree indexing. In *RTA '95: Proceedings of the 6th International Conference on Rewriting Techniques and Applications*, pages 117–131, London, UK, 1995. Springer-Verlag.

[8] Peter Graf. *Term Indexing.* Lecture Notes in Artificial Intelligence 1053. Springer-Verlag, Berlin, Germany, 1995.

[9] H. Hashimoto, Y. Hijikata, and S. Nishida. Incorporating breadth first search for indexing MathML objects. In *IEEE International Conf. Systems, Man and Cybernetics*, pages 3519–3523, Oct. 2008.

[10] Djoerd Hiemstra. Information retrieval models. In *Information Retrieval: Searching in the 21st Century*, pages 1–19. Wiley, West Sussex, UK, 2009.

[11] Shahab Kamali and Frank Tompa. Improving mathematics retrieval. In *Proc. DML 2009: Towards a Digital Mathematics Library*, pages 37–48, Grand Bend, Canada, July 2009.

[12] Donald Knuth. *The TEXbook.* Addison Wesley, Reading, Massachusetts, 1986.

[13] Andrea Kohlhase and Michael Kohlhase. Reexamining the MKM value proposition: From math web search to math web research. In *Proc. Symposium Towards Mechanized Mathematical Assistants*, pages 313–326, Berlin, Heidelberg, 2007. Springer-Verlag.

[14] Michael Kohlhase and Ioan Sucan. A search engine for mathematical formulae. In Jacques Calmet Testuo Ida and Dongming Wang, editors, *Proc. Artificial Intelligence and Symbolic Computation*, number 4120 in LNAI, pages 241–253. Springer Verlag, 2006.

[15] Hsi-Jian Lee and Jiumn-Shine Wang. Design of a mathematical expression understanding system. *Pattern Recognition Letters*, 18:289–298, 1997.

[16] Paul Libbrecht and Erica Melis. Methods for access and retrieval of mathematical content in activemath. In *Proc. International Congress on Mathematical Software*, volume 4151 of *LNCS*, pages 331–342. Springer-Verlag, 2006.

[17] William A. Martin. Computer input/output of mathematical expressions. In *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, pages 78–89, March 1971.

[18] Bruce R. Miller and Abdou Youssef. Technical aspects of the digital library of mathematical functions. In *Annals of Mathematics and Artificial Intelligence*, pages 121–136. May 2003.

[19] Robert Miner and Rajesh Munavalli. *An Approach to Mathematical Search Through Query Formulation and Data Normalization*, volume 4573 of *Lecture Notes in Artificial Intelligence*. 2007.

[20] Rajesh Munavalli and Robert Miner. MathFind: A math-aware search engine. In *SIGIR '06: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, page 735, New York, NY, 2006.

[21] Brigitte Pientka. Higher-order substitution tree indexing. In Catuscia Palamidessi, editor, *Logic Programming*, volume 2916 of *Lecture Notes in Computer Science*, pages 377–391. Springer Berlin / Heidelberg, 2003.

[22] Gerald Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1983.

[23] Clare M. So and Stephen M. Watt. Determining empirical characteristics of mathematical expression use. In *Proc. Mathematical Knowledge Management*, volume 3863

of *LNCS*, pages 361–375. Springer, 2005.

[24] Ellen M. Voorhee. Overview of TREC 2003. In *Proc. TREC*, 2003.

[25] Abdou Youssef. Roles of math search in mathematics. In Jonathan Borwein and William Farmer, editors, *Mathematical Knowledge Management*, volume 4108 of *Lecture Notes in Computer Science*, pages 2–16. Springer Berlin / Heidelberg, 2006.

[26] Abdou Youssef. Methods of relevance ranking and hit-content generation in math search. In *Proc. Towards Mechanized Mathematical Assistants*, pages 393–406, Berlin, 2007. Springer-Verlag.

[27] Richard Zanibbi and Dorothea Blostein. Recognition and retrieval of mathematics. *Int'l. Journal of Document Analysis and Recognition*, 2011 to appear.

[28] Richard Zanibbi, Amit Pillay, Harold Mouchere, Christian Viard-Gaudin, and Dorothea Blostein. Stroke-based performance metrics for handwritten mathematical expressions. In *Int'l Conf. Document Analysis and Recognition*, pages 334–338, Beijing, 2011 to appear.

[29] Richard Zanibbi and Bo Yuan. Keyword and image-based retrieval for mathematical expressions. In *Proc. Document Recognition and Retrieval XVIII*, volume 7874 of *Proc. SPIE*, pages OI1–OI9, San Francisco, CA, Jan. 2011.

[30] Jin Zhao, Min-Yen Kan, and Yin Leng Theng. Math information retrieval: User requirements and prototype implementation. In *Proc. ACM/IEEE Joint Conf. Digital libraries*, pages 187–196, New York, NY, USA, 2008.