Source Code Retrieval using Conceptual Graphs

Gilad Mishne

Master of Logic Thesis

Institute for Logic, Language and Computation (ILLC)
University of Amsterdam
Plantage Muidergracht 24, 1018 TV Amsterdam
The Netherlands.

E-mail: gilad@science.uva.nl

Supervisors:
Dr. Maarten de Rijke
Dr. Maarten Marx

December 2003

Contents

Acknowledgments **5**

Abstract **6**

Preface **7**

1 Introduction **8**
1.1 Why Code Retrieval? 8
1.2 Structured Document Information Retrieval 9
1.3 Conceptual Graphs 11
1.4 Using Conceptual Graphs for Code Retrieval 13
1.5 Related Work 14
1.5.1 Code Retrieval 14
1.5.2 Retrieval using Conceptual Graphs 15

2 Representing Source Code as Conceptual Graphs **17**
2.1 Source Code Representations 17
2.2 Converting Code to Conceptual Graphs 18
2.2.1 A Taxonomy for Source Code 18
2.2.2 Graph Construction Process 21
2.2.3 Examples 22

3 Code Retrieval using Conceptual Graph Representations **26**
3.1 The Retrieval Model 26
3.1.1 The Retrieval Process 27
3.2 A Similarity Measure for Source Code Conceptual Graphs 28
3.2.1 Retrieval Similarity Measures 28
3.2.2 Comparing Conceptual Graphs 28
3.2.3 Complexity 36
3.2.4 Additions and Modifications 38
3.2.5 Summary 38

4 Evaluation **39**
4.1 Experimental Setting 39
4.1.1 Document Collection 39
4.1.2 Experiments 42
4.1.3 Measures 44
4.1.4 Assessment 45
4.1.5 Retrieval Parameters 45
4.1.6 Baseline Models 46
4.1.7 Implementation 49
4.1.8 Additional Computational Aspects and Filtering 50
4.2 Results and Discussion 52
4.2.1 Testing for Optimized Parameters 52
4.2.2 Comparison with Baselines 54
4.2.3 Significance Tests 56
4.2.4 Combining Methods 57

5 Conclusions **59**
5.1 Open Issues and Future Work 60

Bibliography **66**

Acknowledgments

The past year, along with the years to come, would be very different without the influence I had from many people in the last months. I would like to thank some of them here.

First and foremost, my supervisors: Dr. Maarten Marx opened the door for me into the LIT group, and followed me closely ever since with helpful comments and discussions (and an equally important electric guitar, courtesy of Dr. Gonzo). Dr. Maarten de Rijke supported me in numerous ways, academic and other, well above any expectation I had from an academic supervisor; provided me with the opportunity to participate as an equal in the research done by the group, supplied countless comments and suggestions regarding the thesis and other work, and solved complex problems that seemed to have no solutions.

My parents, Nava and David, who accepted my choices, some of them they did not necessarily like.

Lotta, for never-ending support, and for the willingness to make a big sacrifice, should that be needed.

All (past and present) members of the Language and Inference Technology group in the university of Amsterdam – Juan Heguiabehere, Valentin Jijkoun, Jaap Kamps, Gabriel Infante-Lopez, Christof Monz, Karin Müller, Detlef Prescher, Börkur Sigurbjörnsson, Khalil Sima'an, Willem van Hage and Stefan Schlobach: for many comments, discussions and suggestions, for support during rougher times, and for the chance to work with the people that make LIT one of the leading groups in its field. I have learned a lot from all of you.

Anny Crajé from the university personnel department, who (together with Maarten de Rijke) worked hard to make the impossible possible.

My friends – in Israel, in Holland, and in other places.

Bansi and Simba for leaving us the piano.

Finally, NWO – the Netherlands Organization for Scientific Research – which supported my work.

Abstract

The sharp increase in the amount of easily accessible information in the last decade resulted in a growing amount of research regarding information extraction and retrieval. Inside the general information retrieval framework, specialized methods emerged for specific domains, trying to exploit features of the information in these domains to improve its accessibility.

Source code – documents written in a computer programming language – is one of these domains. Source code is a form of structured data, data in which information is stored both in the structure and in the content; it is, however, different from other structured document domains both in the nature of the structure and the nature of the content. Retrieval of information from source code is crucial for large-scale software development and maintenance, and is recognized as a problem both by software developers and information retrieval researchers; it is a vast research area with multiple interests and various sub-tasks. This thesis focuses on one aspect of these: the usage of the structure of the code to improve the retrieval.

Our approach for improving the retrieval from source code uses conceptual modeling of the code. We employ conceptual graphs – a knowledge representation formalism – to design a retrieval model for code that uses both its structure and its content. The model contains a formal definition of the problem, a method for representing the code as conceptual graphs, and procedures for ranking their similarity to a query.

We evaluate our model and show that for the code retrieval task it performs better than standard information retrieval approaches. The associated complexity implications are analyzed and discussed, searching for a balance of computational costs and retrieval performance. Our main conclusion is that using the structure for retrieval of code improves retrieval results substantially, and further work in this direction can improve the results even more.

Preface

The complex task of retrieving, classifying and extracting information from source code files — *Code Retrieval* — is essential in the development cycle of large software systems [85]. A number of reasons make source code retrieval difficult: most notably, the interleaved nature of the structure and the content inside the documents, and the differences between programming language syntax and semantics and natural language syntax and semantics.

If we consider source code to be a specific type of *structured text* — a document which contains a layer of structure in addition to the standard information layer — it is known that using the structure helps to improve retrieval performance [17, 46, 40]. But, while using structure for retrieval may be beneficial, it comes with a cost: extraction and manipulation of structure is complicated, and results in a higher complexity for the retrieval process [15].

In this work, we explore this idea: we present a retrieval model for code that uses its structure, evaluate it, and discuss its strengths, weaknesses, and complexity implications. We search for a balanced amount of usage of structural knowledge, looking for a compromise between complexity and accuracy. We pose the following questions: *How can we use code structure to improve retrieval? What are the associated costs of such improvements? Is this usage feasible?*

To represent the structural information in source code, we use *conceptual graphs*, a knowledge representation language combining ideas both from first order logic and from graph theory [80]. We develop a mechanism for converting source code documents into conceptual graphs, and procedures for comparing these graphs in a way that combines structure and information.

The remainder of the thesis is organized as follows. In chapter 1 we provide background regarding source code retrieval and conceptual graphs, and discuss previous work done in these areas. In chapter 2 we present our method for representing code as conceptual graphs and give examples for it. Chapter 3 deals with the retrieval model that we construct using the graphs, and presents a similarity measure for the source code graphs constructed. In chapter 4 we describe evaluation experiments and discuss their results; we conclude in chapter 5, which also contains a discussion of open issues and possible future work.

Chapter 1

Introduction

1.1 Why Code Retrieval?

A method for locating "relevant" information in a large collection of code given a specific information need is beneficial in the following scenarios:

Software Systems Analysis. A common scenario for software developers is the need to modify, adapt or use large-scale software components which they did not design or implement. Many times it is the case that these so-called legacy systems are badly documented, loosely structured and hard to understand. In some instances, code may even seem cryptic to its own author, if it is poorly written or if a long time has passed since its implementation. Studies show that application maintenance consumes the majority of the software budget for large-scale systems [7]. Under these circumstances, code retrieval methods may help the developers have a quicker, and deeper, understanding of the software system; programmers involved in maintenance often claim that they are missing required tools for this task [54, 18].
Related tasks to system analysis include software reverse-engineering, recovery of design and documentation from source code and software restruction [53].

Software Components Library Lookup. For many developers, the fastest way to learn how to accomplish a programming task is to look at an example of a similar implementation, or at a library of existing, tested, related components [48]. Lookup of such examples is often done with "grep-like" searches in a local code corpus, or using standard retrieval engines. Developers may benefit from a tool that allows deeper queries of a code corpus, local or external, to locate examples of implementations. A specific problem of example-lookup arises in the special case of working with a number of releases of a specific software suite; many times,

in the shift between versions (i.e. 2.1 → 3.0), some components are re-written or changed thoroughly. In these cases, the developer may want to look up the way in which a task was performed in one of the previous versions.

Code Duplication. According to previous studies [5], duplicated code accounts for up to 20% of the total amount of code in large software systems (millions of lines-of-code). Code duplication originates in two common software developer tendencies: using the "copy-and-paste" method for insertion of code which is known to function in one place in the program into other places, and incomplete knowledge about the system's design and components that results in "reinventing the wheel" and re-implementation of already existing pieces of code. Code duplication is a known software engineering problem for the following reasons:

- While it seems a fast, efficient solution at first, in effect it causes more work because changes to a piece of code must be made in multiple places, each requiring variations to match other modifications made to the original code that was duplicated. Moreover, it may lead to erroneous code: a bug which is fixed in one place must be found and fixed in all the duplicated versions – a task often neglected at the bug-fixing stage.
- Duplication violates standard coding reuse methodologies – resulting in code that is less clear and less easy to analyze.
- Finally, duplication causes unnecessary resource consumption (increase in source code length, program size and compilation times).

Plagiarism Detection. Plagiarized work is a piece of writing that has been copied from someone else and is presented as being original work. As any digital document, source code files are easy to copy and modify. An unfortunate result of this fact, combined with the stressed nature of a student's life, leads to an alarmingly high number of programming assignments plagiarisms, mainly in low-level programming courses [77]. Code plagiarism occurs also in commercial, larger-scale scenarios, with the most well-known example being the recent legal struggle between SCO and IBM regarding Unix source code [74, 75].

While the first two tasks focus on high-level abstraction of the code, the last two seem to benefit from less abstract, more detailed knowledge about it; we focus our work on retrieval aimed at these goals.

1.2 Structured Document Information Retrieval

Source code is a special case of the more general notion of a *structured* or *semi-structured* document, a document which contains additional information embedded

```
#include <dcopclient.h>

void KonqViewManager::saveViewProfile( KConfig & cfg, bool saveWindowSize )
{
  //kdDebug(1202) << "KonqViewManager::saveViewProfile" << endl;
  if( m_pMainWindow->childFrame() != 0L ) {
    QString prefix = QString::fromLatin1(
        m_pMainWindow->childFrame()->frameType() )
                    + QString::number(0);
    cfg.writeEntry( "RootItem", prefix );
    prefix.append( '_' );
    m_pMainWindow->saveConfig( &cfg, prefix, saveURLs, m_pDocContainer, 0, 1);
  }

  // Save menu/toolbar settings in profile. Relys on konq_mainwindow calling
  // setAutoSaveSetting( "KongMainWindow", false ). The false is important,
  // we do not want this call save size settings in the profile, because we
  // do it ourselves. Save in a separate group than the rest of the profile.
  QString savedGroup = cfg.group();
  m_pMainWindow->saveMainWindowSettings( &cfg, "Main Window Settings" );
  cfg.setGroup( savedGroup );

  cfg.sync();
}
```

Figure 1.1: Typical source code (simplified)

in its structure. More specifically, code is *structured text*; other types of structured text which are common in the retrieval setting are HTML and XML, and extensive research has been done regarding effective retrieval of them. One of the clearest conclusions of this research is that using structural knowledge may improve results significantly [90, 51].

In [4], text structure is defined as "information present in a text apart from its content, which relates its different portions in a semantically meaningful way". The main difference between retrieval of source code and retrieval of other structured data is the embedded nature of the structure in the code. Contrary to markup languages such as XML which follow the above notion of keeping the structure apart from the content, structure and content are mixed in source code. Another difference is the increased importance of the structure: while most structured documents have a large amount of content and some structural information that classifies the different types of content, in a source code file there is potentially little information other than the structure itself. Finally, an examination of typical source code text (see figure 1.1) reveals that usually a single document combines two different languages: natural language text (in comments, `print` statements etc.), and "programming language text". The latter includes both the tokens of the programming language itself, but also variable names, function names and so on; these tend to have different lexical and morphological rules from natural language.

1.3 Conceptual Graphs

The Conceptual Graph (CG) formalism is a knowledge representation language proposed by John Sowa in 1984 [80], based on the work of Charles Sanders Peirce. In Sowa's words, the purpose of conceptual graphs is "to express meaning in a form that is logically precise, humanly readable, and computationally tractable".

A conceptual graph is a bipartite, directed, finite graph; each node in the graph is either a *concept node* or *relation node*. Concept nodes represent entities, attributes, states, and events, and relation nodes show how the concepts are interconnected. A node (concept or relation) has two associated values: a *type* and a *referent* or *marker*; a referent can be either the single *generic* referent, or an *individual* referent. A conceptual graph is always related to a *support* or *canon*, a semantic-web-like knowledge base providing background on the domain within which the graph is presented. This support contains:

- A set of *concept types*, structured in a single connected hierarchical lattice; the relation between two neighbor concepts types A and B in the lattice is "B is-a-kind-of A", and multiple inheritance is allowed.
- A set of *relation types*.
- An indication, for each relation type, what kind of concept types it is permitted to connect. One way to provide such an indication is a set of *"star graphs"* for every relation type, graphs that connect this relation type to a totally ordered set of all permitted concept types.
- A set of *referent sets* for each concept type; each referent set must include at least the generic referent, also marked as "*".

More formally, the support is a 4-tuple $S = \langle T_c, T_r, B, R \rangle$ where:

1. T_c, the set of *concept types*, is a finite lattice with $\leqslant$ as order, 1 as supremum (the universal type), 0 as infinum (the absurd type), $\wedge$ and $\vee$ denoting the lower and upper bounds.
2. T_r, the set of relation types, is a finite set; T_c and T_r are disjoint.
3. B is a set of *"star graphs"*, $B_{r_i}, r_i \in T_r$, in bijection with T_r; every B_{r_i} is built as follows: exactly one vertex of B_{r_i} is labeled by the element r_i of T_r, this vertex has a non empty and totally ordered set of neighbors, these neighbors being pairwise non adjacent, and each of them is labeled by an element of T_c.
4. R is a set of countable sets of individual referents, each set R_t associated with a concept type $t \in T_c$. In addition, there exist a referent called generic

("*") and an absurd marker 0; each set $R_t \cup *, 0$ is provided with a lattice structure by the order, denoted by $<_t$, such that any two elements of R_t are incomparable, and $\forall r_t \in R_t : 0 < r_t < *$.

A conceptual graph related to this support is a 5-tuple $g = \langle N_C, N_R, E, ord, label \rangle$ bipartite finite graph such that:

1. N_C and N_R are the concept and relation nodes, respectively; $N_C \cap N_R = \emptyset$, and $N_C \neq \emptyset$.
2. E is the set of the graph edges; edges adjacent to a relation node r are totally ordered by ord.
3. $label$ is a mapping from every concept node of the graph to a label; a label for concept node c is a pair such that $label(c) = (t, r)$, $t \in T_c, r \in R$.

Given a canon, an infinite set of well formed graphs can be constructed.

Example of a simple canon is given in figure 1.2; concept types are presented as text and the inheritance hierarchy by lines. Simple conceptual graphs based on it are shown in figure 1.3, using the standard graphical notation for CGs which uses rectangles for concept nodes and ovals for relation nodes.

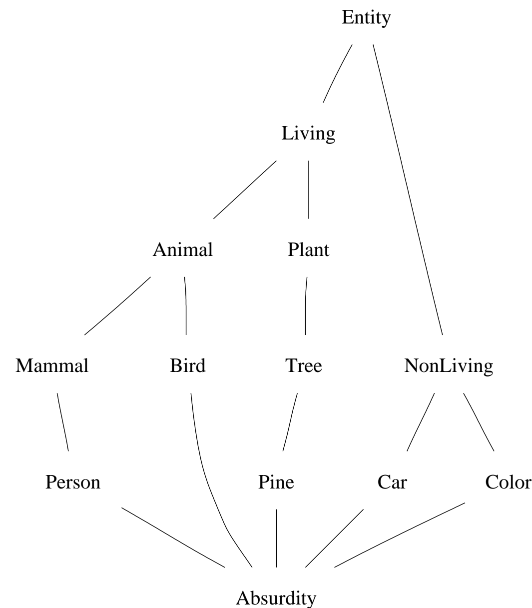

Figure 1.2: Partial canon

After introducing these basic notions, Sowa describes a wide range of operations, techniques and extensions, such as various projections and morphisms,

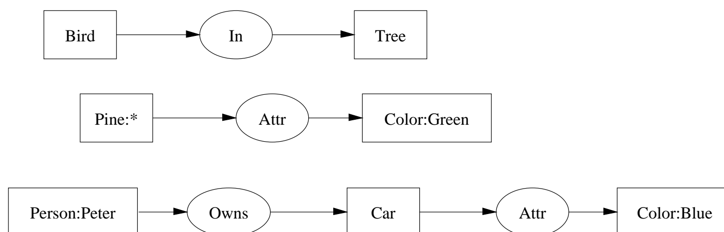

Figure 1.3: Examples of conceptual graphs

specialization and generalization, partitioning, and so on. These notions will not be used later and therefore will not be discussed here, other than a summary of conceptual graph similarities presented in section 3.2.2. A good summary of conceptual graphs and their operations can be found in [10].

Sowa showed that the conceptual graph formalism is a subset of first order logic; however, much of the research work on conceptual graphs is focused on graph theory [81][27][19]. This demonstrates the attractiveness of CGs: they allow exploration of new directions in first order logic research by applying knowledge and tools from a completely different domain (graph theory). This comes with a cost, both computational and relating to expressive power, and in this sense CGs are similar to modal logics.

1.4 Using Conceptual Graphs for Code Retrieval

We discussed the need for retrieval from source code, the structured nature of code and the usefulness of retrieval using structural knowledge. We have also presented conceptual graphs as a modeling language that can capture both the structure and the content of code. Now we can propose a retrieval method based on conceptual modeling of the source code: our aim is to explore the usefulness of this kind of abstraction for code retrieval, and in general to investigate ways of exploiting the structural properties of source code for retrieval.

Applying the conceptual graph formalism for retrieval can be divided into two main subtasks. First, we need to describe a mechanism for representation of the code as CGs. This will be done in chapter 2. Then, we must define a retrieval model for the graphs, and most importantly, a similarity measure between them; this will be done in chapter 3. We can then test the model to test our hypothesis regarding conceptual modeling of code as a retrieval helper and draw conclusions; such experiments and our conclusions are presented in chapters 4 and 5 respectively.

1.5 Related Work

1.5.1 Code Retrieval

As discussed earlier, the problem of duplicated and similar source code is recognized as an important one both in the software development cycle and from the plagiarism detection aspect. A number of approaches exist to address this issue, and can be roughly divided into two categories: methods for retrieval from source code, and similarity analysis methods for code.

Source Code Retrieval

Early forms of retrieval from code were based on a classification scheme for cataloging code components with a set of keywords, such as the method specified in [64]. Such methods yield good results, but the manual effort required for them is very high, mostly for the classification (which is often done manually) but also for the retrieval (which requires knowledge about the legal, relevant keywords).

Many tools for software development provide *context-tagging*, a process where information about the location of functions, variables etc. is kept in an indexed structure, allowing fast access. While context-tagging is an important tool for developers, it is appropriate for *browsing* the code – when the user knows what she is looking for – rather than *searching* it.

Tools such as GURU [41] and ROSA [30] make use of natural language processing and information retrieval techniques to index and retrieve software and software-related documents (design, specifications). These approaches focus on the natural language text that exists in the code (comments, documentation, meaningful variable names etc.), and is therefore suited for well-documented projects. Since almost no structural knowledge is taken into account, they are of limited use for the common case of sparse documentation in large code bases.

Formal approaches such as [60, 36] require the information need to be specified in a specialized query language (in which requests such as "find all functions that contain a variable `arr`" can be stated formally). While these are very powerful methods for maintainers of large software projects, they lack the common retrieval "fuzziness" where documents are *relevant* for a query, but not necessarily *match* it. Additionally, these methods require some training prior to usage, because their query language is not standard. A similar method, making use of standard (XML) markup of the code, was proposed in [12]; it is more standardized but shares the same advantages and disadvantages of other formal methods.

Conceptual modeling and retrieval of code has been researched and tested (for example, in system such as LaSSIE [20]); however, the modeling tends to focus on higher-level concepts rather than the micro-concepts expressed through the code, resulting in a tool fit for architectural queries such as "what functions enable feature X", and not queries like "find functions that are related to function

Y". Additionally, the knowledge base for every software project is hand-crafted, making the solution not general.

Code Similarity Analyzers

Currently implemented similarity analyzers can be grouped as follows:

Pattern-based Analyzers This approach checks for shallow similarity between lines of codes, using pattern matching techniques and tiling algorithms; examples of tools using it are PMD [61] and Simian [78]. This approach is very effective (and fast) mostly at detecting simply duplicated ("copy-pasted") chunks of code scattered around large-scale enterprise projects, or very similar pieces of code. However, very simple structural code changes render it almost completely useless.

Code Signature Analyzers This group of analyzers calculates certain metrics of the code (ranging from the number of blank lines or unique tokens to properties of the resulting program's function-call tree); such approaches are presented in [37] and [29]. Each program is associated with a "code signature" – a value or set of values summarizing its features; programs with similar signatures are considered to be similar. Since this approach relies on statistical properties of the code, it is effective for plagiarism detection on source code files which are relatively large, such as student assignments in programming courses, rather then detection of short repeating sections of code. One analyzer of this type which stands out of the rest in terms of performance is MoSS [73], which uses local fingerprinting mechanisms and is actively used for student assignments plagiarism detection; however, little written documentation is available regarding the internals of this approach.

Structural Analyzers These analyzers are considered the most advanced ones. Analyzers of this type – such as YAP [91] and JPlag [63] – compare structural properties of the programs. The comparison is performed by representing the code structures as strings, and then measuring the string distance between them. It has been shown that this approach is highly effective [84]. However, these methods usually ignore information such as comments, dependency files etc — information that may help to locate code that is not highly similar in structure, but similar in "spirit", i.e. addresses the same issue (such as two different sorting methods).

1.5.2 Retrieval using Conceptual Graphs

Conceptual graphs were identified as an abstraction layer for information that can be useful for classifying and retrieving it. Work has been done on usage of

CGs for unstructured information retrieval [92, 57, 65], using parsing of the natural language to build the structure of the document, with reported good results. Conceptual graphs were also used with varying success for retrieval of legal arguments [21], medical information [11], and multimedia documents [58, 93]. Clearly, the common feature of these document types is, just like source code, the inherent nature of the structure inside the contents. There is also work regarding usage of CGs for structured document classification and retrieval [43, 89]; this work relies on manual annotation and a WordNet-like extensive ontology.

An area which is closely related to conceptual graphs is *Description Logics* (DL) [2], a class of logic-based knowledge representation languages. Description Logics are especially useful in modeling database-like settings and provide a set of powerful reasoning and subsumption methods. DL has been used for information retrieval [45, 8]; in a nutshell, the idea is to use formal reasoning methods to show relevance of documents to queries. However, these methods tend to be too rigid and require either an enormous amount of world-knowledge or a very strict relation (such as subsumption) between the document and the query. There is no published work on state-of-the-art retrieval systems based on Description Logics (or other logics) that are comparable with leading retrieval models today, i.e., vector-space and probabilistic [4]: furthermore, there is evidence that computationally, these methods are impractical [15].

Chapter 2

Representing Source Code as Conceptual Graphs

As mentioned earlier, one of the challenges of retrieval from source code is the interconnected nature of the structure and content, rather than the common structured text form of markup languages. In this chapter we discuss the process of separating the structure from the content and representing it in a conceptual graph form.

2.1 Source Code Representations

Structural source code representations are roughly divided into two categories: *Syntactic* representations and *Semantic* representations. An additional *Hybrid* approach merges ideas from both of the categories.

- *Syntactic* methods focus on annotation of the code, transforming it into more common structured document formats. Usually, the markup is done in XML or other SGML variations, such as srcML [42] and JavaML [3]. This markup provides very good low-level, detailed representation of the code, and is especially useful for tasks where maximal knowledge regarding the code is required, such as syntax coloring, code indexing (context-tagging) etc; however, the amount of abstraction achieved is limited.

- *Semantic* methods start where the syntactic methods stop: using various tools, the code is abstracted and presented in higher-level representations. The academic and industry de-facto standard for software modeling is the Unified Modeling Language [44][83]. UML provides a rich set of abstract views of software, including *structural modeling*, *behavioral modeling*, *interaction paradigms*, and *constraints*; tools that follow these techniques and employ them for software engineering and re-engineering (and to a limited

extent – reverse-engineering) are widespread, most notably the Rational family [66]. However, these methodologies are usually aimed at macro-modeling rather than micro-modeling, i.e. analysis of complete software suites and not a short piece of code.

- *Hybrid* methods are a combination of syntactic and semantic analysis: while retaining the low-level analysis required for handling short code fragments, they also provide some abstraction level. This is of course achieved at a cost: the representation is not as detailed as annotated code is, and does not supply as much conceptual knowledge as the semantic methods. A good reference point for such methods is provided in [47].

For the case of retrieval it seems that a hybrid method is appropriate, one where relatively abstract concepts are described, but details from the underlying textual data are also given. In the next section, we describe such a hybrid method for graph abstraction of the code.

2.2 Converting Code to Conceptual Graphs

2.2.1 A Taxonomy for Source Code

Before designing a mechanism to build conceptual graphs from code, we need to decide which concepts and relations we permit in our graphs and their semantics – the *support* of the graph (see section 1.3). This includes the concept and relation types, as well as an indication of which relations can be made between given concepts, and the possible referents for each concept type.

Based on [24], [76], [13], and on examination of various source code file we constructed a simplified model of code representation at the micro-level; the concepts of this model are presented in table 2.1, and are arranged in a hierarchy in figure 2.1.

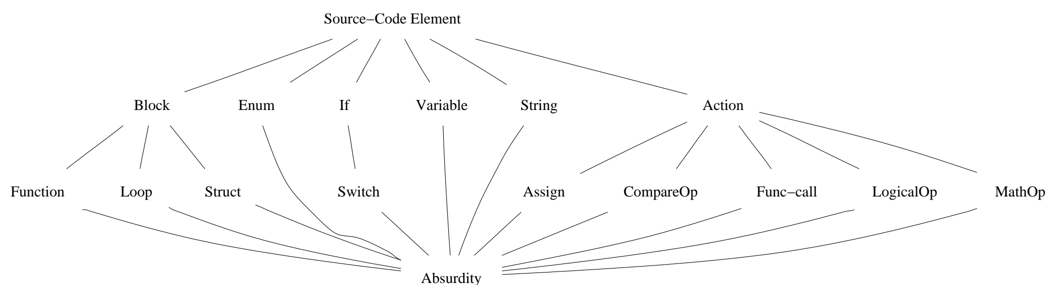

Figure 2.1: Hierarchy of the Source Code Canon

Name	Description
Assign	Assignment of value, or operation including assignment such as "+="
Block	A set of other concepts, logically grouped together
CompareOp	A binary comparison, such as "$\leqslant$", "$\neq$" etc.
Enum	An enumerated set of values
Func-call	An execution of a function
Function	A declaration or definition of a function
If	A conditional branching statement
LogicalOp	A binary logical operation, such as "$\vee$", "$\wedge$" etc.
Loop	An iterative statement, dependent on a condition
MathOp	A binary mathematical operation, such as "+", "$\div$" etc.
String	Textual string; literals such as numbers are interpreted as strings too
Variable	An entity which holds values during the program execution
Struct	A named Block, containing variables only
Switch	A multiple-branch conditional statement

Table 2.1: Concept Types

Formally, we must define a partial order on these concepts, but since we do not use features of conceptual graphs that require such order, we may use any arbitrary order (say, lexicographic).

The possible referents of the concepts are as follows:

- String concepts always have an individual referent, which is text of any length.

- {Variable, Func-call, Function, Struct} concepts always have an individual referent, which is a legal identifier of the programming language (in C, for example, this includes strings containing alphanumeric characters and the "underscore" symbol, that do not start with a number).

- Block concepts may either have the generic referent ("*") or an individual referent that is a legal identifier as above.

- All other concepts may only have the generic referent.

  The relation types of the model, along with the concepts they can connect, are specified in table 2.2. For space reasons, *Action* is short for the concepts {Assign, CompareOp, Func-call, LogicalOp, MathOp}, and * is short for "any concept".

Name	From...	To	**Description**
Condition	If	*Action* String Variable	Specifies that this branching statement depends on the concept.
	Loop	*Action* String Variable	Specifies the condition which is checked to determine continuation of the loop iteration
Contains	*Action*	*Action* String Variable	The action is performed on or using this concept.
	Block	*	The block contains the concept.
	Enum	String	The string is defined in the enumeration.
	Function	*	The function contains the concept.
	If	*Action* Block	The branching statement executes the concept, depending on its condition.
	Loop	*Action* Block Variable	The concept is executed or initialized in the loop.
	Struct	Variable	The structure defines the concept.
Comment	*	String	The concept is commented with the string.
Defines	Block	String	The block contains a definition of the concept.
Depends	Block	String	The block requires a file as a dependency.
Jumps	Block	String	The blocks jumps to the label marked in the string.
Parameter	Function	String	The function definition contains the concept as a parameter.
	Func-call	String Variable	The function is called with this value. The function is called with this parameter.
Returns	Block	*Action* Variable String	The concept is the return value of the block.
Typedef	Block	String	The block defines the string as a type.

Table 2.2: Relation Types and their possible placement

2.2.2 Graph Construction Process

As discussed earlier, we require a tool that analyzes short code fragments at the micro-level, rather than doing a global code base analysis. We experimented with a number of approaches for this task. An examination of shallow pattern-matching techniques and various lexical analysis tools quickly revealed that they are too weak for the task, as the syntactic knowledge acquired is limited. We therefore decided to focus on syntactic parsing of the source code, an approach which is close to the conceptual graph ones. Programming language parsers, that usually are part of a language *compiler*, generate a specific parse tree – an Abstract Syntax Tree (AST) – from the code. The AST, similarly to the conceptual graph representation, is a formal description of the meaningful attributes and behavior of expressions in the code; unlike our source code conceptual graph, the AST is very detailed, providing no abstraction layer.

Our graph constructor is an extension of a parser for a programming language grammar (part of a compiler for the language), in which relevant concepts and relations are instantiated during the parse process according to the code and the described taxonomy. Where a new concept is required according to the grammar it is instantiated and connected to the existing concepts; a symbol-table mechanism similar to the one used in compilers is used to locate concepts in the correct context. Pseudo-code examples of fragments of the grammar are presented in figures 2.2, 2.3, and 2.4; we enriched all relevant parts of the grammar with such construction procedures for a complete process of graph creation from code. We refer to a graph constructed using this process as a *Source-code Conceptual Graph*, or *SCG*.

```
statement   →   if "(" expr e ")" statement s1
                    concept1 := create_concept(IF)
                    connect_concepts(concept1, e, CONDITION)
                    connect_concepts(concept1, s1, CONTAINS)
                (else statement s2)?
                    connect_concepts(concept1, s2, CONTAINS)
            |   ...
```

Figure 2.2: Fragment of the Graph Construction Grammar - `if` statement

However, one problem of using language parsing for construction of conceptual graphs representing the code is the exclusion of certain parts of a source code file, such as comments and dependency statements, from the parsing process. These parts are invisible to the parser since they are not actually part of the grammar of the language; usually they are removed prior to the parsing by a preprocessor. In the code retrieval case this was undesirable, since these specific parts contribute important information to the resulting graph. Additionally, using a standard pre-

statement → **for** “(“ expr e_1 “;” expr e_2 “;” expr e_3 “)” statement s
$concept_1$:= create_concept(LOOP)
connect_concepts($concept_1$, e_1, CONTAINS)
connect_concepts($concept_1$, e_2, CONDITION)
connect_concepts($concept_1$, e_3, CONTAINS)
connect_concepts($concept_1$, s, CONTAINS)
| . . .

Figure 2.3: Fragment of the Graph Construction Grammar - `for` statement

statement → expr e_1 “>” expr e_2
$concept_1$:= create_concept(COMPAREOP)
connect_concepts($concept_1$, e_1, CONTAINS)
connect_concepts($concept_1$, e_2, CONTAINS)
| . . .

Figure 2.4: Fragment of the Graph Construction Grammar - “>” expression

processor would replace all dependency statements (`#include`s) with an actual document, instead of preserving the statement of dependency on that document.

Inserting these tokens into the grammar of the language is not a good solution for reasons we will not discuss here (basically, they make the grammar ambiguous and require a very strong lookahead mechanism to handle); our solution included implementation of a bypass channel through which this “extra-grammatical” information was transferred through the parser and included in the graph.

In addition to the construction process described, some technical measures were taken, for issues such as whitespace unification, redundant character removal, etc.

2.2.3 Examples

We now show two examples of code and the conceptual graph resulting from the construction. Since “real-life” examples produce large graphs that take up a lot of display space, we choose two toy examples. A birds-eye view of a representation of a real function (dealing with testing of matrix operations) can be seen in figure 2.9, and demonstrates how complicated the graphs quickly become.

The first example contains a very short function with a simple loop and a comment (figure 2.5):

```
void aFunction(int n, int* pInt)
{
    // just decrease pInt according to n
    while (n > 0) {
        *pInt--;
    }
}
```

Figure 2.5: `example1.c`

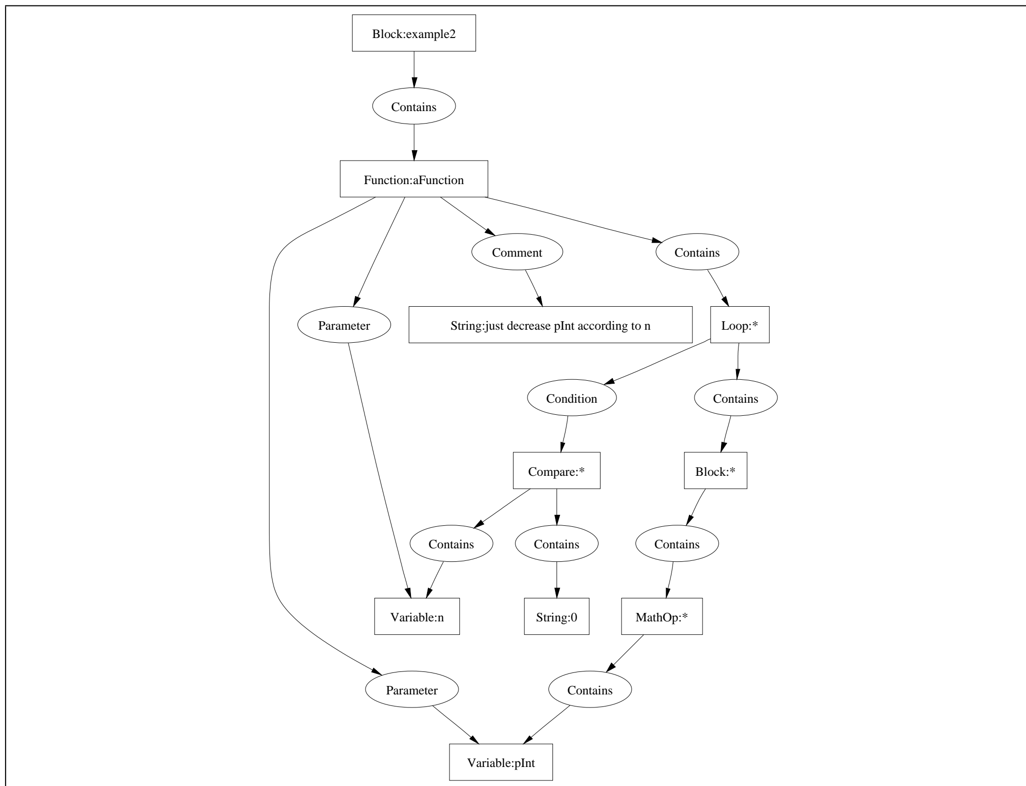

Figure 2.6: Conceptual graph for `example1.c`

The second example is a bit longer, with a dependency file, a value definition, and a short `main` that performs a mathematical operation and prints it (figure 2.7):

The average size of the graph, relative to the code from which it was constructed, varies largely and depends on settings such as the programming language, the coding style, the amount of comments and others. An estimation of the average graph size for a specific task is given in section 4.1.

```
#include "stdio.h"

#define RET_CODE -1

int main() {
        int i = 10;
        int j = 20;
        int mul = i * j;

    printf ("i * j = %d\n", mul);

    return RET_CODE;
}
```

Figure 2.7: `example2.c`

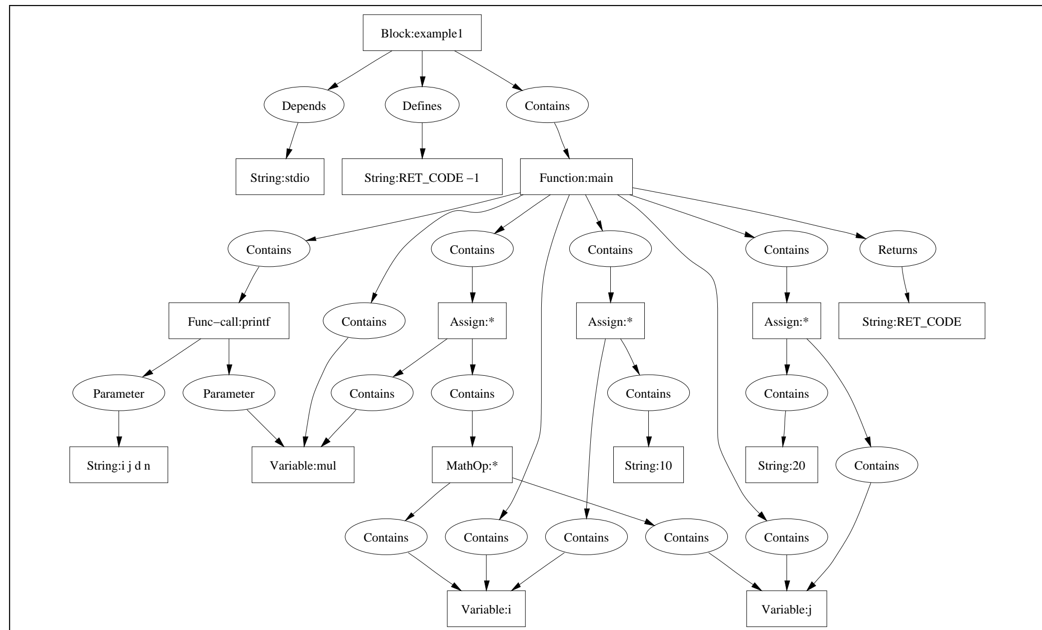

Figure 2.8: Conceptual graph for `example2.c`

We have presented a a concept hierarchy and associated relations for conceptual graphs representing source code – *SCGs*. Using this hierarchy, we described procedures for constructing such SCGs; the resulting graphs are a compromise between low level, detailed representations of source code and abstract code representations. Although the representation is not complete and can be enhanced to support more features of source code, it allows us to continue to the next step, which is construction of a retrieval process for the graphs.

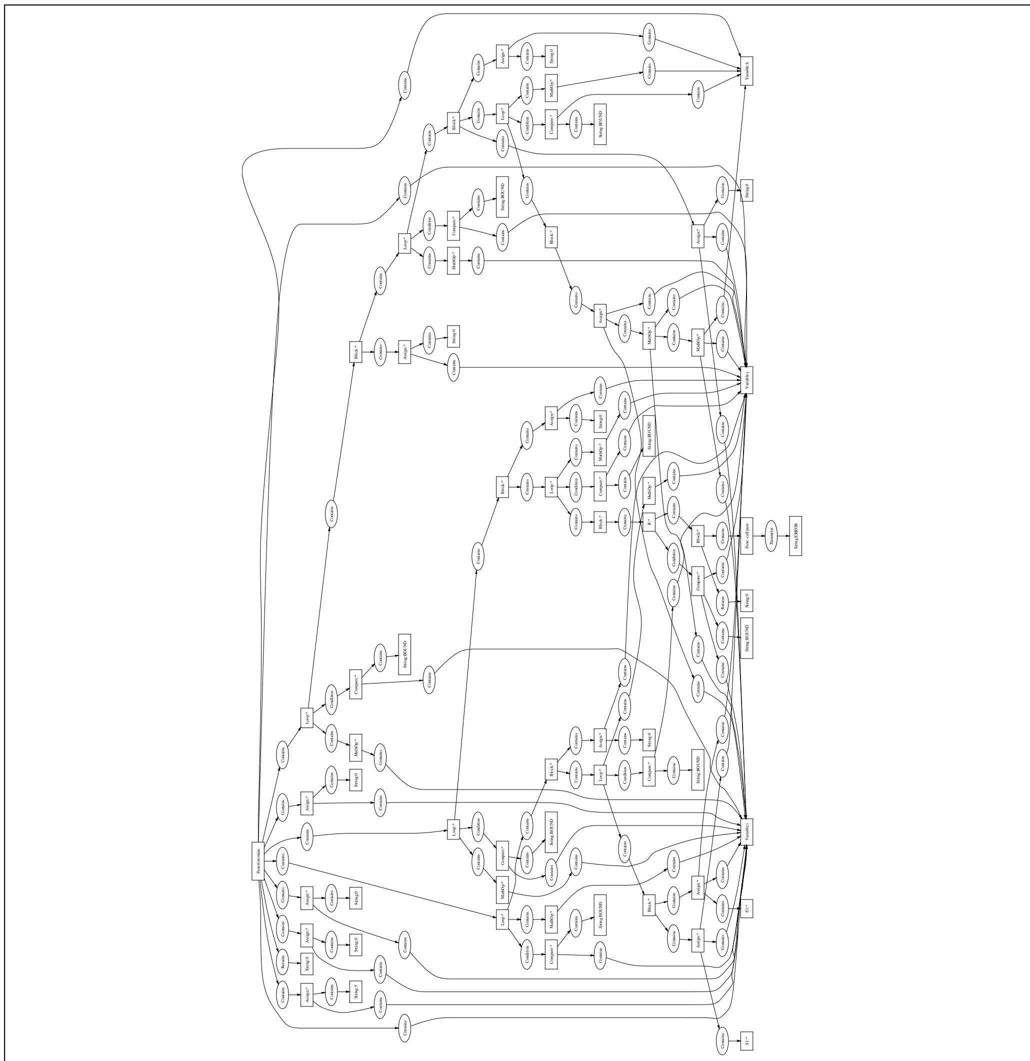

Figure 2.9: Conceptual graph representation of a "real-life" function

Chapter 3

Code Retrieval using Conceptual Graph Representations

We now turn to a description of the actual task at hand: using the conceptual graph representation for retrieving source code.

3.1 The Retrieval Model

Before we design our retrieval procedure, we must abstract one level and define the ingredients of our retrieval model. All retrieval models contain a definition of (at least) the following basic elements [4]: *documents*, *queries* or *information needs*, and *relevance*. We will now define them in the context of graph-based code retrieval.

- The *documents* are the units of information to be retrieved. In this case, they are source code files which are all written in the same well-defined programming language. We assume all files are written according to the grammatical rules of the programming language, and ignore those who do not comply with this. This is not a very strict constraint since the retrieval process is aimed at existing, working code, and since tools that verify the grammaticality of source code (compilers, interpreters) are used by everyone who is creating such documents.
- The *queries* in our model are also grammatical source code texts.
- The definition of *relevance* is more complex. We define a document d to be relevant to query q if the code in d and the code in q perform an identical or related task, or more generally – if a programmer writing q and needing examples of similar code would find d helpful. This is of course a subjective

measure, but this can be said about all relevance judgments in the context of information retrieval: were they not subjective, the ideal retrieval procedure could be explicitly stated and implemented.

After defining our top-level concepts, it is now possible to discuss the actual procedures for retrieval using the graph representations.

3.1.1 The Retrieval Process

All information retrieval systems — whether they are aimed at unstructured text, semi-structured text, or other types of documents — operate in the same spirit. The ingredients of a retrieval system include:

- A document collection $D = \langle d_1, d_2, \ldots, d_n \rangle$.
- A query q.
- A function f which maps a query and a document to a value which is ordered (usually, a real number). Usually, this function is a similarity measure between two documents of the same format, and the query is considered to be a document or transformed to one.

The retrieval process is then a straightforward one: for each document d_i in D, compute $f(q, d_i)$. Then, rank the documents according to the order between the computed values, to get their relative relevance to the query.

In many cases, a preprocessing step is performed prior to the retrieval itself, mainly for efficiency reasons. In this step the document collection is converted to an intermediate format, to make the computation of f easier. The conversion may include a number of separate stages; for example, in the application of the vector-space retrieval model to natural language text retrieval, the preprocessing step includes optional stemming, optional stopping, conversion of the documents to vectors of terms, and creating reverse-indexes of them according to the terms.

In essence, retrieval of source code is no different than the described algorithm: first, the source codes text files are preprocessed and converted to an intermediate format; when a query is presented, it is also converted to this format. Then, using a similarity measure between two instances of this intermediate representation, the code files are ranked. In the case of retrieval using CGs, the intermediate format is an SCG, and the conversion process was described in the previous section. The process does not include indexing (see discussion in section 5.1).

We are now left with defining a similarity measure between two SCGs that will approximate the relevance notion discussed earlier; this will be done in the following section.

3.2 A Similarity Measure for Source Code Conceptual Graphs

3.2.1 Retrieval Similarity Measures

Measurement of the similarity between two information entities can be thought as an attempt to find the amount of shared information between the entities and comparing it to the amount of distinct (non-shared) information of both. This is a fundamental concept in information retrieval, where an information need is usually compared to multiple sources of information, and the latter are ranked according to their “relevance” to it - many times estimated using the similarity to them.

Standard textbooks [70, 4] introduce many similarity measures; 67 different measures were identified and compared in [55]. The similarity measures used to retrieve information range from ones comparing numerical feature vector representation of the information, such as cosine similarity and Dice coefficient, to distance-based measures that assume the domain is represented in a hierarchical network. However, most measures are based on the same idea, where the mutual and disjoint information of the entities is formalized and compared.

3.2.2 Comparing Conceptual Graphs

Calculation of the distance between conceptual graphs is an important notion not only in the retrieval settings, but also for case based reasoning and for machine learning methods. A number of techniques exist for conceptual graph comparison: most importantly, a family of projections and morphisms was defined already with the presentation of CGs in [80] and extended later (for example, in [31]). Sowa also discussed the idea of *semantic distance*, a metric related to the distance between two concepts in the knowledge canon (but not extended to an entire graph - see also [25]).

The main disadvantage of morphisms is their strictness: in essence, they are aimed at locating identical graphs or subgraphs. Such similarity measures are unfit for “fuzzy” matching criteria, the ones needed for IR, although theoretic work about using them in the IR context has been done [33].

To address this issue, more relaxed measures were developed. In [92], a similarity measure is presented which is composed of a *conceptual similarity* and *relational similarity*; the first one measures the amount of shared concepts between the graphs, and the second – the structural similarity, measured as a ratio between the size of the overlap graph between the graphs and their combined size. Another approach, based on defining a set of legal transformations between graphs (each associated with a cost), is presented in [9]. A method for comparing graphs with different canons is presented in [22], but most of the work focuses on uniting the canons and the comparison itself is again based on morphisms. A similar approach,

which requires definition of an interest function for the graph, is discussed in [62].

The problem with most of these techniques is that they tend to require a high level of structural similarity, basing the similarity on morphisms and generalizations. In this way, the importance of the information in the concept nodes itself is assigned a much lower importance. Structural fuzziness is permitted, but at the cost of complex prerequisites (sets of transformations between graphs, interest functions). Additionally, the information contained in the concepts is not compared flexibly, but only used as a "classification" of the nodes, to decide if they are matching or not. The actual distance between two nodes, semantic or other, is not taken into account, as the "atomic operation" of the comparison — the operation of comparing two concepts — is a simple match with a binary result.

For example, the reported similarity measures will not render the graphs in figure 3.1 highly similar, although they may be related:

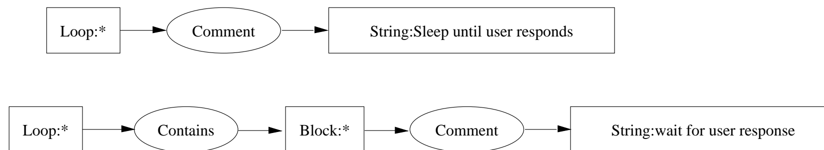

Figure 3.1: Related graphs

Requirements

A good similarity measure for conceptual graphs, like any similarity measure used for information retrieval, should follow a number of "rules of thumb". Given a graph query G_q, a similarity measure sim between graphs and a collection of graphs $\{G_1, G_2, \ldots, G_n\}$, it would be desirable for sim to have the following properties:

1. $\forall G_i\ \forall G_j,\ sim(G_i, G_j) = sim(G_j, G_i)$: a symmetry requirement.

2. $\forall G_i\ \forall G_j,\ sim(G_i, G_i) \geqslant sim(G_i, G_j)$: a graph should be maximally similar to itself.

3. "Correctness": if graphs G_i, G_j represent fragments of code which are decided to be "similar" (according to the definition set earlier) then $sim(G_i, G_j)$ should have a higher value than $sim(G_i, G_k)$, where G_k is another graph that represents a less similar code fragment (as decided by the same means).

4. The complexity of computing sim should be "reasonably" bounded.

Overview and Intuition

As mentioned earlier, the similarity measure captures the relative shared information between to entities, relative to their distinct information. In a conceptual graph, there are a number of layers of information: first, the knowledge taxonomy behind the graph. Since all the SCGs share the same canon, this aspect will be ignored. Second, there is the information represented by the concept nodes: this includes both the node type (LOOP, STRING) and, if applicable, the referent data in the node (STRING:IOSTREAM.H). Third, the structure of the graph is informative in itself, both since the relation nodes may hold attributes and since the structure provides knowledge about how concepts relate to each other. In [92], the latter two sources of information are compared separately and referred to as "conceptual similarity" and "relational similarity".

To measure the similarity between graphs, we would like to use both the notion of "conceptual similarity" and "relational similarity", but in a way that is interleaved; the strong connection between the structure and the content in the case of source code should also be expressed in the similarity measure used for it. An idea which seems promising is to compare the graphs node-by-node, traversing them and acquiring through the traversal both information about the content of the nodes and the structure of the graph. For a good traversal, both in terms of comparison accuracy and complexity, we should locate good candidate nodes from which the traversal has a high chance of giving meaningful results: we would like to start the traversal of both graphs from nodes which share a lot of content. For example, if we are comparing two SCGs which represent loops (but maybe one of them is a `for` loop and the other a `while` and the variable names the loops are using are also different), it will seem reasonable to start the traversal either concurrently from the nodes representing the LOOP concept, or from the nodes representing the variable, but not traverse one graph from the loop node and the other from the variable node. We refer to such a concept, which is a good candidate for comparison to another, as its "most similar concept". But, even if we can discover the concept nodes from which we should start, we are still faced with a complex situation. We can easily measure the similarity between the nodes by comparing their type and their referents, but comparing their context in the graph — the concepts to which they are related — is a harder task. We must identify, at every stage of the traversal, "similar edges" out of the two compared nodes (to follow them and continue the comparison). These edges are not guaranteed to exist at all, and even if they exist, their identification can be very complex [26].

Instead of performing a simultaneous traversal when comparing two nodes, we opt to use a simpler technique: collapse the information "around" the node into it. In this context, the information "around" the node is the one contained in the relations and concepts around it. This information should be of lower significance than the information kept in the node itself; a natural parameter to use for deciding

how much lower the significance should be is the relation type between the two nodes. If we assign a weight to every relation type, we can easily include the "weighted down" information around the nodes being compared, as depicted in figure 3.2.

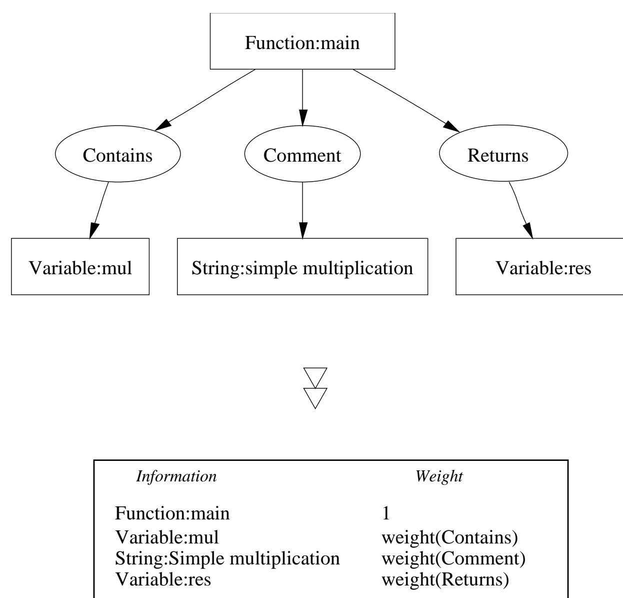

Figure 3.2: Concept Extension

Once the information expansion step has been defined, we can apply and continue to apply it recursively, and include in the comparison also information that is referenced from the information we are including, etc. For the same reason, the information should be added to the node being compared with a weight which is a combination of all the weights of the relations on the path from the original node to the information.

At this stage, the "expanded nodes" that we are comparing contain multiple bits of information (each with its own weight). These information bits can be of different types - one can be a string (coming from a comment) and another a number, some value assigned to a variable. A naive approach to coping with this diverse information is to treat all information as strings and compare it as such. A more involved approach would be to store the information bits in separate partitions, according to their type, and comparing each partition to its matching partition, summing up the intermediate results.

Repeating this process for all nodes in the graph yields a similarity score between the two graphs, as follows:

```
sim := 0
foreach concept c1 ∈ G1
    msc := most similar concept to c1 in G2
    sim := sim + similarity(c1, msc)
end foreach
foreach concept c2 ∈ G2
    msc := most similar concept to c2 in G1
    sim := sim + similarity(c2, msc)
end foreach
```

We are still left with defining a method for locating the "most similar concepts": our approach is to regard a concept $c_2 \in G_2$ as the most similar one to $c_1 \in G_1$ if there are no concepts in G_2 with a higher similarity to c_1, using the similarity measure discussed.

Limitations

There are two main drawbacks of this similarity measure: the first one is loss of a lot of the structural information in the graph. As stated earlier, graph comparison methods tend to focus on structure, while here the only usage of structure is for distance calculation and weight assignment. In our view, for the specific conceptual graphs we are dealing with, this is not necessarily a drawback but can also be viewed as an advantage: in such a graph, the "real" information is in the concepts, and the relations indeed specify "how important" the connection is between two concepts. The second drawback is complexity; the suggested method requires quite a lot of computations and for large graphs seems problematic. We will return to this issue later, in sections 3.2.3 and 5.1, where we also discuss possible variations of the process.

Going back to the (desired) requirements we have set in section 3.2.2, it is easy to see that the proposed measure conforms to requirement 1. As for requirements 2 and 3, they will be evaluated in chapter 4. A bound on the complexity of the process, as needed by requirement 4, is given in section 3.2.3.

Details

We will now formally define the components of the discussed similarity measure.

First, we define a number of weights associated with components of the conceptual graph:

Concept Type Weight : Marked $w_c^t(c)$, the concept type weight is a value measuring the "importance" of a concept type, and is fixed for all concepts of the same type. For example, a concept of type STRING should probably have a higher weight then the type IF-STATEMENT.

Concept Referent Weight : Marked $w_c^r(c)$, the concept referent weight is a value proportional to the amount of information kept in a concept. The

amount of information is simply measured by the length of it (in bits or bytes for example).

Concept Weight : The concept weight, $w_c(c)$, is simply the product of the two components of the weight of the concept: $w_c(c) = w_c^t(c) \cdot w_c^r(c)$.

Relation Weight : The relation weight, $w_r(r)$, is similarly a value associated with a relation, and is composed solely of a fixed value according to the relation type (a "relation type weight"). It can also be referred to as Relation Type Weight, or $w_r^t(r)$.

Next, we define the basic similarity notions between the concepts in the graph:

Concept Type Similarity : This value, marked $sim_c^t(c_i, c_j)$, measures the similarity between the types of two concepts. For example, it gives a notion of how much a LOOP concept type is similar to a BLOCK type, and can be simply predefined in a matrix of all concept types. An even more naive approach would be assigning 1 to this value if the concept type is identical or if one concept type is inherited from the other, and some very low value otherwise.

Concept Referent Similarity : Marked $sim_c^r(c_i, c_j)$, this value measures the similarity between the content of the concepts. If the referent is empty in both concepts, it is defined as 1. Otherwise, we use the Levenstein string-distance value (sometimes referred to as "edit-distance", [14]) to quantify it; this standard method measures the number of "edits" required to turn from string A into string B and defines their relative distance as dependent on the number of edit steps and their length.

Concept Similarity : This value, marked $sim_c(c_i, c_j)$, measures the similarity between two concepts (without the contextual information, i.e. the relations and concepts around the compared concepts). It is simply the product of the concept type similarity and concept referent similarity between the concepts, normalized by their weight: $sim_c(c_i, c_j) = sim_c^t(c_i, c_j) \cdot sim_c^r(c_i, c_j) \cdot w_c(c_i) \cdot w_c(c_j)$. We chose a product of the weights rather than a sum of them to give a high significance to very low weights: if one of the weights is very low, its effect on the entire similarity should be substantial.

We now formalize the "information extension" process described earlier:

Extended Concept : This is an extension of the conceptual graph standard concept. A standard concept has only one type and one referent; an extended concept has a set of $\langle concept_type, concept_referent \rangle$ pairs. This extension can easily transformed back to standard conceptual graph concepts by extending the concept lattice with new concept types which are in fact a representation of the combined "concept sets", and change the possible referents

accordingly. This notion is required for the concept extension step: we need a formal way of representing multiple bits of information in a single concept.

Weighted Extended Concept : This is an extended concept with a weight associated with every referent (every information bit).

Concept Extension : This is a process of extending the knowledge in a concept to include information from another concept with some weight; it can also be viewed as a function $f : C \times C \times \Re \rightarrow C$, i.e., a function that takes a pair of (possibly extended) concepts and a number, and produces a new (weighted extended) one. Let c_1 be the concept to be extended with c_2 and with weight x; then the result of the extension $ext(c_1, c_2, x)$ is defined as follows:

- If $c_1 = c_2$, then $ext(c_1, c_2, x) = c_1^*$, where c_1^* is identical to c_1 but with all referent weights multiplied by x.
- Otherwise, for all concept types, we add the referent information in c_2 to the corresponding referent in c_1, with weight x.

Extended Concept of order n : The extension of a concept c of order n, marked $ext_n(c)$, augments the information in concept c with the information kept in the concepts related and recursively in the concepts related to them, up to depth n. It is defined as follows:

- $ext_0(c) = ext(c, c, 1)$: The extended concept of order 0 is the concept itself.
- $ext_1(c)$ is defined according to the number of concepts related to c: if there is one related concept c_1, which is related to c with relation r_1 then $ext_c(1) = ext(c, c_1, w_r(r_1))$. Similarly, for n related concepts, $ext_c(1) = ext(\ldots(ext(ext(c, c_1, w_r(r_1)), c_2, w_r(r_2)), \ldots, c_n, w_r(r_n))$. In simpler words, it is just the extension of c with all the concepts "around" it, where the extension weight for any concept is determined by the weight of its relation to c.
- $ext_n(c)$ is the extension of c with all related concepts, where they themselves are extended to order $n - 1$.

As a special case, $ext_\infty(c)$ is the maximal extension of c – extending over all reachable nodes. We can of course set n to be $|E|$, the number of edges in the graph - and get the same.

Equipped with these definitions, it is now possible to discuss the similarity between extended concepts, and its derivatives:

Extended Concept Similarity : Corresponding to the notion of concept similarity, this measures the similarity between extended concepts. Since both concepts may have more than 1 concept type and referent, we simply sum over all possible pairs of concept similarities between the concepts. So, assume the two extended concepts c_1, c_2 have the referents $c_{1,1} \dots c_{1,n}$ and $c_{2,1} \dots c_{2,m}$, where the concept types are $T(c_{1,1}) \dots T(c_{1,n})$ and $T(c_{2,1}) \dots T(c_{2,m})$, the referents $R(c_{1,1}) \dots R(c_{1,n})$ and $R(c_{1,1}) \dots R(c_{1,n})$, and the weights $w_{1,1} \dots w_{1,n}$ and $w_{2,1} \dots w_{2,m}$ respectively. The extended concept similarity is then

$$extsim(c_1, c_2) = \sum_{i=1}^{n} \sum_{j=1}^{n} w_{1_i} \cdot w_{2_j} \cdot sim_c(c_{1,i}, c_{2,j})$$

where c_i, c_j are concepts created by leaving only type/referent i and j respectively out of the concepts c_1 and c_2. For example, given the two extended concepts in figure 3.3, and using weighting schemes where the type similarity is 1 if the types are identical and 0 otherwise, and where the referent weight is proportional to its length, we calculate the similarity as follows:

Information	*Weight*
Loop:*	1
Block:*	0.5
String:wait for user response	0.45

Information	*Weight*
Loop:*	1
String:Sleep until user responds	0.9

Figure 3.3: Sample extended concepts

$$\begin{aligned} extsim &= 1 \cdot 1 \cdot sim(\text{[Loop:*],[Loop:*]}) \\ &+ 1 \cdot 0.9 \cdot sim(\text{[Loop:*],[String:Sleep} \dots \text{]}) \\ &+ 0.5 \cdot 1 \cdot sim(\text{[Block:*], [Loop:*]}) \\ &+ 0.5 \cdot 0.9 \cdot sim(\text{[Block:*], [String:Sleep} \dots \text{]}) \\ &+ 0.45 \cdot 1 \cdot sim(\text{[String:wait} \dots \text{], [Loop:*]}) \\ &+ 0.45 \cdot 0.9 \cdot sim(\text{[String:wait} \dots \text{], [String:Sleep} \dots \text{]}) \\ &= 1 + 0 + 0 + 0 + 0 + 49 \cdot 0.41 \cdot \frac{23+26-12}{23+26} = 15.98 \end{aligned}$$

Extended Concept Similarity of order n : Marked $extsim_n(c_i, c_j)$, this value measures the similarity between two concepts with contextual information up to depth n: $extsim_n(c_i, c_j) = extsim(ext_n(c_1), ext_n(c_2))$.
It is simply the similarity between the extended concepts c_i, c_j (of order n).

Maximally Similar Concept : Given a concept $c_1^* \in G_1$ and another graph G_2, this is a concept $c_2^* \in G_2$ that has the highest concept similarity to c_1^* (without contextual information): $\forall c_i \in C_1, \forall c_j \in C_2, sim_c(c_1^*, c_j) \leqslant sim_c(c_1^*, c_2^*)$. We will use the notation $MSC(c_1, G_2)$ for this concept; note that there can exist

more than one Maximally Similar Concept; in that case, one can be selected randomly.
It is possible to include contextual information when searching for the MSC (by defining a "Maximally Similar Concept of order n", that uses $extsim_n$ instead of sim_c), at an additional computational cost.

Finally, we can define the similarity measure between two SCGs G_1, G_2 using the above definitions and notations, as follows:

$$\begin{aligned} sim_n(G_1, G_2) \quad &= \quad \sum_{c_i \in G_1} extsim_n(c_i, MSC(c_i, G_2)) \\ &+ \quad \sum_{c_j \in G_2} extsim_n(c_j, MSC(c_j, G_1)) \end{aligned}$$

Actually, this gives us a family of similarity measures, since we can select the order of the Extended Concept Similarity, both for computing the MSC and for the matching itself.

3.2.3 Complexity

One of the requirements for the similarity measure was bounding the complexity of it (see section 3.2.2). We mentioned earlier that graph-based similarity measures come at a high cost: we now analyze the complexity of calculating the various building blocks discussed in the previous section, and the complexity of the process as a whole.

First, we give the complexity of the defined weights; since the length of the information kept in the nodes is of limited size, it will be treated as constant.

- Concept Type Weight: $O(1)$ - simple lookup.
- Concept Referent Weight: $O(1)$ - determined by information length.
- Concept Weight: $O(1)$ - product of previous two measures.
- Relation Weight: $O(1)$ - again, lookup.

Next, we discuss the complexity of the basic similarities between concepts:

- Concept Type Similarity: $O(1)$ - comparison of two values.
- Concept Referent Similarity: $O(1)$ - comparison of constant amount of information.
- Concept Similarity: $O(1)$ - product of two $O(1)$ measures.

We now bound the complexity of the concept extension process, and the derived similarities:

- Concept Extension (of order n): An extension of the concept requires calculation of the extension with all neighbor nodes; at most, we will have to extend to the entire graph, or make $|G|$ extension steps (each of them at cost $O(1)$). So, the final complexity is $O(|G|)$.
- Extended Concept Similarity: We must compare every possible referent type; in the worst case there are $|G_1|$ referents for the first concept and $|G_2|$ for the second, and the complexity is $O(|G_1| \cdot |G_2|)$.
- Extended Concept Similarity of order n: First, we need to calculate the extended concepts - this will be $O(|G_1|)+O(|G_2|)$. Then, we need to compare them, which is $O(|G_1| \cdot |G_2|)$ as shown earlier, so the final complexity is also $O(|G_1| \cdot |G_2|)$.
- Maximally Similar Concept: We must compare a concept $c_1 \in G_1$ to all concepts in G_2, so we get a complexity of $O(|G_2|)$. If we choose to look for an MSC of order n, we similarly get $O(|G_2| \cdot (|G_1| \cdot |G_2|))$.

Finally, we can calculate and bound the total complexity of the graph similarity measure and the retrieval process:

- Conceptual Graph Similarity: First, we must find all Maximally Similar Concept pairs for the concepts of G_1 and G_2. This costs, as shown above, $O((|G_2| + |G_2|) \cdot (|G_1| \cdot |G_2|))$. Then, we require a calculation of $|G_1| + |G_2|$ Extended Concept Similarities of order n, or $(|G_1| + |G_2|) \cdot O(|G_1| \cdot |G_2|)$. The total complexity is then $O((|G_1| + |G_2|) \cdot O(|G_1| \cdot |G_2|))$, for MSCs of order 0.
  Let $|G| = \max(|G_1|, |G_2|)$, then we arrive at a total complexity of $O(|G|^3)$.
- Retrieval using CG Similarity: Given a document collection with N documents, we must perform N comparisons, for a total complexity of $O(|G|^3 \cdot N)$.

We end up with a polynomial complexity for the similarity measure; while this is a typical result for graph algorithms, it is much higher than the linear complexity (depending on the query size) of classic textual similarity measures [71]. Moreover, the "atomic operation" we have is string distance measuring, and not simple term comparisons, resulting in our $O(1)$ measure of it being in practice substantially higher than the equivalent $O(1)$ operation in the classic similarity measures. Finally, the fact that no preprocessing takes place requires us to compare the query to the entire collection, unlike classic models which use *indexing* to acquire fast access to the documents that should be compared.

We discuss practical implications of the high complexity and some methods for handling it in section 4.1.8, and suggest further ways to reduce it in section 5.1.

3.2.4 Additions and Modifications

Many changes and enhancements can be introduced into this basic framework; we experimented with some, and list others in section 5.1. Among the modifications that we implemented, based on improving results of initial experiments and on practical constraints, were filtering procedures for reducing the number of graphs being compared; more details on this can be found in section 4.1.8.

An additional modification we used was *document length normalization* of the similarity measure. Document length normalization is a known performance booster in retrieval systems [70], and we expected similar findings here. Basically, this normalization gives a "fair chance" to documents of all lengths by weighting down long documents (which are more likely to contain matches) and weighting up short documents similarly. For the document length, we used a standard string representation of the graph, including all structural and content information. Given graphs G_1, G_2 and their similarity *sim* as defined above, we computed the normalized similarity in a simple way:

$$sim_{norm}(G_1, G_2) = \frac{sim(G_1, G_2)}{|G_1| + |G_2|}$$

Experiments proved that using the normalized similarities yielded equivalent precision scores to unnormalized ones (and in a few cases, slightly better), so the normalization was added to the similarity definition.

Again, we discuss further possible modifications in section 5.1.

3.2.5 Summary

We presented a similarity measure for the graphs that we construct from source code, that uses both the information from the code and the knowledge about how this information is interconnected. At this stage, we face two issues. First, we want to validate the fitness of the measure to the task at hand, and see if the assumptions we made have concrete support. Second, the complexity of the measure seems rather high, and we need to test whether the method is applicable; if not, perhaps various constraints help to reduce this complexity.

We therefore proceed by conducting experiments using our retrieval method and evaluating their results.

Chapter 4

Evaluation

In this chapter we describe the experiments carried out to evaluate the SCG retrieval method and discuss their results. We proceed as follows: first, we describe the environment in which the experiments were made, the types of experiments and the way in which they were compared to baselines. We then present the results of the experiments, starting with experiments for tuning of various parameters and moving on to the actual retrieval experiments; a discussion follows each result presented.

4.1 Experimental Setting

An evaluation platform for retrieval experiment includes a number of ingredients [4]. First, a document collection fit for the task needs to be defined. Next, a set of experiments should be described, each stating a goal and including a measure for evaluating its results and procedures for relevance assessment. Finally, in a setting where new models are introduced, a baseline model should be described and used to compare the results to.

We define these ingredients in the following sections.

4.1.1 Document Collection

The main problem in evaluation of source code retrieval is the lack of assessed corpora. While open-source projects exist in many programming languages – some of them spanning millions of lines of code, such as *OpenOffice* [56] or *KDE* [39] — there is no publicly available corpus of code which is grouped in clusters of "similar code", or a corpus of code annotated with relevancy assessments regarding queries. Since assessing an entire corpus in this way is a very laborious task, it was decided to obtain a corpus that contains, *with high likelihood*, many clusters of similar documents.

The selected document collection was a subset of the source code of `gcc` - the popular GNU compiler suite [28]. The subset includes the compiler's test-suite for the C language, and consists of 2932 files written in GNU-C (standard C, with a number of extensions). The total number of lines of code is 88363 and the number of tokens is 326238. The tested gcc version was 3.3.2 (released October 2003, latest at the time of the experiments). The reasons for choosing this corpus include:

Language. C, or GNU-C in this case, is a relatively unambiguous language, with a grammar which is not too complicated on the one hand but allows complex structure on the other. Additionally, the language is very popular for large-scale software projects (the Linux kernel, *gcc/gdb* and *Perl* are all written in GNU-C, to name just a few); this renders the results of the experiments applicable to the required domain, i.e., enterprise software development. Finally, the frequent usage of C as a programming language ensures existence of many helper tools to simplify tasks such as tokenization and parsing.

Document Size. For the retrieval evaluation, relatively simple Perl programs were used, as the purpose was a proof-of-concept rather than efficient retrieval. It was therefore desirable to have a relatively small average document size for the collection; in the gcc test-suite, the average document size is 111.3 tokens and 30.1 lines. However, the approach itself is not dependent on the document size, and in any case chunking and "windowing" techniques can be used to generate smaller documents.

For this collection, the average ratio between nodes in a graph and lines-of-code was 2.49.

Structure. Being a test-suite, the collection includes many different aspects of the language, making use of literally all options GNU-C offers. Additionally, the test-suite is written by many different contributers, ensuring an inconsistent programming style and (with high likelihood) repetition of code. To make things even "better" (in terms of fitness of the corpus to the problem), the documents in the test-suite are in many cases cryptic, with meaningless variable names and with little documentation. This, of course, makes them an interesting test case for comparison and retrieval.

The following is an example of a typical piece of code out of one of the documents in the corpus (docid 3111):

```
#define MAX_COPY 15

char A = 'A';

int main () {
  int len;
  char *p;

  /* off == 0 */
  for (len = 0; len < MAX_COPY; len++) {
    reset ();

    p = memset (u.buf, '\\0', len);
    if (p != u.buf) abort ();
    check (0, len, '\\0');

    p = memset (u.buf, A, len);
    if (p != u.buf) abort ();
    check (0, len, 'A');

    p = memset (u.buf, 'B', len);
    if (p != u.buf) abort ();
    check (0, len, 'B');
  }
}
```

...and here is another piece of code, from a different document (docid 3103), which is similar to the previous one and probably shares a copy-paste relation with it:

```
#include <string.h>

#define MAX_OFFSET (sizeof (long long))
#define MAX_COPY (10 * sizeof (long long))
#define MAX_EXTRA (sizeof (long long))

char A = 'A';

main () {
  int off, len, i;
  char *p, *q;

  for (off = 0; off < MAX_OFFSET; off++)
    for (len = 1; len < MAX_COPY; len++) {
      for (i = 0; i < MAX_LENGTH; i++)
        u.buf[i] = 'a';

      p = memset (u.buf + off, '\0', len);
      if (p != u.buf + off) abort ();

      q = u.buf;
      for (i = 0; i < off; i++, q++)
        if (*q != 'a') abort ();
   }
}
```

As visible in the example, the code demonstrates many of the “sicknesses” of code in large project: lack of documentation, meaningless variable names, inconsistent coding convention and so on.

4.1.2 Experiments

The following experiments were carried out:

Parameter Optimization: First, a limited amount of experiments was performed to test various parameters within the graph retrieval process (match depths, relations weights). The purpose of these experiments was to function as a basic tuning mechanism for the parameters, which are later fixed for the actual retrieval experiments. An interesting side-effect of these experiments was insight into the significance of various settings for the retrieval, and hints regarding which additional optimization could be beneficial.

Identical Document Retrieval: For this experiment, we composed a list of queries which were documents chosen randomly out of the collection. For each query, the top ranking 10 documents were retrieved using the graph comparison mechanism. The purpose of this experiment was twofold: first, to serve as a sanity check for the entire retrieval process. When using a document from the collection as a query, we expect a good similarity measure to rank the document itself at a very high rank – ideally at rank 1, as stated in section 3.2.2. The second purpose of the experiment, perhaps a more important one, was a more "classic" retrieval experiment aim: to analyze the rest of the top ranking documents, and check whether they are relevant to the query. The number of queries issued was 25; this number is based on [86] and is considered the minimal amount needed to provide some statistical significance to the results.

Modified Document Retrieval: For this experiment we introduced a new set of documents derived from the documents used for the previous experiment. Each document was modified in a number of ways, based on code changes that have been reported as frequent in [88]:

1. Changing names of tokens (such as variables and functions). Where the old token names had "meaning", the new names were also chosen to have some meaning, i.e., `loop_counter` → `numLoop`. It should be noted, however, that in the vast majority of the cases the tokens were meaningless strings and were thus converted to other meaningless strings: `bmu1` → `cl_1`.
2. Comment modification - comments were changed in a way that preserved the natural-language meaning, but modified the phrasing, location, amount of details and so on. For example, the comment `will never be executed` was changed to `never reached`. Other comments were simply removed, and new comments were sometimes added.
3. Changing order of statements (where it does not effect the program).

4. Adding bogus statements.
5. Code style changes (`for` loops to `while` loops, reverse conditional statements, change order of parameters in function calls etc).

In this way, we created documents (to be used as queries) for which we were guaranteed the existence of at least one highly "relevant" document in the collection (according to our relevance definition): the document which was modified to create the query. The goal of this experiment was similar to our second goal in the Identical Document task: to evaluate the "retrieval effectiveness" of the method, this time using a more real-life scenario where retrieval is done with a query that is not identical to any document in the collection. The number of queries issued in this task was also 25.

Statistical Significance Tests: A statistical significance test determines the probability that a given result of an experiment occurred by chance. The *sign-test* is a specific significance test suited for comparing results of two retrieval methods [34] since it does not make any assumption regarding the results (whereas other tests assume certain properties about the distribution of the results).

The sign test is calculated as follows. First, we align the results of the two methods so that for the queries $i = 1 \dots n$ the results are X_i and Y_i respectively. Now, we define D_i to be the "difference" in the results, i.e. $D_i = Y_i - X_i$. With this notation, the test yields the following upper bound on the probability that the "null hypothesis" is correct, i.e. that the difference between the results is coincidental:

$$T = \frac{2 \cdot \sum I[D_i > 0] - n}{\sqrt{n}}$$

where

- n is the number of queries.
- $I[D_i > 0]$ is 1 if $D_i > 0$ and 0 otherwise.

In addition to the sign test, there are "multi-method" significance tests, aimed at comparing results of more than two retrieval methods (ANOVA [34]). Although we have multiple baseline methods, we do not use such significance tests; we compare our method only to one of the baselines, for reasons explained in section 4.2.3.

For our null hypothesis, we assume that our method is equivalent to the baseline method we are comparing it to (the baselines are specified in the section 4.1.6).

Method Combinations: In our final set of tests we combined the results of two different methods, the most successful baseline and the graph retrieval method. For the combination we used the same method as described in [38], i.e. a linear combination of the normalized scores of the two methods, with various λ values.

4.1.3 Measures

The standard measures for retrieval performance [4] are *precision* and *recall*:

$$Precision \equiv \frac{|Relevant \cap Retrieved|}{|Retrieved|}$$

$$Recall \equiv \frac{|Relevant \cap Retrieved|}{|Relevant|}$$

where

- *Relevant* is the set of relevant documents in the collection
- *Retrieved* is the set of documents retrieved by the method

Other often-used metrics are the *F-measure*, which is a combination of precision and recall, *R-Precision* which is the precision at rank $|Relevant|$, *precision@n* which is the precision at rank n, and various average values of the precision.

An additional measure which is common in the case of known item search — retrieval where a single item is known to exist and answer the query — is the *Reciprocal Rank*, or RR [87]. The RR is a value which is inversely proportional to the rank given by the retrieval method to the single relevant item. Given a query q with a single relevant item d, the RR score for a system will be $\frac{1}{rank(d)}$, i.e. 1 if the d was the top retrieved document, $\frac{1}{2}$ if it was the second, $\frac{1}{3}$ if it was the third and so on. The *Mean Reciprocal Reciprocal*, or MRR, is the average RR of all queries.

While measuring the recall requires assessing the entire collection, precision requires assessment of a fixed amount of documents only; since our document collection is not pre-assessed, we decided to focus on precision measures rather than recall. In such scenarios, and also in cases where the retrieval is aimed at a human user that will probably not look at more than a few results, it is common to use the precision@n measure and to fix n to low values. We have chosen to measure the **average precision@5** - the average percent of relevant documents in the top 5 documents retrieved, and the Mean Reciprocal Rank. For measuring MRR, a single correct document should be defined; for the Identical Document Retrieval we used the document identical to the query, and for the Modified Document Retrieval the document from which the query was originally derived. In both cases we ignored additional retrieved relevant documents.

The MRR measure serves two purposes. For the Modified Document Retrieval task, it serves as a primary measure for the method's accuracy: a good method should find the original document, given the modified one, and assign a high rank to it. For the Identical Document task, it is a secondary measure, functioning as sanity-check of the retrieval methods, assuming a good method should retrieve a document taken out of the collection at a very high rank.

The aims of the precision@5 measure are inverse to the MRR one: it is a primary measure for the Identical Document task, measuring the actual "relevant" documents found. For the Modified task, it is a secondary measure since relevant documents (other than the one that should be detected by the MRR score) are not even guaranteed to exist in the collection.

Other measured values included the percentage of queries for which the "exact document" (as defined for MRR) was found and the average number of relevant, non-exact matches in the top retrieved 5 documents.

4.1.4 Assessment

The results were assessed by a C-literate programmer; a document was defined as "relevant" to a query if the assessor decided that the query and the document perform an identical, similar or related task, or that the code in the document serves as an example/reference for someone writing the code in the query. As mentioned in section 3.1, this is a subjective measure – like most relevance judgments.

4.1.5 Retrieval Parameters

A weak point of the SCG retrieval process is the large number of free parameters which require tuning; we come back to this point in section 5.1. For the evaluation of the methods, we experimented with some of the parameters, but not with all of them (due to time limitations). The two main sets of parameters that were predefined were the *concept type weights* and the *relation weights*; for both, we manually assigned weights according to intuition (i.e., COMMENT is more important than CONTAINS, etc), and according to the concept hierarchy presented in figure 2.1: a concept did not have a lower weight than its parent concept. Limited experimenting resulted in other weights performing worse; this is reported in the next section.

For the *concept type similarity* between two types, we assumed 1 if the types are the same, and a small value otherwise.

4.1.6 Baseline Models

Probabilistic Retrieval

One of the simplest approaches to code retrieval is handling source code files as standard text files; since this method is straightforward, both conceptually and implementation-wise, it seems to serve as a good baseline for other code retrieval methods. The two mainstream approaches today to textual retrieval are the vector-space approach and the probabilistic model approach [4, 70] (with language modeling, the new kid on the block, advancing quickly [32, 16]); both are equally successful, with advantages and disadvantages to both. Initial experiments were conducted with both models, evaluating Okapi [67] (a probabilistic model) and the Lnu.ltc weighting scheme [71] with no feedback (a vector-space approach). The probabilistic model yielded slightly better results, and thus was chosen as a baseline.

In the standard probabilistic approach to retrieval, the document collection is first preprocessed and indexed; when presented with a query, all documents are ranked according to the probability that they are similar to a query; since the real probability is not known, estimates are used to calculate these probabilities. The different types of estimates used are the main differences between the various flavors of probabilistic retrieval.

The following is a description of the stages of the retrieval in this model:

- *Preprocessing.* First, each document was *chunked* to a number of separate documents. A chunk includes a single piece of code (such as a `function` or a `struct`) from the document, along with any global parts of it. The chunking algorithm is as follows:

```
i := 0
global := ∅
foreach line of code in file f
    if line contains OPEN_BLOCK
        mark all lines until matching CLOSE_BLOCK as chunk i
        i := i + 1
    endif
end foreach
foreach line of code in file f not marked as chunk
    add line to global
end foreach
foreach resulting chunk
        add global to chunk
end foreach
```

  Where OPEN_BLOCK and CLOSE_BLOCK are set according to the language; in C they are the curly brackets symbols, i.e. "{" and "}" respectively. A "matching" CLOSE_BLOCK is defined as a CLOSE_BLOCK which is on the same nesting level as its open block, in the usual programming-language way.

Next, the files were tokenized to different words; token limits were the standard punctuation and white spaces, but also language-specific symbols (such as `->`, marking reference to a pointed variable in C). An additional delimiter used at this stage was the underscore character, as is it commonly used in C to create compound words, such as `matrix_size`. We further address *compound splitting* issues in 5.1.

Finally, stopwords from the chunks were removed using a stop list developed for C documents, and with a simplified version of the Snowball [79] English stopword list.

The files were indexed using *FlexIR* [49], a vector-space model information indexing and retrieval system developed at the University of Amsterdam.

After preprocessing, the collection consisted of 5324 chunks and 145720 tokens (i.e. the average document size is 27.4 tokens). The decrease in the amount of tokens and average document size before preprocessing (326238 and 30.1 tokens respectively) comes both from the chunking process (which resulted in 1.8 chunks per file, on average) and from the stopping process which removed all language-specific keywords. It is interesting to note that the stopping process removed more than 55% of the text, while stopping natural language text results in a reduction of up to 40% [4] in the text size.

- *Retrieval.* The actual ranking of relevant documents was done in the Okapi BM25 variant of the probabilistic retrieval model. In the Okapi model [67], metrics from the vector-space approach such as term frequency (both in the document and in the query) and document length are used for estimating the probability that a document is relevant to a query. The similarity measure we use is Jacques Savoy's version of Okapi [72], i.e. Okapi weighting with "default" English parameters ($k_1 = 2$, $b = 0.8$) used for the documents, and *npn* weighting used for the queries. This approach was shown to have good *precision@n* for low n values, which is a desired feature in case the recall is difficult to assess.

  The Okapi weighting scheme for term j in document i is as follows:

$$okapi_{ij} = \frac{(k_1 + 1) \cdot tf_{ij}}{K + tf_{ij}}$$

  where:

  - tf_{ij} is the term frequency of term j in document i
  - $K = k_1 \cdot ((1 - b) + b \cdot \frac{|d_i|}{avgdoclen}$
  - *avgdoclen* is the average document length

The npn weighting scheme for the term j in the query is as follows:

$$npn_j = ntf_{qj} \cdot \log \frac{n - df_j}{df_j}$$

where:

- $ntf_{qj} = 0.5 + 0.5 \cdot \frac{tf_{qj}}{\max tf_q}$
- tf_{qj} is the term frequency of term j in the query
- $\max tf_q$ is the maximal term frequency of any term in the query
- n is the collection size
- df_j is the document frequency of term j in the collection

The similarity between a document and a query for a term j is then the product of the Okapi weights and the npn weights, and the total similarity of document i to the query q is the sum of the term similarities:

$$sim(q, d_i) = \sum_{j \in q} okapi_{ij} \cdot npn_{ij}$$

String-distance Measures

While the Okapi weighting scheme has performed very well for early high-precision [23], it can be argued that the probabilistic model is too weak to serve as a baseline for our experiments. Many typical differences between similar code files include tokens which have a small string distance between them — for this reason, string distance measures were incorporated into the SCG comparison. But any difference between two strings — small as it may be by a distance measure — renders them completely different to standard probabilistic models. Methods for indexing a document collection in a way that supports approximate queries exist [52]; however, these methods are not fully incorporated into probabilistic (or vector-space) models yet; they allow fast *search* of a large text collection for an approximated pattern rather than *retrieval* of a query, i.e. a set of tokens. In essence, these approximate indexing methods simply provide a faster way for approximate search of a pattern in a corpus, and do not provide the richness of a full retrieval model. It is not clear, for example, how to calculate term frequency measures, both for the query and for the document collection. Therefore, it seems that evaluating a standard retrieval model, be it vector-space or probabilistic, is indeed beneficial. Nevertheless, using string-distance measures also appears useful when evaluating the graph based code retrieval approach, to check how substantial the usage of string-distance measures is without the structural context. We have therefore implemented two separate string-distance retrieval models:

Simple string-distance retrieval In this model, all files were first preprocessed in an identical way to the Okapi baseline, i.e. chunked, tokenized and stopped. The same preprocessing was performed on the queries. The documents were then ranked according to their string-distance from the query; to measure the distance, we use the same Levenstein metric as the one used in the graph retrieval process, and end up with the following similarity between a preprocessed document d_i and a preprocessed query q:

$$sim(d,q) = \frac{|d_i| + |q| - distance(d_i, q)}{|d_i| + |q|}$$

Typed string-distance retrieval In this model, we associate an "information type" with each token of the code; the types we used were COMMENT, DEPENDENCY_STATEMENT, and OTHER, and were obtained by shallow parsing of the code. These categories were selected for a number of reasons: first, these are distinct texts that are usually important in program understanding and classification. Additionally, they do not require parsing, only pattern-matching, which is a compromise between full structural knowledge and no such knowledge. The last reason for choosing these categories was practical: since no parsing is involved, their extraction is "cheap". After the text classification, three separate strings were composed for every document, each one containing all tokens from the document (in the original order) of one of these types. The same process was carried out on the queries, so for each document and query we had three separate string representations. The comment string was stopped using an English stopword list, and the other strings were stopped using both English and C lists. The documents were then ranked according to the sum of three distance measures – the distances of the corresponding string representations. For a document d_i, we mark $c(d_i)$ as the described "comments string", and similarly $d(d_i)$ as the "dependency statements string" and $o(d_i)$ as the "others string", and get:

$$\begin{aligned} sim(q,d_i) &= \tfrac{|c(d_i)|+|c(q)|-distance(c(d_i),c(q))}{|c(d_i)|+|c(q)|} + \\ &+ \tfrac{|d(d_i)|+|d(q)|-distance(d(d_i),d(q))}{|d(d_i)|+|d(q)|} + \\ &+ \tfrac{|o(d_i)|+|o(q)|-distance(o(d_i),o(q))}{|o(d_i)|+|o(q)|} \end{aligned}$$

4.1.7 Implementation

The algorithms were implemented in mixture of Java and Perl; for the graph construction phase, we used the ANTLR [1] package, a compiler-compiler which reads grammar files and produces parsers for these grammars, similar to the well-known tool combinations `lex/yacc` or `flex/bison`. In our case, the grammar file

was a modified version of the GNU-C grammar from the ANTLR distribution, and the generated code was a Java program which was then used to build the graphs out of the C source code documents.

Perl was used for implementing the graph similarity measure, the baselines, and various text-processing tasks such as tokenization, string-distance etc. One of the more involved tasks, for example, was building the C preprocessor described in section 2.2.

4.1.8 Additional Computational Aspects and Filtering

As noted in section 3.2, the complexity of the similarity measure is high and the retrieval process is therefore quite resource consuming. While vector-space retrieval on a relatively small corpus such as the one used for the evaluation produces results almost instantly, in the graph retrieval approach the average query time was much higher. On a 1.5GHz Pentium processor, an average query took 5-6 minutes (the memory requirements were low and are therefore not discussed here). Although this was much faster than string-distance retrieval (30 minutes per query on the same machine), it renders the process inapplicable for "online" methods, where a user presents a query and gets an immediate response, more or less. It is possible to use it in an "offline" manner, i.e. for overnight sanity checks of a project code base, but even so the process is rather slow. There are two causes to this high computational cost. The first is the combinatorial explosion of the graph sizes, which causes the $O(n^3)$ complexity of the graph similarity measure to be high in practice. The second is the $O(n^2)$ complexity of the string-distance calculation, the "atomic operation" of the similarity measure. It is therefore desirable to have additional mechanisms to reduce the complexity of the entire retrieval process. One of the simplest mechanisms to do so is to limit the amount of documents which actually go through the rigorous comparison. There are various ways to implement such "filtering" mechanisms (see section 5.1), and we incorporated two of the simpler ones.

The first implemented mechanism was a simple size sanity check: it was decided not to compare two documents (or, more precisely, to consider them as having minimal similarity) if their size difference was larger than a threshold, which was set to 50%. For the document size, we used a standard string representation of the graph, which includes all structural and content information in it. This filtering mechanism decreased the average amount of graph comparisons by 14%, and was also used for the baseline models, to provide all models with the same settings.

The second limiting mechanism was an "essential concept types" filter: we defined the concepts types {FUNCTION, FUNC-CALL, LOOP, IF, COMPARE} as essential components of an SCG. We then added a pre-comparison stage for the similarity measure between G_1, G_2 in which graphs which had a mismatch in their essential concept types were considered minimally similar. In other words, if graph

G_1 contains an essential concept of type t, then graph G_2 must also include a concept of type t to be considered a candidate for comparison. The idea was that code that contains a loop somewhere is more likely to be highly similar to code that contains a loop too. This mechanism filtered out an additional 62% of the comparisons, bringing down the actual amount of comparisons performed on average to 24% of the original ones (see figure 4.1). The essential types filter was of course not used for the baselines, since it requires code-analysis schemes that are found only in the SCG retrieval method.

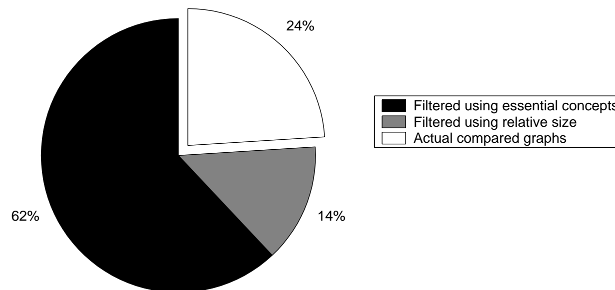

Figure 4.1: Effect of filtering mechanisms on amount of compared graphs

While it seems that such mechanisms are a practical step only, and that using them may decrease the accuracy of the retrieval, initial experimenting revealed that not only did they not harm the results (compared to the retrieval process without them), but even slightly improved them, on average; this suggests that these "prefetch" steps are beneficial on the theoretic side too, and not on the practical side only. For these reasons, the experiments reported were carried out using the described filtering mechanisms. It should be noted that while the filtering does not seem to degrade the precision, it probably reduces the recall (which was not measured); the "essential concepts" requirement should be relaxed and re-evaluated.

Even after these filtering steps, the retrieval takes longer than desired; the 25 queries included in the experiments, which include a mixture of small and large graphs, take more than 2 hours to complete. It should be noted, however, that the implementation of the algorithms was not optimized, and although it includes some performance boosting methods such as caching and adjustments of external packages to the specific task, much more can be done both on the theoretic and the practical levels to increase performance times: we discuss this further in section 5.1.

4.2 Results and Discussion

4.2.1 Testing for Optimized Parameters

In the first phase, we tested two of the parameters of the graph retrieval process: the comparison match depth and the Most Similar Concept (MSC) depth. As defined in section 3.2, the comparison match depth is the level of neighboring nodes from which the information is added to every node before comparison; depth 0 means no information is added. Similarly, the MSC depth is the comparison match depth for the initial phase, where a concept $c_2 \in G_2$ is selected to be compared to the concept $c_1 \in G_1$. We chose these parameters because of the large influence that we assumed they will have over the entire process: these are the main factors determining the amount of structure used for the comparison in our method. We predicted that using higher values – resulting in more structure – will improve results.

We tested both the Identical Document setting and the Modified Document one, with both match depth and MSC depth ranging between 0 and 2, for a total of 18 settings; the experiments were evaluated using MRR. Our aim was twofold: first, to explore the effect of incorporating additional structural (in particular, non-local) knowledge in the retrieval process; and second, to identify optimized values for later experiments, for which the assessments are more laborious than MRR (initial experiments proved that the MRR scores are consistent with the more involved precision scores). Our results are presented in tables 4.1 and 4.2.

SCG Retrieval

MSC \ Match	0	1	2
0	0.727	0.872	0.863
1	0.731	**0.905**	0.747
2	0.725	0.839	0.827

Table 4.1: MRR scores for different depth values, Identical Document Retrieval

A clear observation from these figures is that a value of 1 both for the MSC depth and the match depth yields the best results. Though this may seem counter-intuitive — after all, higher depth values mean including more meaningful information in the comparison — it seems that a balance between extending concepts with too much information and extending them with too little needs to be set for the retrieval to function optimally. Since we are comparing connected graphs, and since no relation weight is zero, allowing too much information to be added to a concept means that information that is only remotely related to it may be added to it, causing a topic drift. Such phenomena are also encountered in the stan-

SCG Retrieval

MSC \ Match	0	1	2
0	0.640	0.787	0.481
1	0.350	**0.813**	0.727
2	0.348	0.579	0.773

Table 4.2: MRR scores for different depth values, Modified Document Retrieval

dard, unstructured document retrieval models, when various techniques for query expansion are used (see, for example, the possible effects of blind feedback [68] and compound splitting [50]). Topic drift is known to be particularly substantial when the number of relevant documents is relatively small, which is the case in our corpus.

Furthermore, a careful examination of the results shows that with a few exceptions, setting the value of one of the depth measures to n achieves the best results if the other value is set to n as well. This reveals the underlying implementation of the MSC selection algorithm: since we select the MSC based on comparison to depth n, it seems reasonable that comparison of that depth level will also be used for the similarity. It is possible, however, that if we choose a different selection method for the MSC (say, based on the concept type, referent, and the *in_degree* or *out_degree* of the concept) we may have different results.

A surprising result is the relatively high MRRs for depth values (0,0). Setting the comparison depth to 0 yields a similarity measure that is similar to the typed string distance used as one of the baseline models: essentially, it boils down to doing a string comparison of all the text in the code according to a small number of categories. One could try to verify this experimentally, but we later show that (at least the MRR scores) are substantially better for graph comparison of depth 0 than for typed string distance, due to the more fine-grained categories achieved through the deeper parsing.

Based on these results, we fixed both the MSC depth and the match depth for the next experiments to 1.

As noted earlier, adding "too much" information to the comparison harmed the retrieval performance; a possible partial solution to this issue is optimizing the relation weights, the weights that determine how important the information in concept c_2 is to c_1, if the relation between them is r. However, this is a complex multi-parameter optimization problem, and since the document collection is not fully assessed for relevance with regards to each query, there is no "reward function" that we can assign to successful values without performing manual assessment. As mentioned in section 4.1.5, we assigned manual, "intuitive" weights to the relations and concepts; to measure the effect of using different weights, we

experimented with weights different from the manually assigned ones. We assigned a fixed value of 0.9 for all relations. In practice, this yields a simple scenario for the comparison: nodes which are further away are less important, regardless of the relations on the path from the compared node to them. As seen in table 4.3, the results were only slightly different (for worse) than the manual weights. Due to time constraints, we did not experiment further with optimization of the relation or concept weights.

SCG Weighting scheme	**Identical Document**		**Modified Document**	
	MRR	**P@5**	**MRR**	**P@5**
Manual	0.905	0.472	0.813	0.296
Fixed (=0.9)	0.898	0.464	0.811	0.291

Table 4.3: Different relation weighting schemes

4.2.2 Comparison with Baselines

For the rest of the experiments reported, we used match depth 1, MSC depth 1, and manual relation weights. We compared the graph retrieval method to the baseline models; the results are summarized in tables 4.4 and 4.5.

	Identical Document	**Modified Document**
Simple distance	0.973	0.093
Typed distance	1.000	0.293
Okapi	0.870	0.400
SCG Retrieval	0.905	0.813

Table 4.4: MRR comparison of retrieval methods

	Identical Document	**Modified Document**
Simple distance	0.400	0.056
Typed distance	0.424	0.128
Okapi	0.464	0.248
SCG Retrieval	0.472	0.296

Table 4.5: Average P@5 comparison of retrieval methods

The tables show a consistent increase of performance, with the exception of the MRR scores of the string distance methods in the Identical Document task; other-

wise, simple string-distance measures perform worst, and graph retrieval performs best.

These MRR exceptions for the string distance baselines are not surprising, as the Levenstein distance measure defines a string as having a distance of 0 to itself. The performance of the string distance measures on the modified retrieval drops sharply, since the modifications are exactly of the types that enlarge the string distance, i.e., changes in names of variables, bogus statements etc. The typed string distance, which brings in a bit of structural (or at least semantic) knowledge into the comparison, performs better than the simple distance throughout all measurements. Probabilistic retrieval does even better than the string distance measures (except the identical document MRRs); a reason for this improvement may be the usage of term frequency measures that reduce the importance of matches of meaningless strings such as repeated variable names (`i, j, count`) and common tokens in comments (`testcase, bug`). Finally, the graph retrieval that uses a combination of structural information and string distance yields the best results. The improvement of graph retrieval over the baseline is substantially better for the Identical Document task than the corresponding improvement in the Identical Document task.

Table 4.6 summarizes two additional measures we used for evaluation of the retrieval methods: the percentage of documents for which the exact match was found, and the average number of relevant documents (ignoring the exact match and counting only additional relevant documents). For both of these measures, only the top-5 retrieved documents were considered.

	Identical Document		**Modified Document**	
	% same document found	Average # of relevant documents	% same document found	Average # of relevant documents
Simple distance	100%	1.00	12%	0.16
Typed distance	100%	1.12	32%	0.32
Okapi	92%	1.40	52%	0.72
SCG Retrieval	92%	1.44	80%	0.68

Table 4.6: Additional measures comparison of retrieval methods

An examination of these values reveals that both Okapi and SCG retrieval yield an equivalent number of additional relevant documents; the higher precision values of the graph retrieval can therefore be attributed to the improved "same document" retrieval, rather than to finding more relevant documents. However, given that the probabilistic retrieval model is a mature one with many evaluations and improvements behind it, and that the graph comparison process is highly

unoptimized, it still seems that it improves over Okapi (and certainly over string-distance measures).

4.2.3 Significance Tests

As the scores of the string-distance measures were far below that of Okapi or the graph comparison, it seemed irrelevant to use statistical significance tests that involve them. We therefore concentrated on comparing Okapi and the graph retrieval; the null hypothesis is that the methods are equivalent, and we assume that if the test yields a probability smaller than 0.05, the results are statistically significant. The results of the sign-test are presented in table 4.7.

Significance measured	**Identical Document**	**Modified Document**
MRR	$n^{+} = 4, n^{-} = 3$ **$\mathbf{p \leqslant 1.0}$**	$n^{+} = 15, n^{-} = 2$ **$\mathbf{p \leqslant 0.0024}$**
P@5	$n^{+} = 10, n^{-} = 7$ **$\mathbf{p \leqslant 0.629}$**	$n^{+} = 12, n^{-} = 8$ **$\mathbf{p \leqslant 0.503}$**

Table 4.7: Sign Test, Okapi vs. SCG retrieval

As can clearly be seen, statistical significance was not established except for the MRR score in the Modified Document case. Although this does not necessarily mean that the null hypothesis must be accepted (i.e. that there is no significant difference between the methods), this seems discouraging. However, since the size of the query set was rather small, using statistical tests is hazardous. We therefore decided to perform an additional experiment, this time with a higher number of queries for the statistical evaluation. For the reasons mentioned earlier (high assessment costs), we decided to focus on the easy-to-calculate MRR score for identical document tests. To justify this decision, we recall that the MRR scores were consistent with the precision scores, i.e., higher MRR scores were always accompanied with higher precision score. Additionally, an examination of table 4.7 reveals that this was our worse statistical significance result, so improvement on its significance means with good likelihood that the other measures were improved too. However, this experiment is of limited importance, since it is only partially evaluated (using MRR) and therefore not comparable to the previous ones.

For this additional experiment we randomly selected 250 documents from the collection and used them as queries for an identical document retrieval, measuring the MRR score. We then repeated the sign test; the results appear in table 4.8.

For the larger set statistical significance was established. An examination of the actual MRR scores reveals that both methods performed worse than on the original 25-query identical document experiment (although Okapi's degradation was more notable). This could hint that our original query set, although randomly chosen,

Value	**Identical Document, 250-queries**
Okapi MRR	0.684
SCG MRR	0.813
Sign Test	$n^{+} = 63, n^{-} = 28$ $\mathbf{p \leqslant 0.000313}$

Table 4.8: Results for 250-query Identical Document Retrieval

had properties which make it easier to retrieve - for example, longer average document length. In general, it stresses the importance of additional experimenting and cross validation of all results presented. Once again, the results of the 250-query experiment should be given lower importance since the measurements for it (MRR only) do not match those of other experiments.

4.2.4 Combining Methods

As stated in the previous section, for our last set of experiments we combined scores from our two most successful methods - the Okapi baseline and the SCG retrieval. Our combination method is identical to [38]: first, all relevance scores are normalized to values between 0.5 and 1. The scores from the two methods are then combined linearly:

$$S_{comb} = \lambda \cdot S_1 + (1 - \lambda) \cdot S_2$$

where

- S_i is the normalized score of method i
- λ is the linear combination factor

Method	**Identical Document**		**Modified Document**	
	MRR	**P@5**	**MRR**	**P@5**
Okapi	0.870	0.400	0.464	0.248
SCG	0.905	0.464	0.813	0.296
Combined ($\lambda = 0.2$)	1.000	0.504	0.630	0.272
Combined ($\lambda = 0.5$)	0.960	0.488	0.840	0.352
Combined ($\lambda = 0.8$)	0.960	0.448	0.833	0.336

Table 4.9: Combining Okapi and Graph-retrieval

The results of the combined methods are presented in table 4.9; the first two lines in the table are repetition of previous results and are displayed for convenience, acting as a reference to the two methods that are combined. In our

experiments, S_1 are scores of the graph retrieval and S_2 of Okapi, so lower values of λ mean a lower importance of the SCG method in the combination.

As can be seen, any combination improves the Okapi scores, and with the exception of the Modified Document task for $\lambda = 0.2$, all combinations improve also over the SCG performance. For the Identical Document task, it seems that lower importance attributed to the graph retrieval yields a better combination: the results for the Modified Document task are mixed and require additional tests to draw conclusions, but indicate that the combined scores are better when the graph retrieval has a substantial weight.

Chapter 5

Conclusions

The initial results of our experiments are encouraging: although little was done in terms of optimization and tuning, the proposed retrieval method outperformed well-established retrieval models for the specific task we tested. Our goal was to prove that exploiting structure yields significant improvements in the case of code retrieval, and that conceptual graphs are a good tool for this task; our results consistently support this. It should be noted, however, that the amount of evaluation done is far from the amount required to support firm conclusions, and we treat our experiments as a proof-of-concept rather than a complete assessment of the method. More work needs to be done both in terms of developing the method and evaluating it; see the next section for more details.

Other than this main conclusion, there are other observations that can be made from our experiment:

- Using a retrieval model which is aimed at standard unstructured text (Okapi, in our case) for a structured text task such as code retrieval yielded relatively good results. An analysis of queries where the Okapi model had good performance, and especially queries where it outperformed the graph retrieval, shows the expected explanation: these were usually documents which contained a relative large amount of free (unstructured) text such as comments and strings which are printed by the program. It was also common to find a token with a very low frequency (some rare function name, for example) in these documents; these are classical cases where probabilistic methods provide good results. However, overall the Okapi performance seemed surprisingly good.

- As noted in section 4.2.1, the depth of the comparison had a very substantial effect on the results. While this in itself is expected, the peak of the performance at a relatively low depth (1) indicates that too much contextual knowledge reduces the effectiveness of the specific similarity measure;

the best performance is achieved via a "shallow parsing" approach, where some contextual knowledge is ignored. Another observation, based on the fact that using depth 0 was also fairly successful, is that the clustering of tokens according to their semantic type (Func-call, Comment, etc) is more important than the actual additional contextual information.

- An additional result presented in section 4.2.1 was that using a different weighting scheme for the concepts and relations only has a minor effect. A possible conclusion from this is that the important knowledge in the graphs is the actual existence of relations rather than their type. It seems that "what is related to what" — which concepts have some kind of relation between them — is more crucial than "how it is related". This conclusion should be taken with a grain of salt, as it is drawn from a single comparison to a different weighting scheme and not from real optimization.
- One of the practical implications of the experiments is the demonstration of the high computational costs of this type of structural retrieval. We conclude that in the present form the method can not be used for online retrievals and is simply incomparable to vector-space or probabilistic methods in terms of complexity. A more detailed analysis of this point, along with solutions, is provided in the next section.

5.1 Open Issues and Future Work

We now give an overview of the areas on which further research should focus, and discuss possible directions to explore and solutions to problems we encountered.

Complexity. We mentioned earlier (section 4.1.8) the relatively high costs of the retrieval method, compared to other methods. Generally, there are two approaches to handle this issue:

- *Reducing the number of comparisons.* Naive ways of filtering graphs before they are compared were discussed previously; these methods use shallow information about the graphs such as their size and the types of concepts they contain. In our experiments, using these shallow measures to avoid a large part of the comparisons proved beneficial. However, their evaluation was limited and needs to be examined more carefully, both for accuracy and for optimal parameters (such as the best set of "essential concept types"). In addition to the mechanisms mentioned, it is also possible to integrate other filtering methods into the retrieval process.

  The first course of action that comes to mind for this task is *indexing*, a preprocessing stage which stores the information in the collection in a way that

makes access to it faster. Such indexing can be done according to concept types, content in the graph concepts or relations, and even purely structural information such as *out_degrees*. An indexing method for conceptual graphs was proposed in [58], although it is not fully automatic; another indexing method, this time automatic, is presented in [59]. Although this method is designed for a different similarity measure than the one we use, ideas from it can be adopted and used.

In section 4.2.4 we explored a combination of the graph retrieval method with another one. This combination, although resulting in improved scores, does not change the complexity of the retrieval, since it required running both methods on the entire document collection. However, it is possible to use a combination method that also reduces the complexity: for example, we can use another model (a significantly faster one, such as vector-space) as a "prefetch" stage, and promote only the better-ranking documents (say, the upper 50%) to the more complex graph matching.

- *Reducing the complexity of a single comparison.* A good candidate for boosting the comparison time of two graphs is pre-calculation of the extended concepts. Since the extension process of a graph is independent of the graph it is compared to, it is useful to calculate in advance the extended concepts and reuse them, rather than create them for every query presented. In our experiments, we used caching techniques that eliminated some of the redundant calculations; however, offline calculation of the extended concepts would have saved even more resources and was not applied simply because it occurred to us in a late stage of the evaluation.

  Since the cost of the comparison is polynomial in the graph size, a different way of looking at reducing the complexity of it is limiting the size of the graphs that are compared. In the spirit of passage information retrieval, a simple chunking technique was used in our experiments and described earlier; this technique structured a document into passages according to shallow parsing of it which identified language-related segments. More involved methods can be used for creating these passages; for example, the *program slice* [82] notion seems appropriate for this task. It is possible that shallow overlapping window methods are also fit for this task, but less likely.

Optimization. The proposed methods for graph construction and graph comparison include many internal parameters; limited experimenting with different values of some of these was performed in section 4.2.1. However, usage of arbitrary values based on limited testing and intuition is clearly a weak point that should be addressed. A survey of successful IR techniques reveals that usage of different parameters (k_1 and b for Okapi, different slope values for Lnu.ltc vector-space retrieval and so on) yields highly different results; these values are sometimes

counter-intuitive, or at least do not seem to have any consistent line of reasoning behind them, and are based purely on evaluation. In the SCG retrieval setting, among the parameters requiring tuning are:

- Concept and relation weights.
- Thresholds for ignoring the comparison (see section 4.1.7).
- The element in the set of "Essential concept types" (see also section 4.1.7).
- Concept extension depth.
- Concept extension parameters; the depth of the extension need not be necessarily the same for incoming relations and outgoing ones (in the experiments reported they were equal). It is also possible to use different weights for incoming and outgoing relations.
- Normalization factors, such as the measure of the graph size which is used to normalize the similarity. Additionally, normalization factors can be used in the concept extension phase; for example, we may use the *out_degree* of a concept to normalize the weights of concepts being added to it, and reduce their relative effect.

It should be noted that finding optimized values depends largely on the specific document collection and query sets; tuning a value or method should always be closely linked to the collection at hand, since it is assumed that different collections will result in different optimized values. Regardless of this comment, the large amount of free parameters is clearly a disadvantage of the SCG retrieval method – although it leaves a lot of room for modifications and possible improvements.

Algorithmic Enhancements. Of course, it is always possible to make modifications on the algorithm itself; this set of changes is perhaps the most complex, and certainly the most interesting. A short list of potentially promising modifications include:

- *Adopt ideas and methods from other retrieval models.* For example, the *term frequency* notion that is a central in vector-space methods (and plays a role in other models such as Okapi as well) can be used to assign variable weights to the concept referents, thus increasing the importance of those that contain rare terms. Other measures that can be useful are the *average document length* and the *inverse document frequency.*

- *Incorporation of more structural knowledge.* Currently, much of the structural information about the program is still lost, by converting actual structure to simple weighting schemes. Additional measures that take into account explicit structural data such as the ones proposed in [59] and [92] may prove beneficial; however, usage of more structural knowledge should be done carefully and in a way that permits "fuzziness" of the structure, in a similar way to the proposed similarity measure.

- *Change basic similarity measure.* The currently used "atomic" information measure is a string-distance one. But the parsing process of the code provides us with tools that can be used to use different measures for different types of text. For example, literals which are explicit numbers can be simply compared for their numerical difference; comments which are usually natural language texts (although sometimes they are "commented-out" code) can be compared using vector-space methods, i.e. dot product of their representations as term vectors, and so on.

- *Stemming.* Stemming provides retrieval systems with a mechanism of generalization over morphological variations; evidence regarding the effect of stemming on IR methods is mixed [35]. Integrating stemming into code retrieval is not trivial, since the text is composed of two separate languages – the tokens of the programming language, and the natural language text embedded in it (in the comments, sometimes in function names, in "print" statements, and so on). A first step could be evaluating the effect of stemming the natural language only; in a later stage, a stemmer can be defined for the programming language too. For example, it seems that removing suffix numbers in token names may be beneficial (`count1` → `count`). However, it should also be noted that the string-distance measure used as the atomic operation of the retrieval is more immune to morphological changes than term-based methods, so the contribution of stemming may be insignificant (unless the atomic operation is modified).

- *Representation.* To a large extent, the representation method for the graphs was simplified; in theory, it encapsulates all forms of knowledge present in the source code, but for various practical reasons the implemented taxonomy was limited; the concept and relation types within them can actually differ substantially. For example, IF statements typically have a CONDITION relation to an expression and one or more CONTAINS relations to statements; knowledge about which statement is carried out when the condition is true and which one is executed when it is false is lost. Additionally, some code components are not converted to a concept at all, i.e. preprocessor directives which are not `#define`s or `#include`s. A more fine-grained taxonomy can be defined and used for the graph construction process, although as noted earlier "too much structure" can also prove as a disadvantage.

- *MSC location method.* Our method for locating the Most Similar Concept $c_2 \in G_2$ so it can be compared to concept $c_1 \in G_1$ is based on the same similarity measure used later for the comparison. This method can be substituted with a different one without changing the rest of the process; it is possible to use shallow measures such as degree of edges, or more involved structural information such as graph edit distance [94] or partial isomorphisms [69].

- *Compound splitting.* This technique, which disassembles tokens to the estimated subtokens from which they were compounded, proved effective for retrieval from languages which are morphologically rich [49]. Programming languages seem natural candidates for compound splitting; a glance at almost any source code reveals many compounded terms such as `CCPUUsage`, `resolve_unique_section`, or `byteDataPacket`. A simple method for splitting was incorporated both into the baselines and into the graph retrieval, as indicated in section 4.1; it is possible to use more involved compound splitting techniques, either rule-based (using capitalization of letters in the term) or based on other mechanisms. It should be noted again that similarly to the stemming case, the string-distance measures are relatively immune to compound words, and therefore the improvements are not expected to be high.

- *Combination Methods.* We explored a basic combination method for the SCG retrieval and Okapi in section 4.2.4; more mixture-of-experts methods for retrieval methods, such as the ones described in [6], can be applied to further improve results. Furthermore, our combination experiments show that the optimized combination of two methods is dependent on the retrieval task: for the Identical Document task much less "structure information" was required, compared to the Modified Document task. This suggests that an analysis of the retrieval settings – such as the query and the document collection – may be helpful in determining which combination to use, before the retrieval is actually performed.

Evaluation. Finally, it should be noted that our experiments were done using a relatively small corpus; the reasons for selecting such a corpus were largely practical, since no evaluated corpora for the task is publicly available and the resources for manual evaluation were limited. It is of course essential to extend the evaluation to a larger corpus, and to test other corpora types, viz., large scale source code projects. Extensive document collections and longer query lists will also render the results of the experiments more statistically well-founded (or refute them). For example, in section 4.2.3, our statistical significance tests yielded different results for different amounts of queries; additional test should be performed to determine the actual values.

Assuming the development of such collections, it will also be useful to measure recall levels for the retrieval and compare precision-recall curves to other methods; from this we can learn more thoroughly were the graph comparison is favorable and how it can be used.

Summary

We state again the goals we have set for ourself in the beginning of this thesis. In the vast area of information retrieval from source code, we intended to explore one corner: the usage of the knowledge embedded in the structure of the code to improve the retrieval. Our aim was to propose a method for extracting and using this “structural knowledge”, to evaluate it and to analyze the associated complexity implications.

Our proposed method uses conceptual graphs to model the code and build a retrieval model around these graphs, including a specialized similarity measure for them. Although the method is not complete, experiments show that it performs substantially better than state-of-the-art retrieval methods for the specified task. Following the evaluation, we analyze the weak points of the method, including the higher complexity bounds (compared to other retrieval models), and propose procedures for enhancing it in various ways to address these weaknesses. Since our approach is a relatively unexplored one, there is much more to be done before it can be considered a proper rival for other approached to source code retrieval; our initial results appear promising and suggest that further work can improve the method significantly.

Bibliography

[1] ANTLR: ANother Tool for Language Recognition, URL: `http://www.antlr.org`.

[2] F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, and P.F. Patel-Schneider, editors. *The Description Logic Handlbook*, chapter 1,10. Cambridge University Press, 2002.

[3] Greg J. Badros. JavaML: a markup language for Java source code. *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(1–6):159–177, 2000.

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

[5] B.S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In L. Wills, P. Newcomb, and E. Chikofsky, editors, *Second Working Conference on Reverse Engineering*, pages 86–95, Los Alamitos, California, 1995. IEEE Computer Society Press.

[6] Brian T. Bartell, Garrison W. Cottrell, and Richard K. Belew. Automatic Combination of Multiple Ranked Retrieval Systems. In *Research and Development in Information Retrieval*, pages 173–181, 1994.

[7] B.W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[8] M. Buchheit, M.A. Jeusfeld, W. Nutt, and M. Staudt. Subsumption between queries to object-oriented databases. In *Proceedings of the 4th international conference on extending database technology on Advances in database technology*, pages 15–22. Springer-Verlag New York, Inc., 1994.

[9] H. Bunke and B.T. Messmer. Similarity measures for structured representations. In *Procs. of the First European Workshop on Case-Based Reasoning*. Springer, 1993.

[10] M. Chein and M.-L. Mugnier. Conceptual graphs: Fundamental notions. *Revue d'intelligence artificielle*, 6(4):365–406, 1992.

[11] S. Chu and B. Cesnik. Knowledge representation and retrieval using conceptual graphs and free text document self-organisation techniques. *International Journal of Medical Informatics*, 62:121–133, July 2001.

[12] C. Clarke, A. Cox, and S. Sim. Searching program source code with a structured text retrieval system. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 307–308. ACM Press, 1999.

[13] Richard Clayton, Spencer Rugaber, and Linda M. Wills. On the knowledge required to understand a program. In *Working Conference on Reverse Engineering*, pages 69–78, 1998.

[14] Thomas H. Cormen, E. Leiserson, Charles, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990. COR t 01:1 1.Ex.

[15] F. Crestani, I. Ruthven, Mark Sanderson, and Cornelis J. van Rijsbergen. The troubles with using a logical model of IR on a large collection of documents. In *Proceedings of TREC-4, Fourth Text Retrieval Conference*, Washington, US, 1995.

[16] B. Croft and J. Lafferty, editors. *Language Modeling for Information Retrieval*. Kluwer Academic Publishers, 2003.

[17] Y. S. M. Cutler and W. Meng. Using the structure of html documents to improve retrieval. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[18] Brown A.W. Dart S., Christie A.M. A case study in software maintenance. Technical report, Software Engineering Institute, Carnegie Mellon University, 1993.

[19] H.S. Delugach and G. Stumme, editors. *Conceptual Structures: Broadening the Base, 9th International Conference on Conceptual Structures, ICCS 2001, Stanford, CA, USA, July 30-August 3, 2001, Proceedings*, volume 2120 of *Lecture Notes in Computer Science*. Springer, 2001.

[20] P.T. Devanbu, R.J. Brachman, P.G. Selfridge, and B.W. Ballard. Lassie: a knowledge-based software information system. In *International Conference on Software Engineering*, pages 249–261, 1990.

[21] J.P. Dick. Representation of legal text for conceptual retrieval. In *Proceedings of the third international conference on Artificial intelligence and law*, pages 244–253, 1991.

[22] R. Dieng. Comparison of conceptual graphs for modelling knowledge of multiple experts. In *International Syposium on Methodologies for Intelligent Systems*, pages 78–87, 1996.

[23] V. Jijkoun et al. The university of amsterdam at trec 2003. In *TREC 2003 Working Notes*. National Institute for Standards and Technology, 2003.

[24] A.E. Fischer and F.S. Grodzinsky. *The Anatomy of Programming Languages*. Pretience Hall, 1993.

[25] N. Foo, B. Garner, A. Rao, and E. Tsui. Semantic distance in conceptual graphs, 1992.

[26] S. Fortin. The graph isomorphism problem. Technical report, Uni. of Alberta, Canada, 1996.

[27] Bernhard Ganter and Guy W. Mineau, editors. *Conceptual Structures: Logical, Linguistic, and Computational Issues, 8th International Conference on Conceptual Structures, ICCS 2000, Darmstadt, Germany, August 14-18, 2000, Proceedings*, volume 1867 of *Lecture Notes in Computer Science*. Springer, 2000.

[28] The GNU Compiler Collection, URL: `http://gcc.gnu.org`.

[29] M. Ghosh, B. Verma, and A. Nguyen. An automatic assessment marking and plagiarism detection. In *International Conference on Information Technology and Applications*, 2002.

[30] M.R. Girardi and B. Ibrahim. Using English to retrieve software. *The Journal of Systems and Software*, 30(3):249–270, September 1995.

[31] Olivier Guinaldo. Conceptual graphs isomorphism: Algorithm and use. In *International Conference on Conceptual Structures*, pages 160–174, 1996.

[32] Djoerd Hiemstra and Arjen de Vries. Relating the new language models of information retrieval to the traditional retrieval models. Technical Report TR-CTIT-00-09, Centre for Telematics and Information Technology, 2000.

[33] T.W.C. Huibers, I. Ounis, and J-P. Chevallet. Conceptual graph aboutness. In P.W. Eklund, G. Ellis, and G. Mann, editors, *Conceptual Structures: Knowledge Representation as Interlingua, Proceedings of the Fourth International Conference on Conceptual Structures, ICCS'96*, volume 1115, pages 130–144, Sydney, Australia, 1996. Springer-Verlag.

[34] D. Hull. Using Statistical Testing in the Evaluation of Retrieval Experiments. In *Proceedings of the 16th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 329–338. ACM Press, 1993.

[35] D. Hull and G. Grefenstette. A detailed analysis of english stemming algorithms. Technical report, Rank XEROX, 1996.

[36] J.J. Jeng and B.H.C. Cheng. Using formal methods to construct a software component library. In Ian Sommerville and Manfred Paul, editors, *Proceedings of the Fourth European Software Engineering Conference*, pages 397–417. Springer-Verlag, 1993.

[37] E.L. Jones. Metrics based plagarism monitoring. In *Proceedings of the sixth annual CCSC northeastern conference on The journal of computing in small colleges*, pages 253–261. The Consortium for Computing in Small Colleges, 2001.

[38] J. Kamps, C. Monz, and M. de Rijke. Combining evidence for cross-lingual information retrieval. In C. Peters, M. Braschler, J. Gonzalo, and M. Kluck, editors, *Proceedings CLEF 2002*. Springer, 2002.

[39] The KDE Desktop Environment, URL: `http://www.kde.org`.

[40] M. Lalmas and I. Ruthven. A model for structured document retrieval: Empirical investigations. In *HIM*, pages 53–66, 1997.

[41] Y.S. Maarek and D.M. Berry et al. Guru: Information retrieval for reuse. Landmark Contributions in Software Reuse and Reverse Engineering.

[42] J. Maletic, M. Collard, and A. Marcus. Source code files as structured documents, 2002.

[43] P. Martin and L. Alpay. Conceptual structures and structured documents. In *International Conference on Conceptual Structures*, pages 145–159, 1996.

[44] N. Medvidovic, D. S. Rosenblum, J. E. Robbins, and D. F. Redmiles. Modeling software architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering Methodologies*, 11(1):2–57, 2002.

[45] C. Meghini, F. Sebastiani, U. Straccia, and C. Thanos. A model of information retrieval based on a terminological logic. In *Proceedings of the 16th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 298–307. ACM Press, 1993.

[46] H. Meuss and C. Strohmaier. Improving index structures for structured document retrieval, 1999.

[47] P. Klint M.G.J. van den Brand and C. Verhoef. Reverse Engineering and System Renovation: an Annotated Bibliography. *ACM Software Engineering Notes*, 22(1):42–57, January 1997.

[48] K. Mills. Requirements engineering for software reuse, 1992.

[49] C. Monz and M. de Rijke. Shallow morphological analysis in monolingual information retrieval for Dutch, German and Italian. In C. Peters, M. Braschler, J. Gonzalo, and M. Kluck, editors, *Evaluation of Cross-Language Information Retrieval Systems, CLEF 2001*, volume 2406 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2002.

[50] C. Monz and M. de Rijke. The University of Amsterdam at CLEF 2001. In *Working Notes CLEF 2001*, 2001.

[51] S.H. Myaeng, D.H. Jang, M.S. Kim, and Z.C. Zhoo. A flexible model for retrieval of sgml documents. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 138–145. ACM Press, 1998.

[52] G. Navarro, R. A. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001.

[53] M. L. Nelson. A Survey of Reverse Engineering and Program Comprehension.

[54] F. Niessink and H. van Vliet. Two Case Studies in Measuring Software Maintenance Effort. In *International Conference on Software Maintenance*, pages 76–85, Bethesda, Maryland, USA, November 16-20, 1998. IEEE Computer Society.

[55] T. Noreault, M. McGill, and M.B. Koll. A performance evaluation of similarity measures, document term weighting schemes and representations in a boolean environment. In *Proceedings of the 3rd annual ACM conference on Research and development in information retrieval*, 1980.

[56] The OpenOffice.org Project, URL: `http://www.openoffice.org`.

[57] I. Ounis and J. Chevallet. Using Conceptual Graphs in a Multifaceted Logical Model for Information Retrieval. In *Database and Expert Systems Applications*, pages 812–823, 1996.

[58] I. Ounis and M. Pasca. Modeling, Indexing and Retrieving Images using Conceptual Graphs. In *Proceedings of 9th International Conference on Database and Expert Systems Applications (DEXA'98)*. Springer, 1998.

[59] I. Ounis and M. Pasca. Organizing Conceptual Graphs for Fast Knowledge Retrieval. In *Proceedings of 10th International Conference on Tools with Artificial Intelligence (ICTAI'98)*. IEEE Computer Society Press, 1998.

[60] S. Paul and A. Prakash. Querying source code using an algebraic query language. In Hausi A. Müller and Mari Georges, editors, *Proceedings of the International Conference on Software Maintenance (ICSM '94)*, pages 127–136, 1994.

[61] PMD: Project Mess Detector, URL: `http://pmd.sourceforge.net`.

[62] J. Poole and J.A. Campbell. A novel algorithm for matching conceptual and related graphs. In *International Conference on Conceptual Structures*, pages 293–307, 1995.

[63] L. Prechelt, G. Malpohl, and M. Philippsen. JPlag: Finding plagiarisms among a set of programs. Technical Report No. 1/00, University of Karlsruhe, Department of Informatics, March 2000.

[64] R. Prieto-Daz. Implementing faceted classification for software reuse. *Commun. ACM*, 34(5):88–97, 1991.

[65] Y. Quintana, M. Kamel, and A. Lo. Graph-based retrieval of information in hypertext systems. In *Proceedings of the 10th annual international conference on Systems documentation*, pages 157–168. ACM Press, 1992.

[66] URL: Rational Software, `http://www.rational.com`.

[67] S.E. Robertson, S. Walker, and M. Beaulieu, editors. *Okapi at TREC-7: Automatic ad hoc, filtering, VLC and interactive track*. National Institute for Standards and Technology, 1997.

[68] I. Ruthven and M. Lalmas. A survey on the use of relevance feedback for information access systems. *Knowledge Engineering Review*, To appear.

[69] S., J. Malik, and J. Puzicha. Matching Shapes. In *Proceedings of The Eighth IEEE International Conference on Computer Vision*, pages 454–463, 2001.

[70] G. Salton and M.J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1986.

[71] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Inf. Process. Manage.*, 24(5):513–523, 1988.

[72] J. Savoy. Report on CLEF-2002 Experiements: Combining multiple sources of evidence. In C. Peters, Braschler, M., Gonzalo, J., Kluck, M., editor, *Results of the CLEF-2002, cross-language evaluation forum*, 2002.

[73] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local Algorithms for Document Fingerprinting. In *SIGMOD2003*, pages 76–85, 2003.

[74] SCO vs. IBM, URL: `http://www.sco.com/ibmlawsuit/amendedcomplaintjune16.html`.

[75] The open-source movement's position on SCO vs. IBM, URL: `http://www.opensource.org/sco-vs-ibm.html`.

[76] R. Sethi. *Programming Languages: Concepts and Constructs (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[77] Judy Sheard, Martin Dick, Selby Markham, Ian Macdonald, and Meaghan Walsh. Cheating and plagiarism: perceptions and practices of first year it students. In *Proceedings of the 7th annual conference on Innovation and technology in computer science education*, pages 183–187. ACM Press, 2002.

[78] Simian: Similarity Analyser, URL: `http://simian.dev.java.net/`.

[79] Snowball: A language for stemming, URL: `http://snowball.tartarus.org`.

[80] J.F. Sowa. *Conceptual structures: information processing in mind and machine*. Addison-Wesley Longman Publishing Co., Inc., 1984.

[81] William M. Tepfenhart and Walling R. Cyre, editors. *Conceptual Structures: Standards and Practices, 7th International Conference on Conceptual Structures, ICCS '99, Blacksburg, Virginia, USA, July 12-15, 1999, Proceedings*, volume 1640 of *Lecture Notes in Computer Science*. Springer, 1999.

[82] F. Tip. A Survey of Program Slicing Techniques. *Journal of programming languages*, 3:121–189, 1995.

[83] URL: The Unified Modeling Language, `http://www.uml.org`.

[84] K.L. Verco and M.J. Wise. Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. In J. Rosenberg, editor, *Proceedings of the First Australian Conference on Computer Science Education*. ACM, 1996.

[85] A. von Mayrhauser and A.M. Vans. Comprehension processes during large scale maintenance. In *Proceedings of the 16th International Conference on Software Engineering* (Sorrento, Italy; May 16-21, 1994), pages 39–48. IEEE Computer Society Press, 1994.

[86] E.M. Voorhees and C. Buckley. The effect of topic set size on retrieval experiment error. In *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 316–323. ACM Press, 2002.

[87] E.M. Voorhees and D.K. Harman, editors. *The Eleventh Text REtrieval Conference (TREC 2002)*, chapter Appendix: Common Evaluation Measures. National Institute for Standards and Technology, 2003.

[88] N.R. Wagner. Plagiarism by Student Programmers. http://www.cs.utsa.edu/ wagner/pubs/plagiarism0.html.

[89] The WebKB set of tools: a common scheme for shared WWW Annotations, shared knowledge bases and information retrieval, URL: `http://www.webkb.org`.

[90] R. Wilkinson. Effective retrieval of structured documents. In *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 311–317. Springer-Verlag New York, Inc., 1994.

[91] M.J. Wise. Yap3: improved detection of similarities in computer program and other texts. In *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, pages 130–134. ACM Press, 1996.

[92] M. Montes y Gomez, A. Lopez, and A. F. Gelbukh. Information Retrieval with Conceptual Graph Matching. In *Database and Expert Systems Applications*, pages 312–321, 2000.

[93] G.C. Yang and J. Oh. Knowledge acquisition and retrieval based on conceptual graphs. In *Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing*, pages 476–481. ACM Press, 1993.

[94] K. Zhang, J.T.L. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs and related problems. In Z. Galil and E. Ukkonen, editors, *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, pages 395–407, Espoo, Finland, 1995. Springer-Verlag.