Source Code Retrieval using Conceptual Similarity

Gilad Mishne and Maarten de Rijke
Language & Inference Technology Group
University of Amsterdam
{gilad,mdr}@science.uva.nl

Abstract

We propose a method for retrieving segments of source code from a large repository. The method is based on conceptual modeling of the code, combining information extracted from the structure of the code and standard information-distance measures. Our results show an improvement over traditional retrieval models, indicating that, for this type of highly-structured documents, usage of structure is indeed beneficial for retrieval.

1 Introduction

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 (von Mayrhauser and Vans, 1994). Code retrieval encompasses many subtasks; some are high-level ones, such as design recovery and reverse-engineering. Other tasks require a lower-level analysis of the code: two such tasks are *Code Duplication Prevention* and *Plagiarism Detection*. Duplicated code accounts for up to 20% of the total amount of code in large software systems (Baker, 1995), and is a well-known software engineering problem. Plagiarism of computer programs is common mainly in low-level programming courses (Sheard et al., 2002), but can also be found in commercial, larger-scale scenarios (SCO vs. IBM). For both tasks, a method which retrieves *similar* (possibly duplicated, plagiarised, or otherwise related) code is beneficial.

For various reasons, source code retrieval is a challenging task: 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. We present an approach to retrieving source code that combines both structural information and the content of the code into a single representation. Our main contribution is a notion of source code similarity that is based on a representation of source code as *conceptual graphs* (CGs), a knowledge representation formalism proposed by Sowa (1984). This choice of representation allows us to combine information-theoretic ideas aimed at capturing *content* aspects of source code with techniques for manipulating graphs that are aimed at capturing the *structure* of source code. Our main experimental finding is that both structure and content are important for source code retrieval; our best results are obtained by combining a moderate amount of structural information with the content of the source code documents.

The rest of the paper is organized as follows. We survey current approaches and tools for code retrieval in Section 2. In Section 3 we describe our method for extraction and representation of source code concepts from the code; then, in Section 4, we use this representation to construct a retrieval model for source code. Section 5 presents the experiments conducted to evaluate our approach and their results; our conclusions and ongoing work are summarized in Section 6.

2 Related Work

Related work comes in several kinds, concerning source code retrieval, code similarity analysis, and retrieval using conceptual graphs.

2.1 Source Code Retrieval

With the arrival and growing availability of HTML and XML documents, recent years have witnessed an increased interest in structured document retrieval; (see e.g., Cutler and Meng, 1997; INEX). Source code provides an example of strucured documents whose retrieval has been researched at least since the mid-late 1980s (Frakes and Nejmeh, 1987). Early forms of code retrieval were based on a classification scheme for cataloging code components with a set of keywords; (see e.g., Prieto-Daz, 1991). Such methods yield good results, but the manual effort required for them is very high, mostly for the classification but also for the retrieval (which requires knowledge about the legal, relevant keywords).

Formal approaches such as (Jeng and Cheng, 1993; Paul and Prakash, 1994) 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 (Clarke et al., 1999); it is more standardized but shares the same advantages and disadvantages of other formal methods.

Tools such as GURU (Maarek et al., 1994) and ROSA (Girardi and Ibrahim, 1995) 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.

Conceptual modeling and retrieval of code has been implemented in systems such as LaSSIE (Devanbu et al., 1990); 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 high-level architectural queries rather than low-level code matching. Additionally, these tools require hand-crafting separate knowledge bases for every software project.

2.2 Code Similarity Analyzers

Tools for locating similar code, for duplication or plagiarism detection, can be grouped as follows:

Pattern-based Analyzers. They check for shallow similarity between lines of codes, using pattern matching techniques and tiling algorithms, (see PMD; Simian). This approach is very effective 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 (Ghosh et al., 2002; Jones, 2001; Schleimer et al., 2003) associates a "code signature" with every piece of code, calculated by examining certain features of the code; programs with similar signatures are considered to be similar. Since this approach relies on statistical properties of the code, it is effective mainly for larger-scale segments of code, rather than detection of short repeating sections of code.

Structural Analyzers. Analyzers of this type (Prechelt et al., 2000; Wise, 1996) compare structural properties of the programs by representing the programs as strings and measuring the string distance between them. This approach is highly effective (Verco and Wise, 1996), but since it ignores information such as comments, dependency files etc., it may fail to locate code that is not highly similar in structure, but similar in "spirit," i.e., addresses the same issue.

2.3 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 the use of CGs for plain document retrieval (Montes-y-Gomez et al., 2000; Ounis and Chevallet, 1996; Quintana et al., 1992), 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 (Dick, 1991), medical information (Chu and Cesnik, 2001), and multimedia documents (Ounis and Pasca, 1998; Yang and Oh, 1993). 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 (Martin and Alpay, 1996; WebKB), however, this work relies on manual annotation and a WordNet-like extensive ontology.

3 Representing Source Code

As mentioned earlier, we use conceptual graphs to model source code. In a nutshell, 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. All nodes have an associated *type*; additionally, concept nodes have a *referent* value, which contains specific information regarding this concept. A conceptual graph is always related to a *support*, a knowledge base providing background on the domain within which the graph is presented. Examples of simple conceptual graphs are given in Figure 1.

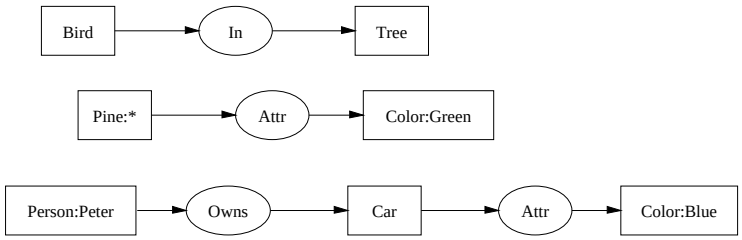

Figure 1: Examples of conceptual graphs

We now describe the support of the source code conceptual graphs *(SCGs)*, and then a procedure to construct them from the code.

3.1 A Taxonomy for Source Code

The concept types we allow in our graphs are presented in Table 1.

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 1: Source Code Concept Types

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.

For space reasons, we only list the possible relations types, and not a specification of which concepts they can connect and the semantics of such connectivity; however, the semantics are mostly self-evident from the relation names. The relation types, with a brief description, are listed in Table 2.

Name	Description
CONDITION	The condition for a branching statement.
CONTAINS	Indicates that a concept is included in another one; for example, code concepts inside a function are contained in the function's concept.
COMMENT	Relates a comment to the concept it is commenting.
DEFINES	Indicates that a concept is defined by another one.
DEPENDS	A dependency on another concept (in C, an `include` statement).
JUMPS	Unconditional jump from concept to concept.
PARAMETER	The concept is a parameter of another concept.
RETURNS	Indicates that a concept is a return value of another one.
TYPEDEF	The concept is defined as a type.

Table 2: Relation Types and their possible placement

3.2 Graph Construction

We now turn to a description of a mechanism for converting source code into SCGs. We define a set of procedures to be carried out while parsing the code, when certain grammar rules of the language are used; as this is a programming (not a natural) language, the parsing process is unambiguous and the points where the procedures are carried out are well-defined. These procedures include creation of new concepts as the code is parsed, assigning referents to them, and connecting them with relations. The entire process is similar to compilation of the code — however, instead of producing the Abstract Syntax Tree, as a compiler would, we generate a conceptual graph. An example procedure (for an IF statement) is presented in Figure 2. Additional mechanisms

statement → **if** "(" expr e ")" statement s_1
    $concept_1$:= create_concept(IF)
    connect_concepts($concept_1$, e, CONDITION)
    connect_concepts($concept_1$, s_1, CONTAINS)
  (else statement s_2)?
    connect_concepts($concept_1$, s_2, CONTAINS)
  | ...

Figure 2: Fragment of the Graph Construction Grammar - IF statement

are used to handle extra-grammatical data, e.g., comments and dependency statements. An example of code and the graph which our construction process produces is given in Figure 3.

4 Retrieving Code

After defining a mechanism for representing code as conceptual graphs, the retrieval process is straightforward: given a source code snippet q and a collection of source code files D, we rank all documents in D according to their similarity to q. Both the documents and the query are represented as conceptual graphs, so we need to use a similarity measure for CGs.

A number of techniques exist for conceptual graph comparison. A family of projections and morphisms was already defined with the presentation of CGs in (Sowa, 1984). The main disadvantage of morphisms is their strictness: in essence, they are aimed at locating identical graphs or subgraphs. Such similarity measures are unfit

```
#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;
}
```

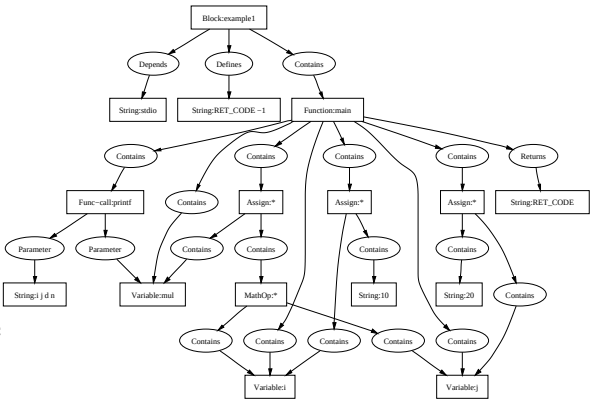

Figure 3: `example.c and its conceptual graph`

for the "fuzzy" matching criteria needed for IR. More relaxed measures exist (Bunke and Messmer, 1993; Dieng, 1996; Montes-y-Gomez et al., 2000; Poole and Campbell, 1995), but most of them tend to require a high level of structural similarity, basing the similarity on morphisms and assigning a lower importance to the information in the concept nodes itself. Structural fuzziness is permitted, but at the cost of complex pre-requisites (sets of transformations between graphs, interest functions). For example, the reported similarity measures will not render the graphs in Figure 4 highly similar, although they may be related. Instead, we introduce a similarity measure that exploits

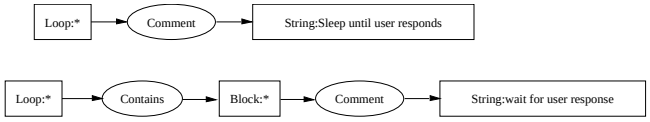

Figure 4: Related graphs

both structure and content. The measure is based on the notion of *contextual similarity*, according to which concepts should be compared not only by taking into account the information contained in them, but also by making use of the information contained in the concepts related to them, i.e., their *context*. Instead of comparing the actual structure of the graphs, we choose to compare the graphs node-by-node. The structure of the graph is implicitly used by augmenting each concept node with the information contained in the concepts which are related to it. This expansion process also takes into account the relation type between the concepts by using the type as a weight, affecting the importance given to the expanded information (see Figure 5).

We will now formally define the components of the similarity measure. Given that we have opted for a rich representation of source code in terms of conceptual graphs, we have to take care of a number of aspects:

- the weights associated with components of a conceptual graph;
- similarity notions between concepts in the graph;

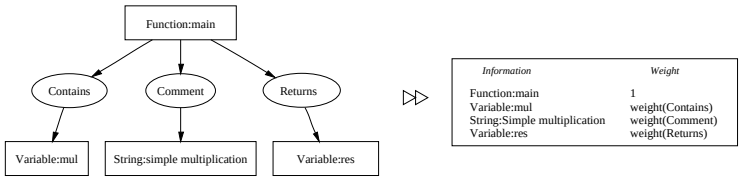

Figure 5: Concept Extension

- the process of expanding a concept node by importing information from related nodes;
- similarity notions between expanded concepts.

Let's examine one after the other now.

We start by discussing the weights associated with components of the conceptual graph. The *concept type weight* ($w_c^t(c)$) is a value indicating 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. The *concept referent weight* ($w_c^r(c)$) is a value proportional to the amount of information kept in a concept (measured, for example, by its length). The *concept weight* ($w_c(c)$) is simply the product of the two previous components of the weight of the concept. Similarly, the *relation weight* ($w_r(r)$) is a value associated with the importance of a relation type.

Next, we define the basic similarity notions between the concepts in the graph. The *concept type similarity* ($sim_c^t(c_i, c_j)$) measures the similarity between the types of two concepts. For example, it indicates how similar a LOOP concept type is to a BLOCK type. A naive approach for calculating this value is to assign 1 to it if the concept type is identical or if one concept type is inherited from the other, and a low value otherwise. The *concept referent similarity* ($sim_c^r(c_i, c_j)$) measures the similarity between the content of the concepts; it can be any information-theoretic measure (we use the Levenstein string-distance value). Finally, the *concept similarity* ($sim_c(c_i, c_j)$) measures the total similarity between concepts, and 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 now turn to the "information extension" process informally described earlier. A *weighted extended concept* 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, and a weight associated with every referent. The *concept extension* process (marked $ext(c)$) is then a function $f : C \times C \times \mathbb{R} \rightarrow C$, i.e., a function that takes a pair of (possibly extended) concepts and a weight, 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.

In a similar way, we expand this notion to an *extended concept of order* n ($ext_n(c)$), which 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 concept c_1, 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 words, it is 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 $n - 1$.

Equipped with these definitions, we are now able to discuss the similarity between extended concepts. The *extended concept similarity* ($extsim(c_i, c_j)$) 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 referents $c_{1,1}, \ldots, c_{1,n}$ and $c_{2,1}, \ldots, c_{2,m}$, where the concept types are $T(c_{1,1}), \ldots, T(c_{1,n})$ and $T(c_{2,1}), \ldots, T(c_{2,m})$, the referents $R(c_{1,1}), \ldots, R(c_{1,n})$ and $R(c_{1,1}), \ldots, R(c_{1,n})$, and the weights $w_{1,1}, \ldots, w_{1,n}$ and $w_{2,1}, \ldots, w_{2,m}$, respectively. The *extended concept similarity* is then defined to be

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

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.

Similarly, the *extended concept similarity of order* n ($extsim_n(c_i, c_j)$) measures the similarity between two concepts with contextual information up to depth n; that is, $extsim_n(c_i, c_j) = extsim(ext_n(c_1), ext_n(c_2))$.

Since the graphs are compared node-by-node, every concept in a graph G_1 must first be matched to a concept in another graph G_2 to which it is compared. For this, we define the notion of a *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 maximal concept similarity to c_1^*. We will use the notation $MSC(c_1, G_2)$ for this concept; note that there may be more than one *MSC*; in that case, one can be selected arbitrarily. 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 define our core notion: the similarity measure between two SCGs G_1, G_2 using the above definitions and notations, as follows:

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

In words, the similarity is a sum of all extended similarities of all most similar concept pairs.

The complexity of the comparison process is polynomial, $O(|G|^3)$; this is substantially higher than the standard linear retrieval complexity. Additionally, the "atomic"

($O(1)$) operations included in this process are in practice much more complex than the simple $O(1)$ operations in standard document retrieval. We implemented a number of techniques both for reducing the number of graphs actually compared to the query and for reducing the complexity of a single comparison; other modifications can be made to reduce both of these. Such modifications include an implementation of an indexing mechanism for the graphs, offline calculation of some of the intermediate results, and shallow, low-complexity "first-step" retrieval of a relatively small number of candidates which are then promoted to a second, deeper, comparison process.

5 Evaluation

In this section we describe the experiments carried out to evaluate the SCG retrieval method and discuss their results.

5.1 Experimental Setting

Although many large open-source projects exist, 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 GNU compiler suite. The collection includes the compiler's test-suite for the C language, and consists of 2932 files written in C. The reasons for using this corpus include the popularity of C as a programming language for large-scale projects, as well as the fact that the collection 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 makes them an interesting test case for comparison and retrieval.

Our experiments concerned two main "tasks":

Identical Document Retrieval. For this task, we used 25 documents chosen randomly out of the collection as queries. For each such query, the 5 top ranking documents were retrieved. The purpose of this task 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. Second, a more "classic" retrieval experiment aim: to analyze the rest of the top ranking documents, and check whether they are indeed relevant to the query.

Modified Document Retrieval. For this task we introduced a new set of documents derived from the 25 documents used for the previous experiment. Each document was subjected to code changes that have been reported as frequent in (Wagner). These include token name changes, comment modifications, coding-convention changes and so on. The new documents were then used as queries which had at least one highly "relevant" document in the collection: the document which was modified to create the query. The goal of this task 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 [1].

For both tasks, we measured the *Mean Reciprocal Rank* (MRR), a common measure for known item search, as well as the *precision@5* (precision for the top 5 retrieved documents) (Baeza-Yates and Ribeiro-Neto, 1999). 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.

We conducted two rounds of experiments: one aimed at comparing our SCG retrieval model against other (baseline) methods on the two tasks just described, and one aimed at understanding the way in which our SCG method mixes content and structure aspects of source code.

5.1.1 Baselines

As a baseline, we use Jacques Savoy's version of the probabilistic retrieval model Okapi (Savoy, 2003), 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 values of n (Jijkoun et al., 2003), which is a desired feature in case the recall is difficult to assess. The retrieval was performed on the source code files after the standard process of tokenization and stopword removal (using both English and C stopword lists). For indexing the files, we used *FlexIR* (Monz and de Rijke, 2002), a vector-space information indexing and retrieval system developed at the University of Amsterdam.

Our SCG retrieval method uses string-distance measures, since many typical differences between similar code files include tokens which have a small string distance between them. To measure the effect of the "contextual knowledge" gained through the graph representation, we also compared our results to two simpler baselines: a simple string-distance measure, and a "typed" string-distance one. In the simple distance case, we rank all documents in the collection according to their string-distance from the (stopped) query; for the "typed" string-distance variation, we first classify each token as belonging to one of the classes {`comment`, `dependency-statement`, `other`}, and then sum the string-distance from each corresponding set of string classes.

After initial testing of the effect of various parameters on the retrieval process, we used a depth of 1 both for the actual comparison and for locating the Most Similar Concepts. This means that each concept was extended with contextual information from its immediate neighbors, but not more. Although it seems that including more contextual data should improve the results, it may also cause irrelevant, noisy information to be added to the actual content of the node, resulting in a decrease in performance.

5.2 Comparison Against the Baselines

A summary of the comparison between the graph retrieval method and the baseline models is given in Table 3. Going down the table, we see a consistent increase of performance, with the exception of the MRR scores of the string distance methods

[1] The coprus and queries are available from `http://www.science.uva.nl/~gilad/scg/`

	Identical Document		Modified Document	
Model	*MRR*	*P@5*	*MRR*	*P@5*
Simple distance	0.973	0.400	0.093	0.056
Typed distance	1.000	0.424	0.293	0.128
Okapi	0.870	0.464	0.400	0.248
SCG Retrieval	0.905	0.472	0.813	0.296

Table 3: MRR and P@5 comparison of retrieval methods

in the Identical Document task; otherwise, 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. The typed string distance, which brings in a bit of structural information 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.

Due to the relatively low number of queries, establishing statistical significance is hard, and indeed only the improvement of the MRR score in the case of the Modified Document task is statistically significant, with $p < 0.003$. Expanding the test set requires substantial manual effort (modifying documents for the Modified task and assessing precision scores for both tasks); therefore, we only repeated the "fully automated" measurement, i.e., the MRR for the Identical Document task, this time with 250 queries instead of 25. Our results indicate a similar improvement over Okapi as in the 25-query experiment (about 4%), but this time with statistical significance ($p < 0.0031$); so, we consider the MRR improvement for both tasks to be statistically significant.

5.3 Combination Experiments

In addition to comparing our method to the baselines, we conducted another experiment which combined the results of two different methods: the most successful baseline method (Okapi) and the conceptual graph retrieval method. The two methods employ very different retrieval approaches: Okapi uses "classical" retrieval notions of term frequency, document length etc., and the SCG method is based on contextual string-distance comparisons. It has been shown that combination of retrieval methods tends to be useful when the retrieval paradigms are different (Lee, 1995; Shaw and Fox, 1994); this makes the combination of Okapi and the SCG method seem promising.

An analysis of the ranked lists returned by the two methods reveals differences both in the retrieved relevant documents and the ordering of the same relevant documents. For example, in the Modified Document task, in almost all cases where there was no

relevant document in the top 5 retrieved documents of one method, the other method ranked at least one relevant document in its top 5 (see also Figure 6); additionally, looking at the MRR scores, in 50% of the cases where the graph retrieval method did not rank the relevant document in the top 10, Okapi retrieved it at the top rank, and in 80% of the cases where Okapi did not get relevant documents in the top 10, the SCG method ranked it at the top position. A closer look at the different successes and failures shows, as expected, that Okapi succeeds in retrieving queries that contain rare tokens which appear only in relevant documents; on the other hand, the SCG method is successful when the token names are common and meaningless, and the structure plays a more important role. We conclude that the different kinds of ranked lists produced by the methods suggest that something can be gained by combining the two approaches.

To combine the scores of the methods we used the same method as described in (Kamps et al., 2003), i.e., a linear combination of the normalized similarity scores of the two methods: $sim_{\text{new}} = \lambda \cdot sim_{\text{SCG}} + (1 - \lambda) \cdot sim_{\text{Okapi}}$, where we follow (Lee, 1995, p. 185) and normalize the similarity scores into $[0, 1]$ using the minimal and maximal similarity scores. The linear combination of two ranked lists tends to improve over the underlying runs by improving recall and/or puts found relevant documents at a higher rank (Kamps et al., 2003).

The results of the combination experiments are presented in Table 4; notice that the combination with $\lambda = 0.0$ is simply the Okapi baseline run, and the one with $\lambda = 1.0$ is the SCG Retrieval run, as reported in Table 3. A further breakdown by topic of

	Identical Document		**Modified Document**	
λ	*MRR*	*P@5*	*MRR*	*P@5*
0.0	0.870	0.464	0.400	0.248
0.2	1.000	0.504	0.630	0.272
0.5	0.960	0.488	0.840	0.352
0.8	0.960	0.448	0.833	0.336
1.0	0.905	0.472	0.813	0.296

Table 4: Combining Okapi and SCG-retrieval

the differences between the precision scores of Okapi, SCG and the combined method (with $\lambda = 0.5$, and sorted by decreasing precision value of the combined method) is presented in Figure 6, showing the different behaviors of the two models and the "smoother" results obtained by combining them. Any combination improves on 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. A possible conclusion of these results is that, as expected, if there are more "visible" differences between the query and the relevant documents, more structural information is needed to establish the similarity.

6 Conclusions and Future Work

We presented a retrieval model for source code documents; this model exploits the highly structured nature of programming language text by extracting the information

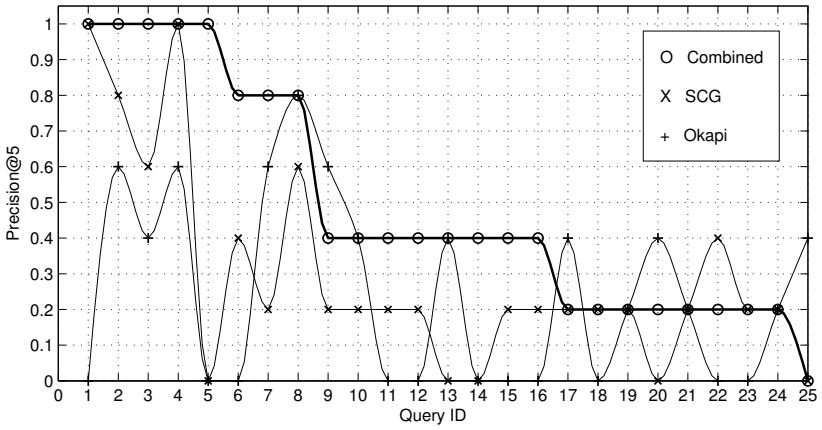

Figure 6: Precision scores for SCG, Okapi and combination: Topic Breakdown

embedded in it and merging it with the content to form a single representation, based on conceptual graphs. We offer a similarity measure for these representations which uses the notion of "contextual similarity," expanding content with its local structural context.

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. While the amount of evaluation done serves only as a proof-of-concept, far from the amount required to support firm conclusions, it appears that exploiting structure helps in the case of code retrieval.

The weak points of our approach are the high complexity and the large amount of free parameters involved in the process (such as relation weights, match depth etc). To address the first issue, we have implemented a number of complexity-reducing mechanisms which boosted our experiments run-times by large factors, and are currently examining other mechanisms to further improve the complexity bounds. The second issue can only be approached with more experimentation and evaluation of our proposed method.

Acknowledgments The authors would like to thank Maarten Marx for useful comments and discussions. Gilad Mishne was supported by the Netherlands Organization for Scientific Research (NWO) under project number 220-80-001. Maarten de Rijke was supported by NWO under project numbers 365-20-005, 612.069.006, 612.000.106, 220-80-001, 612.000.207, and 612.066.302.

References

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

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.

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

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.

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.

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.

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.

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.

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

W. B. Frakes and B. A. Nejmeh. Software Reuse through Information Retrieval. *SIGIR Forum*, 21(1-2):30–36, 1987.

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

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

INEX. Initiative for the Evaluation of XML retrieval, URL: `http://inex.is.informatik.uni-duisburg.de:2004`, Accessed February 2004.

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.

V. Jijkoun, G. Mishne, C. Monz, M. de Rijke, S. Schlobach, and O. Tsur. The university of amsterdam at trec 2003. In *TREC 2003 Working Notes*. National Institute for Standards and Technology, 2003.

E.L. Jones. Metrics based plagarism monitoring. In *The journal of computing in small colleges*, pages 253–261. The Consortium for Computing in Small Colleges, 2001.

J. Kamps, C. Monz, and M. de Rijke. Combining evidence for cross-lingual information retrieval. In C. Peters, Braschler, M., Gonzalo, J., Kluck, M., editor, *Results of the CLEF-2002, cross-language evaluation forum*. Springer, 2003.

J.H. Lee. Combining multiple evidence from different properties of weighting schemes. In *SIGIR'95*, pages 180–188, 1995.

Y.S. Maarek, D.M. Berry, and G.E. Kaiser. Guru: Information retrieval for reuse. In *Landmark Contributions in Software Reuse and Reverse Engineering*. Prentice Hall, 1994.

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

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.

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.

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.

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.

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.

PMD. Project Mess Detector, URL: `http://pmd.sourceforge.net`, Accessed February 2004.

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.

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.

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

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.

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*. Springer, 2003.

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

SCO vs. IBM. URL: `http://www.sco.com/ibmlawsuit/`, Accessed February 2004.

J.A. Shaw and E.A. Fox. Combination of multiple searches. In *Text REtrieval Conference 2*, pages 243–252. National Institute for Standards and Technology, 1994.

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.

Simian. Similarity Analyser, URL: `http://simian.dev.java.net/`, Accessed February 2004.

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

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.

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.

N.R. Wagner. Plagiarism by Student Programmers. `http://www.cs.utsa.edu/~wagner/pubs/plagiarism0.html`, Accessed February 2004.

WebKB. The WebKB set of tools: a common scheme for shared WWW Annotations, shared knowledge bases and information retrieval, URL: `http://www.webkb.org`, Accessed February 2004.

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.

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.