Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing

Andrian Marcus, Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent Ohio 44242
330 672 9039
amarcus@cs.kent.edu, jmaletic@cs.kent.edu

Abstract

An information retrieval technique, latent semantic indexing, is used to automatically identify traceability links from system documentation to program source code. The results of two experiments to identify links in existing software systems (i.e., the LEDA library, and Albergate) are presented. These results are compared with other similar type experimental results of traceability link identification using different types of information retrieval techniques. The method presented proves to give good results by comparison and additionally it is a low cost, highly flexible method to apply with regards to preprocessing and/or parsing of the source code and documentation.

1. Introduction

Extensive effort in the software engineering community (both research and commercial) has been brought forth to improve the explicit connection of documentation and source code. A number of integrated development environments and CASE tools are particularly focused on this issue. These tools and techniques have made great strides in documentation to source code traceability for the development of new software systems. Unfortunately, many of these methods are inherently unsuited to being applied to existing and/or legacy systems.

The need for tools and techniques to recover documentation to source code traceability links in legacy systems is particularly important for a variety of software engineering tasks. These include general maintenance tasks, impact analysis, program comprehension, and more encompassing tasks such as reverse engineering for redevelopment and systematic reuse.

The major obstacle facing construction of useful tools for this type of link recovery is that the links are rarely explicit and (not without exception) based on the semantic meaning of the prose in the documentation. Relating some sort of natural language analysis of the documentation with that of the source code is an obviously difficult problem.

Our solution to this problem is to utilize an advanced information retrieval technique (i.e., latent semantic analysis) to extract the meaning (semantics) of the documentation and source code [19, 21]. Then use this information to identify traceability links based on similarity measures.

The method utilizes all the comments and identifier names within the source code to produce semantic meaning with respect to the entire input document space. This is supported well by the work of Anquetil [1] and others in determining the importance of this information in existing software. This implies the assumption that the comments and identifiers are reasonably named however, the alternative bares little hope of deriving a meaning automatically (or even manually).

A distinct advantage of using our method is that it does not rely on a predefined vocabulary or grammar for the documentation or source code. This allows the method to be applied without large amounts of preprocessing or manipulation of the input, which drastically reduces the costs of link recovery.

The following section describes the use of information retrieval (IR) methods to software systems and how latent semantic analysis works. We try to highlight the other researchers who have worked in this area in the next section and again in the following related work section. Section 4 details the traceability link recovery process and the following section describes our experiments on existing software systems. These results are then compared with the results of other researchers to highlight the validity and usability of our method.

2. IR and Program Analysis

There is a wide variety of information retrieval methods. Traditional approaches [11, 22] include such methods as signature files, inversion, classifiers, and clustering. Other methods that attempt to capture more information about the documents, to achieve better performance, include those using parsing, syntactic information, natural language processing techniques, methods using neural networks, and advanced statistical methods. Much of this work deals with natural language text and a large number of techniques exist for indexing, classifying, summarizing, and retrieving text documents. These methods produce a profile for each document where the profile is an abbreviated description of the

original document that is easier to manipulate. This profile is typically represented as vector, often real valued. The method we use also has an underlying vector space model. We now present a discussion of this type of representation.

2.1 Vector Space Model

The vector space model (VSM) [23] is a widely used classic method for constructing vector representations for documents. It encodes a document collection by a term-by-document matrix whose [i, j]th element indicates the association between the ith term and jth document. In typical applications of VSM, a term is a word, and a document is an article. However, it is possible to use different types of text units. For instance, phrases or word/character n-grams can be used as terms, and documents can be paragraphs, sequences of n consecutive characters, or sentences. The essence of VSM is that it represents one type of text unit (documents) by its association with the other type of text unit (terms) where the association is measured by explicit evidence based on term occurrences in the documents. A geometric view of a term-by-document matrix is as a set of document vectors occupying a vector space spanned by terms; we call this vector space *VSM space*. The similarity between documents is typically measured by the cosine or inner product between the corresponding vectors, which increases as more terms are shared. In general, two documents are considered similar if their corresponding vectors in the VSM space point in the same (general) direction.

2.2. Latent Semantic Indexing

Latent Semantic Indexing (LSI) [7, 8] is a VSM based method for inducing and representing aspects of the meanings of words and passages reflective in their usage.

Work applying LSI to natural language text by [5, 14] has shown that that LSI not only captures significant portions of the meaning of individual words but also of whole passages such as sentences, paragraphs, and short essays. The central concept of LSI is that the information about word contexts in which a particular word appears, or does not appear, provides a set of mutual constraints that determines the similarity of meaning of sets of words to each other.

2.3. Singular Value Decomposition and LSI

In its typical use for text analysis, LSI uses a user-constructed corpus to create a term-by-document matrix. Then it applies Singular Value Decomposition (SVD) [23] to the term-by-document matrix to construct a subspace, called an LSI subspace. New document vectors (and query vectors) are obtained by orthogonally projecting the corresponding vectors in a VSM space (spanned by terms) onto the LSI subspace.

According to the mathematical formulation of LSI, the term combinations which are less frequently occurring in the given document collection tend to be precluded from the LSI subspace. This fact, together with our examples above, suggests that one could argue that LSI does "noise reduction" if it was true that less frequently co-occurring terms are less mutually-related, and therefore less sensible.

The formalism behind SVD is rather complex and lengthy to be presented here. The interested reader is referred to [23] for details. Intuitively, in SVD a rectangular matrix $\mathbf{X}$ is decomposed into the product of three other matrices. One component matrix ($\mathbf{U}$) describes the original row entities as vectors of derived orthogonal factor values, another ($\mathbf{V}$) describes the original column entities in the same way, and the third is a diagonal matrix ($\mathbf{\Sigma}$) containing scaling values such that when the three components are matrix-multiplied, the original matrix is reconstructed (i.e., $\mathbf{X} = \mathbf{U\Sigma V^T}$). The columns of $\mathbf{U}$ and $\mathbf{V}$ are the left and right singular vectors, respectively, corresponding to the monotonically decreasing (in value) diagonal elements of $\mathbf{\Sigma}$ which are called the singular values of the matrix $\mathbf{X}$. When fewer than the necessary number of factors are used, the reconstructed matrix is a least-squares best fit. One can reduce the dimensionality of the solution simply by deleting coefficients in the diagonal matrix, ordinarily starting with the smallest. The first $\mathbf{k}$ columns of the $\mathbf{U}$ and $\mathbf{V}$ matrices and the first (largest) $\mathbf{k}$ singular values of $\mathbf{X}$ are used to construct a rank-$\mathbf{k}$ approximation to $\mathbf{X}$ through $\mathbf{X_k} = \mathbf{U_k\Sigma_k V_k^T}$. The columns of $\mathbf{U}$ and $\mathbf{V}$ are orthogonal, such that $\mathbf{U^TU} = \mathbf{V^TV} = \mathbf{I_r}$, where $\mathbf{r}$ is the rank of the matrix $\mathbf{X}$. $\mathbf{X_k}$ constructed from the $\mathbf{k}$-largest singular triplets of $\mathbf{X}$ (a singular value and its corresponding left and right singular vectors are referred to as a *singular triplet*), is the closest rank-$\mathbf{k}$ approximation (in the least squares sense) to $\mathbf{X}$.

With regard to LSI, $\mathbf{X_k}$ is the closest $\mathbf{k}$-dimensional approximation to the original term-document space represented by the incidence matrix $\mathbf{X}$. As stated previously, by reducing the dimensionality of $\mathbf{X}$, much of the "noise" that causes poor retrieval performance is thought to be eliminated. Thus, although a high-dimensional representation appears to be required for good retrieval performance, care must be taken to not reconstruct $\mathbf{X}$. If $\mathbf{X}$ is nearly reconstructed, the noise caused by variability of word choice and terms that span or nearly span the document collection won't be eliminated, resulting in poor retrieval performance.

The implementation of LSI has been empirically studied. The work in [8] investigates the effects of several term weighting schemes to instantiate the input term-by-document matrix. Evaluation was based on the precision/recall curves on the retrieval tasks with the dimensionality of the LSI subspace being fixed. Several term-weighting schemes, which combine global weights

(i.e., statistics in the collection) and local weights (i.e., statistics within each document), were investigated. LogEntropy, which is a combination of a local log weight and a global entropy weight, showed superiority over the combinations of the local term frequency and global weighting schemes or no global weighting. Two well-known global weightings (i.e., Gfldf and Normal) produced performance worse than no global weighting.

Importantly, several studies have shown that the performance of LSI significantly varies over the dimensionalities of the LSI subspace [7-9]. In practice, the dimensionality is determined experimentally, or "blindly" picked by following the previous work.

Once the documents are represented in the LSI subspace, the user can compute similarities measures between documents by the cosine between their corresponding vectors or by their length. These measures can be used for clustering similar documents together, to identify "concepts" and "topics" in the corpus. This type of usage is typical for text analysis tasks. The LSI representation can also be used to map new documents (or queries) into the LSI subspace and find which of the existing documents are similar (relevant) to the query. This usage is typical for information retrieval tasks.

2.4. Why Use LSI?

A common criticism of VSM is that it does not take account of relations between terms. For instance, having "automobile" in one document and "car" in another document does not contribute to the similarity measure between these two documents.

The fact that VSM produces zero similarity between text units that share no terms is an issue, especially in the information retrieval task of measuring the relevance between documents and a query submitted by a user. Typically, a user query is short and does not cover all the vocabulary for the target concept. Using VSM, "car" in a query and "automobile" in a document do not contribute to retrieving this document (i.e., the synonym problem). LSI attempts to overcome this shortcoming by choosing linear combinations of terms as dimensions of the representation space. The examples in [7, 15] show that LSI may solve this synonym problem by producing positive similarity between related documents sharing no terms.

As the LSI subspace captures the most significant factors (i.e., those associated with the largest singular values) of a term-by-document matrix, it is expected to capture the relations of the most frequently co-occurring terms. This fact is understood when we realize that the SVD factors a term-by-document matrix into the largest one-dimensional projections of the document vectors, and that each of the document vectors can be regarded as a linear combination of terms. In this sense, LSI can be regarded as a corpus-based statistical method. However, the relations among terms are not modeled explicitly in the computation of LSI subspace, which makes it hard to understand LSI in general. Although the fact that an LSI subspace provides the best low rank approximation of the term-by-document matrix is often referred to, it does not imply that the LSI subspace approximates the "true" semantics of documents.

Another of the criticisms of this type method, when applied to natural language texts is that it does not make use of word order, syntactic relations, or morphology. However, very good representations and results are derived without this information [6]. This characteristic is very well suited to the domain of source code and internal documentation. Because much of the informal abstraction of the problem concept may be embodied in names of key operators and operands of the implementation, word ordering has little meaning. Source code is hardly English prose but with selective naming, much of the high level meaning of the problem-at-hand is conveyed to the reader (i.e., the programmer). Internal source code documentation is also commonly written in a subset of English [10] that also lends itself to the IR methods utilized. This makes automation drastically easier and directly supports programmer defined variable names that have implied meanings (e.g., avg) yet are not in the English language vocabulary. The meanings are derived from usage rather than a predefined dictionary. This is a stated advantage over using a traditional natural language type approach.

Like a number of other IR methods, LSI does not utilize a grammar or a predefined vocabulary. However, it uses a list of "stop words" that can be extended by the user. These words are excluded from the analysis. Regardless of the IR method used in text analysis, in order to identify two documents as similar they must have in common concepts represented by the association of terms and their context of usage. In other words, two documents written in different languages will not appear similar. In the case of source code, our main assumption is that developers use the same natural language (e.g., English, Romanian, etc.) in writing internal documentation and external documentation. In addition, the developer should have some consistency in defining and using identifiers.

3. Related Work

The work presented in this paper addresses two specific problems: using IR methods to support software engineering tasks and recovering source code to documentation links. The research that has been conducted on the specific use of applying information retrieval methods to source code and associated documentation typically relates to indexing reusable components [12, 13, 16, 17]. Notable is the work of Maarek [16, 17] on the use of an IR approach for automatically constructing software libraries. The

success of this work along with the inefficiencies and high costs of constructing the knowledge base associated with natural language parsing approaches to this problem [10] are the main motivations behind our research. In short, it is very expensive (and often impractical) to construct the knowledge base(s) necessary for parsing approaches to extract even reasonable semantic information from source code and associated documentation. Using IR methods (based on statistical and heuristic methods) may not produce as accurate results, but they are quite inexpensive to apply. If this is then coupled with the structural information (about the program) it should produce good quality and low cost results.

More recently, Maletic and Marcus [19-21] used LSI to derive similarity measures between source code elements. These measures were used then to cluster the source code to help for the identification of abstract data types in procedural code and the identification of concept clones. In addition, these measures were used to define a cohesion metric for components. The work presented here extends these results in a new direction. At the same time, Antoniol et al. [2-4] investigated the use of IR methods to support the traceability recovery process. In particular, they used both a probabilistic method [3, 4] and a vector space model [2] to recover links between source code and documentation, and between source code and requirements. Their results were promising in each case. Additionally this work also supports the choice of vector space models over probabilistic IR. Using LSI makes the work presented here quite different in many aspects and yet provides complementary results.

4. The Traceability Recovery Process

Our traceability recovery process centers on LSI and is partially automated. However, user input is necessary and the degree of user involvement depends on the type of source code and the user's task. As mentioned, recovering the links between source code and documentation may support various software engineering tasks. Different tasks (and users) typically require different types of information. For example, there are times completeness is important. That is, the user needs to recover ALL the correct links even if that means recovering many incorrect ones at the same time. Other times, precision is preferred and the user restricts the search space so all the recovered links will be correct ones, even if this means not finding them all. Our system tries to accommodate both needs (separately of course). One way to accommodate the user needs is by offering multiple ways to recovering the traceability links. We will discuss these approaches in the section 4.3.

Figure 1 depicts the major elements in the traceability recovery process. The entire process is organized in a pipeline architecture; the output from one phase constitutes the input for the next phase. The user's involvement in the process occurs in the beginning for selecting the source code and documentation files. Then the user selects the dimensionality of the LSI subspace. After the LSI subspace is generated, the user determines what type of threshold will be used in determining the traceability links.

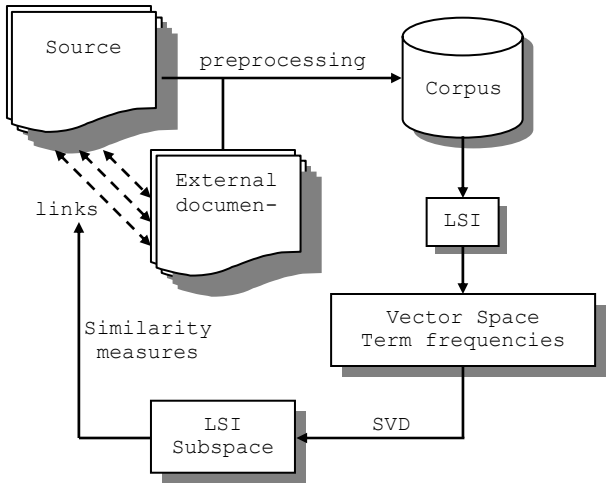

Figure1: The traceability recovery process

4.1. Building the Corpus

The input data consists of the source code and external documentation. In order to construct a corpus that suits LSI, a simple preprocessing of the input texts is required. Both the source and the documentation need to be broken up into the proper granularity to define the documents, which will then be represented as vectors.

In general, when applying LSI to natural text, a paragraph or section is used as the granularity of a document. Sentences tend to be to small and chapters too large. In source code, the analogous concepts are function, structure, module, file, class, etc. Obviously, statement granularity is too small. More than that, the choice of the granularity level is influenced by the particular software engineering task. In previous experiments involving LSI and source code, we used functions as documents in procedural source code [19, 21] and class declarations in OO source code [20]. The goal there was to cluster elements of the source code based on semantic similarity, rather than mapping them to documentation.

In this application, a part of the documentation may refer to different structures in the source code (i.e., a class, a hierarchy of classes, a set of functions or methods, a data structure, etc.). Therefore, in order to allow for flexibility and simplicity, which in turn better support automation, we define each file as a document. Obviously, some files will be too large. In those

situations, the files are broken up into parts roughly the size of the average document in the corpus. This ensures that most of the documents have a close number of words and thus may map to vectors of similar lengths. Of course, in some cases this break up of the files could be rather unfortunate, causing some documents from the source code to appear related to the wrong manual sections. It is a trade-off we are willing to take in favor of simplicity and low-cost of the preprocessing. If this situation is unacceptable for the user, they have the option of (re)combining a number of documents into a new one and identify which existing documents are most similar.

As far as documentation is concerned, the chosen granularity is determined by the division in sections of the documents, defined by the original authors (usually summarized in the table of content).

Some text transformations are required to prepare the source code and documentation to form the corpus for LSI. First, most non-textual tokens from the text are eliminated (e.g., operators, special symbols, some numbers, keywords of the programming language, etc.). Then the identifier names in the source code are split into parts based simply on well-known coding standards. For examples all the following identifiers are broken into the words "hot" and "chocolate": "hot_chocolate", "Hot_chocolate", "hot_Chocolate", "Hot_Chocolate", "HotChocolate", "HOTchocolate". The original form of the identifier is also preserved in the documents. Since we do not consider n-grams, the order of the words is not of importance. Finally, the white spaces in the text are normalized, blank lines separate documents, and the source code and documentation are merged.

Important to note is that in this process, no grammar based parsing of the source code is necessary. LSI does not use a predefined vocabulary, or a predefined grammar, therefore no morphological analysis or transformations are required.

One can argue that the mnemonics and words used in constructing the identifier may not occur in the documentation. That is certainly true. It is in fact the reason why we chose to also use the internal documentation (i.e., comments) in constructing the corpus. It has been shown [10] that internal source code documentation is commonly written in a subset of the language of the developer, similar to that of external documentation. In these situations, the performance of LSI is of great benefit since it is able to associate the terms in the text that are in correct natural language (and also found in the external documentation) with the mnemonics from the identifiers. These mnemonics in turn, will contribute to the similarity between to elements of source code that use the same identifiers. Of course, our assumption is that developers define and use the identifiers with some rationale in mind and not completely randomly.

As mentioned previously, we chose not to do parsing and defined files (or parts of) as documents. Several reasons motivated this choice. First, building a high quality parser for programming languages such as C++ is a non-trivial task due to the nature of the language. Second, often time's large software systems, which we target in our analysis tasks, are written using multiple programming languages. In these situations, a number of parsers are needed and this can turn the traceability problem into something simply impractical. Third, sometimes the user has access to only a part of a very large system, which may not be compile-able, thus causing a hard time for a parser. Parsing is used to extract structural information from the source code, rather than semantic (i.e., domain semantics). This type of information does help in the analysis, but we believe that the semantic analysis and the structural analysis should be performed separately and then the results combined. The traceability recovery process will link a piece of documentation to one or more pieces of code, and then structural information can be used to determine what elements of the systems (e.g., classes) are implemented in that piece of code.

4.2. Defining the Semantic Similarity Measure

Before we give a detailed explanation of this process, some mathematical background and definitions are necessary.

Notation. A bold lowercase letter (e.g., $\mathbf{y}$) denotes a *vector*. A vector is equivalent to a matrix having a single column. The i^{th} entry of vector $\mathbf{y}$ is denoted by $\mathbf{y}_{[i]}$.

Notation. A bold uppercase letter (e.g., $\mathbf{X}$) denotes a *matrix*; the corresponding bold lowercase letter with subscript i (e.g., $\mathbf{x}_i$) denotes the matrix's i^{th} column vector. The $[i,j]^{th}$ entry of matrix $\mathbf{X}$ is denoted by $\mathbf{X}_{[i,j]}$. We write $\mathbf{X} \in \boldsymbol{R}^{mxn}$ when matrix $\mathbf{X}$ has m rows and n columns whose entries are real numbers.

Definition. A *diagonal matrix* $\mathbf{X} \in \boldsymbol{R}^{nxn}$ has zeroes in its non-diagonal entries, and is denoted by $\mathbf{X} = \text{diag}(\mathbf{X}_{[1,1]}, \mathbf{X}_{[2,2]}, \ldots, \mathbf{X}_{[n,n]})$.

Definition. An *identity matrix* is a diagonal matrix whose diagonal entries are all one. We denote the identity matrix in $\boldsymbol{R}^{mxm}$ by $\boldsymbol{I}_m$. For any $\mathbf{X} \in \boldsymbol{R}^{mxn}$, $\mathbf{X}\mathbf{I}_n = \mathbf{I}_m\mathbf{X} = \mathbf{X}$. We omit the subscript when the dimensionality is clear from the context.

Definition. The *transpose* of matrix $\mathbf{X}$ is a matrix whose rows are the columns of $\mathbf{X}$, and is denoted by $\mathbf{X}^T$, i.e., $\mathbf{X}_{[i,j]} = (\mathbf{X}^T)_{[j,i]}$. The columns of $\mathbf{X}$ are *orthonormal* if $\mathbf{X}^T\mathbf{X} = \mathbf{I}$. A matrix $\mathbf{X}$ is *orthogonal* if $\mathbf{X}^T\mathbf{X} = \mathbf{X}\mathbf{X}^T = \mathbf{I}$.

Definition. The *vector 2-norm* of $\mathbf{x} \in \boldsymbol{R}^m$ is defined by $|\mathbf{x}|_2 = \sqrt{\mathbf{X}^T\mathbf{X}} = \sqrt{\sum_{i=1}^{m} (\mathbf{x}_{[i]})^2}$ we call it the *length* of $\mathbf{x}$.

Definition. The inner product of $\mathbf{x}$ and $\mathbf{y}$ is $\mathbf{x}^T\mathbf{y}$. The *cosine* of $\mathbf{x}$ and $\mathbf{y}$ is the length-normalized inner product, defined by

$$\cos(\mathbf{x},\mathbf{y}) = \frac{\mathbf{x}^T\mathbf{y}}{|\mathbf{x}|_2 \times |\mathbf{y}|_2} \quad (1)$$

For $\mathbf{x}, \mathbf{y} \neq 0$; note that $\cos(\mathbf{x}, \mathbf{y}) \in [-1, 1]$. A larger cosine value indicates that geometrically $\mathbf{x}$ and $\mathbf{y}$ point in similar directions. In particular, if $\mathbf{x} = \mathbf{y}$ then $\cos(\mathbf{x}, \mathbf{y}) = 1$, and $\mathbf{x}$ and $\mathbf{y}$ are orthogonal if and only if $\cos(\mathbf{x}, \mathbf{y}) = 0$.

Definition. In this process a *source code document* (or simply document) d is any contiguous set of lines of source code and/or text. Typically a document is a file of source code or a program entity such as a class, function, interface, etc.

Definition. An *external document* (e) is any contiguous set of lines of text from external documentation (i.e., manual, design documentation, requirement documents, test suites, etc.). Typically an external document is a section, a chapter, or maybe an entire file of text.

Definition. The external documentation is also a set of documents $M = \{e_1, e_2, \ldots, e_m\}$. The total number of documents in the documentation is $m = |M|$.

Definition. A *software system* is a set of documents (source code and external) $S = D \cup M = \{d_1, d_2, \ldots, d_n\} \cup \{e_1, e_2, \ldots, e_m\}$. The total number of documents in the system is $n + m = |S|$.

Definition. A *file* f_i, is then composed of a number of documents and the union of all files is S. Size of a file, f_i, is the number of documents in the file, noted $|f_i|$.

LSI uses the set $S = \{d_1, d_2, \ldots, d_n, e_1, e_2, \ldots, e_m\}$ as input and determines the *vocabulary* V of the corpus. The number of words (or terms) in the vocabulary is $v = |V|$. Based on the frequency of the occurrence of the terms in the documents and in the entire collection, each term is weighted with a combination of a local log weight and a global entropy weight. A term-document matrix $\mathbf{X} \in \boldsymbol{R}^{vxn}$ is constructed. Based on the user-selected dimensionality (k), SVD creates the LSI subspace. The term-document matrix is then projected onto the k-dimensional LSI subspace. Each document $d_i \in D \cup M$, will correspond to a vector $\mathbf{x_i} \in \mathbf{X}$ projected onto the LSI subspace.

Definition. For two documents d_i and d_j, the *semantic similarity* between them is measured by the cosine between their corresponding vectors $\text{sim}(d_i, d_j) = \cos(\mathbf{x_i}, \mathbf{y_i})$ The value of the measure will be between [-1, 1] with value (almost) 1 representing that the two are (almost) identical.

One important aspect to consider is the granularity of the documents. The external documentation is usually composed of paragraphs, sections, or chapters. These are then natural choices in determining the definition of an external document in particular cases. The organization of the source code differs from one programming language to another. The simplest way to determine the documents granularity is using the file decomposition. Obviously this will not suffice for all tasks. Therefore choosing classes, functions, or interfaces as source code documents is often time more desirable. Since one of the goals in for the framework to be as flexible as possible, using a full parser for each possible language is impractical. We developed a couple of simple lexical parser that can be used to split up C, C++, and Java source code into documents of different granularity levels (i.e., functions, methods, interfaces, and classes).

4.3. Recovering Traceability Links

At this point in the process the similarity between each pair of documents from *M*x*D* are computed. The user has two options, depending on the desired results. One is to determine a threshold ε for the similarity measure, which identifies which documents are considered "linked". In other words among all the pairs from *M*x*D*, only those will be retrieved which have a similarity measure greater than ε. As mentioned, a good, and widely used, heuristic is ε = 0.7. This measure corresponds to a 45° angle between the corresponding vectors. This threshold has yielded good results [18, 21] when assessing the similarity between pieces of source code. The higher the threshold used the closer the weights on the terms occurring in the documents are. One important thing to consider here is that the documents from the M are different in nature than the ones from D. Therefore, a lower threshold may yield good results. The issue of the "best" threshold for this type of corpus (i.e., combining source code and documentation) is still open and further research is needed.

Before we explain the other alternative, a way to determine the "goodness" of the results is needed. Two of the most common measures for the quality of the results in experiments with IR methods are *recall* and *precision*. In general, for a given document d_i, the similarity measure and the defined threshold will be used to retrieve a number N_i of documents, based on the LSI subspace that are deemed similar to d_i. Among these N_i documents, $C_i \leq N_i$ of them are actually similar to d_i. Assume that there are a total of $R_i \geq C_i$ documents that are in fact similar to d_i. With these numbers we define the recall and precision for d_i as follows:

$$\text{Recall} = \frac{C_i}{R_i} = \frac{\#correct \wedge retrieved}{\#correct}\,\%$$

$$\text{Precision} = \frac{C_i}{N_i} = \frac{\#correct \wedge retrieved}{\#retrieved}\,\%$$

Both measures will have values between [0, 1]. If recall = 1, it means that all the correct links are recovered, though there could be recovered links that are not correct. If the precision = 1, it means that all the

recovered links are correct, though there could be correct links that were not recovered. For the entire system the recall and precision are computed as follows:

$$\text{Recall} = \frac{\sum_{i=n+1}^{m} C_i}{\sum_{i=n+1}^{m} R_i}\%; \qquad \text{Precision} = \frac{\sum_{i=n+1}^{m} C_i}{\sum_{i=n+1}^{m} N_i}\%$$

With this in mind, choosing a higher threshold for the link recovery will result in higher precision, while lowering the threshold will increase the recall. In general, the consequence of higher precision is a lower recall (and vice versa).

A second option for the user is to impose a threshold on the number of recovered links, regardless of the actual value of the similarity measure. Thus, the user may chose to select the top θ ranked links for each document, where $\theta \in \{1, 2, 3,\ldots, n\}$. This is the approached preferred by Antoniol et al [2-4] and is a common way to deal with a list of ordered solutions. Finally, the user can opt to combine the two types of thresholds, for example to retrieved the top θ ranked links among those that have a similarity measure greater than ε.

The different choices accommodate different user needs. Using the threshold method, with a high enough threshold value, will yield high precision. Choosing the right threshold is in fact tricky. The only heuristic the user has is past experience. As mentioned, in text retrieval a 0.7 value for the threshold provided good results in a number of earlier experiments. The same threshold may or may not be best suited fro different software systems. Determining a more generally usable heuristic for selection of the appropriate threshold is an issue under investigation and future research will report on it.

Using the ranking method, the user will achieve 100% recall in a in a few number of steps. Precision however will be significantly reduced from one step to the next.

5. Experiments and Results

In the experiments presented here, we have attempted to approximate the ones done by Antoniol et al [2-4] to give us a means of relevant comparison to their work. The goal is to assess how well LSI performs in this type of software engineering task, with respect to other IR methods used by Antoniol et al. There are a number of small differences between the experiments that will be explained.

5.1. Experiment and Results for LEDA

The software system used for analysis is release 3.4 of LEDA (Library of Efficient Data types and Algorithms), a well known library developed and distributed by Max Planck Institut für Informatik, Saarbrücken, Germany (and lately by Algorithmic Solutions Software GmbH) together with its manual pages. This is the same release used by Antoniol et al.

We included in the analysis the entire library, the demo programs, and the entire manual. Table 1 contains the size of the system and manual, as well as the dimensionality used for the LSI subspace and the determined vocabulary.

Table 1. Elements of the LEDA source code, documentation, and LSI settings used in the analysis

LEDA 3.4	Count	Documents
Source code files	491	684
Manual sections	115	119
Total # of documents		803
Classes	219	In 218 files
Vocabulary	3814	-
LSI dimensions used	200 - 400	Increments of 50

We decided to use the entire manual and available source code to ensure the generation of a rich enough semantic space and vocabulary. In the end, we recovered the links for only the 88 manual sections 2.1 through 11.5 that were used in the experiments of Antoniol et al. Although we realized that using extra source code files might corrupt the results in lowering the precision, we decided on this approach since it mimics the situation when the system to be analyzed is new to the engineer and they do not know what is the core part of the system.

In addition, since the number of manual documents is much smaller than the number of source code files, we decided to trace the links from the manual to the source code, rather than vice versa (as done by Antoniol et al). The assumption is that the user can easily read the manual, and the source code is the unknown factor. A typical query would be to find out which parts of the source code are described by a given manual section. Of course, in a bottom-up type of analysis the user might ask which manual section describes a given piece of source

Table 2. Recovered links, recall, and precision using cosine value threshold. For LEDA.

Cosine threshold	Correct links retrieved	Incorrect links retrieved	Missed links	Total links recovered	Precision	Recall
0.60	81	109	33	190	42.98 %	71.01 %
0.65	68	58	46	126	53.97 %	59.65 %
0.70	49	20	65	69	71.05 %	42.63 %

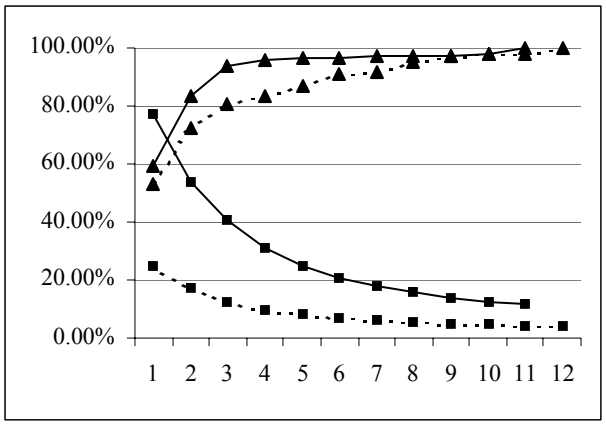

Figure 2. Recall and precision values for experiment by Antoniol and experiments with LSI using LEDA. x-axis represents the cut point and y-axis represents recall/precision values.

code. That can be established with one user query. To retrieve all the links, we believe starting from the smaller set yields better results. One more consideration determined our choice. Since our method does not require parsing of the code, its is not practical to set as starting point the implementation of a class (which can only be determined by some syntactic parsing) to recover the traceability links. Among the 219 classes, 116 are implemented in one file, 95 classes are implemented in two files, seven classes in three files, and one class in 12 files.

We found that the 88 manual sections relate to 80 classes (we did not consider the children in the inheritance hierarchies) implemented in 104 files. 34 of the manual documents relate to two source code files, 46 to one file only, and 8 to relate no class file. 10 of the files contain implementation of multiple classes, described by more than one manual section. Essentially, we found 114 correct links against which we computed recall and precision of LSI. The numbers differ from those in the Antoniol et al experiments mainly because they used classes as units of text, while we used source code files.

Before discussing our results, we should reiterate that the LEDA manual is generated, in large part, directly from the source and include sections of the code and comments. One can argue that in this case the recovery should be trivial. Although it is easier than on a different kind of system with poorer documentation (as it is often the case), it is still far from trivial. By using LEDA for these experiments, we can compare the results with previous research and better assess the quality of our experiments.

As mentioned previously, there are at least two possibilities in determining the traceability links between documentation and source code, using a semantic similarity measure. One is to use a threshold based on the value of the similarity measure and consider that a pair of documents determine a correct traceability link if their semantic similarity is larger than the established threshold. Second (as used by Antoniol et al) is to establish a cut point and consider as correct links all the top ranked pairs down to the cut point. The ranking being determined based on the semantic similarity measure between each pair of documents.

Our assumption was that using the first method with a threshold around 0.7 will yield better precision and lower recall that the second option. Table 2 summarizes the results we obtained on recovering the traceability links between manual pages and source code for LEDA. The first column (Cosine) represents the threshold value; column 2 (Correct links retrieved) represents the number of correct links recovered; column 3 (Incorrect links retrieved) represents the number of incorrect links recovered; column 4 (Missed links) represents the number of correct links that were not recovered; column 5 (Total links retrieved) represents the total number of

Table 3. Retrieved links, recall, and precision using top ranking (for LEDA).

Cut point	Correct links retrieved	Incorrect links retrieved	Missed links	Total links retrieved	Precision	Recall
1	68	20	27	88	77.27 %	59.65 %
2	95	81	19	176	53.98 %	83.33 %
3	107	157	7	264	40.53 %	93.86 %
4	109	243	5	352	30.97 %	95.61 %
5	110	330	4	440	25.00 %	96.49 %
6	110	418	4	528	20.83 %	96.49 %
7	111	505	3	616	18.02 %	97.37 %
8	111	592	3	703	15.79 %	97.37 %
9	111	680	3	791	14.03 %	97.37 %
10	112	767	2	879	12.74 %	98.25 %
11	114	853	0	967	11.79 %	100.0 %

recovered links (correct + incorrect); and the last two columns the precision and recall for each threshold.

We used 0.7 as initial threshold, and although the precision value was indeed good, the recall was rather low. Therefore, we decided to relax the selection criteria and lowered incrementally the threshold. As expected, the recall improved, but the precision deteriorated. Our goal was to determine an approximate correlation between the increase and decrease of the two measures.

To validate our hypothesis we repeated the experiment using a cut point for the best-ranked pairs of documents, as done by Antoniol et al [2-4]. Table 3 summarizes the results obtained in this case. The table is defined just as table 2, except that the first column represents the cut point rather than similarity measure threshold. As we can see, recall and precision seem to be a bit better than in the previous case, contradicting our initial assumptions. In order to compare the results with the Antoniol et al's experiments [2-4], which used both a probabilistic and a VSM IR methods, with parsing the code and morphologically analyzed the texts, we used as many number of cut points as necessary to obtain 100% recall. Figure 2 compares the precision and recall between the two sets of experiments. The values used for Antoniol's experiments are the better they had among the probabilistic and VSM. Dashed lines marked with squares and triangles show the precision and recall, respectively, obtained by Antoniol, while the solid lines indicate the same measures obtained using LSI.

The recall values we obtained are slightly better than the ones of Antoniol, LSI helps reach 100% recall value one step before their methods. The precision however, is much better for LSI in this case, with respect to the probabilistic and the VSM methods used by Antoniol. This came as no surprise considering the very reasons that motivated our preference for LSI to be used in this type of analysis and our choice of starting point (documents rather than source code).

The recall values prompted a closer inspection of the results. We expected better results by comparison (similar to the precision). As seen in table 3, all but 7 of the correct links are recovered after selecting the top 3 ranked pairs of documents. More than that, all but 3 of the correct links are recovered after selecting the top 7 ranked pairs of documents. We looked closer to the remaining 3 pairs. These were the manual sections describing the classes: *integer*, *integer matrix*, and *set*, respectively (i.e., sections 3.1, 3.6, and 4.9, respectively). As most of the other section in the manual, these describe the structure of the classes to help in and reflect the usage of them, rather than describing implementation details. Therefore, files that intensively use any of these classes will have a larger similarity measure that the files which implement the class. Even more, these particular classes are basic types, ubiquitously used throughout the LEDA package.

This fact also explains the precision values. Remember that more than half of the source code does not contribute to the implementation of the classes directly referred to by the manual sections. Again, using LSI will identify pairs of similar documents where the similarity indicates that the source code file uses the methods described in the documentation. In reality, this is a valid similarity and one can claim that in fact the manual page also relates to those source code files. We subscribe to this point of view. However, in order to perform a thorough comparison of the results, we had to define very conservatively the "correct links". The point here is that the method lends itself naturally to recover other types of links between documentation and source code.

5.2. Experiments and Results for Albergate

For a second set of experiments, we used the Albergate system, kindly provided by Giuliano Antoniol and Massimiliano Di Penta. Albergate is implemented in Java by Italian students and has 95 classes. Antoniol et al [2] analyzed 60 classes together with 16 requirements documents. We had only 58 of the classes and 13 of the requirement documents. This fact did not influence the results significantly. In this case, only three files contained the implementation for more than one class. We broke those files so that each file contained only one class. The numbers in table 4 reflect these changes. In other words, the setup for this experiment is almost identical with the one described in [2], since a document in our space corresponds to a class (in most cases). However, this is not true for all documents and a small number represent more than one class. Some of the files were large and were broken into multiple documents. Note that all the documentation is written in Italian. However, our process was essentially unchanged due to the fact our method is not dependent on a language or grammar.

Table 4. Elements of the Albergate source code, documentation, and LSI settings used in the analysis

Albergate	Count	Documents
Source code files	58	76
Requirements files	13	13
Total # of documents		89
Classes	58	76
Vocabulary	1198	
LSI dimensions used	300	

Albergate is a very different system than LEDA. First, it is implemented in Java and has documentation in Italian. Second the external documentation is in the form of requirement documents which describe elements of the problem domain, while in the case of LEDA often the manual pages referred to elements of the solution domain

Table 5. Retrieved links, recall, and precision using top rankings (for Albergate).

Cut point	Correct links retrieved	Incorrect links retrieved	Missed links	Total links retrieved	Precision	Recall
1	26	32	31	58	44.83 %	45.61 %
2	33	83	24	116	28.45 %	57.89 %
3	43	131	14	174	24.71 %	75.44 %
4	49	183	8	232	21.12 %	85.96 %
5	52	238	5	290	17.93 %	91.23 %
6	57	291	0	348	16.38 %	100.00 %

(much better represented in the source code). In addition, the requirement documents are supposed to have been written before implementation and do not include any parts of the internal documentation or the source code. Finally, the requirement documents are very short and have a fixed format with common headings. These headings have nothing in common with the problem domain and are the same in each document. The size of the system was also a concern for us. IR methods in general and LSI in particular, are designed to work on very large corpora. That is, the larger and richer (in semantics) the corpus, the better results. The entire philosophy of LSI is on the reduction of this large corpus to a manageable size without loss of information. When the corpus is small with terms and concepts distributed scarcely throughout the space, reduction of the dimensionality could result in significant loss of information. In consequence, and considering previous results, we expected lower recall and precision values than in the case of LEDA.

Table 5 summarizes the results of the traceability recovery process for Albergate. The structure of the table is the same as table 4, described above.

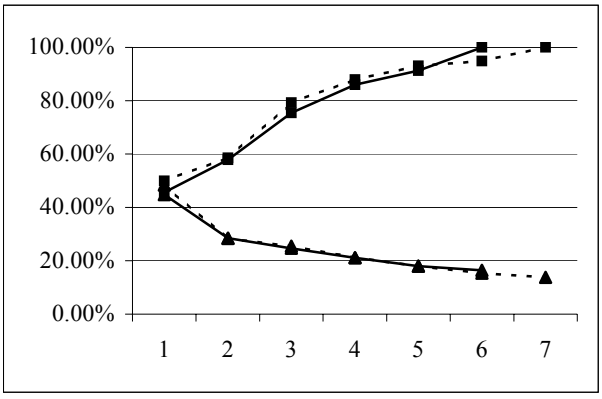

Figure 3. Recall and precision values for experiment by Antoniol and experiments with LSI using Albergate. x-axis represents the cut point and y-axis represents recall/precision values.

Confirming our hypothesis, the precision was lower, however the 100% recall target was reached faster than in the case of LEDA, with better precision. The explanation is that, unlike in the LEDA case, the uses of classes by each other are less intensive in Albergate.

Figure 3 shows graphically how our results compare with those obtained by Antoniol et al [2, 3]. Remember that this time the setup of the experiments and the benchmark mapping are almost identical. The results are very close the only significant difference is that using LSI the 100% recall is reached one step sooner (selecting the top 6 ranked pairs, rather than 7). One more thing to note is that the Albergate source code contains less than 300 lines of internal documentation (i.e., comments).

7. Conclusions and Future Work

The paper presents a method to recover traceability links between documentation and source code, using an information retrieval method, namely Latent Semantic Indexing (LSI). A set of experiments was presented and the results analyzed by comparing them with previous related research by Antoniol et al [2-4].

The method using LSI performs at least as well as Antoniol's methods using probabilistic and VSM IR methods combined with full parsing of the source code and morphological analysis of the documentation.

Our method requires less processing of the source code and documentation, implicitly, less computation. It is language, programming language, and paradigm independent, thus more flexible, and better suited for automation. These characteristics allow us to use the internal documentation in the analysis (not used by Antoniol), which we believe allows LSI to produce better results. The Albergate case is an example of this hypothesis. With almost no comments in the source code, LSI does perform at least as well as the other methods.

The results are promising enough to warrant future research. We shall repeat these experiments on other types of software systems (i.e., written in different programming languages and with different type of documentation). In addition, we will work on improving the results by combining structural and semantic information extracted from the source code and its associated documentation.

Although promising, we believe the results could be further improved. The semantic similarity measure defined using LSI (or any other IR method) could be augmented by considering structural information of the program. For example, in the case of LEDA selecting the top ranked pair of documents (cut point 1 row in table 4) the precision is very high. The recall value is of relatively low. The reason is simple. 95 of the classes are implemented in more than one file so recovering only one of them we miss a large percentage of the links. In this situation, one could use extra structural information to extend this step. For example, file include information (for C/C++), and inheritance information can be utilized to improve results. We are currently experimenting with these enhancements to the method and future work will report on the results.

Finally, we are trying to determine some good heuristics that the user and the system can use to determine the appropriate threshold value for similarity measures. This is dependent on the maintenance task that is addressed, the programming language, the quantity and quality of the documentation.

9. Acknowledgements

This work was supported in part by a grant from the National Science Foundation (CCR-02-04175).

10. References

[1] Anquetil, N. and Lethbridge, T., "Assessing the Relevance of Identifier Names in a Legacy Software System", in Proceedings CASCON 1998, December 1998, pp. 213-222.

[2] Antoniol, G., Canfora, G., Casazza, G., and De Lucia, A., "Information Retrieval Models for Recovering Traceability Links between Code and Documentation", in Proceedings IEEE International Conference on Software Maintenance (ICSM'00), San Jose, CA, October 11-14 2000, pp. 40-51.

[3] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E., "Tracing Object-Oriented Code into Functional Requirements", in Proceedings 8th International Workshop on Program Comprehension (IWPC'00), Limerick, Ireland, June 10-11 2000, pp. 79 - 87.

[4] Antoniol, G., Canfora, G., De Lucia, A., and Merlo, E., "Recovering Code to Documentation Links in OO Systems", in Proceedings 6th IEEE Working Conference on Reverse Engineering (WCRE'99), October 6-8 1999, pp. 136-144.

[5] Berry, M. W., "Large Scale Singular Value Computations", Int. Journal of Supercomputer Applications, 6, 1992, pp. 13-49.

[6] Berry, M. W., Dumais, S. T., and O'Brien, G. W., "Using Linear Algebra for Intelligent Information Retrieval", SIAM: Review, 37, 4, 1995, pp. 573-595.

[7] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R., "Indexing by Latent Semantic Analysis", Journal of the American Society for Information Science, 41, 1990, pp. 391-407.

[8] Dumais, S. T., "Improving the retrieval of information from external sources", Behavior Research Methods, Instruments, and Computers, 23, 2, 1991, pp. 229 - 236.

[9] Dumais, S. T., "Latent Semantic Indexing (LSI): TREC-3 report", in The Third Text REtrieval Conference (TREC3), 1995, pp. 135-144.

[10] Etzkorn, L. H. and Davis, C. G., "Automatically Identifying Reusable OO Legacy Code", IEEE Computer, 30, 10, October 1997, pp. 66-72.

[11] Faloutsos, C. and Oard, D. W., "A Survey of Information Retrieval and Filtering Methods", University of Maryland, Technical Report CS-TR-3514, August 1995.

[12] Fischer, B., "Specification-Based Browsing of Software Component Libraries", in Proceedings of ASE, 1998, pp. 74-83.

[13] Frakes, W., "Software Reuse Through Information Retrieval", in Proceedings 20th Annual HICSS, Kona, HI, Jan. 1987, pp. 530-535.

[14] Landauer, T. K. and Dumais, S. T., "A Solution to Plato's Problem: The Latent Semantic Analysis Theory of the Acquisition, Induction, and Representation of Knowledge", Psychological Review, 104, 2, 1997, pp. 211-240.

[15] Landauer, T. K., Foltz, P. W., and Laham, D., "An Introduction to Latent Semantic Analysis", Discourse Processes, 25, 2&3, 1998, pp. 259-284.

[16] Maarek, Y. S., Berry, D. M., and Kaiser, G. E., "An Information Retrieval Approach for Automatically Constructing Software Libraries", IEEE Transactions on Software Engineering, 17, 8, 1991, pp. 800-813.

[17] Maarek, Y. S. and Smadja, F. A., "Full Text Indexing Based on Lexical Relations, an Application: Software Libraries", in Proceedings SIGIR89, Cambridge, MA, June 1989, pp. 198-206.

[18] Maletic, J. I. and Marcus, A., "Using Latent Semantic Analysis to Identify Similarities in Source Code to Support Program Understanding", in Proceedings 12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI), Vancouver, British Columbia, November 13-15 2000, pp. 46-53.

[19] Maletic, J. I. and Marcus, A., "Supporting Program Comprehension Using Semantic and Structural Information", in Proceedings 23rd International Conference on Software Engineering (ICSE 2001), Toronto, Ontario, Canada, May 12-19 2001, pp. 103-112.

[20] Maletic, J. I. and Valluri, N., "Automatic Software Clustering via Latent Semantic Analysis", in Proceedings 14th IEEE International Conference on Automated Software Engineering (ASE'99), Cocoa Beach Florida, October 1999, pp. 251-254.

[21] Marcus, A. and Maletic, J. I., "Identification of High-Level Concept Clones in Source Code", in Proceedings Automated Software Engineering (ASE'01), San Diego, CA, November 26-29 2001, pp. 107-114.

[22] Salton, G., Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer, Addison-Wesley, 1989.

[23] Salton, G. and McGill, M., Introduction to Modern Information Retrival, McGraw-Hill, 1983.