RECOVERY OF TRACEABILITY LINKS BETWEEN SOFTWARE DOCUMENTATION AND SOURCE CODE

ANDRIAN MARCUS
Department of Computer Science, Wayne State University
Detroit, Michigan, 48202, USA
amarcus@wayne.edu

JONATHAN I. MALETIC
Department of Computer Science, Kent State University
Kent, Ohio, 44242, USA
jmaletic@cs.kent.edu

ANDREY SERGEYEV
Department of Computer Science, Wayne State University
Detroit, Michigan, 48202, USA
andrey@wayne.edu

Received (received date)
Revised (revised date)
Accepted (accepted date)

An approach for the semi-automated recovery of traceability links between software documentation and source code is presented. The methodology is based on the application of information retrieval techniques to extract and analyze the semantic information from the source code and associated documentation. A semi-automatic process is defined based on the proposed methodology.

The paper advocates the use of latent semantic indexing (LSI) as the supporting information retrieval technique. Two case studies using existing software are presented comparing this approach with others. The case studies show positive results for the proposed approach, especially considering the flexibility of the methods used.

Keywords: traceability, information retrieval, latent semantic indexing

1. Introduction

The issue of traceability between software artifacts is currently of great interest to the research and commercial software engineering communities. The state of the art is centered on model definitions, integrated development environments that support such models, and CASE tools. Central to this research is the identification and recovery of explicit links between documentation and source code. Very few existing approaches address the issue of the recovery of links in legacy systems even though a wide variety of software engineering tasks would directly benefit. These include general maintenance tasks, impact analysis, program comprehension, and more encompassing tasks such as reverse engineering for redevelopment and systematic reuse.

Several issues make this problem particularly difficult. First of all, the connection between the documentation and the source is rarely explicitly represented. Second, an inherent problem is that the documentation and the source code are represented at different abstraction levels in the system and in different formalisms (i.e., natural or formal languages versus programming languages). Relating some sort of natural language analysis of the documentation with that of the source code is an obviously difficult problem.

Traditionally, developers are aware of these links, even though they are not explicitly or formally represented. When the information about the links is missing or the software engineers need to deal with someone else's code (as is often the case during maintenance and evolution of software), they try to infer this data manually by inspecting the code, the documentation, and by talking with the other developers. This leads to another associated problem; the size of legacy system is often a prohibitive factor in this manual approach. In consequence, there is a need for tools that automate, at least in part, the process of recovering traceability links between source code and documentation.

1.1. Approach Overview

The approach taken here to the traceability problem is to utilize an advanced information retrieval technique (i.e., latent semantic indexing) to extract the meaning (semantics) of the documentation and source code [Maletic'01, Marcus'01]. We then use this information to define similarity measures between elements of the documentation (expressed in natural language) and components of the software system. These measures are used to identify parts of the documentation that correspond to particular software components, and vice versa.

The methodology is based on the extraction, analysis, and mathematical representation of the comments and identifiers from the source code. A large amount of information from the problem and solution domains is encoded in these elements by the developers. This type of information is used regularly in supporting program comprehension during maintenance and evolution [Anquetil'98a, b, Tjortjis'03]. Information from the documentation is also extracted and the same mathematical representation is used for its encoding. The assumption in this approach is that the comments and identifiers are reasonably named as the alternative bares little hope of deriving a meaning automatically (or even manually).

The approach presents several advantages. One of the most important is its flexibility in usage, determined by the fact that the methodology does not rely on a predefined vocabulary or grammar for the documentation and source code. This also allows the method to be applied without large amounts of preprocessing or manipulation of the input, which drastically reduces the costs of link recovery.

1.2. Bibliographic Notes and Paper Organization

The work presented here extends previous results, described in one of our earlier papers [Marcus'03], in several directions. The proposed process is refined in this paper, in particular the last step, which is redefined here (i.e., three approaches to link recovery are addressed based on the same underlying technology). A new set of case studies are designed and implemented, with the express goal of evaluating the proposed extensions to the methodology. The new results are compared with the previous ones. In addition, another case study is presented, which completes the shortcomings of our previous experiments (i.e., generation of the corpus at different granularity levels and recovery of traceability links from the source code to documentation as well). A number of issues have also been explained in more detail that was not allowed in the conference venue, where the previous paper is published.

The paper is organized as follows: section **Error! Reference source not found.** gives an extended overview of related work; section 3 presents an overview of LSI (based on our earlier paper); section **Error! Reference source not found.** (re)defines the traceability link recovery process, based on the same underlying model form our previous paper; section **Error! Reference source not found.** presents each case study, their results, and analysis. In order to make this paper self-contained, the old results are also presented here; and section **Error! Reference source not found.** concludes the paper by revisiting the main results and outlines future research directions.

2. Related Work

The work presented in this paper addresses two specific issues: using information retrieval (IR) methods to support software engineering tasks and recovering source code to documentation traceability links.

2.1. IR and Software Engineering

The research that has been conducted on the specific use of applying IR methods to source code and associated documentation typically relates to indexing reusable components [Fischer'98, Frakes'87, Maarek'91, Maarek'89]. Notable is the work of Maarek [Maarek'91, Maarek'89] 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 [Etzkorn'97] 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 we hypothesis that this approach should produce high quality and low cost results.

More recently, Maletic and Marcus [Maletic'01, Maletic'99, Marcus'01] used LSI to derive similarity measures between source code components. These measures were used to cluster the source code to help in 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 software components. The work presented here extends these results in a new direction. At the same time, Antoniol et al. [Antoniol'02] investigated the use of IR methods to support the traceability recovery process. In particular, they used both a probabilistic method [Antoniol'00b, Antoniol'99] and a vector space model [Antoniol'00a] to recover links between source code and documentation, and between source code and requirements. Their results were promising in each case and support the choice of vector space models over probabilistic IR.

2.2. Traceability

Requirements traceability and its importance in software development process have been well described [Gotel'94, Watkins'94]. A number of requirements tracing tools have been developed and integrated into software development environments [Antoniol'00a, Antoniol'02, Antoniol'99, Marcus'03, Pinheiro'96, Pohl'96, Reiss'99]. Other research seeks to develop a reference model for requirements traceability that defines types of requirement documentation entities and traceability relationships [Knethen'02, Ramesh'01, Toranzo'99]. Dick [Dick'02] extends the traceability relationships to support more consistency analyses. Inconsistency management and impact analysis have been studied since the late 1980s [Spanoudakis'01]. According to Spanoukadis and Zisman [Spanoudakis'01], inconsistency management can be viewed as a process composed of six activities: detection of overlaps between software artifacts, detection of inconsistencies, diagnosis of inconsistencies, handling of inconsistencies, history tracking of inconsistency management process, and specification of an inconsistency management policy. The activities to be taken depend on the type of inconsistency being addressed [Nuseibeh'00]. The methods and techniques developed to support inconsistency management activities are based on logics [Hunter'98, van Lamsweerde'00], model checking [Chan'98, Heitmeyer'96], formal frameworks [Grundy'98, Nuseibeh'94, Sommerville'99], human-centered approaches [Cugola'96, Robinson'99, van Lamsweerde'00], and knowledge engineering [Zisman'01]. The work of Antoniol et al. also important from this point of view, since it deals with several aspects related to traceability: recover of traceability links between code and documentation [Antoniol'02], maintenance of traceability links during software evolution [Antoniol'01], and traceability between design and code in OO systems [Antoniol'00c].

The problem with many existing approaches to traceability and inconsistency management is that they are effective for only a limited portion of the development process, while having little or no support for software product fragments from other parts of the software life cycle [Strasunskas'02]. Traceability and inconsistency management support is most frequently found between representations such as formal specifications and program source code that are amenable to automated analysis. Most approaches use formal methods to encode software documents and require that software artifacts share a formal representational model such as formal specification languages, structured requirement templates, logics, or special conceptual diagrams. No support exists for managing relationships between these representations and less formal representations such as natural language design documents [Strasunskas'02]. We believe that advanced linking representation models such as Open Hypermedia Systems [Anderson'00] provide an excellent relationship management infrastructure for traceability and inconsistency management across a broad range of software document types.

3. Overview of Latent Semantic Indexing

We utilize an information retrieval method, latent semantic indexing (LSI), to drive the link recovery process. LSI [Deerwester'90, Dumais'91] is a machine-learning model that induces representations of the meaning of words by analyzing the relation between words and passages in large bodies of text. LSI has been used in applied settings with a high degree of success in areas like automatic essay grading and automatic tutoring to improve summarization skills in children. As a model, LSI's most impressive achievements have been in human language acquisition simulations and in modeling of high-level comprehension phenomena like metaphor understanding, causal inferences and judgments of similarity. For complete details on LSI see [Deerwester'90].

LSI was originally developed in the context of information retrieval as a way of overcoming problems with polysemy and synonymy that occurred with vector space model (VSM) [Salton'83] approaches. Some words appear in the same contexts (synonyms) and an important part of word usage patterns is blurred by accidental and inessential information. The method used by LSI to capture the essential semantic information is dimension reduction, selecting the most important dimensions from a co-occurrence matrix decomposed using singular value decomposition (SVD). As a result, LSI offers a way of assessing semantic similarity between any two samples of text in an automatic, unsupervised way.

There is a wide variety of information retrieval methods. Traditional approaches [Faloutsos'95, Salton'89] 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. LSI also has an underlying vector space model.

3.1. The Vector Space Model

The vector space model (VSM) [Salton'83] 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.

3.2. LSI and Singular Value Decomposition

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) [Salton'83] 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 to lengthy to be presented here. The interested reader is referred to [Salton'83] for details. Intuitively, in SVD a rectangular matrix **X** is decomposed into the product of three other matrices. One component matrix (**U**) describes the original row entities as vectors of derived orthogonal factor values, another (**V**) describes the original column entities in the same way, and the third is a diagonal matrix (Σ) containing scaling values such that when the three components are matrix-multiplied, the original matrix is reconstructed (i.e., $\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^{\mathbf{T}}$). The columns of **U** and **V** are the left and right singular vectors, respectively, corresponding to the monotonically decreasing (in value) diagonal elements of Σ which are called the singular values of the matrix **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 **k** columns of the **U** and **V** matrices and the first (largest) **k** singular values of **X** are used to construct a rank-**k** approximation to **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 $\mathbf{U^T U} = \mathbf{V^T V} = \mathbf{I_r}$, 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.

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.

3.3. Advantages of using 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 [Deerwester'90, Landauer'98] 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 also 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, making it difficult 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 [Berry'95]. 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 [Etzkorn'97] that also lends itself to the IR methods utilized. This makes automation drastically easier and directly supports programmer defined variable names having implied meanings but not found in the English language vocabulary (e.g., avg). 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.

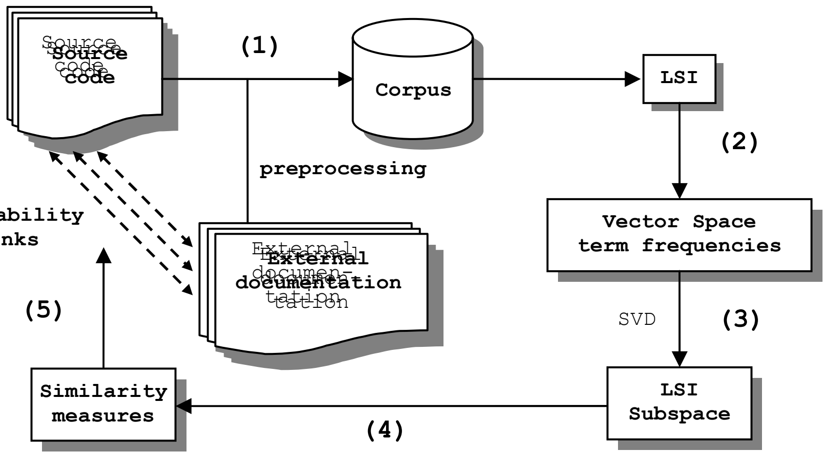

Figure 1: The traceability recovery process. There are five phases in the process: corpus generation (1), LSI subspace generation (2 and 3), computation of similarity measure (4), and recovery of traceability links (5).

4. The Traceability Recovery Process

Our traceability link recovery process (see Figure 1) has five steps and is partially automated: corpus generation (1), LSI subspace generation (2 and 3), computation of similarity measure (4), and recovery of traceability links (5). The user is involved in the process in phases 1 and 5. The degree of user involvement depends on the type of source code and the user's task. The entire process is organized in a pipeline architecture; the output from one phase constitutes the input for the next phase. In a first step, the external documentation and the source code are used to create a corpus that is used to generate the semantic space for information retrieval. This part is largely automated and the user is only involved in selecting the granularity of the documents that will compose the corpus. Details of this phase are given in the next section of the paper.

The semantic space, named the LSI subspace, is automatically generated in phases (2) and (3). The only involvement of the user at this point is the selection of the dimensionality reduction that SVD will generate. This step is based on the LSI mechanism described in the previous section.

Once the LSI subspace is constructed, each part of the documentation and source code component will be represented as a vector in this space. Based on this representation, a semantic similarity measure is defined (see section 4.2). The measure is used to identify elements of the source code that relate closely to a given part of the documentation or vice-versa. The granularity used here for source code components is the one defined in the first phase of the process. The similarity measures are automatically computed while the user is involved in selecting the appropriate pairs (or groups) of documents that correspond to traceability links. Phase five of the process consists of the selection of the traceability links.

4.1. First Step - 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 [Maletic'01, Marcus'01] and class declarations in OO source code [Maletic'99]. The goal there was to cluster elements of the source code based on semantic similarity, rather than mapping them to documentation. In other cases, we used source code files as granularity for documents [Marcus'03].

In the traceability link recovery process, different granularities may be of interest. 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.). Two approaches were investigated and implemented as part of the process. In one of them, files granularity level is used – each file is defined as a document in the corpus. 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. However, our experience [Marcus'03] shows that the results are still relatively good in this situation. It is a trade-off we are willing to take in favor of simplicity and low-cost of the preprocessing. This approach is also programming language independent.

If this situation is unacceptable for the user, a different granularity is available – class level. Classes will correspond to one document in the corpus. Once again, some classes may be too large, in which case they can broken into several smaller documents. This approach requires a more complex parsing of the source code in order to determine where classes start (definition) and end (implementation). This approach is of course programming language dependent. We implemented a simple parser that identifies class definitions and implementations for C++ and Java. It is fairly easy to extend the system to support other similar type programming languages.

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). The same decomposition is used in both approaches.

In each case, 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 "traceability" and "link": "traceability_link", "Traceability_link", "traceability_Link", "Traceability_Link", "TraceabilityLink", "TRACEABILITYLink". This step can be customized and the users can defined their own identifier format that needs to be split, based on regular expressions. 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 LSI does not use a predefined vocabulary, or a predefined grammar, therefore no morphological analysis or transformations are required. Thus parsing of the source code is very minimal.

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 [Etzkorn'97] 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 elements of the 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 at random.

4.2. Fourth Step - Defining the Semantic Similarity Measure

Before we give a detailed explanation of this and following steps of the process, some mathematical background and definitions are necessary. This aligns our formal definition with the notation introduced in section 3.

Notation. A bold lowercase letter (e.g., $\mathbf{y}$) denotes a *vector*. A vector is equivalent to a matrix having a single column. The ith 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 ith 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 $\mathbf{X} = \text{diag}(\mathbf{X}_{[l,l]}, \mathbf{X}_{[2,2]}, \ldots, \mathbf{X}_{[n,n]})$.

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 $\mathbf{X} \in \boldsymbol{R}^{mxn}$, $\mathbf{XI}_n = \mathbf{I}_m\mathbf{X} = \mathbf{X}$. 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

$$|x|_2 = \sqrt{X^T X} = \sqrt{\sum_{i=1}^{m} (x[i])^2}$$

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

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

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 $E = \{e_1, e_2, ..., e_m\}$. The total number of documents in the documentation is $m = |E|$.

Definition. The source code is also a set of documents $D = \{d_1, d_2, ..., d_m\}$. The total number of documents in the documentation is $n = |D|$.

Definition. A *software system* is a set of documents (source code and external) $S = D \cup E = \{d_1, d_2, ..., d_n\} \cup \{e_1, e_2, ..., e_m\}$. 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 $S = \{d_1, d_2, ..., d_n, e_1, e_2, ..., e_m\}$ 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 $\mathbf{X} \in \boldsymbol{R}^{\mathrm{vxn}}$ 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 $\mathrm{sim}(d_i, d_j) = \cos(\mathbf{x_i}, \mathbf{y_i})$ The value of the measure will be between [-1, 1] with value (almost) 1 representing that the two are (almost) identical.

4.3. Fifth Step - Recovering Traceability Links

In this step of the process, the similarities between each pair of documents from $E \times D$ are computed and ranked. The user has the option of retrieving traceability links starting from the documentation or the source code. For a given external document e_i (also termed as a query document), the system will return the most similar source code document d_i, based on the $\mathrm{sim}(e_i, d_i)$ measure. It is the user's task to verify the validity of the link suggested by the system. In some cases, part of the documentation may refer to more than one source code document, or a source code document may be described by more than one external document. In such cases, the user needs to investigate the next suggested document. Since the process is only partially automated, the stopping criterion is defined by the user, once all the links relevant to a query document are retrieved. The process can be used on a single query document or multiple ones, essentially for the entire system under analysis.

To operate at system level, the user has two options. One is to determine a threshold ε for the similarity measure that identifies which documents are considered "linked". In other words among all the pairs from $E \times D$, only those will be retrieved that have a similarity measure greater than ε. The threshold is determined empirically and varies from corpus to corpus. 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. Many IR methods (especially search engines) use this approach, where the retrieved documents are ranked by the "relevancy" to a query.

The alternative option for the user is simply to retrieve the top θ ranked links for each document, where $\theta \in \{1, 2, 3, \ldots, n\}$. In this case, a threshold on the number of recovered links, regardless of the actual value of the similarity measure, is imposed. This approach is preferred by Antoniol et al [Antoniol'02] 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 allow the system to suggest few false positives. Too high of a threshold will result in missing relevant links. Using the ranking method, the user will retrieve more relevant links at the expense of yielding more false positives.

5. Case Studies

A set of case studies was designed and executed to evaluate the proposed methodology, centered on LSI. The case studies are designed such that we can compare the results with related approaches proposed by Antoniol et al. [Antoniol'02]. 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. The remainder of the section describes the case studies and the obtained results.

5.1. Evaluation of the Results

In order to compare the results with the methods proposed by Anotniol et al., two of the most common measures for the quality of the results in experiments with IR methods were used: *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:

$$recall = \frac{C_i}{R_i} = \frac{\#correct \wedge retrieved}{\#correct}\%$$

$$precision = \frac{C_i}{N_i} = \frac{\#correct \wedge retrieved}{\#retrieved}\%$$

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:

$$recall = \frac{\sum_{i=n+1}^{m} C_i}{\sum_{i=n+1}^{m} R_i}\% \qquad precision = \frac{\sum_{i=n+1}^{m} C_i}{\sum_{i=n+1}^{m} N_i}\%$$

For each of the case studies presented in the following sections, the recall and precision of the results were directly compared with those obtained by Antoniol et al.

5.2. First Case Study - Recovery of Traceability Links from Documentation to Source Code in LEDA, using File Level Document Granularity

The first case study is aimed at assessing LSI with respect to the other IR methods used by Antoniol. With that in mind, we chose to build a large corpus, with minimal preprocessing, in order to simplify the process.

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 vocabulary determined by LSI.

Table 1. Elements of the LEDA corpus in the first case study

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	-

We used 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 (i.e., 2.1 through 11.5) that were used in Antoniol's experiments.

In this first case study we chose to recover the traceability links from documentation to the source code. The chosen granularity for the source code is at file level (i.e., each document is a file form LEDA). Some larger files were broken into smaller parts (see Table 1) in order to generate uniform size documents. No parsing of the source code is done in this case study. However, it is interesting to note that there are many classes in this version of LEDA that have inline implementation or they are internal classes to other ones. This explains why 219 classes are implemented in only 218 files. Since we chose file as document granularity, it was logical to map documents from the manual to source code. A typical query is then to find out which parts of the source code are described by a given manual section.

One more consideration determined our choice. Since the chosen granularity does not require parsing of the code, it 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, 7 classes in three files, and 1 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 our method.

Table 2. Recovered links, recall, and precision using the cosine value threshold for LEDA, with file level document granularity, starting from the manual sections.

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

As presented in section 4.3, the system can be used in different ways to suggest pair of documents to the user, which correspond to traceability links. One is to use a threshold based on the value of the similarity measure and consider that a pair of documents determine a 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 traceability links all the top ranked pairs down to the cut point.

Table 2 summarizes the results we obtained on recovering the traceability links between the LEDA manual pages and source code. The first column (Cosine threshold) represents the threshold value; column 2 represents the number of correct links recovered; column 3 represents the number of incorrect links recovered; column 4 (Missed links) represents the number of correct links that were not recovered; column 5 represents the total number of recovered links (correct + incorrect); and the last two columns is 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. A threshold around 0.65 yields approximately equal precision and recall.

Table 3. Recovered links, recall, and precision using the cut point approach for LEDA, with file level document granularity, starting from the manual sections.

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 %

To further validate the data, we repeated the experiment using a cut point for the best-ranked pairs of documents, as done by Antoniol et al. [Antoniol'02]. 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 that the threshold method would give better precision.

Just as in their case study, we used as many number of cut points as necessary to obtain 100% recall. Figure 2 shows the precision and recall between the two sets of experiments. The values used for Antoniol's experiments are the better they found 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). In particular, the better precision is due to the fact that LSI is able do deal with all the comments and identifier names included in the corpus. In contrast, with Antoniol's method, many identifiers and comments that were not grammatically and lexically correct were not used.

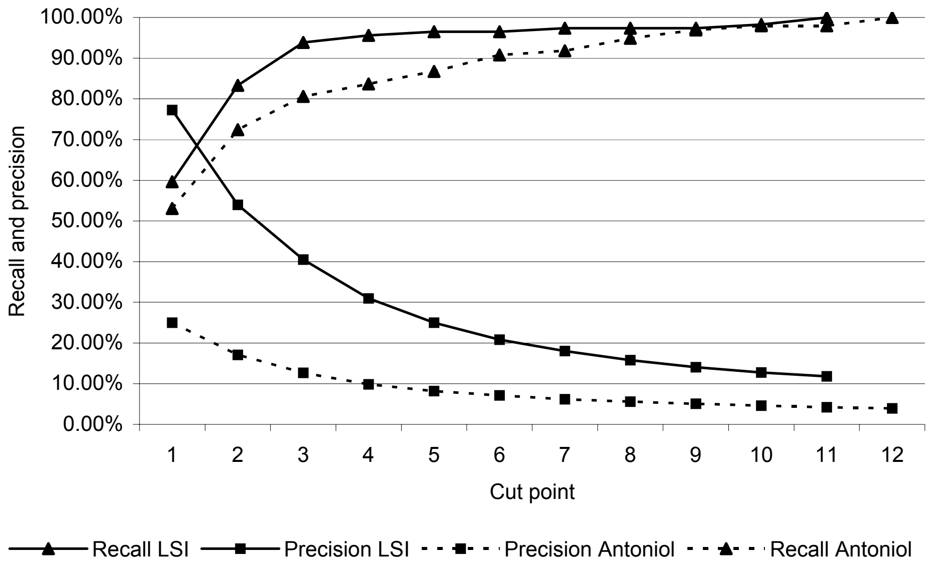

Figure 2. Recall and precision values for experiment by Antoniol and experiments with LSI using LEDA. The x-axis represents the cut point and the y-axis represents recall/precision values

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 seven of the correct links are recovered after selecting the top three ranked pairs of documents. More than that, all but three of the correct links are recovered after selecting the top seven ranked pairs of documents. We looked closer to the remaining three 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 sections 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 then the files which implement the class. Even more, these particular classes are basic types, ubiquitously used throughout the LEDA package.

5.3. Second Case Study - Recovery of Traceability Links from Documentation to Source Code in LEDA, using Class Level Document Granularity

The second case study is done on the same software package (i.e., LEDA). In this case study we rebuilt the corpus such that the source code documents reflect the classes in LEDA. The goal of the case study is to see how much the corpus definition is influencing the results. In addition, we used all three methods for link recovery and compared the results. We followed the same steps of the process as before. Table 4 summarizes the elements of the LEDA corpus, built for this case study.

Table 4. Elements of the LEDA corpus in the second case study

LEDA 3.4	Count	Documents
Source code files	491	140
Manual sections	88	88
Total # of documents		228
Vocabulary	2347	-

As mentioned previously, in the LEDA source code, many classes are implemented inline, are internal classes to other, or inherited and implemented in the same file. We did not separate these groups of related classes and kept them in the same documents. Thus we obtained 140 documents corresponding to the 219 classes. Among these, 84 documents contain one class and 56 contain more than one class (i.e., 2, 3, or 4 maximum). For example, groups of classes such as *bin_heap* and *bin_heap_elem*, *ch_array* and *ch_array_elem*, or *skiplist* and *skiplist_node*, etc. are in the same documents respectively. This time we did not split the large documents. Also, we only included the 88 documents corresponding to the manual sections used in the first case study (i.e., sections 2.1 through 11.5). While, starting from the same source code and manual, the corpus has quite different characteristics than in the previous case. For example, the vocabulary is smaller since we did not use the demo files and all the manual sections.

As explained in the previous experiment, 8 of the 88 manual sections did not relate specifically to any one class, so there are 80 pairs (manual section, class) we needed to recover. With this new corpus, we used the system to recover the traceability links from the documentation to the source code (as before) in all three possible ways. First, we use a threshold on the similarity measure starting at 0.7 and decreased it by 0.1 for each step. For each manual section the system returned all the source code documents that have a similarity measure to the manual section larger than the threshold. The user stopped when all the links are retrieved (100% recall). Table 5 summarizes the results in this case. The structure of the table is the same as for Table 2 in section 5.2. We had to lower the threshold from 0.7 to 0.3 in five steps to reach 100% recall.

Table 5. Recovered links, recall, and precision using the cosine value threshold for LEDA, with class level document granularity, starting from the manual sections.

Cosine	Correct links retrieved	Incorrect links retrieved	Missed links	Total links recovered	Precision	Recall
0.70	23	1	57	24	95.83%	28.75%
0.60	55	22	25	77	71.42%	68.75%
0.50	69	69	11	180	38.33%	86.25%
0.40	76	337	4	413	18.40%	95.00%
0.30	80	757	0	837	9.55%	100.00%

The next experiment in the case study is to recover the same links using the ranked pairs of documents, establishing a cut point for each step, as we did in the previous case study. Table 6 summarizes the results in this case. The structure of the table is the same as for Table 3 in section 5.2. It took 5 steps to reach 100% recall.

Table 6. Recovered links, recall, and precision using the cut point approach for LEDA, with class level document granularity, starting from the manual sections.

Cut point	Correct links retrieved	Incorrect links retrieved	Missed links	Total links recovered	Precision	Recall
1	71	8	9	88	80.68%	88.75%
2	75	96	5	176	42.61%	93.75%
3	77	184	3	264	29.17%	96.25%
4	79	272	1	352	22.44%	98.75%
5	80	360	0	440	18.18%	100.00%

Finally, we used the combined approach with the cut point and the 0.3 threshold for the similarity measure. In other words, we retrieved on pair in each step only of the similarity measure was higher than 0.3. Table 7 summarizes the results in this case. The structure of the table is the same as for Table 3 in section 5.2. It took 5 steps to reach 100% recall. Figure 3 shows the recall and precision values for each of the three approaches.

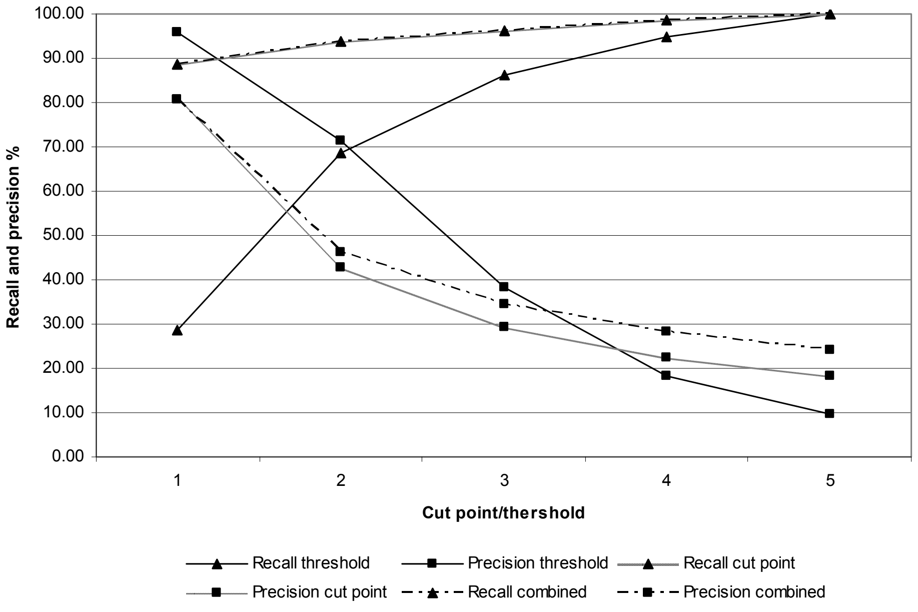

Figure 3. Recall and precision values for the recovery of traceability links for LEDA from manual documents to source code. All the three methods are represented: using a threshold, using a cut point, and combined. For the cut point and combined methods the recall values are the same (lines overlap).

Table 7. Recovered links, recall, and precision using the combined approach with cut point and 0.3 threshold for LEDA, with class level document granularity, starting from the manual sections.

Cut point	Correct links retrieved	Incorrect links retrieved	Missed links	Total links recovered	Precision	Recall
1	71	8	9	88	80.68%	88.75%
2	75	82	5	162	46.29%	93.75%
3	77	144	3	224	34.37%	96.25%
4	79	200	1	280	28.21%	98.75%
5	80	253	0	333	24.02%	100.00%

The results largely confirmed our hypothesis, based on the type of corpus we built from LEDA. Since the manual is in part generated from the documentation, we expected that the recall and precision curves for the threshold value (solid lines) will intersect for a relative high threshold value (i.e., 0.6) with good results (i.e., about 70% recall and precision). On the other hand, we expected to reach 100% recall with a higher precision.

As expected the best result (highest precision for 100% recall) is given by the combined approach, using both a threshold and the cut point. More importantly, the cut point and the combined methods gave better results than those obtained with the previous corpus (see Figure 2). This is a clear indication that it is worth performing a little extra source code parsing (yet still quite simple) to obtain a document decomposition that better reflects the source code decomposition (i.e., classes).

5.4. Third Case Study - Recovery of Traceability Links from Source Code to Documentation in LEDA, using Class Level Document Granularity

With the same corpus as in the previous case study (i.e., class level granularity) we could perform another one, in which we recovered traceability links form the source code to the documentation. This is the same way Antoniol et al did [Antoniol'02] and we could compare the results best. The same steps of the process were followed and we used the cut point method to recover the links. Starting from the 140 documents representing the LEDA classes we retrieved in each step one document from the manual sections. Table 8 summarizes the results in this case. The structure of the table is the same as for Table 3 in section 5.2. It took XXX steps to reach 100% recall.

Table 8. Recovered links, recall, and precision using the cut point approach for LEDA, with class level document granularity, starting from the source code.

Cut point	Correct links retrieved	Incorrect links retrieved	Missed links	Total links recovered	Precision	Recall
1						
2						
3						
4						
5						

We need to fill in this table and discuss the results. Andrey is supposed to get the data by 2:30 pm.

5.5. Fourth Case Study - Recovery of Traceability Links in Albergate

The LSI based method is versatile enough to accommodate different languages with minor or no modifications to the process and tools. Since the LEDA manual was largely generated, another case study was done on a different software, with different kind of documentation available.

For the next case study, 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 [Antoniol'00a, Antoniol'02] 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. In other words, the setup for this experiment is almost identical to the one described in [Antoniol'00a, Antoniol'02], since a document in our space corresponds to a class (in most cases). One more thing to note is that the Albergate source code contains less than 300 lines of internal documentation (i.e., comments).

Note that all the documentation is written in Italian. Our process was essentially unchanged due to the fact our method is not dependent on a language or grammar.

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 (much better represented in the source code). In addition, the requirement documents are purported 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 (using SVD). When the corpus is small with terms and concepts distributed scarcely throughout the LSI subspace, 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 9 summarizes the results of the traceability recovery process for Albergate. The structure of the table is the same as Table 3, described in section 5.2. Confirming our hypothesis, the initial precision was lower, however the 100% recall target was reached faster than in the case of LEDA and with better precision. The explanation is that, unlike in the LEDA case, the coupling between classes is less intensive in Albergate.

Table 9. Recovered links, recall, and precision using the cut point approach for Albergate, with class level document granularity, starting from the source code.

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 %

Figure 4 shows graphically how our results compare with those obtained by Antoniol et al [Antoniol'00a, Antoniol'00b, '02]. This time the setup of the experiments and the benchmark mapping are almost identical. The results are very similar and the only significant difference is that 100% recall is reached one step sooner (selecting the top 6 ranked pairs, rather than 7) with LSI.

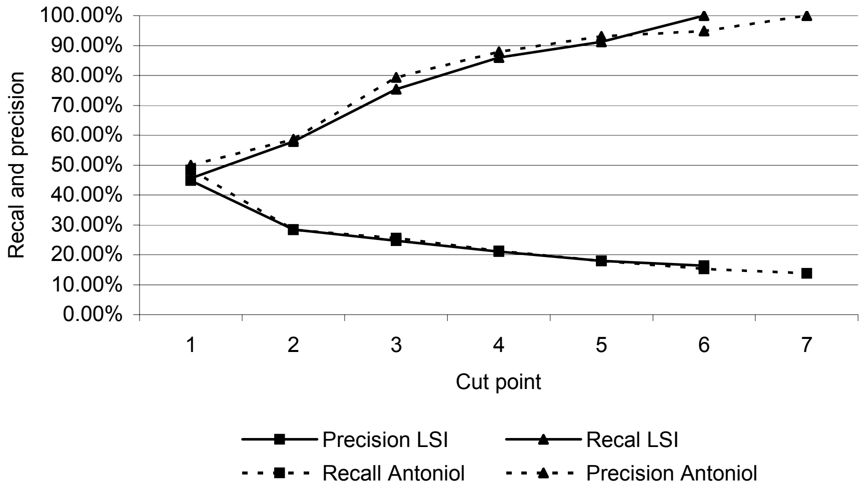

Figure 4. Recall and precision values for experiment by Antoniol and experiments with LSI using Albergate. The x-axis represents the cut point and The y-axis represents recall/precision values

The LSI-based method performed just as well as the other IR methods (from the recall and precision point of view). The major difference that needs to be reiterated is that, since LSI does not need a predefined vocabulary or grammar, we did not need to use any additional tools when migrating from C++ to Java and English to Italian, respectively.

6. 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 case studies is presented and the results analyzed by comparing them with previous related research by Antoniol et al [Antoniol'02]. The case studies are designed to assess the use of LSI as the underlying technology for traceability link recovery versus other IR methods, used by Antoniol et al. In addition, the results of the different case studies provide insight into the better techniques for build the corpus.

The results show that the method using LSI performs better than Antoniol's methods using probabilistic and vector space model-based IR methods combined with full parsing of the source code and morphological analysis of the documentation.

Using LSI requires less preprocessing of the source code and documentation and implicitly less computation. It is entirely domain independent with respect to natural language, programming language, and programming paradigm, therefore it is more flexible and better suited for automation. These characteristics allow us to use internal documentation in the analysis (not used by Antoniol), which allows LSI to produce better results. The Albergate case supports this hypothesis. With almost no comments in the source code, LSI does perform at least as well as the other methods.

The case studies also highlight the importance of building the corpus in such a way that it reflects the original source decomposition. While it requires more processing, the results of the link recovery process are also better. Building the corpus is a one-time expense in the process and it is done automatically. Once the corpus is generated and the LSI space is built, the subsequent steps in the process are computationally fast, allowing software engineers to use the system in real-time. With this in mind, we plan to incorporate the system into existing development environments, such as Eclipse or Microsoft Studio .NET. Thus, the proposed methodology can be used during development to help improve the quality of the newly created (internal or external) documentation, such that it will preserve existing traceability links while creating new ones that are unambiguous.

7. Acknowledgements

We greatly appreciate and thank Giuliano Antoniol and Massimiliano DiPenta for sharing their results and experience. We also thank Denys Poshyvanyk for his support in verifying the results. This work was supported in part by a grant from the National Science Foundation (CCR-02-04175).

References

[Anderson'00] Anderson, K. M., Taylor, R. N., and Whitehead, E. J. J., (2000), "Chimera: hypermedia for heterogeneous software development enviroments", *ACM Transactions on Information Systems*, vol. 18, no. 3, pp. 211-245.

[Anquetil'98a] Anquetil, N. and Lethbridge, T., (1998a), "Assessing the Relevance of Identifier Names in a Legacy Software System", in Proceedings of Annual IBM

Centers for Advanced Studies Conference (CASCON'98), December, pp. 213-222.

[Anquetil'98b] Anquetil, N. and Lethbridge, T., (1998b), "Extracting Concepts from File Names; a New File Clustering Criterion", in Proceedings of 20th International Conference on Software Engineering (ICSE'98), Kyoto, Japan, pp. 84-93.

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

[Antoniol'01] Antoniol, G., Canfora, G., Casazza, G., and De Lucia, A., (2001), "Maintaining Traceability Links During Object-Oriented Software Evolution", *Software - Practice and Experience*, vol. 31, no. 4, April, pp. 331-355.

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

[Antoniol'02] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E., (2002), "Recovering Traceability Links between Code and Documentation", *IEEE Transactions on Software Engineering*, vol. 28, no. 10, October, pp. 970 - 983.

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

[Antoniol'00c] Antoniol, G., Caprile, B., Potrich, A., and Tonella, P., (2000c), "Design-Code Traceability for Object Oriented Systems", *Annals of Software Engineering*, vol. 9, no. 1/4, pp. 35-58.

[Berry'95] Berry, M. W., Dumais, S. T., and O'Brien, G. W., (1995), "Using Linear Algebra for Intelligent Information Retrieval", *SIAM: Review*, vol. 37, no. 4, pp. 573-595.

[Chan'98] Chan, W., Anderson, R., Beame, P., Burns, S., Modugno, F., Notkin, D., and Reese, J., (1998), "Model checking large software specifications", *IEEE Transactions on Software Engineering*, vol. 24, no. 7, pp. 498-520.

[Cugola'96] Cugola, G., Nitto, E. D., Fugetta, A., and Ghezzi, C., (1996), "A framework for formalizing inconsistencies and deviations in human-centered systems", *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 3, pp. 191-230.

[Deerwester'90] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R., (1990), "Indexing by Latent Semantic Analysis", *Journal of the American Society for Information Science*, vol. 41, pp. 391-407.

[Dick'02] Dick, J., (2002), "Rich traceability", in Proceedings of Automated Software Engineering, Edinburgh, Scotland.

[Dumais'91] Dumais, S. T., (1991), "Improving the retrieval of information from external sources", *Behavior Research Methods, Instruments, and Computers*, vol. 23, no. 2, pp. 229 - 236.

[Etzkorn'97] Etzkorn, L. H. and Davis, C. G., (1997), "Automatically Identifying Reusable OO Legacy Code", *IEEE Computer*, vol. 30, no. 10, October, pp. 66-72.

[Faloutsos'95] Faloutsos, C. and Oard, D. W., (1995), "A Survey of Information Retrieval and Filtering Methods": University of Maryland.

[Fischer'98] Fischer, B., (1998), "Specification-Based Browsing of Software Component Libraries", in Proceedings of 13th ASE, pp. 74-83.

[Frakes'87] Frakes, W., (1987), "Software Reuse Through Information Retrieval", in Proceedings of 20th Annual HICSS, Kona, HI, Jan., pp. 530-535.

[Gotel'94] Gotel, O. and Ginkelstein, A., (1994), "An analysis of the requirement traceability problem", in Proceedings of International Conference on Requirements Engineering, Colorado Springs, Colorado, pp. 94-102.

[Grundy'98] Grundy, J., Hosking, J., and Mugridge, W., (1998), "Inconsistency management for multiple-view software development environments", *IEEE Transactions on Software Engineering*, vol. 24, no. 11, pp. 960-981.

[Heitmeyer'96] Heitmeyer, C. L., Jeffords, R. D., and Labaw, B. G., (1996), "Automated consistency checking of requirements specifications", *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 3, pp. 231-261.

[Hunter'98] Hunter, A. and Nuseibeh, B., (1998), "Managing inconsistent specifications: reasoning, analysis, and action", *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 4, pp. 335-367.

[Knethen'02] Knethen, A., (2002), "Automatic change support based on a trace model", in Proceedings of Automated Software Engineering, Edinburgh, Scotland.

[Landauer'98] Landauer, T. K., Foltz, P. W., and Laham, D., (1998), "An Introduction to Latent Semantic Analysis", *Discourse Processes*, vol. 25, no. 2&3, pp. 259-284.

[Maarek'91] Maarek, Y. S., Berry, D. M., and Kaiser, G. E., (1991), "An Information Retrieval Approach for Automatically Constructing Software Libraries", *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pp. 800-813.

[Maarek'89] Maarek, Y. S. and Smadja, F. A., (1989), "Full Text Indexing Based on Lexical Relations, an Application: Software Libraries", in Proceedings of SIGIR89, Cambridge, MA, June, pp. 198-206.

[Maletic'01] Maletic, J. I. and Marcus, A., (2001), "Supporting Program Comprehension Using Semantic and Structural Information", in Proceedings of 23rd International Conference on Software Engineering (ICSE'01), Toronto, Ontario, Canada, May 12-19, pp. 103-112.

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

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

[Marcus'03] Marcus, A. and Maletic, J. I., (2003), "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing", in Proceedings of 25th IEEE/ACM International Conference on Software Engineering (ICSE'03), Portland, OR, May 3-10, pp. 125-137.

[Nuseibeh'00] Nuseibeh, B., Easterbrook, S., and Russo, A., (2000), "Leveraging inconsistency in software development", *IEEE Computer*, vol. 33, no. 4, pp. 24-29.

[Nuseibeh'94] Nuseibeh, B., Kramer, J., and Finkelstein, A., (1994), "A framework for expressing the relationships between multiple views in requirements specification", *IEEE Transactions on SoftwareEngineering*, vol. 20, no. 10, pp. 760-773.

[Pinheiro'96] Pinheiro, F. and Goguen, J., (1996), "An Object-Oriented Tool for Tracing Requirements", *IEEE Software*, vol. 13, no. 2, pp. 52-64.

[Pohl'96] Pohl, K., (1996), "PRO-ART: Enabling requirements pre-traceability", in Proceedings of International Conference on Requirements Engineering, Colorado Springs, Colorado, pp. 76-85.

[Ramesh'01] Ramesh, B. and Jarke, M., (2001), "Toward reference model for requirements traceability", *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 58-93.

[Reiss'99] Reiss, S., (1999), "The Desert environment", *ACM Transactions on Software Engineering and Methodology*, vol. 8, no. 4, pp. 297-342.

[Robinson'99] Robinson, W. and Pawlowski, S., (1999), "Managing requirements inconsistency with development goal monitors", *IEEE Transactions on Software Engineering*, vol. 25, no. 6, pp. 816-835.

[Salton'89] Salton, G., (1989), *Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer*, Addison-Wesley.

[Salton'83] Salton, G. and McGill, M., (1983), *Introduction to Modern Information Retrival*, McGraw-Hill.

[Sommerville'99] Sommerville, I., Sawyer, P., and Viller, S., (1999), "Managing process inconsistency using viewpoints", *IEEE Transactions on Software Engineering*, vol. 25, no. 6, pp. 784-799.

[Spanoudakis'01] Spanoudakis, G. and Zisman, A., (2001), "Inconsistency management in software engineering: Survey and open research issues", in *Handbook of Software Engineering and Knowledge Engineering*, S. K. Chang, Ed., pp. 24-29.

[Strasunskas'02] Strasunskas, D., (2002), "Traceability in collaborative systems development from lifecycle perspective - position paper", in Proceedings of Automated Software Engineering, Edinburgh, Scotland.

[Tjortjis'03] Tjortjis, C., Sinos, L., and Layzell, P. J., (2003), "Facilitating Program Comprehension by Mining Association Rules from Source Code", in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, May 10-11, pp. 125-133.

[Toranzo'99] Toranzo, M. and Castro, J., (1999), "A comprehensive traceability model to support the design of interactive systems", in Proceedings of ECOOP Workshops, pp. 283-284.

[van Lamsweerde'00] van Lamsweerde, A. and Letier, E., (2000), "Handling obstacles in goal-oriented requirements engineering", *IEEE Transactions on Software Engineering*, vol. 26, no. 10, pp. 978-1005.

[Watkins'94] Watkins, R. and Neal, M., (1994), "Why and how of requirements tracing", *IEEE Software*, vol. 11, no. 4, pp. 104-106.

[Zisman'01] Zisman, A. and Kozlenkov, A., (2001), "Knowledge-based approach to consistency management of UML specifications", in Proceedings of Automated Software Engineering, San Diego, California.