

# Using Latent Semantic Analysis to Identify Similarities in Source Code to Support Program Understanding

Jonathan I. Maletic, Andrian Marcus  
Division of Computer Science  
The Department of Mathematical Sciences  
The University of Memphis  
Campus Box 526429 Memphis TN 38152  
[jmaletic@memphis.edu](mailto:jmaletic@memphis.edu), [amarcus@memphis.edu](mailto:amarcus@memphis.edu)

## Abstract

*The paper describes the results of applying Latent Semantic Analysis (LSA), an advanced information retrieval method, to program source code and associated documentation. Latent Semantic Analysis is a corpus-based statistical method for inducing and representing aspects of the meanings of words and passages (of natural language) reflective in their usage. This methodology is assessed for application to the domain of software components (i.e., source code and its accompanying documentation). Here LSA is used as the basis to cluster software components. This clustering is used to assist in the understanding of a nontrivial software system, namely a version of Mosaic. Applying Latent Semantic Analysis to the domain of source code and internal documentation for the support of program understanding is a new application of this method and a departure from the normal application domain of natural language.*

## 1. Introduction

The tasks of maintenance and reengineering of an existing software system require a great deal of effort to be spent on understanding the source code to determine the behavior, organization, and architecture of the software not reflected in documentation. The software engineer must examine both the structural aspect of the source code (e.g., programming language syntax) and the nature of the problem domain (e.g., comments, documentation, and variable names) to extract the information needed to fully understand any part of the system [3, 8, 14, 18, 19]. In the research presented here, the later aspect is being examined and tools to help automate this part of the understanding process are being investigated. Experiments using an advanced information retrieval technique, Latent Semantic Analysis (LSA), to identify similarities between

pieces of source code are being conducted. The objective of this research is to determine how well such a method can be used to support aspects of program understanding, comprehension, and reengineering of software systems.

Latent Semantic Analysis (LSA) [1, 12] is a corpus-based statistical method for inducing and representing aspects of the meanings of words and passages (of natural language) reflective in their usage. The method generates a real valued vector description for documents of text. This representation can be used to compare and index documents using a variety of similarity measures. By applying LSA to source code and its associated internal documentation (i.e., comments), candidate components can be compared with respect to these similarity measures. A number of metrics are defined based on these similarity measures to help support program understanding.

Results have shown [1, 12] that LSA captures significant portions of the meaning not only of individual words but also of whole passages such as sentences, paragraphs, and short essays. Basically, the central concept of LSA 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.

This research is attempting to address some of the following issues: Does LSA apply equally well to the domain of source code and internal documentation? What is the best granularity of the software documents/components for use with LSA? Can LSA be utilized to help support reverse engineering and program understanding? The following section gives a brief overview of different approaches to information retrieval. A detailed description of LSA is then given along with some of the reasoning behind choosing LSA. The results of applying this method to a reasonable sized software system (Mosaic) are then presented. The measures derived

by LSA are used to cluster the source code (at a function level) into semantically similar groups. Examples of how this clustering helps support the program understanding process are also given.

## 2. The LSA Model

There are a variety of information retrieval methods including traditional [9] approaches such as signature files, inversion, and clustering. Other methods that try to capture more information about documents to achieve better performance include those using parsing, syntactic information, and natural language processing techniques; methods using neural networks; and Latent Semantic Analysis (also referred to as Latent Semantic Indexing).

LSA relies on a Single Value Decomposition (SVD) [17, 20] of a matrix (word  $\times$  context) derived from a corpus of natural text that pertains to knowledge in the particular domain of interest. SVD is a form of factor analysis and acts as a method for reducing the dimensionality of a feature space without serious loss of specificity. Typically, the word by context matrix is very large and (quite often) sparse. SVD reduces the number of dimensions without great loss of descriptiveness. Single value decomposition is the underlying operation in a number of applications including statistical principal component analysis [10], text retrieval [2, 6], pattern recognition and dimensionality reduction [5], and natural language understanding [11, 12].

Latent Semantic Analysis is comprised of four steps [4, 12]:

1. A large body of text is represented as an occurrence matrix ( $i \times j$ ) in which rows stand for individual word types, columns for meaning bearing passages such as sentence or paragraphs (granularity is based on problem or data), that is (word  $\times$  context). Each cell then contains the frequency with which a word occurs in a passage.
2. Cell entries  $freq_{i,j}$  are transformed to:

$$\frac{\log(freq_{i,j} + 1)}{\sum_{1-j} \left( \left( \frac{freq_{i,j}}{\sum_{1-j} freq_{i,j}} \right) * \log \left( \frac{freq_{i,j}}{\sum_{1-j} freq_{i,j}} \right) \right)}$$

a measure of the first order association of a word and its context.

3. The matrix is then subject to Singular Value Decomposition (SVD) [6, 10, 17, 20]:

$[ij] = [ik] [kk] [jk]$  where  $[ij]$  is the occurrence matrix,  $[ik]$  and  $[jk]$  have orthonormal columns,  $[kk]$  is a diagonal matrix of singular value where

$k \leq \max(i,j)$ . In SVD, a rectangular matrix is decomposed into the product of three other matrices. One component matrix describes the original row entities as vectors of derived orthogonal factor values; another describes the original column entities in the same way. The third is a diagonal matrix containing scaling values such that when the three components are matrix multiplied, the original matrix is reconstructed.

4. Finally, all but the  $d$  largest singular values are set to zero. Pre-multiplication of the right-hand matrices produces a least squares best approximation to the original matrix given the number of dimensions,  $d$ , that are retained. The SVD with dimension reduction constitutes a constraint satisfaction induction process in that it predicts the original observations on the basis of linear relations among the abstracted representations of the data that it has retained.

The result is that each word is represented as a vector of length  $d$ . Performance depends strongly on the choice of the number of dimensions. The optimal number is typically around between 200 and 300 and may vary from corpus to corpus, domain to domain. The similarity of any two words, any two text passages, or any word and any text passage, are computed by measures on their vectors. Often the cosine of the contained angle between the vectors in  $d$ -space is used as the degree of qualitative similarity of meaning. The length of vectors is also useful as a measure.

## 3. Advantages of Using LSA

A fundamental deficiency of the many other IR methods is that they fail to deal properly with two major issues: synonymy and polysemy. Synonymy is used in a very general sense to describe the fact that there are many ways to refer to the same object. People in different contexts, with different knowledge, or linguistic habits will describe the same information using different terms. Polysemy refers to the general fact that most words have more than one distinct meaning. In different contexts or when used by different people the same term takes on varying referential significance [4]. Although software developers tend to use standard terms for the concepts they are working on, a flexible technique capable to deal with variability is needed.

Also, LSA does not utilize a grammar or a predefined vocabulary. This makes automation much simpler and 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 the usage rather than a predefined dictionary. This is a stated advantage over using a traditional natural language approach, such as in [7, 8], where a (subset) grammar for the English language must be developed.

#### 4. Previous Experiments with LSA

Experiments into how domain knowledge is embodied within software are being investigated in an empirical manner. The work presented here focuses on using the vector representations to compare components (at a specific level of granularity) and classify them into clusters of semantically similar concepts.

Given a software system, it can be broken down into a set of individual documents to be used as input to LSA. To cluster the source code documents they are partitioned based on similarity value  $\lambda$  with respect to the other documents, in the semantic space. A minimal spanning tree (MST) algorithm is used to cluster the documents based on a given threshold for the similarity measure. A document is added to a cluster if it is at least  $\lambda$  similar to any one of the other documents in the cluster. This strategy attempts to group as many documents together within the given similarity range. The similarity measures are computed by the cosine of the two vector representations of the source code documents. The similarity value therefore has a domain of  $[-1, 1]$ , with the value 1 being "exactly" similar.

A simple parsing of the source code is done to break the source into the proper granularity and remove any non-essential symbols. Comment delimiters and many syntactical tokens are removed as they add little or no semantic knowledge of the problem domain. Also, the LSA method inherently will see such ubiquitous tokens such as a semi-colon as a totally non-discriminating feature between to source code components. That is, every meaningful C++ component contains a semi-colon. Therefore, the variance of this feature is very low (most likely zero) thus; if two components have a semi-colon then nothing can be said about their similarity.

The granularity of the source code input to LSA is of interest at this point. In the applications of LSA to natural language corpuses, typically a paragraph or section is used as the granularity of a document. Sentences tend to be too small and chapters too large. In source code, the analogous concepts are function, structure, module, file, class, etc. Obviously, statement granularity is too small and a file containing multiple functions may be too large.

In previous experiments the function and class declaration levels have been used [15]. Two readily

available software systems were used as data for the experiments: LEDA [13] (Library for Efficient Data structures and Algorithms) and MINIX [21] (Operating System). LEDA is a library of the data types and algorithms for combinatorial computing and provides a sizable collection of data types and algorithms in a form that allows them to be used by non-experts. LEDA is composed of over 140 C++ classes. MINIX is a simple version of the UNIX operating system and widely used in university level computer science OS courses. It is written in C and consists of approximately 28,000 lines of code. Given that LEDA is written in C++ using an object-oriented methodology the granularity chosen is that of the class LEDA has 144 source code documents. For MINIX the function level is used along with some whole files that are made up of data structure definitions. This resulted in 498 source code documents for MINIX.

The previous work supported the concept of using LSA as a similarity measure for clustering software at a given level of granularity, namely a class and function level [15]. The clusters automatically produced by this method tended to reflect the reality of the source code [15]. Pieces of source code that had large amounts of semantic similarity were in general grouped together and modules with no relation to others remained apart. The clusters in the LEDA library seem to reflect class categories, that is, groups of related classes that function on similar concepts or solve common types of problems. In the MINIX system, the clusters are quite different due to the different methodology and programming language utilized. In this case, the clusters represented sets of documents that represent a class or abstract data type. Basically, the larger clusters are typically composed of one or two data structure definitions and a number of functions that utilize these data structures.

While these experiments support the use of LSA to source code, the fact is that both of these software systems are very well written, documented, and organized. Also, neither of these systems is very large. In general, one does not need complex tools to help in understanding these types of software systems. In order to test these methods usefulness to the problem, a more real world type software system is now investigated.

#### 5. Experiments with Mosaic

To determine how well LSA supports the program understanding process, the source code for version 2.7 of Mosaic [16] was used as training input into LSA and clustered using the described methods. The resulting partitioning was used to help support

understanding portions of the source code. Mosaic is written in C and was programmed and developed by multiple individuals. No single coding standard is observed over the entire system and differing standards are often used within a given file. Little or no external documentation on the design or architecture is available and the internal documentation is often scarce or missing completely. In short, Mosaic reflects the realities often found in commercial software due to the many external issues that affect a software development project.

<b>Number of Files</b>	269
<b>LOC</b>	95,000
<b>Vocabulary</b>	5,114
<b>Number of Parsed Documents</b>	2,347
<b>Number of Clusters Produced</b>	655

**Table 1. Vitals for Mosaic.**

### 5.1. Clustering Mosaic

Table 1 gives the size of the Mosaic system (269 files containing approximately 95 KLOC). A semantic space, using a dimensionality of 350, was generated by LSA for the 2,347 documents. The documents were then clustered, based on a cosine value of 0.7 (45°) or greater into 655 groupings.

A distribution of the clusters based on the number of documents they contain is given in table 2. There are a large number of singleton clusters (481) and few really large clusters. These numbers reflect the same type of trends that were found in the earlier experiments. The large number of clusters of size one reflects the fact that many functions often stand by themselves semantically. For the most part, the largest cluster is composed of a common header comment that is found in almost every file. It also includes a large number of very small documents that were parsed out to be only one or two lines of code.

A number of scenarios were envisioned that require such understanding of a large software system with little existing external documentation. The system may be under maintenance by a person with little knowledge of the system or a reengineering of the system may be planned. In such a case, the software is written in C, a reengineering of the system in another language, say C++, may be planned. In fact, such a reengineering of Mosaic actually took place and current versions are written in C++.

The clustering of the source code gives another dimension to view relationships among pieces of source code. Grouping functions and structures together within a file often represent some semantic relationship within the grouping. For instance, an abstract data type (ADT) is often encapsulated in the C language within an implementation file (.c) and an associated specification file (.h). Unfortunately, not all software systems are written with good habits of coupling and cohesion in mind. In legacy systems, it is quite common that little (or no) semantic encapsulation is used, concepts are spread over multiple files, and files contain multiple concepts.

<b>Number of Documents</b>	<b>Number of Clusters</b>
1	481
2	98
3 - 5	46
6 - 10	15
11 -30	8
38	1
99	1
1084	1

**Table 2. A distribution of the size of clusters. The number of cluster that contain a given number of documents.**

With this in mind, a number of simple metrics can be developed that give some heuristics about the semantic cohesion of a particular file or cluster based

- **A software system is a set of files  $S = \{f_1, f_2, \dots, f_n\}$ .**
- **The Total number of files in the system is  $n = |S|$ .**
- **A file is a set of documents  $f_i = \{d_{1,i}, d_{2,i}, \dots, d_{k_i,i}\}$ , all  $f_i$ 's in  $S$  are disjoint.**
- **A document is any contiguous lines of source code and/or text. Typically, a document is a function, block of declarations, definitions, or a class declaration including its associated internal documentation (comments).**
- **The set of all documents in a system is noted as  $S_d = f_1 \cup f_2 \cup \dots \cup f_n$**
- **The Total number of documents in a system is then  $|S_d|$ .**
- **A cluster,  $c_k$ , is a set of documents from the files of  $S$  such that  $c_k \subseteq S_d$ .**
- **Let  $C$  be the set of all clusters,  $c_k$ , such that  $C$  is a set of disjoint clusters that represent a complete partition of all documents in  $S$ .**

**Figure 1. Definitions**

on the intersections of documents. The following defines a set of metrics that is utilized to assist in program understanding based on the given clustering and file organization of a software system.

## 5.2. Measures on Clusters and Files

There are a number of terms that require some explicit definitions: software systems, files, documents, and clusters. These definitions are given in figure 1. With these definitions a set of metrics are defined. The following is a set of measures and metrics that pertain to clusters of source code documents:

- Size of cluster,  $c_k$ , is the number of documents in a cluster, noted  $|c_k|$ .
- Number of files that contain a document from a given cluster is  $|FDC_k|$  where
$$FDC_k = \{f \in S \mid c_k \cap f \neq \emptyset\}$$
- Semantic cohesion of a cluster with respect to files is  $SCCF_k = 1 - \frac{|FDC_k| - 1}{|c_k|}$ .
- Number of documents in a cluster from a given file is  $|DCF_{i,k}|$  where
$$DCF_{i,k} = \{d \mid d \in c_k \cap f_i, c_k \in C, f_i \in S\}$$
- Degree of relationship of a given file with a given cluster  $R_{i,k}$  is  $|DCF_{i,k}| / |f_i|$ .

Below is a set of measures and metrics that deal directly with files of the software system:

- Size of a file,  $f_i$ , is the number of documents in the file, noted  $|f_i|$ .
- Number of clusters that contain a document from a given file is  $|CDF_i|$  where
$$CDF_i = \{c_k \in C \mid c_k \cap f_i \neq \emptyset\}$$
- Semantic cohesion of a file with respect to clusters is  $SCFC_i = 1 - \frac{|CDF_i| - 1}{|f_i|}$ .
- Number of files related by a cluster to a given file,  $f_i$ , is  $|RF_i|$  where
$$RF_i = \{f \in S \mid c_k \cap f \cap f_i \neq \emptyset, c_k \in C\}$$
- Number of files strongly related by a cluster to a given file,  $f_i$ , is  $SRF_i$ :  $SRF_i = |RF_i| - \max |c_k| - 1$  and  $c_k \in LC_k$  where  $LC_k$  is the set of clusters that contain documents from  $f_i$  and have a low semantic cohesion with respect to files.

$$LC_k = FDC \cap \left\{ c_j \in C \mid 1 - \frac{|FDC_j| - 1}{|c_j|} < \epsilon \right\}$$

where  $\epsilon$  is an empirically established threshold.

## 5.3. Understanding Mosaic

The above measures and metrics were computed for the clustering of Mosaic that was generated. The resulting values are used to identify groups of documents in the software system that should be investigated as a whole. The following guidelines are utilized in assessment of clusters and files:

- Semantic cohesion of a file with respect to clusters ( $SCFC_i$ ) should be high.
- Number of files strongly related by a cluster to a given file ( $SRF_i$ ) should be low.
- Semantic cohesion of a cluster with respect to files ( $SCCF_k$ ) should be high.
- Degree of relationship of a given file with a given cluster ( $R_{i,k}$ ) should be high.

Each of the following examples presents a group of files and clusters that are related. They were selected either solely based on the values of their associated metrics, or in conjunction with some additional domain knowledge (e.g., file names, existence of some variables in the files, etc.). Files that satisfy the above-mentioned conditions were considered for further manual inspection. This step, selecting the files that are candidates for manual inspection, can be automated and reduces the amount of manual work needed to understand the software system. For the files and clusters the measurements and metrics are computed and presented in three tables: one for the metrics and measurements dealing with files (tables 3, 6 and 9), another for the metrics dealing with clusters (tables 4, 7, and 10), and the

File ( $f_i$ )	$ f_i $	$SCFC_i$	$SRF_i$
DrawingArea.c	11	0.73	3
DrawingArea.h	1	1.00	2
DrawingAreaP.h	1	1.00	2
HTML.c	91	0.69	14

**Table 3. Metrics on the important files related to DrawingArea.c**

third for the degree of relationship between the files and clusters (tables 5 and 8).

### 5.3.1. Example: DrawingArea

The first example shows a group of related files (see table 3) that were selected based on the file names, a natural choice that an analyst would do when starting to understand a software system. The goal of the experiment was to see if using the measurements and values of the metrics, one could identify the right related files. DrawingArea.c was the first selected file, and the existing measurements indicated that it is strongly related with 3 other files (see  $SRF_i$  in table

3). The degree of relationship with cluster  $c_1$  is very low (see table 5), so the related files through that cluster were not considered for further analysis. The measurements and the metrics (table 3, 4 and 5) indicated DrawingArea.h and DrawingAreaP.h as strong candidates to analysis. Given the names of the files, this is not a surprising finding. The values also indicated HTML.c as the best candidate among the rest of the related files. HTML.c does not satisfy entirely the above-mentioned constraints. However, the fact that it relates to DrawingArea.c through three clusters of which two have high metric values (see table 4), and it has a high cohesion in table 3, promoted it as candidate for in-depth analysis.

The in-depth analysis revealed that indeed the three strongly related files implement a minimalist drawing area widget. More than that, the files implement a well-defined abstract data type – drawing area. A form of information hiding was even used by using a separate file namely, DrawingAreaP.h, to implement some “private” functions.

File ( $f_i$ )	Cluster	$R_{i,k}$
DrawingArea.c	$c_1$	0.18
DrawingArea.c	$c_{327}$	0.73
DrawingArea.c	$c_{331}$	0.09
DrawingArea.h	$c_{327}$	1.00
DrawingAreaP.h	$c_{327}$	1.00
HTML.c	$c_1$	0.01
HTML.c	$c_{327}$	0.01
HTML.c	$c_{331}$	0.01

**Table 4. Degree of relationship of a given file with a given cluster**

Two of the functions in DrawingArea.c connect the files with over 200 other files, through cluster  $c_1$ . Closer inspection revealed that the two functions are in fact constructors and have only two lines of code. This makes them similar with many other constructor-type functions, so the induced relationships were ignored. The values of the metrics in table 1 and table 3, would have already eliminated this relationships. The analysis confirmed that this was a coincidental relationship.

Cluster	$SCCF_k$
$c_{327}$	0.64
$c_{331}$	0.50
$c_1$	0.81

**Table 5. Semantic cohesion of clusters with respect to files**

Although the metrics indicated a weak relationship with the HTML.c file, the relating functions were analyzed. The two functions (from

DrawingArea.c and HTML.c) in cluster  $c_{331}$  are related because they are definitions to similar structures: one defines htmlClassRec, while the other defines drawingAreaClassRec. Both definitions use the same constant names and similar identifiers (e.g., TRUE, FALSE, NULL, Initialize, Inherit, Resize, etc.). These semantic similarities indicated that both functions use the same global constructs (user defined types and identifiers) and the ADTs that they relate to could be in fact specializations of the same parent (or abstract) class. The other related function from the HTML.c file, is a geometry manager for a widget that is also used by the drawing area ADT.

These findings indicated that HTML.c should be also analyzed in conjunction with DrawingArea.c and its closely related files. Using the same metrics and considering HTML.c as the starting file, it was found that the HTMLWidget.c file is also related to the concepts derived previously. Finally, it was concluded that these five files contain definitions for a general (abstract) widget structure (ADT or class) and implementation of at least two specializations of it: drawingAreaClassRec and htmlClassRec.

This example proved that analyzing the metrics, groups of files that contain cohesive implementation of ADTs or classes representing some concepts (drawing area) could be easily identified. The metrics helped identify files that contain implementation of similar structures (html record) and implementation of a general (abstract) concept (widget) that is a generalization of the previous ones.

### 5.3.2. Example: Chunk Handling & Flexible Arrays

In this experiment a files was selected at random, among those with very high semantic cohesion with respect to clusters, containing between 5 and 20 documents.

The selected file was HTChunk.c, with 8 documents and a cohesion value of 0.88 in table 6. From this point on a similar procedure with the one described in first example is followed. The metrics indicated a highly cohesive set of files: HTChunk.h, HTChunk.c, and HTAAFile.c. Upon further analysis, it was determined that the two functions from

File ( $f_i$ )	$ f_i $	$SCFC_i$	$SRF_i$
HTChunk.c	8	0.88	3
HTChunk.h	1	1.00	3
HTAAFile.c	5	0.60	16
HTNews.c	55	0.49	17

**Table 6. Metrics on the important files related to HTChunk.c**

HTAAFile.c that are related perform similar functions on different data structures (e.g., adding a

character to a list of characters) and share variables and constants with the same name (e.g., ch, FILE, NULL). In addition, the size of these functions is relatively small (5-10 lines of code). Therefore, this file was not considered further in the analysis. However, the similarity shows that HTAAFile.c contains functions that implement some sort of list, even if not directly related to the chunks concept. Therefore, a separate analysis of this is recommended. Similar facts were found for the HTNews.c file, although its metrics were even lower.

Cluster	SCCF <sub>k</sub>
C <sub>472</sub>	0.67
C <sub>466</sub>	0.63

**Table 7. Semantic cohesion of clusters with respect to files**

It was also found that the remaining two files (HTChunk.c and HTChunk.h)

implement an ADT that deals with flexible arrays or chunks. A chunk, in this system, is a structure that may

be extended. These routines create and append data

File (f <sub>i</sub> )	Cluster	R <sub>i,k</sub>
HTChunk.c	C <sub>472</sub>	1.00
HTChunk.h	C <sub>472</sub>	1.00
HTAAFile.c	C <sub>472</sub>	0.40
HTAAFile.c	C <sub>466</sub>	0.60
HTNews.c	C <sub>472</sub>	0.02

**Table 8. Degree of relationship of a given file with a given cluster**

to chunks and automatically reallocate them as necessary. The generality of the structure determined the other (weaker) relationships with the other files. This suggested that those files implement similar structures (lists) but using other concepts (e.g. files and news articles rather than chunks).

The study of the related files and clusters indicated that cluster c<sub>466</sub> (see tables 8 and 9) should be analyzed separately. This supports the values in table 7 that also indicated the “interestingness” of the HTAAFile.c that strongly relates to the cluster c<sub>466</sub> (table 8). This example showed that, solely using the metrics, groups of strongly related and cohesive files could be identified and that they implemented a general structure (chunks or flexible arrays). The metrics helped to identify files that contained similar structures (lists). The fact that Mosaic was written by several authors lead to interesting facts such as the fact that often, different authors implemented their own list processing module, instead of using one general one, across the system. Again, in this case the identified similarities help finding these structures (e.g., lists). After that, it is easy to manually identify the concepts (objects) that are handled by the structures (e.g., stored in lists).

File (f <sub>i</sub> )	f <sub>i</sub>	SCFC <sub>i</sub>	SRF <sub>i</sub>
newsr.c	55	0.74	1
HTNews.c	31	0.49	17

**Table 9. Metrics on the important files related to newsr.c**

### 5.3.3. Example: cluster c<sub>466</sub>, the Password and Access Control

In this example a cluster was chosen as starting element in the analysis. The starting cluster is c<sub>466</sub> that was indicated in the previous example as candidate for separate analysis. The cluster spans over 14 highly related and cohesive files. The manual analysis revealed that 10 of these files implemented the basic functions and structures to handle passwords and access control. The other files were using these functions and structures. This time, the names of the files would not have indicated the relationships. Due to limited space, the actual metrics are not shown here.

### 5.3.4. Example: top clusters and newsgroup

This example deals with the analysis of a number of clusters with very high cohesion with respect to files. The second example (chunk handling) showed that the HTNews.c file should be a candidate for manual inspection, but the observed relationships

Cluster	SCCF <sub>k</sub>
C <sub>1</sub>	0.81
C <sub>200</sub>	0.93
C <sub>205</sub>	0.88
C <sub>206</sub>	0.89
C <sub>201</sub>	0.50

**Table 10. Semantic cohesion of clusters with respect to files**

were not relevant to that group of files. As mentioned, the relationships only indicated that there is a type of list structure implemented in this file. HTNews.c is related to newsr.c file that has high cohesion (table 9). More than

that, newsr.c is related to three of the top ten (table 10) most cohesive clusters (c<sub>200</sub>, c<sub>205</sub>, and c<sub>206</sub>). Therefore, these were the exact type of clusters to be considered as starting elements in this experiment.

The analysis showed that newsr.c implements a number of structures that deal with the concepts of “news group” and “news article”. The relationship with the HTNews.c file, although not very strong, is significant because HTNews.c implements, among other things, a “Network News Transfer protocol module for the WWW library”. In order to do that it uses the structures implemented in the newsr.c file. More than that, the list structure, indicated by the relationships in the second example, is in fact a list of news articles.

## 6. Conclusions

LSA seems to be a promising tool to assist in supporting some of the activities of the program understanding process. The methods described here can be used not only as initial step, but also in an interactive way throughout the understanding process. Once a module is identified and understood, the similarities that have been initially discarded can be reanalyzed, considering the new knowledge gleaned from the process.

The next step in the research will be to expand the sets of software systems being examined. It will most likely be prudent to select some very orthogonal domains and some closely inter related domains to assess the application of LSA. Each domain must have a number of example components with varying degrees of internal and external documentation, which will give a good spectrum of the particular domain and result in a valid representation of the domain knowledge. Assessing the relative quality and validity of the constructed semantic spaces is the main goal of this research.

Combining this method with structural methods is an important direction. These methods will not produce excellent results without integrating other types of features. Coupling this with data and control flow information, for example, will work to address both dimensions of the program understanding process.

## 7. References

- [1] Berry, M. W., "Large Scale Singular Value Computations," *Int. J. of Supercomputer Applications*, vol. 6, 1992, pp. 13-49.
- [2] Berry, M. W., Dumais, S. T., and O'Brien, G. W., "Using Linear Algebra for Intelligent Information Retrieval," *SIAM: Review*, vol. 37, no. 4, 1995, pp. 573-595.
- [3] Biggerstaff, T. J., Mitbender, B. G., and Webster, D. E., "Program Understanding and the Concept Assignment Problem," *CACM*, vol. 37, no. 5, May 1994, pp. 72-82.
- [4] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R., "Indexing by Latent Semantic Analysis," *J. of the Am. Soc. for Info. Sci.*, vol. 41, 1990, pp. 391-407.
- [5] Duda, R. O. and Hart, P. E., *Pattern Classification and Scene Analysis*, Wiley, 1973.
- [6] Dumais, S. T., "Latent Semantic Indexing (LSI) and TREC-2," in Proceedings of The 2<sup>nd</sup> Text Retrieval Conference (TREC-2), March 1994, pp. 105-115.
- [7] Etzkorn, L. H., Bowen, L. L., and Davis, C. G., "An Approach to Program Understanding by Natural Language Understanding," *Natural Language Eng.*, vol. 5, no. 1, 1999, pp. 1-18.
- [8] Etzkorn, L. H. and Davis, C. G., "Automatically Identifying Reusable OO Legacy Code," *IEEE Computer*, vol. 30, no. 10, Oct. 1997, pp. 66-72.
- [9] Faloutsos, C. and Oard, D. W., "A Survey of Information Retrieval and Filtering Methods," University of Maryland, CS-TR-3514, August 1995.
- [10] Jolliffe, I. T., *Principal Component Analysis*, Springer Verlag, 1986.
- [11] 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*, vol. 104, no. 2, 1997, pp. 211-240.
- [12] Landauer, T. K., Laham, D., Rehder, B., and Shreiner, M. E., "How Well Can Passage meaning Be Derived without Using Word Order? A Comparison of Latent Semantic Analysis and Humans," in Proceedings of Proceedings of the 19<sup>th</sup> Annual Conf. of the Cognitive Science Society, 1997, pp. 412-417.
- [13] LEDA, "The LEDA Manual Version R-3.7," LEDA Research, Date Accessed: 4/29/1999, <http://www.mpi-sb.mpg.de/LEDA/index.html>, 1998.
- [14] Maletic, J. I. and Reynolds, R. G., "A Tool to Support Knowledge Based Software Maintenance: The Software Service Bay," in Proceedings of The 6th IEEE ICTAI, Nov. 6-9 1994, pp. 11-17.
- [15] Maletic, J. I. and Valluri, N., "Automatic Software Clustering via Latent Semantic Analysis," in Proceedings of 14th IEEE Int. Conf. on ASE, October 1999, pp. 251-254.
- [16] Mosaic, "Mosaic Source Code v2.7b5," NCSA, ftp site, Date Accessed: 4/12/2000, <ftp://ftp.ncsa.uiuc.edu/Mosaic/Unix/source/>, 1996.
- [17] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P., *Numerical Recipes in C, The Art of Scientific Computing*, Cambridge University Press, 1996.
- [18] Rist, R., "Plans in Program Design and Understanding," in Proc. of Workshop Notes for AI & APU, AAAI-92, 1992, pp. 98-102.
- [19] Soloway, E. and Ehrlich, K., "Empirical Studies of Programming Knowledge," *IEEE Trans SE*, vol. 10, no. 5, Sept. 1984, pp. 595-609.
- [20] Strang, G., *Linear Algebra and its Applications*, 2nd ed., Academic Press, 1980.
- [21] Tanenbaum, A. and Woodhull, A., *Operating Systems Design and Implementation*, Prentice Hall, 1997.