SUPPORT FOR SOFTWARE MAINTENANCE USING LATENT SEMANTIC ANALYSIS

JONATHAN I. MALETIC
ANDRIAN MARCUS

Division of Computer Science
The Department of Mathematical Sciences
The University of Memphis
Campus Box 526429 Memphis TN 38152, USA
Phone: (901) 678-3140. Fax: (901) 678-2480
jmaletic@memphis.edu
amarcus@memphis.edu

ABSTRACT

The paper describes the results of applying semantic (versus structural) methods to the problems of software maintenance and program comprehension. Here, the focus is on tools to assist programmer to understand large legacy software systems. The method applied, 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). The intent of applying Latent Semantic Analysis to software components is to automatically induce a specific semantic meaning of a given component. Here, LSA is used as the basis to group software components, across files, to assist in program comprehension. This clustering is used in the understanding of a nontrivial software system, namely a version of Mosaic.

KEYWORDS

Software Maintenance, Program Understanding, Latent Semantic Analysis

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 [2, 5, 9, 13, 14]. 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.

The reasoning for choosing LSA as underlying method in this research is discussed in the following section. 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. A number of metrics are defined based on these similarity measures to help support program understanding. Examples of how these metrics help support the program understanding process are also given. Finally, conclusions to these experiments are presented along with future research directions.

2. ADVANTAGES OF USING LSA

There are a variety of information retrieval methods including traditional approaches [6] 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) [12, 15] of a matrix (word $\times$ context) derived from a corpus of natural text that pertains to knowledge in the particular domain of interest. The corpus is used as training set and LSA builds a semantic space. The result is that each word is represented as a vector in this space. The similarity of any two words, any two 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 the semantic space is used as the

degree of qualitative similarity of meaning [3]. The length of vectors is also useful as a measure.

One of the criticisms of this method, when applied to natural language texts is that it does not make use of word order, syntactic relations, or morphology. But very good representations and results are derived without this information [1]. This characteristic is very well suited to the domain of software, both 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. Internal documentation is also commonly written in a subset of English [5] that may also lend itself to the methods utilized by LSA. 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 then a predefined dictionary. This is a stated advantage over using a traditional natural language approach, such as in [4, 5], were a (subset) grammar for the English language must be developed.

3. 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. A simple parsing of the source code is done to break the source into the proper granularity (documents) 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.

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 grouped based on similarity value λ with respect to the other documents, in the semantic space. A minimal spanning tree (MST) [7] 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 λ similar to any one of the other documents in the cluster. 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.

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 is too large. In previous experiments the function and class declaration levels have been used [10]. Two readily available software systems were used as data for the experiments: LEDA [8] (Library for Efficient Data structures and Algorithms) and MINIX [16] (Operating System). The work supported the concept of using LSA as a similarity measure for clustering software at a given level of granularity, namely a class or function level. The clusters in the LEDA library reflected 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.

4. EXPERIMENTS WITH MOSAIC

To determine how well LSA supports the program understanding process, the source code for version 2.7 of Mosaic [11] was used as training input into LSA and clustered using the described methods. The resulting partitioning was used to help support understanding of 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 often different standards are 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. In short, Mosaic reflects the kinds of realities often found in commercial software due to the many external issues that affect a software development project.

4.1. 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 (this value corresponds to a 45 degree angle between the vectors) or greater into 655 groupings.

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++.

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

Table 1. Vitals for Mosaic.

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.
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 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.

4.2. Metrics 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
$$DCF_{i,k} = \{d \mid d \in c_k \cap f_i, c_k \in C, f_i \in S\}.$$

- **A software system is a set of files $S = \{f_1, f_2, \ldots, f_n\}$.**
- **The Total number of files in the system is $n = |S|$.**
- **A file is a set of documents $f_i = \{d_{1,i}, d_{2,i}, .., d_{ki,i}\}$, 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 $S_d = f_1 \cap f_2 \cap \ldots \cap f_n$**
- **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

- Degree of relationship of a given file with a given cluster is $R_{i,k} = |DCF_{i,k}| / | f_i |$.

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:
$$SRF_i = | RF_i | - \max | c_k | - 1 \text{ and } c_k \in LC_k$$

where LC_k is the set of clusters that contain documents from f_i and have a low semantic cohesion with respect to files. $LC_k = FDC_k \cap \{c_j \in C \mid SCCF_j < \epsilon\}$ where ϵ is an empirically established threshold.

4.3. 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. Files that satisfy the above-mentioned conditions were considered for further manual inspection. This step, selecting the files that are

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 2. Metrics on the important files related to DrawingArea.c

candidates for manual inspection, can be automated and reduces the amount of manual work needed to understand the software system.

4.3.1. Example: DrawingArea and the Widget Structure

The first example shows a group of related files (see table 2) 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 table 2). The degree of relationship with cluster c_1 is very low (see table 4), so the related files through that cluster were not considered for further analysis. The measurements and the metrics (table 2, 3 and 4) indicated DrawingArea.h and DrawingAreaP.h as strong candidates to analysis. Given the names of the files, this is not a surprising finding. The in-depth analysis revealed that indeed the three strongly related 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.

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

Table 3. Semantic cohesion of clusters with respect to files

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,

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

so the induced relationships were ignored. The values of the metrics in table 2 and table 4 would have already eliminated this relationship. The analysis confirmed that this was a coincidental relationship.

The values also indicated HTML.c as the best candidate among the rest of the related files. The analysis indicated that the documents linking these files 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.

Although the degree of relationship (see table 4) between the documents in HTML.c and the identified clusters was small, these findings indicated that HTML.c should be also analyzed in conjunction with DrawingArea.c and its closely related files. Using the same procedure and considering HTML.c as the starting file, it is 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.

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 5. Metrics on the important files related to HTChunk.c

4.3.2. Example: Chunk Handling and Flexible Arrays

In this experiment a file 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 5. 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 also found that the files HTChunk.c and HTChunk.h implement an ADT that deals with flexible arrays called chunks (a form of lists). The generality of the structure determined the other (weaker) relationships with the other files. This

File (f_i)	Cluster	$R_{i,k}$
HTChunk.c	c_{472}	1.00
HTChunk.h	c_{472}	1.00
HTAAFile.c	c_{472}	0.40
HTAAFile.c	c_{466}	0.60
HTNews.c	c_{472}	0.02

Table 6. Degree of relationship of a given file with a given cluster

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_{466} should be analyzed separately. The related functions from HTAAFile.c and HTNews.c were found to implement in fact similar structures to chunks (i.e., lists). 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 (i.e., chunks). The metrics helped to identify files that contained similar structures (i.e., 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.

4.3.3. Example: Cluster c_{466}, the Password and Access Control

In this example a cluster was chosen as starting element in the analysis. The starting cluster is c_{466} 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. Similar experiments were done on clusters selected solely based on the value of their cohesion metric.

5. 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 an initial step, but also in an interactive way throughout the software 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.

6. REFERENCES

[1] 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.

[2] Biggerstaff, T. J., Mitbander, B. G., and Webster, D. E., "Program Understanding and the Concept Assignment Problem," *CACM*, vol. 37, no. 5, May 1994, pp. 72-82.

[3] Dumais, S. T., "Latent Semantic Indexing (LSI) and TREC-2," in Proceedings of The Second Text Retrieval Conference (TREC-2), March 1994, pp. 105-115.

[4] Etzkorn, L. H., Bowen, L. L., and Davis, C. G., "An Approach to Program Understanding by Natural Language Understanding," *Natural Language Engineering*, vol. 5, no. 1, 1999, pp. 1-18.

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

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

[7] Kruskal, J. B., "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem," *Proc. Amer. Math. Soc.*, vol. 7, no. 1, 1956, pp. 48-50.

[8] LEDA, "The LEDA Manual Version R-3.7," LEDA Research, Webpage, Date Accessed:

4/29/1999, http://www.mpi-sb.mpg.de/LEDA/index.html, 1998.

[9] 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 International Conference on Tools with Artificial Intelligence, New Orleans LA, Nov. 6-9 1994, pp. 11-17.

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

[11] Mosaic, "Mosaic Source Code v2.7b5," NCSA, ftp site, Date Accessed: 4/12/2000, ftp://ftp.ncsa.uiuc.edu/Mosaic/Unix/source/, 1996.

[12] 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.

[13] Rist, R., "Plans in Program Design and Understanding," in Proceedings of Workshop Notes for AI & Automated Program Understanding, AAAI-92, San Jose CA 1992, pp. 98-102.

[14] Soloway, E. and Ehrlich, K., "Empirical Studies of Programming Knowledge," *IEEE Transactions on Software Engineering*, vol. 10, no. 5, September 1984, pp. 595-609.

[15] Strang, G., *Linear Algebra and its Applications*, 2nd ed., Academic Press, 1980.

[16] Tanenbaum, A. and Woodhull, A., *Operating Systems Design and Implementation*, Prentice Hall, 1997.