Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code

Denys Poshyvanyk, Andrian Marcus[1]

Department of Computer Science
Wayne State University
Detroit Michigan 48202
313 577 5408
denys@wayne.edu, amarcus@wayne.edu

Abstract

The paper addresses the problem of concept location in source code by presenting an approach which combines Formal Concept Analysis (FCA) and Latent Semantic Indexing (LSI). In the proposed approach, LSI is used to map the concepts expressed in queries written by the programmer to relevant parts of the source code, presented as a ranked list of search results. Given the ranked list of source code elements, our approach selects most relevant attributes from these documents and organizes the results in a concept lattice, generated via FCA.

The approach is evaluated in a case study on concept location in the source code of Eclipse, an industrial size integrated development environment. The results of the case study show that the proposed approach is effective in organizing different concepts and their relationships present in the subset of the search results. The proposed concept location method outperforms the simple ranking of the search results, reducing the programmers' effort.

1. Introduction

Identifying the parts of the source code that correspond to a specific functionality is a prerequisite to program comprehension and is one of the most common activities undertaken by developers. This process is called *concept* (or *feature*) location.

One of the most commonly used technique for concept location is the source code text search, where developers write queries and a search engine returns a list of source code elements relevant to the query. In many cases, only a small fraction of the result set is relevant to the concept being located. In these situations, the developers either undertake the daunting task to investigate in detail as much as they can from the results, or they reformulate their query to reduce the size of result list. Eventually, even after a series of queries, the user will still need to investigate the set of results. Our work aims to help the user in reducing his search effort by providing additional structure among the search results, such that parts of the source code and documentation are grouped based on common topics. Our inspiration comes from similar approaches used in web searching, such as the Vivisimo[2] and Clusty[3] clustering engines.

Specifically, we augment an existing information retrieval (IR) based technique for concept location [1] with automatic organization of the search results using formal concept analysis (FCA). The IR based concept location technique uses a search engine based on Latent Semantic Indexing (LSI) [2], which allows the user to search source code and related textual documentation by writing natural language queries and retrieving a list of source code elements (for example, classes, methods, functions, files), ranked based on their similarity to the query. Based on the ranked results of the search we automatically generate a labeled concept lattice. Developers can determine whether a node from the concept lattice (that is, topic or category) is relevant or not to their query by simply examining its label; they can then explore only relevant nodes in the lattice and ignore the other ones, thus reducing their search effort.

2. Related work

This section outlines research related to our work, where we present existing approaches to feature and concept location, with specific focus on the use of FCA in this context.

Concept location is also referred to in the literature as *feature* identification or *concern* location. *Features* are special concepts that are associated with the user visible functionality of the system. The shared goal of these techniques is to identify the computational units

[1] Corresponding author
[2] www.vivisimo.com
[3] www.clusty.com

(for example, methods, function, classes, etc.) that specifically implement a concept of interest from the problem or solution domain of the software. Concept location is an essential part of the incremental change process [3]. Through the rest of the paper we use the term *concept location*, even when we refer to techniques that are named differently. When the context may produce confusion between the use of the word *concept* in *concept location* and *concept analysis* we use *feature* instead of *concept*.

Existing approaches to concept location use different types of software analyses. They can be broadly classified into static, dynamic, and combined analysis based approaches.

Wilde et al. [4] was the first to address the problem of feature location using the Software Reconnaissance method, which utilizes dynamic information. The approach is based on building two execution traces based on two sets of test cases – one that exercises the feature of interest and one that does not. The resulting traces are used to identify elements of the source code which implement that feature. This approach has been recently revisited by several researches to improve its accuracy by using new methods on how to analyze execution traces [5] as well as selecting execution scenarios [6].

Biggerstaff et al. [7] introduced the problem of concept assignment in the context of static analysis. They implement a tool which extracts identifiers from the source code and clusters them to support identification of concepts. The simplest and most commonly used static technique is based on searching the source code using regular expression matching tools, such as the Unix utility *grep*. Modern development environments like Eclipse and MS Visual Studio build many useful add-ons on top of simple pattern matching, including references to class and method names, etc. A significant improvement over regular expression matching is brought by information retrieval-based approaches [1, 8], which allow more general queries and rank the results to these queries.

Among other static-based techniques for concept location is the one proposed by Chen et al. [9], which is based on the search of abstract system dependence graph. This approach has been recently extended in [10] via analysis of dependency topologies to rank elements of interest in source code. Some other methods combine other types of information obtained via static analysis (that is, textual and structural), such as Zhao et al. [11] who proposed the technique which combines information retrieval with branch-reserving call-graph information to automatically assign features to respective elements in the source code. Gold et al. [12] proposed an approach for binding concepts with overlapping boundaries to the source code which is formulated as a search problem using genetic and hill climbing algorithms. A comparison and overview of static feature location techniques can be found in [13].

Eisenbarth et al. [14] combined both static (that is, dependencies) and dynamic (that is, execution traces) information to identify features in programs and use FCA to relate features together. Salah and Mancoridis [15] use static and dynamic data to identify feature interaction in Java source code. Poshyvanyk et al. [16] combined an information retrieval based technique with scenario-based probabilistic ranking of the execution traces to improve the precision of feature location.

A comparison of different approaches for feature location in legacy systems is presented in [17]. A more up-to-date summary of all existing approaches can be found in [5], whereas a summary of industrial tools available for feature location is available in [18].

FCA has many uses in software engineering [19] such as identification of objects in legacy code however we discuss here the ones that specifically address concept location. In addition to the work of Eisenbarth et al. [14] (mentioned above), Tonella et al. [20] use dynamic analysis together with FCA to identify aspects in execution traces, while more recently, FCA has been applied for mining cross-cutting concerns from software repositories [21]. Mens et al. [22] apply FCA to mine source code to support various program comprehension tasks, including concept location.

3. Background

In this section we present background information on FCA, a mathematical technique for analyzing binary relations and LSI, an advanced information retrieval method. Readers familiar with FCA or LSI may skip the respective section(s).

3.1. Formal concept analysis

Formal concept[4] analysis is a branch of mathematical lattice theory that provides means to identify meaningful groupings of *objects*[5] that share common *attributes* [23] as well as provides a theoretical model to analyze hierarchies of these groupings.

The main goal of FCA is to define a *concept* as a unit of two parts: *extension* and *intension*. The extension of a concept covers all the objects that belong to the concept, while the intension comprises all the attributes, which are shared by all the objects under consideration.

[4] Note the difference between the use of the term *concept* in FCA and concept location

[5] Also note the difference between the terms object and attribute in FCA and OOP

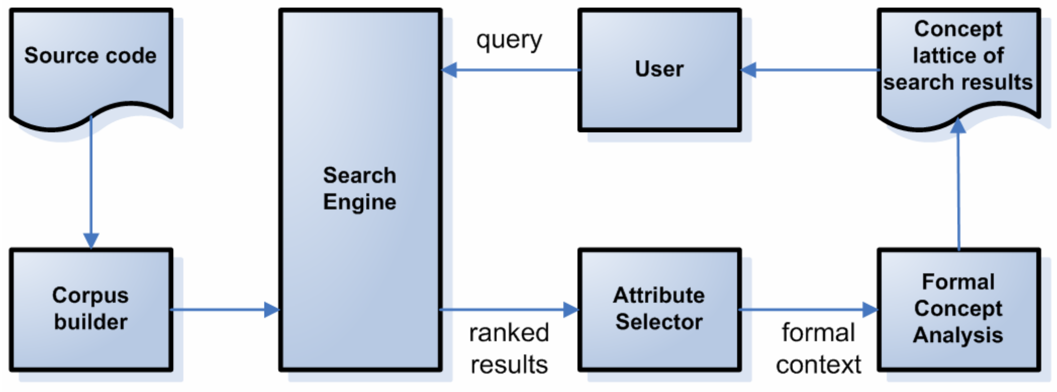

Figure 1. Concept location process using LSI and FCA

In order to apply FCA, the formal context or incidence table of objects and their respective attributes is necessary. Formal context consists of a set of objects O, a set of attributes A, and a binary relation $R \subseteq O \times A$ between objects and attributes, indicating which attributes are possessed by each object. Formally, it can be defined as $C = (A, O, R)$. From the formal context, FCA generates a set of concepts where every concept is a maximal collection of objects that possess common attributes. More formally, a concept is a pair of sets (X, Y) such that:

$$X= \{o \in O \mid \forall a \in Y: (o,a) \in R\}$$
$$Y= \{a \in A \mid \forall o \in X: (o,a) \in R\}, \text{ where}$$

X is considered to be the *extent* of the concept and Y is *intent* of the concept. This set of concepts is called a *complete partial order* where some concepts are super- or sub-concepts with respect to others.

The set of all concepts constitutes a concept lattice and there are several algorithms to compute concepts and concept lattices form a given formal context. For details on these algorithms as well as more complete description on FCA, refer to [23].

3.2. Latent semantic indexing

In the proposed concept location approach we utilize an information retrieval method, LSI [2], as a text indexing and search engine.

LSI is based on a Singular Value Decomposition (SVD) of the co-occurrence matrix of identifiers and comments in source code documents of a software system. SVD is a form of factor analysis, which is used to reduce dimensionality of the feature space to capture most essential semantic information. The formalism of SVD is rather lengthy to be presented in the paper, thus we refer the reader to [2] for complete details.

Originally LSI has been mostly applied on natural language corpora, however, the method has been shown to lend itself well on other types of data, for example, textual information extracted from source code and associated documentation. Some of the software engineering problems, related to concept location, which have been addressed using LSI are concept [1] and feature [16] location, traceability link recovery between source code and documentation [24, 25], tracing requirements [26] and other software artifacts [27], etc.

Details on how LSI is used for feature location in source code are available in [1] and [13].

4. Concept Location using Concept Lattices

In this section we present the details of our approach to concept location, which uses FCA to organize in a concept lattice the results of a search performed by a developer using the LSI based source code search engine. Part of the approach is similar to the one presented in [1] and offers users the same main features, such as the ability to write queries in natural language and sort the results based on their similarity to the query. With the LSI-based source code search engine, developers search the software much the same way they do the internet with popular search engines like Google.

Figure 1 shows the main steps in the concept location process using LSI and FCA. The first two steps are usually performed once, while the other ones are performed repeatedly until the user finds the desired parts of the source code.

1. **Creating a corpus of a software system.** The source code is parsed using a developer-defined granularity level (that is, methods or classes) and documents are extracted from the source code. A corpus is created, so that each method (and/or class) will have a corresponding document in the

resulting corpus. Only identifiers and comments are extracted from the source code. We developed tools that automatically create corpora for MS Visual Studio projects [28] and Eclipse projects [29]. In addition, we also created corpus builder for large C++ projects, using srcML [30] and Columbus [31].

2. **Indexing.** The corpus is indexed using LSI and a representation of the corpus as a real-valued vector subspace is created. Dimensionality reduction is performed in this step, capturing important semantic information about identifiers, comments and their relationships in the source code. In the resulting subspace, each document (method or class) has a corresponding vector.
3. **Formulating a query.** A developer selects a set of terms that describe the concept of interest (for example, 'print page'). This set of words constitutes the initial query. The tool spell-checks all the terms from the query using the vocabulary of the source code (generated by LSI). If any word from the query is not present in the vocabulary, then the tool suggests similar words based on editing distance and removes the term from the search query.
4. **Ranking documents.** Similarities between the user query and documents from the source code (for example, methods or classes) are computed. The similarity between a query reflecting a concept and a set of data about the source code indexed via LSI allows generating a ranking of documents relevant to the feature. All the documents are ranked by the similarity measure in descending order.
5. **Selecting descriptive attributes.** The top *k* attributes from the first *n* documents in the ranked list (for example, methods) are selected. These terms are mostly similar to the selection of the *n* documents but not common to all other documents in the search results.
6. **Applying Formal Concept Analysis.** Before applying FCA we prepare the formal context, which is generated from a set of *n*-first documents (*objects*) in the ranked list and *k* descriptive terms (*attributes*) extracted in the previous step. Subsequently, we apply the FCA bottom-up algorithm [23] to build the set of concepts for a given context which forms a complete partial order, or simply a concept lattice.
7. **Examining results.** The resulting concept lattice, with annotated descriptions for concept nodes and with links to actual documents in source code is presented to the user. The user can browse the results by traversing the lattice and refining queries if desired. If a user finds a part of the concept, then the search succeeds, otherwise, the user formulates a new query, taking into account new knowledge obtained from the investigated documents in the lattice, which may help formulate more specific query (for example, narrow search criteria by taking into account relations between node descriptions in the lattice) and returns to step 3.

4.1. Selecting descriptive attributes

There are several published solutions to extract descriptive terms for sub-collections of documents. For example, okapi weighting scheme and terminological formula are two of the approaches proposed for free-text IR systems [32]. We adopt and adapt here the technique proposed by Kuhn et al. in [33], since it was defined in the context of source code to select relevant terms with respect to given clusters of source code elements. Following we present how this technique is adapted and used to select terms to be used in FCA.

We define a corpus for a software system as a set of documents $D = \{d_1, d_2 \dots d_s\}$. A set of documents in the ranked list which we use to build a formal context is denoted as D_n, where the number of documents is $n=|D_n|$. To denote the rest of the corpus, which does not contain documents in D_n, we use $D^1 = \{D - D_n\}$, where the number of documents is $|D^1| = s - n$.

We define a set of unique terms which occur in D as $T_D = \{t_1, t_2 \dots t_r\}$. A set of unique terms which occur in D_n only is defined as T_{Dn}, where $T_{Dn} \subseteq T_D$.

In order to rank every term $t_i \in T_{Dn}$ (for $i=1 \dots |T_{Dn}|$) with respect to a document collection D_n we apply the following formula to determine the ranking of the terms:

$$sim(t_i, D_n) = sim(t_i, D_n) - \frac{1}{|D^1|} \times \sum sim(t_i, D^1)$$

Using this approach we are able to rank all the unique terms in D_n (for example, T_{Dn}) so that the terms highly similar to the documents in D_n but not to the documents in D^1 are ranked higher. We penalize those terms which are highly similar to D^1, since it is mentioned in [33] that there might be identifiers for data structures or utility classes, which would pollute the top ranked list of terms (for example, *atoi*, *class*, *sqrt*, etc).

4.2. Applying formal concept analysis

We decided to use FCA instead of clustering algorithms because of the following reasons: FCA provides an *intentional* description for each cluster, which makes groupings more interpretable; the generated cluster organization is a *lattice*, rather than a hierarchy, allowing recovery from bad decisions, while exploring the hierarchy; FCA is generally, richer and more *flexible* way of browsing the document space than hierarchical clustering [34].

With the approach presented in this paper we tackle the problem of scalability of FCA in the context of a software system by applying it on the *subset* of relevant search results only. Using this approach, the top search results, that is, the first *n* methods or classes in the ranked list are organized in the concept lattice based on the attributes automatically selected from identifiers and comments implemented in their source code.

To illustrate how FCA works with respect to the problem that we are addressing in this work, that is, concept location, we present the following example of locating the feature '*print page*' in the source code of Eclipse 3.1[6] with the following methods returned as the result of our initial query of the same name as the feature: *getBounds* which obtains the size of the paper, *startPage* and *endPage* which start and end printing a page, *startJob* which initiates a print job which may include printing several pages, *endJob* which finalizes printing a page(s) and *cancelJob* which ends and cancels the print job respectively.

Using the algorithm for selecting descriptive terms, described in section 4.1, the following terms are selected from the identifiers and comments of the returned methods: *printer*, *print, page, job, device, paper* and *rendering.* Note that these terms are specific only to those six methods but not to the rest of the source code in Eclipse.

Using top methods from source code and their descriptive attributes, we generate a formal context $C = (A, O, R)$, where the objects O are aforementioned methods and A are words (*attributes*) extracted from implementation of the methods in O. Note that in this example we choose n top objects (n=6) and k most similar terms to these objects (k=7). The set of binary relations R among O and A are summarized in Table 1.

Table 1. Formal context: objects (six methods from source code of Eclipse) and attributes (shared in identifiers and comments of those methods)

	printer	print	page	job	device	paper	rendering
startJob		X		X			
endJob		X		X			
cancelJob		X		X			
startPage			X			X	X
endPage			X			X	X
getBounds	X				X	X	

While applying FCA on our example, the following concepts are identified:

[6] www.eclipse.org

C_1= ({},{paper})
C_2= ({*getBounds*}, {printer, device})
C_3= ({*startPage*, *endPage*}, {page, rendering})
C_4= ({*startJob*, *endJob*, *cancelJob*}, {print, job})

This set of concepts is referred to as a *complete partial order* whereas some concepts are super- or sub-concepts with respect to others (see Figure 2).

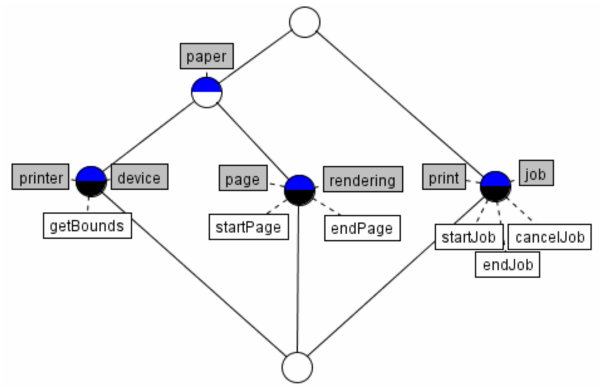

Figure 2. Concept lattice for the '*print page*' feature. Grey boxes are attributes (words) and white boxes are objects (methods).

For example, the concept C_2 is a sub-concept of concept C_1. Intuitively, from the term 'paper' in C_1 we also may assume that C_1 is more general than concepts C_2 'printer device' and C_3 'page rendering'; moreover, implementation of methods which belong to these concepts indeed reflect this fact. In addition, both methods implement different actions related to the paper – *getBounds* is used to obtain physical properties of the paper based on current system device, whereas *startPage* and *endPage* implement operations which initialize and finalize printing of a page respectively.

5. Evaluation of the Proposed Approach

We performed a case study to evaluate our approach and better understand the effects of selecting various values for n and k when applying FCA. The case study design is based on recommendations from [35]. The results of the new approach are compared with those of its predecessor that uses list of ranked results.

5.1. Design of the case study

One recurrent issue in case studies on concept location is the verification of the results. It is often difficult to determine for sure that a certain method implements at least in part a given concept. The best way to validate such a fact is to implement a change that alters that concept and confirm if that method changed or not. Of course, a given change request may be designed and implemented in many ways. In order to minimize the threats to the validity of our results, we opted to design the case studies similarly to those constructed in [16]. Specifically, we decided to locate concepts that are associated with particular bugs reported for the given software. This way we could verify the correctness of the location process by checking the final patches for these bugs, as those are available as well and are not implemented by people associated with the authors. The documentation for every bug used in the case study specifies which methods were changed in response to bug fix. We consider these methods as (part of) the implementation of the concept associated with the bug, which we see as an *unwanted feature*. We used the following criteria to select bugs for the case study: (1) bugs should be well-documented and reproducible; (2) bugs should have approved patches applied in recent releases; (3) none of the authors know the parts of the program corresponding to the features to eliminate potential bias; (4) we could formulate ad-hoc queries using words in the description of the bug which would correctly describe the associated concept. For each bug we are interested in locating at least one of the methods modified during its fix. We define the scope of concept location to finding the starting point of a change, as defined in [3], as it is the role of impact analysis and change propagation to get the full extent of the change in the source code.

5.1.1. Research questions and propositions

The goal of the case study is to evaluate the impact on the size and quality of the concept lattice of the following parameters:

- the number of documents n in the ranked list that should be kept for selection of descriptive attributes and the final concept lattice and
- the number of attributes k that should be selected for the number of n documents.

In addition, we expected that the resulting concept lattice will reduce the searching effort of the developers when compared to the simple ranking of the results based on the similarity of the methods to the user query. This proposition is based on the fact that the new approach can effectively utilize information about relationship among the results of the search based on common attributes rather than only those used in the original user query. In other words, it can effectively group relevant documents and provide informative labels as node descriptions in concept lattice, helping the user to navigate the resulting lattice more effectively, possibly scanning only the fraction of the documents. Such a representation should provide a structural view about different sub-topics present in the results of the search and provide additional information, such as descriptive labels, which can be used as visual cues to navigate results more effectively than a simple ranked list.

5.1.2. Object and settings of the case study

We chose the Eclipse (version 3.1), integrated development environment, a large open-source software system used in research and industry. Eclipse is easily accessible and has well documented bug reports, which will make possible replication or extension of the case study easy in the future.

Table 2. Eclipse source code and corpus vitals

Item	Count
MLOC	2.9
Vocabulary	56,863
Number of parsed documents	86,208

We indexed the source code of Eclipse using the approach outlined in Section 4. We chose method level granularity (that is, each document in the corpus corresponds to a method) and we did not index the class interfaces. We construct the corpus for Eclipse by extracting all comments and identifiers from the source code. The resulting text is processed using the following set of rules: some types of tokens are eliminated (for example, operators, special symbols, some numbers, keywords of the programming language, standard library function names, etc.); the identifiers in the source code are split into parts based on known coding standards while the initial form of each identifier is kept as well; each document in the corpus is created with the comments and identifiers corresponding to each method. No morphological analysis or transformations are applied since we do not use a predefined vocabulary, or a predefined grammar.

The size of the resulting corpus and number of indexed methods from Eclipse is presented in Table 2. We used LSI with a dimensionality reduction factor of *500*, which accurately represents the semantic space of this size.

5.1.3. Evaluation criteria and measures

We compare the results of our new technique with the sorted list of results, obtained with LSI based rankings. We assume that with a ranked list, a user has to scan each document until the relevant document is found (in our case, we consider the first method that relates to the feature of interest). In reality users may use visual cues such as the method name to skip some elements in the ranked list. To simplify the evaluation we consider the case when a user must diligently go through the whole list until the sought method is found.

We want to measure whether the concept lattice structure effectively groups or *distillates* relevant

Table 3. Descriptions, user queries, method ranking, and modified methods for Eclipse bugs

Bug #	Description	Query	Rank	Methods
34160[2]	The task list, which uses the native table widget, cannot be ***sorted*** by clicking on the ***table headers***	"table header sort"	71	org.eclipse.swt.widgets.Table.createHandle **org.eclipse.swt.widgets.Table.createWidget** org.eclipse.swt.widgets.Table.kEventMouseDown org.eclipse.swt.widgets.Table.itemNotificationProc
25457[3]	***Renaming project*** to the same name but with different case causes ***source files*** to be deleted if project's folder is locked by other application.	"rename project source"	89	**org.eclipse.core.internal.localstore.FileSystemStore.move**

documents, thus enabling a developer to locate relevant information faster than in a ranked list of documents. We use two measures proposed in [36], *lattice distillation factor* and *lattice browsing complexity*. Since the authors originally used the measures to find all relevant documents, whereas we are concerned only with the first element that belongs to the feature, we introduce modifications to these measures to accommodate this notion.

Lattice distillation factor

Let C be the set of nodes in the resulting concept lattice. We assume that the programmer, while visiting a node in the lattice can view the actual object which corresponds to this node (for example, methods from the software system). We define $C_{FEATURE} \subseteq C$ as the subset of the methods relevant to the feature, which are present in the concept lattice. We redefine the *minimal browsing area* (MBA) as the minimal part of the lattice that a user should explore, starting from the very top node, to reach the first object in $C_{FEATURE}$. P_{MBA}, the precision of MBA, is the upper bound of the capacity of the lattice to distillate relevant information from the initial list of ranked results. Obviously, the lower bound is the size of the ranked list of results that the user has to scan while he identifies the first method belonging to the feature. We denote the precision of the ranked list as P_{RL}.

We redefine the lattice distillation factor (LDF) as the potential precision gain obtained with the concept lattice compared to the precision of the ranked list.

$$\text{LDF(C)} = \frac{P_{MBA} - P_{RL}}{P_{RL}} \times 100\,\% \qquad \textbf{(1)}$$

Consider the example from Figure 2 and let us assume that the developer is locating the method which cancels printing operation and the method of interest occurs in position 6, having P_{RL}=0.16. However, in the concept lattice it is in position 3, thus P_{MBA}=0.33. Eventually, LDF(C) = (0.33-0.16)/0.16 = 106%, meaning that the concept lattice can distillate related information approximately two times more effectively than the simple ranked list in this particular case.

Lattice browsing complexity

As mentioned in [36], the LDF is only concerned with the cost of reading the documents however the structure of concept lattice has additional *browsing costs*. Thus, we need to consider the number of nodes and the structure of the lattice to evaluate its adequacy for browsing purposes. In order to measure this property of concept lattices we use the second measure from [36], namely *lattice browsing complexity* (LBC). For our problem we redefine LBC to capture the proportion of nodes in the lattice that the developer will see while traversing the MBA (also note that when a node is explored, all its sub-concepts or nodes will be considered, while only some of them will be explored).

$$\text{LBC(C)} = \frac{|C_{VIEW}|}{|C|} \times 100\,\% \qquad \textbf{(2)}$$

where C_{VIEW} is formed by the sub-concepts of each node which belongs to the MBA.

We assume that while visiting a node in the concept lattice the user will read all documents associated with this node. Thus, we impose the same worst case scenario for exploration costs for concept lattices as we did for ranked list, minimizing any bias.

Using the same example in Figure 2, C_{VIEW} = 3, which is the minimal number of nodes the user has to visit in order to locate the feature of interest. Thus LBC(C) = 3/6 × 100 = 50%, which means that the developer will need to explore at most half of the nodes in the lattice while locating the feature.

5.2. Locating features in Eclipse

We chose to locate the following features associated with two bugs in Eclipse (see Table 3): sorting by clicking on table header (associated with bug #34160) and renaming project source files (associated with bug #25457).

The search queries formulated are self-descriptive with respect to the features associated with bugs that we are locating. The terms from each bug description (extracted from Bugzilla) used as cues to formulate queries are highlighted in *italics* (see Table 3). Table 3 also includes the methods that were changed in order to fix the bugs (these are extracted from the official patches released to fix the bugs). Among those methods, the ones that occur first in the ranked links of results are in bold. The table also shows their rank in the list of the results.

For more details on these bugs the interested reader is referred to https://bugs.eclipse.org/bugs/.

5.3. Results and discussion

We studied how the number of documents and terms (attributes) affects the size and quality of the concept lattice. After some initial testing with different configurations of documents and attributes, we decided to keep the number of attributes in the range from 10 to 25 and study the generated concept lattices for the top 80 to 100 documents from the ranked list. One of the observations that we had while trying to use less than 10 attributes is that the clustering capacity was low in grouping related concepts, while when we tried to use more that 25 attributes, the number of concept nodes in the lattice became relatively high making lattices difficult to navigate.

Table 4 shows the results of applying FCA with different configurations of documents and attributes for two features to be located in the Eclipse source code. We computed the LDF and LBC measures for 20 concept lattices of different configurations and compared those with the simple ranked list. We did not include the four lattices constructed based on 80 top documents for bug #25457, as the ranking for the relevant method is 89, so it will not be included in any of these lattices.

LDF ranges from 15% to 318%, which indicates that even in the worst case scenario, FCA brings some improvement over the LSI ranking alone. Note that this measure is an upper bound on the behavior of developers (also note that we also make a worst case assumption that the user will have to read all documents associated with the nodes in the concept lattice, even though we consider the exploration strategy of concept lattices as optimal in this case).

The trade-off for different values of documents *n* and attributes *k* becomes clearer as we analyze the results. When using 25 attributes, we obtain the highest values of LDF (96.6%-318%) for any number of documents, but at the cost of larger lattice sizes (31-39 concept nodes), although the LBC values are the lowest, which again is a benefit (that is, we have larger lattices but they are easier to browse). Note that LDF is growing linearly with the number of attributes and the complexity factor, while LBC is decreasing linearly. On the other hand, the number of nodes, C, grows much faster. For example, the concept lattice created based on the first 100 documents for bug #34160 with 10 attributes consists of 17 nodes, whereas the same lattice but with 25 attributes contains 39 concept nodes. It is interesting to note that the number of nodes in the lattice that a developer needs to investigate is between 7 and 13 in any case.

Table 4. Experimental results for locating two features in Eclipse with 24 concept lattices of various configurations

Bug	Docs	Terms	P_{MBA}	LDF	C	C_{VIEW}	LBC
34160	100	10	0.02	**42.8%**	17	8	**47%**
34160	100	15	0.025	**78.5%**	25	11	**44%**
34160	100	20	0.026	**85%**	33	11	**33%**
34160	100	25	0.033	**96.6%**	39	10	**26%**
34160	90	10	0.023	**62.3%**	17	8	**47%**
34160	90	15	0.027	**98.4%**	24	11	**46%**
34160	90	20	0.029	**104%**	31	11	**36%**
34160	90	25	0.039	**174%**	36	11	**31%**
34160	80	10	0.026	**87.9%**	15	7	**47%**
34160	80	15	0.033	**138%**	22	10	**45%**
34160	80	20	0.034	**146%**	28	10	**36%**
34160	80	25	0.043	**207%**	33	10	**30%**
25457	100	10	0.013	**15%**	13	8	**62%**
25457	100	15	0.013	**15%**	17	9	**53%**
25457	100	20	0.019	**72%**	25	10	**40%**
25457	100	25	0.021	**91%**	33	10	**30%**
25457	90	10	0.013	**15%**	13	8	**62%**
25457	90	15	0.013	**15%**	17	9	**53%**
25457	90	20	0.017	**55%**	25	11	**44%**
25457	90	25	0.046	**318%**	32	13	**40%**

We manually analyzed all 20 concept lattices for which we computed the measures in Table 4. Due to space limitations it is not possible to present all the results here, however we present one reduced lattice, generated based on the query for locating the feature associated with bug #25457. The concept lattice is generated from 20 documents with 10 first descriptive terms (the concept lattices are visualized using Concept Explorer[7]), see Figure 3.

Figure 3 shows that the resulting lattice distillates relevant information well and the selected attributes are descriptive, allowing the user to explore the subset of search results effectively.

The results show that concept lattices are very effective in terms of grouping relevant information and the grouping effect is higher for larger attribute spaces. We obtained the best results when applying FCA over first 90 documents with 20-25 attributes. For the feature associated with bug #25457, considering 90 documents and 25 attributes the precision is four times better compared to the ranked list, while the lattice browsing complexity remains relatively low (40%).

[7] http://sourceforge.net/projects/conexp

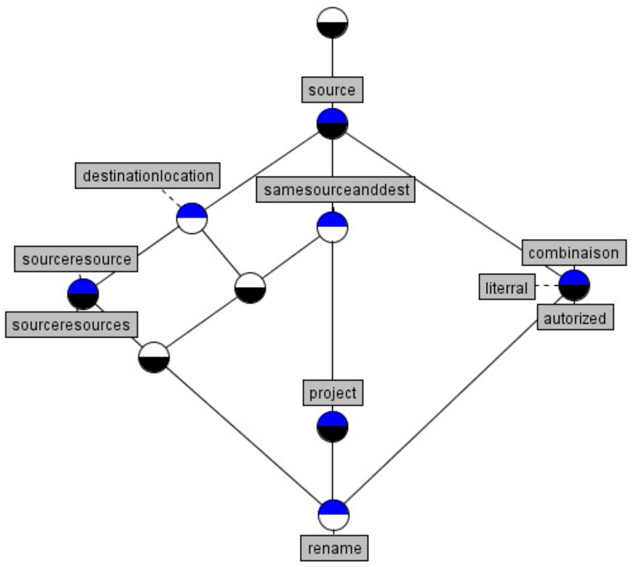

Figure 3. Concept lattice generated for the feature associated with bug #25457 from the first 20 documents and 10 selected terms

5.4. Threats to validity

Several issues may have affected the results of the case study and thus may limit generalizations. We made all efforts to minimize the effect of these issues.

One of the issues is that in our case studies we use the number of documents to build concept lattices that range from 80-100. However, if we do not have any relevant results in this range, we can not compute any of the measures we used for evaluation (for example consider the case with 80 documents for bug #25457).

Another issue is the extent to which the software and features used in the case study are representative to those actually used in practice. Although Eclipse is a real-world program this threat could be reduced if we experiment with other programs of different sizes and domains, as well as locating more concepts.

The queries formulated to obtain the LSI based rankings are dependent on the developer's knowledge, thus the results may be impacted by the actual query. However, as we discussed in the examples, the developer does not need to have an extensive knowledge of the source code to formulate LSI queries. Regardless of the query, the proposed approach is shown to help users understand search results better than simple ranked list. The gain over the ranked list alone is not affected as we use the same query.

The features may be implemented by more methods than those suggested by a patch, as correcting the problem may involve just part of the implementation. Once again the assessment of the gain remains valid, as both methods are equally influenced by this issue.

6. Conclusions and Future Work

The proposed concept location method, which combines information retrieval and formal concept analysis, provides very good results when considering a relatively small number of methods (100 out of 80,000), hence it is easy to use for software of any size.

Moreover, concept lattices are shown to be quite effective (up to four times improvement over simple ranking) in terms of grouping relevant information and labeling topics, concepts, and relationships between them, offering the user additional cues when exploring the results of a search.

We plan to move this research in several directions. First, we plan to compare our approach with at least two other different strategies on how to rank and select descriptive attributes to build concept lattices, for example terminological weighting formula and Okapi. Second, we plan on devising a heuristic-based approach to experiment with different strategies for selecting attributes, which may be specific to source code, for example, selecting only attributes that represent data types or only class or methods names, etc. Third, we plan to incorporate information about the rank of the method into the structure of the concept lattice, which may be helpful in terms of choosing the direction in the lattice. Finally, we plan to investigate the impact of concept lattices on query reformulation strategies, which we did not address in the current work.

7. Acknowledgements

This research was partially supported by grants from the US National Science Foundation (CCF-0438970) and the US National Institute of Health (NHGRI 1R01HG003491).

8. References

[1] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," *Proc. 11th IEEE Working Conf. on Reverse Engineering*, 2004, pp. 214-223.

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

[3] V. Rajlich and P. Gosavi, "Incremental Change in Object-Oriented Programming," in *IEEE Software*, July/August 2004, pp. 2-9.

[4] N. Wilde, T. Gust, J. A. Gomez, and D. Strasburg, "Locating User Functionality in Old Code," *Proc. IEEE Conf. on Software Maintenance,* 1992, pp. 200-205.

[5] G. Antoniol and Y. G. Guéhéneuc, "Feature Identification: An Epidemiological Metaphor," *IEEE Transactions on Software Engineering,* vol. 32, no. 9, pp. 627-641, 2006.

[6] A. D. Eisenberg and K. De Volder, "Dynamic Feature Traces: Finding Features in Unfamiliar Code," *Proc. 21st IEEE Int. Conf. on Software Maintenance*, Budapest, Hungary, 2005, pp. 337-346.

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

[8] D. Poshyvanyk, M. Petrenko, A. Marcus, X. Xie, and D. Liu, "Source Code Exploration with Google" *Proc. 22nd IEEE Int. Conf. on Software Maintenance*, Philadelphia, PA, 2006, pp. 334 - 338.

[9] K. Chen and V. Rajlich, "Case Study of Feature Location Using Dependence Graph," *Proc. 8th IEEE Workshop on Program Comprehension,* 2000, pp. 241-249.

[10] M. Robillard, "Automatic Generation of Suggestions for Program Investigation," *Proc. Joint European Software Engineering Conf. and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Lisbon, Portugal, 2005, pp. 11 - 20

[11] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "SNIAFL: Towards a Static Non-interactive Approach to Feature Location," *ACM Trans. on Software Engineering and Methodologies,* vol. 15, no. 2, pp. 195-226, 2006.

[12] N. Gold, M. Harman, Z. Li, and K. Mahdavi, "Allowing Overlapping Boundaries in Source Code using a Search Based Approach to Concept Binding," *Proc. 22nd IEEE Int. Conf. on Software Maintenance (ICSM'06)*, Philadelphia, PA, 2006, pp. 310-319.

[13] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev, "Static Techniques for Concept Location in Object-Oriented Code," *Proc. 13th IEEE Int. Workshop on Program Comprehension*, 2005, pp. 33-42.

[14] T. Eisenbarth, R. Koschke, and D. Simon, "Locating Features in Source Code," *IEEE Transactions on Software Engineering,* vol. 29, no. 3, pp. 210 - 224, March 2003.

[15] M. Salah and S. Mancoridis, "A hierarchy of dynamic software views: from object-interactions to feature-interactions," *Proc. 20th IEEE Int. Conf. on Software Maintenance*, Chicago, IL, 2004, pp. 72-81.

[16] D. Poshyvanyk, Y. Gael-Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval," *IEEE Transactions on Software Engineering,* to appear, 2007.

[17] N. Wilde, M. Buckellew, H. Page, V. Rajlich, and L. Pounds, "A Comparison of Methods for Locating Features in Legacy Software," *J. of Systems and Software,* vol. 65, no. 2, pp. 105-114, February 15 2003.

[18] S. Simmons, D. Edwards, N. Wilde, J. Homan, and M. Groble, "Industrial tools for the feature location problem: an exploratory study," *J. of Software Maintenance: Research and Practice,* vol.18, no. 6, pp. 457-474, 2006.

[19] G. Snelting, "Concept Lattices in Software Analysis," *Proc. Formal Concept Analysis*, 2005, pp. 272-287.

[20] P. Tonella and M. Ceccato, "Aspect Mining through the Formal Concept Analysis of Execution Traces," *Proc. 11th IEEE Working Conf. on Reverse Engineering*, 2004, pp. 112 - 121

[21] S. Breu, T. Zimmermann, and C. Lindig "Mining Eclipse for Cross-Cutting Concerns," *Proc. Int. Workshop on Mining Software Repositories*, 2006, pp. 94 - 97

[22] K. Mens and T. Tourwe, "Delving source code with formal concept analysis," *Computer Languages, Systems & Struct.,* vol. 31, no. 3-4, pp. 183-198, Oct.-Dec. 2005.

[23] B. Ganter and R. Wille, *Formal Concept Analysis*. Berlin, Heidelberg, New York: Springer-Verlag, 1996.

[24] A. Marcus, J. I. Maletic, and A. Sergeyev, "Recovery of Traceability Links Between Software Documentation and Source Code," *Int. J. of Software Engineering and Knowledge Engineering,* vol. 15, no. 4, pp. 811-836, October 2005.

[25] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Can Information Retrieval Techniques Effectively Support Traceability Link Recovery?" *Proc. 14th IEEE Int. Conf. on Program Comprehension*, Athens, Greece, 2006, pp. 307-316.

[26] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Advancing candidate link generation for requirements tracing: the study of methods," *IEEE Trans. on Software Engineering,* vol. 32, no. 1, pp. 4-19, January 2006.

[27] M. Lormans and A. Van Deursen, "Can LSI help Reconstructing Requirements Traceability in Design and Test?," *Proc. 10th IEEE European Conf. on Software Maintenance and Reengineering*, 2006, pp. 47-56.

[28] D. Poshyvanyk, A. Marcus, Y. Dong, and A. Sergeyev, "IRiSS - A Source Code Exploration Tool," *in Tool and Demo Proc. 21st IEEE Int. Conf. on Software Maintenance*, 2005, pp. 69-72.

[29] D. Poshyvanyk, A. Marcus, and Y. Dong, "JIRiSS - an Eclipse plug-in for Source Code Exploration," *Proc. 14th IEEE Int. Conf. on Program Comprehension*, Athens, Greece, 2006, pp. 252-255.

[30] J. I. Maletic, M. L. Collard, and A. Marcus, "Source Code Files as Structured Documents," *Proc. 10th IEEE Int. Workshop on Program Comprehension*, Paris, France, 2002, pp. 289-292.

[31] R. Ferenc, I. Siket, and T. Gyimóthy, "Extracting facts from open source software," *Proc. 20th Int. Conf. on Software Maintenance*, 2004, pp. 60-69.

[32] A. Penas, F. Verdejo, and J. Gonzalo, "Corpus-Based Terminology Extraction Applied to Information Access," *Corpus Linguistics,* no. 2001.

[33] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic Clustering: Identifying Topics in Source Code," *Information and Software Technology,* vol. 49, no. 3, March 2007, pp. 230-243.

[34] J. Cigarran, A. Peñas, J. Gonzalo, and F. Verdejo, "Evaluating Hierarchical Clustering of Search Results," *Proc. 12th Int. Conf. on String Processing and Information Retrieval (SPIRE'05)*, 2005, pp. 49-54.

[35] R. K. Yin, *Applications of Case Study Research*, 2 ed. CA, USA: Sage Publications, Inc, 2003.

[36] J. M. Cigarrán, J. Gonzalo, A. Peñas, and F. Verdejo, "Browsing Search Results via Formal Concept Analysis: Automatic Selection of Attributes," *Proc. 2nd Int. Conf. on Formal Concept Analysis*, Sydney, Australia, 2004, pp. 74-87.