COMBINING INFORMATION RETRIEVAL MODULES AND STRUCTURAL INFORMATION FOR SOURCE CODE BUG LOCALIZATION AND FEATURE LOCATION

by

PENG SHAO

RANDY K. SMITH, COMMITTEE CHAIR

NICHOLAS A. KRAFT, CO-CHAIR

TRAVIS ATKISON
JEFFREY C. CARVER
ALLEN S. PARRISH

A DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2011

Copyright Peng Shao 2011

ALL RIGHTS RESERVED

ABSTRACT

Bug localization and feature location in source code are software evolution tasks in which developers use information about a bug or feature present in a software system to locate the source code elements, such as classes or methods. These classes or methods must be modified either to correct the bug or implement a feature. Automating bug localization and feature location are necessary due to the size and complexity of modern software systems. Recently, researchers have developed static bug localization and feature location techniques using information retrieval techniques, such as latent semantic indexing (LSI), to model lexical information, such as identifiers and comments, from source code. This research presents a new technique, LSICG, which combines LSI modeling lexical information and call graphs to modeling structural information. The output is a list of methods ranked in descending order by likelihood of requiring modification to correct the bug or implement the feature under consideration. Three case studies including comparison of LSI and LSICG at method level and class level of granularity on 25 features in JavaHMO, 35 bugs in Rhino, 3 features and 6 bugs in jEdit demonstrate that The LSICG technique provides improved performance compared to LSI alone.

LIST OF ABBREVIATIONS AND SYMBOLS

A	A matrix
A^T	The transpose of A
ADAMS	ADvanced Artifact Management System: a traceability recovery tool
AMAP	Automatically Mining Abbreviation expansions in Programs: an automated approach to mining abbreviation expansions from source code
CA	Cognitive assignment: an approach for concept location.
CG	Call graph: the calling information for a method or class.
COCONUT	COde COmprehension Nurturant Using Traceability: an approach to indicate the similarity between the source code under development and a set of high-level artifacts.
D_i	A Document
d_i	A vector in a document
df	Degrees of freedom: number of values free to vary after certain restrictions have been placed on the data

FCA	Formal concept analysis: a technique to generate a concept lattice that organizes the most relevant attributes of the documents.
H_0	The null hypotheses
H_1	The alternative hypotheses
H_{CG}	A hash function to return LSI score for a method
IR	Information retrieval: a technique to retrieve information
IRRF	Information retrieval with relevance feedback: an approach for concept location
LDA	Latent Dirichlet Allocation: an IR module
L_{LSI}	A ranked list of methods or classes based on their LSI scores
L_{LSICG}	A ranked list of methods or classes based on their LSICG scores
LSA	Latent semantic analysis: the same as LSI.
LSI	Latent Semantic Indexing: an IR module
LSICG	Latent semantic indexing and call graph: a technique which combines lexical information and structural information for bug localization and feature location
$L_{THRESHOLD}$	A minimum similarity score threshold
M	A method
m_i	A method with index number of i

MMS	Multiple Minimum Support: a technique to identify hidden relationships between classes, methods and member data.
N	The total number of words
N_{CG}	The number of call graph nodes for a method or class
N_{dq}	The total number of relevant Documents for a query
N_{dqr}	The total number of relevant Documents retrieved for a query
N_{dr}	The total number of documents retrieved for a query.
N_T	The total number of call graph nodes for methods or classes which are in $L_{THRESHOLD}$
p	Probability associated with the occurrence under the null hypothesis of a value as extreme as or more extreme than the observed value
Pr	Probability
PROMESIR	A technique for bug localization
Q	A query
q	A query
RETRO	REquirements TRacing On-target: a requirements tracing tool
RTC	Requirement Centric Traceability: an approach to analyze the change impact at the requirement level.

S	A diagonal matrix whose diagonal elements are the singular values of **A** in decreasing order.
S_{CG}	Call graph score
S_{LSI}	LSI score
SPR	Scenario-based probabilistic ranking: a technique for bug localization
SVD	Singular value decomposition: a decomposition technique
t	Computed value of *t* test
tf-idf	Term-frequency-inverse document frequency: a metric weighting terms in corpus.
TREC	Text Retrieval Conference
U	A matrix whose columns are the eigenvectors of the AA^T matrix.
V	A matrix whose columns are the eigenvectors of the $\mathbf{A^TA}$ matrix.
V^T	The transpose of **V**.
W	Computed value of Wilcoxon's signed-rank test
w	A word
WTA	Winner Takes All: a clustering algorithm
λ	A weight in the range (0,1) that represents confidence in the ability of S_{LSI} or S_{CG}
$<$	Less than
$=$	Equal to

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Smith for providing me the opportunity to be in this PhD program. I appreciate his help, guidance and patience for me to complete the degree.

I would also like to thank my co-advisor Dr. Kraft for his help. His intelligent ideas, writing skills, advices and patience are great help for me.

Furthermore, I would like to thank my committee members: Dr. Atkison, Dr. Carver and Dr. Parrish for their guidance and advices to my dissertation and research.

This research would not have been possible without the support from my family and friends. Finally I thank all faculty, staff and students at the University of Alabama who helped with my study and research.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Factors such as software aging, documentation deficiency, and developer mobility can contribute, individually or in concert, to a lack of information about a software system and thus can render the software system difficult to understand. Consequently, these factors hinder software evolution, because software understanding activities consume 50% to 90% of effort during software evolution (Müller et al., 2000). This hindrance increases the cost of software evolution, which is the most costly phase of software development and accounts for up to 80% of total software cost (Boehm, 1981; Standish, 1984; Alkhatib, 1992; Erlikh, 2000). To help reduce developer effort, and thereby curb total software cost, many recent research efforts focus on partially or fully automating software evolution tasks such as development of traceability links (Maeder et al., 2006), bug localization (Lukins et al., 2010), and feature location (Poshyvanyk et al., 2007).

Bug localization and feature location are software evolution tasks in which a developer uses information about a bug or feature present in a software system to locate the source code elements that must be modified to correct the bug or changed to implement the feature. The developer extracts information about the bug from the title and description of the corresponding bug report, or extracts the feature information from feature description or change requests, and then uses that information to formulate a query. The developer then uses that query to search the software system and to determine which parts of the source code must be changed to correct the bug or apply the feature. Bug localization and feature location (Liu et al., 2007; Poshyvanyk et

al., 2007; Shao et al., 2009), are instances of the concept assignment problem (Biggerstaff et al., 1993).

Effective automation of bug localization and feature location is necessitated by the size and complexity of modern software systems. Automated bug localization and feature location techniques are recommender systems (Haruechaiyasak et al., 2006) that take as input information about a software system and produce as output a ranked list of source code elements such as classes, methods, or statements. The top-ranked source code element is the one most likely to require modification to correct the bug or implement the feature under consideration (and so on down the list). The developer starts at the top-ranked element and examines each element in turn until reaching an element that actually needs to be modified to correct the bug or apply the feature. From this first pertinent element, the developer can locate other related elements through their coding links with that element (Lukins et al., 2010).

Static bug localization techniques and static feature location techniques (Zhao et al., 2006) operate on the source code or a model of the source code, while dynamic bug localization techniques and dynamic feature location techniques (Maia et al., 2008) operate on execution traces. Unlike dynamic bug localization and feature location techniques, static bug localization and feature location techniques require neither a working software system, nor a test case (or test suite) that triggers the bug. An additional prerequisite for the application of dynamic bug localization and feature location techniques, but not static bug localization and feature location techniques, is source code instrumentation. However, static bug localization and feature location techniques typically return a ranked list of methods, while dynamic bug localization and feature location techniques typically return a ranked list of statements.

Recently, researchers have developed automated static bug localization and static feature location techniques (Poshyvanyk et al., 2007; Liu et al., 2007; Lukins et al., 2010) using information retrieval (IR) models such as latent semantic indexing (LSI) (Deerwester et al., 1990) and latent Dirichlet allocation (LDA) (Blei et al., 2003). These techniques show efficacy but leave room for improvement. Indeed, though structural information (e.g., inheritance relationships between classes or calling relationships between methods) is statically determinable, these techniques use only lexical information (i.e., terms from identifiers, comments, and possibly string literals).

This research details a static bug localization and feature location technique that uses lexical and structural information to provide improved performance compared to an existing static bug localization and feature location technique that uses only lexical information. In particular, the technique uses call graphs and terms from identifiers and comments when computing a ranked list of methods or classes. The computation assigns a score for each method or class using a formula that integrates two values: textual similarity, determined using LSI and cosine distance, and method or class proximity, determined using a call graph metric based on graph distance.

The primary contributions of this research are:

- A description of an automated static bug localization and feature location technique that combines lexical and structural information to compute a ranked list of methods or classes.
- An extensive study applying the technique to three medium-sized Java projects at both the method level and class level of granularity. The technique is applied to

25 features in JavaHMO, 35 bugs in Rhino, 3 features and 6 bugs in jEdit. The results demonstrate that the combined approach provides improved performance compared to the current automated static bug localization and feature location technique that uses only lexical information.

The next chapter provides the background of information retrieval techniques. Chapter 3 reviews the research that relates to the technique including bug localization, feature location, source code retrieval and LSI. Chapter 4 describes the research method. Chapter 5 details the research study with study design, results and analysis. Chapter 6 provides the discussion of the research. Threats to validity are given in Chapter 7. Finally, Chapter 8 presents the conclusion and describes future work.

CHAPTER 2

BACKGROUND

Chapter 2 provides background information on information retrieval techniques. Details of information retrieval techniques are given first followed by applications of IR to software evolution tasks.

2.1 Information Retrieval Techniques

IR techniques have been applied to mine free text documents from source code. It succeeds in managing a large amount of documentation and being a useful support tool for the development of software projects. IR is the science and technology concerned with the effective and efficient retrieval of information for the subsequent use by interested parties. It searches information within different repositories such as documents, metadata of documents, related databases as well as World Wide Web. Typically, users enter queries describing the needs into an IR system, which finds and returns a set of related objects among a large number of collections. Objects are repositories of information and can be free text documents, videos and images. The IR system finds a set of objects that satisfy the query and calculates a related score on how exactly each object matches the query. Then it ranks objects by their score and provides the top scored object to users.

Two IR metrics, recall and precision (Frakes et al., 1992), are used in case study to evaluate the accuracy of the technique.

$$\text{Recall} = \frac{N_{dqr}}{N_{dq}}$$

Where N_{dqr} is the total number of relevant Documents retrieved for the query q, and N_{dq} is the total number of relevant Documents for q.

$$\text{Precision} = \frac{N_{dqr}}{N_{dr}}$$

Where N_{dqr} has the same definition as it is in Recall, and N_{dr} is the total number of documents retrieved for query q.

There are two popular IR models, Probabilistic IR model and Vector Space IR model, used to establish links between free text documentation and source code. In the two models, a set of documents are collected and then ranked based on their relationship to a user's query.

2.1.1 Probabilistic Module

In a probabilistic IR model, the relationship between a Document D_i and a query Q is calculated by the probability that D_i is related to Q, which is Pr (D_i|Q).

Using Baye's Theorem to Pr (D_i|Q), there is:

$$\Pr(D_i|Q) = \frac{\Pr(Q|D_i)\Pr(D_i)}{\Pr(Q)}$$

It is reasonable to assume all documents should have the same probability to be related to a query Q as well as Pr (Q) is the same for all D_i by given source code. Thus D_i that is related to Q

can be ranked by Pr ($Q|D_i$). A query Q consists of a sequence of words w_1, w_2, w_3,..., w_m based on the hypothesis that Q and D_i have the same vocabulary V. Therefore:

$$\Pr(D_i|Q) = \Pr(w_1, w_2, w_3, ..., w_m \mid D_i)$$

$$= \Pr(w_1 \mid D_i) \prod_{k=2}^{m} \Pr(w_k|w_1, w_2, w_3, \ldots, w_{k-1}, Di) \quad (1)$$

A simplification is used on the above formula to avoid the difficulty to calculate the probability if m is too large. With n<m, there is:

$$\Pr(w_1 \mid D_i) \prod_{k=2}^{m} \Pr(w_k|w_1, w_2, w_3, \ldots, w_{k-1}, Di)$$

$$\approx \Pr(w_1,w_2,w_3,...,w_{n-1}|D_i)\prod_{k=2}^{m} \Pr(w_k|w_{k-n+1}, \ldots, w_{k-1}, Di) \quad (2)$$

Here an assumption that all words are independent, i.e. k=1, is used to calculate Pr ($D_i|Q$), so based on formula (1) and (2), we have:

$$\Pr(D_i|Q) \approx \prod_{k=1}^{m} \Pr(w_k|Di)$$

If a word w_k does not exist in D*i*, Pr (w_k |D_i) would be zero. To avoid this problem, two parameters are introduced:

$$\lambda = \frac{n}{N*|V|}\beta$$

$$\beta = \frac{n(1)}{n(1)+2*n(2)}$$

where n is the number of total different words in D_i, N is the total number of words, and n(j) is number of words occurring j times in D_i.

Then $Pr(w_k|D_i) = \begin{cases} \frac{c_k - \beta}{N} + \lambda & \text{if } w_k \text{ exsits in } Di \\ \lambda & \text{otherwise} \end{cases}$

where c_k is the number of w_k occurs in D_i.

A well known probabilistic IR module is Robertson and Jones module (Robertson et al., 1977), which estimates the probability that each document is related to a query. Fuhr 1989 and Turtle et al. 1991 both provided probabilistic IR modules considering the integration of indexing and retrieval modules. Wong 1989 proposed a module in which they use a probability distribution to represent documents and a maximum likelihood estimator for term probabilities. Crestani et al. 1998 compared four probabilistic IR models using different formulas: Joint Probability, Conditional Probability, Logical Imaging (Crestani et al., 1995), General Logical Imaging (Lewis et al., 1981). They executed the four models on three cases, and analyzed the transfer of probabilities occurring in the term space at retrieval time for the four models. The results indicate that the ranking of the four models based on their performances (precision and recall) from high to low is:

1) General Logical Imaging: the model with the one-to-many probability transfer and similarity.

2) Conditional Probability: the model with one-to-many transfer and ratio.

3) Logical Imaging: the model with one-to-one transfer.

4) Joint Probability: the model with no transfer.

2.1.2 Vector Space Module

In vector space approaches, documents and queries are represented as vectors (J. Konclin et al., 1988; B. Ramesh et al., 1992; Crestani, F. et al, 1995; Pinhero et al., 1996). Given vocabulary |V|, a document D_i can be represented as [d_i,*1*, d_i,*2*, …d_i,|*V*|].

There are several methods to calculate the weight of each term, and a well-known one is term-frequency-inverse document frequency, *tf-idf* (Frakes et al., 1992). $tf_{i,j}$ is the term frequency of *j*th element in D_i. $tf_{i,j} = n_{i,j}/N$, where $n_{i,j}$ is the number of occurrence of *j*th element in D_i, and N is the total number of words in D_i.

idf_j is the inverse document frequency, which represents the general importance of a term, and $idf_j = \log (|D|)/(|D_j|)$, where |D| is the total number of documents and D_j is total number of documents which include term *j*. Therefore, $d_{i,j} = tf_{i,j} * idf_j$.

A query Q can be also represented as a vector | q_i,*1*, q_i,*2*, …q^i,|*V*||. And the similarity of D_i and Q is calculated as:

$$\text{Similarity } (D_i, Q) = \frac{\sum_{j=1}^{V} di,jqj}{\sqrt{\sum_{h=1}^{V}(di,h)^2 * \sum_{k=1}^{V}(qk)^2}} \text{ (Salton et al. 1988)}$$

Salton et al. 1988 used 6 different cases to compare vector space IR models using different metrics to calculate the vector distance of Document and Query. The results showed that two models had better performance than others. The first is Best fully weighted Model, and the second is weighted model without using inverse frequency for documents. Ponte et al. 1998 compared the performance of a vector space model (tf-idf) and their probabilistic model by using two different TREC (Text Retrieval Conference) data sets. Their results indicate that

probabilistic model obtained higher precision than vector space model. Antoniol et al. 2002 performed the same two models on two cases, one C++ project and one Java project. The results showed that both models achieve 100% of recall with almost the same number of documents retrieved. The probabilistic model achieved a high percent of recall with a smaller number of documents retrieved and performed better if 100% of recall was required. The vector space model, with less effort on query construction and document representations, started at a low percent of recall and achieved 100% of recall in a regular progress when the number of retrieved document was increasing. The precision for both case studies was low, due to the fact that not all the classes have association with documents.

2.2 IR Applications

IR succeeds in managing large amounts documentation and is a useful support tool for the development of software projects. These techniques can be applied to many software engineering tasks.

2.2.1 Program comprehension

Program comprehension is the process used by developers to understand programs. This process usually occurs during a program's evaluation before modification. Program comprehension is critical before a change is made. It is useful and important for software documentation, visualization, program design and analysis, refactoring and reengineering. Different techniques and tools such as categorization and abbreviation expansion are applied to assist program comprehension. This chapter reviews the research related to program comprehension.

Lu et al. 2005 and Lu et al. 2007 developed a functionality-based categorization of JavaScript, then view understanding embedded scripts as a text categorization problem. They

indicated how traditional information retrieval methods can be augmented with the features distilled from the domain knowledge of JavaScript and software analysis to improve classification performance. Their program comprehension included that static and dynamic features alone do not perform well, but their combination greatly reduces individual mistakes. The combined feature set also does not outperform the simple lexical approach, but serves to augment its performance.

Kanellopoulos et al. 2007 presented a methodology and an associated model for extracting information from object oriented code by applying clustering and association rules mining. K-means clustering produces system overviews and deductions, which support further employment of an improved version of **M**ultiple **M**inimum **S**upport (MMS) Apriori that identifies hidden relationships between classes, methods and member data. By such a novel algorithmic framework which combines two different kinds of data mining algorithms, one for clustering and one for mining association rules from code, developers could have a more comprehensive view of the system under maintenance, at various levels of abstraction.

Hill et al. 2007 proposed a technique that retrieves neighborhood information for a software component. Their tool Dora was compared with a structural technique Suade, and two base line techniques: Boolean-AND (AND) and Boolean-OR (OR), and Dora performed best. Their integrated lexical-based and structural-based approach was significantly more effective in helping programmers explore programs.

Maskeri et al. 2008 investigated latent Dirichlet allocation (LDA) in the context of comprehending large software systems and proposed a human assisted approach based on LDA for extracting domain topics from source code. Their results indicate that their tool was able to satisfactorily extract some of the domain topics but not all, and certain human input is needed in

order to improve the quality of topics extracted.

Hill et al. 2008 presented an automated approach to mining abbreviation expansions from source code to enhance software maintenance tools that utilize natural language information. Their tool **A**utomatically **M**ining **A**bbreviation (AMAP) expansions in **P**rograms used contextual information at the method, program, and general software level to automatically select the most appropriate expansion for a given abbreviation. AMAP is helpful for developers to understand the abbreviations in programs.

Hill et al. 2009 discussed an approach that automatically extracts natural language phrases from source code identifiers and categorizes the phrases and search results in a hierarchy. Their technique allowed developers to explore the word usage in a piece of software, helping them to quickly identify relevant program elements for investigation or to quickly recognize alternative words for query reformulation.

2.2.2 Impact analysis

Impact analysis is used to evaluate the impact that a proposed change in one part of a program may have on another (Arnold et al. 1993; Turver et al., 1994; Fyson et al., 1998).

Antoniol et al. 2000 propose an IR based method for impact analysis. The authors use a vector space model and a probabilistic model to trace maintenance requests onto software components that are affected by the requests. In their case study of LEDA, they use change log files to extract 11 maintenance requests.

Canfora et al. 2006 provided an approach to predict impacted files from a change request definition. Their approach exploited information retrieval algorithms performed on code entities, such as source files and lines of code, indexed with free text contained in software repositories. Their results indicated that indexing fine grained entities improved precision at the cost of

indexing a much higher number of code entities.

Tan et al. 2006 presented a technique to investigate the impacts of the new factors on software errors. They analyzed bug characteristics by first sampling hundreds of real world bugs in two large, representative open-source projects. Their findings included: (1) memory-related bugs have decreased because quite a few effective detection tools became available recently; (2)some simple memory-related bugs such as NULL pointer dereferences that should have been detected by existing tools in development are still a major component, which indicates that the tools have not been used with their full capacity; (3)semantic bugs are the dominant root causes, as they are application specific and difficult to fix, which suggests that more efforts should be put into detecting and fixing them; (4) security bugs are increasing, and the majority of them cause severe impacts.

Li et al. 2008 introduces a Requirement Centric Traceability (RCT) approach to analyze the change impact at the requirement level. The RCT combines with the requirement interdependency graph and dynamic requirement traceability to identify the potential impact of requirement change on the entire system in late phase. Their contributions included: 1. The approach addresses the change on the requirement specification rather than code, which can help the non-technical people who don't know the detail of the code to predict the change cost quantitatively. 2. The impact to both requirements and artifacts was unified into the number of artifacts, which is easy to use and understand. 3. It is time-saving to apply the IR to automatically establish the traces among the requirements and artifacts, and is highly probable to adopt in practice. Without the dynamic generating approach, the analyst has to analyze every possible relation among requirements and artifacts

Lindvall et al. 2009 proposed the creation of Semantic Networks to search for relevant

software change artifacts. It convey such relationships and assist in automatically discovering not only the requested artifacts based on a user query, but additional relevant ones that the user may not be aware of. Experiments results show that the combination of semantic networks and context significantly improve both the precision and recall of search results.

2.2.3 Traceability Recovery

Traceability recovery is the process to recover the link between requirements and their corresponding design artifacts, source code components as well as test cases. Traceability is an important process and helpful in program comprehension, impact analysis, and reuse of existing software, etc.

Cleland-Huang et al. 2005 provided a Goal Centric approach for effectively maintaining critical system qualities such as security, performance, and usability throughout the lifetime of a software system. Their case study results indicated the feasibility of using a probabilistic approach to dynamically retrieve traceability links for non-functional requirements. The imprecision problems introduced through use of this method are largely mitigated through user inspection of retrieved links and through establishing a sufficiently low threshold that minimizes the number of omission errors. Although users' feedback is required to filter out unwanted links, the effort is only a small fraction of that which would be required to perform the trace manually.

ClelandHuang et al. 2007 further applied the Goal Centric approach to detect and classify stakeholders' quality concerns across requirements specifications containing scattered and non-categorized requirements, and also across freeform documents such as meeting minutes, interview notes, and memos.

Hayes et al. 2007 presented REquirements TRacing On-target (RETRO), a special purpose requirements tracing tool and discusses how RETRO automates the generation of RTMs

and present the results of a study comparing manual RTM generation to RTM generation using RETRO. The study showed that RETRO found significantly more correct links than manual tracing and took only one third of the time to do so.

Li et al. 2008 introduces a Requirement Centric Traceability (RCT) approach to analyze the change impact at the requirement level, as we talked about in Impact Analysis section.

2.2.4 Reuse

Reuse is the use of existing software. The purpose of software reuse is to save time and energy by reducing redundant work.

Etzkorn et al. 2001 described the analysis, in terms of quality factors related to reusability, contained in an approach that aids significantly in assessing existing OO software for reusability. An automated tool implementing the approach is validated by comparing the tool's quality determinations to that of human experts. The comparison provides insight into how OO software metrics should be interpreted in relation to the quality factors they purport to measure.

Chang and Mockus 2008 evaluated a simple-to-use method that needs only a set of file pathnames to identify directories that share filenames and improve reuse detection at the file level. The authors applied the method and four additional file copy detection methods that utilize the underlying content of multiple versions of the source code on the FreeBSD project. Their tool extended the concept of copy detection to the comparison files having multiple versions and exemplified the methods and the validation process on FreeBSD CVS version repository.

2.2.5 Concept Location

Concept Location is the process to locate the implementation of a concept in source code. The assumption of concept location is that the developers understand the concept, but do not know where in source code they are located. There are several definitions of concept. A concept could

be an object in object-oriented programs, or it could be a set of attributes in source code. Rajlich and Wilde 2002 defined concepts as units of human knowledge that can be processed by the human mind (short-term memory) in one instance.

Etzkorn et al. 1996 addresses an approach to the automated understanding of object-oriented code and a knowledge-based system that implements the approach. The understanding approach worked fairly well. Keyword recognition has been the primary source of information, since the packages analyzed have all been fairly sparsely commented. Some false recognition does occur when only keywords from identifiers are available.

Liu et al. 2001 and Liu et al. 2002 used intelligent search techniques to automate the process of transforming a query string, so that it will find suitable results, in the context of source code. These techniques include abbreviation expansion, abbreviation contraction, as well as techniques based on dictionary or knowledgebase lookup. Their methods are relatively simple to implement, and are shown to be useful to Software Engineers in preliminary experiments.

Marcus and Maletic 2001 apply LSI to identifiers and comments to detect similar high-level concepts such as abstract data types. In a case study of Mosaic, the authors demonstrate that their approach outperforms a simple word matching method. Though their approach is easy to implement and has a low computational cost, it lacks some precision and is not well automated. Marcus et al. 2004 again applied LSI for concept location and their results show that LSI is almost as easy and flexible to use as grep based techniques and it provides better results, and LSI was able to identify certain parts of a concept that was missed by the dependence graph search approach. Marcus et al. 2005 further applied three different techniques: grep, Dependency Search, LSI, for concept location in Object-Oriented projects. They focused on static concept location techniques that share common prerequisites and are search the source code using regular

expression matching, or static program dependencies, or information retrieval. The paper analyses these techniques to see how they compare to each other in terms of their respective strengths and weaknesses. Their findings included 1. Of the three location techniques used, the dependency search technique utilizes the class structure to the largest degree; even there the programmer occasionally took a wrong turn in the search and had to backtrack. Hence, they concluded that the OO structure of the software is not always sufficient to allow programmers to easily and unerringly select the appropriate class as the concept location and thus, the use of the concept location techniques is necessary. 2. Compared to other structuring principles, OO structuring does not provide any advantage for concept location. 3. The concepts may be delocalized among several classes. 4. The grep-based technique is very much dependent on the developer's existing knowledge about the system and the problem domain. 5. The static dependency search is the most structured technique among these three and it can be used without any specific tool support since the dependencies can be followed manually through the code. 6. The IR-based technique suffers from similar problem as the grep-based technique.

Poshyvanyk and Marcus 2007 combine LSI and formal concept analysis (FCA) for concept location. They use LSI to map concepts expressed as queries to relevant parts of source code and return a ranked list of results. Next, they use FCA to generate a concept lattice that organizes the most relevant attributes of the documents. In their case study they apply the technique to Eclipse to demonstrate its effectiveness. The limitations of the approach are in the need to select an appropriate number of documents (they use 80 to 100) and the need to formulate appropriate queries.

Baysal et al. 2007 proposed a method to find relations between two artifacts, discussion archives and source code and provide a possible path forward for designing techniques and

approaches to monitor, plan, and predict software changes. Their conceptual correlation can provide useful recommendations about source code modifications, and identify a set of correlation patterns between discussion and changed code vocabularies and discover that some releases referred to as minor should instead fall under the major category. These patterns can be used to give estimations about the type of a change and time needed to implement it.

Fry et al. 2008 presented strategies for extracting precise verb–DO pairs from method signatures and comments, with the goal of identifying verb–DO pairs that accurately represent the behavior of each method.

Eaddy et al. 2008 propose CERBERUS a hybrid technique for concept location that combines information retrieval, execution tracing, and a new technique, prune dependency analysis. First, an IR ranking and an execution trace ranking are produced. Next, the PROMESIR framework is used to combine the rankings. Select seeds for the prune dependency analysis by applying a threshold to the combined ranking to identify highly relevant source code elements. Finally, apply the prune dependency analysis to the seeds to infer additional pertinent elements. The case study of Rhino shows that CERBERUS outperforms the constituent techniques individually or in pairs and that prune dependency analysis boosts the performance of IR and execution tracing.

Cleary et al. 2009 present cognitive assignment (CA), a new approach to concept location. CA uses information flow and co-occurrence information from non-source code documents to implement a query expansion based concept location technique that utilizes implicit information available in system documentation. The case study uses non-source code documents of the JDT and compares CA to other techniques. CA performs as well as or better than the other techniques.

Gay et al. 2009 present information retrieval with relevance feedback (IRRF) for concept location. IRRF combines the Apache Lucene implementation of the vector space model and explicit relevance feedback. The results of three case studies on Eclipse, jEdit, and Adempiere indicate that IRRF effectively reduces developer effort. The advantage of IRRF is that it eases the burden on developers to formulate high quality queries to obtain good results from IR based concept location techniques. Moreover, it explicitly captures knowledge gained by developers during the concept location process. However, performance does suffer when the initial query is poor.

2.2.6 Other Applications

Gui and Scott 2005 presented an approach to software component ranking intended for use in searching for such components on the Internet. The method used introduces a novel method of weighting keywords that takes account of where within the structure of a component the keyword is found. This hierarchical weighting scheme is used in two ranking algorithms: one using summed weights, the other using a vector space model. The results demonstrate the consistent superiority of the hierarchical weighting algorithms. The vector space algorithm performed slightly better than the simple summed weight method. It also demonstrates the importance of component structure to deriving an accurate ranking of the relevance of components to queries.

Canfora et al. 2006a proposed an approach to select the best candidate set of developers able to resolve a given change request based on past similar changes. This approach is useful for project managers in order to choose the best candidate to resolve a particular change request and/or to construct a competence database of developers working on software projects.

Lawrie et al. 2006 applied language processing techniques to extend human judgment

into situations where obtaining direct human judgment is impractical due to the volume of information that must be considered. Their tool QALP can be used to evaluate third-party coded subsystems, to track quality across the versions of a program, to assess the compression effort (and subsequent cost) required to make a change, and to identify parts of a program in need of preventative maintenance.

DeLucia et al. 2007a and DeLucia et al. 2007b presented an approach based on Winner Takes All (WTA), a competitive clustering algorithm, to support the comprehension of static and dynamic Web applications during Web application reengineering. This approach adopts a process that first computes the distance between Web pages and then identifies and groups similar pages using the considered clustering algorithm. Their findings include:

1) In most cases the trade-off configuration is very close to the break-even configuration.
2) When the density of actual pairs of similar pages is high the approach generally produced better results in terms of precision and recall values.
3) In case the trade-off and the break-even configurations are different and the Web applications have a large number of pages similar at the structural level, the trade-off and the break-even configurations produced similar results.
4) Case when the Web applications present only few static or dynamic pages similar at the structural level, the break-even configuration and the trade-off configuration are different and the obtained results are in general bad.

Canfora et al. 2007 introduced a technique to track the evolution of source code lines, identifying whether a CVS change is due to line modifications rather than to additions and deletions. The technique compares the sets of lines added and deleted in a change set, combining

the use of Information Retrieval techniques, in particular Vector Space Models, with the Levenshtein edit distance. Results obtained indicated that the proposed approach ensured both high precision and high recall.

Kim et al. 2008 proposed change classification to predict latent software bugs. Change classification uses a machine learning classifier to determine whether a new software change is more similar to prior buggy changes or clean changes. In this manner, change classification predicts the existence of bugs in software changes. Case study results show that the trained classifier can classify changes as buggy or clean, with a 78 percent accuracy and a 60 percent buggy change recall on average.

Hindle et al. 2009 proposed windowing the topic analysis to give a more nuanced view of the system evolution. By using a defined time-window of, for example, one month, they could track which topics come and go over time, and which ones recur. They also proposed visualizations of this model that allows them to explore the evolving stream of topics of development occurring over time. Windowed topic analysis offers advantages over topic analysis applied to a project lifetime because many topics are quite local.

CHAPTER 3

RELATED WORK

Chapter 3 gives an in-depth look at the research directly related to this research study. Sections 3.1 and 3.2 introduce the evolution tasks of bug localization and feature location using information retrieval, and also reviews research papers which are most related to our research. These papers use either LSI, LSI augmented with other information, and other IR modules for bug localization and feature location. Section 3.3 then introduces the process of source code retrieval, in which lexical information is retrieved from source code for IR modules to apply. Section 3.4 discusses the details of LSI. In addition, this section provides research papers in which LSI is used for different tasks in software evolution such as program comprehension, traceability recovery and software system categorization.

3.1 Bug localization

Bug localization is a software evolution task in which a developer uses information about a bug present in a software system to locate the source code elements, such as classes or methods that must be modified to correct the bug.

Poshyvanyk et al. 2007 introduce PROMESIR, a technique that combines LSI and scenario-based probabilistic ranking (SPR) for bug localization. The technique uses different scenarios to associate features (bugs) with source code elements that implement the features. The authors present case studies that use Mozilla and Eclipse to demonstrate that the combined technique outperforms LSI or SPR individually. The disadvantage of the technique is that it

depends on the skill of the developer to formulate appropriate queries.

Lukins et al. (2008, 2010) present an LDA based approach to bug localization. In five case studies on Mozilla, Eclipse, and Rhino they demonstrate show that the approach outperforms LSI as well as the accuracy and scalability of the approach. Further, the authors show that the approach is not sensitive to the size of the subject software system and that there is no relationship between the accuracy of the approach and the stability of the subject system. Finally, they demonstrate that coding links can be used to navigate from the first relevant method in the ranking to other methods modified to correct the bug.

3.2 Feature location

A feature in a program represents some functionality that is accessible by and visible to the developers, and, usually, it is captured by explicit requirements. **Feature location** is a process that identifies parts of the source code related to a specific feature.

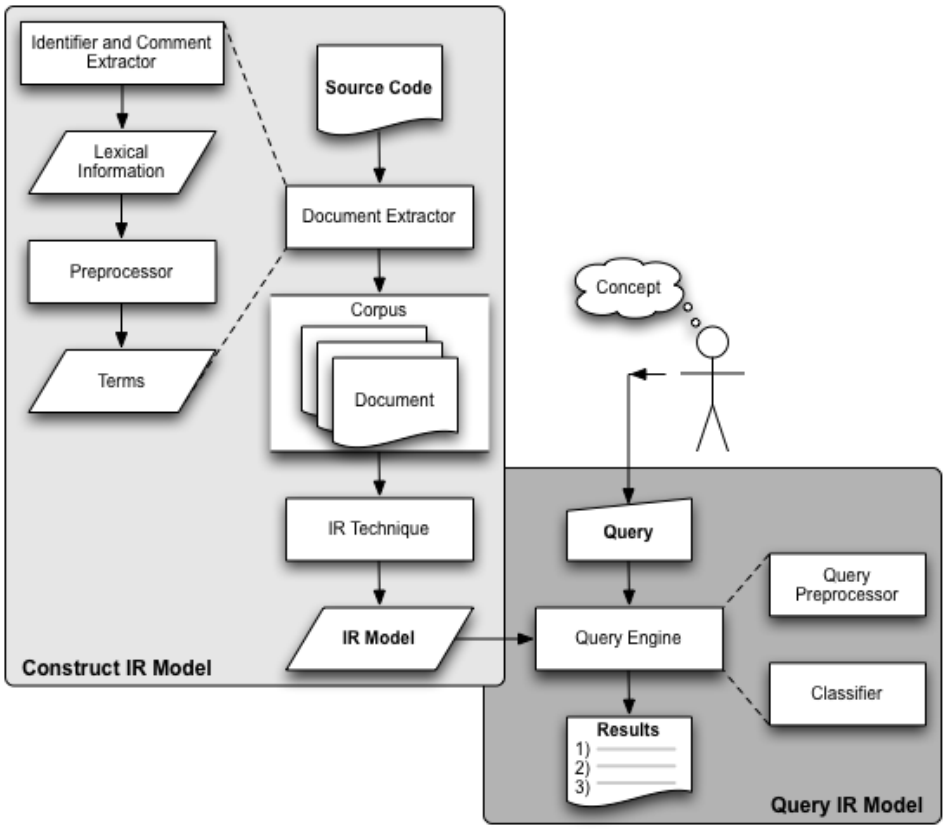
Liu et al. 2007 propose a semi-automated technique for feature location based on information from identifiers, comments, and an execution trace. In their case study of Eclipse, they show that the combination of LSI and a scenario execution trace outperforms LSI and SPR, and produces results similar to PROMESIR, for bug localization. Their jEdit case study shows that the new technique is less sensitive to poor user queries than LSI alone.

Shao et al. 2009 present a preliminary version of the LSICG technique and apply it to JavaHMO, Rhino, and jEdit. The focus of the paper and its case studies is feature location, though the authors present a small bug localization case study, which uses three bugs in Rhino. The details of the LSICG process differ from those presented in this paper, as do the queries used in the bug localization case study. In particular, the formula used in the previous work to

integrate the LSI score (S_{LSI}) and the call graph score (S_{CG}) is not the same (or as effective) as the current formula (see Chapter 4.2 for details of the current formula).

3.3 Source code retrieval

Source code retrieval is the application of an IR technique, such as LSI, to a source code model constructed from lexical information embedded in the source code, including terms from identifiers and comments. Each "document" is extracted from a source code element such as a class or a method, and the source code element used to partition the source code into documents determines the granularity of the results returned by the IR model in response to a user query. The results are a ranked list of documents, where the rank of each document corresponds to the similarity between the document and the user query. The more similar a document is to the user query, the higher its rank in the list. Figure 3.1 illustrates the process, which we describe in more detail in the following paragraphs.



During document extraction each identifier or comment is preprocessed, or normalized, to obtain one or more terms to be added to the document. Common preprocessing steps include (Marcus et al., 2004):

- Identifier separation – split identifiers (or comment words) that follow common coding style conventions into constituent terms (e.g., bug_localization, bugLocalization, and BugLocalization are each split into two terms)
- Case normalization – replace each lower case letter with the corresponding upper case letter (or vice-versa)
- Stop word elimination – remove common words such as English language articles (e.g., “an” or “the”) and programming language keywords

- Stemming – strip suffixes to reduce words to their stem, or root, forms (e.g., “jumper”, “jumping”, and “jumps” all become “jump”)

Together, the extracted, preprocessed documents form the corpora, from which an IR model is created. To query the IR model, a user formulates a query string and provides it to a query engine, which applies to the query string the same preprocessing steps previously applied to the corpora. A classifier, or similarity measure, is applied pairwise to the query and each document in the corpora. The resulting similarity scores are used to rank the documents in descending order. The user examines the results and possibly refines or rewrites the query string if the results are not satisfactory.

IR techniques have shortcomings that impact the accuracy of source code retrieval techniques. For example, many IR techniques can not represent polysemy (different meanings for the same word) or synonymy (different words with the same meaning). These deficiencies negatively affect bug localization, because bug reports often are written by end-users and source code typically has multiple authors, and different authors likely use different vocabularies (Poshyvanyk et al., 2007). Moreover, software developers often use abbreviations (e.g., “msg” for “message”), while end-users are not necessarily aware of such conventions. A common requirement of all source code retrieval techniques is that meaningful identifiers be used (Marcus and Maletic, 2001).

3.4 Latent semantic indexing (LSI)

In this section we provide introduction and details to the particular IR technique we used in our papers: Latent semantic indexing (LSI).

3.4.1 Introduction

LSI is the application of latent semantic analysis (LSA) to document indexing and retrieval (Deerwester et al., 1990; Grossman and Frieder, 2004). LSI is based on the vector space model, which is an algebraic model that represents documents as vectors of terms and relationships among terms and documents as a term-document co-occurrence matrix. Each row in the matrix is a vector that represents a term, and each column in the matrix represents a document. The value stored in each cell in the matrix represents the relationship between the term and the document. Commonly, the value stored in a cell is the number of times that the term occurs in the document, the term frequency. However, other value assignments are possible, including 0 or 1 (to indicate that the term does not or does appear in the document, respectively) and term frequency inverse document frequency (Salton and Buckley, 1988).

LSI uses singular value decomposition (SVD) to reduce the co-occurrence matrix. This reduction retains the semantic meaning of the corpora but shrinks the search space. The reduced co-occurrence matrix represents the LSI space (referred to as the IR model in Chapter 2.1). The classifier used to compute the similarity between two documents in the LSI space is the cosine of the angle between the vectors representing the documents, that is, the cosine distance. A small cosine distance indicates a high degree of similarity (Deerwester et al., 1990).

Singular Value Decomposition (SVD), proposed by Golub and Kahan 1965, is a decomposition technique which can find singular values for a matrix. When a matrix A times a vector X, it generates a new vector $AX = Y$. The vector X actually does two things on matrix A. It either does rotation in which the vector changes coordinates, or it does scaling in which the length of the vector changes. For the matrix A, the singular values of A determine maximum

stretching and minimum squeezing that a vector could influence on it. And these inherent, unique or "singular" to A that can be uncovered by SVD. A could be decomposed as:

$$A = USV^T \quad (M*N)$$

The definitions of the initials in the above equation are listed below:

U: a matrix whose columns are the eigenvectors of the $\mathbf{AA^T}$ matrix. (M*M)

S: a diagonal matrix whose diagonal elements are the singular values of **A** in decreasing order. (M*N)

V: a matrix whose columns are the eigenvectors of the $\mathbf{A^TA}$ matrix. (N*N)

$\mathbf{V^T}$: the transpose of **V**.

The singular values highlight which dimension(s) is/are affected the most by the matrix and the largest singular value has the greatest influence on the overall dimensionality.

In the case of LSI, SVD unveils the hidden or "latent" data structure, where terms, documents, and queries are embedded in the same low-dimensional and critical space.

3.4.2 LSI Application

Maletic et al. 2000 applied LSI for software maintenance and program comprehension. LSI is used as the basis to group software components, across files, to assist in program comprehension. Their results indicated that 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. Maletic et al. 2001 again applied LSI and file organization to support the comprehension tasks involved in the maintenance and reengineering of software systems. Results showed that

the semantic similarity of source code documents provides valuable information that can be used in the tasks of software maintenance and evolution. It also shows that concepts from the problem domain are often spread over multiple files, and files contain multiple concepts.

Marcus and Maletic 2003 applied LSI to automatically identify traceability links from system documentation to program source code. The method using LSI performed at least as well as Antoniol et al 2000a's methods using probabilistic and VSM IR methods combined with full parsing of the source code and morphological analysis of the documentation, but LSI performed with less computation. The method identified semantic similarities in source code but some lack of precision and limited automation was the price to pay for the low costs of the proposed method. The method failed to identify two functions with similar structure and functionality if comments did not exist and the identifier names were completely different.

Kawaguchi et al. 2003 presented an automatic software categorization algorithm to help finding similar software systems in software archive. Findings about LSI included that the software systems in the manual classification groups of editor, video conversion, and xterm showed very high similarity each other. On the other hand, systems in board game, compiler, and database did not show high similarity. This is because in board game, compiler, or database, there are little common concept which characterize overall systems. Editor, video conversion, and xterm contain a lot of characterizing system call names and variable names among systems. There are two systems in board game which show high similarity with editors. This is because it shares the same GUI framework.

DeLucia et al. 2004 proposed ADAMS (ADvanced Artifact Management System), a traceability recovery method and tool based on Latent Semantic Indexing (LSI) in the context of an artifact management system. Their case study of using the traceability recovery tool on

different types of software artifacts showed how better results can be achieved using a variable threshold and categorizing the artifacts in the repository in different subspaces.

DeLuci et al. 2005 used the same tool and suggested that the tool was a good support for the software engineer during the traceability recovery process. The IR based tool reduced the artifacts space; therefore, it allowed the software engineer to discover the traceability links across the analysis of a shorter list of links (lost links). Moreover, the tool helped the software engineer to check the consistency of the artifacts content; in particular, the tool highlighted likely inconsistencies in the usage of the domain terms in the traced artifacts (warning links), which was a useful information and helped the software engineer to guide the developers to produce better quality artifacts.

DeLucia et al. 2006 described an Eclipse plug-in, called COCONUT (COde COmprehension Nurturant Using Traceability) that used LSI to indicate the similarity between the source code under development and a set of high-level artifacts. The authors reported results of a controlled experiment performed to assess the usefulness of the proposed approach and tool, and results of a code inspection made after the experiment with the purpose of assessing the quality of code identifiers and comments. Results indicated that the plug-in significantly helped to improve the similarity between code and related requirements in presence of comments, while a practical (while not always statistically significant) improvement was also seen without considering comments. Code inspection over the artifacts produced during the experiment indicated that the plug-in helps to improve the quality of comments and of variable names.

DeLucia et al. 2006a gave a critical analysis of using feedbacks within an incremental traceability recovery process, and performed several case studies with the aim of discussing the strengths and limitations of using relevance feedbacks within traceability recovery processes.

Findings included that the use of feedbacks was not a silver bullet for the problem of recovering all correct links, as this still implied an unaffordable effort required to discard false positives. Moreover, when the results of IR methods were already good, feedbacks did not provide any improvement at all. However, despite these limitations, when feedbacks improved the retrieval performances, they could still be used within an incremental traceability recovery process.

DeLucia et al 2007 again used ADMS on traceability recovery: 1. They gave an assessment of LSI as a traceability recovery technique where they showed how using such a technique to recover all traceability links is not feasible in general, as the number of false positives grew up too rapidly when the similarity of artifact pairs decreased below an optimal threshold. They also showed how the optimal similarity threshold changes depending on the type of artifacts and projects and how such a threshold can be approximated case by case within an incremental traceability recovery process, where the similarity threshold is tuned by incrementally decreasing it; 2. They gave the definition and implementation of a tool that helps the software engineer to discover emerging traceability links during software evolution, as well as to monitor previously traced links; 3. They did an evaluation of the usefulness of the proposed approach and tool during the development of seventeen projects involving about 150 students; the results of the experience show that IR-based traceability recovery tools help software engineers to improve their recovery performances, although they are still far to support a complete semi-automatic recovery of all traceability links. Their findings also included 1. Experimental results should drive the design decisions in the implementation of a tool. 2. IR methods provide a useful support to the identification of traceability links, but are not able to identify all traceability links. 3. The incremental approach to traceability recovery was preferred to a one shot approach. 4. IR-based methods can help in the identification of quality problems in

the text description of software artifacts.

DeLucia et al. 2008 further used the ADAMS representation of software artifacts and indicated that it is possible to support a fine-grained traceability management (e.g., a single requirement can be traced onto single use cases). They combined the versioning management subsystem of ADAMS with LSI to support both traceability link versioning and evolution. By integrating the tool in the Eclipse-based client of ADAMS it was possible to use the traceability recovery functionality directly from a widely used IDE.

Lormans et al. 2005 and Lormans et al. 2006 presented a method for generating requirements coverage views. They investigated to what extent LSI can help recovering the information needed for automatically reconstructing traceability of requirements during the development process. Their findings included that if the documents to be analyzed are setup according to a well-designed traceability structure, LSI is capable of reconstructing it, and tracing requirements in test cases is easier than tracing requirements in design. Lormans et al. 2007 and Lormans et al. 2008 provided a methodology, MAREV, for automating the process of reconstructing traceability and generating requirements views. They defined a new two-dimensional vector filter strategy for selecting traceability links from an LSI similarity matrix. They also provided a tool suite, ReqAnalyst, for reconstructing traceability links including support for quantitative and qualitative assessment of the results. Their approach was applied in three case studies of which one was an industrial strength case in the consumer electronics domain. For each of the case studies, they offered an analysis of factors contributing to success and failure of reconstructing traceability links. Then they identified the most important open research issues pertaining to the adoption of LSI for link reconstruction purposes in industry. Lormans et al. 2008a used same approach to an ongoing project at LogicaCMG, and illustrated

how the software development process steers the reconstruction process and determines the meta-model used, how the quality of the reconstructed traceability matrix can vary per link type, and how the traceability matrices can be used to obtain requirements views.

Kawaguchi et al. 2006 proposed MUDABlue, a tool that automatically categorizes software systems. Results showed that MUDABlue has helped in determining categories that were not determined manually through the Open Source process. MUDABlue method does not only categorize software systems, but also automatically determines categories from the collection. MUDABlue method can categorize without any knowledge about target software systems. In addition, a MUDABlue interface was implemented that enables category-based browsing of a software repository, where a given software system can belong to multiple categories.

McCarey et al. 2006 described a RASCAL which enables a developer to effectively and conveniently make use of large scale libraries. RASCAL could recommend a set of task-relevant reusable components to a developer.

Gross et al. 2007 introduced a novel technique for automatically linking requirements to component specification documents through applying latent semantic analysis. LSA helps to identify few relevant components out of a large repository. The experiments performed are quite promising with that respect.

Kuhn et al. 2007 described an approach to retrieve the topics present in the source code vocabulary to support program comprehension. This approach used semantic clustering, a technique based on Latent Semantic Indexing and clustering to group source documents that use similar vocabulary. They called these groups semantic clusters and interpreted them as linguistic topics that reveal the intention of the code. The case studies give evidence that the approach

provided a useful first impression of an unfamiliar system, and that revealed valuable developer knowledge. The Distribution Map together with the labeling provided a good first impression of the software's domain. Semantic clustering captured topics regardless of class hierarchies, packages and other structures. One can, at a glance, saw whether the software covers just a few or many different topics, how these were distributed over the structure, and – due to the labeling – what they were about. Their experiments also showed that most linguistic topics related to application concepts or architectural components.

David 2008 suggest a hybrid approach based on Latent Semantic Indexing (LSI) and machine learning methods to recommend software development artifacts, that is predicting a sequence of configuration items that were committed together.

Jiang et al. 2008 proposed iLSI to automatically manage traceability link evolution and update the links in evolving software. iLSI allows for the fast and low-cost LSI computation for the update of traceability links by analyzing the changes to software artifacts and by reusing the result from the previous LSI computation before the changes. Their case study showed that iLSI maintains almost as good performance as the traditional LSI method, but saved more time compared with LSI.

Marcus et al. 2008 proposed a new measure for the cohesion of classes in OO software systems based on the analysis of the unstructured information embedded in the source code, such as comments and identifiers. They defined the conceptual cohesion of classes, which captures new and complementary dimensions of cohesion compared to a host of existing structural metrics. The measure captures different aspects of class cohesion compared to any of the existing cohesion measures. The combination of structural and conceptual cohesion metrics defines better models for the prediction of faults in classes than combinations of structural metrics alone.

Highly cohesive classes need to have a design that ensures a strong coupling among its methods and a coherent internal description.

CHAPTER 4

RESEARCH METHOD

Chapter 4 defines the Call Graph structure used to augment LSI. Once defined, LSI and CG are combined into a new automated static bug localization and feature location technique (LSICG). LSICG combines lexical information (terms from identifiers and comments) and structural information (call graphs) extracted from source code to rank each method and class according to their similarity to the description of the bug or feature under consideration. The implementation of LSI and LSICG are described as well as the measures used to examine the accuracy of LSICG compared to LSI alone.

4.1 Call graph

A call graph is a directed multi-graph in which the nodes represent the methods or the classes in a program and the edges represent the potential calls between those methods or classes. For example, given two methods M_0 and M_1, an edge (M_0, M_1) appears in the set of edges if there is a potential call to M_1 by M_0. A call graph extracted from source code is called a static call graph (Murphy et al., 1998), and to extract a static call graph, information about at least the following source code elements is required: method definitions (call targets) and method calls (call sites). Figure 4.1 illustrates a sample call graph with seven methods — A, B, C, D, E, F, and G — and six method calls — (A,F), (A,G), (B,A), (C,A), (D,A), and (E,A).

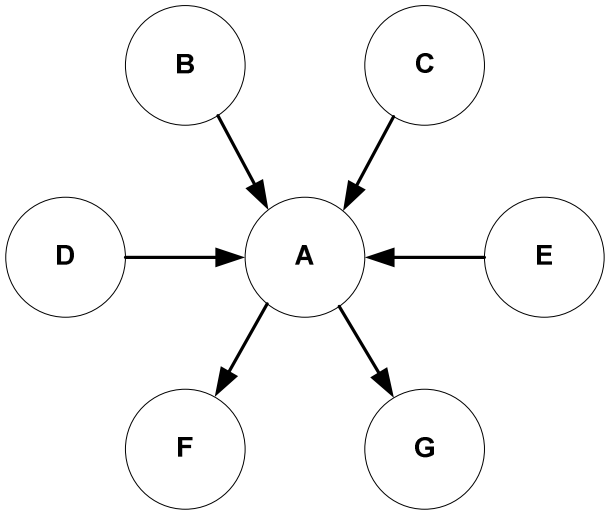


Figure 4.1 Sample call graph

In strictly first-order procedural languages, static call graph extraction is straightforward — at each call site the call target is directly evident from the source code. However, in Java, polymorphism and dynamic binding complicate static call graph extraction. Structural information, such as inheritance relationships between classes, and semantic information, such as type definitions and bindings, are required to determine the target of a call made through an object reference. Even with that structural and semantic information, it is often possible to determine only a set of potential call targets for a given call site (Grove et al., 1997).

4.2 LSICG bug localization and feature location technique

LSICG is defined by combining two processes: LSI-based source code retrieval (at the method level of granularity) and static call graph extraction. Implementation details of these processes are described in Section 4.2, but first, an overview of the LSICG process is provided. The LSICG process comprises six steps:

1. Request a query from the user, who formulates the query based on information extracted from a bug report.

2. Construct and query the LSI model to obtain a list, L_{LSI}, containing all methods in the program ranked in descending order by similarity score (as described in Section 3.3 and illustrated in Figure 1)

3. Define a minimum similarity score threshold and add methods with similarity scores greater than or equal to the threshold to a new list, $L_{THRESHOLD}$. Note that methods in $L_{THRESHOLD}$ retain their original ranks from L_{LSI}.

4. Extract a call graph from the source code of the program. For each method m_i in $L_{THRESHOLD}$, extract a list, L_{CG}, containing each method it calls directly. That is, L_{CG} contains each method m_t such that the edge (m_i,m_t) exists in the program call graph. To ease explication, a hash function H_{CG}, is defined where $H_{CG}(m_i)$ returns the L_{CG} for m_i.

5. For each method m_i in $L_{THRESHOLD}$, compute the combined score as follows:

   $$S_{COMBINED}(m_i) = (\lambda * S_{LSI}(m_i)) + ((1- \lambda) * (N_{CG}(m_i) / N_T) * S_{CG}(m_i))$$

   where:

   - λ — a weight in the range (0,1) that represents confidence in the ability of S_{LSI} or S_{CG} to rank m_i correctly (in terms of relevance to the bug of interest)
   - S_{LSI} — the rank of m_i in $L_{THRESHOLD}$
   - N_{CG} — the sum of $a_1, a_2, \ldots, a_z$ where z equals $|L_{THRESHOLD}|$, a_k equals the number of occurrences of m_i in $H_{CG}(m_k)$ and m_k is the k^{th} element of $L_{THRESHOLD}$

N_T the sum of b_1, b_2, …, b_z where z equals $|L_{THRESHOLD}|$, b_k equals $|H_{CG}(m_k)|$ and m_k is the k^{th} element of $L_{THRESHOLD}$

S_{CG} 1 if there exists a method m_k that is in $H_{CG}(m_i)$ and in $L_{THRESHOLD}$; 0 otherwise

The term $(N_{CG}(m_i) / N_T)$ is used to convert $S_{CG}(m_i)$ from a binary value to a value in the range [0,1], which is the range of S_{LSI} when the minimum threshold is 0. The resulting value is the density of method m_i in the call graphs of all methods in $L_{THRESHOLD}$. That is, the resulting value measures how often m_i appears as a one-edge-away node among all of the call graphs for the methods in $L_{THRESHOLD}$.

6. Create a new list, L_{LSICG}, containing the methods in $L_{THRESHOLD}$ ranked in descending order by $S_{COMBINED}$, and return it to the user.

The LSI-based source code retrieval at the class level of granularity is the same with it at the method level of granularity, except that each document is a class rather than a method. So L_{LSI} Contains all classes ranked in descending order by their similarity scores. Call graphs contains all calling information among classes. The LSICG score of a class is a combination of the class's LSI score and its call graph score.

4.2.1 One-Edge-Away rule

In Step 4 of the LSICG process, for each method m_i in $L_{THRESHOLD}$, we create list L_{CG} that contains each method called directly by m_i. That is, when constructing L_{CG} for m_i, we only add a method m_t to L_{CG} if the edge (m_i,m_t) exists in the program call graph. We call this the one-edge-away rule. The intuition behind this rule is that if a method is highly ranked in $L_{THRESHOLD}$, it is likely to be relevant to the query, and if the method directly calls other methods in $L_{THRESHOLD}$,

those methods may also be relevant to the query, even if they are not highly ranked. Further, the one-edge-away neighbors of the highly ranked method may be more likely to be relevant to the query than methods that are call transitively or not at all.

LSICG considers `call' edges (e.g., A calls B) but not `called by' edges (e.g., A is called by B). To consider the latter kind of edge, we would need to build a complete call graph for the subject system. However, a primary goal (and benefit) of LSICG is scalability, and we can build a one-edge-away call graph on a per-method, on-demand basis. In addition, using only 'call' edges simplifies LSICG, making it easy to implement as a plug-in to an IDE such as Eclipse or NetBeans.

Figure 4.2 illustrates a portion of a sample program call graph with 11 methods and 10 method calls. Step 4 of the LSICG process is executed as follows. Let A be the method under consideration and let $L_{THRESHOLD}$ contain methods A, B, C, D, E, F, and G. In this case, L_{CG} for A is extracted from the program call graph as follows (the nodes in L_{CG} for A are shown as unfilled circles in Figure 3):

- Add the one-edge-away methods — D, E, F, H, and K — to the candidate list.
- Remove methods not contained in $L_{THRESHOLD}$ — H and K — from the candidate list.
- The remaining methods — D, E, and F — are added to L_{CG}.

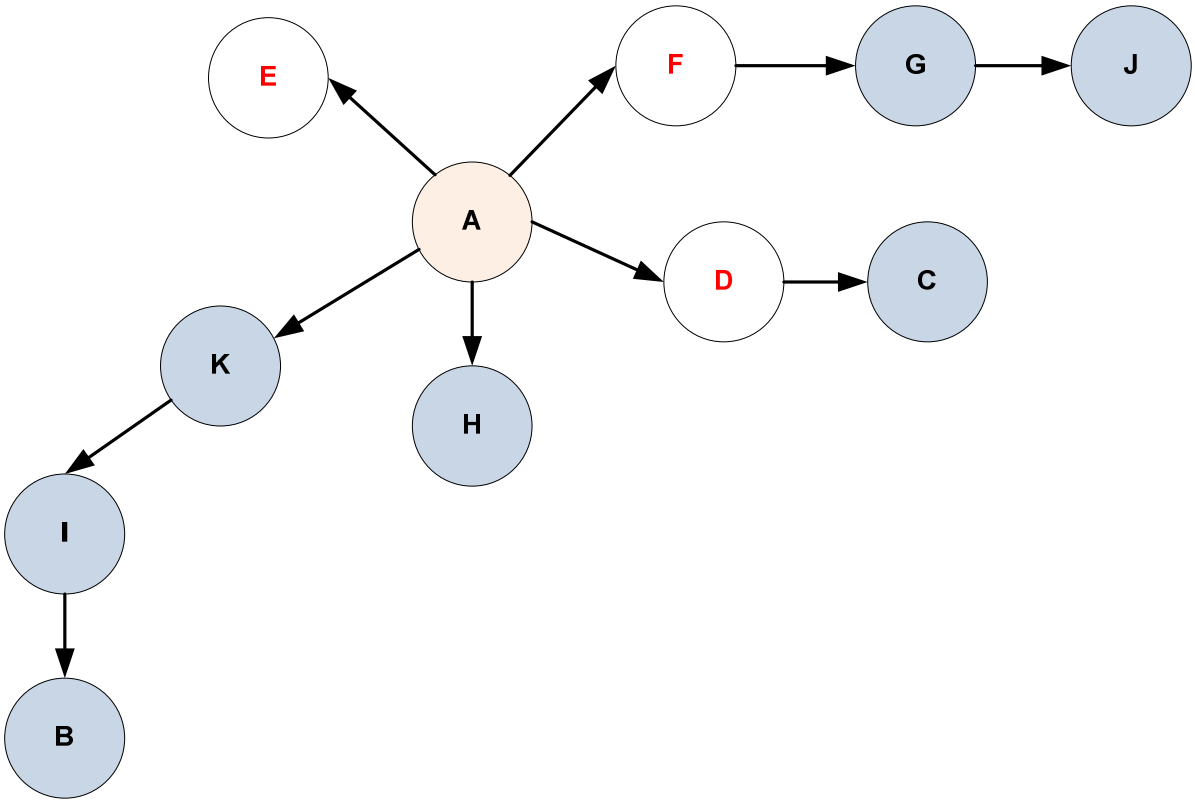


Figure 4.2 One-edge-away rule

The one-edge-away rule is also applied at class level granularity, i.e., for a class, we only consider its one-edge-away classes in its call graph.

4.2.2 LSI-Precedence rule

In Step 5 of the LSICG process, we use a weight, λ, to express confidence in the ability of S_{LSI} or S_{CG} to rank a method correctly. The S_{LSI} term must have precedence over the S_{CG} term when computing $S_{COMBINED}$. The intuition behind this rule is that we wish to use LSI to compute similarity between a method and the query, whereas we wish to use call graph information only to complement LSI. To implement this rule, we set the value of λ to be greater than 0.5. In addition, in the case that two $S_{COMBINED}$ scores are equal, we attempt to break the tie by selecting the score with the greater S_{LSI} term.

4.3 LSICG implementation

LSICG comprises LSI-based source code retrieval (at the method level of granularity) and static call graph extraction. Our implementation of LSI-based source code retrieval is an instantiation of the generic process described in Section 2.1 and illustrated in Figure 1, and is based on Eclipse and Matlab. Our static call graph extractor is also based on Eclipse.

4.3.1 LSI-based source code retrieval implementation

To instantiate the generic process, we use specific implementations of the Document Extractor, the IR Technique, and the Query Engine.

Document Extractor. We extract documents at the method level of granularity. That is, in our corpora, each document represents one method from the source code. From within each method definition, we extract all identifiers and comments. Next, we apply the four preprocessing steps defined in Section 3.3. In particular, we separate identifiers, convert all letters to lower case, remove stop words, and apply stemming (Porter, 1980). After preprocessing, we write terms to the document. The result of document extraction is a corpora, V, containing documents M_1, M_2, …, M_n, where *n* is the total number of methods in the source code.

We also extract documents at the class level of granularity. Each document represents one class from the source code in our corpora.

IR Technique. LSI requires a term-document co-occurrence matrix, which we construct using the term frequency model. That is, as described in Section 3.3, the value in each cell of the co-occurrence matrix is the number of times the term (represented by the row) occurs in the document (represented by the column). The resulting co-occurrence matrix, A, has *m* rows and *n*

columns, where *m* equals the total number of unique terms in the corpora V, and *n* is the number of documents in V. Using Matlab, we compute singular value decomposition (SVD) of the co-occurrence matrix A:

$$A = USV^T$$

By setting a *k* value, we retain the *k* largest singular values in matrices U, S, and V^T. Unlike our previous research in which we set *k* = 100, for these three cases we set up multiple *k*s, in order to obtain U_k, S_k, and V_k^T. In particular, we set up *k* equals to 16, 32, 64, 128, and 256, respectively. The reason we do this is that we want to use different dimensions in order to obtain best performance from LSI and compare it with our new algorithm.

Query Engine. To each user query, we apply the same four preprocessing steps that we apply to each document in the corpora. Next, we construct a term-document co-occurrence matrix using the term frequency model. The new co-occurrence matrix, Q, has *m* rows and one column. We then map the query into the concept space, obtaining a new vector, q:

$$q = Q^T U_k S_k$$

Finally, we use our classifier, cosine distance, to compute the similarity between q and each column of V_k^T, which represents method M_i:

$$S_{cos}(q,M_i) = (q \bullet M_i) / (||q|| \, ||M_i||)$$

The result is a list containing the methods ranked in descending order by S_{cos}.

4.3.2 Static call graph extractor implementation

The static call graph extractor is a reverse engineering system (Xu and Song, 2006) that is based on the Eclipse JDT. Each vertex is a method, and the adjacency list for each vertex stores the

called methods. Because the JDT is an IDE, not an advanced static analysis tool, it does not use advanced call graph construction algorithms such as RTA or 0-CFA (Grove et al., 1997). Such algorithms have the potential to improve the accuracy of LSICG, but they are computationally expensive, and they require access to the full source code for a program (including all standard or third-party libraries).

4.3.3 LSICG Implementation

The implementation of LSICG combines the LSI-based source code retrieval implementation from Section 4.3.1 and the static call graph extractor from Section 4.3.2. The current implementation constructs $L_{THRESHOLD}$ (see Section 4.2) using a minimum similarity score threshold of 0. In addition, λ is set to 0.5.

CHAPTER 5

RESEARCH STUDY

The purpose of the research study is to examine whether LSICG, which combines lexical and structural information, is more accurate than LSI alone in concept location (feature location and bug localization). The null and alternative hypotheses for the research study are:

- **H_0:** LSICG is NOT more accurate than LSI for concept location using the same corpora and the same queries/targets.
- **H_1:** LSICG is more accurate than LSI on concept location using the same corpora and the same queries/targets.

We evaluate LSICG against LSI on three Java projects: JavaHMO, Rhino, and jEdit. The concept localization targets for JavaHMO are features previously identified by Shao et al. (2009). For concept location in Rhino, we use bugs using queries formulated by Lukins et al. (2008). The concept localization targets for jEdit are three features identified by Liu et al (2007) as well as six identified bugs.

For clarity we developed our test case procedures using the goal question metric (GQM) approach (Basili and Weiss, 1984) as follows:

Goal: To examine whether LSICG is more accurate than LSI for concept location over the corpora used in previous studies (Shao et al. 2009; Lukins et al. 2008; Lukins et al. 2010; Liu et al. 2007).

Question: How accurate are LSICG and LSI for concept location over the corpora of test data?

Metric: Rank of the highest-ranking method (or class) actually identified to add the desired feature or to correct the bug under consideration.

5.1 Projects Introduction

Table 5.1 lists general information of three Java projects which we used in our case study, including project name, project version, number of methods and classes, and number of unique terms extracted.

Table 5.1 Three projects information

Project Name	Version	No. of Methods	No. of Classes	No. of Unique Terms
JavaHMO	2.4	1,787	246	2,134
Rhino	1.5R5	2,428	201	2,543
jEdit	4.2	5,534	776	3,172

Java Home Media Option (JavaHMO) is a Java media server implemented for the Tivo® environment. JavaHMO allows users to display/play different media types such as mp3, FlashPix, JPEG, TIFF and others. In this study we are using the final version of JavaHMO, JavaHMO 2.4. JavaHMO has been used in other concept location studies including a concept location and comprehension study (Shepherd et al. 2007) and a context extraction and categorization study (Hill et al. 2009). We chose JavaHMO 2.4 specifically because it is used in these two studies. JavaHMO version 2.4 contains 1,787 methods, 246 classes and 2,134 unique terms.

Rhino is an open-source implementation of JavaScript in Java and is the subject software system we use in the case study. In particular, we use Rhino version 1.5 release 5 (1.5R5). Rhino has been used in other concept location studies, including a requirements tracing study (Eaddy et al., 2008) and two bug localization studies (Lukins et al., 2008; Lukins et al., 2010). We chose Rhino 1.5R5 specifically because it is used in the two latter studies. Rhino 1.5R5 comprises 2,428 methods, 201 classes and contains 2,543 unique terms.

jEdit is a text editor written for programmers and developers. It is licensed under the GNU GPL. jEdit has been used in previous case studies (Liu et al. 2007, Kuhn et al. 2007, DeLucia et al. 2008, Kim et al. 2008, Gay et al. 2009). We selected jEdit version 4.2 because it is the subject of other feature location studies Liu et al. 2007. jEdit version 4.2 includes 5,534 methods and 776 classes written in Java as well as 3,172 unique terms.

5.2 Research Study Design

In the research study, we examine the accuracy of LSICG and compare it to the accuracy of LSI alone. We test feature location on JavaHMO, bug localization on Rhino, and both bug localization and feature location on jEdit. For each bug and feature it is necessary to determine the relevant methods, that is, the methods actually modified to correct the bug or implement the feature. To determine the relevant methods for each feature in JavaHMO, we inspected the JavaHMO source code and found candidate methods and classes for a feature. After discussion we reached an agreement on whether a candidate method or class is relevant to a feature. We marked the methods which are relevant to a feature as our targets. To determine the relevant methods and classes for each bug in Rhino, we inspected the associated software patch posted in the Rhino Bugzilla repository. We compared the lines of code in each patch to the Rhino source code and noted the name of each modified method and class. We used relevant methods and

classes identified for jEdit features in Liu et al. 2007, and we inspected the associated software patch posted in the jEdit repository to identify relevant methods and classes for jEdit bugs.

To compare LSICG and LSI, for each bug and feature we compare the rank of the highest-ranking relevant method. This rank indicates the number of methods the developer must inspect before reaching a method actually requiring modification (assuming that the developer begins and the top-ranked method and proceeds down the list). Previous studies (Poshyvanyk et al., 2007; Lukins et al. 2008; Lukins et al. 2010) use this same measure of effort to compare bug localization and feature location techniques. The assumption is that from the first relevant method, the developer can locate other related elements through their coding links with that method (Lukins et al., 2010).

We use statistical testing to demonstrate that the results are unlikely to be obtained by chance. In particular, we use Wilcoxon's signed-rank test, a non-parametric paired samples test, to test for statistical significance. We also use the paired-samples t test as an alternative to test for statistical significance, especially when the size of samples is large, i.e., when the number of samples is greater than 25. The goal of this test in the context of our study is to confirm that any improvement in accuracy obtained using LSICG is statistically significant when compared to LSI, the baseline technique.

In our case study, we run three experiments on the three projects. In each experiment we run LSI and LSICG separately on the three projects.

- In the first experiment, we use a subset of bugs and features identified from the three projects. The experiment is run at method level, i.e., in our corpora, each document represents one method from the source code.

- In the second experiment, we enlarge the scale and use all bugs and features identified for the three projects. Multiple queries are used for each bug in Rhino and jEdit, as well as each feature in jEdit. This experiment is also run at method level.
- In the third experiment, we use all bugs and features from the three projects, with one fixed query for each bug and feature. The experiment is run at class level, which means in our corpora, each document represents one class from the source code.

5.3 Subset of bugs/features

In the first experiment, we run a subset of bugs and features for the three projects. Particularly, we use 12 features out of 25 features from JavaHMO for feature location, 21 bugs out of 35 bugs from Rhino for bug localization, and 3 features from jEdit for feature location, with 4 different queries for each feature.

5.3.1 JavaHMO

From the features list which includes feature names and descriptions of JavaHMO on its official website, we randomly selected twelve features in JavaHMO to test the performance of LSI and LSICG. Table 5.2 gives the features examined in this study.

Table 5.2 Features in JavaHMO

Feature Number	Feature Description
1.	Rotate images
2.	View MP3 file tag information.
3.	Play MP3 streaming stations on the internet
4.	Automatically download Shoutcast playlists of your favorite streaming stations.
5.	Use the streaming proxy server to significantly improve on the inadequate support TiVo provides for online streaming stations.

6. Play your MP3 files and streaming stations using both .m3u and .pls playlist formats.

7. View live local weather conditions including current conditions, 5-day forecasts and radar images.

8. Automatically download and view any image on the internet.

9. iTunes playlists integration.

10. Audio Jukebox.

11. Organize images files based on their date information.

12. Play interactive games such as TicTacToe.

5.3.2 Rhino

We randomly selected 21 bugs in Rhino 1.5R5 from those used in two previous bug localization studies by Lukins et al. (2008; 2010). Moreover, we use the same queries used in the first of these studies (Lukins et al., 2008). Table 5.3 lists information about the 21 bugs, including the bug numbers, bug titles, and terms extracted from the bug description. For each bug, the words in bold typeface constitute the query which was previously constructed by Lukins et al. 2008.

Table 5.3 Rhino 1.5R5 bugs analyzed (LSI query words in bold)

Bug no.	Bug title [query words added from bug description]
256836	**Dynamic** scope and nested **functions**
274996	Exceptions with multiple **interpreters** on **stack** may lead to **ArrayIndex**OutOf**BoundsException** [**java wrapped**]
256865	Compatibility with **gcj**: changing **ByteCode**.<**constants**> to be int
257423	**Optimizer** regression: **this**.**name** += expression generates wrong **code**
238699	**Context.compileFunction** throws **InstantiationException**
256621	**throw statement**: **eol** should not be allowed
249471	**String index out** of range **exception [parse bound float global js native char]**
239068	Scope of **constructor functions** is not **initialized**
238823	**Context.compileFunction** throws **Null**Pointer **exception**
258958	**Lookup** of excluded **objects** in **ScriptableOutputStream** doesn't traverse

	prototypes/**parents**
58118	ECMA Compliance: **daylight savings time** wrong prior to **year** 1 [**day offset timezone**]
256389	Proper **CompilerEnvirons**.is**Xml**Available()
258207	**Exception name** should be **DontDelete [delete catch ecma obj object script]**
252122	**Double** expansion of **error message**
262447	NullPointerException in **ScriptableObject**.**getPropertyIds**
258419	**copy paste** bug in org.mozilla.javascript.**regexp**.**NativeRegExp [RE data back stack state track]**
266418	Can not **serialize reg**ular **exp**ressions **[regexp RE compile char set]**
263978	cannot **run** xalan example with Rhino 1.5 release 5 **[line number negative execute error]**
254915	Broken "**this**" for **name**() calls (CVS tip regression) **[object with**]
255549	**JVM**-dependent resolution of **ambiguity** when **calling Java methods [argument constructor overload]**
253323	Assignment to variable '**decompiler**' has no effect in **Parser** [**parse**]

5.3.3 jEdit

We selected the three features studied by Liu et al. 2007 for this case study. The features were extracted based on two change requests for jEdit as reported in Liu et al. 2007:

- Add a "Search and mark all" menu item in the "Search" menu, which will locate all matches to a search phrase and add markers to all of the lines.
- Currently jEdit shows a red dot at the end of every line. Newline is the only whitespace symbol that jEdit shows. *Add a menu item "Show/Hide whitespace" under menu "View"* to allow the user to choose whether all whitespace symbols (newlines, blanks, and tabs) will be shown. At this stage you do not have to worry about editing of the text with whitespace showing.

To accommodate the two change requests, three features were added to jEdit. The descriptions of the three features are given in Table 5.4.

Table 5.4 Three features in jEdit

Feature Name	Description
Search	searching for the occurrence of the provided search phrase.
Add marker	adding a marker to the selected line in the text.
Show white space	showing whitespaces as a symbol in the text.

The features “Search” and “Add marker” were extracted from change request 1, and the feature “Show whitespace” was extracted from change request 2.

5.4 Subset bugs/features Results

The research study compares the performance of LSI and LSICG for two different concept location tasks: feature localization and bug localization. We compare the two techniques across three open-source Java projects (JavaHMO, Rhino, and jEdit) that have been studied previously in the literature. In particular, we select a subset of bugs and features identified from the three projects, and compare the performance of LSI and LSICG at method level for the bugs and features. Where possible, we have used the same concept (bug report or feature request) as used in the literature. The following sections give the results of the study for the subset of bugs and features from the three projects followed by an analysis of those results.

5.4.1 JavaHMO

The concept location task for JavaHMO involved locating the targeted method for each of 12 requested features. Table 5.5 below gives the features and the corresponding target method in bold beneath each feature.

Table 5.5 Features and identified targeted method in JavaHMO

Feature Number	Feature Description Targeted method
1.	Rotate images **org.lnicholls.JavaHMO.media.ImageManipulator.rotate**
2.	View MP3 file tag information. **org.lnicholls.JavaHMO.plugins.organizer.OrganizerContainer.findTas**
3.	Play MP3 streaming stations on the internet **org.lnicholls.JavaHMO.media.Mp3Proxy.stream**
4.	Automatically download Shoutcast playlists of your favorite streaming stations. **org.lnicholls.JavaHMO.plugins.shoutcast.ShoutcastStationContainer.getPlaylists()**
5.	Use the streaming proxy server to significantly improve on the inadequate support TiVo provides for online streaming stations. **org.lnicholls.JavaHMO.server.ServerConfiguration.getUseStreamingProxy()**
6.	Play your MP3 files and streaming stations using both .m3u and .pls playlist formats. **org.lnicholls.JavaHMO.model.FileSystemContainer.getItems(TiVoQueryContainerRequest)**
7.	View live local weather conditions including current conditions, 5-day forecasts and radar images. **org.lnicholls.JavaHMO.plugins.weather.WeatherData.WeatherData()**
8.	Automatically download and view any image on the internet. **org.lnicholls.JavaHMO.media.InternetImageItem.getImage(TiVoImageDocumentRequest, BufferedImage)**
9.	iTunes playlists integration. **org.lnicholls.JavaHMO.plugins.iTunes.iTunesContainer.onPlaylist(Playlist)**
10.	Audio Jukebox. **org.lnicholls.JavaHMO.plugins.jukebox.JukeboxItem.gatherItems(TiVoQueryContainerRequest)**
11.	Organize images files based on their date information. **org.lnicholls.JavaHMO.plugins.imageOrganizer.OrganizerContainer.categorize(String, ImageProxy)**
12.	Play interactive games such as TicTacToe. **org.lnicholls.JavaHMO.plugins.games.GamesContainer.start()**

For a given feature, we first run LSI in 16, 32, 64, 128, 256 dimensions respectively, then we run LSICG in the same dimensions. LSICG gave the best average performance for the 128 dimensions case. For this reason, the discussion and analysis below is restricted to the LSICG in 128 dimension case. Table 5.6 lists all results for JavaHMO.

Table 5.6 Rankings for JavaHMO (subset)

Feature Number	LSI16	LSI32	LSI64	LSI128	LSI256	LSICG128
1	683	129	353	1219 *(-0.043)*	1649	Not in threshold
2	1404	1159	1647	355	42	352
3	1108	510	1	180	180	117
4	1194	1543	1203	1620 *(-0.117)*	163	Not in threshold
5	1462	710	421	648	971	611
6	425	864	499	738	394	434
7	1491	1251	1212	895 *(-0.002)*	373	Not in threshold
8	288	481	505	533	1387	269
9	760	1148	778	598	1471	241
10	642	81	660	457	1322	411
11	258	434	515	614	621	36
12	242	1306	1122	1613 *(-0.115)*	1054	Not in threshold

For four of the features in JavaHMO the LSI score at 128 dimensions, shown above in italics, fell below our initial threshold of 0.0. For those features which have indicated their rank but we did not computer an LSICG score. Those features are: No. 1, No. 4, No. 7 and No. 12. The best performance technique for each is LSI32, LSI256, LSI256, LSI16, respectively.

Figure 5.1 gives the average ranking for each dimension. LSICG128 provides the best average ranking (309). The second best average ranking is 516 provided by LSI 128. The LSICG average ranking shows a 207 position improvement, or 40.1%, over LSI128 alone.

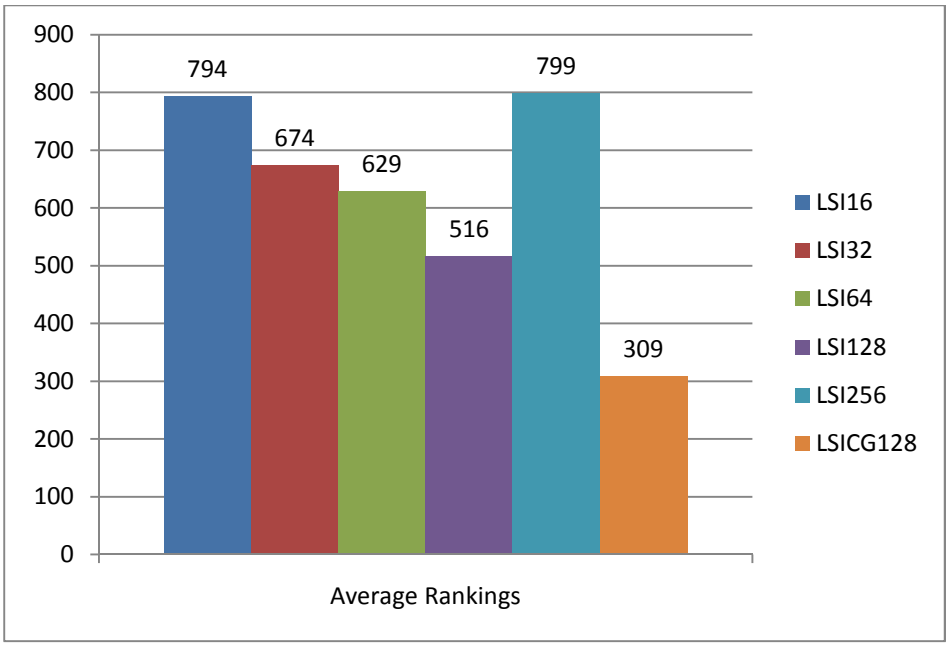


Figure 5.1 Average rankings for Java HMO (subset)

To accommodate the four features that fell below our initial threshold of 0.0, we reran the study with a threshold of -0.12. Table 5.7 gives the rankings with new threshold.

Table 5.7 Rankings for JavaHMO (subset and adjusted threshold)

Feature Number	LSI16	LSI32	LSI64	LSI128	LSI256	LSICG128
1	683	129	353	1219	1649	1237
2	1404	1159	1647	355	42	352
3	1108	510	1	180	180	117
4	1194	1543	1203	1620	163	1492
5	1462	710	421	648	971	611
6	425	864	499	738	394	434
7	1491	1251	1212	895	373	918
8	288	481	505	533	1387	269
9	760	1148	778	598	1471	241
10	642	81	660	457	1322	411
11	258	434	515	614	621	36
12	242	1306	1122	1613	1054	1610

Likewise, we compute the average ranking for all of the treatments with the new threshold. These are shown in Figure 5.2.

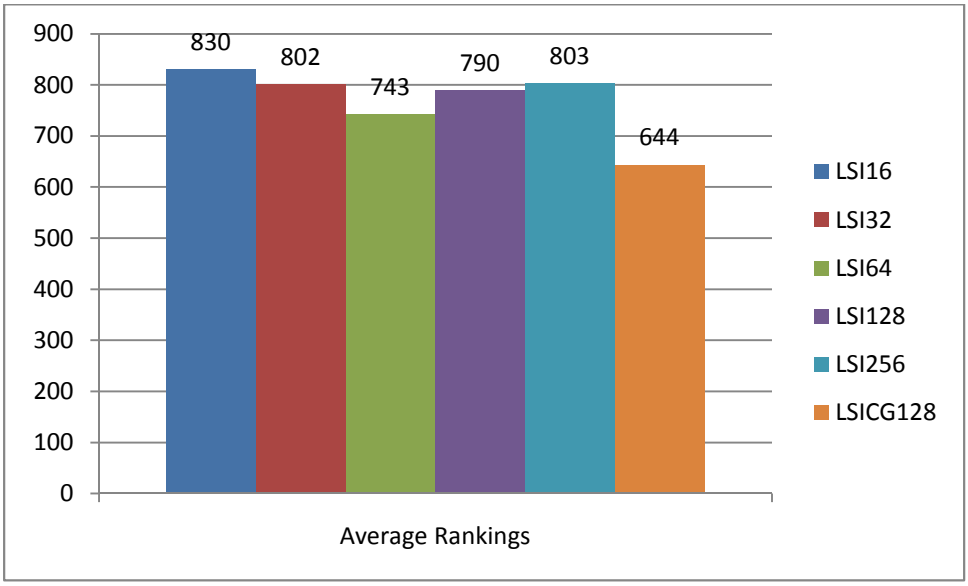


Figure 5.2 Average rankings for Java HMO (subset and adjusted threshold)

LSICG maintains the best average ranking, 644, over the second place performer, LSI64 at an average ranking of 743. LSICG provided an average ranking 99 positions higher than LSI64 for average ranking improvement of 13.32%.

To compare LSICG and LSI at the same dimension, we use Wilcoxon's signed-rank test to determine whether the improvement in accuracy of LSICG128 over LSI128 is statistically significant. We test the research study hypothesis and the result of Wilcoxon's test is W=7, while the critical value W*= 7, $p < 0.01$. W<=W*, so we reject the null hypothesis and accept the alternative hypothesis. There is significant evidence to indicate that LSICG128 is more accurate than LSI128.

5.4.2 Rhino

For Rhino, the concept location task involved isolating the correct method based on a given bug description. Table 5.8 below gives the bug number, the bug description with the query string

shown in bold followed by targeted class and method in italics for each bug. For the 21 bugs, we first run LSI and LSICG individually at the dimension of 100 with λ set to 0.5 and threshold set to 0.0.

Table 5.8 Bug number, query words and target methods for Rhino (subset)

Bug Number	Bug title [query words added from bug description] Targeted (*Class, Method*)
256836	**Dynamic** scope and nested **functions** *(Context, hasCompileFunctionsWithDynamicScope)*
274996	Exceptions with multiple **interpreters** on **stack** may lead to **ArrayIndex**OutOf**BoundsException** [**java wrapped**] *(Interpreter, interpret)*
256865	Compatibility with **gcj**: changing **ByteCode**.<**constants**> to be int *(ClassFileWriter, addInvoke)*
257423	**Optimizer** regression: **this**.**name** += expression generates wrong **code** *(Codegen, visitSetProp)*
238699	**Context.compileFunction** throws **InstantiationException** *(Codegen, compile)*
256621	**throw statement**: **eol** should not be allowed *(Parser, statementHelper)*
249471	**String index out** of range **exception [parse bound float global js native char]** *(NativeGlobal, encode)*
239068	Scope of **constructor functions** is not **initialized** *(IdScriptable, addAsPrototype)*
238823	**Context.compileFunction** throws **Null**Pointer **exception** *(Context, compile)*
258958	**Lookup** of excluded **objects** in **ScriptableOutputStream** doesn't traverse **prototypes**/**parents** *(ScriptableOutputStream, lookupQualifiedName)*
58118	ECMA Compliance: **daylight savings time** wrong prior to **year** 1 [**day offset timezone**] *(NativeDate, date_format)*
256389	Proper **CompilerEnvirons**.is**Xml**Available() *(CompilerEnvirons, isGeneratingSource)*
258207	**Exception name** should be **DontDelete [delete catch ecma obj object script]** *(ScriptRuntime, newCatchScope)*
252122	**Double** expansion of **error message** *(ScriptRuntime, undefWriteError)*
262447	NullPointerException in **ScriptableObject**.**getPropertyIds** *(ScriptableObject, getPropertyIds)*
258419	**copy paste** bug in org.mozilla.javascript.**regexp**.**NativeRegExp [RE data back stack state track]** *(NativeRegExp, matchRegExp)*
266418	Can not **serialize reg**ular **exp**ressions **[regexp RE compile char set]**

	(NativeRegExp, emitREBytecode)
263978	cannot **run** xalan example with Rhino 1.5 release 5 **[line number negative execute error]** *(Context, compileString)*
254915	Broken "**this**" for **name**() calls (CVS tip regression) **[object with**] *(ScriptRuntime, getBase)*
255549	**JVM**-dependent resolution of **ambiguity** when **calling Java methods [argument constructor overload]** *(NativeJavaMethod, findFunction)*
253323	Assignment to variable '**decompiler**' has no effect in **Parser** [**parse**] *(Parser, parse)*

Table 5.9 below lists the results for LSICG and LSI the 21 Rhino 1.5R5 bugs. The first column lists the bug number and the second and third columns list the rank of the highest-ranking method actually modified to correct the bug under consideration for LSICG and LSI, respectively.

Wilcoxon's signed-rank test is used to determine whether the improvement in accuracy obtained using LSICG is statistically significant. In particular, the research study hypotheses in Section 4.3.3 are tested. The results of Wilcoxon's test (W = 204, p = 0.001) with alpha = .05 signify that the median difference between the results for LSICG and the results for LSI is significantly greater than zero. Thus, the null hypothesis is rejected and the alternative hypothesis is accepted.

Table 5.9 Comparison of LSICG to LSI (subset, 100 dimensions)

Bug no.	LSICG rank	LSI rank
256836	27	209
274996	34	120
256865	15	29
257423	787	1,626
238699	467	585
256621	652	1,521
249471	87	8

239068	638	710
238823	154	99
258958	1,143	1,418
58118	629	736
256389	168	180
258207	321	351
252122	757	2,128
262447	1,168	1,715
258419	497	466
266418	618	782
263978	996	1,101
254915	178	65
255549	723	867
253323	287	473

The results in Table 5.9 show that LSICG is more accurate than LSI alone for 17 of the 21 bugs. Note that when the LSI rank is above 1,000, the LSICG rank is better for all 6 bugs and substantially better for 4 bugs. For example, the LSI rank for bug 252122 is 2,128, while the LSICG rank for that bug is 757. When the LSI rank is between 200 and 1,000, the LSICG rank is better for 7 of 9 bugs and substantially better for 2 bugs. For example, the LSI rank for bug 256836 is 209, while the LSICG rank for that bug is 27. When the LSI rank is less than 200, the LSICG rank is better for 3 of 6 bugs and substantially better for 2 bugs. For example, the LSI rank for bug 274996 is 120, while the LSICG rank for that bug is 34. However, for 2 bugs the LSI rank is substantially better than the LSICG rank. For example, the LSI rank for bug 249471 is 8, while the LSICG rank for that bug is 87. If many methods have SLSI values greater than the threshold value and also have a high fan-in metric value, LSICG may be less accurate than LSI. However, this study supports the efficacy of LSICG for a data set of 21 bugs.

Figure 6 illustrates the overall results of the comparison between LSICG and LSI. To categorize the results, we define three terms:

- Higher Ranking: For a given bug, the LSICG rank is at least 50 positions better than the LSI rank.
- Similar Ranking: For a given bug, the LSICG rank is within 50 positions (higher or lower) of the LSI rank.
- Lower Ranking: For a given bug, the LSICG rank is at least 50 positions worse than the LSI rank.

The results are promising. In particular, note that LSICG is more accurate than LSI for 67% (14) of the 21 bugs. Further, LSICG performs similarly to LSI for 19% (4) of the 21 bugs. However, LSICG is less accurate than LSI for 14% (3) of the 21 bugs.

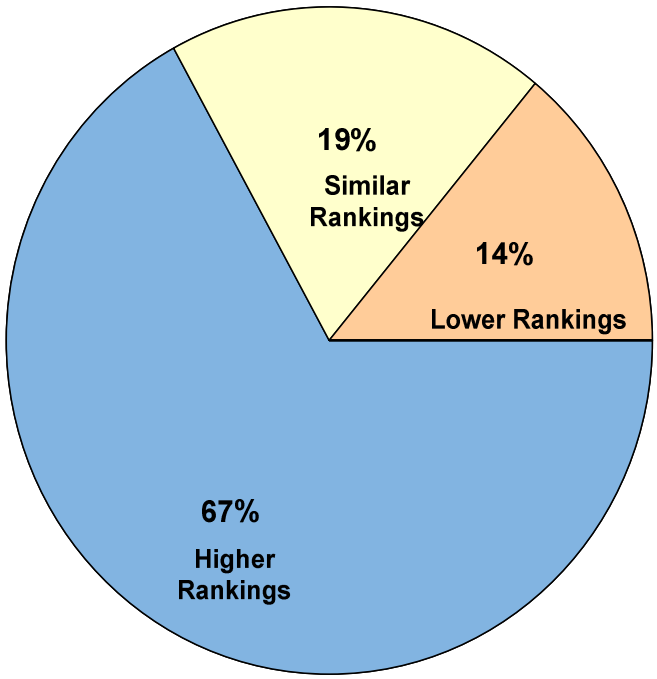


Figure 5.3 Results comparing the accuracy of LSICG to that of LSI alone

After obtaining the results from LSI and LSICG on single dimension 100, for a given bug query, we test their performance at multiple dimensions. In particular, we run LSI in 16, 32, 64, 128, 256 dimensions respectively, then we run LSICG. As the same in JavaHMO, the discussion and analysis for Rhino uses LSICG in 128 dimensions compared to LSI at 128 dimensions. Table 5.10 lists the rankings of LSI at each dimension as well as the rankings of LSICG at the dimension of 128 (LSICG128).

Table 5.10 Rankings for Rhino (subset)

Bug Number	LSI16	LSI32	LSI64	LSI128	LSI256	LSICG128
256836	999	832	653	339	641	366
274996	1303	1696	602	163	165	3
256865	62	288	299	693	1017	197
257423	589	1113	634	2059 *(-0.105)*	1313	Not in threshold
238699	1862	1331	996	1556 *(-0.036)*	1528	Not in threshold
256621	343	1659	1248	1099	2084	103
249471	1267	64	988	125	473	117
239068	238	617	1361	1082	967	365
238823	790	174	220	163	790	99
258958	721	1226	503	816	1560	720
58118	699	259	1656	1208 *(-0.0002)*	1408	Not in threshold
256389	347	542	814	327	283	964
258207	597	694	2062	1154	474	1268
252122	1525	1031	2097	1900 *(-0.0723)*	2233	Not in threshold
262447	890	1780	1516	2301 *(-0.148)*	2158	Not in threshold
258419	2218	1632	169	101	673	93
266418	180	406	920	1443 *(-0.020)*	1469	Not in threshold
263978	831	1367	1577	1260 *(-0.004)*	1092	Not in threshold
254915	289	80	1168	8	125	8
255549	536	278	1090	1131	1653	749
253323	1718	226	958	711	1811	465

As was the case for JavaHMO, seven bug queries returned an LSI score below 0.0 indicated above in italics. From Table 10, LSICG obtains higher rankings than LSI for most of the bug queries. Note that when LSI rankings are quite low, such as rankings below 1,000, LSICG gives much better rankings than LSI. When LSI rankings are fairly low, such as between 200 and 1000, LSICG gives similar rankings as LSI. When LSI rankings are near the top, such as in the first 200, LSI may have better rankings than LSICG. Bug 249471 is such an example. However, when LSI rankings are near the top, LSICG rankings, as expected, are also near the top. When LSI rankings are poor, LSICG gives much higher rankings saving developers time in locating the bugs.

Figure 5.4 gives the average ranking for each dimension of LSI and for LSICG. LSICG has the best average ranking 395. The second best average ranking is LSI128 at 566. LSICG achieved an improvement of 171 places rankings for a 30.2% improvement.

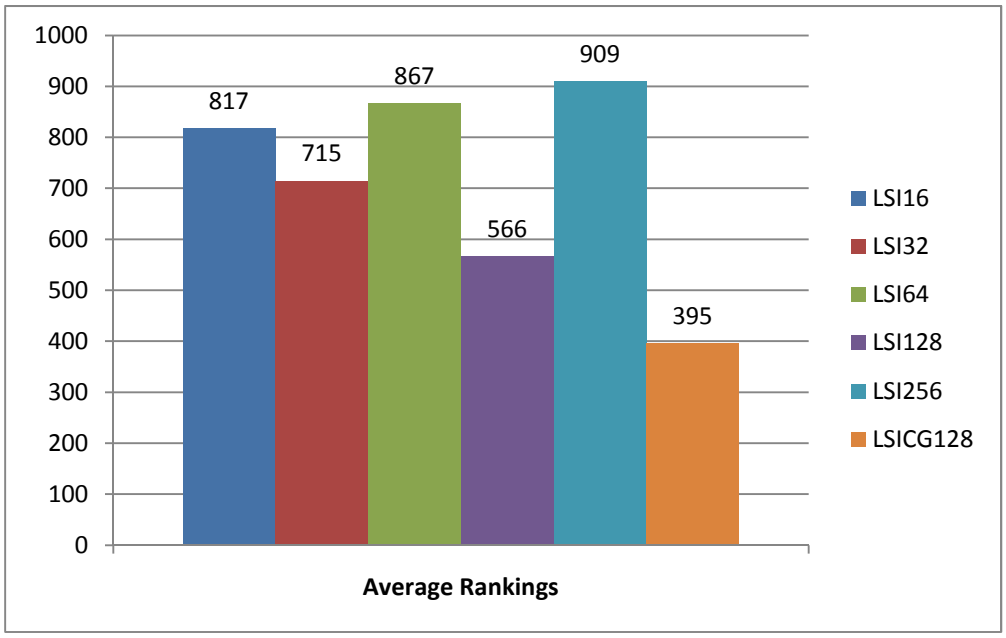


Figure 5.4 Average rankings from each technique for Rhino

For Rhino, as with the JavaHMO, we reran the study with a lower threshold to accommodate those bug queries that had an LSI score below 0.0. These bugs are numbers 257423, 238699, 58118, 252122, 262447, 266418, and 263978. In looking at the data, we adjusted our threshold value to -0.15 for these bugs. Table 5.11 gives the ranking results.

Table 5.11 Rankings for Rhino (subset and adjusted threshold)

Bug Number	LSI16	LSI32	LSI64	LSI128	LSI256	LSICG128
256836	999	832	653	339	641	366
274996	1303	1696	602	163	165	3
256865	62	288	299	693	1017	197
257423	589	1113	634	2059	1313	2011
238699	1862	1331	996	1556	1528	1431
256621	343	1659	1248	1099	2084	103
249471	1267	64	988	125	473	117
239068	238	617	1361	1082	967	365
238823	790	174	220	163	790	99
258958	721	1226	503	816	1560	720
58118	699	259	1656	1208	1408	1148
256389	347	542	814	327	283	964
258207	597	694	2062	1154	474	1268
252122	1525	1031	2097	1900	2233	1853
262447	890	1780	1516	2301	2158	2249
258419	2218	1632	169	101	673	93
266418	180	406	920	1443	1469	1289
263978	831	1367	1577	1260	1092	1267
254915	289	80	1168	8	125	8
255549	536	278	1090	1131	1653	749
253323	1718	226	958	711	1811	465

Figure 5.5 gives the average ranking for each dimension of LSI with this new threshold and for LSICG. LSICG128 achieved the best average ranking at 799. The second highest average ranking for Rhino is LSI32 with an average ranking of 824. LSICG demonstrated an improvement of 25 positions for first returned relative method of each bug, which is a 3.03% improvement.

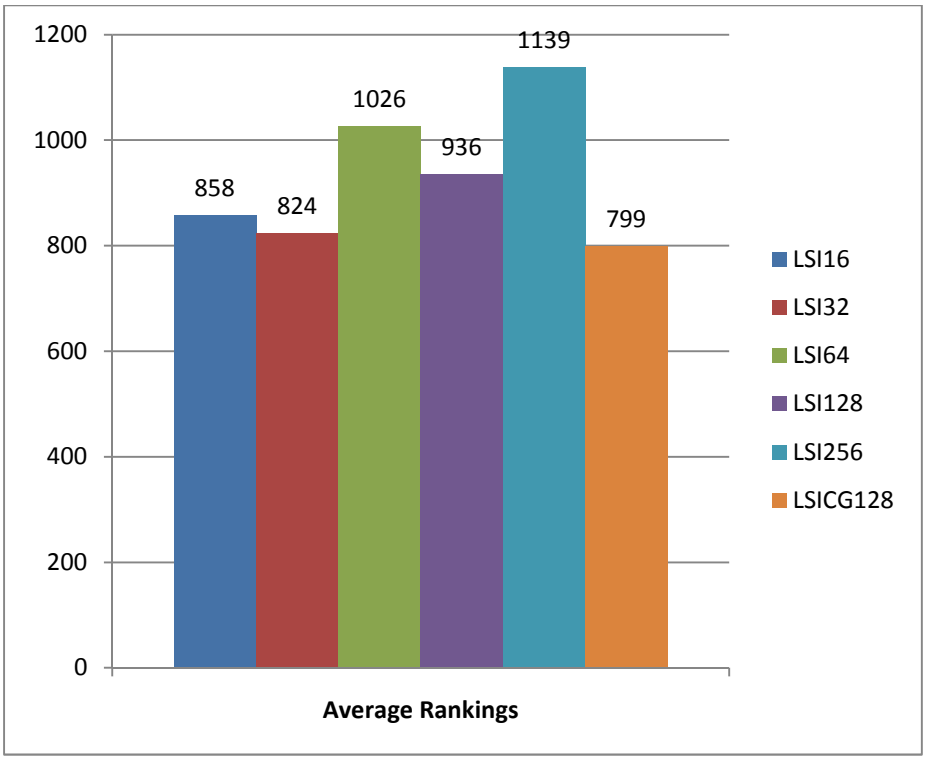


Figure 5.5 Average rankings from each technique for Rhino (adjusted threshold)

To compare LSICG and LSI at the dimension of 12, we use Wilcoxon's signed-rank test to determine whether the improvement in accuracy of LSICG128 over LSI128 is statistically significant for bug localization. We test the research study hypothesis and the result of Wilcoxon's test is W=34. The critical value W*= 38, $p < 0.01$. W<W*, so we reject the null hypothesis and accept the alternative hypothesis. There is significant evidence to indicate that LSICG128 is more accurate than LSI128.

5.4.3 jEdit

For jEdit, we picked 3 features (Liu et al. 2007) and applied feature location. The descriptions of the three features are given in Table 4 in section 5.3.3. We do the same evaluation in jEdit, as we did in JavaHMO and Rhino. The difference from the previous two cases is that for each feature

in jEdit, we used four different queries, which are obtained from Liu et al. 2007. Table 5.12 gives the queries for the three features.

Table 5.12 Queries for three features in jEdit

Feature	Query Number	Query
Search	Q11	Search find phrase word text
	Q12	Search final all forward backward case sensitive
	Q13	find search locate match indexof findnext
	Q14	searchdialog find findbtn searchselection save searchfileset searchandreplace
Add marker	Q21	Marker select word display text
	Q22	add remove marker markers
	Q23	Select highlight mark change background
	Q24	buffer addmarker marker selection
Show white space	Q31	red dot newline whitespace view show display tab
	Q32	show hide whitespace blank space display
	Q33	symbol replace changecolor setvisible addlayer whitespace loadsymbol
	Q34	userinput textareapainter paint whitespace newline pnt

The first relevant methods encountered in the search results for each feature are provided in Table 13 as follows:

Table 5.13 Three features and corresponding first relevant method in jEdit

Feature Name	First relevant method
Search	jedit.search.SearchAndReplace.find
Add marker	jedit.Buffer.addMarker
Show white space	jedit.textarea.TextAreaPainter.paintValidLine

For each query in Table 5.13, we run LSI in 16, 32, 64, 128, 256, and LSICG128 respectively. Table 5.14 gives the results for each treatment.

Table 5.14 Rankings for jEdit (subset)

Feature	Queries	LSI16	LSI32	LSI64	LSI128	LSI256	LSICG128
Search	Q11	5420	**186**	**10**	**24**	4639	**1**
	Q12	5403	340	807	225	2930	40
	Q13	**5068**	3326	3907	3512 *(-0.025)*	4602	Not in threshold
	Q14	5404	306	663	457	**1445**	109
Add Marker	Q21	3877	5260	4991	4375 *(-0.073)*	4178	Not in threshold
	Q22	4278	**4002**	4229	2911 *(-0.004)*	**415**	Not in threshold
	Q23	4154	5505	5040	2697	4618	1389
	Q24	**1845**	4834	**3713**	**2648**	807	**1197**
Show White Space	Q31	2743	2379	4214	3827 *(-0.040)*	3089	Not in threshold
	Q32	2804	5028	**1926**	495	1239	345
	Q33	3955	4074	2676	4746 *(-0.091)*	5127	Not in threshold
	Q34	**1298**	**2142**	2938	**78**	**259**	**49**

We use the average rankings to evaluate the results for the three features generated by each technique. Note that for each feature, we selected the best rankings at each dimension, regardless of which query is used. For example, for the "Search" feature, in Table 5.12 we selected the best rankings at the position of (LSI16, Q13) for LSI16, (LSI32, Q11) for LSI32, (LSI64, Q11) for LSI64, (LSI128, Q11) for LSI128, and (LSI256, Q14) for LSI256. We also select (LSICG128, Q11) as the best rankings returned by LSICG128 (best rankings are in bold).

LSICG returned at least one ranking for each query type (Search, Add Marker, Show White Space) because we used multiple queries for each query type. This enables us to directly compared LSICG's performance with LSI.

Figure 5.6 provides the average rankings for each feature by LSI and LSICG. The data indicate LSICG128 has the best average rankings among all the techniques. More specifically, LSICG's average of 416 is 291 positions better than the second best performer, LSI256 at an average ranking of 707. LSICG gives 41.2% improvement in the rankings over the second best results.

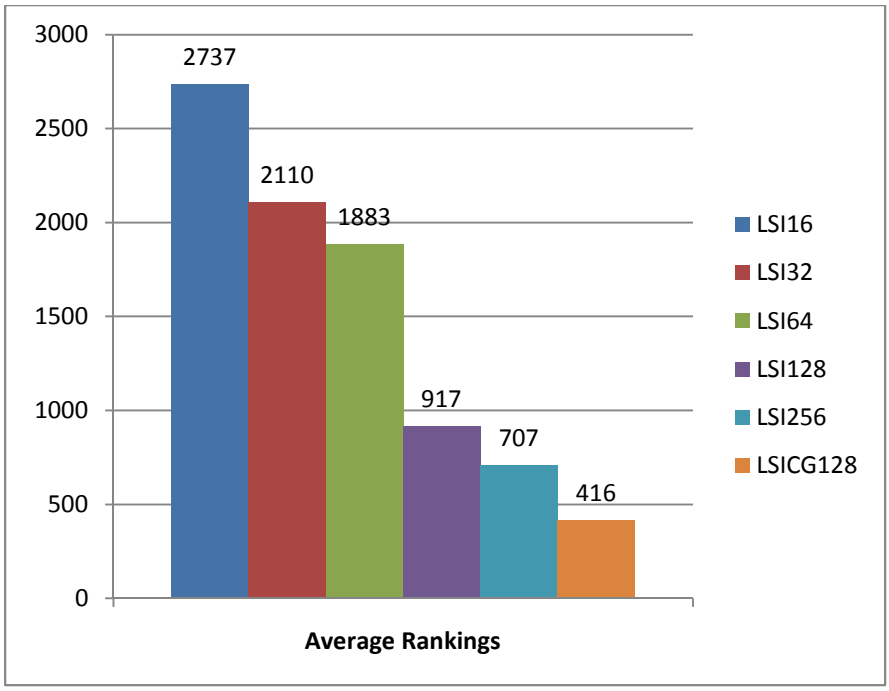


Figure 5.6 Average rankings for jEdit (subset)

5.4.4 Discussion of the results

The first experiment compares the performance of LSICG to LSI for concept location over three different Java projects. The concept location tasks included feature location and bug localization. Overall, LSICG provides a substantial increase in average rankings compared to LSI alone. The improvement in rankings for JavaHMO, Rhino and jEdit are 40.1%, 30.2%, and 41.2% respectively. The average ranking increase of 37.2% across the three case studies when using the optimal threshold value of an LSI score of 0.0. In the two cases, JavaHMO and Rhino, where we relaxed the threshold value, LSICG still gave an average ranking improvement of almost 10%.

For JavaHMO and Rhino, we also provided Wilcoxon's signed-rank test to prove that LSICG's improvement is statistically significant. These findings are promising and allow us to reject the $\mathbf{H_0}$ hypothesis of LSICG not being as accurate as LSI alone and accept the alternate hypothesis, $\mathbf{H_1}$, that LSICG is more accurate than LSI for concept location.

LSICG's improvement is most evident when LSI alone performs poorly. This is to be expected as the formulation of LSICG is to complement LSI with structural information. This is best seen in the Rhino study. For the Rhino study, the results in Table 11 show that LSICG is more accurate than LSI alone for 17 of the 21 bugs. Note that when the LSI rank is above 1,000, the LSICG rank is better for all 6 bugs and significantly better for 4 bugs. For example, the LSI rank for bug 252122 is 2,128, while the LSICG rank for that bug is 757. When the LSI rank is between 200 and 1,000, the LSICG rank is better for 7 of 9 bugs and significantly better for 2 bugs. For example, the LSI rank for bug 256836 is 209, while the LSICG rank for that bug is 27. When the LSI rank is less than 200, the LSICG rank is better for 3 of 6 bugs and significantly better for 2 bugs. For example, the LSI rank for bug 274996 is 120, while the LSICG rank for that bug is 34. However, for 2 bugs the LSI rank is significantly better than the LSICG rank. For example, the LSI rank for bug 249471 is 8, while the LSICG rank for that bug is 87.

5.5 All bugs/features

In the second experiment, all bugs and features identified from the three projects are examined. Specifically, the study uses 25 features from JavaHMO for feature location, 35 bugs from Rhino for bug localization, 6 bugs and 3 features from jEdit for bug localization and feature location.

5.5.1 JavaHMO

From the features list which includes 30 feature names and descriptions of JavaHMO on its official website, we successfully identified 25 features in JavaHMO to test the performance of LSI and LSICG. Table 5.15 lists the features in this study.

Table 5.15 All features in JavaHMO

Feature Number	Feature Description
1.	Rotate images
2.	View MP3 file tag information.
3.	Play MP3 streaming stations on the internet
4.	Automatically download Shoutcast playlists of your favorite streaming stations.
5.	Use the streaming proxy server to significantly improve on the inadequate support TiVo provides for online streaming stations.
6.	Play your MP3 files and streaming stations using both .m3u and .pls playlist formats.
7.	View live local weather conditions including current conditions, 5-day forecasts and radar images.
8.	Automatically download and view any image on the internet.
9.	iTunes playlists integration.
10.	Audio Jukebox.
11.	Organize images files based on their date information.
12.	Play interactive games such as TicTacToe.
13.	View images in the following formats: BMP, GIF, FlashPix, JPEG, PNG, PNM, TIFF, and WBMP.
14.	Play MP3 files.
15.	Sort items by different criteria.
16.	Organize MP3 files based on their ID3 tags.
17.	View a real-time image of your PC desktop.
18.	View local cinema listings
19.	Supports TiVo Beacon API.
20	View stock quotes.
21	Read email.
22	View NNTP images from newsgroups.
23	View RSS feeds.
24	View national weather alerts.
25	ToGo.

5.5.2 Rhino

We use all 35 bugs in Rhino 1.5R5 from two previous bug localization studies by Lukins et al. (2008; 2010). Except using the same queries in the first of these studies (Lukins et al., 2008), we further used two more queries for each bug. One type of query is made of the natural description of bug, and another is the combination of Lukins' queries and natural description of the bug. Table 5.16 lists information about the 35 bugs, including the bug numbers, bug titles, and terms extracted from the bug description. For each bug, the words in bold typeface were query used in Lukins et al., 2008.

Table 5.16 Thirty-five bugs in Rhino

Bug Number	**Bug title** [query words added from bug description]
256836	**Dynamic** scope and nested **functions**
274996	Exceptions with multiple **interpreters** on **stack** may lead to **ArrayIndex**OutOf**BoundsException** [**java wrapped**]
256865	Compatibility with **gcj**: changing **ByteCode**.<**constants**> to be int
257423	**Optimizer** regression: **this**.**name** += expression generates wrong **code**
238699	**Context.compileFunction** throws **InstantiationException**
256621	**throw statement**: **eol** should not be allowed
249471	**String index out** of range **exception [parse bound float global js native char]**
239068	Scope of **constructor functions** is not **initialized**
238823	**Context.compileFunction** throws **Null**Pointer **exception**
258958	**Lookup** of excluded **objects** in **ScriptableOutputStream** doesn't traverse **prototypes**/**parents**
58118	ECMA Compliance: **daylight savings time** wrong prior to **year** 1 [**day offset timezone**]
256389	Proper **CompilerEnvirons**.is**Xml**Available()
258207	**Exception name** should be **DontDelete [delete catch ecma obj object script]**
252122	**Double** expansion of **error message**
262447	NullPointerException in **ScriptableObject**.**getPropertyIds**
258419	**copy paste** bug in org.mozilla.javascript.**regexp**.**NativeRegExp [RE data back stack state track]**
266418	Can not **serialize reg**ular **exp**ressions **[regexp RE compile char set]**
263978	cannot **run** xalan example with Rhino 1.5 release 5 **[line number negative execute error]**
254915	Broken "**this**" for **name**() calls (CVS tip regression) **[object with**]
255549	**JVM**-dependent resolution of **ambiguity** when **calling Java methods**

	[argument constructor overload]
253323	Assignment to variable '**decompiler**' has no effect in **Parser** [**parse**]
244014	Removal of **code complexity limits** in the **interpreter**
244492	**JavaScript**Exception to extend **Runtime**Exception and common **exception** base
245882	**JavaImporter constructor**
254778	Rhino treats **label** as separated **statement**
255595	Factory **class** for **Context** creation **[*call default enter java runtime thread*]**
256318	NOT_FOUND and **ScriptableObject**.**equivalentValues**
256339	**Stack**less **interpreter**
256575	Mistreatment of **end**-of-**line** and **semi**/**colon [*eol*]**
257128	**Interpreter** and **tail call** elimination **[*java js stack*]**
258144	Add option to set **Runtime Class** in classes **generated** by **jsc [*run main script*]**
258183	**catch** (e if **condition**) does not **rethrow** of original **exception**
258417	java.long.**ArrayIndexOut**Of**BoundsException** in org.mozilla.javascript.**regexp**.**NativeRegExp [*stack size state data*]**
258959	**ScriptableInputStream** doesn't use **Context**'s application **ClassLoader** to **resolve** classes
261278	**Strict mode**

5.5.3 jEdit

For jEdit, we still use the three features in Liu et al. 2007. The details of the three features are in section 5.3.3.

Besides the three features, we further select 6 bugs from jEdit repository. For each bug, we construct 4 different queries to test LSI and LSICG. Table 5.17 lists the details of these bugs.

Table 5.17 Bug Information in jEdit

Bug Number	Bug Description
1275607	"find" field does not always receive focus
1467311	files were not* restored on startup when no file names were specified on command line
1469996	switching buffers from a macro confuses textarea
1488060	Menu "File/Recent Files" status line is inconsistent (working only when using mouse)
1607211	jEdit freezes when set "Search for:" to "||", check "Regular expressions", click "Replace All"
1642574	Change number of visible rows in buffer switcher in GUI

5.6 All bugs/features Results

This study compares the performance of LSI and LSICG for feature localization and bug localization. We compare the two techniques across three open-source Java projects (JavaHMO, Rhino, and jEdit) that have been studied previously in the literature. In particular, we select all bugs and features identified from the three projects, and compare the performance of LSI and LSICG at method level for these bugs and features. Where possible, we have used the same concept (bug report or feature request) as used in the literature. The following sections give the results of the study for the entire set of bugs and features from the three projects followed by an analysis of those results.

5.6.1 JavaHMO

The feature location task for JavaHMO includes locating the targeted method for each of 25 requested features. Table 5.18 below gives the features and the corresponding target methods in bold beneath each feature.

Table 5.18 Twenty-five features and targeted methods from JavaHMO

Feature Number	Feature Description **Targeted Methods (org.lnicholls.JavaHMO.)**
1.	Rotate images **media.ImageManipulator.rotate**
2.	View MP3 file tag information. **plugins.organizer.OrganizerContainer.findTas**
3.	Play MP3 streaming stations on the internet **media.Mp3Proxy.stream**
4.	Automatically download Shoutcast playlists of your favorite streaming stations. **plugins.shoutcast.ShoutcastStationContainer.getPlaylists()**
5.	Use the streaming proxy server to significantly improve on the inadequate support TiVo provides for online streaming stations. **server.ServerConfiguration.getUseStreamingProxy()**
6.	Play your MP3 files and streaming stations using both .m3u and .pls playlist formats. **model.FileSystemContainer.getItems(TiVoQueryContainerRequest)**
7.	View live local weather conditions including current conditions, 5-day forecasts and radar images. **plugins.weather.WeatherData.WeatherData()**
8.	Automatically download and view any image on the internet. **media.InternetImageItem.getImage(TiVoImageDocumentRequest, BufferedImage)**
9.	iTunes playlists integration. **plugins.iTunes.iTunesContainer.onPlaylist(Playlist)**
10.	Audio Jukebox. **plugins.jukebox.JukeboxItem.gatherItems(TiVoQueryContainerRequest)**
11.	Organize images files based on their date information. **imageOrganizer.OrganizerContainer.categorize(String, ImageProxy)**
12.	Play interactive games such as TicTacToe. **games.GamesContainer.start()**
13.	View images in the following formats: BMP, GIF, FlashPix, JPEG, PNG, PNM, TIFF, and WBMP. **ImageProxy.getImage()** **ImageProxy.getImageUsingImageLoader()**
14.	Play MP3 files. **media.Mp3Proxy.loadMetaData()**
15.	Sort items by different criteria. **TiVoQueryContainerRequest.TiVoQueryContainerRequest (ItemURL)** **TiVoQueryContainerRequest.setSortOrderTypes (String)**
16.	Organize MP3 files based on their ID3 tags. **plugins.organizer.OrganizerContainer.findTas()** **plugins.organizer.OrganizerContainer.Mp3OrganizerThread.Mp3 OrganizerThread(OrganizerContainer)**

	plugins.organizer.OrganizerContainer.Mp3OrganizerThread.findFiles() **plugins.organizer.OrganizerContainer.Mp3OrganizerThread.run()** **plugins.organizer.OrganizerContainer.categorize()**
17.	View a real-time image of your PC desktop. **plugins.desktop.DesktopItem.getImage(TiVoImageDocumentRequest, BufferedImage)**
18.	View local cinema listings **plugins.movies.MovieContainer.reload()**
19.	Supports TiVo Beacon API. **server.Beacon.Beacon(boolean, byte[])** **server.Beacon.Beacon(byte[])**
20	View stock quotes. **plugins.stocks.StocksContainer.reload()**
21	Read email. **plugins.email.EmailContainer.reload()**
22	View NNTP images from newsgroups. **plugins.nntp.NntpContainer.reload()**
23	View RSS feeds. **plugins.rss.RssChannel.reload()** **plugins.rss.RssContainer.start()**
24	View national weather alerts. **plugins.weather.WeatherContainer.reload()**
25	ToGo. **server.ToGoThread.run()** **util.ToGo.getRecordings(ArrayList, ProgressIndicator, ArrayList)**

For each feature, we run LSI at the dimension of 16, 32, 64, 128, and 256 respectively, and then we test LSICG at the same dimensions. For each feature, from the results of LSI at five dimensions, we select the best LSI result regardless of which dimension it comes from. The best result is named “BestLSI” and listed as the second last column in Table 5.19. Then for each feature we also select the best result of LSICG regardless of which dimension it is generated and also report it in Table 5.19. The best result of LSICG for each feature is named “BestLSICG” and listed as the last column in Table 5.19. Table 5.19 also lists rankings of LSI at each dimension. BestLSI gave the best average performance of LSI. For this reason, the discussion and analysis below is restricted to the BestLSICG and BestLSI.

Table 5.19 Rankings for JavaHMO (all set)

Feature Number	LSI16	LSI32	LSI64	LSI128	LSI256	BestLSI	BestLSICG
1	683	129	353	1219	1649	129	137
2	1404	1159	1647	355	42	42	42
3	1108	510	1	180	180	1	1
4	1194	1543	1203	1620	163	163	91
5	112	666	421	648	362	112	110
6	425	864	499	738	394	394	145
7	1491	1251	1212	895	373	373	390
8	288	481	505	533	1387	288	269
9	760	1148	778	598	1471	598	241
10	642	81	660	457	1322	81	79
11	258	434	515	614	621	258	134
12	242	1306	1122	1613	1054	242	238
13	530	292	1277	1389	1068	292	279
14	1499	457	1034	1052	1001	457	350
15	609	694	949	953	309	309	297
16	122	170	82	240	111	82	74
17	153	638	1621	1151	1236	153	151
18	1156	1360	1735	531	619	531	452
19	1217	1316	1460	898	85	85	80
20	363	135	218	119	1573	119	87
21	40	672	1065	1397	553	40	37
22	313	154	1068	95	266	95	91
23	985	559	695	575	778	559	286
24	628	208	923	188	76	76	52
25	1349	96	49	132	40	40	24

Figure 5.7 gives the average rankings of LSI for each dimension. It also lists the average rankings of BestLSI and BestLSICG. For LSI, LSI32 provides the best average ranking (653). The second best average ranking is 670 provided by LSI256. The average ranking of BestLSI is 221, which is much higher than LSI at any single dimension. However, the average ranking of BestLSICG obtains the best performance (166). On average, it is 55 positions higher than BestLSI for every feature giving a 24.89% improvement.

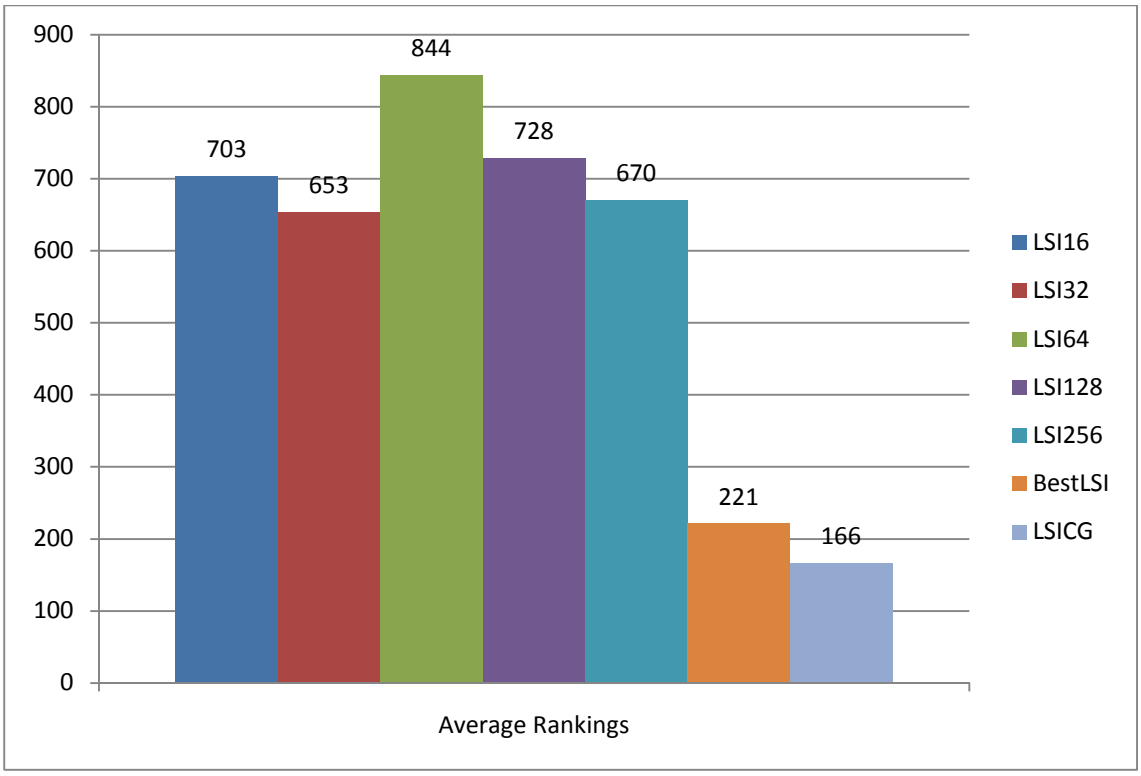


Figure 5.7 Average Rankings for JavaHMO (all set)

To determine whether this improvement of accuracy from BestLSICG over BestLSI is statistically significant, we use Wilcoxon's signed-rank test to test the case study hypothesis The result of Wilcoxon's test is W=21.5, and the critical value W*= 68, $p < 0.01$. W<W*, so we successfully reject the null hypothesis and accept the alternative hypothesis. There is significant evidence to indicate that BestLSICG is more accurate than BestLSI.

5.6.2 Rhino

For Rhino, the bug localization task includes isolating the correct method based on a given bug description. Table 20 below gives the bug number, the bug description followed by targeted class and method in italics for each bug.

Table 5.20 Thirty-five bugs in Rhino

Bug Number	Bug title [query words added from bug description]	Targeted (*Class, Method*)
256836	**Dynamic** scope and nested **functions**	*Context, hasCompileFunctionsWithDynamicScope*
274996	Exceptions with multiple **interpreters** on **stack** may lead to **ArrayIndex**OutOf**BoundsException** [**java wrapped**]	*Interpreter, interpret*
256865	Compatibility with **gcj**: changing **ByteCode**.<**constants**> to be int	*ClassFileWriter, addInvoke*
257423	**Optimizer** regression: **this**.**name** += expression generates wrong **code**	*Codegen, visitSetProp*
238699	**Context.compileFunction** throws **InstantiationException**	*Codegen, compile*
256621	**throw statement**: **eol** should not be allowed	*Parser, statementHelper*
249471	**String index out** of range **exception [parse bound float global js native char]**	*NativeGlobal, encode*
239068	Scope of **constructor functions** is not **initialized**	*IdScriptable, addAsPrototype*
238823	**Context.compileFunction** throws **Null**Pointer **exception**	*Context, compile*
258958	**Lookup** of excluded **objects** in **ScriptableOutputStream** doesn't traverse **prototypes**/**parents**	*ScriptableOutputStream, lookupQualifiedName*
58118	ECMA Compliance: **daylight savings time** wrong prior to **year** 1 [**day offset timezone**]	*NativeDate, date_format*
256389	Proper **CompilerEnvirons**.is**Xml**Available()	*CompilerEnvirons, isGeneratingSource*
258207	**Exception name** should be **DontDelete [delete catch ecma obj object script]**	*ScriptRuntime, newCatchScope*
252122	**Double** expansion of **error message**	*ScriptRuntime, undefWriteError*
262447	NullPointerException in **ScriptableObject**.**getPropertyIds**	*ScriptableObject, getPropertyIds*
258419	**copy paste** bug in org.mozilla.javascript.**regexp**.**NativeRegExp [RE data back stack state track]**	*NativeRegExp, matchRegExp*
266418	Can not **serialize reg**ular **exp**ressions **[regexp RE compile char set]**	*NativeRegExp, emitREBytecode*
263978	cannot **run** xalan example with Rhino 1.5 release 5 **[line number negative execute error]**	*Context, compileString*
254915	Broken "**this**" for **name**() calls (CVS tip regression) **[object with**]	*ScriptRuntime, getBase*
255549	**JVM**-dependent resolution of **ambiguity** when **calling Java methods [argument constructor**	*NativeJavaMethod, findFunction*

	overload]	
253323	Assignment to variable '**decompiler**' has no effect in **Parser** [**parse**]	*Parser, parse*
244014	Removal of **code complexity limits** in the **interpreter**	*Interpreter,interpret, generateICode, do_nameAndThis*
244492	**JavaScript**Exception to extend **Runtime**Exception and common **exception** base	*29 targeted methods, see Appendix A*
245882	**JavaImporter constructor**	*18 targeted methods, see Appendix A*
254778	Rhino treats **label** as separated **statement**	*27 targeted methods, see Appendix A*
255595	Factory **class** for **Context** creation **[*call default enter java runtime thread*]**	*Context,call; IFGlue,call, ifglue_call; JavaAdapter,callMetho)*
256318	NOT_FOUND and **ScriptableObject**.**equivalentValues**	*9 targeted methods, see Appendix A*
256339	**Stack**less **interpreter**	*26 targeted methods, see Appendix A*
256575	Mistreatment of **end**-of-**line** and **semi**/**colon [*eol*]**	*54 targeted methods, see Appendix A*
257128	**Interpreter** and **tail call** elimination **[*java js stack*]**	*11 targeted methods, see Appendix A*
258144	Add option to set **Runtime Class** in classes **generated** by **jsc [*run main script*]**	*7 targeted methods, see Appendix A*
258183	**catch** (e if **condition**) does not **rethrow** of original **exception**	*31 targeted methods, see Appendix A*
258417	java.long.**ArrayIndexOut**Of**BoundsException** in org.mozilla.javascript.**regexp**.**NativeRegExp** **[*stack size state data*]**	*NativeRegExp,MatchRegExp*
258959	**ScriptableInputStream** doesn't use **Context**'s application **ClassLoader** to **resolve** classes	*ScriptableInputStream, ResolveClass, ScriptableInputStream*
261278	**Strict mode**	*Main,processOptions; ScriptRuntime,evalSpecial, setName; ShellContextFactory,hasFeature, setStrictMode*

We use three different queries and five dimensions (16, 32, 64, 128, 256) to test the performance of LSI and LSICG on the 35 bugs. The three types of queries are:

- Lukins: Queries from Lukins' paper. The query strings are bold words in Table 5.20. For example, for bug 274996, the Lukins query is: **interpreters stack ArrayIndex BoundsException [java wrapped].**
- Lukins+Original: Queries from the combination of Lukins and bug descriptions. The query strings are all the words in the bug title in Table 20. For example, for bug 274996, the Lukins+Original query is: Exceptions with multiple **interpreters** on **stack** may lead to **ArrayIndex**OutOf**BoundsException [java wrapped].**
- Original: Queries from bug descriptions. For example, for bug 274996, the original query is: Exceptions with multiple **interpreters** on **stack** may lead to **ArrayIndex**OutOf**BoundsException.**

We provide the results of Lukins queries in Table 5.21. For each bug, we run LSI at the dimension of 16, 32, 64, 128, and 256 respectively, and then we test LSICG at the same dimensions. For each bug, we select the best LSI result regardless of which dimension it comes from named it "BestLSI". The result of BestLSI is listed as the second last column in Table 5.21. Again for each bug we also choose the best result of LSICG regardless of which dimension it is generated and also report these best results in Table 5.21. The best result of LSICG for each bug is named "BestLSICG" and listed as the last column in Table 5.21. In Table 5.21 the rankings of LSI at each dimension are reported as well. Because BestLSI had the best average performance of LSI, the discussion and analysis below is restricted to the BestLSICG and BestLSI.

Table 5.21 Rankings of thirty-five bugs (Lukins Query)

Bug Number	LSI16	LSI32	LSI64	LSI128	LSI256	BestLSI	BestLSICG
256836	766	565	1121	914	389	389	369
274996	1236	1926	1091	233	149	149	14
256865	112	364	70	93	106	70	18
257423	647	1405	1487	2060	978	647	624
238699	1862	1331	996	1556	1528	996	882
256621	317	1586	1293	965	1810	317	103
249471	277	69	291	144	323	69	68
239068	172	287	1365	876	835	172	156
238823	795	173	220	168	789	168	145
258958	1327	1789	672	929	1565	672	644
58118	644	369	1469	767	486	369	357
256389	218	75	330	116	21	21	25
258207	684	453	1283	278	343	278	286
252122	1495	1014	2089	1900	2238	1014	959
262447	763	1622	2178	2057	1848	763	706
258419	727	1299	50	82	361	50	49
266418	766	393	416	820	866	393	393
263978	833	632	731	1191	230	230	217
254915	142	128	237	15	40	15	13
255549	533	284	1080	1137	1548	284	225
253323	33	425	154	537	1062	33	33
244014	7	37	198	650	991	7	1
244492	161	142	15	40	167	15	4
245882	49	308	157	303	215	49	27
254778	59	76	34	106	201	34	26
255595	1097	235	693	538	664	235	222
256318	142	389	1603	2341	2251	142	130
256339	490	293	184	98	126	98	20
256575	117	38	13	8	7	7	5
257128	1195	1389	413	34	85	34	5
258144	664	870	614	1021	1387	614	590
258183	367	61	190	100	186	61	55
258417	1595	1418	706	68	741	68	66

258959	476	32	2274	2204	1400	32	243
261278	455	474	318	944	525	318	299

Figure 5.8 shows the average rankings of LSI for each dimension. It also lists the average rankings of BestLSI and BestLSICG. Among LSI, LSI16 provides the best average ranking (606) and LSI32 has the second best average ranking (670). The average ranking of BestLSI is 252, which is much higher than LSI at any single dimension. The average ranking of BestLSICG obtains the best performance (228) of all. It averages 24 positions higher than BestLSI for every feature yielding a 9.52% improvement.

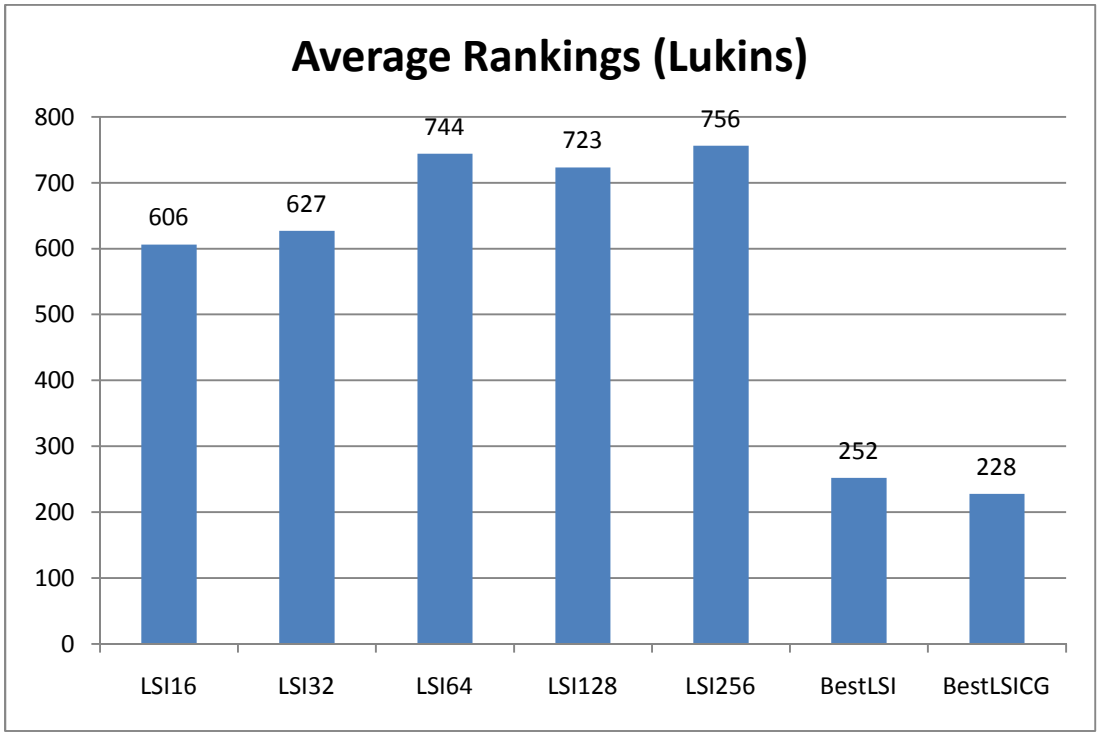


Figure 5.8 Average Rankings for Rhino (all set, Lukins Query)

To determine whether this improvement of accuracy from BestLSICG over BestLSI is statistically significant, we use Paired-Samples t test to test the research study hypothesis. We did not use Wilcoxon's signed-rank test in this case because Wilcoxon's signed-rank test works

well for small size of subjects (size < = 25). When the number of subjects is large (usually larger than 25), we use Paired-Samples t test as an alternative. The assumptions of this test are:

1. Scores are independent of each other within samples, but related between samples.
2. Scores are normally distributed.

A bivariate analysis is also provided to determine the relationship between LSICG and LSI. In particular, we use Pearson product-moment correlation coefficient as the measure of correlation between LSICG and LSI. The correlation is reported in the result of Paired-Samples t test.

For this case, since there are 35 bugs, we use Paired-Samples t test. The results are listed in Table 5.22.

Table 5.22 Paired-Samples t test (all set, Lukins Query)

Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	LSICG	227.9714	35	266.57539	45.05947
	LSI	251.8000	35	280.65908	47.44004

Paired Samples Correlations				
		N	Correlation	Sig.
Pair 1	LSICG & LSI	35	.977	.000

Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval of the Difference				
					Lower	Upper			
Pair 1	LSICG - LSI	-23.82857	60.86266	10.28767	-44.73563	-2.92151	-2.316	34	.027

The result of Paired-Samples t test is t=-2.316, and the critical value t* is between 1.697 and 1.684 with p = 0.027 and alpha = .05. |t|>t*, so we successfully reject the null hypothesis and accept the alternative hypothesis. There is significant evidence to indicate that BestLSICG is more accurate than BestLSI.

Next, we provide the results of Lukins+Original queries in Table 5.23 including rankings obtained from BestLSI and BestLSICG.

Table 5.23 Rankings of thirty-five bugs (Lukins+Original Query)

Bug Number	LSI16	LSI32	LSI64	LSI128	LSI256	BestLSI	BestLSICG
256836	808	832	653	339	641	339	366
274996	1303	1696	602	163	165	163	1
256865	50	177	248	693	507	50	9
257423	589	1113	634	2059	1313	589	554
238699	1862	1331	996	1556	1528	996	883
256621	343	1659	1248	1099	2084	343	103
249471	281	64	132	125	284	64	59
239068	238	617	1361	1082	967	238	175
238823	790	174	220	163	790	163	99
258958	721	1226	503	816	1560	503	456
58118	699	259	1561	1208	509	259	226
256389	241	87	59	182	131	59	59
258207	597	694	2062	1154	474	474	467
252122	1525	1031	2097	1900	2233	1031	968
262447	890	1780	1516	2301	2158	890	836
258419	864	1632	169	101	673	101	93
266418	61	372	920	1443	1469	61	52
263978	831	616	734	1260	183	183	174

254915	289	80	109	8	125	8	8
255549	536	278	1090	1131	1653	278	224
253323	187	226	423	711	1422	187	185
244014	7	37	198	650	991	7	1
244492	92	68	51	174	103	51	4
245882	49	308	157	303	215	49	27
254778	51	75	22	134	141	22	8
255595	1113	234	712	539	468	234	208
256318	142	417	1462	2281	339	142	125
256339	490	293	184	98	126	98	20
256575	117	38	13	8	7	7	5
257128	1195	1389	413	34	85	34	5
258144	206	1277	1368	1116	1358	206	195
258183	169	34	83	47	52	34	3
258417	1831	1719	1193	90	779	90	76
258959	1035	157	747	189	609	157	175
261278	455	474	318	944	525	318	299

Figure 5.9 indicates the average rankings of LSI at each dimension as well as the average rankings of BestLSI and BestLSICG. Among LSI, LSI16 provides the best average ranking (590) and LSI32 has the second best average ranking (642). The average ranking of BestLSI is 241, which is higher than LSI at any single dimension. The average ranking of BestLSICG obtains the best performance (204) of all. It averages 37 positions higher than BestLSI for every bug for a 15.35% improvement.

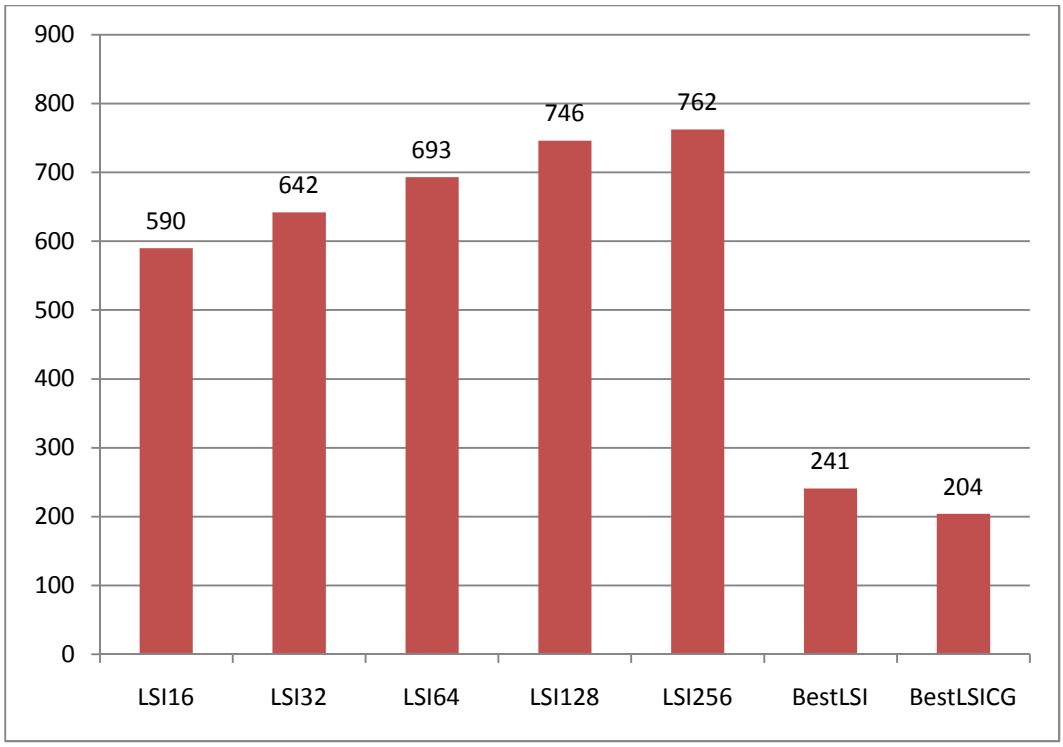


Figure 5.9 Average Rankings for Rhino (all set, Lukins + Original Query)

Again, we use Paired-Samples t test to test the research study hypothesis and to determine whether this improvement of accuracy from BestLSICG over BestLSI is statistically significant. The statistical results are listed in the following Table 5.24.

Table 5.24 Paired-Samples t test (Lukins+Original Query)

Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	LSICG	204.2286	35	259.06587	43.79012
	LSI	240.8000	35	270.63040	45.74489

Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	LSICG & LSI	35	.983	.000

Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval of the Difference				
					Lower	Upper			
Pair 1	LSICG - LSI	-36.57143	50.78402	8.58407	-54.01635	-19.12651	-4.260	34	.000

The absolute of t equals to 4.260, and the critical value t* is between 1.697 and 1.684 with p = 0.000 and alpha = 0.05. |t|>t*, so we successfully reject the null hypothesis and accept the alternative hypothesis. There is significant evidence to indicate that BestLSICG is more accurate than BestLSI.

At last, we report the results of original queries in Table 5.25. Table 5.25 gives the rankings from LSI at each dimension as well as BestLSI and BestLSICG. For 20 out of the 35 bugs, the results of the original queries are the same as those of Lukins+Original queries. The reason is that the queries are the same for both types. For example, for bug 256836, the query from original and Lukins+Original is the same: **Dynamic** scope and nested **functions.** For the remaining 15 bugs, the queries differed and the original queries provided different results from Lukins+Original.

Table 5.25 Rankings of thirty-five bugs (Original Query)

Bug Number	LSI16	LSI32	LSI64	LSI128	LSI256	BestLSI	BestLSICG
256836	808	832	653	339	641	339	366
274996	1178	1830	791	155	255	155	3
256865	50	177	248	693	507	50	9
257423	589	1113	634	2059	1313	589	554
238699	1862	1331	996	1556	1528	996	883
256621	343	1659	1248	1099	2084	343	103
249471	256	66	356	36	120	36	34
239068	238	617	1361	1082	967	238	175
238823	790	174	220	163	790	163	99
258958	721	1226	503	816	1560	503	456
58118	729	453	1206	1088	667	453	411
256389	241	87	59	182	131	59	59
258207	704	416	1845	128	57	57	56
252122	1525	1031	2097	1900	2233	1031	968
262447	890	1780	1516	2301	2158	890	836
258419	1983	2118	469	59	136	59	47
266418	284	101	197	40	109	40	19
263978	637	262	629	2139	179	179	173
254915	411	220	125	59	384	59	55
255549	544	275	918	1201	876	275	228
253323	229	180	631	630	1525	180	148
244014	7	37	198	650	991	7	1
244492	92	68	51	174	103	51	4
245882	49	308	157	303	215	49	27
254778	51	75	22	134	141	22	8
255595	1364	279	577	659	844	279	259
256318	142	417	1462	2281	339	142	125
256339	490	293	184	98	126	98	20
256575	115	40	12	8	7	7	7
257128	837	1177	702	626	188	188	2
258144	191	1364	1680	770	1017	191	180
258183	169	34	83	47	52	34	3
258417	1387	1784	1295	98	429	98	93
258959	1035	157	747	189	609	157	175
261278	455	474	318	944	525	318	299

Figure 5.10 indicates the average rankings of LSI, BestLSI and BestLSICG from the original queries including the 20 duplicated queries with Lukins+Original. Among the LSI results, LSI16 provides the best average ranking (611) and LSI32 has the second best average ranking (642).

The average ranking of BestLSI is 238, which is higher than LSI at any single dimension. The average ranking of BestLSICG has the best performance (197). It averages 41 positions higher than BestLSI for every bug. That is a 17.23% improvement.

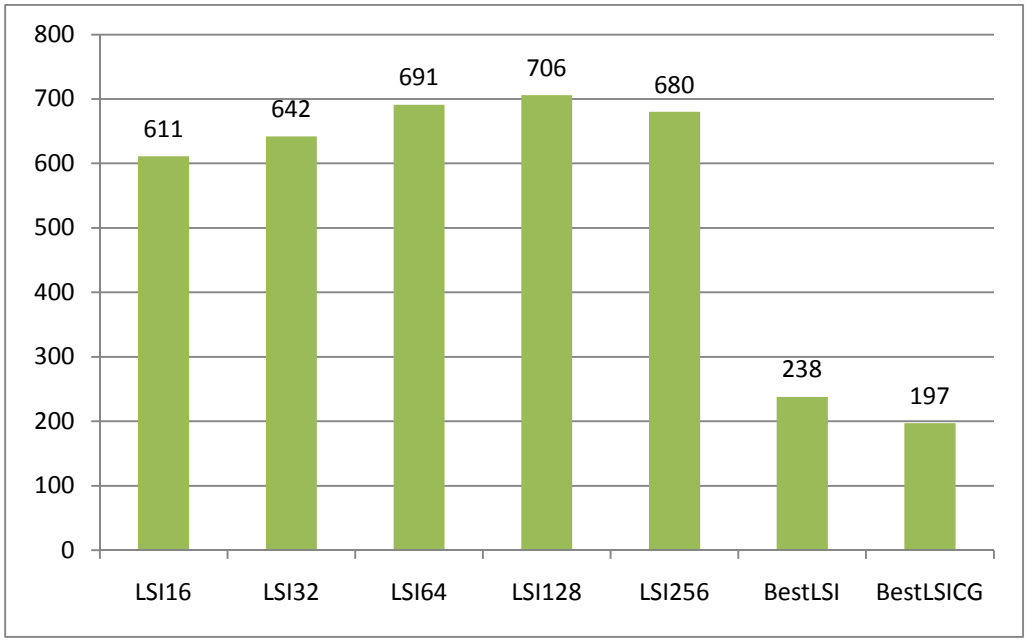


Figure 5.10 Average Rankings for Rhino (all set, Original Query)

We used a Paired-Samples t test to evaluate the research study's hypothesis and to determine whether this improvement of accuracy from BestLSICG over BestLSI is statistically significant. The statistical results are given in Table 5.26.

Table 5.26 Paired-Samples t test (Original Query)

Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	LSICG	196.7143	35	259.65606	43.88988
	LSI	238.1429	35	270.50108	45.72303

Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	LSICG & LSI	35	.979	.000

Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval of the Difference				
					Lower	Upper			
Pair 1	LSICG - LSI	-41.42857	55.81497	9.43445	-60.60168	-22.25546	-4.391	34	.000

The absolute of t equals to 4.391, and the critical value t* is between 1.697 and 1.684 with p = 0.000 and alpha = .05. |t|>t*, so we successfully reject the null hypothesis and accept the alternative hypothesis. There is significant evidence to indicate that BestLSICG is more accurate than BestLSI.

Figure 5.11 shows the average ranking of LSI at each dimension including BestLSI for the three different queries used in the study. From this figure observe that the performance of LSI using Original Queries is slightly better than LSI using the other two types of queries. In particular, LSI with Original queries (238) is 3 positions higher than LSI with Lukins+Original (241), and 14 positions higher than Lukins (252). For this reason, original queries are used for the bug localization task on Rhino at the class level.

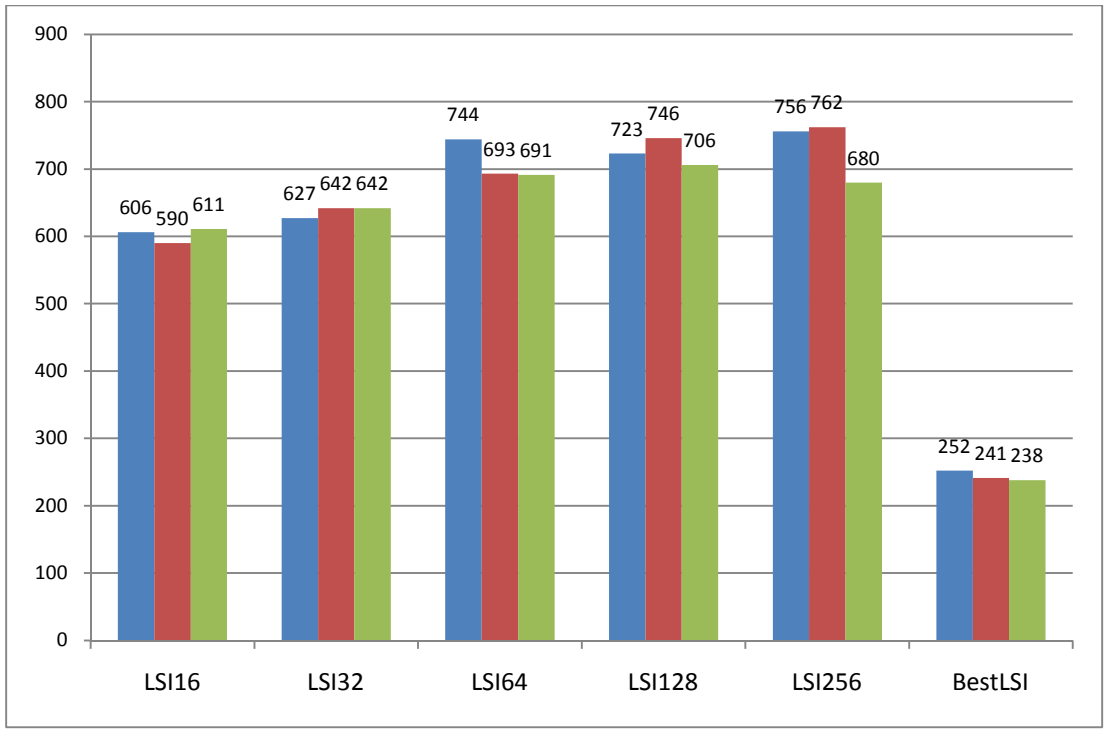


Figure 5.11 Average rankings for Rhino (Three Queries)

Figure 5.12 lists the average ranking of BestLSI and BestLSICG from the three types of queries. Statistical evidence demonstrates that for each type of query BestLSICG outperforms BestLSI.

For additional analysis, the best results of BestLSI regardless of query are selected. This set of results is named it BestofALL. Similarly, the best results from BestLSICG are evaulated. The average ranking of BestofALL from LSICG is 179, which is 29 positions higher than the result from LSI (209), which is a 13.94% improvement.

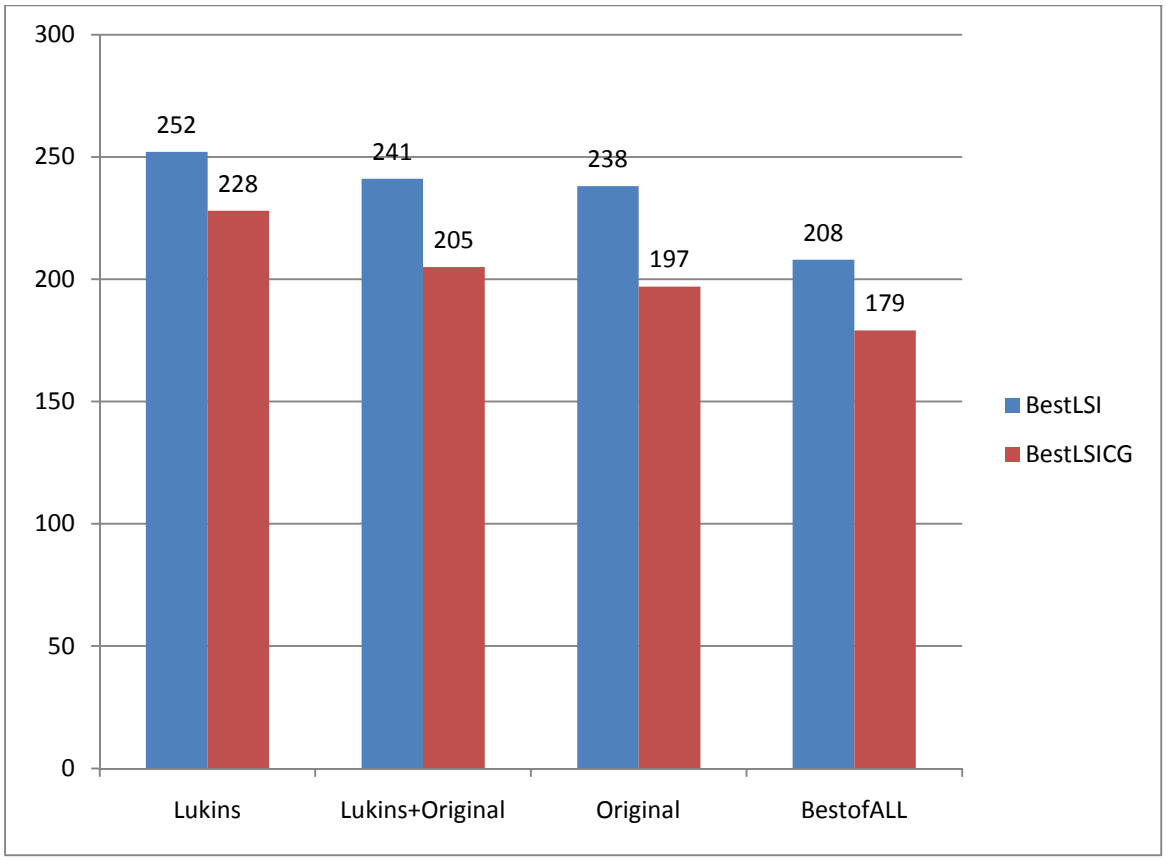


Figure 5.12 Average rankings of BestLSI and BestLSICG (three queries)

The improvement of the BestofAll result from BestLSICG over the BestofAll result from BestLSI is evaluated using a Paired-Samples t test to test the research study hypothesis. The statistical results are listed Table 5.27.

Table 5.27 Paired-Samples t test (BestofALL)

Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	BestofALL_LSICG	178.8286	35	250.12520	42.27888
	BestofALL_LSI	207.5429	35	266.19689	44.99549

Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	BestofALL_LSICG & BestofALL_LSI	35	.979	.000

Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval of the Difference				
					Lower	Upper			
Pair 1	BestofALL_LSICG BestofALL_LSI	-28.71429	55.12423	9.31770	-47.65012	-9.77845	-3.082	34	.004

|t| = 3.082, while the critical value t* is between 1.697 and 1.684 with p = 0.004 and alpha = 0.05. |t|>t*, so we successfully reject the null hypothesis and accept the alternative hypothesis. There is significant evidence to indicate that the BestofALL from BestLSICG is more accurate than the BestofALL from BestLSI.

Note that the average rankings of BestLSICG from Original (197) and Lukins+Original (205) are even higher than the BestofAll results from LSI (209). This indicates that LSICG using a single type of query outperforms LSI using all three types of queries. This result indicates that LSICG, with just one type of query, outperforms the others using multiple query types. LSICG provides better results in less time than LSI with three different types of queries.

5.6.3 jEdit

For the feature location task in jEdit, we continue to use the 3 features from Liu et al. 2007. For each feature, there are four different queries. The evaluation executes LSI at 16, 32, 64, 128 and 256 dimensions and then run LSICG at the same dimensions. When evaluating the performances of LSI and LSICG, instead of comparing their results at the dimension of 128 (LSI128 and LSICG128), the results are compared from BestLSI and BestLSICG. BestLSI and BestLSICG are defined as the best result returned for each feature, regardless of which query and which dimension it is from. Table 5.28 provides the rankings of them as well as the rankings of LSI results from every dimension.

Table 5.28 Rankings of features for jEdit

	Queries	LSI16	LSI32	LSI64	LSI128	LSI256	BestLSI	BestLSICG
Search	Q11	5420	186	10	24	4639	10	1
	Q12	5403	340	807	225	2930		
	Q13	5068	3326	3907	3512	4602		
	Q14	5404	306	663	457	1445		
Add Marker	Q21	3877	5260	4991	4375	4178	415	377
	Q22	4278	4002	4229	2911	415		
	Q23	4154	5505	5040	2697	4618		
	Q24	1845	4834	3713	2648	807		
Show White Space	Q31	2743	2379	4214	3827	3089	78	49
	Q32	2804	5028	1926	495	1239		
	Q33	3955	4074	2676	4746	5127		
	Q34	1298	2142	2938	78	259		

Figure 5.13 provides the average rankings for each feature by BestLSI and BestLSICG. The data indicate BestLSICG provides better average rankings than BestLSI. More specifically, BestLSICG's average of 142 is 26 positions better than BestLSI at an average ranking of 168. BestLSICG gives 15.48% improvement in the rankings over the results of BestLSI.

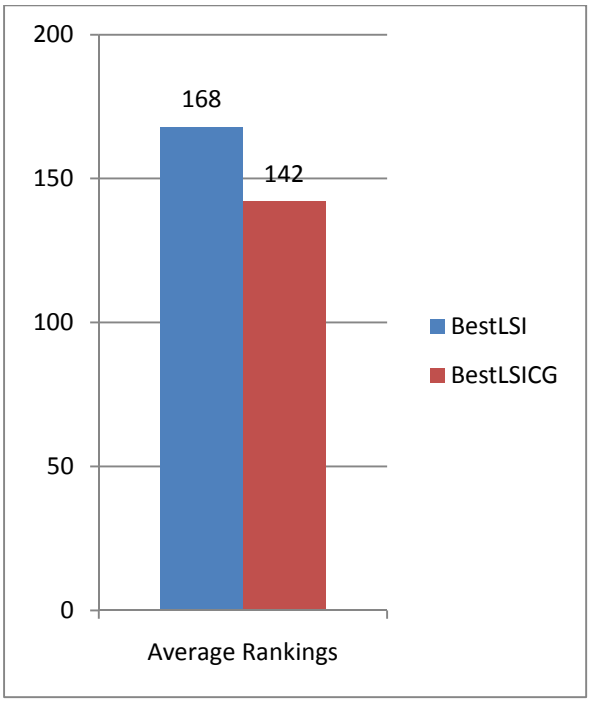


Figure 5.13 Average rankings for features in jEdit

For bug localization task, it includes locating the correct method based on a given bug description. We use 6 bugs extracted from jEdit repository. Table 5.29 below gives the bug number, the bug description followed by targeted method in bold for each bug.

Table 5.29 Six bugs for jEdit

Bug no.	Bug Description **Targeted Method**
1275607	"find" field does not always receive focus **requestFocus(final Window win, final Component comp)** **windowActivated(WindowEvent)** **run()**
1467311	files were not* restored on startup when no file names were specified on command line **finishStartup(boolean, boolean, String, String[])** **run()**
1469996	switching buffers from a macro confuses textarea **invoke(View)**
1488060	Menu "File/Recent Files" status line is inconsistent (working only when using mouse)

	update(JMenu)
1607211	jEdit freezes when set "Search for:" to "||", check "Regular expressions", click "Replace All" **_replace(Buffer, SearchMatcher, int, int, boolean)**
1642574	Change number of visible rows in buffer switcher in GUI **_init()** **_save()** **handleBufferUpdate(BufferUpdate)**

For each bug, we construct four different queries. The details of the queries are listed in Table 5.30.

Table 5.30 Queries for six bugs in jEdit

Bug Number	Bug Queries	
1275607	**Q11**	"find" field does not always receive focus
	Q12	find search request focus window
	Q13	focus request
	Q14	focus focusWindow request requestWindow Window window Win win
1467311	**Q21**	files were *not* restored on startup when no file names were specified on command line
	Q22	restore file startup
	Q23	restore file startup command line
	Q24	startup Startup startupFile restore Restore restoring _restore restoreFile restore_file file user command line
1469996	**Q31**	Menu "File/Recent Files" status line is inconsistent (working only when using mouse)
	Q32	Menu file recent status mouse
	Q33	Menu file recent status mouse path name
	Q34	menu Menu menus Menus file recent file recentFile status mouse Mouse mouseMenu MouseMenu _mouse _Mouse path paths pathMenu path_menu
1488060	**Q41**	Menu "File/Recent Files" status line is inconsistent (working only when using mouse)
	Q42	Menu file recent status mouse
	Q43	Menu file recent status mouse path name
	Q44	menu Menu menus Menus file recent file recentFile status mouse Mouse mouseMenu MouseMenu _mouse _Mouse path paths

		pathMenu path_menu
1607211	**Q51**	jEdit freezes when set "Search for:" to "||", check "Regular expressions", click "Replace All"
	Q52	search replace searchMatcher
	Q53	replace match matcher search segment line text
	Q54	replace match replaceMatch matcher replaceMatcher ReplaceMatcher search searchMatcher segment line text
1642574	**Q61**	Change number of visible rows in buffer switcher in GUI
	Q62	change buffer switcher
	Q63	change set edit view buffer switcher list file text area status
	Q64	change changeBufferSwitcher buffer Buffer bufferSwitcher BufferSwitcher buffer_switcher switcher Switcher setBufferSwitcher editBufferSwitcher viewBufferSwitcher set edit view

For each bug and each query, we run LSI at 16, 32, 64, 128 and 256 dimensions as well as LSICG at the same dimension. By comparing the results from BestLSI and BestLSICG, we evaluate the two techniques' performances. Table 5.31 provides the rankings of BestLSI, BestLSICG, and rankings of LSI results from every dimension.

Table 5.31 Rankings for six bugs in jEdit

Bug no.	Queries	LSI16	LSI32	LSI64	LSI128	LSI256	BestLSI	BestLSICG
1275607	Q11	1006	1361	1333	1419	688	369	360
	Q12	2068	369	896	2683	1903		
	Q13	1709	487	768	643	2071		
	Q14	1586	2068	2993	2951	583		
1467311	Q21	833	2767	1502	1811	4053	349	350
	Q22	2233	1591	349	745	1077		
	Q23	1176	2102	2548	1429	3287		
	Q24	1736	2704	542	785	2081		
1469996	Q31	3731	2826	2198	560	115	97	84
	Q32	3762	2799	2160	452	97		
	Q33	3759	3714	2554	1471	2222		
	Q34	3779	3343	2833	1385	1953		
1488060	Q41	3113	2642	773	155	48	47	31
	Q42	3022	2980	1869	140	48		
	Q43	3000	3802	1607	126	47		
	Q44	2830	3547	1786	189	142		

1607211	Q51	1691	2784	3490	2117	2629	225	191
	Q52	5470	4199	4624	3080	851		
	Q53	4997	4587	5249	5104	1021		
	Q54	5260	4460	5147	3760	225		
1642574	Q61	323	658	328	861	759	94	81
	Q62	517	599	493	632	562		
	Q63	1036	88	94	326	698		
	Q64	587	651	1381	2164	2774		

Figure 5.14 lists the average rankings of LSI at each dimension. It also lists the average ranking for BestLSI and BestLSICG. For LSI, LSI256 provides the best average ranking (432) and the second best average ranking is from LSI128 (735). The average ranking of BestLSI is 196, which is much higher than LSI at any single dimension. The average ranking of BestLSICG obtains the best performance at a ranking of 183. That is 13 positions higher than BestLSI for each bug with an improvement of 6.63%.

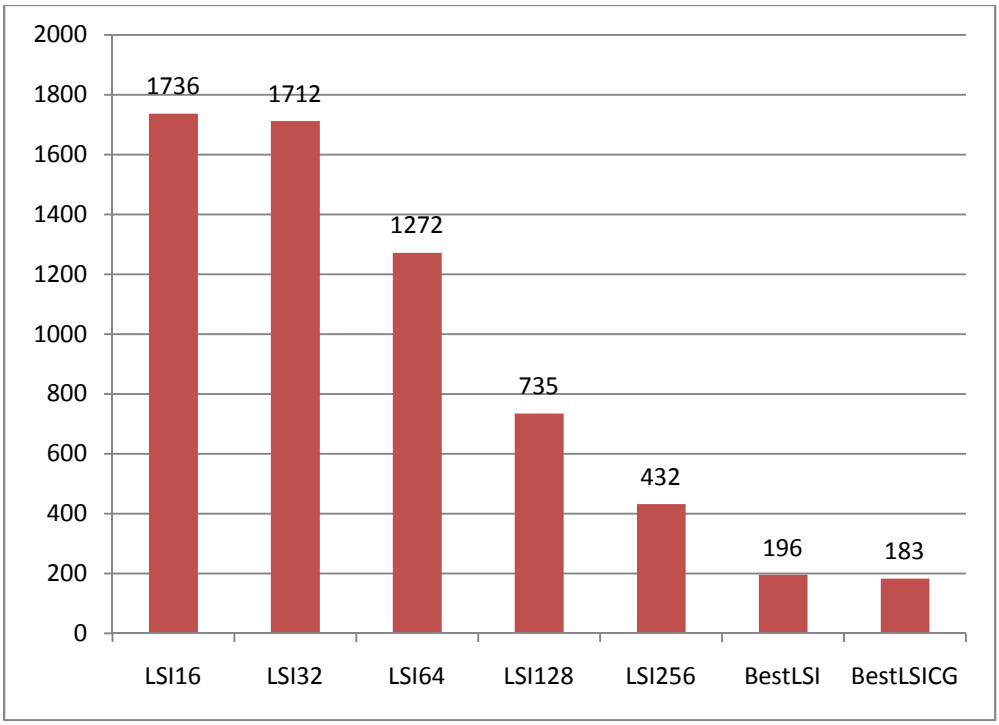


Figure 5.14 Average rankings for six bugs in jEdit

Note that both BestLSI and BestLSICG obtain better results from bug 1469996, bug 1488060 and bug 1642574 than from the other 3 bugs. The reasons that the other 3 bugs have lower rankings include:

1. They have a small number of relevant methods.
2. The relevant methods have a small size.
3. The key words in relevant methods are common.

Figure 5.15 provides the average ranking of BestLSI and BestLSICG for the top 3 bugs.

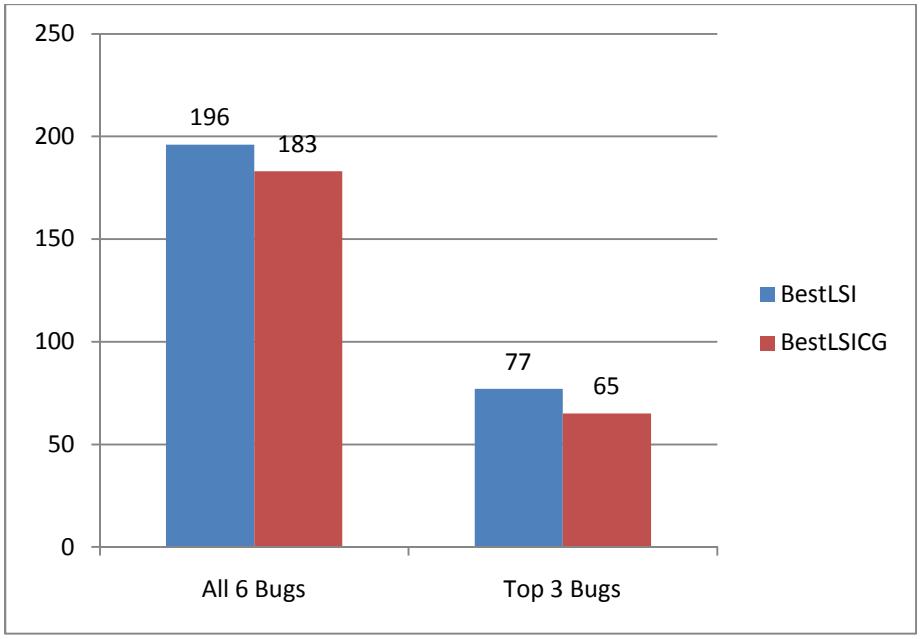


Figure 5.15 Average rankings of BestLSI and BestLSICG for bugs in jEdit

BestLSI has an average of 77 for the top 3 bugs. Compared with BestLSI for all 6 bugs, BestLSI for 3 bugs averages 119 positions higher. BestLSICG has an average of 65 for the top 3 bugs. It is 118 positions higher than BestLSICG for all 6 bugs (185). BestLSICG for the top 3 bugs also gains, on average, 12 positions higher than BestLSI for the top 3 bugs, which is a 15.58% improvement.

To provide the statistical evidence that LSICG improves accuracy over LSI for feature location and bug localization in jEdit, we combine the results from 3 features and 6 bugs. Figure

5.16 gave the average rankings of BestLSI and BestLSICG. For all the features and bugs, BestLSICG obtains an average ranking of 169. Compared with the result from BestLSI which is 186, it is 17 positions higher or a 9.14% improvement.

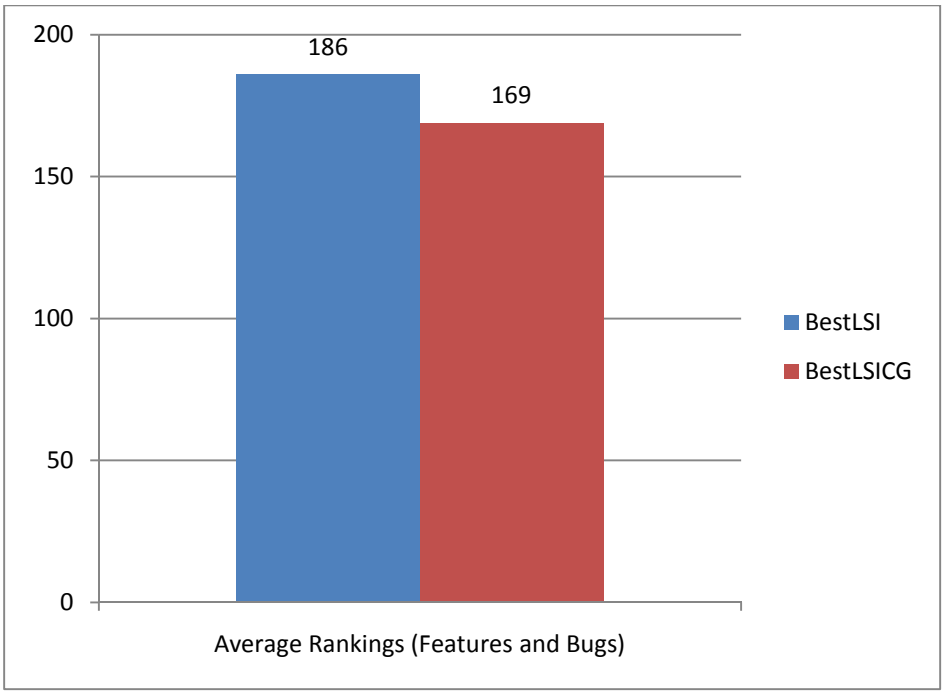


Figure 5.16 Average rankings of LSI and LSICG for all features and bugs in jEdit

To determine whether this improvement of accuracy from BestLSICG over BestLSI is statistically significant, we use Wilcoxon's signed-rank test to test the research study hypothesis The result of Wilcoxon's test is W=1, and the critical value W*= 6, $p < 0.05$. W<W*, so we successfully reject the null hypothesis and accept the alternative hypothesis. There is significant evidence to indicate that BestLSICG is more accurate than BestLSI for feature location and bug localization in jEdit.

5.6.4 Discussion of results

The second experiment compares the performance of LSICG to LSI for concept location over three different Java projects. All identified features and bugs from the three projects are tested in the concept location tasks including feature location and bug localization. LSICG provides a substantial increase in average rankings compared to LSI alone. The improvement of BestLSICG in rankings over BestLSI for JavaHMO, Rhino and jEdit are 24.89%, 13.94%, and 15.58% respectively. The average ranking increases 18.14% across the three case studies when using the optimal threshold value of an LSI score of 0.0. For JavaHMO, Rhino, and jEdit, we also provided Wilcoxon's signed-rank test and Paired-samples t test to prove that LSICG's improvement is statistically significant. The statistical results allow us to reject the $\mathbf{H_0}$ hypothesis of LSICG not being as accurate as LSI alone and accept the alternate hypothesis, $\mathbf{H_1}$, that LSICG is more accurate than LSI for concept location.

For the JavaHMO study, the results in Table 19 indicate that LSICG is more accurate than LSI alone for 84% (21) of 25 features. LSICG performs the same to LSI for 8% (2) of the 25 features. However, LSICG is less accurate than LSI for 8% (2) of the 25 features.

For the Rhino study, the results in Table 5.21 using Lukins query indicate that LSICG is more accurate than LSI alone for 85.7% (30) of 35 features. LSICG performs the same to LSI for 5.7% (2) of the 35 bugs. However, LSICG is less accurate than LSI for 8.6% (3) of the 35 bugs. The results in Table 5.23 using Lukins+Original query indicate that LSICG is more accurate than LSI alone for 88.6% (31) of 35 features. LSICG performs the same to LSI for 5.7% (2) of the 35 bugs. However, LSICG is less accurate than LSI for 8.6% (2) of the 35 bugs. The results in Table 5.25 using original query indicate that LSICG is more accurate than LSI alone for 88.6% (31) of

35 features. LSICG performs the same to LSI for 5.7% (2) of the 35 bugs. However, LSICG is less accurate than LSI for 5.7% (2) of the 35 bugs.

For the jEdit study, the results in Table 5.28 and Table 5.30 indicate that LSICG is more accurate than LSI alone for 89% (8) of 3 features and 6 bugs. LSICG performs the same to LSI for 11.1% (1) of the 3 features and 6 bugs.

Overall, for the three projects JavaHMO, Rhino and jEdit, LSICG is more accurate than LSI alone for 87% (60) of the 69 features and bugs. LSICG performs the same to LSI for 7% (5) of the 69 features and bugs, and LSICG is less accurate than LSI for 6% (4) of the 69 features and bugs. The overall performance of LSICG is promising. It indicates that LSICG outperforms or at least performs the same as LSI on 94% of all identified bugs and features in this research study.

5.7 Class Level all bugs/features

In the third experiment, we examine class level structural connectivity. The study uses the bugs and features identified from the three projects: 25 features from JavaHMO, 35 bugs from Rhino, and 3 features as well as 6 bugs from jEdit. For this class level study, each document in the corpora represents one class from the source code. The ranking of the documents represents the ranking of relevant classes to a given bug or feature.

5.7.1 JavaHMO

For JavaHMO, we use the identified 25 features to test the performance of LSI and LSICG at the class level. Table 5.32 lists the features information including feature number, feature description as well as targeted class names for each feature.

Table 5.32 Features and targeted classes for JavaHMO

Feature Number	Feature Description **Targeted Classes**
1.	Rotate images **org.lnicholls.JavaHMO.media.ImageManipulator**
2.	View MP3 file tag information. **org.lnicholls.JavaHMO.plugins.organizer.OrganizerContainer**
3.	Play MP3 streaming stations on the internet **org.lnicholls.JavaHMO.media.Mp3Proxy**
4.	Automatically download Shoutcast playlists of your favorite streaming stations. **org.lnicholls.JavaHMO.plugins.shoutcast.ShoutcastStationContainer**
5.	Use the streaming proxy server to significantly improve on the inadequate support TiVo provides for online streaming stations. **org.lnicholls.JavaHMO.server.ServerConfiguration**
6.	Play your MP3 files and streaming stations using both .m3u and .pls playlist formats. **org.lnicholls.JavaHMO.model.FileSystemContainer**
7.	View live local weather conditions including current conditions, 5-day forecasts and radar images. **org.lnicholls.JavaHMO.plugins.weather.WeatherData**
8.	Automatically download and view any image on the internet. **org.lnicholls.JavaHMO.media.InternetImageItem**
9.	iTunes playlists integration. **org.lnicholls.JavaHMO.plugins.iTunes.iTunesContainer**
10.	Audio Jukebox. **org.lnicholls.JavaHMO.plugins.jukebox.JukeboxItem**
11.	Organize images files based on their date information. **org.lnicholls.JavaHMO.plugins.imageOrganizer.OrganizerContainer**
12.	Play interactive games such as TicTacToe. **org.lnicholls.JavaHMO.plugins.games.GamesContainer**
13.	View images in the following formats: BMP, GIF, FlashPix, JPEG, PNG, PNM, TIFF, and WBMP. **org.lnicholls.JavaHMO.media.ImageProxy**
14.	Play MP3 files. **org.lnicholls.JavaHMO.media.Mp3Proxy**
15.	Sort items by different criteria. **org.lnicholls.JavaHMO.model.TiVoQueryContainerRequest**
16.	Organize MP3 files based on their ID3 tags. **org.lnicholls.JavaHMO.plugins.organizer.OrganizerContainer**
17.	View a real-time image of your PC desktop. **org.lnicholls.JavaHMO.plugins.desktop.DesktopItem**
18.	View local cinema listings **org.lnicholls.JavaHMO.plugins.movies.MovieContainer**
19.	Supports TiVo Beacon API. **org.lnicholls.JavaHMO.server.Beacon**

20	View stock quotes. **org.lnicholls.JavaHMO.plugins.stocks.StocksContainer**
21	Read email. **org.lnicholls.JavaHMO.plugins.email.EmailContainer**
22	View NNTP images from newsgroups. **org.lnicholls.JavaHMO.plugins.nntp.NntpContainer**
23	View RSS feeds. **org.lnicholls.JavaHMO.plugins.rss.RssChannel** **org.lnicholls.JavaHMO.plugins.rss.RssContainer**
24	View national weather alerts. **org.lnicholls.JavaHMO.plugins.weather.WeatherContainer**
25	ToGo. **org.lnicholls.JavaHMO.server.ToGoThread** **org.lnicholls.JavaHMO.util.ToGo**

There are totally 27 targeted classes for 25 features. On average, each feature has 1.08 relevant classes.

5.7.2 Rhino

We also use all 35 bugs in Rhino 1.5R5 from two previous bug localization studies by Lukins et al. (2008; 2010). The Rhino results in Section 5.6.2 indicate that using Original queries obtains the best average rankings of LSI. So instead of using three different queries as described in Section 5.6.2, we only used Original queries for each bug. Original queries are constructed from the natural description of the bug. The details of the queries are in Table 20 in Section 5.6.2. In this section Table 5.33 lists information about the 35 bugs, including the bug numbers, bug description as well as targeted classes for each bug. In Rhino, there are a total of 85 targeted classes for 35 bugs. Each bug has an average of 2.43 relevant classes.

Table 5.33 Bugs and targeted classes for Rhino

Bug Number	Bug Description	*Targeted Classes*
256836	Dynamic scope and nested functions	*Context*
274996	Exceptions with multiple interpreters on stack may lead to ArrayIndexOutOfBoundsException	*Interpreter*
256865	Compatibility with gcj: changing ByteCode.<constants> to be int	*ClassFileWriter*
257423	Optimizer regression: this.name += expression generates wrong code	*Codegen*
238699	Context.compileFunction throws InstantiationException	*Codegen*
256621	throw statement: eol should not be allowed	*Parser*
249471	String index out of range exception	*NativeGlobal*
239068	Scope of constructor functions is not initialized	*IdScriptable*
238823	Context.compileFunction throws NullPointer exception	*Context*
258958	Lookup of excluded objects in ScriptableOutputStream doesn't traverse prototypes/parents	*ScriptableOutputStream*
58118	ECMA Compliance: daylight savings time wrong prior to year 1	*NativeDate*
256389	Proper CompilerEnvirons.isXmlAvailable()	*CompilerEnvirons*
258207	Exception name should be DontDelete	*ScriptRuntime*
252122	Double expansion of error message	*ScriptRuntime*
262447	NullPointerException in ScriptableObject.getPropertyIds	*ScriptableObject*
258419	copy paste bug in org.mozilla.javascript.regexp.NativeRegExp	*NativeRegExp*
266418	Can not serialize regular expressions	*NativeRegExp*
263978	cannot run xalan example with Rhino 1.5 release 5	*Context*
254915	Broken "this" for name() calls (CVS tip regression)	*ScriptRuntime*
255549	JVM-dependent resolution of ambiguity when calling Java methods	*NativeJavaMethod*
253323	Assignment to variable 'decompiler' has no effect in Parser	*Parser*
244014	Removal of code complexity limits in the interpreter	*Interpreter*
244492	JavaScriptException to extend RuntimeException and common exception base	*EcmaError, EvaluatorException, Interpreter, JavaAdapter, JavaScriptException, Main, NativeGlobal, NativeJavaObject, ToolErrorReporter,*

		WrappedException
245882	JavaImporter constructor	*Context, IdScriptable, ImporterFunctions, ImporterTopLevel*
254778	Rhino treats label as separated statement	*Codegen, CompilerEnvirons, Interpreter, IRFactory, Jump, Node, NodeTransformer, OptTransformer, Parser*
255595	Factory class for Context creation	*Context, IFGlue, JavaAdapter*
256318	NOT_FOUND and ScriptableObject.equivalentValues	*Namespace, QName, ScriptableObject, ScriptRuntime, XMLObjectImpl*
256339	Stackless interpreter	*Context, InterpretedFunction, Interpreter, State*
256575	Mistreatment of end-of-line and semi/colon	*Decompiler, IRFactory, NativeRegExp, Parser, Token, TokenStream*
257128	Interpreter and tail call elimination	*Interpreter*
258144	Add option to set Runtime Class in classes generated by jsc	*ClassCompiler, Codegen, Main*
258183	catch (e if condition) does not rethrow of original exception	*BodyCodegen, Interpreter, IRFactory, Node, ScriptRuntime, Token*
258417	java.long.ArrayIndexOutOfBoundsException	*NativeRegExp*
258959	ScriptableInputStream doesn't use Context's application ClassLoader to resolve classes	*ScriptableInputStream*
261278	Strict mode	*Main, ScriptRuntime, ShellContextFactory*

5.7.3 jEdit

For jEdit, we use the three features and six bugs which we previously used in section 5.6.3. For each feature and bug, we still use four different queries to test the performance of LSI and LSICG at class level.

Table 5.34 lists the details of the three features including feature number, feature description and targeted class for each of them. Each feature has just one relevant class.

Table 5.34 Features and targeted class in jEdit

Feature Name	Feature Description **Targeted Class**
Search	searching for the occurrence of the provided search phrase. **org.gjt.sp.jedit.search.SearchAndReplace**
Add marker	adding a marker to the selected line in the text. **org.gjt.sp.jedit.Buffer**
Show white space	showing whitespaces as a symbol in the text. **org.gjt.sp.jedit.textarea.TextAreaPainter**

Table 5.35 lists the details of the six bugs including bug number, bug description as well as targeted class names for each bug. Each feature has just one relevant class. We identified 8 relevant classes for the 6 bugs. Each bug has an average of 1.33 relevant classes.

Table 5.35 Bugs and targeted class in jEdit

Bug no.	Bug Description **Targeted Classes**
1275607	"find" field does not always receive focus **org.gjt.sp.jedit.GUIUtilities**
1467311	files were not* restored on startup when no file names were specified on command line **org.gjt.sp.jedit.jEdit**
1469996	switching buffers from a macro confuses textarea **org.gjt.sp.jedit.Macros** **org.gjt.sp.jedit.Macros.Macro**
1488060	Menu "File/Recent Files" status line is inconsistent (working only when using mouse) **org.gjt.sp.jedit.menu.RecentFilesProvider**
1607211	jEdit freezes when set "Search for:" to "||", check "Regular expressions", click "Replace All" **org.gjt.sp.jedit.search.SearchAndReplace**
1642574	Change number of visible rows in buffer switcher in GUI **org.gjt.sp.jedit.options.ViewOptionPane** **org.gjt.sp.jedit.EditPane**

5.8 Class Level all bugs/features Results

This study compares the performance of LSI and LSICG for the tasks of feature localization and bug localization. We still compare the two techniques across three open-source Java projects (JavaHMO, Rhino, and jEdit) that have been studied previously in the literature. In particular, we select all bugs and features identified from the three projects, however, this time we compare the performance of LSI and LSICG at class level rather than method level for all bugs and features. Where possible, we have used the same concept (bug report or feature request) as used in the literature. The following sections give the results of the study at class level for the entire set of bugs and features from the three projects followed by an analysis of those results.

5.8.1 JavaHMO

The feature location task for JavaHMO includes locating the targeted classes for each of 25 requested features. For each feature, at class level we run LSI with the dimension of 16, 32, 64, 128, and 246 respectively. We use 246 instead of 256 because 246 is the maximum number of documents (classes) for JavaHMO. Then we apply LSICG at the same dimensions. For each feature, from the results of LSI at five dimensions, we select the best LSI result regardless of which dimension it comes from. The best result is named “BestLSI” and listed as the second last column in Table 5.36. For each feature we select the best result of LSICG regardless of which dimension it is generated and also report it in Table 5.36. The best result of LSICG for each feature is named “BestLSICG” and listed as the last column in Table 5.36. Table 5.36 also lists rankings from LSI at each dimension. BestLSI gave the best average performance of LSI. For this reason, the discussion and analysis below is restricted to the BestLSICG and BestLSI.

Table 5.36 Rankings for JavaHMO (class level)

Feature Number	LSI16	LSI32	LSI64	LSI128	LSI246	BestLSI	BestBLSICG
1	81	164	72	59	223	59	50
2	59	203	232	209	12	12	4
3	59	2	10	97	45	2	1
4	45	208	187	166	181	45	36
5	53	1	2	54	60	1	1
6	124	231	150	48	78	48	30
7	73	138	139	95	40	40	40
8	35	43	101	65	202	35	23
9	1	34	8	13	10	1	1
10	49	8	1	7	7	1	1
11	163	230	239	146	44	44	15
12	115	83	65	209	41	41	32
13	92	21	31	107	197	21	5
14	12	2	1	54	50	1	1
15	106	76	74	56	192	56	19
16	64	162	151	128	14	14	5
17	122	145	184	58	136	58	44
18	27	30	19	178	31	19	2
19	105	138	152	45	110	45	35
20	173	166	125	87	63	63	30
21	74	69	23	25	179	23	18
22	27	41	91	135	70	27	17
23	2	29	43	22	52	2	1
24	130	198	172	226	196	130	95
25	79	141	174	28	104	28	19

Figure 5.17 gives the average rankings of relevant class for LSI at each dimension. It also lists the average rankings for BestLSI and BestLSICG. For LSI, LSI16 provides the best average ranking (75). The second best average ranking is 93 provided by LSI128. LSI256 gave the similar average rankings (94) with LSI128. The average ranking of BestLSI is 33, which is much higher than LSI at any single dimension. However, the average ranking of BestLSICG obtains the best performance (21). It averages 12 positions higher than BestLSI for every feature provding a 36.36% improvement.

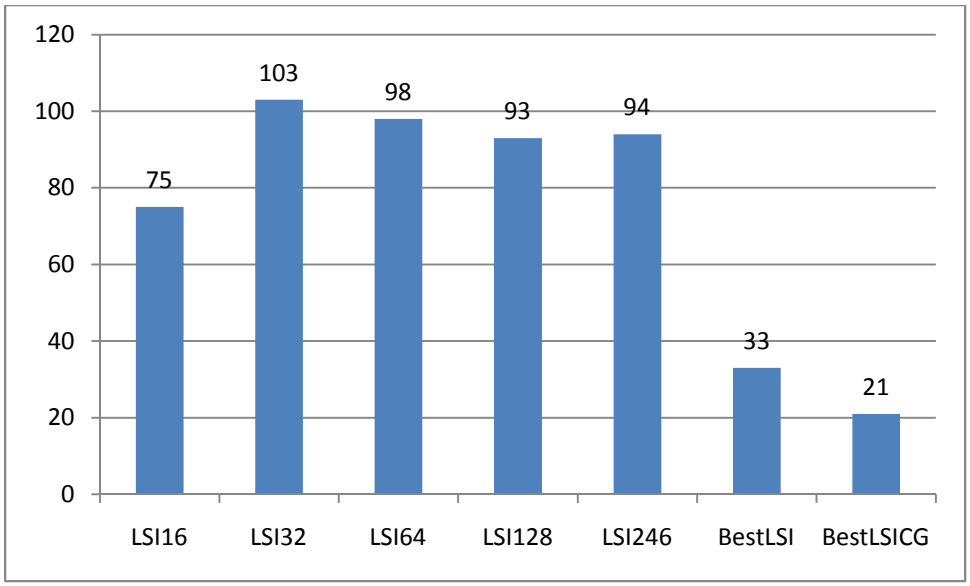


Figure 5.17 Average Rankings for JavaHMO (class level)

To determine whether this improvement of accuracy from BestLSICG over BestLSI at class level is statistically significant, we use Wilcoxon's signed-rank test to test the research study hypothesis. The result of Wilcoxon's test is W=0, while the critical value W*= 38, $p < 0.01$. W<W*, so we successfully reject the null hypothesis and accept the alternative hypothesis. There is significant evidence to indicate that BestLSICG is more accurate than BestLSI at class level for JavaHMO.

Except analyzing the results for the entire set of features in JavaHMO, we also select a series of subsets for evaluation. Figure 5.18 provides the average rankings of BestLSI and BestLSICG for top 25 features, top 20 features, top 15 features and top 10 features.

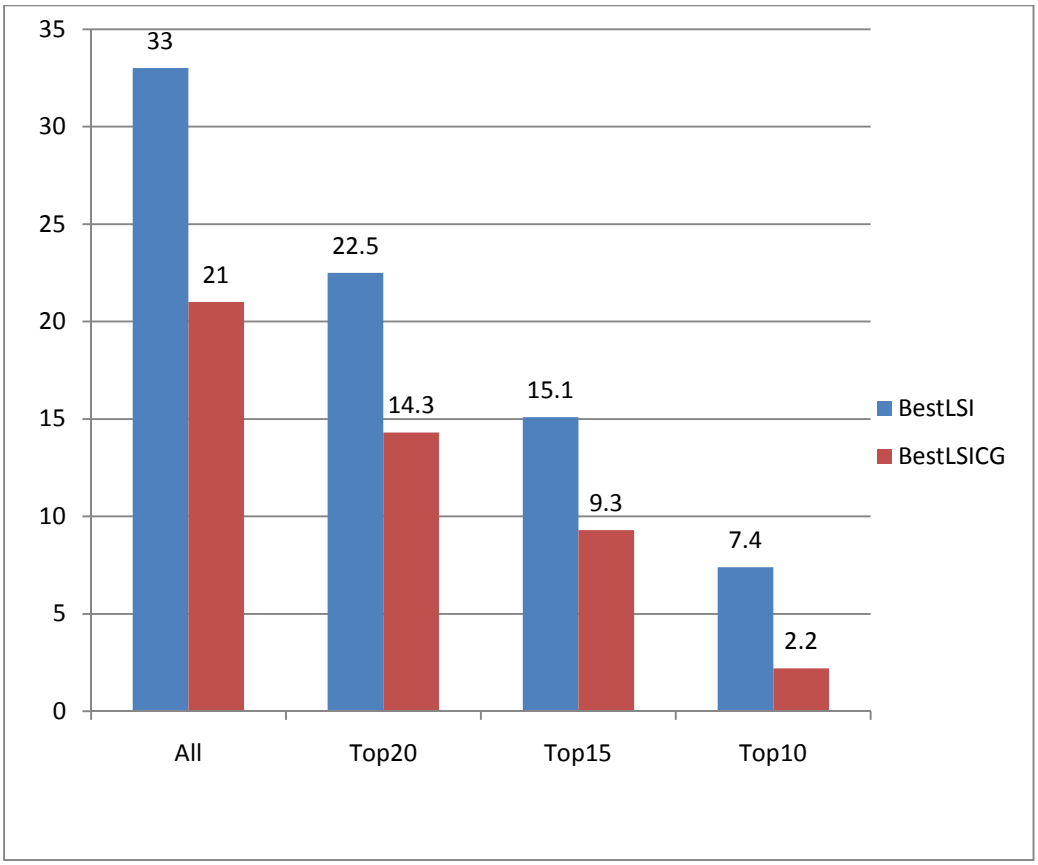


Figure 5.18 Average rankings of BestLSI and BestLSICG for features in JavaHMO

For the top 20 features, BestLSICG obtains a 14.3 average rankings, which is 8.2 positions higher, or 36.44% improvement than BestLSI (22.5). For top 15 features, BestLSICG provided an average ranking of 9.3, which is 5.8 average positions higher than BestLSI (15.1), or an improvement of 38.41%. For top 10 features, BestLSICG achieved 2.2 average rankings, compared with 7.4 from BestLSI. This improvement is 5.2 positions for each feature, or 70.27%.

5.8.2 Rhino

For Rhino, the bug localization task includes isolating the correct class based on a given bug description. For 35 bugs in Rhino, at class level we use Original queries and five dimensions (16, 32, 64, 128, and 201) to test the performance of LSI and LSICG. We use 201 rather than 256 for the fifth dimension because Rhino only contains 201 classes. In this study, we provide results

from both LSI and LSICG at each dimension. For this reason, the evaluation and discussion below is restricted to LSI and LSICG at the same dimension. We also compared and analyze BestLSI and BestLSICG.

The results of rankings from LSI are given in Table 5.37. For each bug, we run LSI at the dimension of 16, 32, 64, 128, and 201 respectively. For each bug, we select the best LSI result regardless of which dimension it comes from named it "BestLSI". The result of BestLSI is listed as the last column in Table 5.37. In Table 5.37, the rankings of LSI at each dimension are also reported.

Table 5.37 Rankings of LSI for bugs in Rhino (class level)

Bug Number	LSI16	LSI32	LSI64	LSI128	LSI201	BestLSI
256836	90	1	16	125	42	1
274996	128	151	96	37	147	37
256865	47	73	122	159	27	27
257423	103	172	32	64	175	32
238699	126	85	13	46	175	13
256621	182	68	43	146	126	43
249471	10	40	117	44	100	10
239068	132	138	180	196	89	89
238823	87	146	137	146	88	87
258958	73	85	81	17	167	17
58118	49	5	14	34	139	5
256389	54	138	75	1	21	1
258207	178	7	138	28	42	7
252122	156	23	22	189	160	22
262447	149	127	92	46	99	46
258419	142	130	1	49	72	1
266418	139	7	30	44	72	7
263978	66	7	132	42	88	7
254915	88	37	1	26	42	1
255549	173	178	168	145	75	75
253323	160	101	68	197	76	68
244014	148	15	40	60	147	15
244492	10	45	51	5	12	5
245882	133	7	38	8	88	7
254778	16	11	17	72	27	11
255595	1	7	19	56	48	1
256318	168	72	89	15	103	15

256339	62	5	59	1	55	1
256575	28	5	19	20	30	5
257128	193	136	63	129	55	55
258144	17	48	56	39	12	12
258183	59	12	72	15	42	12
258417	178	153	58	141	72	58
258959	74	182	187	21	54	21
261278	11	61	61	69	4	4

We provide the results of rankings for LSICG in Table 5.38. For each bug, we run LSICG at the dimension of 16, 32, 64, 128, and 201 respectively. For each bug, we also select the best LSICG result regardless of which dimension it comes from named it “BestLSICG”. The result of BestLSICG is listed as the last column in Table 5.38. In Table 5.38 the rankings of LSICG at each dimension are reported as well.

Table 5.38 Rankings of LSICG for bugs in Rhino (class level)

Bug Number	LSICG16	LSICG32	LSICG64	LSICG128	LSICG201	BestLSICG
256836	27	2	3	2	2	2
274996	93	117	13	3	6	3
256865	11	14	30	36	1	1
257423	85	167	11	21	31	11
238699	116	59	6	21	32	6
256621	174	33	12	86	8	8
249471	8	34	99	32	21	8
239068	127	134	175	196	41	41
238823	39	86	51	90	5	5
258958	78	88	88	20	180	20
58118	44	5	10	26	42	5
256389	34	107	27	1	8	1
258207	140	1	33	1	19	1
252122	117	4	1	98	38	1
262447	128	84	26	6	31	6
258419	140	110	1	45	49	1
266418	125	4	9	14	9	4
263978	31	3	38	1	29	1
254915	51	40	1	2	18	1
255549	171	174	164	130	22	22
253323	144	90	18	187	9	9
244014	122	1	2	4	33	1

244492	11	16	3	3	27	3
245882	86	1	35	8	29	1
254778	1	11	4	2	2	1
255595	1	4	4	5	29	1
256318	136	13	3	1	5	1
256339	47	1	5	1	29	1
256575	2	4	1	1	1	1
257128	179	79	10	33	10	10
258144	24	55	44	11	13	11
258183	11	1	5	1	2	1
258417	171	148	27	104	8	8
258959	101	187	190	25	141	25
261278	12	16	65	1	19	1

Figure 5.19 shows the average rankings of LSI compared with LSICG at each dimension. It also lists the average rankings of BestLSI and BestLSICG. From the figure we can observe that LSICG has higher average rankings than LSI at every dimension. In particular, at the dimension of 16 (D16), the average ranking of LSICG is 80. Compared with the average rankings of LSI (98), LSICG gains 18 positions higher than LSI for each bug, an 18.37% improvement. At the 32 dimension (D32), LSICG has an average ranking of 54, which is 17 positions higher than the result from LSI (71), or an improvement of 23.94% over LSI. For 64-dimension (D64), LSICG provided the average ranking of 35, while LSI only provided the average rankings of 69. LSICG has averages a 34 position higher rankings than LSI for each bug, or a 49.28% improvement. This situation is similar for the dimension of 128 (D128), where LSICG has an average ranking of 35, which is a 50% improvement or averages a 35 position improvement over LSI with the average ranking of 70. For dimension 201 (D201), LSICG obtains the best results among the five dimensions, which equals to 27. The average ranking for LSI is 79. Therefore, at this dimension, LSICG's ranking avearges 52 positions higher than LSI's. LSICG has an improvement of 65.82% on LSI. The average ranking of BestLSI is 23, which is much higher than LSI at any single dimension. The average ranking of BestLSICG, 6, obtains the best performance of all. It is 17 positions higher than BestLSI for every bug, which is a 73.91% improvement.

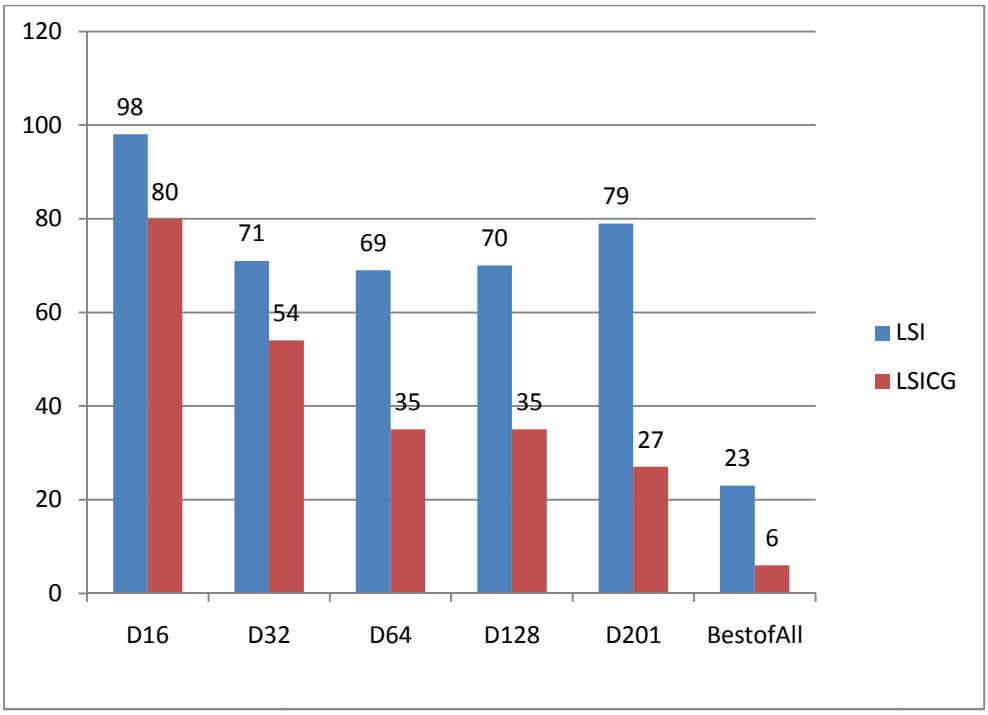


Figure 5.19 Average rankings for Rhino (class level)

To determine whether at every dimension, the accuracy improvement of result from LSICG over the result from LSI is statistically significant, we run Paired-Samples t test using SPSS to test the research study hypothesis. We choose six pairs to test the hypothesis: (LSI16, LSICG16), (LSI32, LSICG32), (LSI64, LSICG64), (LSI128, LSICG128), (LSI201, LSICG201), (BestLSI, BestLSICG). The statistical results are listed in Table 5.39.

Table 5.39 Paired-Samples t test (LSI and LSICG at every dimension)

Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	LSICG16	79.6286	35	59.23388	10.01235
	LSI16	98.0000	35	59.98480	10.13928
Pair 2	LSICG32	54.0857	35	58.37020	9.86637
	LSI32	70.8000	35	61.41364	10.38080
Pair 3	LSICG64	34.6857	35	50.23578	8.49140
	LSI64	68.7714	35	51.64135	8.72898

Pair 4	LSICG128	34.8000	35	52.11063	8.80830
	LSI128	69.4857	35	60.69808	10.25985
Pair 5	LSICG201	27.1143	35	36.24327	6.12623
	LSI201	79.1714	35	49.62061	8.38741
Pair 6	BestLSICG	6.3714	35	8.67751	1.46677
	BestLSI	23.3714	35	25.85717	4.37066

Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	LSICG16 & LSI16	35	.947	.000
Pair 2	LSICG32 & LSI32	35	.943	.000
Pair 3	LSICG64 & LSI64	35	.789	.000
Pair 4	LSICG128 & LSI128	35	.825	.000
Pair 5	LSICG201 & LSI201	35	.332	.052
Pair 6	BestLSICG & BestLSI	35	.602	.000

Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval of the Difference				
					Lower	Upper			
Pair 1	LSICG16 - LSI16	-18.37143	19.38175	3.27611	-25.02929	-11.71357	-5.608	34	.000
Pair 2	LSICG32 - LSI32	-16.71429	20.50763	3.46642	-23.75890	-9.66967	-4.822	34	.000
Pair 3	LSICG64 - LSI64	-34.08571	33.10089	5.59507	-45.45627	-22.71516	-6.092	34	.000
Pair 4	LSICG128 - LSI128	-34.68571	34.40398	5.81533	-46.50389	-22.86753	-5.965	34	.000
Pair 5	LSICG201 - LSI201	-52.05714	50.82088	8.59030	-69.51472	-34.59956	-6.060	34	.000
Pair 6	BestLSICG - BestLSI	-17.00000	21.76479	3.67892	-24.47647	-9.52353	-4.621	34	.000

For LSICG16 and LSI16, |t| = 5.608, t*, which is the critical value of t, is between 1.697 and 1.684, p = 0.000 and alpha = 0.05, |t|>t*, so reject H0. There is significant evidence to indicate that H1.

We can have the same decision on the pairs of LSI32 and LSICG32, LSI64 and LSICG64, LSI128 and LSICG128, LSI201 and LSICG201, BestLSI and BestLSICG, because their |t| values, which are 4.822, 6.092, 5.965, 6.060 and 4.621 respectively, are all greater than t*, with p = 0.000 and alpha = 0.05. So for LSICG and LSI on every dimension including the best of LSICG and the best of LSI, there is significant evidence to indicate that LSICG is more accurate than LSI for Rhino.

5.8.3 jEdit

In jEdit, we perform the tasks of feature location and bug localization. For the feature location task in jEdit, we use the same 3 features from the previous study. For each feature with four different queries, we run LSI at 16, 32, 64, 128 and 256 dimensions and then run LSICG at the same dimension, but at this time both techniques are performed at class level. To evaluate the performances of LSI and LSICG, we compared the results from BestLSI and BestLSICG. BestLSI and BestLSICG are defined as the best result returned for each feature, regardless of which query and which dimension it is from. Table 5.40 provides the rankings of them as well as the ranking of LSI results from every dimension.

Table 5.40 Rankings for features in jEdit (class level)

Feature	Queries	LSI16	LSI32	LSI64	LSI128	LSI256	BestLSI	BestLSICG
Search	Q11	626	60	217	484	349	5	2
	Q12	677	417	290	5	287		
	Q13	677	417	290	5	287		
	Q14	438	515	494	224	101		
Add Marker	Q21	592	281	428	642	510	119	9
	Q22	267	119	455	616	368		
	Q23	572	382	586	360	264		
	Q24	479	419	375	345	283		
Show White Space	Q31	584	576	511	578	516	3	3
	Q32	594	512	269	574	679		
	Q33	439	380	314	3	648		
	Q34	364	22	79	85	759		

Figure 5.20 provides the average rankings for each feature by BestLSI and BestLSICG. The data indicate BestLSICG has better average rankings than BestLSI. More specifically, BestLSICG's average of 4.67 is more than 37 positions better than BestLSI at an average ranking of 42.33. BestLSICG gives 88.98% improvement in the rankings over BestLSI.

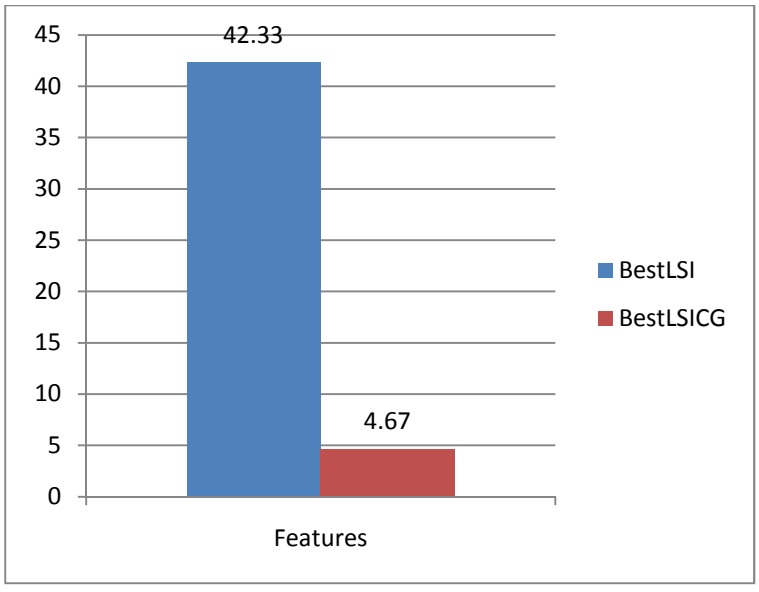


Figure 5.20 Average rankings of BestLSI and BestLSICG for features in jEdit (class level)

For bug localization task in jEdit, it includes locating the correct class based on a given bug description. We still use six bugs and four different queries for each one in previous case study. For each bug and each query, we run LSI at 16, 32, 64, 128 and 256 dimensions as well as LSICG at the same dimension. By comparing the results from BestLSI and BestLSICG, we evaluate the two techniques' performances. Table 5.41 provides the rankings of relevant class using LSI for each bug under each dimension and each query. It also lists the rankings from BestLSI and BestLSICG.

Table 5.41 Rankings of bugs in jEdit (class level)

Bug No.	Queries	LSI16	LSI32	LSI64	LSI128	LSI256	BestLSI	BestLSICG
1275607	Q11	465	703	571	665	388	94	63
	Q12	259	94	643	399	367		
	Q13	381	515	758	612	274		
	Q14	612	517	715	424	540		
1467311	Q21	689	687	736	394	323	152	1
	Q22	497	573	736	261	478		
	Q23	604	570	689	152	427		
	Q24	532	587	728	194	511		
1469996	Q31	53	343	161	127	286	16	6
	Q32	52	351	147	119	16		
	Q33	351	255	381	100	339		
	Q34	284	386	593	487	668		
1488060	Q41	663	282	88	335	682	25	34
	Q42	612	309	85	90	546		
	Q43	534	102	25	91	528		
	Q44	556	139	121	55	486		
1607211	Q51	646	696	749	733	676	84	53
	Q52	529	474	339	209	283		
	Q53	685	525	259	136	657		
	Q54	580	517	186	84	731		
1642574	Q61	244	283	192	375	370	79	78
	Q62	372	79	307	468	485		
	Q63	646	237	159	343	375		
	Q64	431	267	326	314	391		

Figure 5.21 shows the average rankings of BestLSI and BestLSICG for all six bugs. Here we also choose top 3 bugs and list BestLSI and BestLSICG for them. The average ranking of BestLSI for all six bugs is 39.17, which is almost 36 positions higher than BestLSI (75). This improvement can reach to 47.77%. For the top 3 bugs, the average ranking of BestLSICG obtains the best performance of 13.67, which is more than 26 positions higher than BestLSI (40). That is a 65.83% improvement.

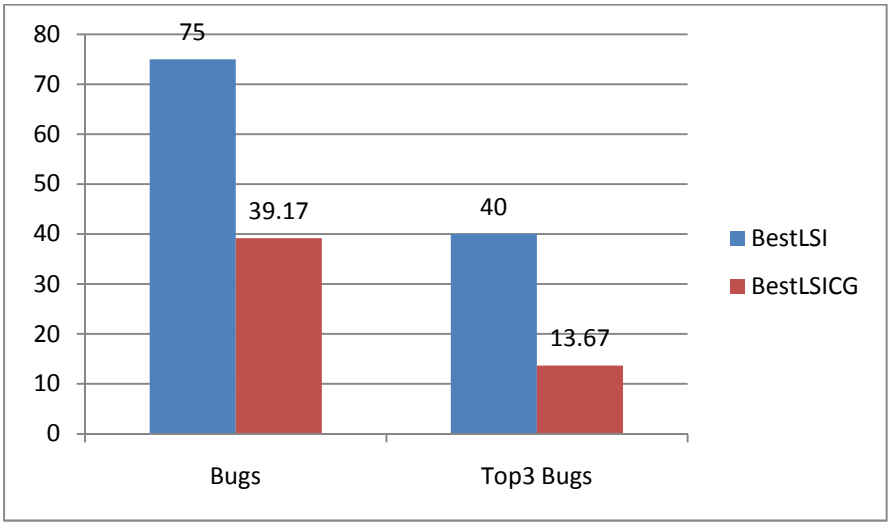


Figure 5.21 Average Rankings of BestLSI and BestLSICG for bugs in jEdit (class level)

To provide the statistical evidence that LSICG improves accuracy over LSI for feature location and bug localization in jEdit, we combine the results from 3 features and 6 bugs. Figure 5.22 gave the average rankings of BestLSI and BestLSICG. BestLSICG obtains an average ranking of 28, while BestLSI has the average ranking of 64. The results from BestLSICG are average 36 positions higher than those from BestLSI. This improvement achieves 56.25%.

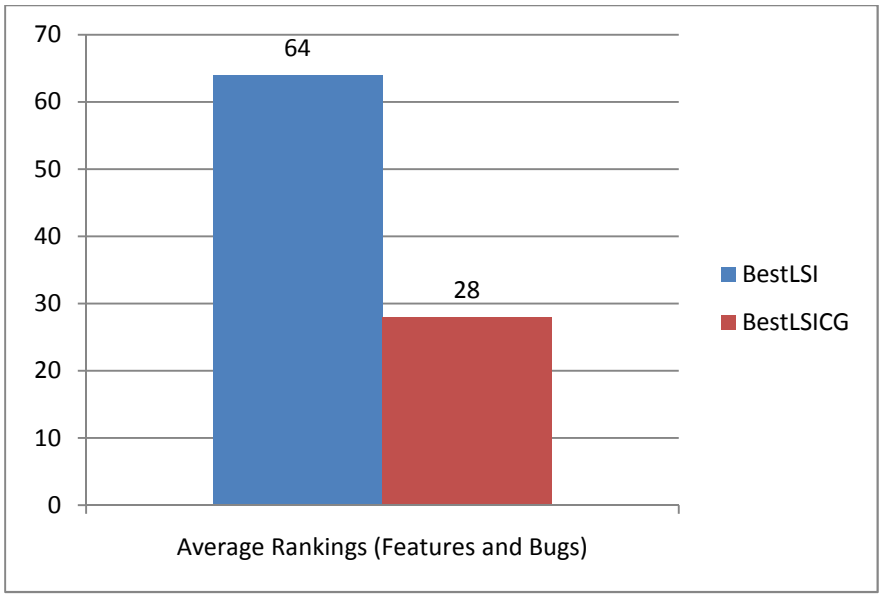


Figure 5.22 Average rankings of BestLSI and BestLSICG for jEdit (class level)

To determine whether this improvement of accuracy from BestLSICG over BestLSI is statistically significant, we use Wilcoxon's signed-rank test to test the case study hypothesis The result of Wilcoxon's test is W=3, and the critical value W*= 4, $p < 0.05$. W<W*, so we successfully reject the null hypothesis and accept the alternative hypothesis. There is significant evidence to indicate that LSICG is more accurate than LSI for feature location and bug localization in jEdit.

5.8.4 Discussion of results

The third experiment compares the performance of LSICG and LSI for concept location over the same three Java projects as we did in the first and second experiments. But in this experiment, all testing are performed at class level. All identified features and bugs from the three projects are tested in the concept location tasks including feature location and bug localization. LSICG provides a substantial increase in average rankings compared to LSI alone. The improvement of BestLSICG in rankings over BestLSI for JavaHMO, Rhino and jEdit are 36.36%, 73.91%, and 56.25% respectively. The average ranking increase of 55.51% across the three case studies

occurs when using the optimal threshold value of an LSI score of 0.0. For JavaHMO, Rhino, and jEdit, we also provided Wilcoxon's signed-rank test and Paired-samples t test to prove that LSICG's improvement is statistically significant. The statistical results allow us to reject the $\mathbf{H_0}$ hypothesis of LSICG not being as accurate as LSI alone and accept the alternate hypothesis, $\mathbf{H_1}$, that LSICG is more accurate than LSI for concept location.

In the JavaHMO case study, the results in Table 5.36 indicate that LSICG is more accurate than LSI alone for 80% (20) of 25 features. LSICG performs the same to LSI for 20% (5) of the 25 features. In the JavaHMO case study, LSICG did not provide less accurate result than LSI for any feature.

In the Rhino case study, the results in Table 5.37 and Table 5.38 indicate that LSICG is more accurate than LSI alone for 74% (26) of 35 bugs. LSICG performs the same to LSI for 20% (7) of the 35 bugs. However, LSICG is less accurate than LSI for 6% (2) of the 35 bugs.

In the jEdit case study, the results in Table 5.40 indicate that LSICG is more accurate than LSI alone for 78% (7) of 3 features and 6 bugs. LSICG performs the same to LSI for 11% (1) of the 3 features and 6 bugs, and LSICG is less accurate than LSI for 11% (1) of the 3 features and 6 bugs.

Overall, for the three projects JavaHMO, Rhino and jEdit, LSICG is more accurate than LSI alone for 77% (53) of the 69 features and bugs. LSICG performs the same to LSI for 19% (13) of the 69 features and bugs, and LSICG is less accurate than LSI for 4% (3) of the 69 features and bugs. The overall performance of LSICG is promising. It indicates that LSICG outperforms or at least performs the same with LSI alone on 96% of all identified bugs and features in these case studies.

CHAPTER 6

DISCUSSION

The results of this research indicate combing structural information and semantic information provides a significant improvement in concept localization tasks when compared to a strictly semantic approach. This is not an either/or situation. The structural information is complementary to the semantic information. Specifically for this research, LSICG provides a significant improvement over LSI alone. LSICG is a complement to LSI because LSICG contains both lexical information (terms, identifiers, etcetera) and structural information (method and class level call graphs) from source code. Considering structural information from methods and classes allows LSICG to augment LSI improving overall performance. In particular, the structural information in this research, call graph nodes for a targeted method or class, essentially examines a targeted node and its nearest neighbor. If a method or class has a high ranking in LSI, i.e. thought to be relevant to a given bug or feature, LSICG further checks whether its neighbors are relevant to this bug or feature. The more neighbors that are relevant, the more weight added that method or class for the particular bug or feature. There are many different types of "neighbors" that can be defined beyond call graph. These neighbors can be found using class dependency diagrams, control flow graphs, and many other structural representations.

Beyond the significant improvements in concept location, ancillary findings from this research include:

1. LSICG, as an augment of LSI, is dominated by LSI. So a good result from LSI is necessary as the first step to obtain a good result from LSICG.
2. The content of the query is critical to obtain a good result from LSI. A well-defined query allows LSI to generate high rankings for targeted methods and classes. This is also the case for other IR modules. Lukins et al. (2008; 2010) and Liu et al. 2007 both polish their queries to improve result for their techniques.
3. If terms in a query are common and have a large number of occurrences in the corpora, the rankings of the targeted methods and classes are low. To gain higher rankings when constructing a query, it is better to use different, relevant and unique terms for a given concept. Simply increasing the occurrence number of a term in a query does not impact the rankings.
4. Comments do not have significant influence to the rankings. When building the corpora for a project, comments except copyright information are included. When constructing documents, only comments within a method or a class are considered. Compared with the entire corpora, comments are negligible and most of them are common terms. As indicated above, comment terms have insignificant influence on the rankings. This research does not exhibit a major difference of performance from LSI and LSICG if comments are excluded.
5. Method and class coupling impact the results of LSICG. In this research the connectivity among methods and classes is built by extracting their call graph nodes. Methods and classes that have a relatively small or zero number of call graph nodes do not benefit from the LSICG approach. As an example, in this research Rhino has

more coupling than the other two projects. For Rhino, LSICG achieved the most improvement (73.91%) over LSI at the class level.

CHAPTER 7

THREATS TO VALIDITY

Our case study has limitations that impact the validity of our findings, as well as our ability to generalize them. Threats to internal validity relate to our implementations of LSI and LSICG, and threats to external validity relate to our use of the projects in this study.

Threats to the internal validity of our case study that relate to our implementation of LSI include:

1. Query construction affects the LSI results (L_{LSI}). We used bug queries and feature searches were originally formulated in the literature to evaluate LSI or other IR techniques.

Threats to the internal validity of our case study that relate to the implementation of LSICG include:

2. Selection of the minimum similarity score threshold, the amount of structural information to extract, and the value of λ affect the LSICG results. We used a minimum threshold of 0, extracted one-edge-away methods from the program call graph, and set λ to 0.5. We did not experiment with different values.

Threats to the external validity of our case study that relate to projects used in this study bugs include:

3. All projects in this study are developed in Java. LSICG may perform differently with projects in other languages.

4. The projects in this study range from 1,787 methods to 5,534 methods. LSICG may perform differently for larger or smaller projects.
5. The set of bugs and features used in this study were taken from the literature and may not be representative of all bugs and features for these projects (or of other projects).
6. Purvis (2000) reports that CASE technology is underutilized. A lack of adoption of tools for bug localization would limit the impact of our approach.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

We have described a static bug localization and feature location technique, LSICG, that combines LSI and call graph information to provide improved performance over LSI alone. The performance improvement is verified by a case study in which we apply LSICG up to 25 features in JavaHMO, 35 bugs in Rhino, 3 features and 6 bugs in jEdit, at both method level and class level of granularity. At the method level, LSICG outperformed LSI by 24.89% for the 25 features in JavaHMO, 13.94% for the 35 bugs in Rhino, 15.58% for the 3 features and 6 bugs in jEdit. Overall, compared to LSI, LSICG obtained better or equal performance on 94% of features and bugs. At the class level, LSICG outperformed LSI by 36.36% for the 25 features in JavaHMO, 73.91% for the 35 bugs in Rhino, 56.25% for the 3 features and 6 bugs in jEdit. Overall, compared to LSI, LSICG obtained better or equal performance on 96% of features and bugs. The result of the improved performance is a potentially large reduction in a developer's effort when attempting to identify the methods or the classes in a program's source code that must be modified to correct a bug or implement a feature.

Corresponding to the threats to validity above, future work includes:

1. Using additional queries for each concept to limit the influence of a poorly formulated query. As discussed by Lukins et al. (2010), slight changes to a query can dramatically impact the resulting rankings.
2. Experimenting with the kind and amount of structural information used, as well as its weighting (i.e., the value of λ). For example, in addition to relaxing the one-

call-away rule, try combining structural program representations other than the call graph. Candidates include the class dependency diagram and the control flow graph.

3. Apply LSICG to a variety of projects in other development languages.
4. Applying LSICG to additional projects, including large projects such as Eclipse and Mozilla.

Other future work includes investigating the use of IR techniques other than LSI, for example, LDA or the vector space model (VSM).

REFERENCES

Alkhatib, G. (1992) 'The maintenance problem of application software: an empirical analysis,' *Journal of Software Maintenance: Research and Practice*, Vol. 4, No. 2, pp. 83-104.

Antoniol, G. and Canfora, G. and de Lucia, A. and Casazza (2000a), G., Information Retrieval Models for Recovering Traceability Links between Code and Documentation, *ICSM '00: Proceedings of the International Conference on Software Maintenance*, Washington, DC, USA, pp. 40.

Antoniol, G., Canfora, G., Casazza, G. and De Lucia, A. (2000) 'Identifying the starting impact set of a maintenance request: a case study,' *Proceedings of the 4th European Conference on Software Maintenance and Reengineering (CSMR)*, Zurich, Switzerland, February, pp. 227-230.

Antoniol, G.; Canfora, G.; Casazza, G.; De Lucia, A.; Merlo, E. (2002), Recovering traceability links between code and documentation Software Engineering, *IEEE Transactions*, Vol. 28, pp. 970 – 983.

Arnold, R.S. and Bohner, S.A. (1993), "Impact Analysis—Towards a Framework for Comparison", *Proc. Int'l Conf. Software Maintenance*, pp. 292–301.

Basili, V.R. and Weiss, D. M. (1984) 'A methodology for collecting valid software engineering data,' *IEEE Transactions on Software Engineering*, Vol. 10, No. 6, November, pp. 728-738.

Baysal, O., Malton. A. (2007), Correlating Social Interactions to Release History During Software Evolution, *Fourth International Workshop on Mining Software Repositories (MSR'07)*, Vol. 8.

Biggerstaff, T.J., Mitbander, B.G. and Webster, D. (1993) 'The concept assignment problem in program understanding', *Proceedings of the 15th International Conference on Software Engineering (ICSE)*, Baltimore, MD, USA, pp. 482-498.

Blei, D.M., Ng, A.Y. and Jordan, M.I. (2003) 'Latent Dirichlet allocation,' *Journal of Machine Learning Research*, Vol. 3, January, pp. 993-1022.

Boehm, B.W. (1981) *Software Engineering Economics*, Prentice Hall.

Canfora, G. and Cerulo, L. (2006a), Supporting Change Request Assignment in Open Source Development, *Proceedings of the 2006 ACM symposium on Applied computing*, Vol, 6, pp. 1767-1772.

Canfora, G. and Cerulo, L. (2006), Fine Grained Indexing of Software Repositories to Support Impact Analysis, *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, Vol. 7, pp. 105-111.

Canfora, G., Cerulo, L., Penta, M.D. (2007), Identifying Changed Source Code Lines from Version Repositories, *Fourth International Workshop on Mining Software Repositories (MSR'07)*, Vol 8.

Chang, H.F. and Mockus, A. (2008), Evaluation of Source Code Copy Detection Methods on FreeBSD, Proceedings of the 2008 international working conference on Mining Software Repositories, pp. 5.

Cleary, B., Exton, C., Buckley, J. and English, M. (2009) 'An empirical analysis of information retrieval based concept location techniques in software comprehension,' *Empirical Software Engineering*, Vol. 14, No. 1, pp. 93-130.

Cleland-Huang, J. and Settimi, R. and BenKhadra, O. and Berezhanskaya, E. and Christina, S. (2005), Goal-centric traceability for managing non-functional requirements, *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pp. 362-371.

Cleland-Huang, J. and Settimi, R. and Zou, X. and Solc, P. (2007), Automated classification of non-functional requirements, *Requir. Eng.*, Vol. 12, number 2, pp. 103-120.

Crestani, F. and Van Rijsbergen, C. J. (1995), Information retrieval by logical imaging, *Journal of Documentation*, Vol. 51, pp. 3–17.

Crestani, F. and van Rijsbergen, C. J. (1998), A study of probability kinematics in information retrieval, *ACM Trans. Inf. Syst.*, pp. 225-255.

David, J. (2008), Recommending Software Artifacts from Repository Transactions, *IEA/AIE '08: Proceedings of the 21st international conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, Vol. 15, number 2, pp. 189-198.

Deerwester, S.C., Dumais, S.T., Landauer, T.K., Furnas, F.W. and Harshman, R.A. (1990) 'Indexing by latent semantic analysis,' *Journal of the American Society for Information Science*, Vol. 41, No. 6, pp. 391-407.

De Lucia, A. and Fasano, F. and Oliveto, R. and Tortora, G. (2004), Enhancing an Artifact Management System with Traceability Recovery Features, *Automated Software Engg.*, pp. 306-315.

De Lucia, A. and Fasano, F. and Oliveto, R. and Tortora, G. (2005), ADAMS Re-Trace: A Traceability Recovery Tool, *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pp. 32-41.

De Lucia, A. and Fasano, F. and Oliveto, R. and Tortora, G. and Massimiliano, D.P. (2006), Improving Comprehensibility of Source Code via Traceability Information: a Controlled Experiment, *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pp. 317-326

De Lucia, A. and Oliveto, R. and Sgueglia, P. (2006a), Incremental Approach and User Feedbacks: a Silver Bullet for Traceability Recovery, *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pp. 299-309.

De Lucia, A. and Fasano, F. and Oliveto, R. and Tortora, G. (2007), Recovering traceability links in software artifact management systems using information retrieval methods, *ACM Trans. Softw. Eng. Methodol.*, Vol. 16, number 4, pp. 13.

De Lucia, A. and Scanniello, G. and Tortora, G. (2007a), Identifying similar pages in Web applications using a competitive clustering algorithm: Special Issue Articles, *J. Softw. Maint. Evol.*, Vol. 19, number 5, pp. 281-296

De Lucia, A. and Risi. M. and Scanniello, G. and Tortora, G. (2007b), Comparing Clustering Algorithms for the Identification of Similar Pages in Web Applications, *ICWE*, pp. 415-420

De Lucia, A. and Fasano, F. and Oliveto, R. and Tortora, G. (2008), Adams re-trace: traceability link recovery via latent semantic indexing, *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pp. 839-842.

Eaddy, M., Aho, A.V., Antoniol, G. and Gueheneuc, Y. (2008) 'CERBERUS: Tracing Requirements to Source Code using Information Retrieval, Dynamic Analysis, and Program Analysis,' *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC)*, Amsterdam, The Netherlands, June, pp. 53-62.

Erlikh, L. (2000) 'Leveraging Legacy System Dollars for E-Business,' *IEEE IT Pro*, May/June, pp. 17-23.

Etzkorn, L., Davis, C. (1996), Automated object-oriented reusable component identification, *Knowl.-Based Syst.*, Vol. 9, number 8, pp. 517-524.

Etzkorn, L.H. and Hughes, W.E.Jr. and Davis, C.G. (2001), Automated reusability quality analysis of OO legacy software, *Information Software Technology*, Vol. 43, number 5, pp 295-308.

Frakes, W.B. and Baeza-Yates, R. (1992), "Information Retrieval: Data Structures and Algorithms", Englewood Cliffs, N.J.: Prentice-Hall.

Fry, Z. P. and Shepherd, D. and Hill, E. and Pollock, L. and Vijay-Shanker, K. (2008), Analyzing source code: looking for useful verb–direct object pairs in all the right places, *IET Software*, Vol. 2, pp. 27-36.

Fuhr, N (1989), "Models for Retrieval with Probabilistic Indexing", *Information Processing and Management*. Vol. 25, No.1.

Fyson, M.J. and Boldyreff, C. (1998), Using Application Understanding to Support Impact Analysis, *J. Software Maintenance—Research and Practice*, Vol. 10, pp. 93–110.

Gay, G., Haiduc, S., Marcus, A. and Menzies, T. (2009) 'On the use of relevance feedback in IR-based concept location,' *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM)*, Edmonton, Alberta, Canada, September, pp. 351-360.

Golub, G. and Kahan, W. (1965), Calculating the Singular Values and Pseudo-Inverse of a Matrix, *Journal of the Society for Industrial and Applied Mathematics: Series B, Numerical Analysis*, Vol. 2, No. 2, pp. 205-224.

Gross, H.G. and Lormans, M. and Zhou, J. (2007), Towards Software Component Procurement Automation with Latent Semantic Analysis, Electron. *Notes Theor. Comput. Sci*., Vol. 189, pp. 51—68.

Grossman, D.A. and Frieder, O. (2004) *Information Retrieval: Algorithms and Heuristics*, Springer, 2nd edition, Vol. 333.

Grove, D., DeFouw, G., Dean, J. and Chambers, C. (1997) 'Call Graph Construction in Object-Oriented Languages,' *SIGPLAN Notices*, Vol. 32, No. 10, *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA)*, Atlanta, GA, USA, October, pp. 108-124.

Gui, G. and Scott, P.D. (2005), Vector Space Based on Hierarchical Weighting: A Component Ranking Approach to Component Retrieval, *APPT 2005*, pp. 184-193.

Haruechaiyasak, C., Shyu, M. and Chen, S. (2006) 'A web-page recommender system via a data mining framework and the Semantic Web concept,' *International Journal of Computer Applications in Technology*, Vol. 27, No. 4. doi:10.1504/IJCAT.2006.012000.

Hayes, J.H. and Dekhtyar, A. and Sundaram, S.K. and Holbrook, E.A. and Vadlamudi, S. and April, A. (2007), REquirements TRacing On target (RETRO): improving software maintenance through traceability recovery, *ISSE*, Vol. 3, number 3, pp. 193-202.

Hill, E., Pollock, L., and Vijay-Shanker, K. (2007), Exploring the Neighborhood with Dora to Expedite Software Maintenance, *International Conference of Automated Software Engineering (ASE '07)*, Vol. 10, pp. 14-23.

Hill, E., Fry, Z.P., Boyd, H., Sridhara, G., Novikova, Y., Pollock, L., and Vijay-Shanker, K. (2008), AMAP: automatically mining abbreviation expansions in programs to enhance software maintenance tools, *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, Vol. 10, pp. 79-88.

Hill, E., Pollock, L., and Vijay-Shanker, K. (2009), Automatically capturing source code context of NL-queries for software maintenance and reuse, *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, Vol. 11, pp. 232-242.

Hindle, A., Godfrey M. W., and Richard C. Holt, R. C. (2009), What Hot and What Not: Windowed Developer Topic Analysis, *Proc. of 2009 IEEE Conference on Software Maintenance (ICSM-09)*, Vol. 10.

Jiang, H., Nguyen, T.N., Chen, I.X., Jaygarl, H., Carl K. Chang, C.K. (2008), Incremental Latent Semantic Indexing for Automatic Traceability Link Evolution Management, *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, Vol. 10, pp. 59-68.

Kanellopoulos, Y. and Makris, C. and Tjortjis, C. (2007), An improved methodology on information distillation by mining program source code, *Data Knowl. Eng.*, Vol. 61, number 2, pp. 359-383.

Kawaguchi, S. and Garg, P.K. and Matsushita, M. and Inoue, K. (2003), Automatic Categorization Algorithm for Evolvable Software Archive, *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, Vol. 61, number 2, pp. 195.

Kawaguchi, S. and Garg, P.K. and Matsushita, M. and Inoue, K. (2006), MUDABlue: an automatic categorization system for open source repositories, *Software Engineering Conference, 2004. 11th Asia-Pacific*, pp. 184-193.

Kim, S., James, W. E. and Zhang, Y. (2008), Classifying Software Changes: Clean or Buggy?, *IEEE Trans. Softw. Eng.*, pp. 181-196.

Konclin, J. and Bergen, M. (1988), "Gibis: A Hypertext Tool for Exploratory Policy Discussion", *ACM Trans. Office Information Systems.*

Kuhn, A., Ducasse, S., Gîrba, T. (2007), Semantic clustering: Identifying topics in source code, *Inf. Softw. Technol.*, Vol. 49, number 3, pp. 230-243.

Lawrie, D.J. and Feild, H. and Binkley, D. (2006), Leveraged Quality Assessment using Information Retrieval Techniques, *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, Vol. 10, pp. 149-158.

Lewis, D., In Ifs, Harper, W.L., Stalnaker, R., and Pearce, G., Eds. (1981),"Probability of conditionals and conditionals probabilities", University of Western Ontario, *Series in Philosophy of Science*, D. Reidel Publishing Co., Inc., New York, NY, pp. 129–147.

Li, Y., Li, J., Yang, Y., and Li, M. (2008), Requirement-Centric Traceability for Change Impact Analysis: A Case Study, *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pp. 100-111.

Lindvall, M. and Feldmann, R.L. and Karabatis, G. and Chen, Z. and Janeja, V. P. (2009), Searching for relevant software change artifacts using semantic networks, *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pp. 496-500.

Liu, D., Marcus, A., Poshyvanyk, D. and Rajlich, V. (2007) 'Feature Location via Information Retrieval Based Filtering of a Single Scenario Execution Trace,' *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Atlanta, GA, USA, November, pp. 234-243.

Liu, H. and Lethbridge, T. (2001), Intelligent search techniques for large software systems, *CASCON '01: Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative research*, Vol. 10.

Liu, H. and Lethbridge, T. (2002), Intelligent search techniques for large software systems, *Information Systems Frontiers*, pp. 409-423.

Lormans, M. and Deursen, A.V. (2005), Reconstructing requirements coverage views from design and test using traceability recovery via LSI, *TEFSE '05: Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*, pp. 37-42.

Lormans, M. and Deursen, A.V. (2006), Can LSI help Reconstructing Requirements Traceability in Design and Test?, *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pp. 47-56.

Lormans, M. and Deursen, A.V. (2007), Reconstructing Requirements Traceability in Design and Test Using Latent Semantic Indexing, *Software Engineering Research Group Technical Reports.*

Lormans, M. and Deursen, A.V., Gross, H.G. (2008a), An industrial case study in reconstructing requirements views, *Empirical Softw. Engg.*, Vol. 13, number 6, pp. 727-760.

Lormans, M. and Deursen, A.V. (2008), Reconstructing Requirements Traceability in Design and Test Using Latent Semantic Indexing, *Software Engineering Research Group Technical Reports.*

Lukins, S.K., Kraft, N.A. and Etzkorn, L.H. (2008) 'Source code retrieval for bug localization using latent Dirichlet allocation,' *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE)*, Antwerp, Belgium, October, pp. 155-164.

Lukins, S.K., Kraft, N.A. and Etzkorn, L.H. (2010) 'Bug localization using latent Dirichlet allocation,' *Information and Software Technology*, Vol. 52, No. 9, September, pp. 972-990.

Lu, W. and Kan, M.Y. (2005), Supervised categorization of JavaScriptTM using program analysis features, *AIRS 2005*, pp. 160-173

Lu, W. and Kan, M.Y., (2007), Supervised categorization of JavaScriptTM using program analysis features, *Inf. Process. Manage.*, Vol. 43, number 2, pp. 431-444

Maeder, P., Riebisch, M. and Philippow I. (2006) 'Traceability for Managing Evolutionary Change,' *Proceedings of the 15th International Conference on Software Engineering and Data Engineering (SEDE)*, Los Angeles, CA, USA, July, pp. 1-8.

Maia, M.A., Sobreira, V., Paixão, K.R., Amo, S.A. and Silva, I.R. (2008) 'Using a sequence alignment algorithm to identify specific and common code from execution traces,' *Proceedings of the 4th International Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, Antwerp, Belgium, October, pp. 6-10.

Maletic, J.I., Marcus, A. (2000), Support for software maintenance using latent semantic indexing, *Software Engineering Application (SEA)*, 2000.

Maletic, J.I., Marcus, A. (2001), Supporting program comprehension using semantic and structural information, *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pp. 103-112.

Marcus, A. and Maletic, J. (2001) 'Identification of high-level concept clones in source code,' *Proceedings of the 16th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, San Diego, CA, USA, November, pp. 107-114.

Marcus, A. and Maletic, J. (2003), Recovering documentation-to-source-code traceability links using latent semantic indexing, *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 125-135.

Marcus, A., Sergeyev, A., Rajlich, V. and Maletic, J. (2004) 'An Information Retrieval Approach to Concept Location in Source Code,' *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE)*, Delft, The Netherlands, November, pp. 214-223.

Marcus, A., Rajlich, V., Buchta, J., Petrenko, M. and Sergeyev, A. (2005) 'Static techniques for concept location in object-oriented code,' *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC)*, St. Louis, MO, USA, May, pp. 33-42.

Marcus, A. and Poshyvanyk, D. and Ferenc, R. (2008), Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems, *IEEE Trans. Softw. Eng*., Vol. 34, number 2, pp. 287-300.

Maskeri, G. and Sarkar, S. and Heafield, K. (2008), Mining business topics in source code using latent dirichlet allocation, *ISEC '08: Proceedings of the 1st conference on India software engineering conference*, pages 113-120

McCarey, F., Cinnéide, M, and Nicholas Kushmerick, N. (2006), Recommending Library Methods: An Evaluation of the Vector Space Model (VSM) and Latent Semantic Indexing (LSI), *8th International Conference on Software Reuse* , Vol. 4039/2006, pp. 217—230

Müller, H.A., Jahnke, J.H., Smith, D.B., Storey, M.A., Tilley, S.R. and Wong, K. (2000) 'Reverse engineering: a roadmap,' *Proceedings of the Conference on The Future of Software Engineering*, Limerick, Ireland, June, pp. 47-60.

Murphy, G.C., Notkin, D., Griswold, W.G. and Lan, E.S. (1998) 'An Empirical Study of Static Call Graph Extractors,' *ACM Transactions on Software Engineering and Methodology*, Vol. 7, No. 2, April, pp. 158-191.

Purvis, R. (2000) 'The extent of CASE technology use within systems development projects,' *International Journal of Computer Applications in Technology*, Vol. 13, No. 3/4/5. doi:10.1504/IJCAT.2000.000234.

Pinhero and Goguen, J.A. (1996), "An Object-Oriented Tool for Tracing Requirements", *IEEE Software*.

Ponte, J.M. and Bruce Croft, W.B. (1998), A Language Modeling approach for information retrieval, *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 275-81.

Porter, M.F. (1980) 'An algorithm for suffix stripping,' *Program*, Vol. 14, No. 3, pp. 130-137.

Poshyvanyk, D. and Marcus, A. (2007) 'Combining formal concept analysis with information retrieval for concept location in source code,' *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC)*, Banff, Alberta, Canada, June, pp. 37-48.

Poshyvanyk, D., Guegeneuc, Y., Marcus, A., Antoniol, G. and Rajlich, V. (2007) 'Feature Location using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval,' *IEEE Transactions on Software Engineering*, Vol. 33, No. 6, June, pp. 420-432.

Rajlich, V. and Wilde, N. (2002), The Role of Concepts in Program Comprehension, *Proc. 10th IEEE Int'l Workshop Program Comprehension (IWPC '02)*, pp. 271-278.

Ramesh, B. and Dhar, V. (1992), "Supporting Systems Development Using Knowledge Captured During Requirements Engineering", *IEEE Trans. Software Eng*.

Robertson, S.E. and Jones, S.K. (1977). "Relevance Weighting Of Search Terms", *Journal of the American Society for Information Science*, Vol. 27.

Salton, G. and Buckley, C. (1988) 'Term-weighting approaches in automatic text retrieval,' *Information Processing & Management*, Vol. 24, No. 5, pp. 513-523.

Shao, P., Kraft, N.A. and Smith, R.K. (2009) 'Combining Latent Semantic Indexing and Call Graphs to Improve Feature Location,' *Proceedings of the 13th Annual IASTED International Conference on Software Engineering and Applications (SEA)*, Cambridge, MA, USA, November, pp. 49-54.

Shepherd, D. and Fry, Z.P. and Hill, E. and Pollock, L. and Vijay-Shanker, K., Using natural language program analysis to locate and understand action-oriented concerns, *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pp. 212-224.

Standish, T.A. (1984) 'An essay on software reuse,' *IEEE Transactions on Software Engineering*, Vol. 10, No. 5, pp. 494-497.

Tan Zhenmin, Lin Wang, Xuanhui Lu, Shan Zhou, Yuanyuan Zhai, Cheng xiang (2006), Have things changed now?: an empirical study of bug characteristics in modern open source, *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pp.25-33.

Turtle H. and W. B. Croft (1991), "Efficient Probabilistic Inference for Text Retrieval", *Proceedings of RIAO 3*.

Turver, R.J. and Munro, M. (1994), "An Early Impact Analysis Technique for Software Maintenance", *J. Software Maintenance—Research and Practice*, Vol. 6, no. 1, pp. 35–52.

Wong, S. K. M. and Y. Y. Yao (1989), "A Probability Distribution Model for Information Retrieval", *Information Processing and Management*, Vol. 25, No. 1, pp. 39-53.

Xu, X.W. and Song, L. (2006) 'Development of an integrated reverse engineering system,' *International Journal of Computer Applications in Technology*, Vol. 25, No. 1. doi:10.1504/IJCAT.2006.008639.

Zhao, W., Zhang, L., Liu, Y., Sun, J. and Yang, F. (2006) 'SNIAFL: Towards a static noninteractive approach to feature location,' *ACM Transactions on Software Engineering and Methodology*, Vol. 15, No. 2, April, pp. 195-226.

Websites

JavaHMO, http://JavaHMO.sourceforge.net

Rhino, http://www.mozilla.org/rhino

Bugzilla, http://www.bugzilla.org

jEdit, http://www.jedit.org

APPENDIX A

Table A.1 Thirty-five bugs and targeted methods in Rhino

Bug Number	**Bug title [query words added from bug description]** Targeted (*Class, Method*)
256836	**Dynamic** scope and nested **functions** *(Context, hasCompileFunctionsWithDynamicScope)*
274996	Exceptions with multiple **interpreters** on **stack** may lead to **ArrayIndex**OutOf**BoundsException** [**java wrapped**] *(Interpreter, interpret)*
256865	Compatibility with **gcj**: changing **ByteCode**.<**constants**> to be int *(ClassFileWriter, addInvoke)*
257423	**Optimizer** regression: **this**.**name** += expression generates wrong **code** *(Codegen, visitSetProp)*
238699	**Context.compileFunction** throws **InstantiationException** *(Codegen, compile)*
256621	**throw statement**: **eol** should not be allowed *(Parser, statementHelper)*
249471	**String index out** of range **exception [parse bound float global js native char]** *(NativeGlobal, encode)*
239068	Scope of **constructor functions** is not **initialized** *(IdScriptable, addAsPrototype)*
238823	**Context.compileFunction** throws **Null**Pointer **exception** *(Context, compile)*
258958	**Lookup** of excluded **objects** in **ScriptableOutputStream** doesn't traverse **prototypes**/**parents** *(ScriptableOutputStream, lookupQualifiedName)*
58118	ECMA Compliance: **daylight savings time** wrong prior to **year** 1 [**day**

	offset timezone] *(NativeDate, date_format)*
256389	Proper **CompilerEnvirons**.is**Xml**Available() *(CompilerEnvirons, isGeneratingSource)*
258207	**Exception name** should be **DontDelete [delete catch ecma obj object script]** *(ScriptRuntime, newCatchScope)*
252122	**Double** expansion of **error message** *(ScriptRuntime, undefWriteError)*
262447	NullPointerException in **ScriptableObject**.**getPropertyIds** *(ScriptableObject, getPropertyIds)*
258419	**copy paste** bug in org.mozilla.javascript.**regexp**.**NativeRegExp [RE data back stack state track]** *(NativeRegExp, matchRegExp)*
266418	Can not **serialize reg**ular **exp**ressions **[regexp RE compile char set]** *(NativeRegExp, emitREBytecode)*
263978	cannot **run** xalan example with Rhino 1.5 release 5 **[line number negative execute error]** *(Context, compileString)*
254915	Broken "**this**" for **name**() calls (CVS tip regression) **[object with**] *(ScriptRuntime, getBase)*
255549	**JVM-**dependent resolution of **ambiguity** when **calling Java methods [argument constructor overload]** *(NativeJavaMethod, findFunction)*
253323	Assignment to variable '**decompiler**' has no effect in **Parser** [**parse**] *(Parser, parse)*
244014	Removal of **code complexity limits** in the **interpreter** *(Interpreter,interpret* *Interpreter,generateICode* *Interpreter,do_nameAndThis)*
244492	**JavaScript**Exception to extend **Runtime**Exception and common **exception** base *(EcmaError,EcmaError* *EcmaError,getColumnNumber*

	EcmaError,getLineNumber *EcmaError,getLineSource* *EcmaError,getSourceName* *EvaluatorException,EvaluatorException* *EvaluatorException,generateErrorMessage* *EvaluatorException,getColumnNumber* *EvaluatorException,getLineNumber* *EvaluatorException,getLineSource* *EvaluatorException,getMessage* *EvaluatorException,getSourceName* *Interpreter,interpret* *JavaAdapter,doCall* *JavaScriptException,getLineNumber* *JavaScriptException,getSourceName* *JavaScriptException,JavaScriptException* *JavaScriptException,toMessage* *Main,evaluateScript* *Main,processSource* *NativeGlobal,js_escape* *NativeJavaObject,getDefaultValue* *ScriptRuntime,getCatchObject* *ScriptRuntime,newObject* *ScriptRuntime,runScript* *ToolErrorReporter,error* *ToolErrorReporter,formatMessage* *ToolErrorReporter,warning* *WrappedException,WrappedException)*
245882	**JavaImporter constructor** *(Context,initStandardObjects* *IdScriptable,addAsPrototype* *IdScriptable,getIdName* *ImporterFunctions,call* *ImporterFunctions,setup* *ImporterTopLevel,execMethod* *ImporterTopLevel,getClassName* *ImporterTopLevel,importClass* *ImporterTopLevel,importClass* *ImporterTopLevel,importPackage* *ImporterTopLevel,importPackage* *ImporterTopLevel,importPackage* *ImporterTopLevel,init* *ImporterTopLevel,initStandardObjects* *ImporterTopLevel,initStandardObjects* *ImporterTopLevel,js_construct* *ImporterTopLevel,methodArity* *ImporterTopLevel,realThis)*

254778	Rhino treats **label** as separated **statement** *(Codegen,transform* *CompilerEnvirons,isFromEval* *CompilerEnvirons,reportSyntaxError* *CompilerEnvirons,reportSyntaxWarning* *CompilerEnvirons,setFromEval* *Interpreter,compile* *IRFactory,createBreak* *IRFactory,createContinue* *IRFactory,createDoWhile* *IRFactory,createFor* *IRFactory,createForIn* *IRFactory,createLabel* *IRFactory,createLoop* *IRFactory,createWhile* *Jump,getContinue* *Jump,getLabel* *Jump,setContinue* *Jump,setLabel* *Node,toString* *NodeTransformer,NodeTransformer* *NodeTransformer,reportError* *NodeTransformer,transformCompilationUnit_r* *OptTransformer,OptTransformer* *Parser,function* *Parser,matchLabel* *Parser,statement* *Parser,statementHelper)*
255595	Factory **class** for **Context** creation **[*call default enter java runtime thread*]** *(Context,call* *IFGlue,call* *IFGlue,ifglue_call* *JavaAdapter,callMethod)*
256318	NOT_FOUND and **ScriptableObject**.**equivalentValues** *(Namespace,equals* *Namespace,equals* *Namespace,equivalentValues* *QName,equals* *QName,equals*

	QName,equivalentValues *ScriptableObject,equivalentValues* *ScriptRuntime,eq* *XMLObjectImpl,equivalentValues)*
256339	**Stack**less **interpreter** *(Context,addInstructionCount* *Context,getSourcePositionFromStack* *InterpretedFunction,call* *Interpreter,activationGet* *Interpreter,activationPut* *Interpreter,do_cmp* *Interpreter,do_eq* *Interpreter,do_getElem* *Interpreter,do_setElem* *Interpreter,do_sheq* *Interpreter,dumpICode* *Interpreter,generateICode* *Interpreter,generateICodeFromTree* *Interpreter,getJavaCatchPC* *Interpreter,getSourcePositionFromStack* *Interpreter,icodeTokenLength* *Interpreter,icodeToName* *Interpreter,interpret* *Interpreter,notAFunction* *Interpreter,stack_boolean* *Interpreter,stack_double* *State,initState* *State,interpret* *State,interpret* *State,releaseState* *State,State)*
256575	Mistreatment of **end**-of-**line** and **semi**/**colon [*eol*]** *Decompiler,addAssignOp* *Decompiler,decompile* *IRFactory,createAssignment* *IRFactory,createAssignmentOp* *IRFactory,createForIn* *NativeRegExp,isLineTerm* *Parser,addError* *Parser,addExpr* *Parser,andExpr*

	Parser,argumentList *Parser,assignExpr* *Parser,attributeIdentifier* *Parser,bitAndExpr* *Parser,bitOrExpr* *Parser,bitXorExpr* *Parser,checkWellTerminated* *Parser,checkWellTerminatedFunction* *Parser,condExpr* *Parser,eqExpr* *Parser,expr* *Parser,function* *Parser,matchLabel* *Parser,memberExpr* *Parser,memberExprTail* *Parser,mulExpr* *Parser,mustMatchToken* *Parser,newOrQualifiedName* *Parser,orExpr* *Parser,parseFunctionBody* *Parser,primaryExpr* *Parser,relExpr* *Parser,reportError* *Parser,reportWarning* *Parser,shiftExpr* *Parser,statement* *Parser,statementHelper* *Parser,statements* *Parser,unaryExpr* *Parser,variables* *Token,name* *TokenStream,getChar* *TokenStream,getLine* *TokenStream,getOp* *TokenStream,getToken* *TokenStream,getTokenno* *TokenStream,isJSLineTerminator* *TokenStream,matchToken* *TokenStream,peekToken* *TokenStream,peekTokenSameLine* *TokenStream,reportCurrentLineError* *TokenStream,reportCurrentLineWarning* *TokenStream,TokenStream* *TokenStream,tokenToString* *TokenStream,ungetToken*

257128	**Interpreter** and **tail call** elimination **[*java js stack*]** *(Interpreter,handleException* *Interpreter,initState* *Interpreter,interpret* *Interpreter,releaseState* *Interpreter,visitCall* *Interpreter,visitDotQery* *Interpreter,visitExpression* *Interpreter,visitGetElem* *Interpreter,visitGetProp* *Interpreter,visitGetRef* *Interpreter,visitIncDec)*
258144	Add option to set **Runtime Class** in classes **generated** by **jsc [*run main script*]** *(ClassCompiler,ClassCompiler* *ClassCompiler,compileToClassFiles* *ClassCompiler,getMainMethodClass* *ClassCompiler,mainMethodClassName* *ClassCompiler,setMainMethodClass* *Codegen,generateMain* *Main,processOptions)*
258183	**catch** (e if **condition**) does not **rethrow** of original **exception** *(BodyCodegen,generateCatchBlock* *BodyCodegen,generateCodeFromNode* *BodyCodegen,generatePrologue* *BodyCodegen,pushBackTrackState* *BodyCodegen,visitCatchScope* *Interpreter,addExceptionHandler* *Interpreter,addIndex* *Interpreter,addStringOp* *Interpreter,allocLocal* *Interpreter,dumpICode* *Interpreter,generateICode* *Interpreter,getExceptionHandler* *Interpreter,getLineNumbers* *Interpreter,getSourcePositionFromStack* *Interpreter,iCodeTokenLength* *Interpreter,interpret* *Interpreter,releaseFrame* *Interpreter,releaseLocal*

	Interpreter,updateLineNumber *Interpreter,visitTry* *IRFactory,createIf* *IRFactory,createLoop* *IRFactory,createTryCatchFinally* *IRFactory,makeJump* *Node,propToString* *Node,toString* *ScriptRuntime,createFunctionActivation* *ScriptRuntime,enterActivationFunction* *ScriptRuntime,getCatchObject* *ScriptRuntime,newCatchScope* *Token,name)*
258417	java.long.**ArrayIndexOut**Of**BoundsException** in org.mozilla.javascript.**regexp**.**NativeRegExp [*stack size state data*]** *(NativeRegExp,MatchRegExp)*
258959	**ScriptableInputStream** doesn't use **Context**'s application **ClassLoader** to **resolve** classes *(ScriptableInputStream,ResolveClass* *ScriptableInputStream,ScriptableInputStream)*
261278	**Strict mode** *(Main,processOptions* *ScriptRuntime,evalSpecial* *ScriptRuntime,setName* *ShellContextFactory,hasFeature* *ShellContextFactory,setStrictMode)*