2015

Multimodal Code Search

Shaowei WANG
Singapore Management University, shaoweiwang.2010@phdis.smu.edu.sg

Multimodal Code Search

by

Shaowei Wang

Submitted to School of Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Information Systems

Dissertation Committee:

David LO (Chair)
Assistant Professor of Information Systems
Singapore Management University

Lingxiao JIANG
Assistant Professor of Information Systems
Singapore Management University

Feida ZHU
Assistant Professor of Information Systems
Singapore Management University

Julia LAWALL
Director of Research
Inria - Laboratoire d'Informatique de Paris 6

Singapore Management University

2015

Multimodal Code Search

Shaowei Wang

Abstract

Today's software is large and complex, consisting of millions of lines of code. New developers of a software project always face significant challenges in finding code related to their development or maintenance tasks (e.g., implementing features, fixing bugs and adding new features). In fact, research has shown that developers typically spend more time on locating and understanding code than modifying it. Thus, we can significantly reduce the cost of software development and maintenance by reducing the time to search and understand code relevant to a software development or maintenance task. In order to reduce the time of searching and understanding relevant code, many code search techniques are proposed.

For different circumstances, the best form of inputs (i.e., queries) users can provide to search for a piece of code of interest may differ. During development, developers usually like to search a piece of code implementing certain functionality for reuse by expressing their queries in free-form texts (i.e., natural language). After deployment, users might report bugs to an issue tracking system. For these bug reports, developers would benefit from an automated tool that can identify buggy code from the descriptions of the symptoms of the bugs. During maintenance, developers may notice that some pieces of code with a particular structure are potentially buggy. A code search technique that allows users to specify the code structure using a query language may be the best choice. In another scenario, developers may have found some buggy code examples and they would like to locate other similar code snippets containing the same problem across the entire system. In this case, a code search technique that takes as input known buggy code examples is the best choice. During testing, suppose developers have execution traces of a suite of test cases,

they might want to use these execution traces as input to search the buggy code. Developers may also like to provide feedback to the code search engine to improve results. From the above examples, we could see that there is a need for *multimodal* code search which allows users to express their needs in multiple input forms and processes different inputs with different strategies. This will make their search more convenient and effective.

In this dissertation, we propose a *multimodal* code search engine, which employs novel techniques that allow developers to effectively find code elements of interest by processing developers' inputs in various input forms including free-form texts, an SQL-like domain-specific language, code examples, execution traces, and user feedback. In the multimodal code search engine, we utilize program analysis, data mining, and machine learning techniques to improve the code search accuracy. Our evaluations show that our approaches improve over state-of-the-art approaches significantly.

Table of Contents

List of Figures

List of Tables

Acknowledgments

My dissertation did not come forth fully-formed, but is the result of continual support and encouragement by those in my professional life. First of all, I would like to thank my advisors, Prof. Lo and Prof. Jiang, for their sustained guidance and support throughout the learning, research and writing process. My growth as a researcher, would have been impossible without their unceasing patience and effort. I would like to thank my collaborators and lab mates have not only been pivotal in helping to develop, evaluate, and publish my work, but also gave valuable suggestions when I needed them most. I also would like to thank my committee members for their commitment to see me through the doctoral process, as well as for their excellent feedback and guidance. Finally, I would like to thank the reviewers of the conferences/journals where I have submitted papers to for their constructive comments to improve my dissertation.

Chapter 1

Introduction

1.1 Motivation

Millions of open source and industrial software systems have been developed and deployed. The development of new systems can benefit from reusable knowledge hidden in many existing systems if developers can search through existing code and find relevant pieces of code. Furthermore, development is only a start of the lifecycle of a project; more work needs to be done during maintenance. New features need to be integrated and bugs would need to be fixed. Studies show that many code changes (e.g., feature additions, bug fixes, code refactorings) are widely spread across the whole system [108], which increases the difficulty of finding all relevant code. This case is even worse for legacy applications, which is often referred as "legacy crisis" [94]. Since both development and maintenance of software systems require constant search through various code bases and documents, we can reduce development and maintenance costs by helping developers to find and understand the source code in which they are interested more effectively and efficiently.

In information retrieval area, there is a type of search called *multimodal search*, which uses different methods to get relevant results [19]. It could use different kinds of search, such as search by keywords, search by concept, search by example, etc. Multimodal search engine uses different inputs of different nature and methods of

search to achieve more accurate results. There are also engines that leverage user feedback and evaluation to improve search effectiveness and produce more relevant results.

In software engineering area, users need to search code via different forms of inputs (i.e., queries) for different purposes. During development, developers usually like to search for a piece of code implementing certain functionality for reuse by expressing their queries in free-form texts (i.e., natural language). After deployment, users might report bugs to an issue tracking system. For these bug reports, developers would benefit from an automated tool that can identify buggy code from the descriptions of the symptoms of the bugs. During maintenance, developers may notice that some pieces of code with a particular structure are potentially buggy. A code search technique that processes SQL-like domain-specific language queries may be the best choice as it allows users to specify the code structure with the query language. In another scenario, developers may have found some buggy code examples and they would like to locate other similar code snippets containing the same problem across the entire system. In this case, a code search technique that takes as input known buggy code examples is the best choice. During testing, suppose developers have execution traces of a suite of test cases, they might want to use these execution traces as input to search the buggy code. Sometimes, developers may would like to provide feedback to the code search engine to improve the search results. From the above examples, we could see that searching by one type of input is not sufficient. There is a need for a *multimodal* code search engine which allows users to express their needs in multiple input forms and is able to process user inputs in different forms using different search strategies.

1.2 Thesis Overview

In this dissertation, we propose a *multimodal* code search engine, which employs novel techniques that allow developers to effectively find code elements of inter-

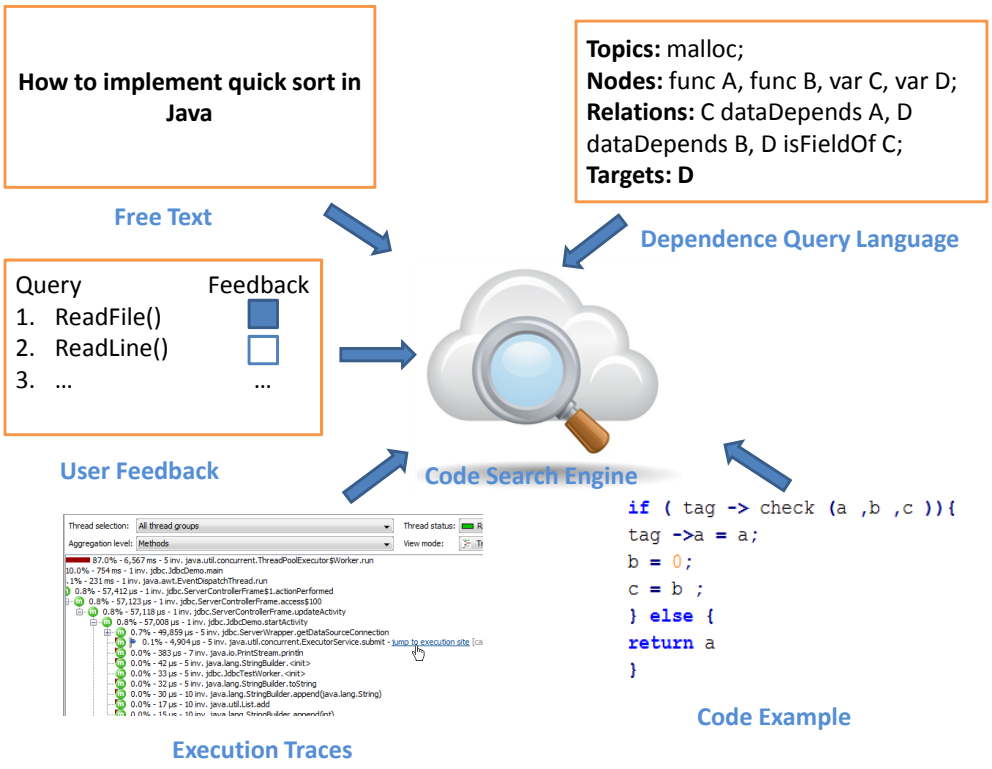

Figure 1.1: Overview of Multimodal Code Search

est by processing various developers' input forms with different search strategies. The input forms could be free-form texts, an SQL-like domain-specific language, code examples, execution traces, and user feedback. We present an overview of multimodal code search in Figure 1.1.

Specifically, this dissertation makes the following contributions by proposing code search techniques that process various kinds of input:

- **Free-form texts** We propose a new technique named Amalgam that integrates information about similar bug report, historical information in the version control system, the structural information of bug reports and source code to locate buggy source code files from bug reports [105]. Our experimental results show that Amalgam outperforms the state-of-the-art technique (i.e., BLUiR+ [92]).

- **SQL-like domain-specific language** We propose an SQL-like domain-specific query language that not only allows developers to specify their search

targets in a specialized graph query language, but also in a free-form format describing desired topics [106]. The query language allows users to specify their search targets in both high level and low level. The results show that our approach improves the state-of-the-art technique significantly.

- **Code examples** We propose a new technique named AutoQuery that can automatically construct dependency queries from a set of code snippets [110]. We realize AutoQuery by the following major steps: firstly, code snippets (that are not necessarily compilable) are converted into program dependence graphs (PDGs); secondly, a new graph mining solution is built to return common properties in the PDGs; thirdly, the common structures are converted to dependency queries, which are used to retrieve results by using a dependence-based code search technique.

- **Execution traces** Many fault localization measures have been proposed to identify buggy program elements by analyzing failing and successful test cases. However, no single fault localization measure completely outperforms others: a measure that is more accurate in localizing some bugs in some programs is less accurate in localizing other bugs in other programs. We propose to compose existing spectrum-based fault localization measures into a more improved measure [111]. Our experimental results demonstrate that our compositional measure improves the existing fault localization measure significantly.

- **User feedback** We present an active approach to incorporate users' opinions on results from a code search engine to refine result lists: as a user forms an opinion about one result, our technique takes this opinion as feedback and leverages it to re-order the results to make truly relevant results appear earlier in the list [109].

The remainder of this dissertation is organized as follows: Chapter 2 discusses

related work. Chapter 3 elaborates the details of Amalgam. Chapter 4 elaborates how we integrate topic models and dependence graphs to improve the search accuracy with Dependency Query Language (DQL). Chapter 5 presents how to address the weaknesses of DQL-based query search by directly processing code examples as input. Chapter 6 presents the search-based compositional fault localization technique. Chapter 7 describes how we use user feedback to improve search results. Finally, Chapter 8 summarizes the contributions of the dissertation and presents future work.

Chapter 2

Literature Review

In this section, we describe closely related studies on code search, concern localization, program dependence graphs, active learning for software engineering and fault localization.

2.1 Code Search

Currently, there are many code search approaches proposed to help users find relevant code. Some of them take textual information as input to search code. McMillan et al. propose Portfolio that takes natural language descriptions as input and outputs a list of functions or code fragments along with the corresponding call graphs [77]. McMillan et al. propose a novel approach to detect similar Java applications and evaluate it on a large repository of Java applications [76]. Thung et al. improve McMillan et al.'s work by utilizing software tags [103]. Chan et al. propose an approach to help developers find usages of API methods provided only simple text phrases. They use an efficient graph search algorithm to return an optimal connected subgraph that matches the user's query [13]. Grechanik et al. propose an approach called Exemplar to find highly relevant software projects from large code base [24]. Exemplar takes a description containing high level concepts written in natural language as an input and outputs a list of applications implementing these

concepts. Haiduc et al. propose a code search tool namely Refoqus [27] that is able to predict the quality of a textual query for further query reformulation. Refoqus takes a query from a developer and returns ranked results along with the quality of the query to the developer. If the quality is low, Refoqus recommends a strategy to reformulate the query for better results. Different from above works, we integrate the dependency among program elements of source code with text information to refine the code search results in Chapter 4 and Chapter 5.

There are also some code search code engines allowing users to query with structural information. Martin et al. propose Program Query Language (PQL) [73]. This query language allows users to specify patterns specifying series of method invocations on an object. A static analysis is first performed to locate candidate program points. The candidate program points are later instrumented. User queries are matched dynamically when the instrumented program is run. Wang et al. [106] propose a semantic dependence search engine that integrates both textual information and structural information from source code and can retrieve code fragments based on queries that describe both topics and dependencies. S^6 is a code search engine that maps high-level queries into a subset of relevant code fragments by using a set of user-guided program transformations [86]. S^6 allows a user to query with textual information and additional low-level details, such as data types.

There are other techniques that accept a code example and return other similar code examples [55, 57]. These techniques are often based on code clone mining [39, 42, 49, 91]. A code clone is a specialized form of code search, i.e., searching for multiple code segments that are similar to one another. There are a few studies on recommendation of code examples [36, 69, 100, 101, 124]. These techniques work at the level of API invocations, while our approach also handles other program elements (*e.g.*, branches, variables, assignments). There are many other specialized code search techniques: Program slicing first proposed in [114] is also a form of code search; the focus is to find code elements that affect or are dependent on a given variable, Maule et al. propose an approach to recommend code elements that

are affected by a change in a database schema [75], Wang et al. propose an approach to find strings that need to be changed to support internationalization [113], etc. These techniques are designed to search code impacted by some particular factors (e.g., a change in code, a given variable), while our approach allows users to search code with particular functionalities, concepts or structure properties.

2.2 Concern Localization

Concern localization refers to the process of linking textual descriptions to corresponding source code files. Concern localization may also be referred to as feature location [12, 116], concept assignment, bug localization [16, 125], or trace recovery [47, 117] in different contexts.

Antoniol et al. [5] apply both a probabilistic and a vector space model to recover links between source code units and free text documents (such as manual pages or requirements). Marcus and Maletic [71] use Latent Semantic Indexing (LSI) for a similar purpose. The underlying assumption is that programmers use meaningful words for code units, and these words capture application-specific knowledge. Therefore, IR techniques can help to link concerns expressed in natural language with code units. Different from their work, we combine the information from source code, historical similar bug report, and version control system to perform more accurate bug localization in Chapter3.

A few hybrid approaches that combine information retrieval (IR) techniques with dynamic/static program analysis or other techniques have been proposed. For example, Zhao et al. [123] present a two-phase approach to feature location, which first applies an IR technique to identify an initial set of feature-code-units links, and then enriches the initial links by exploring the program call graph. Poshyvanyk and Marcus [83] use Formal Concept Analysis (FCA) to group LSI-based feature location results according to common topics. In Chapter 7, we leverage active learning to dynamically refine the results based on users' feedback to improve existing

approaches.

2.3 Program Dependence Graph, Its Construction and Usages

Program dependence graphs was proposed by Horwitz and Reps [37]. Data and control dependencies can be detected more accurately with better pointer analysis and string analysis algorithms. Many studies propose new algorithms for pointer analysis [29, 54] and string analysis [21]. In Chapter 5, we generate program dependence graphs from code fragments. We address the challenge of generating PDGs from non-compilable code fragments.

PDGs have been utilized by many past studies. Komondoor and Horwitz use PDGs for detecting duplicated code, aka. clones [53]. Baah et al. build a probabilistic PDG and use it to localize bugs in programs given a set of failing and correct executions [10]. In this work, we focus on the usage of PDGs for automatic construction of queries for dependence-based code search.

2.4 Active Learning for Software Engineering

Several existing studies in software engineering also employ active learning. Li et al. use active and semi-supervised learning for defect prediction [58]. They address the problem of the lack of historical defect data for software modules. They propose an active sample selection method to pick the most helpful modules to be analyzed for defects. A semi-supervised learner is then used to learn a classification model from the selected small sample of defect data. Lucia et al. propose an approach that adds an active learning layer on top of existing clone-based bug detection tools to increase the true positive rate [66]. They conduct their experiment on three large programs (Linux Kernel, Eclipse, and ArgoUML) and demonstrate significant improvements. Hayes et al. use the standard Rocchio algorithm to improve the quality

of requirement tracing techniques which infer links between two textual documents, e.g., high level to low level requirements with relevance feedback [31].

2.5 Fault Localization

Recently, there have been many studies on fault localization and automated debugging. There are even different ways to categorize these studies. Based on the data used in the approaches, fault localization techniques can be classified into *spectrum-based* and *model-based*.

Spectrum-based fault localization techniques always use program spectra, which are program traces or abstractions of program traces that represent program runtime behaviors in certain ways, to correlate program elements (e.g., statements, basic blocks, functions, and components) with program failures, often with the help of statistical analyses.

Many spectrum-based fault localization techniques [15, 45, 60, 65, 87, 120] take as inputs two sets of spectra, one for successful executions and the other for failed executions, and report candidate locations where root causes of program failures (i.e., faults) may reside. Given a failed program spectrum and a set of correct spectra, Renieris and Reiss present a fault localization tool WHITHER [87] that compares the failed execution to the nearest correct execution and reports the most suspicious locations in the program. Zeller and Hildebrandt propose a technique called Delta Debugging that can simplify and isolate failure-inducing inputs in a binary-search-like fashion [121]. Zeller applies Delta Debugging to search for the minimum state differences between a failed execution and a successful execution that may cause the failure [120]. Cleve and Zeller also extended the work by incorporating a search capability for cause transitions, namely locations in the program where new relevant variables become a failure cause, in their tool called AskIgor [15]. Liblit et al. propose a technique to search for predicates whose true evaluation correlates with failures [60]. Chao et al. extend the work by incorpo-

rating information on the outcomes of multiple predicate evaluations in a program run in their tool called SOBER [65]. Artzi et al. propose a directed test generation technique to provide enough test cases for fault localization techniques [7]. They evaluate their approach by using Ochiai. Artzi et al. also extend Tarantula for debugging web applications [8]. Their proposed approach employs a mapping that maps parts of an output to parts of the program generating it. This mapping is very effective for HTML pages since they are rich structures. However, it is unlikely to be effective for a program that produces simple structures or even a single number – the code responsible for producing this output might be the entire program.

Other spectrum-based techniques [25, 41, 98, 122] only use failed executions as the input and systematically alter the program structure or program runtime states to locate faults. Zhang et al. [122] search for faulty program predicates by switching the states of program predicates at runtime. Sterling and Olsson use the concept of program chipping [98] to automatically remove parts of a program so that the part that contributes to the failure may become more apparent. While their tool, ChipperJ, works on syntax trees for Java programs, Gupta et al. [25] work on program dependency graphs and use the intersection of forward and backward program slices to reduce the sizes of failure-relevant code for further inspection. Jeffrey et al. use a value profile based approach to rank program statements according to their likelihood of being faulty [41]. As these techniques need to alter program states, in general they are more heavyweight than many of those presented in the previous paragraph.

We summarize the main differences between our works and existing works in terms of five input forms as follows: 1. In terms of free-form texts, we are the first to integrate information about similar bug report, historical information in the version control system, the structural information of bug reports and source code together to perform bug localization, while the existing works only involve at most two kinds of information. 2. In terms of SQL-like domain-specific language, we integrate topic mode and dependency among program elements to search code, while the

past works use topic model or dependency among program elements separately. 3. In terms of code examples, compared with the existing works which require users to format query in SQL-like domain-specific language manually, we take input as code examples to construct SQL-like domain-specific language automatically. 4. In terms of execution traces, different from the past works which use individual measure or combine measure heuristically, we use genetic algorithm to get a near optimal combination of 22 fault localization measures to perform fault localization. 5. In terms of user feedback, we are the first to make use of user feedback to refine code search results with active learning.

Chapter 3

Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization

3.1 Introduction

Software systems are often plagued with bugs. To improve the reliability of systems, developers often allow users to submit bug reports to bug tracking systems. Unfortunately the number of these reports is often too large for developers to handle manually in a timely manner. Anvik et al. cited a Mozilla triager that mentioned "Everyday, almost 300 bugs appear that need triaging. This is far too much for Mozilla programmers to handle" [6]. One of the most time consuming task to resolve a bug report is to find the buggy files that are responsible for a reported bug. A system may contains thousands or more files and often only one or a few of these files need to be changed to fix a bug. For example, Lucia et al. analyzed 374 bugs from Rhino, AspectJ and Lucene and found that 84-93% of the bugs reside in 1-2 source code files [68]. Thus, localizing the buggy files is like finding one or two needles in a big haystack.

To address the above mentioned challenge a number of studies have proposed

ways to identify buggy program files given a bug report. Many of these approaches are information retrieval-based and they work by computing similarities between a reported bug and the source code files [85, 92, 97, 125]. The source code files are then ranked based on their similarities to a reported bug. In this work, we are particularly interested in three recent approaches, we highlight each of them below.

Sisman and Kak leverage version history data for bug localization [97]. Their approach makes use of history data to compute the probability that a file is buggy. They also compute a similarity score between the file and a bug report. The probability and the similarity scores are then added up, and the resulting score is used to rank source code files (see [97]). They propose various variants of similarity and the best performing one is named TFIDF-DHbPd.

Zhou et al. leverage similarities among bug reports for bug localization [125]. Given a new bug report, their approach, named BugLocator, finds files that are fixed to resolve similar older bug reports. Based on the similarity of the new and older bug reports, and the number of files that are fixed in each of the older bug reports, their proposed approach computes a bugginess score (referred to as SimiScore) for each source code file. They also compute a similarity score between a file and a bug report (referred to as rVSMScore). Weighted SimiScore and rVSMScore scores are then added, and the resulting score is used to rank source code files.

To measure similarity between a bug report and a file, Saha et al. propose an approach named BLUiR which leverages the structure of a bug report and a source code file [92]. A bug report has many fields (description and summary) and a file has many parts (class names, method names, variable names, and comments). They thus transform a bug report and a file into structured documents and employ structured information retrieval. In structured information retrieval, the textual contents of each field in a bug report and each part of a source code file are considered separately. Each field of a bug report is compared to each part of a source code file, and a similarity score is computed for each comparison. The sum of these scores is then used to rank source code files. Saha et al. also creates an extension of BLUiR, we

Table 3.1: Comparison of Our Approach to State-of-The-Art Bug Localization Techniques

Approach	Version History	Similar Report	Structure
TFIDF-DHbPd [97]	Yes	No	No
BugLocator [125]	No	Yes	No
BLUiR [92]	No	No	Yes
BLUiR+ [92]	No	Yes	Yes
Our Approach	Yes	Yes	Yes

refer to as BLUiR+, which integrates similar report information into BLUiR, in the same manner as BugLocator integrates SimiScore into rVSMScore.

The above three approaches presented leverage different sources of information – see Table 3.1. Sisman and Kak's approach uses version history but neither related reports nor structure. Zhou et al.'s approach uses related reports but neither version history nor structure. Saha et al. approach uses both related reports and structure (i.e., for BLUiR+) but does not use version history. Thus, none of these approaches combine version history, related reports, and structure together. Based on this observation, in this work, we combine these three together, and investigate if the resulting approach works better. To do so, we combine a bug prediction technique used in Google, with the works of Zhou et al. and Saha et al. The bug prediction technique computes the probability of a file to be buggy based on historical data in a version control system. We then have a solution that puts together version history, similar report, and structure. We name our approach AmaLgam, which stands for Automated Localization of Bug using Various Information. The way AmaLgam combines historical information is different from that of Sisman and Kak in the following respects:

1. Our approach uses a well-tested bug prediction formula that is used in Google and it takes into consideration the effect of change burst [56].

2. Sisman and Kak consider the complete version history to compute a probability. Our approach only considers very recent version history and totally discards historical information that are more than k days away from the time a new bug report is submitted.

3. Sisman and Kak simply sums up the probability of a file to be buggy and the similarity of a bug report to the file. Our approach assigns weights that govern the contribution of the probability of a file to be buggy (computed by the bug prediction technique) and the similarity of a bug report to a file (computed by integrating BugLocator and BLUiR).

The contributions in this chapter are as follows:

1. We are the first to put together version history, similar reports and structure for bug localization. Past bug localization studies have only used one or two of these information sources. We have evaluated our approach on more than 3,000 bug reports from four open source projects: AspectJ, Eclipse, SWT, and ZXing. Our experiments show that we can achieve an MAP of 0.33, 0.35, 0.62, and 0.41 for each of the four projects, respectively.

2. The MAP scores of our proposed approach improve on those of Saha et al. (i.e., BLUiR+) that put together similar reports and structure by an average of 12.5%. The MAP scores of our proposed approach improve on those of BLUiR and Zhou et al. (BugLocator) by an average of 16.4% and 24.4%, respectively. Comparing with the reported results of the history-aware bug localization solution of Sisman and Kak (TFIDF-DHbPd), which was only evaluated on AspectJ bug reports from the iBugs dataset, our approach improves the MAP score by 46.1%.

The structure of the remainder of the chapter is as follows. In Section 3.2, we first present preliminary information on bug reports and a motivating example. We elaborate the details of our approach in Section 3.3. We describe our experimental

setup and results in Section 3.4. We finally conclude and mention future work in Section 3.5.

3.2 Preliminaries and Example

In this section, we first describe some preliminary information on bug reports. We then outline some text pre-processing steps that are applied to the bug reports. Finally, we show an example to illustrate why it is useful to consider version history, similar report, and structure.

3.2.1 Bug Reports

A bug report is a document submitted by users to describe an error that they experience when they use a system. A bug report contains a number of fields; we are particularly interested in four of them, namely bug identifier (id), the date a bug report was submitted (open date), the summary of the error (summary), and the more detailed description of the error (description).

We present a bug report from Eclipse in Figure 3.1, The bug report can be downloaded from Eclipse's Bugzilla[1]. The identifier of this bug report is 76138 and it describes a problem with the ant editor which does not follow a display setting. The bug id provides a reference number that can be used to identify commits in version control systems that fix it, c.f. [125]. The open date helps us to identify other bug reports that are submitted a number of days prior to bug 76138. The summary and description fields help us to understand the error that the user experienced.

A bug localization tool takes as input a bug report and returns the potentially buggy files. The corresponding buggy Java files for the bug report shown in Figure 3.1, which are identified by checking the corresponding bug fixing commits, are AntEditor.java and AntEditorSourceViewerConfiguration.java.

[1] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=76138`

Bug ID	76138
Open date	2004-10-12 21:53:00
Summary	**Ant editor** not following tab/space setting on shift right
Description	This is from 3.1 M2. I have **Ant**->**Editor**->Display tab width set to 2, insert spaces for tab when typing" checked. I also have **Ant**->**Editor**->Formatter->Tab size set to 2, and "Use tab character instead of spaces _unchecked_. Now when I open a build.xml and try to do some indentation, everything works fine according to the above settings, except when I highlight a block and press tab to indent it. It's the tab character instead of 2 spaces that's inserted in this case.
Fixed Files	org.eclipse.ant.internal.ui.editor.**AntEditor**.java org.eclipse.ant.internal.ui.editor.**AntEditorSourceViewerConfiguration**.java

Figure 3.1: An Eclipse's Bug Report and the Buggy Source Code Files Corresponding to it

3.2.2 Text Pre-processing

An information-retrieval based bug localization technique usually performs three pre-processing steps: text normalization, stopword removal, and stemming. The goal of the text pre-processing steps is to break a bug report or a source code file into terms that can then be analyzed by an information retrieval technique. In the pre-processing step, some compound words (e.g., program identifiers) are broken into parts, and some related words are mapped to the same term. We briefly describe these three pre-processing steps below.

First, text normalization is performed which involves the removal of punctuation marks, tokenization (i.e., extraction of words from paragraphs or identifiers from source code), and identifier splitting. During this step, when a source code is processed, it is converted into an Abstract Syntax Tree (AST) and using this tree, identifiers are identified. These identifiers are split into its constituent words following Camel Case splitting [5]. For example the identifier "getMethodName" is split into "get", "Method", and "name". In this study, both the split words and the full identifier name are kept. For example, for the class name "AntEditorSourceViewerConfiguration", which is one of the buggy files corresponding to the bug report

shown in Figure 3.1, we convert it to 6 words: “ant”, “editor”, “source”, “viewer”, “configuration” and the full identifier name “AntEditorSourceViewerConfiguration”.

Second, we remove stopwords such as “on”, “the”, “are”, “is”, and so on. These stopwords carry little meaning. Finally, we perform stemming which reduces inflected or derived words into a common root form. For example the word “reading” and “reads” are reduced to the root form “read”. By doing this, similar words are represented using the same term. We use the standard Porter Stemmer [82] to perform this stemming step.[2]

3.2.3 Motivating Example

A traditional IR-based bug localization approach usually first performs text preprecessing on a query (a bug report) and the documents in a corpus (source code files). Then, a similarity score between the query and each of the documents are computed based on a particular information retrieval technique (e.g., TFIDF, LDA, LSI, etc), e.g., [85]. From Figure 3.1, we note that the buggy source code file names share a number of common words with the summary and description of the bug report, i.e., “ant” and “editor”. Based on these common words, a traditional IR-based bug localization approach would try to link the bug report with the source code files. In this section, we highlight how version history, similar reports, and structure can be used to improve the accuracy of traditional IR-based bug localization techniques.

Version History. There are lots of historical data of changes to source code files that are stored in a version control system during program evolution. This historical data can be used to improve bug localization performance. Kim et al. found that bugs happen in bursts, and not in isolation [50]. The files responsible for a bug recently are more likely to be responsible for other bugs in the near future. Figure 3.2 presents the commit logs of Eclipse before bug 76138 occurred. We

[2] `http://tartarus.org/martin/PorterStemmer/`

could see that the class files “AntEditor.java” and “AntEditorSourceViewerConfiguration.java”, which were responsible for bug 76138, were also responsible for other bugs that happen prior to the reporting of bug 76138 (they are highlighted in bold). ‘AntEditor.java” is fixed just one day prior to the reporting of bug 76138 and “AntEditorSourceViewerConfiguration.java” is fixed just 7 days prior to the reporting of bug 76138. Thus, we that historical data can be used to better locate bug.

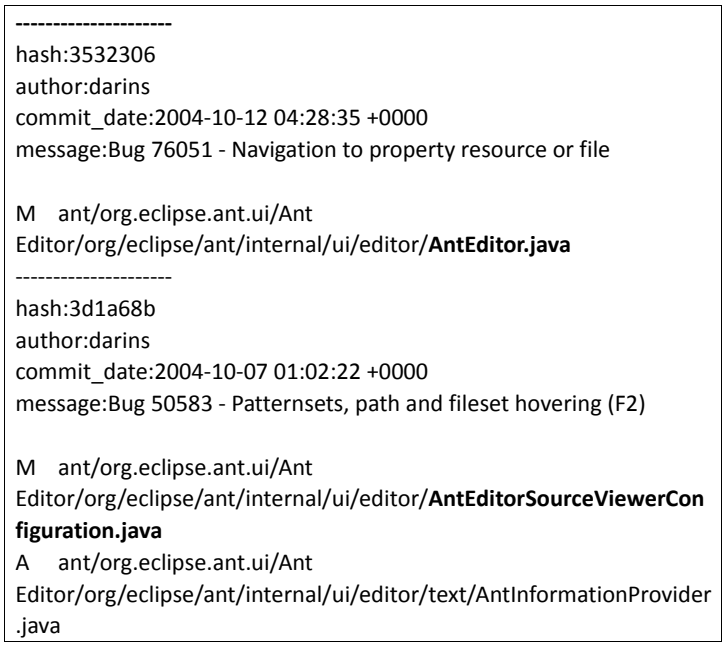

Figure 3.2: Recent Commit Logs Prior to the Reporting of Bug Report 76138

Similar Reports. User often submit many similar bug reports that correspond to different errors that affect the same buggy program elements. For example, Figure 3.3 shows an older report with identifier 50303[3], which was reported 9 months before bug report 76138. Note that this report shares the common words “ant” and “editor” with bug report 76138. Bug report 50303 was fixed on March 17, 2004 and was re-fixed on March 18, 2004 and “AntEditor.java” was modified on both times. By analyzing bug report 50303, we can get a hint on the files that need to be changed to fix bug 76138. From the example, we could see that similar reports can be used to locate bugs.

[3] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=50303`

Bug ID	50303
Open date	2004-01-20 20:55
Summary	**Ant Editor** outline "Link with **Editor**"
Description	Similar to the Java **Editor** it would be a nice enhancement to have a "Link with **Editor**" toggle button for the **Ant Editor** outline page.
FixedFiles	org.eclipse.ant.internal.ui.editor.**AntEditor**.java 7 other files

Figure 3.3: An Older Eclipse's Bug Report and the Buggy Source Code Files Corresponding to It

Structure. Bug reports and source code files have structure. Bug reports have several fields including the summary and the description. Source code files can be split into class names, method names, variable names, and comments. This structural information can be leveraged for bug localization. Traditional IR-based bug localization approaches compute the similarity between a bug report and the entire contents of a source code file (which contains a class name, many variable names, and many comments). For localizing the bug report in Figure 3.3, the class names contain the most important terms. Unfortunately, the impact of the terms "ant" and "editor" in the class names would be weakened by other tokens, which would make the performance poor. Structural information could be used to overcome this problem by computing the similarities of a query against different fields (e.g., class, method, variable, comment) in a source code file separately and summing up those similarities. In this way, the tokens "ant" and "editor" would have stronger impact to the overall similarity. Thus, we conjecture that structure can be used to locate bugs.

3.3 Approach

In this section, we first describe our overall framework named AmaLgam. We then present each of the four main components of AmaLgam.

3.3.1 Overall Framework of AmaLgam

Figure 3.4 presents the overall framework of AmaLgam. AmaLgam takes as input a bug report to be localized (new bug report), a set of source code files of the system for which the bug report is submitted (source code files), a history of commits made to the system as stored in a version control system (version history data), and a set of older bug reports stored in a bug tracking system (bug repository).

The inputs are processed by the three components of AmaLgam namely: version history component, similar report component, and structure component. The version history component makes use of version history information to rank files. The similar report component makes use of older reports in bug repository to rank files. The structure component makes use of the structure of bug reports and source code files to rank files. The three components each output a suspiciousness score for each source code file. These three sets of suspiciousness scores are then input to the composer component which produces the final ranked files.

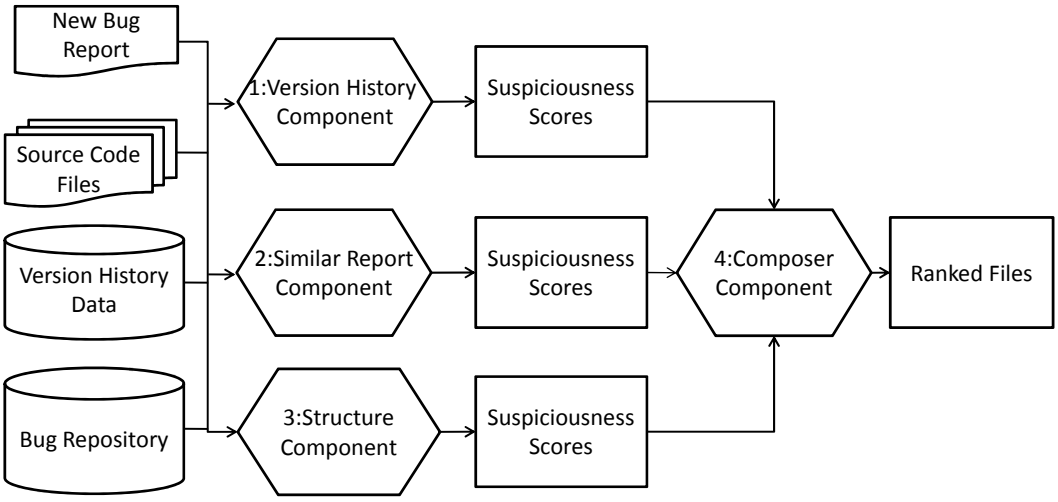

Figure 3.4: The Overall Framework of AmaLgam

3.3.2 Version History Component

For the version history component, we make use of studies on bug prediction whose goal is to predict which files are likely to be buggy in the future, e.g., [50, 84]. Kim et al. propose BugCache which predicts future bugs by maintaining a relatively short list of most fault-prone program entities [50]. Rahman et al. propose a cheaper algorithm which only sorts files based on the number of bug fixing commits that touch each of them [84]. Rahman et al. show that this simple and cheap approach achieves almost the same performance as BugCache. Google's developers have adapted the simple algorithm proposed by Rahman et al to predict bugs on their large systems [56]. The resulting algorithm is simple and fast. Thus, we decide to adapt this well-tested bug prediction algorithm of Google as our version history component. We briefly describe how we adapt this algorithm in the following paragraphs.

The algorithm takes as input commit logs and outputs a list of files with their suspiciousness scores. It first identifies relevant bug-fixing commits. The relevant bug fixing commits are identified by following two rules:

1. The commit log must match the following regular expression $regex$: $(.*fix.*)|(.*bug.*)$. This regular expression matches that all commit logs containing the word "fix" or "bug".

2. The commit must be made in the past k days

We modify Google's approach by including the second requirement [56]. Our experience shows that including older bug-fixing commits does not affect performance much and even can slightly decrease performance. Also, it is computationally cheaper to only consider recent commits. The parameter k can be set empirically. By default, we set k to 15. The algorithm analyzes these relevant commits and assigns a suspiciousness score to each source file f using the following equation:

$$score^H(f,k,R) = \sum_{c \in R \wedge f \in c} \frac{1}{1 + e^{12(1-((k-t_c)/k))}} \quad (3.1)$$

In the above equation, R refers to the set of relevant commits and t_c is the number of days that has elapsed between a commit c and the input bug report. The output of this algorithm is a set of suspiciousness scores, one for each file. We denote the suspiciousness score of file f assigned by the version history component as $Susp^H(f)$.

Example: Consider the input bug report in Figure 3.1 and the two commit logs in Figure 3.2 with identifiers 3532306 and 3d1a68b. For simplicity sake, let us assume that there are no other commit logs. We illustrate how Equation 3.1 is used to compute the suspiciousness scores of the files AntEditor.java and AntEditorSourceViewerConfiguration.java. As both commits contain the word "bug" and they are committed within 15 days before the bug report 76138 was submitted (i.e., Oct 12, 2004, at 21:53:00), they are considered to be relevant bug fixing commits. The value of (k-t_c)/k for commit 3532306 is 0.95 (since the commit was made around 17 hours, i.e., 0.7 day, before the time bug report 76138 was submitted). Thus, the suspiciousness score for AntEditor.java is 1.82. The suspiciousness score of AntEditorSourceViewerConfiguration.java can be computed in a similar way and it is 0.009.

3.3.3 Similar Report Component

For our similar report component, we adapt BugLocator [125], in particular the algorithm that computes SimiRank scores. We describe briefly how we use this algorithm in the following paragraphs.

The algorithm takes in an input bug report and older bug reports that have been fixed in the bug repository. It then measures the similarity of the input bug report to the older fixed bug reports. Based on the similarity scores of the bug reports and the number of files that are modified to fix each bug report, we compute a

suspiciousness score for each source code file.

To measure the similarity of two bug reports, the following steps are followed. First, each bug report is represented by their constituent pre-processed terms. Considering the universe of all terms as $\{t_1, \ldots, t_n\}$, we can compute for a bug report b, a vector $\overrightarrow{b}$:

$$\overrightarrow{b} = tf_b(t_1)idf(t_1), tf_b(t_2)idf(t_2), \ldots, tf_d(t_n)idf(t_n)$$

In the above formula, $tf_b(t_i)$ corresponds to the number of times term t_i appears in bug report b, $idf(t_i)$ corresponds to the reciprocal of the number of documents that contain term t_i. Given vector representations of two bug reports $\overrightarrow{b1}$ and $\overrightarrow{b2}$, their similarity can be measured by computing the standard cosine similarity [11] of their vector representations.

To compute a suspiciousness score for source code file f, we use the following equation:

$$score_R(f, b, B) = \sum_{b' \in \{b' | b' \in B \wedge f \in b'.Fix\}} \frac{sim(b, b')}{|b'.Fix|} \tag{3.2}$$

In the above equation, b is the input bug report, B is the set of older fixed bug reports, $sim(b, b')$ is the similarity of bug reports b and b', $b'.Fix$ is the set of files that are modified to fix bug report b', and $|b'.Fix|$ is the size of set $b'.Fix$. The output of this algorithm is a set of suspiciousness scores, one for each file. In this component, we do not enforce a similarity threshold following what Zhou et al. did in their work [125]. We take all older bug reports to compute the suspiciousness score. We denote the suspiciousness score of file f assigned by the similar report component as $Susp^R(f)$.

Example: Consider the input bug report shown in Figure 3.1 and the older bug report in Figure 3.3. For simplicity sake, let us assume that there are no other bug reports in the bug repository. We would like to illustrate how Equation 3.2 is used

to compute the suspiciousness scores of files AntEditor.java. Let us assume for simplicity sake the similarity of the two bug reports is 0.15. The suspiciousness score of AntEditor.java iss computed as: 0.15/8 = 0.01875.

3.3.4 Structure Component

For the structure component we use BLUiR [92] which performs structured retrieval for bug localization. For completeness, we briefly describe BLUiR in the following paragraphs.

BLUiR breaks a bug report into 2 parts: summary and description. It breaks a source code file into 4 parts: class names, method names, variable names, and comments. Each of these parts can be converted into a vector following a procedure similar to the one described in Section 3.3.3. The suspiciousness score of a source code file f given an input bug report b can then be computed as:

$$score_S(f, b) = \sum_{fp \in f} \sum_{bp \in b} sim(fp, bp)$$

where fp is a part of file f, bp is a field in bug report b, and $sim(fp, bp)$ is the cosine similarity of the vector representations of fp and bp. The output of the structure component is a set of suspiciousness scores, one for each file. We denote the suspiciousness score of the file f assigned by the structure component as $Susp^S(f)$.

Example: Consider a bug report and a file shown in Figure 3.5. After preprocessing, the terms in the summary field of the bug report are: “bug”, “average”, and “function”. The terms in the description field of the bug report are: “us”, “average”, “function”, “measure”, “class”, “comput”, “got”, “wrong”, and “result”. The term in the class name of the file is “measure”. The term in the method name of the file is “average”. The terms in the variable names of the file are “list”, “sum”, and “d”. The set of terms in the comments of the file is Ø. Based on these two fields of the bug report and these four parts of the source code file, we can compute a suspiciousness score which would be a sum of 6 similarity scores.

Bug summary: bug in average function

Bug description: When I used the average function in measure class to compute average, I got a wrong result.

Fixed file:

```
public class Measure{
    static double average(double[] lists){
        double sum = 0;
        for(double d : lists)
            sum += d;
        return sum;
    }
}
```

Figure 3.5: Example Bug Report and Source Code File

3.3.5 Composer Component

This component processes the 3 sets of suspiciousness scores output by the first 3 components of AmaLgam and computes a set of final suspiciousness scores. The composer component first combines the scores output by the structure component and the similar report component for a file f, as follows:

$$Susp^{S,R}(f) = (1-a) \times Susp^{S}(f) + a \times Susp^{R}(f)$$

The value of a can be empirically determined. Saha et al. have also tried to combine standard BLUiR with similar report data and they set the value of a to be 0.2. We use the same value of a. The impact of various a on the effectiveness of incorporating similar report information has been shown in Zhou et al.' work [125]. Based on $Susp^{SR}$ we compute the final suspiciousness score of f as follows:

$$Susp^{S,R,H}(f) =$$

$$(1-b) \times Susp^{S,R}(f) + b \times Susp^{H}(f), \quad if Susp^{S,R}(f) > 0$$

$$0, \quad otherwise$$

In the above equation, we set the final suspiciousness score to 0, if $Susp^{S,R}(f)$ is 0. We do this since we observe that the likelihood of a file to be relevant to a bug report is very small if $Susp^{S,R}(f)$ is 0. The value of b can be empirically determined. By default, we set this value to 0.3.

In the end, the composer component sorts all source code files based on their final suspiciousness scores and this ranked list of files is the output of AmaLgam.

Example: Suppose, the $Susp^{S,R}$ scores of 3 files are $\{f_1 = 0.2,\ f_2 = 0.1,\ f_3 = 0\ \}$ and the $Susp^{H}$ scores of the 3 files are $\{f_1 = 0.1,\ f_2 = 0.5,\ f_3 = 0.9\ \}$. Since the $Susp^{S,R}$ score of f_3 is 0, then the final suspiciousness score of f_3 is still 0 even though its $Susp^{H}$ is large. Thus, the final suspiciousness scores are $f_1 = 0.17$, $f_2 = 0.22$, $f_3 = 0$.

3.4 Experiments

In this section, we first describe the dataset that we use to evaluate our approach. Next, we describe our evaluation metrics, followed by our research questions. Finally, we describe our experimental results which answer the research questions.

3.4.1 Dataset

We use the same dataset used by Zhou et al. and Saha et al. to evaluate BugLocator and BLUiR respectively [92, 125]. This dataset contains a total of 3,379 bug reports from four popular open source projects, AspectJ, Eclipse, SWT, and ZXing. For each bug report, the dataset also provides information on files that were modified to fix the bug. The AspectJ bug reports originate from the iBugs benchmark [16] which was also used by Sisman and Kak to evaluate their approach [97]. Table 3.2 describes the dataset in more detail. For our version history component, we collect commit logs from the Git repositories of those four projects.

Table 3.2: Dataset Details

Project	Description	Period	#Fixed Bugs	#Source Files
AspectJ	Aspect-oriented extension of Java	07/2002-10/2010	286	6485
Eclipse	Open source IDE	10/2004-03/2011	3075	12863
SWT	Open source widget toolkit	10/2004-04/2010	98	484
ZXing	Barcode image processing library for Android platform	03/2010-09/2010	20	391

3.4.2 Evaluation Metrics

To measure the effectiveness of the proposed bug localization approach, we use the following metrics:

- **Top-N Rank (Hit@N)**: This metric calculates the number of bug reports where one of the buggy files appears in the top N (i.e., 1, 5, or 10) ranked files. Given a bug report, if at least one of its buggy files is in the top N results, we consider that the bug is successfully located. The higher the value of this metric is, the better the performance of an approach is.

- **Mean Average Precision (MAP)**: MAP is the most commonly used IR metric to evaluate ranking approaches. It takes the ranks of all buggy files into consideration, instead of only the first one. MAP is computed by taking the mean of the *average precision* scores across all queries. The average precision of a single query is computed as:

$$AP = \sum_{k=1}^{M} \frac{P(k) \times pos(k)}{number\ of\ positive\ instances},$$

where k is a rank in the returned ranked files, M is the number of ranked files and $pos(k)$ indicates whether the k^{th} file is a buggy file or not. $P(k)$ is the precision at a given top k files and is computed as follows:

$$P(k) = \frac{\#buggy\ files}{k}.$$

- **Mean Reciprocal Rank (MRR)**: The reciprocal rank for a query is the reciprocal of the position of the first buggy file in the returned ranked files. MRR is the mean of the reciprocal ranks over a set of queries Q and it can be computed by following equation:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i}$$

where $rank_i$ is the position of the first buggy file in the returned ranked files for the first query in Q.

3.4.3 Research Questions

Research Question 1 How effective is AmaLgam for bug localization?

To answer this research question, we apply AmaLgam to the four sets of bug reports in our dataset. We then evaluate the returned ranked lists and compute Hit@N, MAP, and MRR to characterize the effectiveness of AmaLgam.

Research Question 2 Does AmaLgam outperform other bug localization techniques?

AmaLgam combines 3 state-of-the-art approaches: TFIDF-DHbPd by Sisman and Kak [97], BugLocator by Zhou et al. [125], and BLUiR by Saha et al. [92]. TFIDF-DHbPd is the best performing variant of the approach proposed by Sisman and Kak. Saha et al. also tried to compose similar report information in the same way as Zhou et al. We refer to this variant of BLUiR as BLUiR+. We would like to investigate whether and to what extent AmaLgam outperforms these existing state-of-the-art approaches. We compare the results of AmaLgam with the results reported in past papers [92, 97, 125].

Research Question 3 How does the performance of AmaLgam vary for various settings of parameters k and b?

AmaLgam accepts 3 parameters: k, a, and b. The setting of parameter a follows

the setting used by Saha et al. [92] which actually follows the setting used by Zhou et al. [125]. In this research question, we would like to investigate the effect of varying the other two parameters of k and b. k is a parameter of the version history component which determines the number of days for which we cache the history. b is a parameter of the composer component that determines the contribution of the history-based suspiciousness score ($Susp^H$).

3.4.4 Experiment Results

The following subsections describe our experimental results which answer the three research questions.

RQ1: Effectiveness of AmaLgam

To answer the RQ1, we measure the effectiveness of AmaLgam in terms of the metrics we listed in Section 3.4.2. Table 3.3 presents the results for all programs. For 127 (44.4%) AspectJ bug reports, AmaLgam successfully locates a buggy source code file in the top 1 ranked file. For 187 (65.4%) AspectJ bugs, at least one buggy source code file is among the top 5 ranked files. For 209 (73.1%) AspectJ bugs, at least one buggy source code file is among the top 10 ranked files. In terms of MAP and MRR, AmaLgam achieves a score of 0.33 and 0.45, respectively.

For Eclipse, 1060 (34.5%) and 1775 (57.7%) bugs could be localized by inspecting top 1 and 5 ranked files, respectively. Also, 2059 (67.0%) bugs could be localized when only the top 10 ranked files are inspected. The scores of MAP and MRR that AmaLgam gets for Eclipse are 0.35 and 0.45, respectively. For SWT, 61 (62.2%) bugs have a buggy file at the top 1 ranked file. Also, 80 (81.6%) and 88 (89.8%) bugs are successfully localized when only the top 5 and 10 ranked files are inspected, respectively. AmaLgam gets MAP and MRR scores of 0.62 and 0.71, respectively. For ZXing, AmaLgam is able to localize 8 (40.0%), 13 (65.0%), and 14 (70%) bugs when only the top 1, 5 and 10 ranked files are inspected, respectively. In

Table 3.3: Comparison among AmaLgam, TFIDF-DHbPd, BugLocator, BLUiR, and BLUiR+

Project	**Approach**	**Hit@1**	**Hit@5**	**Hit@10**	**MAP**	**MRR**
AspectJ	AmaLgam	**127 (44.4%)**	**187 (65.4%)**	**209 (73.1%)**	**0.33**	**0.54**
	TFIDF-DHbPd	N/A	N/A	N/A	0.23	N/A
	BugLocator	88 (30.8%)	146 (50.1%)	170 (59.4%)	0.22	0.41
	BLUiR	92 (32.2%)	146 (51.0%)	173 (60.5%)	0.24	0.41
	BLUiR+	97 (33.9%)	150 (52.4%)	176 (61.5%)	0.25	0.43
Eclipse	AmaLgam	**1060 (34.5%)**	**1775 (57.7%)**	**2059 (67.0%)**	**0.35**	**0.45**
	BugLocator	896 (29.1%)	1653 (53.8%)	1925 (62.6%)	0.30	0.41
	BLUiR	952 (31.0%)	1636 (53.2%)	1933 (62.9%)	0.32	0.42
	BLUiR+	1013 (32.9%)	1729 (56.2%)	2010 (65.4%)	0.33	0.44
SWT	AmaLgam	**61 (62.2%)**	**80 (81.6%)**	**88 (89.8%)**	**0.62**	**0.71**
	BugLocator	39 (39.8%)	66 (67.3%)	81 (62.6%)	0.45	0.53
	BLUiR	54 (55.1%)	75 (76.5%)	85 (86.7%)	0.56	0.65
	BLUiR+	55 (56.1%)	75 (76.5%)	86 (87.8%)	0.58	0.66
ZXing	AmaLgam	**8 (40.0%)**	**13 (65.0%)**	**14 (70.0%)**	0.41	**0.51**
	BugLocator	8 (40.0%)	12 (60.0%)	14 (70.0%)	**0.44**	0.50
	BLUiR	8 (40.0%)	13 (65.0%)	14 (70.0%)	0.38	0.49
	BLUiR+	8 (40.0%)	13 (65.0%)	14 (70.0%)	0.39	0.49

terms of MAP and MRR, AmaLgam achieves a score of 0.41 and 0.51, respectively.

We also explore the results manually. In most cases, amalgam performs better than BLUIR+, which demonstrates that our version history component could help to locate the buggy files at most cases. However, in few case, version history component could bring some noise to the final results. For example, when locating the buggy files for bug report 36234 in dataset AspectJ, BlUIR+ could find the buggy file org.aspectj.tools.ajc.Main.java after inspecting 4 results. However AmaLgam needs to inspect 6 results to locate the buggy file. This is because version history component scores two files org.aspectj.ajdt.internal.core.builder.AjBuildConfig.java and org.aspectj.ajdt.internal.core.builder.AjState.java with high scores as they were very frequently modified in the past 15 days before the bug report 36234 was submitted, even they were not related to bug 36234.

RQ2: AmaLgam VS. Other Bug Localization Approaches

Table 3.3 also compares the results of AmaLgam with those of TFIDF-DHbPd, BugLocator, and two versions of BLUiR (BLUiR and BLUiR+) in terms of Hit@1, Hit@5, Hit@10, MAP and MRR. Sisman and Kak only evaluates TFIDF-DHbPd using the AspectJ bug reports from the iBugs benchmark. They also did not compute Hit@N or MRR. Thus, in the table, we only show the MAP score of TFIDF-DHbPd for AspectJ. Our approach improve on their approach's MAP score on AspectJ bug reports by 46.1%.

AmaLgam outperforms BugLocator with respect to all metrics for AspectJ, Eclipse, and SWT bug reports. Both techniques have the same performance in terms of Hit@1, Hit@5, and Hit@10 for ZXing. For ZXing, AmaLgam improves BugLocator in terms of MRR, bug marginally loses to BugLocator in terms of MAP. On average, AmaLgam improves the MAP and MRR scores of BugLocator by 24.4% and 19.3%, respectively.

Comparing AmaLgam with BLUiR and BLUiR+, we could note that AmaLgam consistently outperforms BLUiR and BLUiR+ in terms of MAP and MRR for all

programs. The Hit@N scores of AmaLgam are better than those of BLUiR and BLUiR+ for all programs except ZXing. For ZXing, the Hit@N scores of AmaLgam are the same with those of BLUiR and BLUiR+. On average, AmaLgam improves the MAP and MRR scores of BLUiR+, which performs better than BLUiR, by 12.5% and 9.9% respectively. We perform Wilcoxon signed-rank test [115] to test whether the improvements obtained by AmaLgam over BLUiR+ are significant. We found that the improvements in terms of MAP and MRR are significant.

RQ3: Effect of Varying k and b

In this study, we select $k \in \{5,10,15,20,25,30,35,40,45,50\}$ and compute MAP and MRR for all bug reports in our dataset. The results are presented in Figures 3.6 and 3.7.

When k increases from 0 to 15, the MAP and MRR scores increase for all programs. Increasing the value of k further from 20 to 50 generally does not improve performance much. For SWT, the MAP and MRR slightly decrease when we increase k from 15 to 50. These results show that there is no need to consider old history. The most important part of the history is commits in the last 15-20 days.

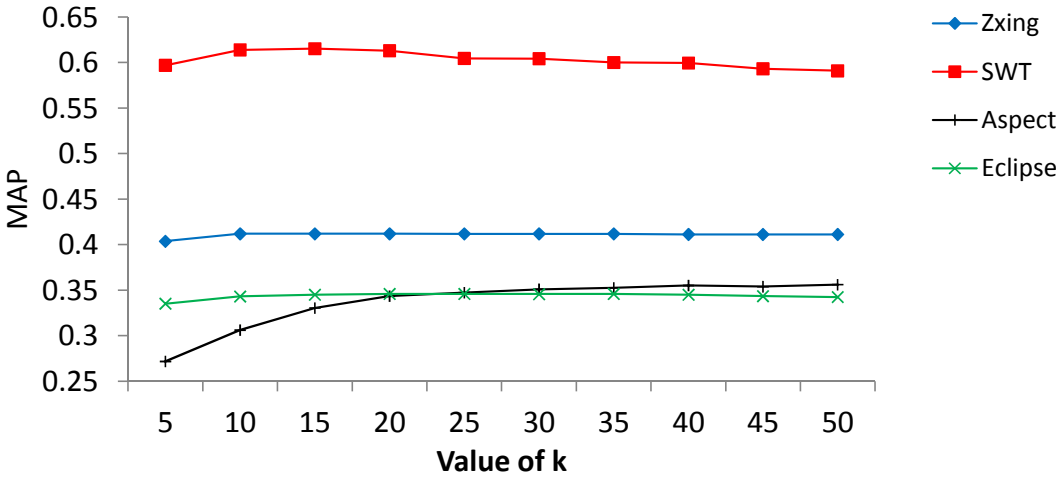

Figure 3.6: Impact of Varying the Value of k on AmaLgam in Terms of MAP

Next, we investigate the impact of different values of b on the performance of AmaLgam. We vary the values of b from 0 to 1 with an interval of 0.1. Figures 3.8

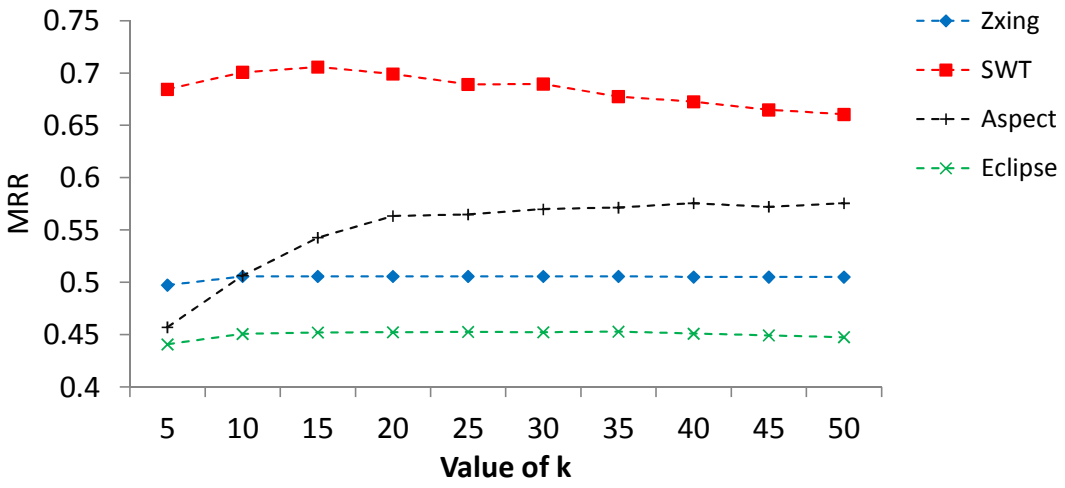

Figure 3.7: Impact of Varying the Value of k on AmaLgam in terms of MRR

and 3.9 show how the performance of AmaLgam varies due to different b values, in terms of MAP and MRR. For ZXing, Eclipse and SWT, both MAP and MRR slightly increase when the value of b is varied from 0 to 0.3; when the value of b is varied from 0.3 to 1, in general the performance goes down.[4] The impact of varying b values for AspectJ is different from the other programs: the MAP value keeps increasing when the value of b is varied from 0 to 0.7, remains relatively stable when b is varied from 0.7 to 0.9, and decreases when b is increased from 0.9 to 1; In terms of MRR, the performance improves when b is varied from 0 to 0.5, remains stable when b is varied from 0.5 to 0.7, and decreases when b is varied from 0.7 to 1.0. For three of the four programs, AmaLgam achieves the best performance when b is 0.3. Averaging across the 4 programs, the performance of AmaLgam remains relatively stable when we vary b between 0.3-0.4.

3.4.5 Threats to Validity

Threats to internal validity include experimenter bias. To reduce this threat, we reuse the bug reports dataset that has been used before to evaluate prior approaches. Thus, the evaluation is not biased to our approach.

Threats to external validity relate to the generalizability of our findings. To

[4]The performance slightly increases for ZXing when b is varied from 0.4 to 0.9.

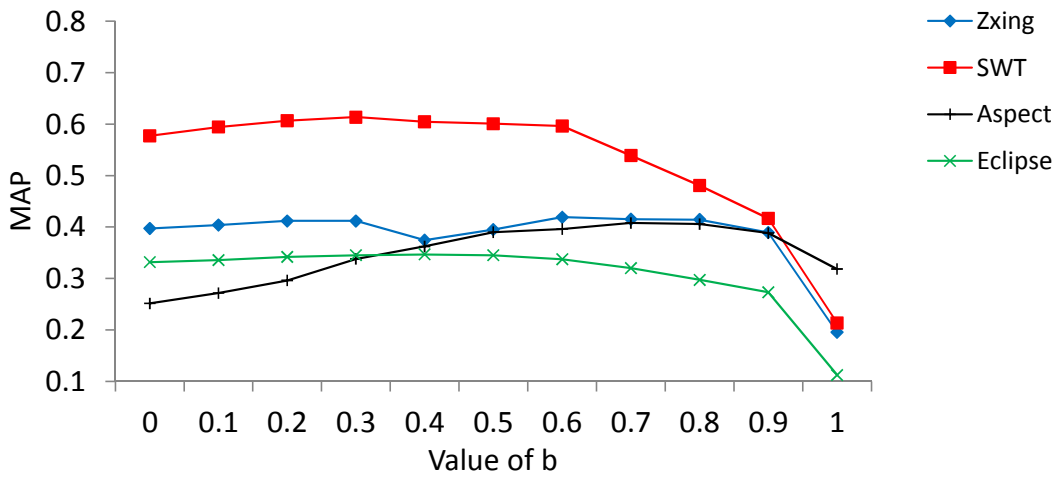

Figure 3.8: Impact of Varying the Value of b on AmaLgam in terms of MAP

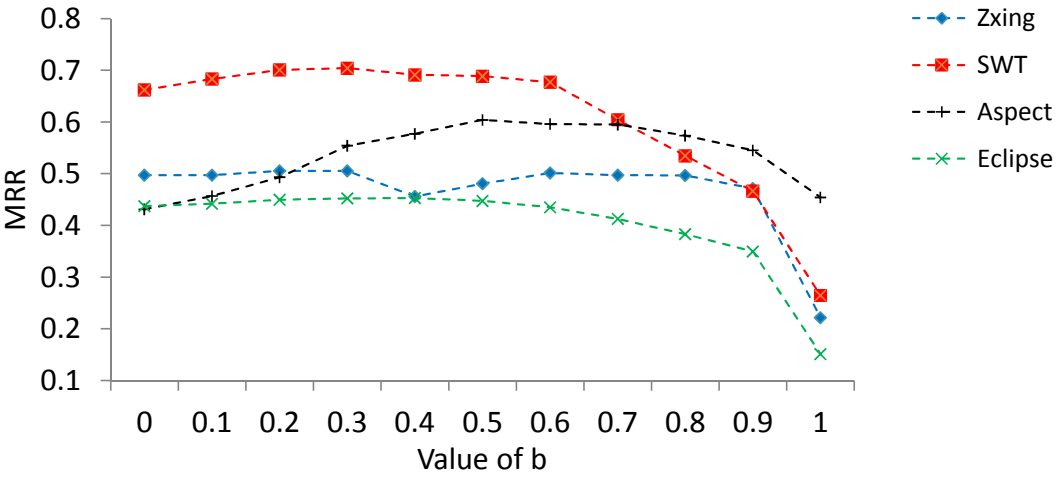

Figure 3.9: Impact of Varying the Value of b on AmaLgam in terms of MRR

reduce this threat, we have analyzed more than 3,000 bug reports from four popular projects. Still in the future, we plan to reduce these threats further by analyzing more bug reports from more projects written in multiple programming languages.

Threats to construct validity refers to the suitability of the set of evaluation metrics that we use in this study. Three metrics are used namely Hit@N, MAP, and MRR. These metrics are well-known information retrieval metrics and have been used before to evaluate many past bug localization approaches, e.g., [85, 92, 97, 125]. Thus, we believe there is little threat to construct validity.

3.5 Conclusion and Future Work

A large number of bug reports are submitted during the evolution of a software system. For a large system, locating the source code files responsible for a bug is tedious and expensive. Thus, there is a need to develop a technique that can automatically select these buggy files given a bug report. A number of bug localization tools have been proposed in recent years. However, the accuracy of these tools still need to be improved. In this chapter, we propose AmaLgam, a new method for locating relevant buggy files that combines historical data, similar report, and structural information to achieve a higher accuracy. We perform a large-scale experiments on four projects, namely AspectJ, Eclipse, SWT and ZXing to localize more than 3,000 bugs. Compared with the state-of-the-art bug localization approaches, BLUiR+ and BLUiR, our approach, on average, achieves 12.5% and 16.4% improvements in terms of mean average precision (MAP). Compared with BugLocator which considers similar reports, our approach, on average, achieves a 24.4% improvement in terms of MAP. Compared with a history-aware bug localization approach proposed by Sisman and Kak, our approach achieves a 46.1% improvement in terms of MAP.

In the future, we would like to reduce the threats to external validity further by applying our approach on more bug reports from various systems. We are also interested integration other bug prediction approaches to AmaLgam. Furthermore, we want to investigate different ways to combine the scores from the three components of AmaLgam. We are also interested to use Principal Component Analysis (PCA) to analyze which component (i.e., version history, similar report, or structure) contributes the most to the final results.

Chapter 4

Code Search via Topic-Enriched Dependence Graph Matching

4.1 Introduction

In many software projects, developing the first release of a system is often not the end. New features would need to be integrated; bugs would need to be fixed. It has been studied that the cost of maintenance activities is often much larger than the cost of developing the first release. Some studies estimate that the software maintenance cost is as high as 90% of the total software cost [18]. This is especially the case for legacy applications, which is often referred as "legacy crisis" [94].

In software maintenance activities, programmers need to modify a code base, either to fix bugs or to insert additional functionalities. Finding relevant code snippets is hard especially if it has been a long time since the code has been maintained, if the original developers have left the company, if the code base is large, or if the system is developed by many people and the relevant code snippets are dispersed in various locations.

To find relevant code fragments, a developer may employ a search utility that comes with their operating systems (*e.g.*, Windows Explorer), or their favorite IDEs (*e.g.*, Eclipse, Visual Studio). However, often the functionality provided by these

tools is not sufficient due to several reasons. First, these tools can only look for exact matches of strings, but a user might not know the exact query for the tools. Second, a code base is not simply text; it is text with structure and semantics; such information is implicit and these standard search tools are unable to capture user queries that involve dependence relations among various program elements. Third, these tools are not able to capture relevant code snippets that are spread in different program locations; one would need to do repeated search with various queries to find all answers.

To address these limitations, a number of studies have proposed the use of natural language processing techniques to find relevant code. Hill *et al.* [34] have proposed a novel approach that automatically extracts natural language phrases from source code identifers and categorizes the phrases and search results in a hierarchy. Marcus *et al.* [72] use information retrieval techniques to find concept locations in software. With natural language processing techniques, a user no longer need to specify exact identifier names for search.

There are also other studies that allow users to specify rich queries involving the specification of the dependencies among program elements of interest [32]. These dependencies are often expressed in terms of control and data dependencies. A recent approach by Wang et al. is also able to capture *transitive dependencies* [112]. This allows code snippets located in different classes to be retrieved if they match a user query. Their approach makes use of advances in graph query techniques in the database community.

Both kinds of approaches have their disadvantages. Natural language processing techniques are useful for specifying high-level concepts related to a piece of code, but cannot easily specify fine-grained relations among particular elements in the code. On the other hand, dependence based search can quantify relationships among code elements directly using graph pattern matching, but it often fails to distinguish code of different high-level concepts but with similar low-level relationship. As the result, both kinds of techniques may produce numerous false positives during

search.

In this work, we marry the use of natural language processing and graph query to realize a code search engine that supports expressive queries including natural language text and dependence relations. We also improve the speed of the current state-of-the-art transitive dependence code search engine tool proposed by Wang et al. [112] by an order of magnitude speedup.

To realize our solution we incorporate a topic modeling engine based on hierarchical Latent Dirichlet Allocation (hLDA) that is able to learn topics and assign them to various documents. We also incorporate recent advances in querying labelled graphs to alleviate the limitation of the code search engine proposed by Jin et al. [44]. Our framework accepts user code and processes the structure and the textual component of the code separately. We convert the user code into a program dependence graph (PDG) [96]. The textual part is converted into topics and these topics are assigned to the various nodes in the PDG to create an enriched PDG (ePDG). Finally, these ePDG are used by our graph query engine to process user queries. We propose a query language that allows a user to specify an expressive query including both free-form texts and dependence relations.

We have evaluated our solution on a number of software projects. We investigate the scalability of our solution and compare it with the approach of Wang et al. [112] that also handles transitive dependence queries. We investigate the utility of our approach by presenting some case studies on how our tool could be used in various settings. We demonstrate the power of both the textual component and the dependence component of our solution when they are used independently and together.

The contributions in this chapter are as follows:

1. We propose an engine that incorporates topic modeling for code search. A user could then search by topics rather than by keywords.

2. We integrate both semantic topic search and dependence search by allowing

users to specify free-form texts and transitive dependence relations among program elements of interest.

3. We improve the state-of-the-art code search tool and its graph matching algorithms [112] by one degree of magnitude speed up.

4. We have performed a preliminary study on the scalability of our tool on a number of cases and demonstrated the power of our search engine in improving the code search experience by supporting expressive queries.

The structure of this chapter is as follows. In Section 4.2, we describe preliminary information about program dependency graph, topic model and graph reachability. In Section 4.3, we describe our framework that combine and extends the building block into a holistic solution. In Section 4.4, we describe the detail about Dependence Query Language with Topic Modelling. Section 4.5, we elaborate our query processing engine. Section 4.6 describes our experiment and case studies. In Section 4.7 we conclude our work and point out some interesting future work we could focus on.

4.2 Preliminaries

In this section, we describe program dependence graph (PDG), topic modeling, and graph reachability queries. We use these concepts as as building blocks in our framework. We describe these concepts in layman terms.

4.2.1 Program Dependence Graph

A program dependence graph (PDG) captures all these dependencies, including call relations. Each node in the graph corresponds to a program element in the code. Each edge corresponds to either data or control dependence. Program dependence graph has been shown to represent certain semantic aspects of code and useful for various purposes [20, 63].

The corresponding PDG for the following sample code is shown in Figure 4.1. The solid arrows represent data dependencies among the nodes; the dashed lines represents control dependencies. As shown in Figure 4.1, we have several commonly used node types in the PDGs, *e.g.*, `call-site` for function calls, `control-point-if` for `if` conditions, `assignment`, `statement`, etc.

```
tag = MALLOC(sizeof(TAG));
if (!tag)
    return XML_ERROR_NO_MEMORY;
tag->buf = MALLOC(INIT_TAG_BUF_SIZE);
if (!tag->buf)
    return XML_ERROR_NO_MEMORY;
```

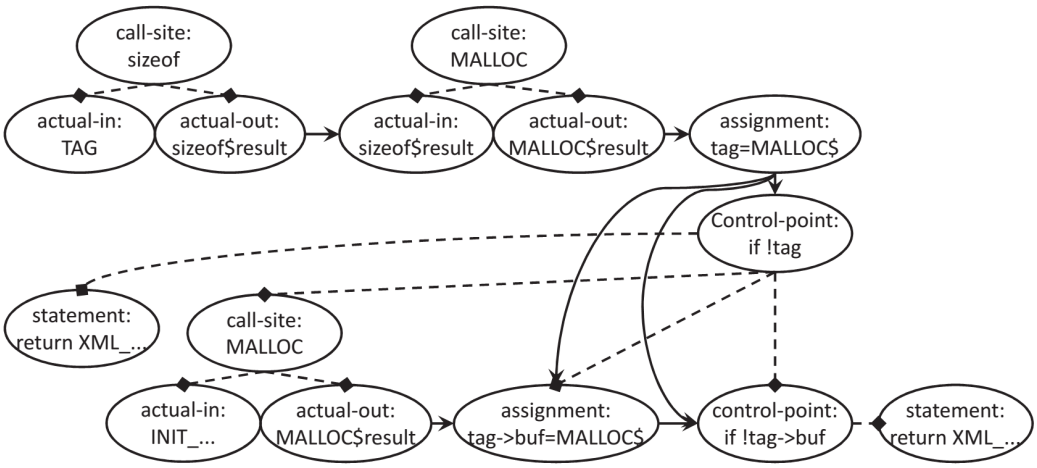

Figure 4.1: Sample PDG.

This piece of code actually contains a bug that may cause potential memory leaks. Section 4.6 uses the code as one of the sample maintenance tasks to evaluate our search engine.

There are a number of off-the-shelf tools that are able to automatically extract program dependence graphs from code. In this work, we use CodeSurfer provided by GrammaTech [23] to extract PDGs from C code.

4.2.2 Topic Modeling

Researchers in the data mining, machine learning, and natural language processing communities have proposed *topic modeling* as a way to discover semantic concepts (or topics) from natural language corpora. Given a collection of textual documents, the goal of topic modeling is to learn a set of topics, each being represented by a set of words, that characterize the various documents in the collection. Each document in the collection could then be assigned some probabilities of belonging to each topic.

In this work, we use Latent Dirichlet allocation (LDA), which takes as input a set of textual documents and the number of desired topics (50 by default), and outputs a model consisting of a set of topics where each topic is a set of related words. Then, the model can be used to associate each query document with a set of topics with various probabilities. We use the implementation provided by JGibbLDA [81].

4.2.3 Graph Reachability

Graph reachability is to compute whether two arbitrary nodes in a large (directed) graph are reachable from each other. A node is reachable from another node, if and only if there is a series of edges in the graph that would connect one node to the other. Such reachability queries can be used to perform graph pattern matching that allows *transitive*, instead of direct, connectivity among nodes in query inputs [14].

A naive algorithm to check reachability is to perform a breadth first search in the graph starting from one node until we reach the other node. This takes linear time in terms of node and edge size but would still be computationally expensive when we consider the reachability among every pair of nodes in the graph or when the reachability is further restricted by certain constraints on the nodes and the edges.

In this chapter, we use an efficient reachability algorithm [44] to answer an questions on whether any two nodes in an PDG are reachable via data or control dependence edges. Also, based on the idea in Cheng et al.'s work [14], we design

and implement an efficient way to use reachability queries for searching desired patterns in PDGs (*cf.* Section 4.5).

4.3 Proposed Framework

Our approach is composed of components for code processing, text processing, graph enrichment, and query processing. The relations among these components are shown in Figure 4.2.

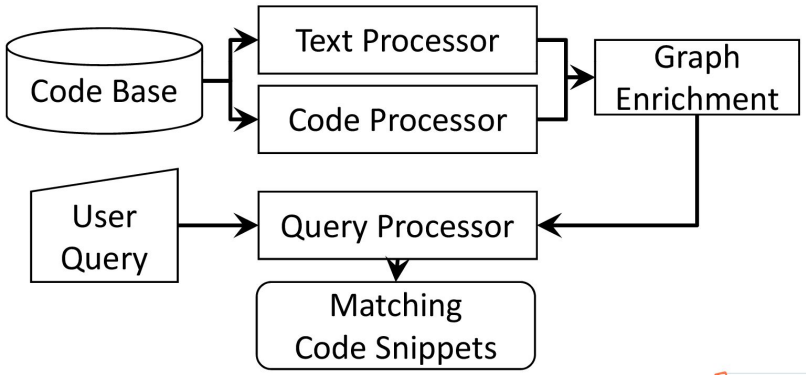

Figure 4.2: Our Code Search Overview.

Code and Text Processor Our code processor component takes in source code and outputs its corresponding Program Dependence Graph (PDG). We use CodeSurfer to generate PDGs [23]. The nodes in the PDGs are labeled with different program element types[1], and edges between nodes represent control or data dependencies.

Our text processor component takes in source code and outputs a topic model. The processor first changes each method in the code base into a bag of words. For each method, we take the name of the method, comments and identifiers in the method, split some names into word tokens, and remove some keywords and non-interesting words to form the bag of words. Each bag of words forms a document. A code base is thus transformed into a set of documents. These documents are then fed into LDA [81] to produce 50 different topics where each topic is represented by a set of words.

[1] We use the node types defined by CodeSurfer. There are 33 different node types for C/C++, e.g., function call, expression, etc.

Graph Enrichment Component. This component takes in the PDGs outputted by the code processor and the topics outputted by the text processor, and enriches the PDG nodes with the topics. After the enrichment process, each PDG node contains not only the type of the program element associated with it, but also a set of corresponding topics with their associated probabilities. For example, a node in the PDG could be associated with topic-1 with probability 0.4, topic-2 with probability 0.3, and topic-3 with probability 0.3. And all nodes belonging to the same method would be associated with the same set of topics and their associated probabilities.

Query Processing Engine. The query processor is the key block in our proposed approach. It takes in a user query and, based on the enriched PDGs, returns all the code snippets that match the query. Inside of the query processor, we first transform a user query in the format of DQL^t (*cf.* Section 4.4) into a graph representation, which is referred to as a *query graph* (*cf.* Section 4.5.1). Then, we search through the enriched PDGs for program elements that match the query graph and return search results (*cf.* Section 4.5.2). As this processor is rather complex, we describe it in the following two sections.

4.4 Query Language

To help users formulate queries and provide inputs for our code search engine, we extend the Dependence Query Language (DQL) proposed by Wang et al. [112] with semantic topics. We refer to the new language as Dependence Query Language with Topic Modeling (DQL^T). Its syntax is shown in Table 4.1.

DQL^T has five parts: topic declaration ($topic$), node declaration ($ndecl$), node description ($ndesc$), relationship description ($rdesc$), and targets ($target$). $Topic$ specifies the topics related to intended search results and can be any free-form text. $Ndecl$ declares node variables and their types. $Ndesc$ specifies constraints on declared node variables. $Rdesc$ specifies constraints on the relations among declared node variables. $Target$ specifies the variables specified in $ndecl$ that are desired

search targets. When a DQL^T query is processed on an enriched PDG, nodes in the ePDG that match the node variables specified in $target$ and satisfy the constraints specified in $ndecl$, $ndesc$ and $rdesc$ would be returned.

Node Declaration. This part of a query is to declare some node variables that will later be mapped to nodes in an ePDG. Each node variable can have one or more types (*i.e.*, a disjunction of types); each type can correspond to a kind of code in the code base (which is mostly in C) and is separated by '|' as defined by $tlist$. We consider six different types of node: $func$, var, $assgn$, $decl$, $ctrlPoint$, and $stmt$. A node of type $func$ corresponds to a function invocation in the code. var corresponds to a variable. $assgn$ corresponds to an assignment. $decl$ corresponds to a variable or function declaration. $ctrlPoint$ corresponds to a branching condition that affects code execution paths, *e.g.*, `if` or `while` conditions. $stmt$ corresponds to statements that change control flows, *e.g.*, `return`, `break`, etc.

Node Description. This part of the query specifies further constraints on declared node variables ($cond$ and $ucond$). To specify constraints, developers can use the following unary operators: $contains$, $inFile$, $inFunc$, $atLine$, $ofType$, and $ofControlType$. The operator $contains$ allow developers to specify that a particular node needs to contain a particular text. The operators $inFile$ and $inFunc$ allow developers to specify a node that is located inside a particular file or function respectively. The operators $ofType$ and $ofControlType$ allow developers to specify a node of a particular type or to be a control node of a particular type (i.e., for, while, switch, or if).

Different from DQL, DQL^T has a topic operator ($similarTopicAs$). With this operator one can specify the topics each node should belong to. Topic modeling makes it possible to use free-form text in the topic operator as topic modeling techniques can handle imprecise textual queries and relate different words that have similar meanings.

Relationship Description. This part of the query specifies constraints governing

$query$	::=	($topic$)*; ($ndecl$)*; ($ndesc$)*; ($rdesc$)*; $target$;
$topic$	::=	$string$
$ndecl$	::=	$tlist\ id$
$tlist$	::=	$tlist$ '|' $type$ | $type$
$type$	::=	$func$ | var | $assgn$ | $decl$ | $ctrlPoint$ | $stmt$
$ndesc$	::=	id ($cond$)*
$cond$	::=	[not] $ucond$ | similarTopicAs $string$
$ucond$	::=	contains $string$ | inFile $string$ |
		inFunc $string$ | atLine $number$ |
		ofType $string$ | ofControlType $ctype$
$ctype$	::=	for | while | switch | if
$rdesc$	::=	$id\ op\ id$
op	::=	[oneStep] $dependOp$ | $textOp$ | $structOp$
$dependOp$	::=	dataDepends | controls | calls
$textOp$	::=	contains
$structOp$	::=	isFieldOf | isElementOf
$target$	::=	(id)*
id	::=	$string$
$string$	::=	(A-Z,a-z,0-9)+
$number$	::=	(0-9)+

Table 4.1: DQL^T Syntax

the relationships between two declared node variables. We have three types of relationships: dependence operator ($dependOp$), textual operator ($textOp$), and structural operator ($structOp$).

Dependence operators are used to specify either data dependence, control dependence, or transitive call relationship (i.e., there is a chain of function invocation from one node to another in the input PDG). They are expressed as operators: $dataDepends$, $controls$, and $calls$, respectively. Textual operators are used to specify that the textual content of one node contains (i.e., is a super string) of that of the other. Structural operators are used to specify that one node is a field of another ($isFieldOf$), or is an element of another ($isElementOf$). For example, "$a.b\ isFieldOf\ a$" holds, and "$a[b]\ isElementOf\ a$" also holds.

Targets. This part of a DQL^T query specifies the target node variables that would be returned as the output of the query. This set of variables is a subset of all declared variables. The declared variables would be matched to nodes in PDG, but only those specified as a target node variable would be returned. The other nodes serve as context for more accurate locating of the target nodes. Currently, the output of our code search engine is a set of line numbers linked to the locations of matching code fragments.

4.5 Query Processing Engine

Taking a user query and topic-enriched PDGs as input, our query processing engine goes through two major steps to produce search results: query graph construction and graph matching. Figure 4.3 illustrates the relations among the components in the query processing engine, which is a zoom-in of the `Query Processor` in Figure 4.2. Query graph construction converts a textual query into a graph representation. Graph matching locates nodes and edges in the input program dependence graphs that match the query graph and composes them into search answers. We describe more details of the query processing engine in the following sub-sections.

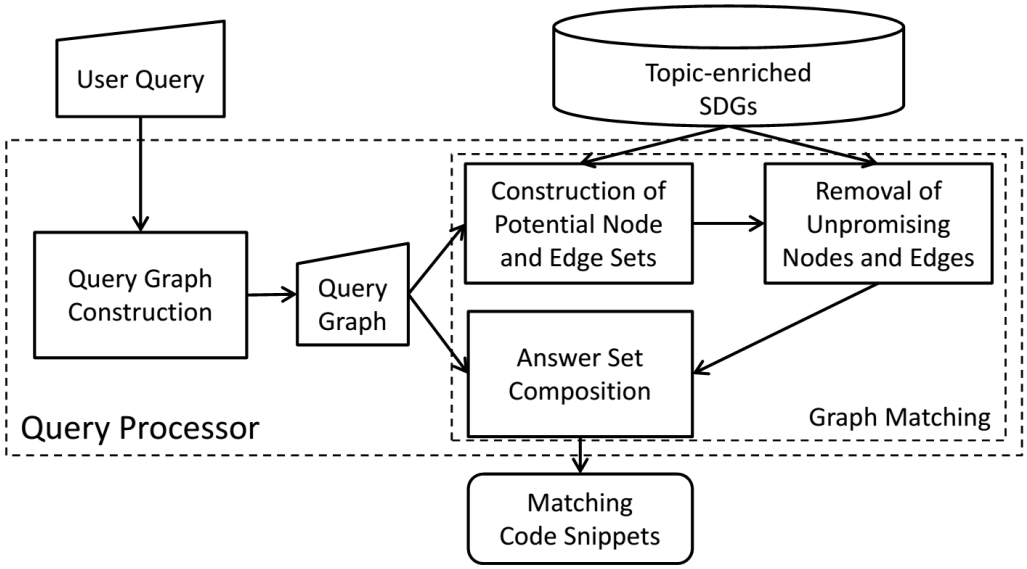

Figure 4.3: Components in Query Processor.

4.5.1 Query Graph Construction

A user query can consist of node declarations, node descriptions, relationship descriptions, and the identifiers of the nodes of interest. We form a graph from this textual description. Our DQL^T language syntax aims to make it intuitive for developers to specify their queries. The intuitive syntax may not directly correspond to the underlying program dependence graphs returned by CodeSurfer [23]. The purpose of the query graph construction step is to transform higher level DQL^T queries into a more low level query graph with nodes, edges, and their labels matching the PDG representations.

There are three kinds of basic node relations in user queries: N_1 calls N_2, N_1 controls N_2, and N_1 dataDepends N_2. We describe the constraints on the node types for each kind of node relations expressed in user queries and how the relations are converted into query graphs as follows:

1. N_1 **calls** N_2**.** For this kind of node relation, both N_1 and N_2 must be of type *function* in DQL^T for a query expression to be valid. This query is converted into a query graph containing two nodes as shown in Figure 4.4(a).

2. N_1 **controls** N_2**.** For this kind of node relation, N_1 must be of type *controlPoint*, while N_2 can be of any one of the types: *function*, *assignment*, *con-*

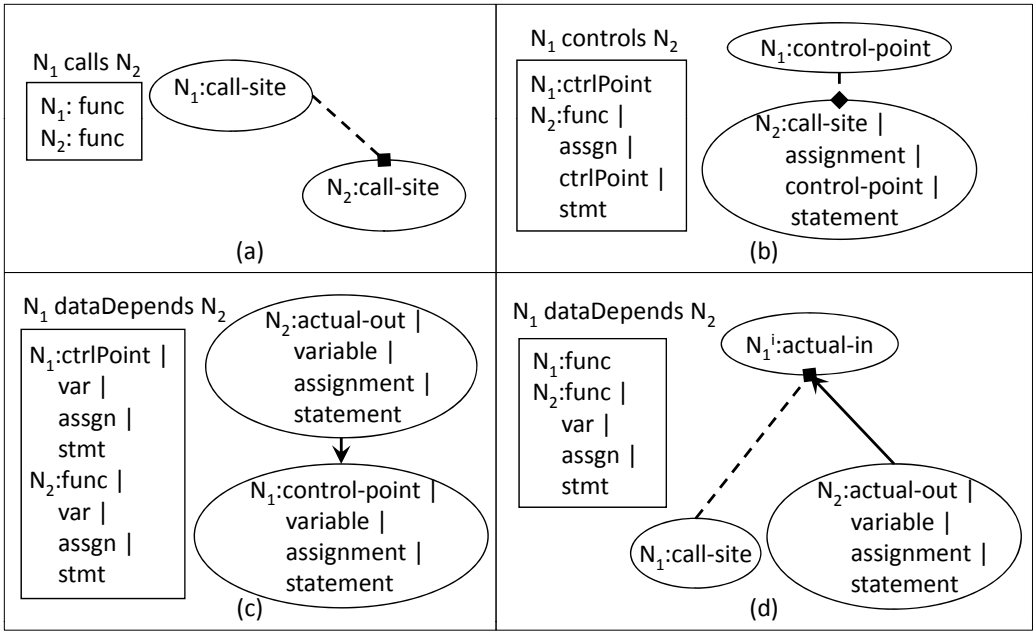

Figure 4.4: Converting User Queries to Query Graphs. We use the "|" separator to specify the different node types that could be assigned to N_1 and N_2. The correspondences between the node types in a user query and those in the query graph are shown.

trolPoint, or *statement*. This query is converted into a query graph containing two nodes as shown in Figure 4.4(b).

3. N_1 **dataDepends** N_2**.** For this kind of node relation, N_1 and N_2 can be of any type except *declaration*. Figure 4.4(c) shows how the query is converted into a query graph when N_1 is not a function. Figure 4.4(d) shows the query graph formed when N_2 is a function.

4.5.2 Graph Matching

After the above step is done, we have a small query graph and a large enriched PDG. We want to match the small graph against the big graph.

This task is computationally expensive, especially when we want to enable transitive (in addition to *oneStep*) dependence queries. Given a query graph consisting of n nodes, a naive approach would be to enumerate all possible n-node subgraphs of the enriched PDG and check if the constraints specified in the query is satisfied

by each of the subgraph. This would incur a very high cost which is not practical for our purpose.

To tackle this challenging graph matching problem, we perform some heuristics to exclude nodes and edges that could not be part of a search result, and break down the transitive dependence queries into a number of graph reachability queries so that the state-of-the-art technique in graph reachability can be used to process these queries.

Our graph matching algorithm is mainly comprised of the three components previously shown in Figure 4.3. Its high level description is as follows. We first identify the nodes in the enriched PDG that potentially match each node in the query graph. We refer to the set of potentially matching nodes as *potential node set (PNS)*, and denote the PNS for a node n in the query graph as PNS_n. We then identify the set of edges that connect nodes in PNSs and potentially satisfy the constraints specified in the query graph. We refer to the set of the edges as *potential edge set (PES)*. We keep these PNSs and PESs in a data structure we refer to as *work list*. We further *filter* away unpromising nodes and edges in PNSs and PESs that could not be part of a search result. Finally, we *compose* answer sets by considering all combinations of the nodes and edges in PNSs and PESs. More detailed descriptions are as follows.

A) Work List Construction

We take in a query graph and topic-enriched PDGs and convert them into a work list. This is done by converting each node n and edge e in the query graph to the corresponding PNS_n and PES_e.

PNS Construction. Each node in a DQL^T query could be specified with its type, textual description, syntactic restriction, and semantic topics, as defined by $tlist$, $textOp$, $ucond$, $cond$, etc. in Table 4.1. These constraints are called *unit conditions* and *topic conditions* of a query. The PNS_n for a node n is obtained by getting the set of nodes satisfying these conditions:

- For each topic condition for n, we get the top-k methods of that topic (in terms of probabilities of methods belonging to that topic as returned by a topic modeling technique). The intersection of the k-method sets of all topic conditions forms the initial set of methods of interest.

- If no topic condition is set for n, we simply consider all methods in the code base as the initial set of methods of interest.

- For each node n' in the PDGs corresponding to each of the method in the initial set, check whether n' satisfies all unit conditions set for n. The set of nodes from the initial method set satisfying all of the unit conditions becomes the PNS_n.

PES Construction. We construct one PES for every edge e in the query graph, denoted PES_e. Each PES_e is the set of edges in PDG that connect two nodes in two different PNSs whose corresponding nodes are connected in the query graph. The procedure of constructing PESs is shown in Figure 4.5.

```
Procedure ConstructPES
1: for each edge e=(N1,N2) in the query graph
2:     for each node n1 in PNS_N1
3:         for each node n2 in PNS_N2
4:             if n2 is reachable from n1
5:                 Add (n1,n2) into PES_e
```

Figure 4.5: PES Construction Process

Line 4 in Figure 4.5 is the critical step. If a $oneStep$ (or direct) data or control dependence is specified, we simply check if n_2 is connected to n_1 in the PDG directly. If normal (transitive) data or control dependence is specified, we employ the graph reachability indexing algorithm described in Section 4.2 for efficient checking.

B) Node & Edge Filtering

In this step, we filter away nodes and edges that can not possibly be part of a search result. We call such nodes and edges *unpromising*. The purpose of the filter is

to reduce unnecessary checks when composing PNSs and PESs into search results.

Properties 1 and 2 specify rules for removing unpromising nodes and edges.

Property 1 (Unpromising Node). *Let N_1 be a node in the query graph and $\{E_1, \ldots, E_k\}$ be the edges connected to N_1. All nodes in the PNS_{N_1} and the following set are unpromising: $\{n \mid \exists i \, \forall e \in PES_{E_i} \cdot n \notin e\}$.*

Proof. By definition, in order to match the node N_1, a node n in PNS_{N_1} should connect to at least k edges that could match every E_i. That is, n should connect to at least one edge from each of PES_{E_i}. Thus, n could be promising only if it is not in the above set. □

Property 2 (Unpromising Edge). *If an edge starts from or ends with an unpromising node, then it is unpromising.*

The two properties could then be applied to filter nodes and edges from PNSs and PESs. Notice that we could iteratively apply the two properties one after the other: when more unpromising nodes are identified via Property 1, more edges could be pruned by applying Property 2; the application of Property 2 may then help to identify more unpromising nodes when Property 1 is re-applied. The algorithm to do this is shown in Figure 4.6.

```
Procedure UnpromisingRemoval
1: noChange = false;
2: while (!noChange)
3:   for each query node n,
4:     Remove unpromising nodes from PNS_n (Prop 1)
5:   Remove unpromising edges from PESs via (Prop 2)
6:   if (no node or edge is removed)
7:     noChange = true;
8: if (any PNS_n or PES_e is empty)
9:    Terminate search
```

Figure 4.6: UnpromisingRemoval Procedure

Property 3 (Termination). `UnpromisingRemoval` *procedure always terminates.*

Proof. Each time we complete an iteration of the while loop (Lines 3-6), either one of the following conditions hold:

1. The total size of nodes and edges in PNSs and PESs is reduced;

2. The total size of nodes and edges is unchanged. We have reached a fix point.

Since there is a finite size of nodes and edges in PNSs and PESs, there is only a finite number of times condition 1) above would hold. Eventually condition 2) or a fix point happens and the procedure would terminate. □

Property 4 (Completeness). *Each node remaining at the end of* `UnpromisingRemoval` *procedure is promising.*

Proof. Let assume for contradiction that one remaining node n is unpromising. This would mean that n could be grown, with its connected neighbor nodes, into a true subgraph of the query graph that *cannot* match the *whole* query graph. Then, this can only because 1) some query node n' has an empty PNS, or 2) some query edge has an empty PES, or 3) some node reachable from n in the PDG has no sufficient edges to connect to nodes in the PNS of some other query node. Case 1) and 2) would terminate the code search; case 3) would require Property 1 to be applied which should not happen if `UnpromisingRemoval` has terminated. This is a contradiction. By proof by contradiction, this property must hold, *i.e.*, all nodes remaining at the end of `UnpromisingRemoval` procedure must be promising. □

C) Answer Set Composition

Based on Property 4, each node in the PNSs at the end of the filtering step is part of one or more search answers. There are potentially many answers to a query based on the sizes of the PNSs and the available PESs. In this last step of our approach, we compose nodes and edges in PNSs and PESs to form search answers.

The procedure is shown in Figure 4.7. The main idea to take every possible combination of the nodes from the PNSs and see whether each of the combination could match edges in the PESs. If the nodes in one combination match all PESs, the combination of the nodes will be saved and translated back to code snippets as search results. This translation works straightforwardly: when the $target$ nodes are specified in the user query, only the code snippets that match them would be returned to users, otherwise, all code snippets matching the nodes in the saved combination would be returned.

The computational complexity of this step is linear to the size of every PNS (*i.e.*, potential number of matching code snippets) and linear to the number of PESs (*i.e.*, the number of relations specified in the user query). Although it could become expensive in general, the queries for our code search purpose are often small, and PNSs are also small due to the applications of semantic topic matching and Property 1 and 2, and thus we expect acceptable search performance. Through empirical studies, Section 4.6.1 demonstrates that our code search engine can be efficient for practical uses.

```
Procedure AnswerComposition
Input: PNSs and PESs from the filtering step
1. for each query node N_i
2.     Take one node n_i from PNS_{N_i};
3. for each query edge E_i = (N_1, N_2)
4.     if (PES_{E_i} does not contain (n_1, n_2))
5.         Discard this combination, Goto 7;
6. Save this combination;
7. Repeat 1 until all combinations are checked;
8. Convert every comb. into code snippets as search results.
```

Figure 4.7: Answer Set Composition

4.6 Empirical Evaluation

In this section we first describe our scalability evaluation. We then present four case studies to show the utility of our topic-enriched dependence search tool on a number of software engineering tasks on various systems.

Table 4.2: Performance Comparison (in seconds)

Dataset	Approach	TP	TQ	TRF	TT
Expat_v32	[112]	1.7	64.7	2.1	68.5
	ours	0.3	6.0	0	57.3
Expat_v38	[112]	0.9	30.4	1.1	31.5
	ours	0.8	0.5	0	1.3
GPSbabel_v1071	[112]	3.3	640.2	3.6	647.1
	ours	0.6	1.0	0	13.2
GPSbabel_v1200	[112]	3.3	265.2	0.9	289.4
	ours	0.7	0.1	0	0.81

4.6.1 Scalability Evaluation

We compare the speed of our processing engine with the previous engine proposed by Wang *et al.* [112]. We use the dataset studied in their paper for comparison. The results of the evaluation are shown in Table 4.2. Column "TP" represents the time consumed for pruning unpromising nodes and edges before actual search. Column "TQ" represents the time for performing queries. In the case of Wang et al. [112], the column also includes the time spent on building graph indices for each query, while our approach performs one-time indexing and just needs to load the index when required by a query. Column "TRF" represents the time of post-filtering on search results which is not applicable to our approach. The last column "TT" is the total time for the whole query process, including all necessary graph indexing costs. From Table 4.2, we notice that on average our approach is more than 130 times faster.

4.6.2 Case Studies

We next describe some case studies showing the utility of our code search engine.

Task One

The first searching task is from the Expat project version 2002-05-17 which is a C language project. The task was also used in Wang et al.' work [112]. One comment of the code change mentioned failed `MALLOC()` calls that can cause potential memory leaks. The code before the change is shown below:

```
tag = MALLOC(sizeof(TAG));
if (!tag)
   return XML_ERROR_NO_MEMORY;
tag->buf = MALLOC(INIT_TAG_BUF_SIZE);
if (!tag->buf)
   return XML_ERROR_NO_MEMORY;
```

The code after the change is shown as below, with an added call to `free`:

```
tag = MALLOC(sizeof(TAG));
if (!tag)
    return XML_ERROR_NO_MEMORY;
tag->buf = MALLOC(INIT_TAG_BUF_SIZE);
if (!tag->buf){
    free(tag);
    return XML_ERROR_NO_MEMORY;
}
```

Our task is to find all similar code that may return without freeing some memory. A sample query for this purpose in our DQL^T syntax is as follows:

Topics: *malloc*;

Nodes: *func A, func B, var C, var D, ctrlPoint E ofControlType if, stmt F return*;

Relations: *C dataDepends A, D dataDepends B, D isFieldOf C, E dataDepends D, E oneStep controls F*;

Targets: *E*

In the query, the topic is $malloc$. A corresponds to a function call to `MALLOC` whose return value is used to initialize a variable C; another variable D, a field of C, is assigned the value from another call and used in a control point E, and one of E's branch may call `return`.

In this task, the developers actually changed two places in the program. Our

approach finds both of the places, plus four other places, which are false positives, while Wang's approach [112] finds the two target places but brings 36 false positive. Our approach provides better precision.

Among the four false positives (*i.e.*, the four places not changed by the developers), one actually missed a call to `free` too; the developers missed the place when fixing the memory leak bugs. This also suggests our code search engine can be useful for detecting inconsistent changes to help reduce bugs.

Task Two

The second searching task is from the $GPSbabel$ project version 2004-10-27, which was also used in [112]. The comment of the code change is "*Aggressively replace open-coded strncpy for space padded strings in various waypoint send functions*." The submitted code changes demonstrate that the developers try to find all the code elements that are in "waypoint send" functions (a list of functions whose names end with "send") and are like the italicized part in the following code portion.

```
UC* p,str; ...
for(i = 0; i < 10; i++){
    str[i] = *p++;
}
```

Such code is contained in functions whose names end with "send" and has the following patterns: a loop (either `for` or `while`) contains an assignment, and the data used in the assignment depends on a pointer declaration. Therefore, we have the user query as below:

Topics: *send*;

Nodes: *var A ofType "UC*", ctrlPoint B ofControlType* (for|while) *inFunc "send", assgn C*;

Relations: *B oneStep controls C, A oneStep dataDepends C*;

Targets: *B*

In this task, there are actually 37 targets in the code. Our approach finds all the

37 targets together with 25 false positives, while the approach proposed by Wang *et al.* [112] also finds all the targets with 25 false positives. Most of the false positives are also all in other kinds of "send" functions (besides "waypoint") that contain similar loops.

Task Three

The third task is from *Media Player Classic–Home Cinema* version 1639. The comment of the code change is to fix the memory leak in decoder initialization functions. The aim is to find all the memory leaks that are caused by `returns` before freeing relevant memory storing structure variables same as the previous one. An example is as below.

```
Decoder_amrState * s; ...
if ( ( s->lsfState = ( D_plsfState * )
        malloc( sizeof( D_plsfState ) ) ) == NULL ) {
    fprintf( stderr, ... );
    return -1;
}
```

The code may cause memory leak since it does not free the memory of occupied by the variable `s` and its pointer fields other than `lsfState`. In this code, the assignment `s->lsfState = ...malloc...` has one-step data dependency on `s` and the return value of `malloc`. The control point (`if (...== NULL)`) has one-step data dependency on the assignment, and controls `fprintf` and the `return` statement. The topics for the changes codes are $decoder$ and $init$ according to the change comment. Therefore, our user query is as below.

Topics: *decoder, init*;

Nodes: *var A, func B contains "malloc", ctrlPoint C ofControlType if, stmt D return, assgn E, var X*;

Relations: *X isFieldOf A, X oneStep dataDepends E, E oneStep dataDepends B, C oneStep dataDepends X, C oneStep controls D*;

Targets: *C*

In this query, $isFieldOf$ is again used to relate a field to its containing structure. C is the $target$ because the developer wants to locate only the branches that may need calls to `free`.

In this task, there are 10 targets in the code, and our approach finds all the targets with one false positive. If no topic is used for search, the search based on dependencies only finds all the targets, but brings 14 false positives. If the search is given a constraint that the name of the involved functions should contain both `decoder` and `init` without using the topics, one target is missed, since this target is in a function `Post_Filter_init` whose name does not contain "decoder". Search with topics can find this target since this function has latent semantic relationship with "decoder" even though its name does not contain "decoder".

Task Four

The fourth task is from $Notepad$++ $Plugin$ project version 1.9. The developers aim to fix memory leaks related to uploading. In the following sample code, when the call to `CreateThread` fails, the program will return without freeing the memory allocated before.

```
if (CreateThread(NULL, 0, doUpload,
        lt, NULL, &id) == NULL) {
    threadError("doUpload");
    busy = false;
}
```

The correct code should contain calls to `delete lt` in the branch where `lt` is allocated before the call. The topic involved is $upload$, and only two relations are needed: one is the data dependence between the return value of `CreateThread` and the `if` condition; the other is the control dependence between the `if` condition and the error handling function `threadError`. So the query can be defined as below.

Topics: *upload*;

Nodes: *ctrlPoint A ofControlType if, func B contains "CreateThread", func C contains "threadError"*;

Relations: *A oneStep dataDepends B, A oneStep controls C*;

Targets: *A*

In this task, our approach can find all the three targets together with three false positives. The search without topic can also find the three targets, but bring 11 false positives. The reason why our approach brings 3 false positives is that the function `WindowProcedure` calls other upload related functions and thus becomes a function involving the topic of "upload" and at the same time it calls `CreateThread` with empty parameters (*e.g.*,`CreateThread(NULL, 0, doRenameDirectory, NULL, NULL, &id) == NULL`) where "delete" is not needed.

4.7 Conclusion and Future Work

We propose a code search approach that utilizes both of the topics and the dependence relations in a code base for more accurate results. Our approach provides a query language that allows users to specify both free-form topics and complex dependence relations. We design and implement algorithms that can utilize topic modeling techniques to refine dependence graph queries. As a result, our approach can not only return more relevant search answers, but also is more scalable for processing transitive dependence queries, achieving on average an order of magnitude speedup than a previous state-of-the-art approach. We have evaluated our tool on several software maintenance tasks in various software systems and demonstrated the utility of our code search engine.

In the future, information besides topic and dependence that is related to the code (*e.g.*, email discussion records, programmer activity patterns, project-specific patterns) may be incorporated into the code search engine to help make it more accurate.

Chapter 5

AutoQuery: Automatic Construction of Dependency Queries for Code Search

5.1 Introduction

Source code could be viewed more than just text. It also contains structures and dependency relations among program elements. To leverage these dependencies to improve search accuracy, dependence-based code search techniques have been proposed. They accept queries expressed as dependency relationships among program elements of interest [106, 112], and return code fragments whose constituent program elements satisfy the dependency relationships. It has been shown that dependence-based code search can outperform text-based code search [106].

However, there is one drawback that potentially hampers the usage of dependence-based code search. Often it is hard to construct dependency queries. Users need to be able to visualize the dependency relationships among program elements, select relevant ones and express these as queries. This process might be daunting for many potential users. In this work, we aim to address the drawback of dependence-based code search by automatically constructing dependency queries

from code examples. Using our tool, developers could input a set of example code fragments, which correspond to snippets of code that users need to change to address a particular need (e.g., new feature implementation, bug fix, etc.). By taking multiple code fragments as input, our tool can learn important dependencies shared by the examples and filter unimportant dependencies that are peculiar to an individual example. Our tool eventually constructs a dependency query, which can then be used to identify other code fragments that need to be changed in a similar way (to address the same need) by leveraging a dependence-based code search tool. Of course, if needed, users can make changes to the dependency query before inputting them to the code search tool. Our setting supports automation while still allowing users to be in control in the code search process and thus the user's domain knowledge can be leveraged for effective code search.

We propose a tool named AutoQuery. Our tool first converts a given set of code fragments (not necessarily compilable) into program dependence graphs (PDGs) (see Section 4.2.1 for detail). These program dependence graphs are then analyzed and their commonalities are highlighted. We develop a new graph mining solution that could mine for these commonalities from the PDGs expressed as multi-labeled graphs with textual and node type labels. The resulting mined sub-graph is then converted to a dependency query. We use the dependency search tool proposed by Wang et al. as the backend code search tool [112]. AutoQuery builds upon this dependence-based code search tool by automating the process of generating dependence queries from sample code fragments. Previously, users of the dependence-based code search tool needed to manually construct dependence queries.

Our tool can help software engineers in various scenarios. For illustration purpose, consider Alex a software engineer who is responsible for performing a corrective maintenance task that may affect a number of files. Alex has localized two buggy code fragments. However, it is likely that there are many other buggy code fragments in other source code files that need to be fixed in a similar way. AutoQuery can help Alex find the remaining buggy code fragments. Alex simply needs

to input the buggy code fragments that he has localized into AutoQuery, and AutoQuery will in turn construct a DQL query (see detail in Section 4.4) and invoke an underlying code search tool to return the other buggy code fragments. Alex can save much time since he does not need to search for the buggy code fragments manually.

There are several challenges that we need to solve to build AutoQuery. First, the code fragments are not necessarily compilable. Many tools that construct PDGs from code, e.g., CodeSurfer [23], require compilable code. Thus, we need to process code fragments into compilable code units. Next, most graph mining solutions, e.g., [118, 119, 126], only work on simple graphs whose nodes and edges are labelled with simple types. PDGs are not simple graphs; each node in a PDG is a program element and contains not only node type information but also textual contents describing the fragment of the source code corresponding to the program element. Thus we need to build a new graph mining solution that handles our special graph representation that captures information in a PDG.

We evaluate our query generation approach on 47 realistic code search examples that we have extracted from the repositories of four software projects (Apache Http Server, Inkscape, Apache Subversion, and Libmpeg2). We show that using AutoQuery we can generate good queries that could be used to retrieve relevant code with precision, recall, and F-measure of 68.4%, 72.1%, and 70.2% respectively. We have also conducted a user study to compare automatically generated queries with manually generated queries. We find that our automatically generated queries can perform as well as human generated queries.

Our contributions in this chapter are as follows:

1. We are the first to propose an approach that can automatically generate dependency queries from several code examples.

2. We generate PDGs from non-compilable code fragments and propose a new graph mining technique that can search for common substructures in specialized multi-label graphs with nodes containing both node type information and

textual contents.

3. We have evaluated our approach on 47 realistic code search scenarios. We show that AutoQuery plus a dependence based code search tool can return relevant codes with precision, recall, and F-measure of 68.4%, 72.1%, and 70.2%. We have performed a user study that shows that our generated queries are comparably as good as human generated queries in returning relevant code.

The structure of this chapter is as follows. In Section 5.2, we describe our overall framework at high level. We zoom into the PDG generation engine of our framework in Section 5.3. We elaborate our query generation engine of our framework in Section 5.4. We present our experiments that evaluate the effectiveness of AutoQuery in Section 5.5. We conclude and mention future work in Section 5.6.

5.2 Overall Framework

In this section, we present the overall framework of our automatic query generation approach. In Section 5.3 and Section 5.4, we elaborate on the core components of our approach.

The structure of our automatic query generation approach is shown in Figure 5.1. It consists of two major processing components (shown as big rounded rectangles) namely PDGs generation engine and query generation engine. The outputs of the PDGs generation engine, which are the PDGs of the code fragments given by the user, is feeded to the query generation engine. The sequence of interactions among the user and the various components of the framework is described in the following paragraphs.

First, a user provides several code snippets to the PDGs generation engine. Inside this engine, the non-compilable code fragments are extended to compilable code by adding missing type definitions, missing variables, and missing methods.

We support non-compilable code as users could find examples online (e.g., from question and answer sites, from software forums, etc.); these examples are often only code fragments that cannot be compiled by themselves. Next, the PDGs generation engine uses CodeSurfer [23] to convert the compilable code to their corresponding program dependence graphs. Each PDG node corresponds to a program element and contains two pieces of information: node type, and textual representation of the program element in the source code.

Next, the PDGs generated by the PDGs generation engine is sent to the query generation engine. The query generation engine first transforms the PDGs to simple graphs by dropping the textual representation information from the nodes. A maximal common subgraph is then mined from these simple graphs using an existing graph mining algorithm [79]. Then, each node in the mined simple subgraph is mapped to its original node in the PDGs generated by the PDG generation engine, to extract the corresponding textual representation information. A node in a subgraph could be mapped to many nodes in the original PDGs. We develop heuristics to choose the most appropriate one. After getting a common subgraph and text information for each node in the subgraph, the engine converts the subgraph to a dependency query acceptable to the dependence-based code search tool developed by Wang et al. [112] (see Chapter 4).

5.3 PDGs Generation Engine

In this section, we present the steps to generate PDGs for given code fragments (or incomplete lines of code). Code fragments given by a user might not be compilable due to various reasons, e.g., missing type definitions, missing method declarations, etc. Many examples, especially those available in software forums and question and answer sites, are in this format. In this work, we are able to add the missing type definitions, variable declarations, and function definitions to make a non-compilable code fragment compilable. There have been studies in literature (e.g.,[43, 102]) that

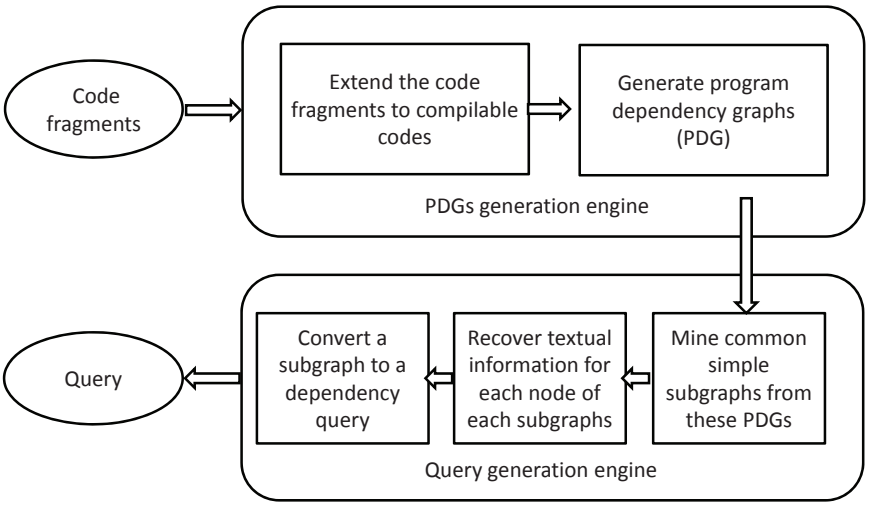

Figure 5.1: Overall Framework

make incomplete code compilable, but our goal is not to preserve the full semantics of the code; rather, we just would like to preserve the dependencies among the various program elements in the code fragment. Our implementation and evaluation focuses on C/C++ code.

Table 5.1: Inference Heuristics

Name	Heuristic
Assignment	If two expressions are on the two sides of an assignment, then they are of the same type.
Operator	If two expressions are connected with an operator in an expression, then they are of the same type.
Switch-case condition	If two expressions are either the condition expression of a switch statement or the expression in a case statement of the same switch statement, then they are of the same type.
Function definition and invocation	A parameter of a function must be of the same type as the corresponding argument of its invocation. The return type of a function must be of the same type as the corresponding return value of its invocation.

5.3.1 Code Extension

Algorithm 1 Code Extension Algorithm

1: **CodeExtension**(RPT)
2: **Input:**
3: RPT: Root of the parse tree of a code fragment
4: **Output:** Extended compilable code
5: **Method:**
6: Initialize DV, UV, UF, UT, EX, RV to {}
7: `Traverse`(RPT, DV, UV, UF, UT, EX ,RV)
8: Let MAP = Mappings from m in ($DV \bigcup UV \bigcup EX$) to its type
9: **repeat**
10: **if** s_1 and s_2 in MAP in a type-equivalence relation in RV **then**
11: Union s_1 and s_2 into MAP
12: **end if**
13: **until** No more change in MAP
14: Replace `Unknown` type in MAP with int
15: Add necessary (global) variable declarations based on UV
16: Add necessary function definitions based on UF
17: Add necessary classes based on UT
18: Add other necessary code (e.g., include statements, etc.)

To infer the types of variables and signatures of invoked functions in a code

fragment, we first obtain the parse tree of the code fragment. We create this parse tree using the python library pycparser[1]. From this parse tree we infer the types of undeclared variables and the signatures of invoked functions (including information about their parameters and return types). Undefined types appearing in the code fragment are inferred as well. We infer relevant data fields and functions of an undefined type that are used in the code fragment. After the above information is gathered, we extend the code fragment by adding:

1. Declarations of undeclared variables
2. Definitions of undefined functions
3. New classes (data types) that specify undefined types

Our code extension algorithm is shown in Algorithm 1. It takes a parse tree as an input and outputs a piece of compilable code. The major task of the algorithm is to infer a list of undeclared/undefined variables, function arguments, and function return values along with their inferred types, a list of undefined functions along with their signatures, and a list of undefined types along with their relevant attributes and functions.

We first initialize several sets to store a list of declared variables and constants (i.e., DV), a list of undeclared/undefined variables, function arguments, and return values (i.e., UV), a list of undefined functions (i.e., UF), a list of undefined types (i.e., UT), a list of expressions (i.e., EX), and a list of type-equivalence relations among variables and expressions (i.e., RV) (Line 6). Note that each expression could be of various types, e.g., function invocation, operator, variable reference, etc. The heuristics in Table 5.1 are used to infer type-equivalence relations and update the RV list. For example, a variable/expression on the left side of an assignment has an "assignment" relationship with a variable/expression on the right side of the assignment. We then update, add or remove items into these lists by

[1]https://bitbucket.org/eliben/pycparser

traversing the parse tree using the procedure `Traverse` defined in Algorithm 2 (Line 7). The `Traverse` procedure walks each node in the parse tree to discover (un)declared/(un)defined variables, expressions, functions, types and store them in corresponding lists. During the traversal, all the type-equivalence relations between the variables and expressions are stored for type inference later. We will elaborate the procedure `Traverse` in later paragraphs. After the lists are initialized, we create a set MAP that contains mappings from a set of variables, arguments, return values, and expressions to their type (Line 8). MAP is initialized as follows:

$$MAP = \{\{m\} \mapsto \texttt{m's type} | m \in DV\} \bigcup \{\{m\} \mapsto Unknown | m \in UV\} \bigcup \{\{e\} \mapsto Unknown | e \in EX\} \quad (5.1)$$

For each member of DV, we have its type, and thus we can insert a mapping between itself and its type in MAP. For each member of UV, initially we set its type in MAP to $Unknown$. Next, we try to identify variables, arguments, return types, and expressions stored in MAP that have the same type (Lines 9-13). If two members of MAP are in a type-equivalence relation, we combine their corresponding mappings in MAP (Lines 10-12). Note that we consider two members, which have a type-equivalence relation if there is a type-equivalence relation between two elements from each member (i.e., s_1, s_2), as a type-equivalence. We perform this unification iteratively by subsequent application of the inference heuristics [78]. We try to put all variables of the same type together and infer the types of undeclared/undefined variables, arguments, and return values based on variables of known types. As the unification proceeds, the number of remaining mappings decreases. The unification process ends when the number of mappings does not change anymore (i.e., a fixed point is reached) (Line 13). Note that this fixed point is guaranteed to eventually be reached since every time line 11 is executed, we reduce the number of mappings in MAP by one. Since there is a finite number of mappings in MAP, the repeat-until structure (Lines 9-14) must eventually terminate. At the end of the process, we have inferred equivalence classes of variables, arguments, and return

values that must be of the same type. For some of these equivalence classes, we can infer the type since one member of the class is defined in the code fragment. For other equivalence classes, the type is still `Unknown`. For these equivalence classes, we replace `Unknown` with int (Line 14). This operation (replacing `Unknown` with int) does not affect the compilability of the resulting code as there is no conflicting type assignment. Please note that we are only interested in recovering dependency relationships among program elements in the code fragment and not the full semantics of the code fragment.

At the end of the above steps, we would be able to infer the types of all variables, function arguments and return values. We also know the signatures of the missing functions (including types of arguments and return values.) and type definition (e.g., name of attribute, member function, and their corresponding types, etc.). Using these pieces of information, we then add necessary code (e.g., variable declarations, function definitions, new classes, include library, etc.) to make the code fragment compile (Lines 15-18).

In Algorithm 2, we traverse the parse tree node by node starting from its root. The goal is to update a list of declared variables and constants (DV), undeclared/undefined variables, arguments, return values (UV), undefined functions (UF), undefined types (UT) and relations of variables and expressions (RV) based on the parse tree. We consider several cases where we need to update our lists: start of a local block (e.g., if, for, while, switch-case blocks, etc.), variable declaration or constant, expression, assignment, operator, function invocation, and undeclared variable reference. When we encounter the start of a local block, we might find a set of declared variables defined locally for that block in the parse tree (Line 17). We add all these local variables to DV when we enter the local block. We then recursively call the procedure `Traverse` to visit the children of the local block's start node in the parse tree (Line 18). We remove these local variables from DV when we exit the local block (Line 19). If the local block is a switch-case block, we update RV with "switch-case condition" (cf. Table 5.1) relationship of expressions in switch

```
switch(x){
    case 1:
        return true;
    default:
        return false;
}
```

Figure 5.2: An Example Code Snippet of Switch-case Condition

statement and case statement (Lines 14-16).

For example, in Figure 5.2, we assign the "switch-case condition" relation to variable x and constant 1 and store it in RV. When we visit a variable declaration or constant node, we update our list of declared variables DV (Lines 22-24). When we visit a reference to an undeclared variable, we update UV (Lines 26-27). If the undeclared variable is an attribute of an object, we also update UT, including updating the attribute for the object. (Lines 28-30). When we visit an expression node, we add it to EX and call `Traverse` recursively (Lines 33-37). When we visit an assignment node, we update RV with an "assignment" relationship with the return value of expressions on the left and right sides (Line 39). Next, the left side and right side of the assignment are traversed recursively (Lines 40-41). For example, for the assignment $c = b + tag \rightarrow getSize()$ in Figure 5.3, we apply the "assignment" relationship to variable c and expression $b + tag \rightarrow getSize()$ and update it to RV. After the `Traverse` procedure, with this information, we could infer c and $b + tag \rightarrow getSize()$ have the same type (see step 6 in Table 5.3). When we visit an operator node, similarly to an assignment node, we update RV with an "operator" relationship between the expressions connected with the operator and call `Traverse` to visit the two expression nodes recursively (Lines 45-47). For example, for operator node $b + tag \rightarrow getSize()$ in above example, we assign the "operator" relationship to the variable b and the expression $tag \rightarrow getSize()$ and store it in RV, for inference use in (in the step 4 in Table 5.3). When we visit a function invocation node, we also consider the arguments and return values of the

Algorithm 2 Traverse Procedure

Traverse(RPT, DV, UV, UF, UT, EX, RV)
Inputs:
RPT: Root of the parse tree of a code fragment
DV: List of declared variables and constants
UV: List of undeclared/undefined variables, arguments, return values
UF: List of undefined functions
UT: List of undefined types
EX: List of expressions
RV: List of relations of variables and expressions
Outputs: DV, UV, UF, UT, EX, RV
Method:
for each child c of RPT **do**
    if c is a start of a local block **then**
        if c is a switch case condition block **then**
            Add into RV the type-equivalence relations between the expressions in the switch statement and the expressions in the case statements based on heuristic in Table 5.1
        end if
        Add locally declared variables into DV
        Traverse(c, DV, UV, UF, UT, RV)
        Remove locally declared variables from DV
        Continue
    end if
    if c is a declared variable or a constant **then**
        Add into DV the variable/constant with its type
        Continue
    end if
    if c is an undeclared variable reference **then**
        Add into UV this new variable
        if c is an attribute of an object **then**
            Add into UT this new type with its member
        end if
        Continue
    end if
    if c is an expression **then**
        Add into EX this new expression
        Traverse(c's left side, DV, UV, UF, UT, EX, RV)
        Continue
    end if
    if c is an assignment **then**
        Add into RV the type-equivalence relations between the expressions of c's left side and c's right side based on heuristic in Table 5.1
        Traverse(c's left side, DV, UV, UF, UT, EX, RV)
        Traverse(c's right side, DV, UV, UF, UT, EX, RV)
        Continue
    end if
    if c is an operator **then**
        Add into RV the type-equivalence relations between the expressions connected with the operator based on heuristic in Table 5.1
        Traverse(c's left side, DV, UV, UF, UT, EX, RV)
        Traverse(c's right side, DV, UV, UF, UT, EX, RV)
        Continue
    end if
    if c is a function invocation **then**
        Update UF
        for each argument of its invocation a of c **do**
            Add into RV the type-equivalence relations between the argument expression a and its corresponding parameter of c based on heuristic in Table 5.1
            Traverse(a, DV, UV, UF, UT, EX, RV)
        end for
        Add into RV the type-equivalence relations between the return expression a and of c's return type based on heuristic in Table 5.1
        Traverse(c's return, DV, UV, UF, UT, EX, RV)
        if c is called from an object **then**
            Update UT
        end if
        Continue
    end if
    Traverse(c, DV, UV, UF, UT, EX, RV)
end for

```
if(tag->check(a,b,c)){
    tag->a = a;
    b = 0;
    c = b + tag->getSize();
    tag1 = tag;
    d = tag1->getSize();
}else{
    return a;
}
```

Figure 5.3: An example of code snippet

function invocations. First, we update RV with a "function definition and invocation" relationship between the expressions in the argument nodes and the parameter of the function (Line 52). Second, we invoke the procedure `Traverse` to visit the argument nodes (Line 53). We deal with the return value of the function in the same way as arguments node (Lines 55-56). For example, the pair of a variable a and an argument $check_{arg1}$[2] is assigned as "function definition and invocation" and used to do inference (in the step 1 in Table 5.3). Furthermore, if the function is called from an object, we also add the object variable to UV and update the list of undefined types (UT), including updating the member function of the object (Lines 57-59). For example, for $tag \rightarrow getSize()$, we add tag to UT as TAG[3], meanwhile we add the function $getSize()$ as TAG's member function. We also assign a "function definition and invocation" relationship to the expression $tag \rightarrow getSize()$ and the return value of function $TAG \rightarrow getSize()$[4] and update it to RV.

Example. We use the simple code fragment shown in Figure 5.3 to illustrate how the algorithm works.

We would like to extend the code fragment and make it compile. First, we generate a parse tree for it and traverse the tree to find the undeclared/undefined variables, functions, expressions, types, and type-equivalence relations.

[2]First argument of function $tag \rightarrow check()$

[3]We simply name a type by using its corresponding variable name in uppercase letters. The signature of the type of variable tag.

[4]The member function of TAG

The results are shown in Table 5.2.

Table 5.2: The Results of Lists UT, UF, UV, and EX.

List Name	Items in List
UT	$TAG, TAG1$
UF	$TAG{\rightarrow}check(), TAG{\rightarrow}getSize(), TAG1{\rightarrow}getSize()$
UV	tag, a, b, c, d, $tag{\rightarrow}a$, $tag{\rightarrow}check()$, $TAG{\rightarrow}check_{return}$, $check_{arg1}$, $check_{arg2}, check_{arg3}, TAG{\rightarrow}getSize_{return}, TAG1{\rightarrow}getSize_{return}$
EX	$tag{\rightarrow}getSize(), tag1{\rightarrow}getSize(), b + tag \rightarrow getSize()$

We notice that tag is an object, thus we create a class named TAG and it has a as an attribute and two methods $check$ and $getSize$. We simply name a type by using its corresponding variable's name in uppercase letters. We deal with tag1 in the same way. If during type inference, we find two objects are the same, we merge the corresponding types into one, and keep the attribute (i.e., signature and type) and member functions (i.e., signature, types of arguments and return value) of the new type consistent with older ones. For example, at step 7 in Table 5.3, we infer that the type of tag and $tag1$ are the same. We merge those two classes and make sure that the functions $getSize()$ in both of them are consistent. In this case, the type of the return values $TAG \rightarrow getSize_{return}$[5] and $TAG1 \rightarrow getSize_{return}$ is indeed the same.

Next, we infer the types of the variables, argument, and return values. The inference process proceeds following the steps listed in Table 5.3. Initially, each variable, argument, and return value either declared or undeclared has its own mapping. So far, we only know the type of tag and 0. In step 1, the return value $TAG \rightarrow check_{return}$ and expression $tag \rightarrow check()$, a and $check_{arg1}$, b and $check_{arg2}$, and c and $check_{arg3}$ are merged by following the function definition

[5]The return value of $TAG{\rightarrow}getSize()$.

Table 5.3: Illustration of our type inference process

Step	Mappings	Inference heuristic
1	$\{tag\} \mapsto TAG$, $\{a, check_{arg1}\} \mapsto$ `Unknown` $\{tag \rightarrow a\} \mapsto$ `Unknown`, $\{b, check_{arg2}\} \mapsto$ `Unknown`, $\{d\} \mapsto$ `Unknown`, $\{c, check_{arg3}\} \mapsto$ `Unknown`, $\{TAG \rightarrow check_{return}, tag \rightarrow check()\} \mapsto$ `Unknown`, $\{0\} \mapsto int$, $\{tag1\} \mapsto TAG1$, $\{tag \rightarrow getSize()\} \mapsto$ `Unknown`, $\{tag1 \rightarrow getSize()\} \mapsto$ `Unknown`, $\{TAG \rightarrow getSize_{return}\} \mapsto$ `Unknown`, $\{TAG1 \rightarrow getSize_{return}\} \mapsto$ `Unknown`, $\{b + tag \rightarrow getSize()\} \mapsto$ `Unknown`	Function definition and invocation (tag→check(a,b,c))
2	$\{tag\} \mapsto TAG$, $\{check_{arg1}, a, tag \rightarrow a\} \mapsto$ `Unknown`, $\{b, check_{arg2}\} \mapsto$ `Unknown`, $\{d\} \mapsto$ `Unknown`, $\{c, check_{arg2}\} \mapsto$ `Unknown`, $\{0\} \mapsto int$, $\{TAG \rightarrow check_{return}, tag \rightarrow check()\} \mapsto$ `Unknown`, $\{tag1\} \mapsto TAG1$, $\{tag \rightarrow getSize()\} \mapsto$ `Unknown`, $\{tag1 \rightarrow getSize()\} \mapsto$ `Unknown`, $\{TAG \rightarrow getSize_{return}\} \mapsto$ `Unknown`, $\{TAG1 \rightarrow getSize_{return}\} \mapsto$ `Unknown`, $\{b + tag \rightarrow getSize()\} \mapsto$ `Unknown`	Assignment (tag→a = a)
3	$\{tag\} \mapsto TAG$, $\{a, check_{arg1} tag \rightarrow a\} \mapsto$ `Unknown`, $\{b, check_{arg2}, 0\} \mapsto int$, $\{d\} \mapsto$ `Unknown`, $\{c, check_{arg3}\} \mapsto$ `Unknown`, $\{tag1\} \mapsto TAG1$, $\{TAG \rightarrow check_{return}, tag \rightarrow check()\} \mapsto$ `Unknown`, $\{tag \rightarrow getSize()\} \mapsto$ `Unknown`, $\{tag1 \rightarrow getSize()\} \mapsto$ `Unknown`, $\{TAG \rightarrow getSize_{return}\} \mapsto$ `Unknown`, $\{TAG1 \rightarrow getSize_{return}\} \mapsto$ `Unknown`, $\{b + tag \rightarrow getSize()\} \mapsto$ `Unknown`	Assignment (b = 0)
4	$\{tag\} \mapsto TAG$, $\{a, check_{arg1} tag \rightarrow a\} \mapsto$ `Unknown`, $\{b, check_{arg2}, 0\} \mapsto int$, $\{tag \rightarrow getSize(), TAG \rightarrow getSize_{return}\} \mapsto$ `Unknown`, $\{d\} \mapsto$ `Unknown`, $\{c, check_{arg3}\} \mapsto$ `Unknown`, $\{TAG \rightarrow check_{return}, tag \rightarrow check()\} \mapsto$ `Unknown`, $\{tag1\} \mapsto TAG1$, $\{tag1 \rightarrow getSize()\} \mapsto$ `Unknown`, $\{TAG1 \rightarrow getSize_{return}\} \mapsto$ `Unknown`, $\{b + tag \rightarrow getSize()\} \mapsto$ `Unknown`	Function definition and invocation (tag→getSize())
5	$\{tag\} \mapsto TAG$, $\{a, check_{arg1}, tag \rightarrow a\} \mapsto$ `Unknown`, $\{b, check_{arg2}, 0, tag \rightarrow getSize(), TAG \rightarrow getSize_{return}\} \mapsto int$, $\{d\} \mapsto$ `Unknown`, $\{c, check_{arg3}\} \mapsto$ `Unknown`, $\{TAG \rightarrow check_{return}, tag \rightarrow check()\} \mapsto$ `Unknown`, $\{tag1\} \mapsto TAG1$, $\{tag1 \rightarrow getSize()\} \mapsto$ `Unknown`, $\{TAG1 \rightarrow getSize_{return}\} \mapsto$ `Unknown`, $\{b + tag \rightarrow getSize()\} \mapsto$ `Unknown`	Operator (b + tag→getSize())
6	$\{tag\} \mapsto TAG$, $\{a, check_{arg1}, tag \rightarrow a\} \mapsto$ `Unknown`, $\{check_{arg2}, check_{arg3}, b, 0, c, tag \rightarrow getSize(), TAG \rightarrow getSize_{return}, b + tag \rightarrow getSize()\} \mapsto int$, $\{TAG \rightarrow check_{return}, tag \rightarrow check()\} \mapsto$ `Unknown`, $\{d\} \mapsto$ `Unknown`, $\{tag1\} \mapsto TAG1$, $\{tag1 \rightarrow getSize()\} \mapsto$ `Unknown`, $\{TAG1 \rightarrow getSize_{return}\} \mapsto$ `Unknown`	Assignment (c = b + tag→getSize())
7	$\{tag, tag1\} \mapsto TAG$, $\{a, check_{arg1}, tag \rightarrow a\} \mapsto$ `Unknown`, $\{b, check_{arg2}, check_{arg3}, 0, c, tag \rightarrow getSize(), TAG \rightarrow getSize_{return}, b + tag \rightarrow getSize()\} \mapsto int$, $\{TAG \rightarrow check_{return}, tag \rightarrow check()\} \mapsto$ `Unknown`, $\{tag1 \rightarrow getSize(), TAG \rightarrow getSize_{return}\} \mapsto$ `Unknown`, $\{d\} \mapsto$ `Unknown`	Assignment (tag1 = tag)
8	$\{tag, tag1\} \mapsto TAG$, $\{a, check_{arg1}, tag \rightarrow a\} \mapsto$ `Unknown`, $\{b, check_{arg2}, check_{arg3}, 0, c, tag \rightarrow getSize(), TAG \rightarrow getSize_{return}, tag1 \rightarrow getSize(), b + tag \rightarrow getSize()\} \mapsto int$, $\{TAG \rightarrow check_{return}, tag \rightarrow check()\} \mapsto$ `Unknown`, $\{d\} \mapsto$ `Unknown`	Function definition and invocation (tag→getSize())
9	$\{tag, tag1\} \mapsto TAG$, $\{a, check_{arg1}, tag \rightarrow a\} \mapsto$ `Unknown`, $\{d, b, check_{arg2}, check_{arg3}, 0, c, tag \rightarrow getSize(), TAG \rightarrow getSize_{return}, tag1 \rightarrow getSize(), b + tag \rightarrow getSize()\} \mapsto int$, $\{TAG \rightarrow check_{return}, tag \rightarrow check()\} \mapsto$ `Unknown`	Assignment (d = tag→getSize())

```
class test{
    int testmain(){
        int a, b, c, d;
        TAG* tag, tag1;
    if(tag->check(a,b,c)){
        tag->a = a;
        b = 0;
        c = b + tag->getSize();
        tag1 = tag;
        d = tag1->getSize();
    }else{
        return a;
    }
        return 0;
    }
     static void main(){
        test *t = new test();
        t->testmain();
     }
}

class TAG{
    public int check(int a0, int a1, int a2){}
    public int getSize(){}
    public int a;
    TAG(){}
}
```

Figure 5.4: Compilable code extended from a code snippet

and invocation heuristic (see Table 5.1) based on the relations stored in RV. In step 2, the mappings for a and $tag \rightarrow a$ are merged following the assignment heuristic. However, their type is still unknown since the type of a and $tag \rightarrow a$ are both unknown. The same heuristic is applied in step 3. In step 5, the operator heuristic is applied and we merge b and $tag \rightarrow getSize()$. In following steps, the function call and invocation and operator heuristics are applied and we are able to infer the types of d, b, $check(arg2)$, $checkarg3$, 0, c, $tag \rightarrow getSize()$, $TAG \rightarrow getSize_{return}$,$tag1 \rightarrow getSize()$ are all int since type of 0 is int. We could not merge any other mappings. Finally, we replace all unknown types with int.

After the above steps, we have the needed information to add variable declarations, a new class, and other needed pieces of code. The code fragment (in gray background) is extended to the compilable code (excluding $\#include$ package part in Figure 5.4.

Finally, we feed the extended code to CodeSurfer and get a PDG. We then remove some nodes from the PDG that correspond to the added code and only keep

those that correspond to the input code fragment.

5.4 Query Generation Engine

In this section, we present how we find commonalities among multiple PDGs generated from a set of example code fragments, and convert these commonalities into a dependency query. As Figure 5.1 shows, there are three steps in our query generation process:

1. Mine simple subgraphs from a set of PDGs
2. Recover text information for each node in the common subgraphs
3. Construct a query from the text-enriched subgraphs

We describe these steps below.

Before elaborating the detail, we give some definitions about the PDG. We represent a PDG as a graph $G = (N, E)$, where N is a set of nodes and E is a set of edges. N is defined as the set $\{n_1 = (ntype_1, text_1), \ldots n_i = (ntype_i, text_i), \ldots\}$. Each node has two labels: $ntype_i$ which is the node type, and $text_i$ which is the textual representations of the corresponding program element in a piece of source code. Let $n_i.ntype$ and $n_i.text$ denote the node type and text label of node n_i. E is defined as the set $\{e_1 = (n_1^L, n_1^R, etype_1), \ldots, e_i = (n_i^L, n_i^R, etype_i), \ldots\}$. Each edge contains two nodes n_i^L and n_i^R; these are the nodes connected by the edge. Each edge also has a type $etype$, $etype$ can either be data dependency or control dependency (i.e., $etype \in \{data, control\}$). Most graph mining algorithms, only accept *simple* graphs, i.e., graphs with one categorical label per node (edge). We create a simple graph representation of a PDG G by dropping the text information from the node labels. We denote the resulting simple graph G^{notext}.

5.4.1 Mine Simple Maximal Common Subgraph

First, we just focus on the node and edge types, i.e., $ntype$ and $etype$. We convert each PDG G into its *simple* graph representation G^{notext}. Next we mine for *maximal* subgraphs that appear on *all* G^{notext}. We get one such simple maximal common subgraph. We mine this maximal common subgraph using an existing graph mining tool called Gaston[6].

5.4.2 Recover Textual Information

At this step, we have a simple subgraph S^{notext} that is common for the set of all PDGs $PDGSet$. Our next job is to recover textual information for each node of this subgraph. The textual information of a node captures semantic and structural information. If the $ntype$ of a node is function, its textual information represents the name of the function. If the $ntype$ of a node is control statement, its textual information represents the type of the control statement (e.g., if, while, etc). We extract the textual information to create "contain *string*" constraints for a node in a DQL query.

Note that this step is not trivial since each node in S^{notext} can match multiple nodes containing different textual information in a particular PDG, and thus there is a need to choose the best matching node. The best matching nodes among different PDGs may contain different textual information, and thus there is a need to unify this textual information by removing information specific to a matching node of a PDG.

High-Level Description. The algorithm performing textual content recovery is shown in Algorithm 3. The idea is to perform graph matching. We match each simple subgraph S^{notext} to each PDG G_j in `PDGSet` based on the labels $ntype$ and $etype$ (we ignore the textual labels) (Lines 8-11). The graph matching operation returns a set of candidate nodes in G_j that potentially match each node of S_i^{notext}.

[6]http://www.liacs.nl/∼snijssen/gaston/

Algorithm 3 Textual Information Recovery Procedure

1: **recoverText**($PDGSet$, G_{sub}^{notext},$Candidate$)
2: **Input:**
3: $PDGSet$: A set of PDGs
4: S^{notext}: A common subgraph of PDGs in $PDGSet$ ignoring $text$ labels
5: **Output:** Text enriched S^{notext} (i.e., S)
6: **Method:**
7: Let $Cand$ = $Pool$ = {}
8: **for** each G_i in $PDGSet$ **do**
9: Perform a graph matching operation between S^{notext} and G_i
10: Let $Cand_n^{G_i}$ stores all candidates for node n in S^{notext}
11: **end for**
12: **for** each node n in S^{notext} **do**
13: Let Rep = {} // Representative nodes from each PDGs
14: **if** $\forall G_i \in PDGSet.|Cand_n^{G_i}| = 1$ **then**
15: Rep = $\{c|\exists_{G_i \in PDGSet}.c \in Cand_n^{G_i}\}$
16: **else**
17: Let $Reference$ = {} //First representative node
18: Let $Others$ = {} //Candidates from other PDGs
19: **if** $\exists G_i \in PDGSet.|Cand_n^{G_i}| = 1$ **then**
20: $Reference$ = $\{c \in Cand_n^{G_i} | \, |Cand_n^{G_i}| = 1\}$
21: $Others$ = $\{Cand_n^{G_i} | G_i \in PDGSet \wedge |Cand_n^{G_i}| > 1\}$
22: **else**
23: $Reference$ = an arbitrary $Cand_n^{G_i}$
24: $Others$ = $\{Cand_n^{G_i} | G_i \in PDGSet\} \setminus Reference$
25: Delete all but (random) node in $Reference$
26: **end if**
27: Let Rep = selectRepCand($Reference$,$Others$)
28: **end if**
29: Remove nodes in Rep from $Cand_{n'}^{G_i}$, $n' \neq n$ and $G_i \in PDGSet$
30: $n.Text$ = Unify text labels of nodes in Rep
31: **end for**

Notation-wise, given a node n in S_i^{notext}, we denote its set of candidate nodes in G_j as $Cand_n^{G_j}$. We select one representative candidate per PDG and extract its textual label (Lines 12-28). A candidate node of a PDG can only represent one node in S^{notext}. Thus we delete the representative nodes from the sets of candidate nodes of other nodes in S^{notext} (Line 29). We then unify this set of labels to recover the label for node n (Line 30). We elaborate our approach to select representative candidates and unify text labels in the following paragraphs.

Selecting Representative Candidates. We would like to select one representative candidate per PDG for a node n. If all candidate sets are of size 1, then we simply take all candidate nodes as the representative nodes (Lines 14-15). Otherwise, we would like to pick representative nodes such that they are similar to one another. The rationale for this would be explained further by an example described in Section 5.4.4. To pick similar representative nodes, first we pick a set of reference nodes (Lines 20, 23, and 25). If there are candidate set of size 1, we take the nodes in these

sets as the reference nodes (Lines 19-20). Otherwise, we pick an arbitrary candidate node as a reference node (Lines 23, 25). Next we pick representative nodes from other PDGs that are similar to the reference nodes (Line 27). Algorithm 4 describe this last step in more detail. In Algorithm 4, we first take the input reference nodes as the representative nodes (Line 7). Next, we visit each candidate node set of the other PDGs and pick a node that is the most similar to the selected representative nodes (Lines 8-12).

Algorithm 4 Selection of Representative Candidates

1: **selectRepCand**($Reference$, $Others$)
2: **Input:**
3: $Reference$: Selected representative candidates
4: $Others$: Candidates from other PDGs
5: **Output:** All representative candidates
6: **Method:**
7: Let $Rep = \{n | n \in Reference\}$
8: **for** each set $Cand_n^{G_i}$ in $Others$ **do**
9: Select a node n' in $Cand_n^{G_i}$ which is the most similar to all nodes in $Reference$
10: $Rep = Rep \bigcup \{n'\}$
11: $Others = Others \setminus Cand_n^{G_i}$
12: **end for**
13: **return** Rep

At line 9 of Algorithm 4, we need to measure the similarity between nodes. We measure the similarity between nodes by considering their text labels. Each node is represented by a vector that captures its textual information. Each element of the vector corresponds to a word token that appears in the corresponding node's text label. In our dataset, the sizes of these vectors are from 1 to 11 with a mean of 4.2. The vocabulary of these vectors contains all words that appear in the textual labels of all nodes in the PDGs. We then weigh each word by using a TF-IDF weighting scheme [70]. Here, the TF (Term Frequency) of a word refers to the number of times the word appear in the text label. IDF (Inverse Document Frequency) refers to 1 / DF (Document Frequency), where DF is the number of representative nodes with text labels containing the word. Each node is then represented as a vector of weights. Similarity between two nodes n_1 and n_2 is measured by the similarity between their corresponding weight vectors v_1 and v_2. We use standard cosine similarity for this purpose [70]:

$$cos(v_1, v_2) = \frac{\sum\limits_{i \in (v_1 \bigcup v_2)} v_1[i] \times v_2[i]}{\sqrt{\sum\limits_{i \in v_1} v_1[i]^2} \times \sqrt{\sum\limits_{i \in v_2} v_2[i]^2}}$$

Finally, we average the similarity scores of the node n in $Cand_n^{G_i}$ and each node in $Reference$ and select the node with the highest score as the most similar one.

Unifying Textual Labels. For this process, for a node n in a subgraph S^{notext}, we have as input a set of representative nodes with their text labels. Our goal is to create a single unified textual label for n (Line 30). To do this, we perform the following steps:

1. For each representative node text label, we pre-process it as follows: If $n.ntype$ = function call, keep on the function name. If $n.ntype$ = expression, keep only the right side of the expression. For all other node types, all text is kept.

2. Get the longest common text from the pre-processed text labels. In this step, we find the longest consecutive sequence of characters [26] (we don't consider case-sensitivity) that appears in all the text labels.

3. Split the resulting text and remove special symbols. We first split the resulting text with white space. We then perform Camel Case splitting on each token to split identifiers [5]. For example, we split "getString" into "get" and "string". We also split each token that contains some special symbols (i.e., operators and numbers). At last, we remove some special symbols (i.e., operators and numbers).

Example: Assume two representative nodes n_1 and n_2 with labels "a = getExtString(para1) + 123 + var12" and "b = ExtString(para2) + 123 + var13" respectively. We first perform step 1 on these two nodes. This step removes "para1" from n_1 and "para2" for n_2 as their ntype is function call. We also remove "a =" and "b =" as we only keep the right hand side of an expression. After step 1, we get

"getExtString + 123 + var12" and "ExtString + 123 + var13" for n_1 and n_2, respectively. Following step 2, we get the longest common text "ExtString + 123 + var1" from "getExtString + 123 + var12" and "ExtString + 123 + var13". At the last step, we split the longest common text "ExtString + 123 + var1" to "ext", "string", "+", "123", "+", "var", and "1" and then remove "+", "123" and "1". Finally, we get "ext", "string" and "var".

5.4.3 Construct Query from Enriched Subgraphs

For this step, we have as input a text-enriched common maximal subgraph S. We create a dependency query (expressed in DQL which was described in Section 4.4) from this subgraph by the following steps:

1. Output the nodes in S and their types as *node declarations* of the query
2. Output the edges in S as *relationship descriptions* of the query
3. Output the text labels of nodes in S as *node descriptions* of the query
4. Identify all nodes defined in the query as *target* nodes

5.4.4 Example

We use Figure 5.5 to illustrate the query generation process. PDGs G_A and G_B are the inputs and a query written in DQL is the output. First, we perform graph mining on G_A and G_B to get a simple maximal common subgraph which is S.

Next, we run Algorithm 3 on G_A, G_B, and S to recover the textual information for the nodes in S. For node s_1, its representative nodes in G_A and G_B can be easily identified as there is only one candidate node for each PDG; they are nodes a_1 and b_1. After we unify the text labels of a_1 and b_1, we get "if" as the text label of s_1. For node s_2, there are two candidate nodes in G_A (i.e., nodes a_2 and a_3), and one candidate node in G_B (i.e., node b_2). For this case, our algorithm first selects node

b_2 as a representative node. It then finds the node in $\{a_2,a_3\}$ that is the most similar to b_2. It selects a_3 as it is more similar to b_2 than a_2. After we unify the text labels of a_3 and b_2, we obtain "ext" by getting the longest common text "ext()" and then removing the parentheses.

Finally, we convert the text-enriched S to a dependency query expressed in DQL. The resulting query is as follows:

Node declarations: *ctrlPoint A, Expression B*;

Node descriptions: *A contains if, B contains ext*;

Relationship descriptions: *A oneStep controls B*;

Targets: *A,B*;

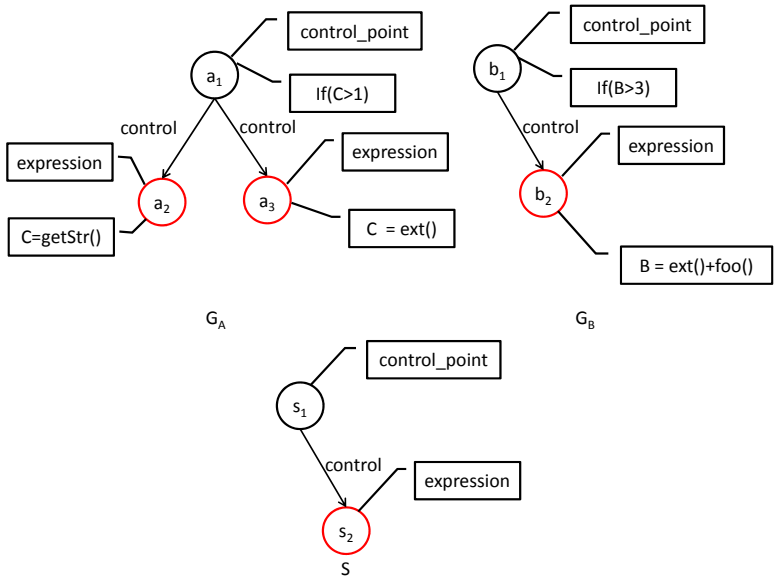

Figure 5.5: Query Generation Example

5.5 Evaluation

5.5.1 Experimental Settings

To evaluate our proposed approach, we extract code search scenarios from repositories of real software systems. We are especially interested in recovering real-life

changes that need to be applied to various locations in a code base. We use these changes to simulate code search scenarios. Developers might know some of these locations but would like to know other relevant locations. Code search could help developers to find these other locations. This experimental setting follows the setting described in Wang et al.' work [112]. The code bases that we use for our experiments are from four realistic programs written in C and C++ namely Apache Http Server (12 versions), Inkscape (9 versions), Apache Subversion (8 versions), and Libmpeg (3 versions), which have undergone at least thirteen years of continuous development, improvement, and optimization. The details of these programs are shown in Table 5.4.

In previous work [108], we perform an empirical study on widespread changes. To identify widespread changes, we look for commits that touch many files such that these files are modified in a similar way structurally and semantically. Files involved in a widespread change are modified for the same purpose, e.g., fixing the same bug, etc. To check whether the files are modified in a semantically similar way, we manually inspect the files to find whether they are indeed changed for the same purpose. We follow the same procedure to identify widespread changes on the 4 programs that we use in this study. Given 47 widespread changes. Each widespread change involves 5 to 53 code locations, with an average of 10 locations. Let us refer to the code requiring change at each location as a fragment. In total, we have 478 fragments. Each fragment is 2 to 20 lines of code. To simulate code search scenarios, for each widespread change, we randomly pick two fragments as the input to a code search task. We test the effectiveness of the code search task using the remaining fragments, i.e., the remaining fragments become the gold set or standard. Since we have 47 widespread changes, we simulate 47 code search tasks.

We implement our AutoQuery approach in Python and Java. AutoQuery converts code examples into dependency queries. We use the approach by Wang et al. [112] as the backend code search engine that processes the dependency queries and return relevant pieces of code. We use a desktop with an Intel Core i5 3.2GHz

Table 5.4: Programs Analyzed in this Study

Name	Description	Size (KLOC)	#Versions
Apache Http Server	HTTP Web Server.	264.5	12
Apache Subversion	Open source version control system.	483.5	8
Inkscape	Open source vector graphics editor.	458.2	9
Libmpeg2	Library for decoding MPEG-2 and MPEG-1 video streams.	37.3	3

CPU installed with 4GiB of memory and 2TiB of hard disks to run experiments.

We compare the performance of AutoQuery with manually constructed queries (we refer to them as UserQuery). We perform a user study involving 10 participants and the 47 code search tasks. Among the 10 participants, 9 of them are PhD students who have at least two years of C and C++ programming experience. One of them is a professional software engineer who has three years of C and C++ programming experience. All of them know are familiar with Program Dependency Graphs (PDGs) – many of them have taken a course on program analysis.

We give each participant tasks in the following format: given a set of code fragments, generate a DQL query that can capture the code fragments in this set. Each participant is assigned four or five tasks. For each task, a participant is given two fragments (i.e., code examples) along with the corresponding PDGs of the fragments, to ease the task. A participant needs to look at the code fragments as well as their corresponding PDGs, and construct a dependency query expressed in DQL to find other similar code. We record the queries users created and the time each of them takes to complete the construction of a query. Before users start with the tasks, each of them is given a 20 minute tutorial about dependence-based code search and DQL. They are also given a 10 minute exercise to construct a query from simple code fragments. These tutorial and exercise are meant to familiarize users with the tasks.

5.5.2 Experiment Results

We aim to answer the following research questions:

1. RQ1: Can AutoQuery generate good dependency queries that can retrieve relevant search results?
2. RQ2: Can AutoQuery perform comparably to developers in constructing good dependency queries?
3. RQ3: Can AutoQuery reduce the time it takes to construct queries?

The first research question investigates the overall effectiveness of our proposed approach. The second research question investigates the effectiveness of AutoQuery as compared to manually constructed queries (UserQuery). The third research question investigates the efficiency of our approach as compared to UserQuery.

RQ1: Effectiveness of AutoQuery

To answer this research question, for each of the 47 tasks, we run the dependency query generated by AutoQuery using the dependence-based code search engine of Wang et al. [112]. We count the number of code fragments in the gold set that are retrieved. Based on this, we compute the average precision, recall, and F-measure which are standard information retrieval measures [70]. For a given task, precision, recall, and F-measures are defined as follows:

$$precision = \frac{\#\ retrieved\ code\ fragments\ in\ the\ gold\ set}{\#\ retrieved\ code\ fragments}$$

$$recall = \frac{\#\ retrieved\ code\ fragments\ in\ the\ gold\ set}{\#\ code\ fragments\ in\ the\ gold\ set}$$

$$F - measure = \frac{2 \times precision \times recall}{precision + recall}$$

F-measure is the harmonic mean of precision and recall and it is often used as a summary measure. It quantifies if an increase in precision outweighs a decrease

Table 5.5: AutoQuery vs. UserQuery: Precision, Recall, and F-measure

	AutoQuery	UserQuery
Precision	0.684	0.584
Recall	0.721	0.767
F-measure	0.702	0.664

in recall (and vice versa). Table 5.5 shows the result. The average precision, recall, and F-measure for the 47 tasks are 68.4%, 72.1%, and 70.2%, respectively. For 25 automatically constructed queries, we are able to find all fragments in the gold set (recall = 1). For 21 automatically constructed queries, all retrieved fragments are in the gold set (precision = 1). For 12 automatically constructed queries, all fragments in the gold set are retrieved and all retrieved fragments are in the gold set (F-measure = 1).

In the default setting, for each task, AutoQuery is given two code fragments to generate a query. We next test the effectiveness of AutoQuery with different numbers of code fragments as input. In this experiment, we give k randomly selected code fragments to AutoQuery. Since the number of relevant code fragments for each task ranges from 5 to 53, we vary the value of k in set $\{1,2,3,4\}$ and measure the effectiveness in terms of recall, precision and F-measure. In Figure 5.6, we present the effectiveness of AutoQuery for different numbers of code fragments. We notice that the recall increases as the number of code fragments increases, and the precision decreases as the number of code fragments increases. As more code fragments are used to generate a query, a more general query is generated. Searching using a more general query returns a larger number of code fragments. As the number of code fragments increases, more false positives are introduced, which leads to a lower precision value. On other hand, as the number of code fragments increases, fewer false negatives are introduced, which leads to a higher recall. In terms of F-measure, the harmonic mean of precision and recall, AutoQuery performs the best when three code fragments are given.

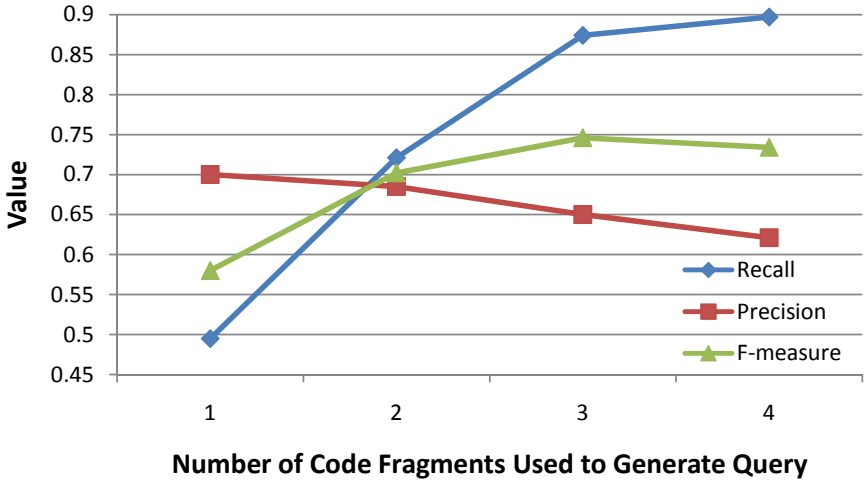

Figure 5.6: Results (y-axis) vary as the number of code fragments (x-axis) increase

We also perform a 5-fold cross validation like experiment to evaluate AutoQuery. We randomly split 1/5 of the code fragments in each gold set of our code search tasks into 5 buckets. We then perform 5 iterations. In each iteration, we use fragments in one of the buckets to generate a query and use the other four buckets to test the effectiveness of AutoQuery. For this experiment, AutoQuery achieves a recall, precision, and F-measure of 0.861, 0.64, and 0.734, respectively

RQ2: AutoQuery Vs. UserQuery

To answer this research question, we compare the results of AutoQuery with those of UserQuery. For each of the 47 code search tasks, we use the same pairs of code fragments as input to AutoQuery and UserQuery.

As shown in Table 5.5, the precision, recall, and F-measure of UserQuery are 58.4%, 76.7%, and 66.4%, respectively. In general, users' queries are more general (contain fewer constraints), while queries generated by AutoQuery are more specific (contain more constraints). This makes the results obtained from the queries generated by AutoQuery more precise (resulting in higher precision). However, some relevant results are missed (resulting in lower recall) because of the more specific queries. These is thus again trade-off between precision and recall. To measure this trade-off, we also compute the F-measure. In terms of the F-measure, AutoQuery

achieves a 5.7% improvement over UserQuery.

We also perform a Wilcoxon signed-rank test [115] to test the significance of the differences in the performance of AutoQuery and UserQuery measured in terms of recall, precision and F-measure. The results show that the differences in terms of F-measure (p-value = 0.49) and recall (p-value = 0.17) are not significant, while the difference in terms of precision is significant (p-value = 0.02). This shows that AutoQuery is comparable to developers in constructing dependency queries in terms of recall and F-measure, and the improvement in terms of precision achieved by AutoQuery in constructing dependency queries over developers is statistically significant.

Table 5.6 presents the number of tasks where AutoQuery outperforms UserQuery (and vice versa). In terms of precision, AutoQuery wins on 18 queries and loses on 7 queries. In terms or recall, AutoQuery wins on 4 queries and loses on 5 queries. In terms of F-measure, AutoQuery wins on 16 queries and loses on 9 queries. AutoQuery and UserQuery do not produce the exact same search results for the remaining queries, however their precision (or recall, or F-measure) values for these queries are the same. Figures 5.7, 5.8, and 5.9 present precision, recall, and F-measure of AutoQuery and UserQuery for each of the 47 tasks. The above results show that AutoQuery is comparable to developers in constructing dependency queries.

We can notice that for Query 20 in Figure 5.7, AutoQuery achieves a precision value of 1, while UserQuery can not find any correct results, which leads to poor precision. We manually check the query generated by the user, and we notice that the user miss important constraints or make wrong constraints which make the query ineffective at finding relevant code fragments. The precisions of the queries formed by users for tasks 13, 18-21 are low. We have checked the queries we got from tasks 13, 18-21 and find that they are generated by users that are able to create relatively good queries for other tasks. In Figure 5.8, for some queries (e.g., Query 41) UserQuery outperforms AutoQuery. We checked the queries generated by both

Table 5.6: AutoQuery vs. UserQuery: Number of Winning Queries

	$AutoQuery$ Wins	$UserQuery$ Wins
Precision	18	7
Recall	4	5
F-measure	16	9

of them and we found that the queries generated by users are more general with fewer constraints, while queries obtained from AutoQuery are more specific with more constraints. More general queries return more results which causes a UserQuery to achieve a higher recall than AutoQuery. Based on this observation, in the future, to improve the effectiveness of AutoQuery, we plan to extend it by developing a machine learning technique that can remove or weaken some of the generated constraints automatically.

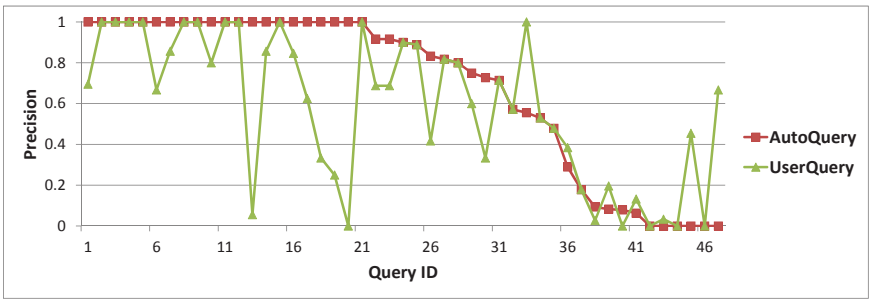

Figure 5.7: AutoQuery vs. UserQuery: Precision per Task

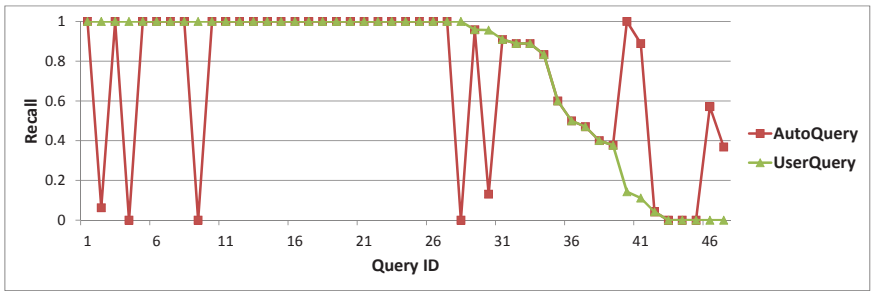

Figure 5.8: AutoQuery vs. UserQuery: Recall per Task

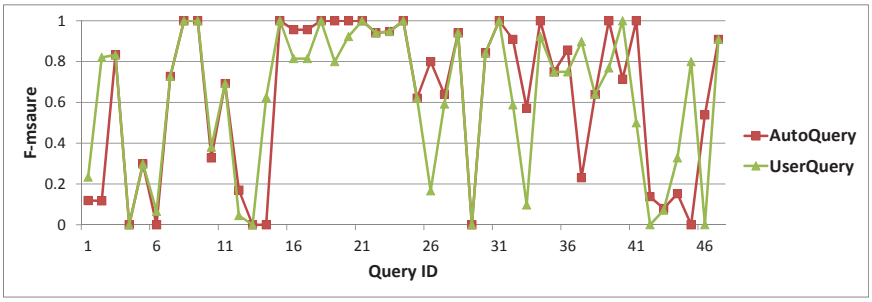

Figure 5.9: AutoQuery vs. UserQuery: F-measure per Task

RQ3: Efficiency of AutoQuery compared with UserQuery

Static analysis and data mining techniques can take much time and resources to run to completion. Some static analysis and data mining techniques can run for hours. Our approach makes use of both static analysis and data mining. Graph mining in particular can be a time consuming operation. Thus, in this research question we want to investigate whether our approach is efficient enough as compared to the time it takes for developers to manually construct queries. If our approach is slower than the time developers take to manually construct queries, then it might not be practical. Another side goal of this research question is to investigate the effort developers need to construct queries as measured by the time they take to construct them.

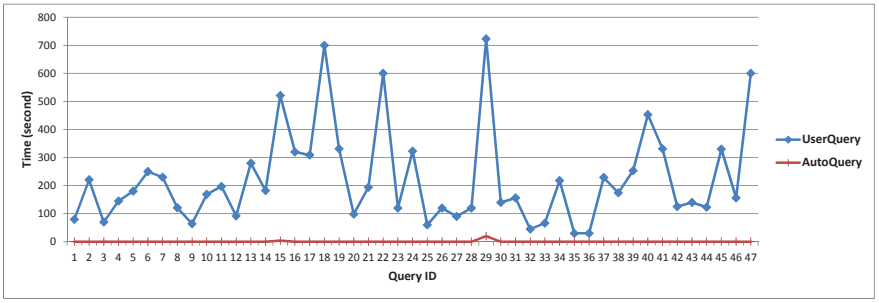

Figure 5.10: AutoQuery vs. UserQuery: Efficiency

To answer this question, we compare the time it takes for AutoQuery to construct queries with that of UserQuery. The total time it takes for AutoQuery to construct the 47 queries is 27.5 seconds. Thus, the average time per query is 0.6 second which is reasonably short. The total time for developers (UserQuery) to construct the 47

queries is 10,509 seconds, with an average of 223.6 seconds. Compared with the time it takes for developers to construct query, our approach is much more efficient. Many developers are likely to be reluctant to use a code search tool if they need to think hard for 3-4 minutes to construct a query. AutoQuery addresses this concern.

Figure 5.10 shows the time it takes for AutoQuery and a developer to construct each of the 47 queries. Almost all queries can by constructed by AutoQuery in less than a second except for two queries: one of them requires 3.9 seconds and another requires 19.8 seconds. For these two most complicated queries, developers spend 521 seconds and 723 seconds, respectively. From these experiment results, we can see that AutoQuery is able to save much developer time.

(1)

```
if (!type) {
  GTypeInfo info = {
       sizeof(SPAttributeTableClass),
       0, // base_init
       0, // base_finalize
       (GClassInitFunc)sp_attribute_table_class_init,
       0, // class_finalize
       0, // class_data
       sizeof(SPAttributeTable),
       0, // n_preallocs
       (GInstanceInitFunc)sp_attribute_table_init,
       0 // value_table
    };
   type = g_type_register_static(GTK_TYPE_VBOX,
       "SPAttributeTable",
       &info,
       static_cast<GTypeFlags>(0));
}
return type;
```

(2)

```
    static GType type = 0;

if (!type) {
    GTypeInfo info = {
        sizeof(SPCtrlRectClass),
       0, // base_init
       0, // base_finalize
       (GClassInitFunc)sp_ctrlrect_class_init,
       0, // class_finalize
       0, // class_data
       sizeof(CtrlRect),
       0, // n_preallocs
       (GInstanceInitFunc)sp_ctrlrect_init,
       0 // value_table
    };
   type = g_type_register_static(SP_TYPE_CANVAS_ITEM,
       "SPCtrlRect",
        &info,
        static_cast<GTypeFlags>(0));
}
return type;
```

Figure 5.11: Two Complex Code Fragments Example Containing 19 lines

```
(1)
        if (item) {
       ec->shape_knot_holder = sp_item_knot_holder(item, ec->desktop);
       Node *shape_repr;
       shape_repr= SP_OBJECT_REPR(item);
       if (shape_repr) {
           ec->shape_repr = shape_repr;
           sp_repr_ref(shape_repr);
           sp_repr_add_listener(shape_repr, &ec_shape_repr_events, ec);
           sp_repr_synthesize_events(shape_repr, &ec_shape_repr_events, ec);
        }
(2)
    if (item) {
       ec->shape_knot_holder = sp_item_knot_holder(item, ec->desktop);
       Node *shape_repr;
       shape_repr= SP_OBJECT_REPR(item);
       if (shape_repr) {
           ec->shape_repr = shape_repr;
           sp_repr_ref(shape_repr);
           sp_repr_add_listener(shape_repr, &ec_shape_repr_events, ec);
           sp_repr_synthesize_events(shape_repr, &ec_shape_repr_events, ec);
        }
```

Figure 5.12: Two Complex Code Fragments Example Containing 10 lines

In Figure 5.11 and 5.12, we present the code fragments for those two queries, which contain 19 (its PDG has 23 nodes and 10 edges) and 10 lines (its PDG has 21 nodes and 27 edges), respectively. Correspondingly, their dependence graphs are bigger and more complex, which take more time to generate query by AutoQuery. In our approach, we perform common graph extraction which is very sensitive to the size of graph (size of nodes and edges). Also, users cost much more time to formulate query based on the given code fragments. In the future, one way we would like to alleviate this problem is by compressing the dependence graph by removing some unimportant edges and nodes.

Threats to Validity

Threats to internal validity include experimenter bias. There might be subjectivity in the queries that a participant formulates. Similar to past studies, e.g., [77, 104, 107], we do not have a large pool of participants such that each query can be formulated by many people. It could be the case that some participants may have formulated very bad queries, which may not be representative of a typical/trained user. Still, we consider 47 widespread changes and take averages, and we believe averaging helps reduce the threats. Also, we have given user study participants a 20 minutes tutorial and 10 minutes exercise (the exercises we gave to the users are similar to the actual tasks) to familiarize them with writing queries in DQL. We also provided helps and hints when users faced difficulties during the tasks. The results that we got may be different if we train them much more. However, if we train users more, users may behave more like AutoQuery. We decided 30 minutes would be sufficient to become familiar with DQL and left query construction to users' diverse creativity.

Threats to external validity relate to the generalizability of our approach. We have investigated 47 widespread changes and experimented on four realistic programs written in C and C++. Admittedly, our programs are just a fraction of the collection of all programs out there. Also, our tasks do not cover all kinds of widespread changes. In the future, we plan to reduce the threats to external validity further by investigating more widespread changes and more programs written in other languages.

5.6 Conclusion and Future Work

Searching through a large base of source code is common activities performed by developers. Without the aid of automated code search tools, developers need to tap on their experience to browse relevant source code files and manually find the code fragments that are related to their tasks at hand. This is not only be time consuming

but also error-prone. A number of code search tools have been proposed to address this problem. Many of them accept textual descriptions as user queries and return relevant code fragments.

Source code contains not only text but also dependencies. Dependence-based code search tools accept queries expressed as dependency relationships among program elements of interest and return code fragments whose constituent program elements satisfy the dependency relationships. Dependence-based code search tools have been shown to be effective and improve the accuracy of search results than text-based code search tools. However, there is one drawback that potentially hampers the usage of dependence-based code search. It may be hard for users who have no or little knowledge of PDGs to construct queries.

To address this drawback, we propose an automatic approach to construct a query based on common dependency structures and textual information extracted from a set of sample code fragments. We have evaluated our approach with 47 realistic code search tasks on 4 real systems, and show that the automatically constructed dependency queries recover relevant code with a precision, recall, and F-measure of 68.4%, 72.1%, and 70.2%, respectively. We have also performed a user study that shows that our automatically constructed queries are comparable to human constructed queries in retrieving relevant codes.

In the future, we plan to experiment with more code search tasks and systems.

Chapter 6

Search-Based Fault Localizaiton

6.1 Introduction

Many approaches have been proposed to help in automating debugging process, especially in localizing faults [2, 15, 45, 60, 65, 87, 120]. One common family of approaches is spectrum-based fault localization, e.g., [2, 45], where program traces or abstractions of traces (called program spectra) of correct and failed executions are compared to identify program elements that likely cause failures and are reported as potential faults. Some well-known studies include Tarantula [45] and Ochiai [2]; Baah et al. [9] improve fault localization accuracy by using a linear model to estimate the causal effect of a program element on failures. Lucia et al. [67] have also investigated the effectiveness of more than 20 association measures, and report that *information gain* may on average perform the best for fault localization.

We observe that although one measure could be more accurate than other measures on some datasets, there are programs where a poorer measure performs better than a better one. We aim to leverage this fact by composing different measures into a composite one that could perform better than any individual constituent measure.

In this chapter, we model the problem of finding an optimal composition of various measures as a search-based optimization problem. Our search-based fault localization engine works on two phases: training and deployment. In the training

phase, the engine takes in a number of buggy programs along with actual fault locations to measure how good a particular composition is. A good composition should be able to locate many bugs in this training set with a high accuracy. Search heuristics (e.g. simulated annealing [51] and genetic algorithm [35]) are employed to traverse the search space of all possible linear combinations of the more than 20 association measures to find near optimal ones that perform best on the training set. In the deployment phase, our engine takes a given buggy program and its spectra, and applies the composite measure to identify potential fault locations.

We evaluate our search-based fault localization engine on ten programs, including seven C programs in the Siemens test suite [38], and three larger programs (Space in C, and NanoXML and XML-Security in Java) from the Software-artifact Infrastructure Repository [90, 93]. We compare the effectiveness of our proposed composite measures with its constituent measures. Similar to past papers [2, 45, 67], effectiveness is measured based on two criteria: 1. The number of bugs that could be localized when at most a given proportion of code in a program is inspected, and 2. The proportion of code to be inspected to localize all bugs. We show that our search-based fault localization could locate 22% and 14% more bugs than Tarantula and Ochiai respectively when only 10% of the code is inspected. Also, in order to localize all bugs, we reduce the average amount of code to be inspected from 27.0% (for Tarantula) and 25.3% (for Ochiai) to 20.6% (23.7% and 18.6% reduction respectively). With Wilcoxon signed-rank test [115], we find that the improved results are statistically significant.

In the experiments, we also demonstrate the insensitivity of the composite measures to the training set by performing a random 10-fold cross validation, a cross-program validation, and a cross-bugtype validation. The validation results show that the fault localization performance of the composite measures constructed from different training sets varies only a little, implying that the composite measures can consistently perform better than individual measures over various kinds of bugs in various programs.

6.2 Preliminaries

In this section, we instroduce spectrum based fault localization, and search based algorithms.

6.2.1 Spectrum Based Fault Localization

Spectrum-based fault localization starts with a faulty program, a set of test cases, and a test oracle. The set of test cases is run over the faulty program and observations of how the program runs on each of the test cases are recorded as program spectra. A program spectrum [30, 88] is simply a set of data or values collected at runtime; each value could be a program state, or a counter or a flag for a program element. A test oracle is available to label whether a particular output or execution of a test case is correct or wrong. Wrong executions are classified as program failures. The task of a fault localization tool is to find the program elements that are responsible for the failures (i.e., the faults or the root causes) based on the program spectra of both correct and wrong executions.

There have been various spectra proposed in the literature [1, 30], but we are particularly interested in *block-hit program spectra*, each of which consists of a set of flags to indicate whether each block is executed or not in each test case.[1] An example of block-hit program spectra is shown in Figure 6.1. The second column is a code excerpt from the Siemens test programs; the "Block ID" column indicates the ID of the containing block of statements. A bug lies in the `if` condition in Block 3, causing Blocks 4–7 to be skipped when the variable `count` is 1. The other columns indicate whether each basic block is executed by test cases: T1, T2, T3, and T4, along with the information whether each test case passes or fails[2]. In

[1] We acknowledge that different spectra may have different effects in fault localization. We use *block-hit program spectra* in this chapter mainly for establishing the same ground for comparison with Tarantula and Ochiai.

[2] Notice that the code excerpt is a part of a function which could be called many times in a single execution of the program. Thus, it is possible for Block 2 and Block 3 to be executed in a single test case.

Block ID	Code	T1	T2	T3	T4
1	int count; int n; Ele *proc; List *src_queue, *dest_queue; if (prio >= MAXPRIO) /*maxprio=3*/	•	•	•	•
2	{return;}		•	•	•

3	src_queue = prio_queue[prio];
	dest_queue = prio_queue[prio+1];
	count = src_queue->mem_count;
	if (count > 1) /* Bug*//* supposed : count>0*/ {
		•	•	•	•

4	n = (int) (count*ratio + 1); proc = find_nth(src_queue, n);		•	•	
5	if (proc) {		•	•	
6	src_queue = del_ele(src_queue, proc);		•	•	
7	proc->priority = prio; dest_queue = append_ele(dest_queue, proc); } }}		•	•	
	Test Case's Success (S)/Fail (F) Status:	S	S	S	F

Figure 6.1: Example of block-hit program spectra

this example, • denotes that a block is executed by a test case and an empty cell denotes that a block is not executed by a test case.

Spectrum-based fault localization techniques find the correlation between the execution of a program element and failures. A number of correlation measures being proposed and these are commonly termed as suspiciousness measures [1, 3, 4, 45, 46, 67]. Those program elements that are more correlated with failures are deemed to be more suspicious than the others. Developers are then presented with a list of program elements sorted based on their suspiciousness scores in descending order.

6.2.2 Search-based Algorithms

There are different kinds of search based algorithms for optimization. These algorithms typically work on a search space of possible solutions. Each solution is given a score based on an objective function. Thus, the search space of the possible solutions forms a landscape; some points in the landscape are hills, i.e., have high objective scores, while some others are valleys, i.e., have low objective scores. A search algorithm looks for an optimum solution i.e., the solution with the maximum score (for maximization problem) or minimum score (for minimization problem).

As the search space could be very large, various metaheuristics solutions have been proposed. The metaheuristic solutions attempt to find a good enough solution or a near optimal solution in a reasonable amount of time.

We discuss two search-based algorithms: simulated annealing (SA) and genetic algorithm (GA). SA aims to minimize a given objective function, while a GA aims to maximize an objective function. We describe these algorithms in the following paragraphs.

Simulated Annealing. Simulated annealing (SA) proposed by Kirkpatrick *et al.* [51] was inspired by the annealing process in metallurgy where a piece of metal is heated and slowly cooled down to result in another piece with a lower entropy. To mimic this, simulated annealing updates a current solution with a randomly picked neighbor if the neighbor has a better objective function score, or the difference between the objective function score of the neighbor and that of itself is less than a threshold, which is a function of a random number and the simulated temperature T (which should be gradually decreased during the process). The process is a global optimization procedure; it prefers going "downhill" in the search process; however, it also allows an "uphill" move in the search landscape with some probability that is a function of T. Initially (when the temperature is high), more "uphill" moves are allowed; as time passes (when the temperature becomes low), fewer "uphill" moves are allowed.

A standard algorithmic template for SA is shown in Figure 6.2. Simulated annealing has two loops. The outer loop reduces the temperature T in every iteration. The inner loop repeatedly swaps a solution with its randomly picked neighbor based on the parameter T until a certain termination condition (*e.g.*, I iterations have been performed) is satisfied.

Genetic Algorithm. A genetic algorithm (GA) is a search procedure that aims to find one or more solutions that maximize a given objective function [35]. It does so by modeling a solution as a chromosome which can be represented as a bit-

Procedure Simulated Annealing

Inputs: T: Initial temperature

L: Lowest temperature

R: Temperature reduction factor

I: Inner loop repetition

Outputs: Best solution found

Method:

1: ln Let S = A random initial solution

2: ln While $T > L$

3: ln Repeat for I times

4: ln Let S' = A random neighbour of S

5: ln Let Δ = S'.Score - S.Score

6: ln If ($\Delta > 0$)

7: ln $S = S'$

8: ln Else If ($e^{-\Delta/T} > random(0, 1)$)

9: ln $S = S'$

10: $T = T \times R$

11: **Output** S

Figure 6.2: Simulated Annealing: Pseudocode

string. Given a population of chromosomes, various operations including selection, crossover, and mutation, are performed. Selection selects the best chromosomes based on an objective function. Crossover combines two chromosomes into a single chromosome. Mutation flips random elements in a chromosome. These operations are performed iteratively many times. At the end of each iteration a new generation of chromosomes is formed. The iterative process stops when a certain criteria is met, *e.g.*, a solution with an objection function score larger than a threshold is found, or

when a specific number of iterations of selection, crossover, and mutation has been performed.

There are many genetic algorithm variants and Figure 6.3 shows the pseudocode of the genetic algorithm variant used in this chapter. The parameters are usually empirically determined by some preliminary experiments.

Procedure Genetic Algorithm

Inputs: N: Number of chromosomes

I: Number of iterations

M: Mutation probability

C: Crossover probability

Outputs: Best solution found

Method:

1: ln Let P = Initial set of population with N members

2: ln Evaluate P and find the best solution so far

3: ln Repeat for I times

4: ln Let P' = Selection(P)

5: ln $P' =$ Crossover(P', C)

6: ln $P' =$ Mutation(P', M)

7: ln Evaluate P' and update the best solution so far

8: ln **Output** the best solution found

Figure 6.3: Genetic Algorithm: Pseudocode

The major tasks to employ a GA for a problem is to identify the mapping from the problem's potential solutions into chromosomes, and to provide an appropriate objective function.

6.3 Fault Localization Measures

We consider 22 different fault-localization measures to be composed into a composite one. These include 2 measures proposed in state-of-the-art fault localization studies, namely Tarantula [45] and Ochiai [2]. The other 20 are taken from an empirical study by Lucia et al. [67] which investigates 20 association measures from the data mining community for fault localization.

6.3.1 Tarantula

Jones and Harrold propose Tarantula [45] to rank program elements based on their suspiciousness scores. Intuitively, a program element is more suspicious if it appears in failed executions more frequently than in correct executions.

To define the suspiciousness score of Tarantula, we first define some notations: Given a program P and a test suite for P, n is the total number of test cases in the test suite; $n(e)$ is the number of test cases that run through a particular element e in P; n_s is the number of test cases that succeed; n_f is the number of test cases that fail; $n_s(e)$ is the number of test cases that run through a particular element e and succeed; $n_f(e)$ is the number of test cases that run through a particular element e and fail.

Based on the above notations, Tarantula's suspiciousness score for a program element e is as follows:

$$suspiciousness_T(e) = \frac{\frac{n_f(e)}{n_f}}{\frac{n_s(e)}{n_s} + \frac{n_f(e)}{n_f}}$$

6.3.2 Ochiai

Abreu et al. [4] propose Ochiai measure as the suspiciousness score for a program element e. It is defined as:

$$suspiciousness_O(e) = \frac{n_f(e)}{\sqrt{n_f(n_f(e) + n_s(e))}} = \frac{n_f(e)}{\sqrt{n_f n(e)}}$$

Similar to Tarantula, Ochiai considers an element to be more suspicious if it occurs more frequently in failed executions than in correct executions (the $\sqrt{\frac{n_f(e)}{n(e)}}$ part). In addition, it considers an element to be more suspicious if it occurs frequently in failed executions (the $\sqrt{\frac{n_f(e)}{n_f}}$ part).

6.3.3 Association Measures

Lucia et al. study 20 association measures from the data mining community and investigate their effectiveness in fault localization tasks [67]. The 20 association measures include: ϕ coefficient, odds ratio, Yule's Q, Yule's Y, Cohen's Kappa, J-Measure, Gini index, Support, Confidence, Clark and Boswell's Laplace accuracy, conviction, interest, cosine, Piatetsky-Shapiro's Leverage, certainty factor, added value, collective strength, Jaccard, Klosgen, and information gain. The formulas for these association measures, which compute the strength of association between two variables A and B, are given in Table 6.1.

An association measure quantifies the degree of relationship between two variables. In our case, we want to measure the association between the execution of a program element (A) and failure (B) or non-execution of a program element (A) and failure (B). The strength of association is then used as a suspiciousness score computed by the formula in Definition 2 which in turn uses the notation in Definition 1.

Definition 1 (Association Measure). *We define $\alpha_i\ (x.EX, FAIL)$ to be the association between the executions of program element x and failure computed using the i^{th} association measure defined in Table 6.1. Similarly,*
$\alpha_i(x.\overline{EX}, FAIL)$ refers to the association between the non-execution of program element x and failure computed using the i^{th} association measure.

Table 6.1: Definitions of Association Measures. A and B are the two variables. $P(A)$ and $P(B)$ correspond to the probabilities of A and B respectively. Other notations follow standard notations from probability and statistics: $P(\overline{A})$ is the probability of $not\ A$; $P(A, B)$ is the joint probability of A and B; $P(A|B)$ and $P(B|A)$ are conditional probabilities.

Name	Formula
ϕ-Coefficient (M_1)	$\frac{P(A,B)-P(A)P(B)}{\sqrt{P(A)P(B)(1-P(A))(1-P(B))}}$
Odds ratio (M_2)	$\frac{P(A,B)P(\overline{A},\overline{B})}{P(A,\overline{B})P(\overline{A},B)}$
Yule's Q (M_3)	$\frac{P(A,B)P(\overline{AB})-P(A,\overline{B})P(\overline{A},B)}{P(A,B)P(\overline{AB})+P(A,\overline{B})P(\overline{A},B)} = \frac{\alpha-1}{\alpha+1}$
Yule's Y (M_4)	$\frac{\sqrt{P(A,B)P(\overline{AB})}-\sqrt{P(A,\overline{B})P(\overline{A},B)}}{\sqrt{P(A,B)P(\overline{AB})}+\sqrt{P(A,\overline{B})P(\overline{A},B)}} = \frac{\sqrt{\alpha}-1}{\sqrt{\alpha}+1}$
Kappa (M_5)	$\frac{P(A,B)+P(\overline{A},\overline{B})-P(A)P(B)-P(\overline{A})P(\overline{B})}{1-P(A)P(B)-P(\overline{A})P(\overline{B})}$
J-Measure (M_6)	$\max(P(A,B)\log(\frac{P(B\|A)}{P(B)}) + P(A\overline{B})\log(\frac{P(\overline{B}\|A)}{P(\overline{B})}),$ $P(A,B)\log(\frac{P(A\|B)}{P(A)}) + P(\overline{A}B)\log(\frac{P(\overline{A}\|B)}{P(\overline{A})}))$
Gini Index (M_7)	$\max(P(A)[P(B\|A)^2+P(\overline{B}\|A)^2]+P(\overline{A})[P(B\|\overline{A})^2+P(\overline{B}\|\overline{A})^2]-P(B)^2-P(\overline{B})^2, P(B)]P(A\|B)^2+P(\overline{A}\|B)^2]+P(\overline{B})[P(A\|\overline{B})^2+P(\overline{A}\|\overline{B})^2]-P(A)^2-P(\overline{A})^2)$
Support (M_8)	$P(A, B)$
Confidence (M_9)	$\max(P(B\|A), P(A\|B))$
Laplace (M_{10})	$\max(\frac{P(A,B)+1}{P(A)+2}, \frac{P(A,B)+1}{P(B)+2})$
Conviction (M_{11})	$\max(\frac{P(A)P(\overline{B})}{P(A\overline{B})}, \frac{P(B)P(\overline{A})}{P(B\overline{A})})$
Interest (M_{12})	$\frac{P(A,B)}{P(A)P(B)}$
Cosine (M_{13})	$\frac{P(A,B)}{\sqrt{P(A)P(B)}}$
Piatetsky-Shapiro's (M_{14})	$P(A, B) - P(A)P(B)$
Certainty Factor (M_{15})	$\max(\frac{P(B\|A)-P(B)}{1-P(B)}, \frac{P(A\|B)-P(A)}{1-P(A)})$
Added Value (M_{16})	$\max(P(B\|A) - P(B), P(A\|B) - P(A))$
Collective Strength (M_{17})	$\frac{P(A,B)+P(\overline{AB})}{P(A)P(B)+P(\overline{A})P(\overline{B})} \times \frac{1-P(A)P(B)-P(\overline{A})P(\overline{B})}{1-P(A,B)-P(\overline{AB})}$
Jaccard (M_{18})	$\frac{P(A,B)}{P(A)+P(B)-P(A,B)}$
Klosgen (M_{19})	$\sqrt{P(A,B)}\max(P(B\|A) - P(B), P(A\|B) - P(A))$
Information Gain (M_{20})	$(-P(B)\log P(B) - P(\overline{B})\log P(\overline{B}))-$ $(P(A) \times (-P(B\|A)\log P(B\|A)) - P(\overline{B}\|A)\log P(\overline{B}\|A)-$ $P(\overline{A}) \times (-P(B\|\overline{A})\log P(B\|\overline{A})) - P(\overline{B}\|\overline{A})\log P(\overline{B}\|\overline{A})))$

Definition 2 (**Suspiciousness$_i$(e)**). *If e is a control block (e.g.,* `if` *or* `while` *statement), and $children$ is the list of direct children of e in the control flow graph of the containing program, the* suspiciousness$_i$ *of e is the maximum of the following values:*

1 $\alpha_i(e.EX, FAIL)$

2 $max_{c \in children}.\ \alpha_i(c.\overline{EX}, FAIL)$

Otherwise, the suspiciousness$_i$ *of e is* $\alpha_i(e.EX, FAIL)$.

Normalization. The suspiciousness scores corresponding to the various association measures have different ranges. Some range from 0 to 1, others from $-\infty$ to ∞, etc. We normalize them to the range of 0 to 1.

6.3.4 Examples and Motivation

Let's first investigate the block-hit program spectra in Figure 6.1. The following paragraphs describe how suspiciousness scores are computed based on this spectra.

Using Tarantula's measure, the suspiciousness score of Block 3 that contains the bug is $\frac{1/1}{3/3+1/1} = 0.5$. Since Block 1 is executed by the same set of test cases as Block 3, it also has the same suspiciousness score. Block 2 receives the highest suspiciousness score: $\frac{1/1}{2/3+1/1} = 0.6$. Following the same calculation, Blocks 4–6 are not suspicious since their scores are zero. Thus, following an ordering based on the suspiciousness scores, the bug in Block 3 can be localized after 3 blocks have been investigated[3].

Using Ochiai's measure, the suspiciousness score of Block 3 is $1/(\sqrt{1 \times 4})$=0.5. Block 1 also receives the same score. Similar to Tarantula, Ochiai also returns Block 2 as the most suspicious one (the suspiciousness score is $1/(\sqrt{1 \times 3}) = 0.58$), while the remaining blocks are not suspicious. Like Tarantula, Ochiai can localize the bug

[3]We consider the worst case.

Block ID	Code	T1	T2	T3	T4
1	int count; int n; Ele *proc; List *src_queue, *dest_queue; if (prio >= MAXPRIO) /*maxprio=3*/	•	•	•	•
2	{n=2;}	•			

3	src_queue = prio_queue[prio];
	dest_queue = prio_queue[prio+1];
	count = src_queue->mem_count;
	if (count > 0) /* Bug*//* supposed : count>1*/ {
		•	•	•	•

4	n = (int) (count*ratio + 1); proc = find_nth(src_queue, n);	•	•	•	•
	Test Case's Success (S)/Fail (F) Status:	S	F	F	F

Figure 6.4: Sample block-hit spectrum - No. 2

after 3 blocks have been investigated.

Next, we illustrate how the other 20 association measure based suspiciousness scores are computed. We can observe that Blocks 1, 3, and 5 control Blocks 2, 4, and 6, respectively, and the bug at the `if` condition belongs to Block 3. The suspiciousness score of Block 3 is determined by the maximum of the association strength between executing Block 3 and failure and the association strength between *not* executing Block 4 and failure. For example, by using one of the 20 association measures, i.e., *Cosine*, the suspiciousness score of Block 1 is $max(\frac{0.25}{\sqrt{1\times0.25}}, \frac{0}{0.25\times0.25}) = 0.5$; the suspiciousness score of Block 2 is 0; the suspiciousness score of Block 3 is $max(\frac{0.25}{\sqrt{1\times0.25}}, \frac{0.25}{\sqrt{0.5\times0.25}}) = 0.71$; the suspiciousness score of Block 4 is 0; the suspiciousness score of Block 5 is 0.71. Blocks 6 and 7 have suspiciousness scores of 0. There is no need to normalize cosine's suspiciousness score as its range is from zero to one. Cosine can localize the bug after 2 blocks (Blocks 3 and 7) have been investigated. Thus for the program, bug, and set of test cases shown in Figure 6.1, Cosine is better than Tarantula and Ochiai.

However, this is not true for all possible programs, bugs, and set of test cases. Consider a slightly different program with a slightly different bug and different program spectra in Figure 6.4. Tarantula and Ochiai would rank the blocks (in descending order of suspiciousness scores) as: (Block 3, Block 4), Block 1, and Block 2[4]. Cosine would rank the blocks (in descending order of suspiciousness

[4]Blocks 3 and 4 have the same $suspiciousness_O$ and $suspiciousness_T$ scores.

scores) as: Block 1, (Block 3, Block 4), and Block 2. Thus, for this case, Tarantula and Ochiai are better than Cosine.

The above shows that a fault localization measure could perform better for some programs, bugs and spectra, and yet perform worse on other programs, bugs, and spectra. Thus, there is a need to compose various association measures. By composing different association measures the weakness of one measure could be addressed by the strengths of other measures. We empirically demonstrate this in Section 6.5.

6.4 Search-Based Composition Engine

Our search-based fault localization process is divided into two phases: training and deployment. The two phases are illustrated in Figures 6.5 & 6.6.

6.4.1 Training Phase

In the training phase, we take as input the measures described in Section 6.3 along with a training dataset. As training data, we take a set of program spectra along with actual bug locations. This training data could be obtained in practice from past fault localization efforts or past bugs encountered in one or more systems. Based on the 22 measures and the training data, the search algorithm would investigate various composite measures and search for the one that performs best in localizing faults based on the set of program spectra and bugs in the training dataset.

In the following paragraphs, we first define the search space of potential composite measures. Next, we describe how we adapt existing search algorithms to the problem of fault localization. The output of the training phase is a near optimal composite measure that performs well for the training data.

Search Space

We first define the search space of potential composite measures. There could be many possible ways to combine the individual spectrum-based fault localization

measures. In this work, we employ one of the many possible composition strategies, which is a weighted sum of the individual fault localization measures. Given 22 measures named as M_1, M_2, ..., M_{22}, and a program element e, the suspiciousness score for e assigned by the composite measure is defined as follows:

$$M_{Composite}(e) = w_1 \times M_1(e) + w_2 \times M_2(e) + \ldots + w_{22} \times M_{22}(e)$$

The search space of all possible compositions corresponds to the various assignments of weights, *i.e.*, $w_1, w_2, \ldots, w_{22}$. We consider a weight to be a real number from zero to one. Many search algorithms however only work on discrete search space, thus we discretize the search space by representing each real number weight by 7 bits. The function mapping the 7 bit signature to weight is:

$$BitToWeight(b_1 b_2 b_3 b_4 b_5 b_6 b_7) = \frac{\Sigma b_i \times 2^i}{2^7 - 1}$$

The more bits we have the more accurate the weight would be represented; still, we believe 7 bits is good enough to represent each weight. The size of the search space for the weights of the 22 measures would thus be equal to $(2^7)^{22}$ which is a huge number.

Objective Functions

Search algorithms require an objective function to measure how good a candidate solution is. As introduced in Section 6.2.2, some search algorithms maximize a given objective function, while others minimize an objective function.

We thus define two types of functions: one where the lower the score is, the better the quality of a fault localization measure would be, and the other where the higher the score is, the better the quality of a fault localization measure would be. The first will be used as the objective function for SA to minimize, and the later will

be used as the objective function for GA to maximize. We define two such objective functions, denoted as f_{min} and f_{max}, as follows:

f_{min} The average percentage of program elements that need to be *investigated* to locate the bugs by traversing down a list of program elements sorted in descending order according to their suspiciousness scores generated by the measure.

f_{max} The average percentage of program elements that can be *skipped* when users of a fault localization measure try to locate bugs by traversing down a list of program elements sorted in descending order according to their suspiciousness scores generated by the measure.

From the definitions, it is clear that $f_{min} + f_{max} = 1$.

According to past studies [4, 15, 45], the accuracy of fault localization can also be measured in term of the proportion of bugs (i.e., buggy versions) successfully localized when a given percentage of program elements are inspected. The higher the proportion, the more accurate a fault localization technique would be. Based on this measure, we define two more objective functions as follows:

p_{min} The proportion of bugs missed when only 10% of program elements are inspected.

p_{max} The proportion of bugs successfully localized when only 10% of program elements are inspected.

In above definition of p_{min} and p_{max}, we set the percentage of program elements inspected to be 10% following past studies [45][5].

[5]Jones et al. compares the effectiveness of Tarantula with other techniques based on how many percentage of bugs can be found when 10% program elements are investigated [45].

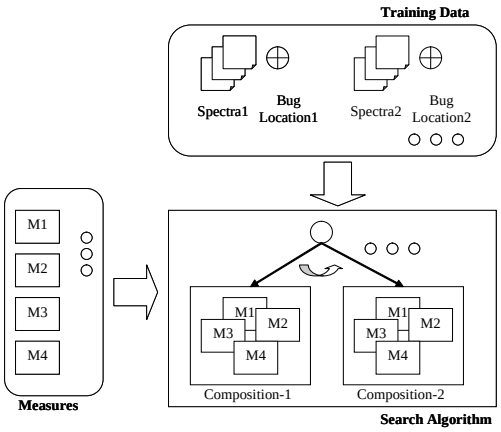

Figure 6.5: Training Phase

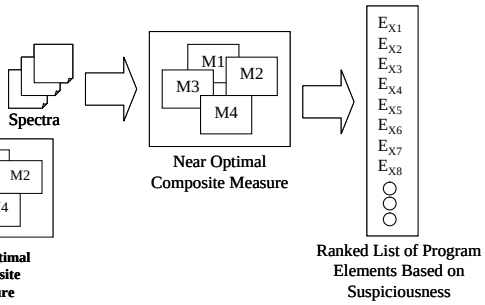

Figure 6.6: Deployment Phase

Adapting Search Algorithms

We describe how we use simulated annealing and genetic algorithm for fault localization in the following paragraphs.

Simulated Annealing. To use simulated annealing (SA) for our problem, we model the search space following the description in Section 6.4.1 which tries to minimize an objective function. We create three variants of simulated annealing algorithm each using a different objective function defined in Section 6.4.1:

SA^F Uses f_{min} as the objective function.

SA^P Uses p_{min} as the objective function.

SA^{FP} Uses $f_{min} + p_{min}$ as the objective function.

As defined in section 6.4.1, the smaller average percentage of program elements inspected to locate all bugs and the more bugs successfully located when 10% of program elements inspected, the more effective an association measure is. So, we use f_{min} and p_{min} as an objective function for the respective variant of simulated annealing (SA^F and SA^P). For the third variant SA_{FP}, since both f_{min} and p_{min} are important quality measures of fault localization techniques, we use an objective function that combines f_{min} and p_{min} and considers both of them as equally impor-

tant. Our intention is to investigate whether SA^{FP} could have a better performance than the other two variants.

Similar to previous work employing search-based techniques in software engineering tasks [59], we use small-scale initial experiments to find out the parameters (T, L, R, and I used in Figure 6.2) that could empirically lead to a better performing SA. In our case, we choose to use T=0.005, L=0.0006, R=0.9, and I=100.

Genetic Algorithm. To use a genetic algorithm (GA) for our problem, we also model the search space as described in Section 6.4.1 which tries to maximize an objective function. We concatenate the 22 7-bit weights together to form a 7×22=154 bits chromosome. The search space is thus all possible binary chromosomes of length 154.

A genetic algorithm requires of the creation of an initial population, and the selection, crossover, and mutation operations. We create the initial population of chromosomes randomly. The crossover and mutation operations are based on [35]. We only adapt the selection step of GA.

The selection operation of GA selects the best chromosomes, according to a *fitness function*(as shown below), to be used as the population for the next generation. Considering obj to be an objective function that needs to be maximized, we define the fitness of a chromosome X_i in a generation R containing $|R|$ chromosomes as the chromosome's relative performance against other chromosomes as follows:

$$Fitness(X_i) = \sum_{j=1}^{|R|} e^{10 \times (obj(X_j) - obj(X_i))}$$

Since the difference between $obj(X_j)$ and $obj(X_i)$ is rather small, we take the exponentiation of this difference multiplied by ten. Similar to what we use for SA, we consider three objective functions defined in Section 6.4.1 for GA:

GA^F Uses f_{max} as the objective function obj.

GA^P Uses p_{max} as the objective function.

GA^{FP} Uses $f_{max} + p_{max}$ as the objective function.

Based on these objective functions, we consider the following selection strategies to be used in GA:

1. Random. In this selection strategy, each chromosome of the current generation has a chance to be selected to be a chromosome in the next generation proportional to its fitness score. As we have three objective functions, we have three variants: GA^F_{Random}, GA^P_{Random}, and GA^{FP}_{Random}.

2. Enhanced. In this strategy, we first select all chromosomes whose fitness scores are no less than the 90% percentile of the fitness scores of all chromosomes in the generation. The other chromosomes are randomly selected for the later generation as in the random strategy. As we have three objective functions, we have three variants: $GA^F_{Enhanced}$, $GA^P_{Enhanced}$, and $GA^{FP}_{Enhanced}$.

Also, we use small-scale initial experiments to fine-tune the parameters of GA (N, I, M, and C in Figure 6.3). In our case, we choose to use N=50, I=200, M=0.01, and C=0.6.

6.4.2 Deployment Phase

In the deployment phase, we use the composite measures constructed in the training phase to locate faults. Given a program spectrum, a composite measure would assign a suspiciousness score to each program element. These suspiciousness scores are used to rank the program elements based on their likelihood to be faults, as individual measures would do. The process is illustrated in Figure 6.6. If a composite

measure is deployed and used, the ranked list of program elements would be the output for the user to inspect to find the location of the fault.

In our experiments, we need to quantify the effectiveness of each fault localization measure by quantifying how accurate the ranked lists from each measure are for locating actual faults. We carry out our evaluation on programs with known bugs in Section 6.5.

6.5 Empirical Evaluation

This section describes our experimental settings and evaluation results.

6.5.1 Settings

Subject Programs

We use the seven programs from the Siemens Test Suite [38] and three larger programs: Space [90], NanoXML and XML-Security [93] downloaded from the Software-artifact Infrastructure Repository [17] (`http://sir.unl.edu/`).

The Siemens test suite contains seven programs: print_tokens, print_tokens2, replace, schedule, schedule2, tcas, and tot_info. Each program contains many versions where each version has one bug. These bugs represent a wide array of realistic bugs. The total number of buggy versions are 132. Among these versions, we use 120 versions. We exclude versions that contain bugs in global variable declarations since our instrumentation tool cannot collect program spectra that record declarations. These versions include versions 6, 10, 19, and 21 of tot_info, version 12 of replace, and versions 13, 14, 15, 36, 38 of tcas. Since versions 4 and 6 of print_token are identical with the original version, we exclude these versions as well. Thus, the total number of buggy versions from the Siemens test suite that we take into consideration is 120.

The average lines of code for Space, NanoXML, and XML-Security are 6,218,

Table 6.2: Dataset Details

Dataset	**LOC**	**Faulty versions**	**Test cases**
print_token	472	5	4030
print_token2	399	10	4115
replace	512	31	5542
schedule	292	9	2650
schedule2	301	10	2710
tcas	141	36	1608
tot_info	440	19	1051
space	6,218	35	13,585
NanoXML_v1	3,497	7	214
NanoXML_v2	4,007	7	214
NanoXML_v3	4,608	9	216
NanoXML_v5	4,782	8	216
XMLSec_v1	21,613	6	92
XMLSec_v2	22,318	6	94
XMLSec_v3	19,895	4	84

4,223, and 21,275 respectively. Space is an interpreter for Array Definition Language (ADL) developed for European Space Agency. We analyze 35 faulty versions of Space. NanoXML is an XML parser written in Java. We analyze 31 faulty versions of NanoXML. XML-Security is a Java library which includes a mature Digital Signature and Encryption implementation. We analyze 16 faulty versions of XML-Security.

Thus, the combined number of versions is 202. Details about these datasets are provided in Table 6.2.

Instrumentation

We have instrumented the programs and run them with the provided test cases to collect program block-hit spectra. Each block-hit spectrum identifies how many times each block is executed when the program is executed with the test case.

Effectiveness Calculation

As introduced in Section 6.4.1, we measure the effectiveness of a fault localization technique by using: 1) the average percentage of program elements that need to be inspected to locate all bugs, and 2) the proportion of bugs successfully localized when only a given proportion of program elements are inspected. The first measure has been used in past study [67], while the second one has been used in past studies [4, 15, 45]. Similar to previous studies, we assume that an inspector could recognize a bug whenever he or she sees the faulty line of code.

To compute the first measure, which corresponds to the objective functions f_{min} and f_{max}[6] we perform the following steps. First, for each program, we decreasingly sort all elements in the program based on their suspiciousness scores generated by each fault localization measure, and count the percentage of elements that have scores greater than or equal to the score for the buggy element. Next, we calculate the average percentage for all programs.

To compute the second measure, which corresponds to the objective functions p_{min} and p_{max}, we perform the following steps. First, for each program, we sort the program elements as with the computation of the first measure. Next, given an ordering for each program, we count the number of located bugs (*i.e.*, contained in the top $x\%$ of the program elements in the ordering, where x is a user-defined parameter). Finally, we calculate the percentage of located bugs in all programs.

[6]Note that f_{min} and f_{max} are *dual* of each other—each can be derived from the other. This is also the case with p_{min} and p_{max}.

Cross Validations

We perform a 10-fold cross validation (i.e., 9/10 of the dataset is used for training and 1/10 is used for testing) [28] to evaluate our compositional fault localization approach. We randomly split the 202 versions into 10 buckets, each bucket containing 20 or 21 versions. In 10-fold cross validation, 10 separate iterations are performed. In each iteration, we keep one of the buckets as the test/validation set and use the other nine buckets for training. The results reported in the following subsection are based on the average of the 10 iterations of the cross validation process.

6.5.2 Evaluation Results

Fault Localization Effectiveness

We measure the accuracies of the nine variants of our proposed search-based approach, namely, SA^F, SA^P, SA^{FP}, GA^F_{Random}, GA^P_{Random}, GA^{FP}_{Random}, $GA^F_{Enhanced}$, $GA^P_{Enhanced}$, and , $GA^{FP}_{Enhanced}$, and compare them with those of Tarantula [45], Ochiai [4], and Information Gain [67][7].

Proportion of Bugs Localized. Looking from an angle, where developers need to localize *all* faults, we examine the mean and standard deviation of the percentages of blocks that need to be inspected to localize all faults for each technique. The results are shown in Table 6.3. $GA^{FP}_{Enhanced}$ localizes all bugs with the lowest average percentage of blocks inspected, which is 20.6%, followed by $GA^F_{Enhanced}$. The relative improvements of $GA^{FP}_{Enhanced}$ over Tarantula, Ochiai, and Information Gain are 23.7%, 18.6%, and 37.6% respectively.

To check whether the differences among these means are statistically significant, we perform the commonly used Wilcoxon signed-rank test [115] for each pair of fault localization techniques with p-value set to 0.05. We notice that $GA^{FP}_{Enhanced}$ statistically significantly improves Tarantula, Ochiai, and Information Gain. We al-

[7]Information gain has been shown in [67] to perform on average better than other association measures for the Siemens test suite.

Table 6.3: Means and Standard Deviations of Percentages of Inspected Blocks (Smaller is Better)

Techniques	**Mean**	**Standard Dev.**
$GA^{FP}_{Enhanced}$	20.6%	23.4%
$GA^{F}_{Enhanced}$	20.6%	23.3%
$GA^{P}_{Enhanced}$	20.9%	23.9%
GA^{FP}_{Random}	20.7%	23.3%
GA^{F}_{Random}	20.7%	23.4%
GA^{P}_{Random}	21.3%	23.5%
SA^{FP}	21.2%	23.6%
SA^{F}	21.2%	23.6%
SA^{P}	21.4%	23.6%
Ochiai	25.3%	25.9%
Information Gain	33.0%	35.4%
Tarantula	27.0%	26.8%

so notice that GA-based techniques statistically significantly outperform SA-based techniques.

Percentage of Program Elements Inspected. Here, the accuracy for each fault localization technique is measured in terms of the number (or percentage) of bugs that could be localized by only examining a certain number (or percentage) of program elements (blocks in our case).

We plot the accuracies of the various techniques in Figures 6.7, 6.8, 6.9, 6.10. The x-axis describes the percentage of blocks inspected and y-axis describes the percentage of buggy versions that can be localized. Assuming an inspector can recognize a bug when a block is presented to him or her, the plot lines shall always reach 100% at the right ends. The more buggy versions localized at lower numbers of inspected blocks, the better a fault localization approach is.

We find that $GA^{FP}_{Enhanced}$ performs better than all other techniques. When

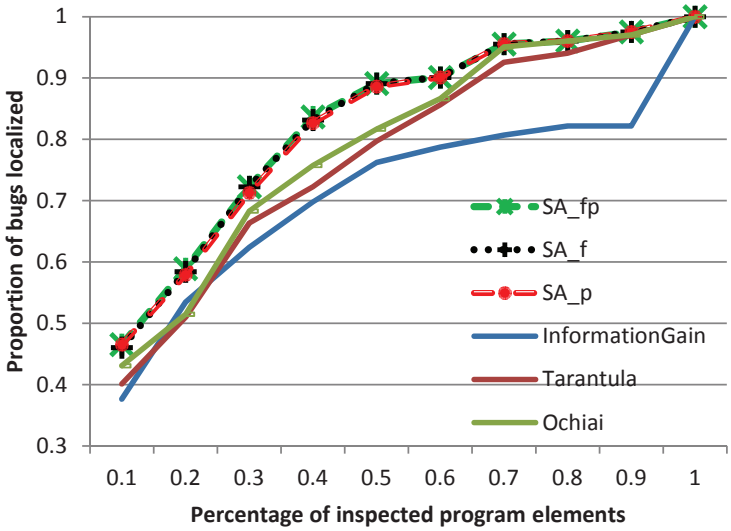

Figure 6.7: Accuracies of SA versus Baseline Approaches

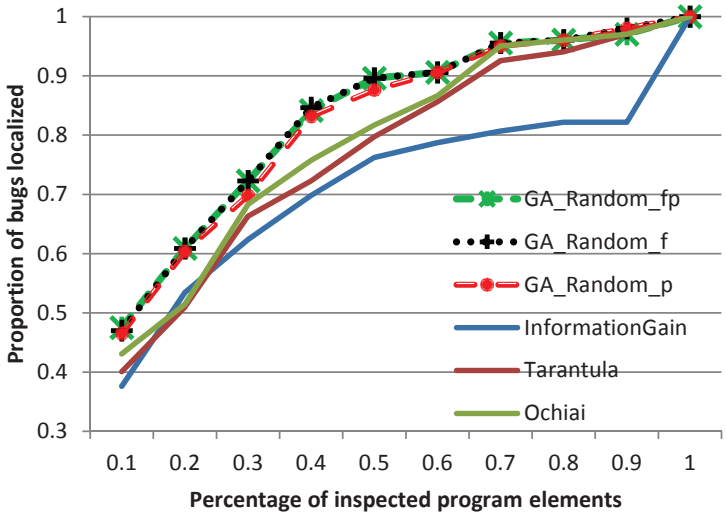

Figure 6.8: Accuracies of GA_{Random} versus Baseline Approaches

less than 10% of the blocks are inspected, $GA^{FP}_{Enhanced}$ localizes 49.0% of all buggy versions. At the same percentage of inspected blocks, $GA^{F}_{Enhanced}$, $GA^{P}_{Enhanced}$,GA^{F}_{Random}, GA^{FP}_{Random}, GA^{P}_{Random}, SA^{F}, SA^{P} and SA^{FP} localize 47.0%, 46.5%, 47.0%, 47.5%, 46.5%, 46.0%, 46.5% and 46.5%, of the bugs respectively, while Ochiai, Tarantula and Information Gain localize 43.1%, 40.1%, and 37.6% respectively. Thus, the relative improvements of $GA^{FP}_{Enhanced}$ over Tarantula, Ochiai, and Information Gain are 22%, 14%, and 30%, respectively.

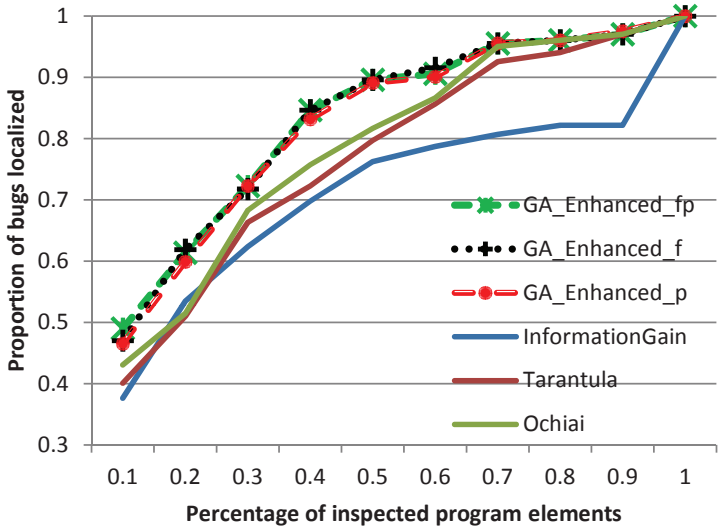

Figure 6.9: Accuracies of of $GA_{Enhanced}$ versus Baseline Approaches

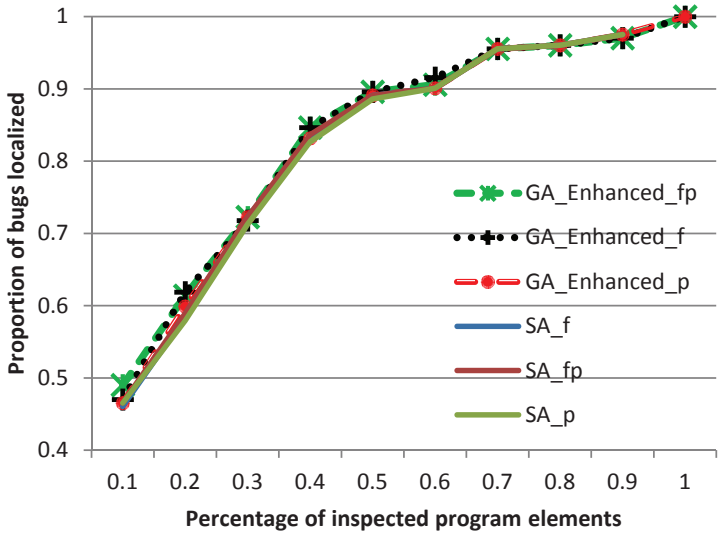

Figure 6.10: Accuracies of of $GA_{Enhanced}$ versus SA

Furthermore, 62% of the buggy versions were localized by $GA^{FP}_{Enhanced}$ and $GA^{F}_{Enhanced}$, $GA^{P}_{Enhanced}$, GA^{F}_{Random}, GA^{FP}_{Random} and GA^{P}_{Random} with only up to 20% of blocks inspected. To achieve the same number of localized bugs, SA^{F}, SA^{P} and SA^{FP} need at least 22%, 23%, 22% of block inspections respectively, while Tarantula, Ochiai, and Information Gain requires at least 27%, 26%, 29% respectively. Also, our nine various versions of search-based approaches can localize more than 90% of the bugs when up to 60% code elements are inspected. Ochiai,

Table 6.4: Proportion of bugs localized when 10% of the code is investigated (PBL) and average percentage of basic blocks investigated for all bugs to be localized (ABB), when the amount of training data is varied from 10% to 90%. The remaining data is used for testing.

	10%	**30%**	**50%**	**70%**	**90%**
PBL	47.8%	46.1%	49.5%	43.0%	50.0%
ABB	20.8%	21.4%	19.4%	20.6%	23.5%

Tarantula, and Information Gain require 64%, 66%, and 94% respectively.

Training Data Sensitivity Analysis

We also experiment by varying the amount of training and test data – from 10% to 90% training data. We experiment with $GA^{FP}_{Enhanced}$ which is the best-performing composite measure. The training data is randomly selected from the pool of 202 versions of the 10 programs. The remaining versions are used as test data. We analyze the results in terms of proportion of bugs localized when 10% of the code is investigated (PBL), and the average percentage of basic blocks investigated for all bugs to be localized (ABB). The results are shown in Table 6.4. The results show that our approach works well with varying amount of training data used. With just 10% of all bugs, our composite measures still achieve on average better fault localization accuracies than previous individual approaches. In addition, the results imply that our compositional fault localization works better than the constituent approaches across different kinds of bugs across different programs, which is further evaluated in the next subsection. With the accumulation of many bug databases, the sensitivity to the training data may become less an issue.

Effectiveness for Various Kinds of Bugs

We divide the bugs into eight categories; our categorization is based on that by Kim et al. [80]. Kim et al. categorize different fixes that are made to various software systems. We categorize bugs based on how the bugs get fixed. The following

Table 6.5: Bug Categories

Bug category	**# of versions**
Change of loop predicate (LP-CC)	11
Change in method calls (CH-MC)	17
Change of assignment expression (AS-CE)	65
Change of return expression (CH-RET)	14
Change of if condition expression (IF-CC)	55
Addition/removal of non-conditional statement (CH-NCS)	16
Addition/removal of conditional statement (CH-CS)	23
Others (OTH)	1

paragraphs describe our bug categories and present the effectiveness of our compositional approach. We add two categories that are not included in the classification of Kim et al. [80], i.e., Change of return expression (CH-RET) and others (OTH). Table 6.5 lists these eight bug categories.

In order to test the effectiveness of the compositional approach across different categories of bug, we do a $|B|$-fold cross validation, where $|B|$ is the number of categories of bug. We put all buggy versions of one category into a bucket, then during the $|B|$-fold cross validation, we pick one category of buggy versions as the test data and the rest are used as the training data.

The results for each bug category are shown in the table 6.6. The results show that our approach performs better for most types of bugs, and on average our approach finds all bugs when 20.5% of the blocks are inspected, while Tarantula, Ochiai, and Information Gain need inspection of 27.0%, 25.3%, and 33.0% respectively. Also, our approach localizes 47.5% of all bugs, while Ochiai, Tarantula and Information Gain localize 43.1%, 40.1%, and 37.6% respectively. Using Wilcoxon signed-rank test with p-value set to 0.05, we find that the differences between $GA^{FP}_{Enhanced}$ and Tarantula, Ochiai, and Information Gain, are statistically significant.

Table 6.6: The comparison of our approach($GA^{FP}_{Enhanced}$), Tarantula, Ochiai and Information Gain in terms of the average percentage of elements inspected to find all bugs.

Bug category	**Our approach**	**Information Gain**	**Tarantula**	**Ochiai**
LP-CC	13.9%	17.9%	11.2%	15.8%
CH-MC	34.9%	52.6%	33.9%	38.4%
AS-CE	21.7%	31.5%	30.8%	27.7%
CH-RET	22.5%	31.9%	26.0%	27.8%
IF-CC	14.9%	30.0%	22.8%	19.5%
CH-NCS	13.4%	26.2%	22.5%	16.0%
CHCS	24.4%	37.4%	30.5%	29.8%
OTH	74.2%	100.0%	74.0%	74.0%
Average	20.5%	33.0%	27.0%	25.3%

We note that the effectiveness of individual fault localization measures can vary for different bug types, and so are for composite measures. A composite measure may not be better than individual measures for some bug types. For example, for LP-CC bugs (see Table 6.6), the composite measure is trained on other bug types that Ochiai performs the best on average during $|B|$-fold cross validation, which leads to a much higher weight for Ochiai than the weight for Tarantula, even though Tarantula performed better than Ochiai for LP-CC bugs. This causes the composite measure to perform worse than Tarantula for LP-CC bugs, even though the composite measures for different bug types on average perform better than individual measures. This observation indicates that our approach is better to be applied to training data containing various types of bugs to improve the generalizability of composite measures, and we further perform cross-program validation in Section 6.5.1.

We give some further observations that may indicate how the effectiveness of individual measures affects that of composite measures for different bug types. Firstly, the effectiveness of each individual measure is built on the basic intuition for using

association measures to locate bugs: the execution profiles of buggy program elements may be quite different from those of correct program elements. Thus, the effectiveness of an association measure may decrease if the profiles of a buggy element are about the same as those of correct elements, or if the profiles of the buggy element from failed test cases are about the same as its profiles from successful test cases. Based on our observation, certain bugs do not change the profiles of the buggy elements, while the profiles of some elements executed after the buggy elements may be affected, causing less accurate localization. For example, here is a sample buggy code fragment from NanoXML:

```
this.systemID = systemID;
this.lineNr = lineNr;
this.encapsulatedException = null;
...
```

The third statement assigns a “null” to “this.encapsulatedException”, while it should be assigned a non-null object. The profiles of this buggy statement from both failed and successful test cases are the same, although it affects the profiles of some correct statements that are executed after it but at locations far away from it. The suspiciousness scores for those correct statements are higher than the buggy statement, causing lower localization accuracy. The statements in this example are all in the same block of control flow graph, and they always have the same block-hit spectra and thus the same suspiciousness score, causing the buggy statement to be ranked even lower. Generally speaking, when there are many correct statements having the same profiles as the buggy statement, localization accuracy for the bug would also be decreased since they all have the same score and our evaluation metric counts all elements that have scores *greater than or equal to* the score of the buggy element. On the other hand, if the correct statements whose profiles are affected by the buggy statements are spatially close to the bug, or if the suspiciousness scores of those correct statements can be ”propagated” back to the buggy statement, local-

ization accuracy may be improved. We have a special propagation for most of the association measures used in this chapter (please see Definition 2 in Section 6.3): the suspiciousness scores of statements contained in a control block (e.g., if and while statement) may be propagated to the control block itself. This may help to improve accuracy if the control block is indeed buggy.

Secondly, we also note that Tarantula can perform better than Ochiai for some bug types (i.e., LP-CC, CH-RET, and CH-MC), although Ochiai is better on average. From the formula of Tarantula ($\frac{1}{1+\frac{n_s(e)/n_s}{f_(e)/n_f}}$) (see Section 4.1), we noticed that if a program element is always executed in all test cases, including failed and passed test cases (i.e., $n_s(e) = n_s$ and $n_f(e) = n_f$), the suspiciousness score is always 0.5, while the score computed by Ochiai still varies for different and as Ochiai's formula ($\frac{n_f(e)}{\sqrt{n_f n(e)}}$ in Section 4.2) can be simplified to $\sqrt{\frac{n_f(e)}{n(e)}}$ under the above condition. A number of program statements (e.g., constructors, initialize functions, etc.) are executed by all test cases, which led to 0.5 for their suspiciousness scores from Tarantula, while their scores from Ochiai change based on available failed and passed test cases for the programs.

General Applicability of Composite Models

In order to test the reusability of composite models, we do a $|P|$-fold cross-program validation, where $|P|$ is the number of programs. Different from random 10-fold cross validation, here we put all of the buggy versions of one program into a bucket, instead of randomly assigning them into a bucket. Then, during each iteration of the cross-program validation, all versions of one program are used as the test/validation set, the other programs are used as the training set.

Our results show that when less than 10% of the blocks are inspected, $GA^{FP}_{Enhanced}$ localizes 45.5% of all bugs, while Ochiai, Tarantula and Information Gain localize 43.1%, 40.1%, and 37.6% respectively. Also, $GA^{FP}_{Enhanced}$ finds all bugs when on average 21.3% of the blocks are inspected, while Tarantula, Ochiai, and Information Gain need inspection of 25.3%, 27.0%, and 33.0% respective-

ly. Using Wilcoxon signed-rank test with p-value set to 0.05, we find that the differences between $GA^{FP}_{Enhanced}$ and Tarantula, Ochiai, and Information Gain, are statistically significant. These results show that our compositional fault localization approach can produce composite measures that perform better than constituent techniques when they are trained on a set of programs but applied to different programs. With the accumulation of bug databases from various programs containing various types of bugs, the sensitivity to the training data may become less an issue, and it may be likely that we could have a compositional fault localization engine in the future that is able to adjust composition strategies with respect to available programs and spectra to locate various bugs in various programs more accurately than constituent techniques.

Discussion

In our initial experiments, we have tested several parameters used for our GA and SA. We find no substantial difference in the performance of GA, but a more substantial difference in the performance of SA. The average time to learn a composite measure using the various variants of search-based approaches are within 22 minutes (ranging from 18 minutes to 28 minutes for the iterations in the random 10-fold cross validation). Although this might seem long, it can be done off-line and no need to train models frequently. In practice, we only need to retrain a composite measure when either much more training data becomes available or the accuracy of the composite measure becomes unacceptably low due to significantly changing programs or bug types. Applying a composite measure to a program spectra is not expensive as it only involves computing a weighted sum of computationally inexpensive mathematical formulas—typically it takes less than a second.

We have also noticed that some constituent fault localization measures are often (but not always) assigned zero weights during our experiments, which implies that they often perform worse than others. In the future, we may remove constituent measures that do not perform well from our approach to help reduce search space.

On the other hand, we shall note that if there is an individual fault localization technique that could perform better than all other techniques under all circumstances, we do not need to employ compositional fault localization. However, based on our experience in cross-program and cross-bugtype validations and the literature, no such panacea exists yet. In the future, it may be worthwhile to investigate deeper into the reasons why certain techniques perform better or worse under certain situations, and consider both improving individual fault localization techniques that can be tailored for particular bugs in particular programs and improving the ways the individual fault localization techniques are composed.

6.5.3 Threats to Validity

Threat to construct validity corresponds to the appropriateness of our performance metrics. We use two metrics to measure the quality of fault localization measures: the percentage of bugs successfully localized when a certain amount of code is examined, and the average percentage of program elements (*i.e.*, code) that need to be investigated to locate all bugs. These two metrics have been used in many past studies [4, 15, 45, 67] and we believe they are suitable for evaluating fault localization techniques. It is not clear if techniques with good performance metrics scores could better help developers. In the future, it is interesting to perform a case study with real professional developers on real projects to see if indeed techniques with higher scores help developers more in debugging.

Threat to internal validity corresponds to the various biases when performing the experiments. We use existing benchmarks with a variety of datasets containing various bugs (real and seeded). We believe this greatly reduces the threat of experimenter bias. We also perform 10-fold cross validation and do not mix test with training data. Furthermore, we have also performed sensitivity analysis to see if we could achieve similar performance with less training data.

Threat to external validity corresponds to the generalizability of our results. In

this study, we have considered 10 programs. All these programs are written in C and Java, ranging from 141 to 22,318 lines of code. Various kinds of bugs exist in these datasets – 202 buggy versions. We have also performed 10-fold and $|P|$-fold cross-program validation to investigate if a model learned on a set of bugs could be used to effectively localize other bugs and the results are promising. Still, there is a threat of external validity especially on the applicability of our approach on even larger programs and programs written in other programming languages.

6.6 Conclusion and Future Work

There are many spectrum-based fault localization techniques proposed in the literature. For some programs, bugs, and program spectra, some measures are better than the others. In this chapter, we leverage this observation to build a solution that combines existing fault localization measures into improved composite measures. We model the possible compositions as a search space and use advances in the machine learning and meta-heuristics community to compute near optimal compositions based on a training set. We demonstrate that our approach can achieve better average performance than all previous spectrum-based fault localization techniques. Compared with Ochiai and Tarantula, our approach (using $GA^{FP}_{Enhanced}$) could localize at least 14% more bugs when 10% of the programs are analyzed, and reduce the average amount of code investigated by at least 18.6%. We also show that the improvements achieved by our approach is statistically significant, and our cross-program and cross-bugtype validations imply our approach could be generally applicable to various bugs in various programs with sufficient test cases.

As future work, we plan to investigate more advanced genetic algorithm solutions to search for optimal compositions. We have only considered one way to compose past fault localization measures, namely using a weighted sum of the constituent measures; we also plan to explore other composition strategies.

Chapter 7

Active Code Search: Incorporating User Feedback to Improve Code Search Relevance

7.1 Introduction

To help improving developer productivity, there is a need for automated tools that can help developers in searching through large amount of code to find relevant code for a specific task.A number of studies in the literature have proposed various code search techniques that can return pieces of code considered to be relevant to a user's query.

Based on the forms of user queries, we can categorize code search techniques into two main families. Most code search techniques, e.g., [13, 62, 77, 95], belong to the first family that takes queries as a set of natural language keywords (similar to the way one provides keywords to general-purpose web search engines, e.g. Google). Portfolio is one of the state-of-the-art techniques that processes a set of keywords from users and internally considers code structures (e.g., call graphs) to find relevant pieces of code [77]. The authors of Portfolio have compared their

approach with Koders[1] and Google Code Search[2] and demonstrated that Portfolio performs better. The second family of code search techniques allows queries in the form of an SQL-like language where users can specify certain constraints that need to be satisfied by relevant pieces of code [74, 106, 112]. In this work, we focus on the first family.

The above studies have shown that code search can be effective to aid developers in finding relevant pieces of code. However, there is still much room for improving the relevance of the search results. For example, McMillan et al. report that participants of their user study that evaluates Portfolio indicate an average precision of 65% for the search results and give an average confidence level of 2.86 out of 4 (with a 4-point Likert scale [61]) [77].

One factor that potentially affects the effectiveness of code search is the difficulty in formulating a precise query that matches the relevant code fragments needed by users. Often, users could not exactly specify what they wanted in the form of a set of keywords or constraints, but they were often able to decide whether a given code fragment satisfies their needs. Given a list of code fragments returned by a code search engine, some code fragments might be close to what a user wants, while others might be very different. As a user checks each code fragment one at a time, he or she will naturally form an opinion about its relevance. It will be beneficial to utilize user opinions on code fragments that they have checked to improve the results.

The key idea of this chapter is to improve code search relevance by integrating users' feedback. Our approach records user's opinions about each code fragment when they check through a list of results returned by a search engine and refines the list so that code fragments that are more relevant to a user's needs appear earlier in the list, which would save the user's time. Our approach can be built on top of any

[1]The service at `http://www.koders.com/` has been merged into Ohloh Code at `https://www.ohloh.net/`

[2]The service has been shut down; only an archive page is available at `http://web.archive.org/web/20101112131244/http://www.google.com//codesearch`

code search engine that takes textual descriptions as query inputs. The feedback required by our approach does not impose much additional effort on users either, as they would need to navigate the search results anyway to decide whether the results match their needs; the only difference is that they need to *explicitly* indicate their opinions about a search result when using our approach.

Specifically in this chapter, we build our approach on one of the state-of-the-art code search engines, Portfolio [77], to incorporate user feedback. Users of our approach first receive a list of search results from Portfolio for their query as usual. Then, they can provide relevance feedback on each search result when they check through the list one by one. The feedback is expressed as a label in a 4-point Likert scale, where 1, 2, 3, and 4 indicate whether the result is completely irrelevant, mostly irrelevant, mostly relevant, and highly relevant, respectively. Once the relevance feedback for a search result is recorded, our approach incorporates this feedback to re-sort the remaining search results. This is performed based on the similarity between each of the remaining results and all known relevant or irrelevant results. The goal is to place search results that are more likely to be relevant nearer to the top of the list. Our approach can also utilize the past feedback collected from previous queries to refine the current and future queries, allowing users benefit from past feedback. We refer to this as *active code search*—users can more actively affect the output of a code search engine by providing relevance feedback.

We have evaluated our approach with 70 queries on a code base containing 19,414 programs written in C and/or C++, comprised of about 169 million lines of code. Ten human participants were asked in our user study to provide relevance scores to each search result. Similar to studies on web search (c.f., [33]), we measure the overall relevance of a list of search results using Normalized Discounted Cumulative Gain (NDCG) [70], which assigns a higher score for an ordering list where results at the beginning of the list have higher relevance scores than those later in the list. Our evaluation shows that Portfolio achieves an average NDCG score of 0.738 for the 70 queries on the code base, while our active code search approach

(Portfolio^{active}) achieves an average NDCG score of 0.821.

The following are the main contributions in this chapter:

1. We present the concept of active code search where users can more actively affect code search results by providing relevance feedback.

2. We design an active code search technique that can take incremental user feedback and iteratively improve the relevance of code search results. It can be implemented on top of any existing code search engine that takes textual descriptions as query inputs.

3. We have implemented an active code search engine on top of Portfolio and evaluated it in a user study involving 70 queries, 10 participants, and a code base containing 19,414 programs and about 169 million lines of code. The resulting Portfolio^{active} can on average achieve 11.3% higher relevance scores, measured in Normalized Discounted Cumulative Gain (NDCG), than the original Portfolio.

The structure of this chapter is as follows. In Section 7.2, we describe Portfolio and introduce several concepts needed in our chapter. We present our active code search approach in Section 7.4.4 and zoom in on the refinement engine component in Section 7.4. Our empirical evaluation results are presented in Section 7.5. We finally conclude and mention future work in Section 7.6.

7.2 Portfolio & Active Code Search

In this section, we briefly summarize Portfolio, and then present the concept of active code search and introduce the metric used to measure the relevance of search results.

7.2.1 Portfolio

Portfolio is a code search engine that takes natural language descriptions as input and outputs a list of functions or code fragments along with corresponding call graphs [77]. Portfolio assumes that call graphs can help users to have a better understanding of the returned code. Portfolio internally analyzes both textual and structural (e.g., call graphs) information from code, and employs Page Rank, Vector Space Model (VSM), and Spreading Activation Network (SAN) to quality the relations among code and the given query, aiming to locate more relevant results.

Page Rank measures the global importance of a function in the whole call graph in a code base: the more other functions call this function, the more important it is. Since the Page Rank score of each function is independent from query, it could be pre-computed and stored in a database. We denote the Page Rank score for a function F_i as $PR(F_i)$, calculated by $PR(F_i) = \sum_{F_j \in B_{F_i}} \frac{PR(F_j)}{|F_j|}$,where B_{F_i} is the set of functions that invoke F_i and $|F_j|$ is the number of functions invoked by function F_j.

When a query comes in, Portfolio first uses VSM to perform keyword matching between the words in each function in the code base and the words in the query. Next, it feeds the functions returned from the previous keyword matching step along with their similarity scores to SAN, and spreads the similarity scores along the call graph. The intuition of SAN is that nodes farther away from the starting nodes should get smaller scores. New scores for different nodes are transitively computed from the starting nodes using the formula $SAN_j = \sum_i f(SAN_i * w_{ij})$, where the score of node j (denoted as SAN_j) equals to a combination of the scores SAN_i of all nodes incident to j. w_{ij} represents the strength of the connection between i and j (so called decay factor), indicating what percentage of the score for i can be propagated to j. Last, Portfolio ranks all functions in the call graph based on a combined similarity score $S = \lambda_1 PR(F) + \lambda_2 SAN(F)$, where λ_i is the interpolation weight for each type of scores, and returns the top ones as search results. Users can also

specify how many search results to get for each query.

7.2.2 Active Code Search Paradigm

Active code search in this chapter refers to a kind of code search that leverages user feedback to improve the quality of the search results for a given query.

A user first provides a query and gets a list of search results returned by a code search engine. As the user navigates through the results one by one, he or she can indicate the relevance of each result. Then, the code search engine takes the user's indications as feedback to refine and improve the list of search results. Thus, the user may get an updated list of search results that is supposed to be more relevant to the given query. It should also allow users to incrementally provide feedback, and ideally does not impose additional effort on the user and is able to take feedback at the right intervals to incrementally improve search results and user experience.

In this chapter, we also aim to make our approach applicable to any code search engine that takes user queries in the form of textual descriptions, and design our active code search technique as an additional layer on top of a list of search results from an existing code search engine. After taking in user feedback, the refinement and improvement of the search results are done without the need for additional queries to the existing code search engine: the initial list of search results, together with user-provided relevance scores for some results in the list, is used to estimate the potential relevance scores for the results that have no user feedback yet, and re-order them so that more relevant results can potentially appear nearer to the top of the list and the user can find what he or she wants more quickly.

There can be many variants in the active code search paradigm, as there are many different feedback mechanisms for web search engines in the information retrieval research community [48, 64, 70, 89]. The particular active code search technique proposed in this chapter refines a list of search results from any code search engine that takes textual descriptions as queries without additional queries to the underly-

ing search engine. It can thus improve the relevance of the list of search results, although it does not retrieve additional relevant results from the whole document (i.e., code) collection. Some information retrieval techniques, e.g., [48, 89], retrieve additional results from the entire collection, however, these techniques require many repeated search operations to be performed on the entire collection and/or are specific to the internals of particular search engines. Our technique works on the result list and thus it does not require many repeated search operations and can potentially work with many underlying search engines while requiring no change to be made to the internals of the engines. A number of web search engines, e.g., [64], take user click-through logs and/or browsing preferences and histories into consideration to improve search results for *future* queries. In this work, we utilize the feedback collected from users to improve the search results for both *current* queries and *future* queries.

7.2.3 Normalized Discounted Cumulative Gain

To compare different code search techniques, we need to measure the relevance of a list of search results for a user query, where the relevance score for each result in the list is given by users who form opinions about whether the result matches the query.

By following the user study performed to evaluate Portfolio [77], the relevance score for an individual search result in our study is given by human participants in a 4-point Likert scale [61], ranging from 1 (completely irrelevant) to 4 (highly relevant), and the meaning of each score is as follows:

1: Completely irrelevant — there is very little correlation between the result and the given query.

2: Mostly irrelevant — the result is only remotely relevant to the given query; it is unclear how to reuse the returned code for the task intended by the query.

3: Mostly relevant — the result has some relevance to the given query; the returned code may be modified with modest effort for the task intended by the query.

4: Highly relevant — the result is highly relevant to the given query with high confidence.

However, different from Portfolio, we employ the Normalized Discounted Cumulative Gain (NDCG) [40] instead of Precision to measure the overall relevance of a list of results. McMillan et al. define precision as $\frac{\text{\# of relevant results in a list}}{\text{Total \# of results in the list}}$, where "relevant results" are the ones with Likert scores 3 or 4 [77]. This metric does not consider the *ordering* of the results in the list and may be suitable for a relatively short list where all results in the list can be expected to be inspected by a user.

In our setting, active code search re-orders search results from a passive code search engine, and its effectiveness has to be measured when the ordering of results in a list is considered and . Thus, we choose to use NDCG, which is a metric commonly used to evaluate the results of web search engines to measure the relevance of a list of ranked results.

The ordering of the results matters in measuring the effectiveness of a code search task. A measure should give higher scores if relevant results appear nearer to the top of the list. NDCG, which is a metric commonly used to evaluate the results of web search engines and measure the relevance of a list of ranked results [33, 40, 70], is such a measure. NDCG has two assumptions: First, relevant documents that appear earlier in the result list are more valuable than those appearing later; Second, highly relevant documents are more valuable than marginally relevant documents. According to these assumption, the NDCG score of a list of ranked results can be calculated as follows:

$$DCG = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{log_2(i+1)}$$

$$NDCG = \frac{DCG}{IDCG}$$

In the equations above, rel_i is the relevance score given by human participants for the result at the i^{th} position in the list; p is the size of the list; ***IDCG*** is the DCG score of a perfectly sorted list. Based on the definition, $NDCG$ ranges from 0 to 1. When $NDCG$ is equal to 1, the ranking algorithm performs perfectly.

Example. We give an example to show how $NDCG$ is calculated. Consider a list of documents ordered by a ranking algorithm: D_1, D_2, D_3, D_4. Let their relevance scores be $1, 3, 2, 4$ respectively. $DCG = \frac{2^1-1}{log_2 2} + \frac{2^3-1}{log_2 3} + \frac{2^2-1}{log_2 4} + \frac{2^4-1}{log_2 5} = 13.4$. The perfect ordering is D_4, D_2, D_3, D_1 with the documents sorted in the descending order of their relevance scores, so, the $IDCG$ is 21.4. Finally, the $NDCG$ for this list of results is 0.626.

7.3 Approach Overview

This section presents the overall framework of our active code search approach. Section 7.4 then elaborates the details about the core component of our approach—the refinement engine that reorders the search results from a normal code search engine based on relevance feedback from users.

Our active code search framework is shown in Figure 7.1. It consists of two major processing components (shown as rectangular blocks in the figure): Code Search Engine and Refinement Engine. The interaction between users and these two major components results in a list of results (in the rounded rectangular block) containing potentially relevant code fragments. As a user can provide feedback piece by piece incrementally, the list of search results is refined in multiple iterations, as illustrated by the circular arrows. The sequence of the interactions in our active code search framework is described in the following paragraphs.

First, a user posts a query to a passive code search engine. The search engine takes the user query and returns a list of code fragments (i.e., the "List of Results" block in Figure 7.1) sorted according to the scores of the code fragments calculated internally by the search engine. The earlier a code fragment appears in the list, the

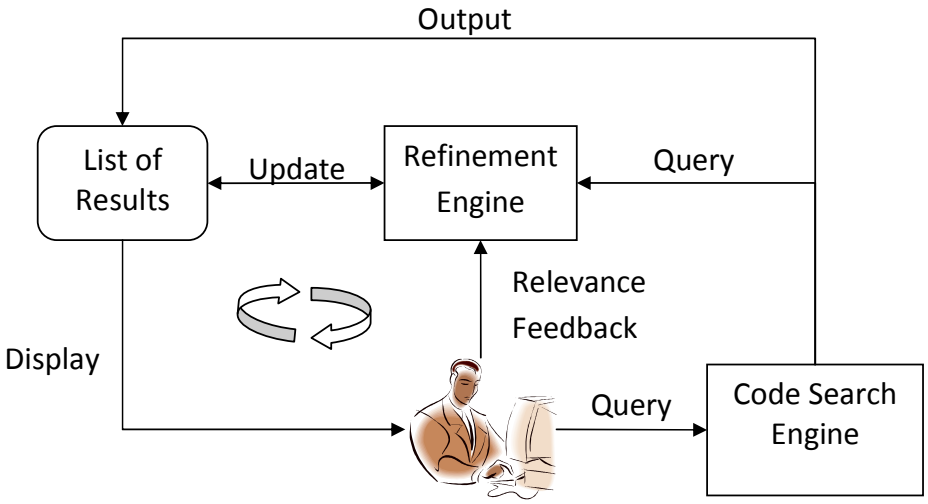

Figure 7.1: Active Code Search: Overall Structure

more similar the search engine thinks it is to the user query. In this chapter, we use Portfolio [77] in the Code Search Engine block.

Second, as the user navigates through the list of results one by one, he or she can provide relevance feedback. For example, after the user investigates the result at the top of the list, the user forms a judgement if the result is relevant. The judgement can be expressed as a label for the result in a 4-point Likert scale [61]. Each judgement for the results investigated by the user is used as input to the Refinement Engine block one at a time.

Third, as the refinement engine receives a piece of relevance feedback, it refines the ordering of the results in the list that have not been investigated by the user, aiming to improve the overall relevance of the refined list. For example, an uninvestigated code fragment that is originally at the 5th position could be shifted to the 48th position, as it is similar to some investigated code fragments that are given low relevance scores; on the contrary, an uninvestigated code fragment at the 48th position could be shifted to the 4th position, as it is similar to some investigated code fragments receiving high relevance scores. The process that involves refining a list of results, displaying a refined list, and providing relevance feedback repeats until the user decides to stop or when the list is exhausted. When a user stops, it

could man that the user has found what he or she wants, or that the user has decided to accept something marginally relevant and use the returned code from there, or where the user has decided to give up and to complete his or her task without resulting code, etc. In the refinement engine, the feedback of previous queries and their corresponding refined results are cached. When a new query is posted, if a similar query is identified in the cache, the refined results of the similar query are returned. We describe how we identify the relevant refined results in Section 7.4.

7.4 Refinement Engine

This section describes the core component of our approach in detail. Figure 7.2 illustrates the structure of our refinement engine. The engine takes in the original user query, a list of results, the relevance feedback from users expressed in 4-point Likert scores, and outputs a refined list of results that shows potentially more relevant results nearer to the top.

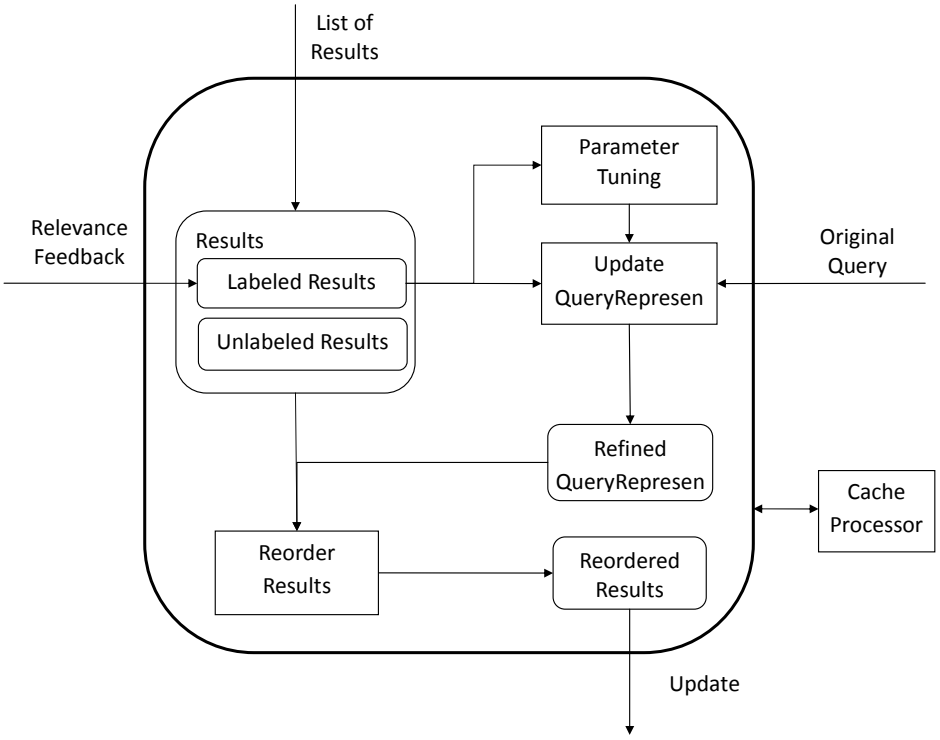

Figure 7.2: Refinement Engine Component

Our refinement engine has several data blocks (in rounded rectangles): "Results", "Labeled Results", "Unlabeled Results", "Refined Query Representation", and "Reordered Results". Results are the list of results displayed to the user so far. Labeled results are results that have received relevance feedback from the user. Unlabeled results are those that have not received relevance feedback from the user. The Refined Query Representation is used internally in our refinement engine to represent the combined effect of the original query and the results that have been investigated and labeled by the user. Reordered results are the refined list outputted by the refinement engine after taking user feedback into consideration and will be presented to the user for additional feedback.

Our refinement engine has also several processing blocks (in rectangles): "Update Query Representation", "Parameter Tuning", "Reorder Results", and "Cache Processor". These processing blocks work together to refine the list of results for the original user query based on the relevance feedback. The "Update Query Representation" block produces the refined query representation from labeled results and the original query. This refined query representation is used by the "Reorder Results" block to produce the refined results. The "Update Query Representation" block accepts parameters that are to be tuned periodically as more relevance feedback is received. This is done by the "Parameter Tuning" block. The "Cache Processor" block stores the original queries that the refinement engine has processed before and their corresponding labeled results, refined query representations, and reordered results. This block also checks if a new input query closely matches a past query; if it does, it will bootstrap the refinement engine with the cached data. If the same query is processed by the refinement engine, the cached data will be updated. We elaborate on these processing blocks in Sections 7.4.1, 7.4.2, 7.4.3, and 7.4.4.

7.4.1 Update Query Representation

In this block, we incorporate information from search results that have received relevance feedback (i.e., Labeled Results) into the original query. We convert the original query into its representative vectors. We also convert the search results that have received relevance feedback into their representative vectors. We then update the representative vectors of the original query to a refined query representation. To elaborate "Update Query Representation", we present some definitions first, and then the algorithm.

Query and Result Representations

We first introduce the representations for a query and a result that can be used to transform a textual query or a code fragment into vectors of numerical scores. We consider two representations: *semantic* and *structural*.

Definition 3 (Semantic Representation)**.** *In the semantic representation, a query and a code fragment are viewed as a bag of words. We use standard tokenization, stop word removal, identifier splitting, and stemming to convert a query or a code fragment into a bag of words [70]. The semantic score of a word is given by the product of its* term frequency *and* inverse document frequency *(tf*idf) [70]. The term frequency (tf) of a word is the number of times the word appears in the query or code fragment normalized by the total number of words in the query or code fragment. The inverse document frequency (idf) of a word is the logarithm of the total number of documents (i.e, the number of methods in the code base in our code search setting) divided by the number of documents that contain the word. We take the logarithm, since otherwise the idf would be very large. Given a query or code fragment* q*, we denote the vector of semantic scores representation of* q *as* $VScore(q)_{sem}$.

Example. Let a query q be *lock unlock file*. Then the term frequencies of words *lock*, *unlock* and *file* are 0.33, 0.33, 0.33 respectively. Also, let the words *lock*,

unlock and *file* appear in 270,872, 154,029, and 800,672 methods out of 7,916,458 methods in the code base. Then the inverse document frequencies for the words calculated by the equation $idf(w, D) = log\frac{|D|}{|\{d \in D : w \in d\}|}$ are 1.46, 1.71, and 0.99, respectively, where w is a word in the query and d is a method in the code base D. Finally, the semantic scores are 0.48, 0.54, and 0.33 for *lock*, *unlock* and *file*, respectively.

Definition 4 (Structural Representation)**.** *In the structural representation, a query and a code fragment are viewed as a bag of function calls. The structural score of a function is given by the product of its term frequency and inverse document frequency. Given a query or a code fragment r in the search results, the term frequency of a function in r is the frequency of the function being called in r normalized by the total number of function calls in r; the inverse document frequency of a function is the logarithm of the total number of code fragments in the search results divided by the number of code fragments in the search results that call the function. We denote the vector of structural scores of r as $VScore(r)_{str}$. Note that the original query entered by the user does not contain function calls, thus it is represented by a vector of zeroes. However, as we incorporate results that have received relevance feedback from users to the original query, the query's vector of structural scores is updated.*

In this definition, we use the names of the functions called in the search results to construct the structural scores because we assume that code fragments performing similar functionality would call the same functions and the similarity among code fragments may be measured by the functions called in the code fragments.

Example. Consider a code fragment r which is a function *ScLockedFile*. There is only one function *unlock* called in *ScLockedFile*. So the term frequency of *unlock* is 1. In total 50 search results are returned, and the function *unlock* is called by 2 code fragments in the results; so its inverse document frequency value is $log\frac{50}{2}$=1.40. Thus, the $VScore(unlock)_{str}$ for *unlock* is 1.40.

Vector Operations

We also define operations that are applied to vectors and help to make it easier to describe the algorithm in the following subsections of this chapter.

Let $v[i]$ be the score in a vector v corresponding to a word or a function i. If the word or the function i does not have a corresponding entry in v, let $v[i]$ be 0. Let us also denote $WS(v)$ to be the set of words and functions that have corresponding entries in vector v.

Definition 5 (Vector Summation and Division)**.** *– Given two vectors* $v1$ *and* $v2$*, the* sum *of these two vectors is a new vector* vr*, and* $\forall i : vr[i] = v1[i] + v2[i]$*. Consider a vector* v *and a constant* c*, the result of dividing of vector* v *by* c *is a new vector* vr*, and* $\forall i : vr[i] = v[i]/c$*.*

Example. Suppose $v1_{sem} = \{file = 1, unlock = 2, lock = 2\}$ and $v2_{sem} = \{file = 1, access = 3, control = 2\}$. The sum of these two vectors $v1_{sem} + v2_{sem}$ is $\{file = 2, unlock = 2, lock = 2, access = 3, control = 2\}$. The result of dividing the vector $v1_{sem}$ by 2 is $\{file = 0.5, unlock = 1, lock = 1\}$.

We also need to define similarity among vectors, so that we can gauge the refinement of queries and search results. We measure vector similarity by adapting the well-known cosine similarity [99].

Definition 6 (Vector Cosine Similarity)**.** *Given two vectors of scores,* $v1$ *and* $v2$*, the cosine similarity of* $v1$ *and* $v2$*, denoted as* $cos(v1, v2)$*, is the sum of the products of corresponding entries in the two vectors normalized by the sizes of the vectors:*

$$cos(v1, v2) = \frac{\sum\limits_{i \in (v1 \bigcup v2)} v1[i] \times v2[i]}{\sqrt{\sum\limits_{i \in v1} v1[i]^2} \times \sqrt{\sum\limits_{i \in v2} v2[i]^2}}$$

Example. Suppose $VScore(q)_{sem}$ and $VScore(r)_{sem}$ are $\{file = 0.01, unlock = 0.03, lock = 0.05\}$ and $\{file = 0.04\}$ respectively. The cosine similarity for them is 0.169. Suppose $VScore(q)_{str}$ and $VScore(r)_{str}$

are $\{file\_unlock() = 0.03, file\_lock() = 0.05\}$ and $\{file\_access() = 0.04\}$ respectively, then the cosine similarity is 0. Finally, the similarity between the query and the result is 0.085.

Algorithm

The procedure for "Update Query Representation" is shown in Algorithm 5. The procedure takes in a set of search results labeled so far (LBL), the new feedback ($fback$), the set of unlabeled results ($ULBL$), the original user query $origquery$, and a set of weights ($\alpha_1, \alpha_2, \alpha_3, \alpha_4$) that determine the contributions of labeled results with the Likert scores 1, 2, 3, and 4 respectively. Weight $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ are set to be -0.3, -0.1, 0.1, and 0.5 initially. The procedure updates the set of labeled results LBL and unlabeled results $ULBL$, and creates a refined query representation $refquery$ which consists of two vectors $refquery_{sem}$ and $refquery_{str}$ which are the semantic and structural representations.

Algorithm 5 Update Query Algorithm

1: **Procedure UpdateQuery**
2: **Input:**
3: LBL: Labeled search results
4: $fback$: New feedback, i.e., a new Likert score to an unlabeled search result
5: $ULBL$: Unlabeled search results
6: $origquery$: Original user query
7: $\alpha_1, \alpha_2, \alpha_3, \alpha_4$: The weights of contributions of labeled results with the Likert score 1, 2, 3, and 4 respectively
8: **Output:** Updated LBL, $ULBL$, and a refined query representation $refquery$
9: **Method:**
10: Add $fback$ into LBL, and remove the result labeled by $fback$ from $ULBL$
11: Let LBL^1, LBL^2, LBL^3, and LBL^4 be the sets of results in LBL with Likert scores 1, 2, 3, and 4 respectively
12: Compute the semantic centers of LBL^1, LBL^2, LBL^3, and LBL^4 and denote them as $C1_{sem}$, $C2_{sem}$, $C3_{sem}$, and $C4_{sem}$
13: Compute the structural centers of LBL^1, LBL^2, LBL^3, and LBL^4 and denote them as $C1_{str}$, $C2_{str}$, $C3_{str}$, and $C4_{str}$
14: Let $refquery_{sem} = VScore_{sem}(origquery) + \sum_{i=1...4} \alpha_i \times Ci_{sem}$
15: Let $refquery_{str} = VScore_{str}(origquery) + \sum_{i=1...4} \alpha_i \times Ci_{str}$
16: Return LBL, $ULBL$, and $(refquery_{sem}, refquery_{str})$

To transform the inputs to the outputs, the procedure works in the following steps. We split the results in LBL into four sets based on the relevance scores (Line 11). After this step, we compute the semantic *center* of each set (Line 12) by the following equation:

$$Ci_{sem} = \frac{\sum_{r \in LBL_i} VScore_{sem}(r)}{|LBL_i|}$$

Similarly, we compute the structural *center* of each set (Line 13) by the following equation:

$$Ci_{str} = \frac{\sum_{r \in LBL_i} VScore_{str}(r)}{|LBL_i|}$$

The (semantic or structural) center of each set is a vector that is the sum of the (semantic or structural) vectors of all results appearing in the set normalized by the size of the set (i.e., the number of results in the set). We then compute the semantic refined query representation ($refquery_{sem}$) by combining the centers of the corresponding four sets (Ci_{sem}) with the semantic score vector of the original query ($origquery_{sem}$) (Line 14). The structural refined query representation ($refquery_{str}$) is computed in a similar way (Line 15). This refined query representation ($refquery_{sem}$,$refquery_{str}$) is then used to help reorder the search results as described in Section 7.4.2.

7.4.2 Reorder Results Block

In this block, we sort the unlabeled results based on their similarity with the refined query representation. The pseudocode is shown in Algorithm 6. It takes in the refined query representation $refquery$ generated by Algorithm 5 and a list of unlabeled search results $ULBL$. It outputs a reordered $ULBL$ by the following steps. First, it iterates through the list of unlabeled results to compute the *structural* and *semantic similarity scores* between the score vectors of each result and those of $refquery$, i.e., $refquery_{sem}$ and $refquery_{str}$ (Lines 8-9). It then takes the average of these two scores to compute the *overall similarity score* (Line 10). Second, it reorders the unlabeled results according to their overall similarity scores in the descending order (Line 12).

Algorithm 6 Reorder Result Algorithm

1: **Procedure ReorderResults**
2: **Input:**
3: $refquery$: Refined query representation
4: $ULBL$: Unlabeled results
5: **Output:** Reordered $ULBL$
6: **Method:**
7: **for all** r in $ULBL$ **do**
8: Let $sim_{sem} = cos(VScore(r)_{sem}, refquery_{sem})$
9: Let $sim_{str} = cos(VScore(r)_{str}, refquery_{str})$
10: Let $sim_{overall} = \frac{sim_{sem}+sim_{str}}{2}$
11: **end for**
12: Sort all results in $ULBL$ in the descending order according to their overall similarities to $refquery$ as computed at Line 10
13: Return $ULBL$

7.4.3 Parameter Tuning

Our refinement engine takes in 4 weight parameters: α_1, α_2, α_3 and α_4. These 4 parameters are initially set to take the following values: -0.3, -0.1, 0.1, and 0.5, respectively[3]. We re-tune these weight parameters whenever we receive a new relevance feedback that rates a result with Likert score 3 or 4 (i.e., the result is marginally or highly relevant). We tune the algorithm by trying many possible parameter settings one at a time. For each parameter setting, we consider its effectiveness on the set of labeled data known so far. We pick the parameter setting that is the most effective (i.e., it achieves the highest NDCG on the set of labeled data known so far).

The pseudocode of our tuning process is shown in Algorithm 7. Let us define a notation $AP(a, b, s)$ to represent an arithmetic progression (AP) between a and b with a step s. We initialize two ranges α_{1-2} and α_{3-4} to be $AP(-0.5, 0.4, 0.1)$ and $AP(0, 0.9, 0.1)$ respectively (Line 6). The first is the range of possible parameter values for α_1 and α_2. The second is the range of possible parameter values for α_3 and α_4. Then we try to adjust the 4 parameters by trying different combinations of values picked from the sets α_{1-2} and α_{3-4} (Line 8).

After a combination of parameters are picked, we evaluate its effectiveness on the set LBL of labeled results known so far (Line 9). We simply reorder LBL using the parameter combination setting and compute NDCG. The larger the resulting NDCG score is, we assume the better a parameter combination is. We repeat the

[3]These values are determined empirically.

above two steps (i.e., Line 8 and 9) until we find a local optimum (Line 10). We detect a locally optimal setting for $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ by trying all combinations of values picked from the sets α_{1-2} and α_{3-4} in order and looking for a combination whose resulting NDCG score is 1 or no smaller than the NDCG scores of its neighboring configurations in the combinations of parameter values in the arithmetic progressions. We finally output the local optimal setting of the 4 parameters (Line 11).

Algorithm 7 Parameter Tuning Procedure

1: **Procedure ParameterTuning**
2: **Input:**
3: LBL: Labeled results
4: **Output:** Parameters $\alpha_1, \alpha_2, \alpha_3, \alpha_4$
5: **Method:**
6: Initialize two sets α_{1-2} and α_{3-4} to be $AP(-0.5, 0.4, 0.1)$ and $AP(0, 0.9, 0.1)$ repectively
7: **repeat**
8: Adjust the value of the 4 parameters
9: Evaluate effectiveness of the adjusted parameters on LBL
10: **until** a local optimum is reached
11: Output the local optimum

7.4.4 Cache Processor

Once a user posts a new query, the "Cache Processor" block checks whether there exists a highly similar query in the cache. If a highly similar query is identified, its corresponding cached results will be used to bootstrap the refinement engine. To identify whether a highly similar query exists, this block computes the similarity of the new query with each of the old queries. The similarity of a new query q_{new} and an old query q_{old} is computed by taking the cosine similarity of their corresponding semantic vectors:

$$Similarity_{sem} = cos(VScore(q_{new})_{sem}, VScore(q_{old})_{sem})$$

We rank the old queries based on their similarity scores. The top one old query whose similarity is larger than a threshold t is identified as the highly similar query (ties are randomly broken). If no old query has similarity above t, then no highly similar query is identified. In this study, by default, we set t to 1 which means only the exactly same query will be identified.

7.5 Empirical Evaluation

In this section, we present our experimental settings, evaluation metrics, results, and some threats to validity.

7.5.1 Experimental Settings

The code base we use in this chapter is from FreeBSD[4], which has undergone over thirty years of continuous development, improvement, and optimization. We download 49,889 program versions from this code base, but use only the latest version for programs that have multiple versions. Finally, we use 19,414 programs written in C and/or C++ in our study. The total size of these programs is around 36GB, and they contain about 169 million lines of code, 8 million functions, and 2 million files. Most code search workloads (except result collection and display) are performed on a machine with an Intel Core i5 3.2GHz CPU with 4GiB of memory and 2TiB of hard disks.

We have used 70 queries used in Portfolio's evaluation [77]. All these queries are formulated as set of keywords to address some programming tasks reported in Portfolio's user study. We show the queries in table 7.1.

We also involve users for evaluation and use a simple web application written in PHP to display search results and collect user feedback. In the user study, we have 10 participants, 9 of them are PhD students who have at least of two years of Java and C++ programming experience, and the other is a professional software engineer who has three years of Java and C++ programming experience. Each participant is assigned a number of queries and asked to examine the top fifty search results for each query and provide a relevance score in a 4-point Likert scale for every result. The guidelines for the participants to assign the 4-point relevant scores are as described in Section 7.2.3.

[4] `ftp://ftp.freebsd.org/pub/FreeBSD/distfiles/`

Table 7.1: List of queries

Id	Content	Id	Content
1	cd image as virtual disk	2	ms dos support emulation on mscdex
3	mount cd image	4	module for controlled file access
5	files lock unlock	6	access file functions
7	sort photos by date	8	lock unlock access by filename
9	lock file filename	10	mount iso file as virtual disk
11	read iso file	12	iso file virtual disk
13	mscdex emulator	14	photo organized date
15	convert raw image to jpeg	16	file lock
17	convert raw images from digital camera	18	dos read from mscdex
19	sort by date	20	mscdex
21	mscdex emulation	22	file access controller
23	image mounting	24	photo sort date
25	convert digital camera images to jpeg	26	raw file to jpeg convert
27	list image files by date	28	sort photos date
29	sort picture by date	30	dos support for emulation of mscdex
31	mount iso as virtual disk	32	convert camera image to jpeg
33	filename lock controlled access	34	img jpeg
35	files lock to prevent access	36	file access lock
37	convert from jpeg to raw images	38	file access locks
39	organize photos by date	40	dos mscdex emulator
41	organize image in digital camera based on date	42	lock unlock file
43	mscdex emulation read	44	sort photos
45	photo organized	46	mount iso virtual disk for program
47	sort modified time	48	mscdex emulate on dos
49	sort photo by date	50	convert raw images to jpeg
51	iso virtual disk mount	52	photos date time sort
53	photo date organization	54	convert raw digital image jpeg
55	raw image to jpeg convert	56	convert jpeg digital image
57	emulate mscdex on using dos	58	jpeg image format
59	sort file	60	convert from raw images to jpeg
61	lock files by filename	62	raw to jpeg
63	camera raw image convert jpeg	64	digital camera to jpeg
65	convert image to jpeg	66	convert digital camera raw image to jpeg
67	mount iso virtual disk	68	dos mscdex emulation
69	jpeg image compression	70	raw image convert jpeg

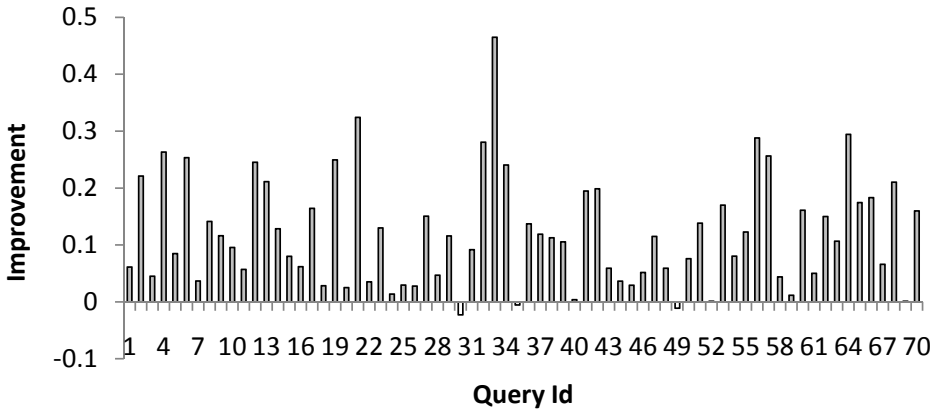

Figure 7.3: Improvement of Portfolio^{active} over Portfolio^{original} in terms of NDCG

7.5.2 Evaluation Results

The objective for our evaluation is to investigate the overall effectiveness of our active code search approach as compared to passive code search. We also want to investigate the benefit of distinctive features of our approach and its practicality. Our objective can be broken down to the following research questions (RQs):

RQ1 Can our active code search approach Portfolio^{active} improve the relevance of search results as compared to the original Portfolio?

We build our approach on top of Portfolio and thus we need to compare the two approaches. The answer to this research question will shed light on whether our active code search approach is useful to improve the relevance of search results. To answer this research question, we compare Portfolio^{active} and Portfolio on 70 queries.

RQ2 Can Portfolio^{active} outperform Portfolio^{rocchio} ?

Rocchio is a standard way to incorporate relevance feedback by refining a query based on a set of labeled results [70]. It has been used in a number of prior software engineering studies [22, 31]. Different from our proposed approach, standard Rocchio does not consider structural scores and use a static set of weights to incorporate labeled results to refine a query. In our experiments, we set the weights of Rocchio

to their recommended values [70]: $a = 1$, $b = 0.75$, and $c = 0$, where a, b, and c are the weights of the original query, results labeled as relevant, and results labeled as irrelevant, respectively. Answer to this research question will shed light on whether our approach is better than a standard relevance feedback technique.

RQ3 Are structural representation and scores useful for active code search?

One distinctive feature of our approach versus existing relevance feedback algorithms is the usage of structural representation and the computation of structural scores (see Section 7.4.1). These are unique to code search as they are derived from function invocations in source code files. The answer to this research question will shed light on whether our distinctive feature is useful for code search. To answer this research question, we compare two versions of Portfolio^{active}: one with structural representation and scores, and another without both of them.

RQ4 Is parameter tuning effective in improving result relevance?

The parameter tuning step is potentially at runtime(see Section 7.4.3). The parameter tuning step is also another additional feature that differentiates us from Rocchio [70]. In Rochio, the various weights are set to their default values and are not dynamically learned. The answer to this research question will shed light on whether this feature is useful for code search. To answer this research question, we compare two versions of Portfolio^{active}: one with parameter tuning, and another without parameter tuning.

RQ5 How efficient, in terms of running time, is Portfolio^{active}?

The efficiency of Portfolio^{active} will affect its practical usage. Portfolio^{active} needs to process each relevance feedback efficiently. Otherwise, a user needs to wait a long time for the list of results to be updated. In this study, our goal is for Portfolio^{active} to process the relevance feedback in less than one second. To answer this question, we compute the average time that Portfolio^{active} needs to spend to process the relevance feedback.

RQ1: Overall Relevance Improvement

We compare the NDCG of Portfolio^{active} with that of Portfolio^{original}. Our evaluation shows that the original Portfolio on average achieves an NDCG score of 0.738 while Portfolio^{active} on average achieves an NDCG score of 0.821, a 11.3% improvement. Figure 7.3 presents a detailed comparison of the NDCG score for every query. Table 7.2 shows a summary of the comparison results: Portfolio^{active} wins in 67 cases while performing marginally worse for 3 other cases (query number 35, 49, and 52). We also perform a Wilcoxon signed-rank test [115] on the NDCGs (p-value < 0.05), which indicates the improvement achieved by Portfolio^{active} is statistically significant.

For illustration, Table 7.3 shows the top three cases where Portfolio^{active} improves the NDCG of the search result the most. For example, for the query "mscdex emulation", Portfolio^{active} reorders two highly relevant results from positions 14 and 15 to positions 2 and 3 just after one round of feedback, and improves the NDCG from 0.557 to 0.738.

In few cases, we could notice that the results from Portfolio^{active} are worse than those from Porfolio^{original}. We investigate these cases and find that some irrelevant functions will be brought to higher rank by the functions marked as relevant because of their similarity in text. For example, in query 49 (i.e., sort photo by date), functions *slotSortDateDec()* and *slotSortDateInc()* are marked as highly relevant as their functionality is to sort photos based on their dates in slots. However, another two irrelevant functions *slotAddPhoto()* and *slotEditCopy()* are made to rank in high positions because these four functions have high similar textual information which is all about operating with slots. One potential way to alleviate the problem is to slice out the source code irrelevant to the original query in the functions when reordering the rest results, which could reduce the effect from the irrelevant part of a function.

In the default setting, for each code search task, users give a feedback to

Table 7.2: Portfolio^{original} vs. Portfolio^{active}: Wins and Loses

Result	Number
Portfolio^{original} > Portfolio^{active}	3
Portfolio^{original} < Portfolio^{active}	67

Table 7.3: The three sample cases where Portfolio^{active} improves the NDCG of the search results the most

Query	NDCG of Portfolio^{original}	NDCG of Portfolio^{active}	Reordering samples
mscdex emulation	0.557	0.738	14→2, 15→3
filename lock controlled access	0.596	0.873	46→2, 47→3
digital camera to jpeg	0.612	0.792	30→2, 50→8

each of the first 50 results and after each feedback we apply our refinement engine. We would like to test the effectiveness of our active code search approach with different amount of feedbacks from users (denoted as K_f). In this experiment, users only give feedback on the first K_f results. We vary K_f in the set $\{$1,2,3,4,5,10,15,20,25,30,35,40,45,50$\}$ and measure the effectiveness in terms of NDCG. Table 7.4 compare the effectiveness of Portfolio^{original} with our Portfolio^{active} with different amount of feedbacks (K_f) to refine the results. When only one piece feedback is given by the user for refining the results, Portfolio^{active} achieves a 7.46% improvement over Portfolio^{original}. As the K_f value increases, Portfolio^{active} achieves more and more improvement until K_f reaches 30. The effectiveness of Portfolio^{active} remains constant when K_f is increased from 30 to 50.

RQ2: Portfolio^{active} vs Portfolio^{rocchio}

To answer this question, we compare Portfolio^{active} with Portfolio^{rocchio} in terms of NDCG. Portfolio^{rocchio} achieves an average NDCG score of 0.764. Porfolio^{active} achieves an average NDCG score of 0.821 which is a 7.5% improvement over Portfolio^{rocchio}'s result. We also perform a Wilcoxon signed-rank test and find that the improvement achieved by Portfolio^{active} is statistically significant (p-value < 0.05).

Table 7.4: Comparison of Portfolio^{original} and Portfolio^{active} in terms of NDCG for different K_f

K_f	Portfolio^{original}	Portfolio^{active}	Improvement
1	0.738	0.794	7.46%
2	0.738	0.806	9.10%
3	0.738	0.805	9.04%
4	0.738	0.808	9.47%
5	0.738	0.809	9.59%
10	0.738	0.816	10.55%
15	0.738	0.820	11.05%
20	0.738	0.820	11.05%
25	0.738	0.820	11.05%
30	0.738	0.821	11.12%
35	0.738	0.821	11.12%
40	0.738	0.821	11.12%
45	0.738	0.821	11.12%
50	0.738	0.821	11.12%

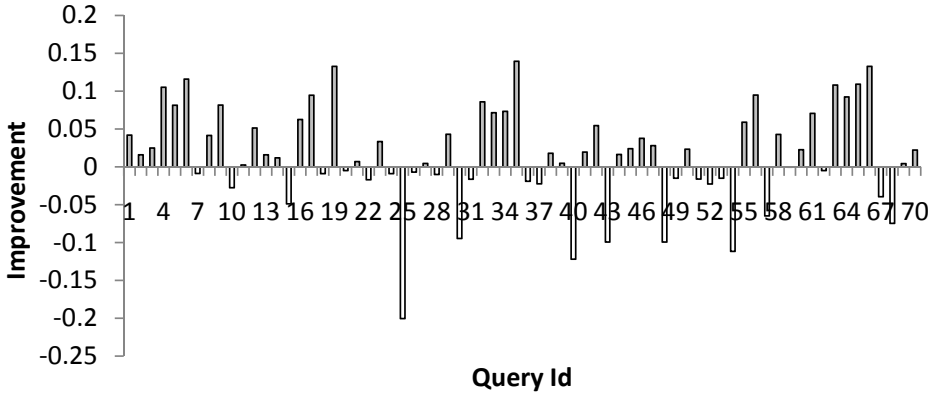

Figure 7.4: Improvement of Portfolio^{active} with structural scores over Portfolio^{active} without structural scores in terms of NDCG

RQ3: Effectiveness of Structural Scores

We compare Portfolio^{active} with structural scores and that without structural scores. Figure 7.4 shows the performance of Portfolio^{active} with structural scores against that without structural scores. Among the 70 queries, Portfolio^{active} with structural scores wins in 43 cases. The evaluation shows that adding structural scores can improve the average NDCG from 0.81 to 0.821. We also perform a Wilcoxon signed-rank test to show that the improvement achieved by Portfolio^{active} with structural scores is statistically significant (p-value < 0.05). Thus, the results show that the distinctive feature of our relevance feedback mechanism helps improve the relevance of the returned results.

RQ4: Effectiveness of Parameter Tuning

We compare the average NDCG score of Portfolio^{active} with parameter tuning against that of Portfolio^{active} without parameter tuning by using different initial configurations for the parameters α_1, α_2, α_3, and α_4. We create 625 different initial configurations as follows:

(1) Initialize two sets $\alpha_{1-2} = AP(-0.5, 0.3, 0.2)$ and $\alpha_{3,4} = AP(0, 0.8, 0.2)$.

(2) Select values of α_1 and α_2 from α_{1-2}. Select values of α_3 and α_4 from α_{3-4}. There are in total 5*5*5*5=625 configurations.

For each of these initial configurations, we run Portfolio^{active} with parameter tuning and Portfolio^{active} without parameter tuning. This results in two sets of NDCG scores. Figure 7.5 shows the results. Among all the 625 initial configurations, Portfolio^{active} with parameter tuning wins in 606 configurations and loses only in 19 configurations.

The average NDCG of Portfolio^{active} with parameter tuning is 0.819, while that of Portfolio^{active} without parameter tuning is 0.792. Figure 7.5 indicates that the performance of Portfolio^{active} with parameter tuning is more stable and better. We also perform a Wilcoxon signed-rank test on the results, which shows that Portfolio^{active} with parameter tuning statistically significantly outperforms Portfolio^{active} without parameter tuning.

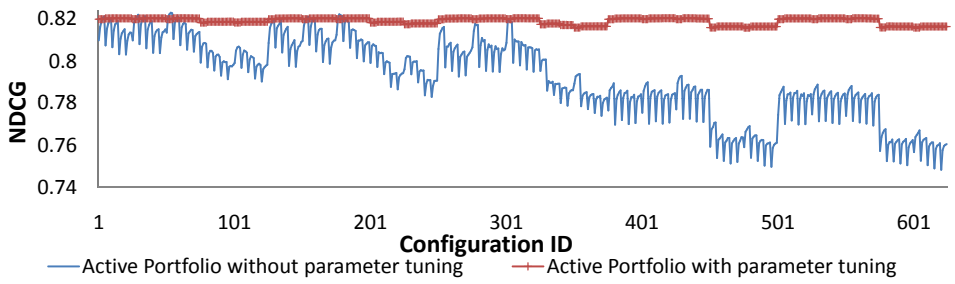

Figure 7.5: NDCGs of Portfolio^{active} with parameter tuning versus Portfolio^{active} without parameter tuning

RQ5: Efficiency of Active Code Search

We measure the average response time for every iteration of the result reordering process on the 70 queries. The total wall-clock time of the simulated active code search process for all results from all queries (50*70) is 1896 seconds. Thus, the average time for all results from each query is 27.1 seconds, and the average response time for each feedback from users is 0.54 seconds.

We believe that this response time is acceptable and it has satisfied our initial goal, i.e., it takes less than one second to process a round of feedback. The most time consuming part of our approach is the parameter tuning step. It can be a valuable future work to optimize parameter tuning and various parts of our implementation to make them more efficient.

7.5.3 Threats to Validity

Threats to internal validity include experimenter bias. There might be subjectivity in the relevance scores that a participant assigns to returned code fragments. Similar to past studies, e.g., [77, 104, 107], we do not have a large pool of participants such that each result can be labeled by many people. Thus, inter-rater reliability is not dealt with in this user study. Still, we consider 70 queries and take averages, and we believe that averaging helps reduce the threats.

Threats to external validity relate to the generalizability of our findings. We have investigated 70 queries and considered a code base consisting of 19,414 projects and 169 million lines of code. Admittedly these queries do not represent all the different kinds of queries that a developer might give to a code search engine. Also, our code base is just a fraction of the collection of all code out there. In the future, we plan to reduce the threats to external validity further by investigating more queries and larger code databases.

Threats to construct validity refer to the suitability of our evaluation metrics. We use normalized discounted cumulative gain (NDCG) [40], a common metric used to

investigate the quality of web search engines, and we believe the threats to construct validity are minimal.

7.6 Conclusion and Future Work

Code search is a common activity required in many software engineering tasks, and needs tool support to help users find pieces of code of interest in a large code base. However, most code search techniques are passive, i.e., the code search engine simply outputs a static list of results given a query. Users' feedback on the relevancy of query results is not taken into consideration.

In this chapter we propose active code search, where a user can provide feedback to the code search engine and guide the engine to improve the relevance of search results. We propose a refinement engine that can take into consideration user relevance feedback: based on a set of results whose relevance feedback has been received, our engine enhances the user's original query and uses it to update the search results by reordering the potentially more relevant search results to the top of the list for the user to see. The refinement process can be repeated for a number of times until the list is exhausted or the user decides to stop searching further down the list. Our active code search technique imposes little additional overhead on users, and can improve the relevance of search results from any passive code search engine that takes textual descriptions as user queries.

We have evaluated our approach on 70 queries, and involved 10 participants to search for relevant code fragments in a corpus of 19,414 software projects. We find that our active code search approach on average improves the effectiveness of code search by 11.3% in terms of Normalized Discounted Cumulative Gain (NDCG).

In this work we only apply our approach to Portfolio. We plan to apply our approach to other passive code search tools and show that the active code search paradigm can benefit those code search engines too. There are many variants of feedback mechanisms (e.g., learning to rank) that we can investigate as well.

Chapter 8

Conclusion and Future Work

Today's software is large and complex, consisting of millions of lines of code. New developers for a software project face significant challenges in locating code related to their development or maintenance tasks, such as, implementing features, fixing bugs and adding new features. In fact, research has shown that developers typically spend more time on locating and understanding code during maintenance than modifying it [52]. Thus, we can significantly reduce the cost of software development and maintenance by using code search techniques to reduce the time of locating code of interest.

In this dissertation, we have presented a multimodal code search engine that allows users to search via multiple input forms, including free-form texts, an SQL-like domain-specific language, code examples, execution traces, and user feedback. Specifically, the contributions of this dissertation are as follows:

- We propose an approach called Amalgam, which integrates the information of similar bug report, historical information in the version control system and the structural information of source code itself to perform bug localization [105]. Our experimental results show that Amalgam outperforms the state-of-the-art technique.

- We propose a SQL-like query language not only allowing developer to specify

their search targets in a specialized graph query language, which allows users to describe low-level data and control dependencies in code, but also in a free-form format describing desired topics [106].

- To address the drawback of dependence-based code search, we propose AutoQuery, which can automatically construct dependency queries from a set of code snippets [110].

- We propose to compose existing spectrum-based fault localization measures into a more improved measure [111]. Our experimental results demonstrate that our compositional approach improves the existing approaches significantly.

- We present an active approach to incorporate users' opinions on the results from a code search engine to refine result lists: as a user forms an opinion about one result, our technique takes this opinion as feedback and leverages it to re-order the results to make truly relevant results appear earlier in the list [109].

In the future, We plan to enhance Amalgam by involving more information, such as bug reporter information, stack trace information. We also plan to extend our compositional fault localization measure by involving more individual fault localization measures and evaluate on more datasets. We would like to experiment with more code search tasks and systems in active code search and dependency-based code search. We would like to apply our active code search to other passive code search tools and show that the active code search paradigm can benefit those code search engines too. I am also interested in applying other feedback mechanism to improve active code search.

As a long term plan, one promising direction that I would like to pursue is to enrich the returned source code. I believe that searching code alone is insufficient and I plan to link the identified code to other related textual materials, such as online

documents of the APIs appearing in the resulting code, examples about these APIs, and discussions related to the resulting code. Another direction that I would like to pursue is code auto-completion based on a context. I plan to do automatic code completion, by employing data mining techniques (e.g., API sequence mining) and natural language processing techniques (e.g., statistical language model) to analyze a large code base and infer missing code based on a particular context.

Besides research on code search, I am also interested to employ program analysis and data mining techniques to improve software testing, such as developing a new test case generation method and improving web application fault localization via log analysis. I am also interested in doing research on mobile security, such as malware detection – I would like to detect the malicious behavior of an application by leveraging program dependency graph.

Bibliography

[1] R. Abreu. *Spectrum-Based Fault Localization in Embedded Software.* PhD thesis, Delft University of Technology, 2009.

[2] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 2009.

[3] R. Abreu, P. Zoeteweij, and A. J. van Gemund. Spectrum-Based Multiple Fault Localization. In *IEEE/ACM International Conference on Automated Software Engineering*, Auckland, New Zealand, 2009.

[4] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the Accuracy of Spectrum-based Fault Localization. In *TAICPART-MUTATION*, 2007.

[5] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10):970–983, Oct. 2002.

[6] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *ETX*, pages 35–39, 2005.

[7] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *ISSTA*, 2010.

[8] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Practical fault localization for dynamic web applications. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, 2010.

[9] G. K. Baah, A. Podgurski, and M. J. Harrold. Causal inference for statistical fault localization. In *ISSTA*, pages 73–84, 2010.

[10] G. K. Baah, A. Podgurski, and M. J. Harrold. The probabilistic program dependence graph and its application to fault diagnosis. *IEEE Trans. Software Eng.*, 36(4):528–545, 2010.

[11] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press/Addison-Wesley, 1999.

[12] T. Biggerstaff, B. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *ICSE*, 1993.

[13] W.-K. Chan, H. Cheng, and D. Lo. Searching connected api subgraph via text phrases. In *SIGSOFT FSE*, page 10, 2012.

[14] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast graph pattern matching. In *ICDE*, pages 913–922, 2008.

[15] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, pages 342–351, 2005.

[16] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *ASE*, pages 433–436, 2007.

[17] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[18] L. Erlikh. Leveraging legacy system dollars for e-business. *IEEE IT Pro*, pages 17–23, May/June 2000.

[19] J. Etzold, A. Brousseau, P. Grimm, and T. Steiner. Context-aware querying for multimodal search engines. In *The 18th International Conference on MultiMedia Modeling*, Klagenfurt, Austria, 2012.

[20] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ICSE*, pages 321–330, 2008.

[21] V. Ganesh, A. Kiezun, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: A string solver for testing, analysis and vulnerability detection. In *CAV*, pages 1–19, 2011.

[22] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. On the use of relevance feedback in IR-based concept location. In *ICSM*, pages 351–360, 2009.

[23] GrammaTech. *CodeSurfer*. `http://www.grammatech.com/`.

[24] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. M. Cumby. Exemplar: Executable examples archive. In *ICSE*, pages 259–262, 2010.

[25] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *ASE*, pages 263–272, 2005.

[26] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, NY, USA, 1997.

[27] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. D. Lucia, and T. Menzies. Automatic query reformulations for text retrieval in software engineering. In *ICSE*, pages 842–851, 2013.

[28] J. Han and M. Kamber. *Data Mining Concepts and Techniques*. Morgan Kaufmann, 2006.

[29] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, pages 290–299, 2007.

[30] M. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, 2000.

[31] J. Hayes, A. Dekhtyar, and S. Sundaram. Advanced candidate link generation for requirements tracing: The study of methods. In *TSE*, 2006.

[32] J. Henrard and J.-L. Hainaut. Data dependency elicitation in database reverse engineering. In *CSMR*, pages 11–19, 2001.

[33] D. Hiemstra, T. Demeester, D. Trieschnigg, and D. Nguyen. Text retrieval conference (trec) federated web search track, 2013.

[34] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *ICSE*, pages 232–242, 2009.

[35] J. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* Univ. of Michigan, 1975.

[36] R. Holmes and G. Murphy. Using structural context to recommend source code examples. In *ICSE*, 2005.

[37] S. Horwitz and T. W. Reps. The use of program dependence graphs in software engineering. In *ICSE*, pages 392–411, 1992.

[38] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *ICSE*, 1994.

[39] J. Jang, A. Agrawal, and D. Brumley. Redebug: Finding unpatched code clones in entire os distributions. In *IEEE Symposium on Security and Privacy (S&P)*, pages 48–62, 2012.

[40] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, Oct. 2002.

[41] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *ISSTA*, 2008.

[42] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105, 2007.

[43] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *ISSTA*, pages 81–92, 2009.

[44] R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang. Computing label-constraint reachability in graph databases. In *SIGMOD*, pages 123–134, 2010.

[45] J. Jones and M. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, 2005.

[46] J. Jones, M. Harrold, and J. Stasko. Visualization of test information to assist fault detection. In *ICSE*, pages 467–477, Orlando, Florida, May. 2002.

[47] E. Keenan, A. Czauderna, G. Leach, J. Cleland-Huang, Y. Shin, E. Moritz, M. Gethers, D. Poshyvanyk, J. I. Maletic, J. H. Hayes, A. Dekhtyar, D. Manukian, S. Hossein, and D. Hearn. Tracelab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. In *ICSE*, pages 1375–1378, 2012.

[48] D. Kelly and N. J. Belkin. Reading time, scrolling and interaction: Exploring implicit sources of user preferences for relevant feedback. In *SIGIR*, pages 408–409, 2001.

[49] J. Kim, S. Lee, S.-w. Hwang, and S. Kim. Towards an intelligent code search engine. In *AAAI*, 2010.

[50] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 489–498, 2007.

[51] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[52] A. J. Ko, B. A. Myers, and D. H. Chau. A linguistic analysis of how people describe software problems. In *VL/HCC*, pages 127–134, 2006.

[53] R. Komondoor and S. Horwitz. Tool demonstration: Finding duplicated code using program dependences. In *ESOP*, pages 383–386, 2001.

[54] C. Lattner, A. Lenharth, and V. S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, pages 278–289, 2007.

[55] M.-W. Lee, J.-W. Roh, S. won Hwang, and S. Kim. Instant code clone search. In *SIGSOFT FSE*, pages 167–176, 2010.

[56] C. Lewis and R. Ou. Bug prediction at google. `http://google-engtools.blogspot.sg/2011/12/bug-prediction-at-google.html`, 2011.

[57] J. Li and M. D. Ernst. Cbcd: Cloned buggy code detector. In *ICSE*, pages 310–320, 2012.

[58] M. Li, H. Zhang, R. Wu, and Z.-H. Zhou. Sample-based software defect prediction with active and semi-supervised learning. *Autom. Softw. Eng.*, 19(2):201–230, 2012.

[59] Z. Li, M. Harman, and R. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 3:225–237, 2007.

[60] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, pages 141–154, 2003.

[61] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 140:1–55, 1932.

[62] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, Apr. 2009.

[63] C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *KDD*, pages 872–881, 2006.

[64] C. Liu, F. Guo, and C. Faloutsos. Bayesian browsing model: Exact inference of document relevance from petabyte-scale data. *TKDD*, 4(4):19, 2010.

[65] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: Statistical model-based bug localization. In *ESEC/FSE*, 2005.

[66] Lucia, L. David, J. Lingxiao, and B. Aditya. Active refinement of clone anomaly reports. In *ICSE*, pages 397–407, 2012.

[67] Lucia, D. Lo, L. Jiang, and A. Budi. Comprehensive evaluation of association measures for fault localization. In *ICSM*, 2010.

[68] Lucia, F. Thung, D. Lo, and L. Jiang. Are faults localizable? In *MSR*, pages 74–77, 2012.

[69] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. Jungloid mining: Helping to navigate the API jungle. In *PLDI*, 2005.

[70] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[71] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE*, pages 125–135, 2003.

[72] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *WCRE*, pages 214–223, 2004.

[73] M. Martin, B. Livshits, and M. Lam. Finding application errors and security flaws using PQL: A program query language. In *OOPSLA*, 2005.

[74] M. C. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. In *OOPSLA*, pages 365–383, 2005.

[75] A. Maule, W. Emmerich, and D. Rosenblum. Impact analysis of database schema changes. In *ICSE*, 2008.

[76] C. McMillan, M. Grechanik, and D. Poshyvanyk. Detecting similar software applications. In *ICSE*, pages 364–374, 2012.

[77] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *ICSE*, pages 111–120, 2011.

[78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[79] S. Nijssen and J. N. Kok. A quickstart in frequent structure mining can make a difference. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 647–652, 2004.

[80] K. Pan, S. Kim, and E. J. Whitehead, Jr. Toward an understanding of bug fix patterns. *Empirical Softw. Engg.*, 14(3):286–315, June 2009.

[81] X.-H. Phan and C.-T. Nguyen. *JGibbLDA*. `http://jgibblda.sourceforge.net/`.

[82] M. Porter. An algorithm for suffix stripping. *Program*, 1980.

[83] D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *ICPC*, pages 37–48, 2007.

[84] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu. Bugcache for inspections: Hit or miss? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 322–331, New York, NY, USA, 2011. ACM.

[85] S. Rao and A. C. Kak. Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In *MSR*, pages 43–52, 2011.

[86] S. P. Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 243–253, Washington, DC, USA, 2009. IEEE Computer Society.

[87] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 141–154, 2003.

[88] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *ESEC/FSE*, 1997.

[89] J. Rocchio. *Relevance Feedback in Information Retrieval*, chapter 14.

[90] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Test case prioritization: An empirical study. In *ICSM*, 1999.

[91] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[92] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *ASE*, pages 345–355, 2013.

[93] R. A. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *ICSE*, pages 56–66, 2009.

[94] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes,and Business Practices*. Addison-Wesley Professional, 2003.

[95] R. Sindhgatta. Using an information retrieval system to retrieve source code samples. In *ICSE*, pages 905–908, 2006.

[96] S. Sinha, M. J. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *ICSE*, pages 432–441, 1999.

[97] B. Sisman and A. C. Kak. Incorporating version histories in information retrieval based bug localization. In *MSR*, pages 50–59, 2012.

[98] C. D. Sterling and R. A. Olsson. Automated bug isolation via program chipping. *Software: Practice and Experience (SP&E)*, 37(10):1061–1086, August 2007. John Wiley & Sons, Inc.

[99] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 2005.

[100] N. Tansalarak and K. Calypool. XSnippet: Mining for sample code. In *OOPSLA*, 2006.

[101] S. Thummalapenta and T. Xie. ParseWeb: A programmer assistant for reusing open source code on the web. In *ASE*, 2007.

[102] S. Thummalapenta and T. Xie. PARSEWeb: A programmer assistant for reusing open source code on the web. In *ASE*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM.

[103] F. Thung, D. Lo, and L. Jiang. Detecting similar applications with collaborative tagging. In *ICSM*, pages 600–603, 2012.

[104] Y. Tian, D. Lo, and J. L. Lawall. Automated construction of a software-specific word similarity database. In *CSMR-WCRE*, pages 44–53, 2014.

[105] S. Wang and D. Lo. Version history, similar report, and structure: putting them together for improved bug localization. In *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, pages 53–63, 2014.

[106] S. Wang, D. Lo, and L. Jiang. Code search via topic-enriched dependence graph matching. In *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011*, pages 119–123, 2011.

[107] S. Wang, D. Lo, and L. Jiang. Inferring semantically related software terms and their taxonomy by leveraging collaborative tagging. In *ICSM*, pages 604–607, 2012.

[108] S. Wang, D. Lo, and L. Jiang. Understanding widespread changes: A taxonomic study. In *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*, pages 5–14, 2013.

[109] S. Wang, D. Lo, and L. Jiang. Active code search: incorporating user feedback to improve code search relevance. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 677–682, 2014.

[110] S. Wang, D. Lo, and L. Jiang. Autoquery: automatic construction of dependency queries for code search. *Automated Software Engineering*, pages 1–33, 2014.

[111] S. Wang, D. Lo, L. Jiang, Lucia, and H. C. Lau. Search-based fault localization. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 556–559, 2011.

[112] X. Wang, D. Lo, J. Chang, L. Zhang, H. Mei, and J. Yu. Matching dependence-related queries in the system dependence graph. In *ASE*, 2010.

[113] X. Wang, L. Zhang, T. Xie, H. Mei, and J. Sun. Locating need-to-translate constant strings for software internationalization. In *ICSE*, 2009.

[114] M. Weiser. Program slicing. In *ICSE*, 1981.

[115] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[116] N. Wilde and M. C. Scully. Software reconnaissance: Mapping program features to code. In *JSME*, 1995.

[117] S. Winkler and J. von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and System Modeling*, 9(4):529–565, 2010.

[118] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.

[119] X. Yan and J. Han. Closegraph: mining closed frequent graph patterns. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '03, pages 286–295, New York, NY, USA, 2003. ACM.

[120] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE*, pages 1–10, 2002.

[121] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transaction on Software Engineering*, 28:183–200, 2002.

[122] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE*, 2006.

[123] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: Towards a static noninteractive approach to feature location. In *TOSEM*, 2006.

[124] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *ECOOP*, 2009.

[125] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *ICSE*, pages 14–24, 2012.

[126] F. Zhu, Q. Qu, D. Lo, X. Yan, J. Han, and P. S. Yu. Mining top-k large structural patterns in a massive network. *PVLDB*, 4(11):807–818, 2011.