Portfolio: Searching for Relevant Functions and Their Usages in Millions of Lines of Code

COLLIN MCMILLAN, University of Notre Dame
DENYS POSHYVANYK, The College of William and Mary
MARK GRECHANIK, University of Illinois at Chicago
QING XIE and CHEN FU, Accenture Technology Labs

Different studies show that programmers are more interested in finding definitions of functions and their uses than variables, statements, or ordinary code fragments. Therefore, developers require support in finding relevant functions and determining how these functions are used. Unfortunately, existing code search engines do not provide enough of this support to developers, thus reducing the effectiveness of code reuse. We provide this support to programmers in a code search system called *Portfolio* that retrieves and visualizes relevant functions and their usages. We have built Portfolio using a combination of models that address surfing behavior of programmers and sharing related concepts among functions. We conducted two experiments: first, an experiment with 49 C/C++ programmers to compare Portfolio to Google Code Search and Koders using a standard methodology for evaluating information-retrieval-based engines; and second, an experiment with 19 Java programmers to compare Portfolio to Koders. The results show with strong statistical significance that users find more relevant functions with higher precision with Portfolio than with Google Code Search and Koders. We also show that by using PageRank, Portfolio is able to rank returned relevant functions more efficiently.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Enhancement, restructuring, reverse engineering, and reengineering*

General Terms: Documentation, Design

Additional Key Words and Phrases: Source-code search, information retrieval, natural language processing, Pagerank, user studies

ACM Reference Format:
McMillan, C., Poshyvanyk, D., Grechanik, M., Xie, Q., and Fu, C. 2013. Portfolio: Searching for relevant functions and their usages in millions of lines of code. ACM Trans. Softw. Eng. Methodol. 22, 4, Article 37 (October 2013), 30 pages.
DOI: http://dx.doi.org/10.1145/2522920.2522930

1. INTRODUCTION

Different studies show that when searching for functions to reuse, programmers prefer to find chains of function invocations rather than arbitrary functions in source code [Hill et al. 2009; Sim et al. 2011, 1998]. More specifically, developers use different tools

This work is supported by NSF CCF-0916139, NSF CCF-0916260, NSF CCF-1016868, and Accenture. Any opinions, findings, and conclusions expressed herein are those of the authors and do not necessarily reflect those of the sponsors.
Authors' addresses: C. McMillan, The University of Notre Dame, South Bend, IN; D. Poshyvanyk (corresponding author), Department of Computer Science, The College of William and Mary, Williamsburg, VA; email: denys@cs.wm.edu; M. Grechanik, The University of Illinois at Chicago, Chicago IL; Q. Xie and C. Fu, Accenture Technology Labs.

DOI: http://dx.doi.org/10.1145/2522920.2522930

including code search engines to answer three types of questions [Fritz and Murphy 2010; Sillito et al. 2008]. First, programmers want to find relevant functions that implement high-level requirements. Second, programmers must understand how a function is used in order to use it themselves. Third, programmers must see the chain of function invocations in order to understand how concepts are implemented in these functions. It is important that source-code search engines support programmers in finding answers to these questions.

In general, understanding code and determining how to use it is a manual and laborious process that takes anywhere from 50% to 80% of programmers' time [Corbi 1989; Davison et al. 2000]. Many source-code search engines return short code fragments, but these fragments do not give enough background or context to help programmers determine how to reuse these code fragments. Programmers must then invest a significant intellectual effort (i.e., they need to overcome a high cognitive distance [Krueger 1992]) to understand how to reuse these code fragments. On the other hand, if code fragments are retrieved as chains of function invocations, it makes it easier for developers to understand how to reuse these functions.

A majority of code search engines treat code as plain text where all words have unknown semantics. However, applications contain functional abstractions that provide a basic level of code reuse, since programmers define functions once and call them from different places in the code. The idea of using functional abstractions to improve code search was proposed and implemented elsewhere [Bajracharya et al. 2010; Chatterjee et al. 2009; Grechanik et al. 2010; Stylos and Myers 2006]; however, these code search engines do not automatically analyze how functions are used in the context of other functions, despite the fact that understanding the chains of function invocations is a key question that programmers ask [Fritz and Murphy 2010; Sillito et al. 2008]. Unfortunately, existing code search engines do little to ensure that they retrieve code fragments in a broader context of relevant functions that invoke one another to accomplish certain tasks.

Our idea is that a code search engine should help programmers understand how to reuse the functions that the engine returns to programmers. Programmers browse returned source code and follow function calls and function declarations. Then, programmers attempt to understand the concepts implemented by these functions by determining the chain of function invocations [Robillard et al. 2004; Starke et al. 2009]. That is, the function invocations are a key part of a programmer's understanding of the concepts implemented by any one function.

These chains of function invocations are easier for programmers to reuse than multiple examples from different components in the code. For example, consider the query "*mip map dithering texture image graphics*", which we use as an example query throughout this article. Programmers do not want to just see examples that implement *mip map* techniques, and others that render texture, and others that manipulate graphic images. A programmer wants to accomplish the complete task of dithering *mip map* images that ac-company a texture. However, among relevant results, there are functions that implement mip mapping, functions that manipulate texture, and there are multiple functions that deal with graphic images. Typically, programmers investigate these functions to determine which of them are relevant and determine how to compose these functions to achieve the goal that is expressed with the query. A programmer wants to see code for the whole task of how to *mip map* images that accompany a texture in computer graphics. A search engine can support programmers efficiently if it incorporates in its ranking how these functions call one another, and displays this information to the user.

We designed a code search system called *Portfolio* that supports programmers in finding relevant functions that implement high-level requirements reflected in query

terms (i.e., finding initial focus points), determining how these functions are used in a way that is highly relevant to the query (i.e., building on found focus points), and visualizing dependencies of the retrieved functions. Portfolio works by combining various Natural Language Processing (NLP) and indexing techniques with PageRank and spreading activation (SAN) algorithms. With NLP and indexing techniques, initial focus points are found that match keywords from queries. With PageRank, we model the behavior of programmers as they navigate through source-code search results. Finally, with SAN we elevate highly relevant chains of function calls to the top of search results. Our specific contributions are as follows.

—We have built Portfolio on two large source-code repositories. Developers can use Portfolio to search in close to 270 million C/C++ LOC in projects from FreeBSD Ports1[1] and 440 million Java LOC from Merobase.[2]
—We conducted an experiment using standard information retrieval methodology [Manning et al. 2008] with 49 professional programmers to evaluate Portfolio and compare it with the well-known and successful source-code search engines Google Code Search and Koders over a large C/C++ repository. The results show with strong statistical significance that users find more relevant code with higher precision with Portfolio than with Google Code Search and Koders. We made the materials of this study publicly available for replication purposes.[3]
—We also conducted an experiment with 19 participants to evaluate Portfolio against Koders when using a large Java repository. We found that Portfolio outperformed Koders in terms of relevance of the source code and by ordering the relevant results higher on the list of results.
—We have built and released tools for generating a function call graph of millions of lines of code for public use.[4] These tools use approximation techniques based on regular expressions to quickly find functions and function calls. We evaluated the precision of this solution and found that our tool extracts function calls with a precision of 76%. Since building a static call graph is an undeciable problem [Landi 1992], our results shed light on the potential precision of such an approximation technique.
—We show that by using PageRank, Portfolio is able to identify the most relevant functions of the located functions and show them earlier in the results. We contribute to a growing body of research showing that PageRank can identify important functions in a call graph [Bajracharya and Lopes 2009; Inoue et al. 2003, 2005; Revelle et al. 2010; Zaidman and Demeyer 2008] by running PageRank of a call graph with tens of millions of functions and function calls, and showing with strong statistical significance that PageRank improves the ordering of the search results.
—We determined that experienced programmers report the same levels of relevant results using the source-code search engines as inexperienced programmers. This result provides evidence suggesting that the results from our study can be generalized for programmers of different experience levels.

To the best of our knowledge, we are not aware of any existing code search engines that have been evaluated using a standard information retrieval methodology against commercial code search engines over a large codebase, and been shown to outperform these engines with strong statistical significance. Portfolio is available for public use.[5]

[1]http://www.freebsd.org/ports (verified on 5/10/12).
[2]http://www.merobase.com/ (verified on 5/10/12).
[3]http://www.cs.wm.edu/semeru/portfolio/ExperimentMaterials.tar.gz (63MB, verified on 5/10/12).
[4]http://www.cs.wm.edu/semeru/portfolio/fundex_v1.0.tar.gz (verified on 5/10/12).
[5]http://www.searchportfolio.net/ (verified on 5/10/12).

2. OUR SEARCH MODEL

The search model of Portfolio uses a key abstraction in which the search space is represented as a directed graph with nodes as functions and directed edges between nodes that specify usages of these functions (i.e., a call graph). For example, if a function g is invoked in the function f, then a directed edge exists from the node that represents the function f to the node that represents the function g. Recent work has shown that programmers benefit from contextual information, such as from the call graph, in order to navigate and locate relevant source code [Holmes et al. 2006; Lawrance et al. 2007, 2010a, 2010b]. Since the main goal of Portfolio is to enable programmers to find relevant functions and their usages, we need models that effectively represent the behavior of programmers while navigating a large graph of functional dependencies. These are navigation and association models that address surfing behavior of programmers and associations of terms in functions in the search graph.

2.1. Navigation Model

When using text search engines, users navigate among pages by following links contained in these pages. Similarly, in Portfolio, programmers can navigate between functions by following edges in the directed graph of functional dependencies using Portfolio's visual interface. To model the navigation behavior of programmers, we adopt the model of the random surfer that is used in popular search engines such as Google. Following functional dependencies helps programmers to understand how to use found functions. The surfer model is called random because the surfer can "jump" to a new URL, or in case of source code, to a new function. These random jumps are called teleportations, and this navigation model is the basis for the popular ranking algorithm PageRank [Brin and Page 1998; Langville and Meyer 2006].

In the random surfer model, the content of functions and queries does not matter; navigations are guided only by edges in the graph that specify functional dependencies. Accordingly, PageRank reflects only the surfing behavior of users, and this rank is based on the popularity of a function that is determined by how many functions call it. However, the surfing model is query independent since it ignores terms that are used in search queries. Our search model uses a query-independent navigation model in order to analyze the popularity of functions, and therefore the likelihood that programmers will navigate to these functions. Recent work has pointed out that programmers navigate to code also based on relevance [Lawrance et al. 2010b], and taking into consideration query terms may improve the precision of code searching. That is, if different functions share concepts that are related to query terms and these functions are connected using functional dependencies, then these functions should be ranked higher. Therefore, we also introduce our association model, which depends on the query and is computed only during search time.

2.2. Association Model

The main idea of an association model is to establish relevance among facts whose content does not contain terms that match user queries directly. Consider again the query "*mip map dithering texture image graphics*". Among relevant results there are functions that implement *mip map* techniques, and others that render texture, and there are multiple functions that manipulate graphic images. This situation is schematically shown in Figure 1, where the function F contains the term *mip map* (e.g., as an identifier name or in the comments), the function G contains the term *dithering*, the function P contains the terms *graphics* and *image*, and the term *texture* is used in the function Q. Function F calls the function G, which in turn calls the function H, which is also called from the function Q, which is in turn called from the function P. The functions F, P, and

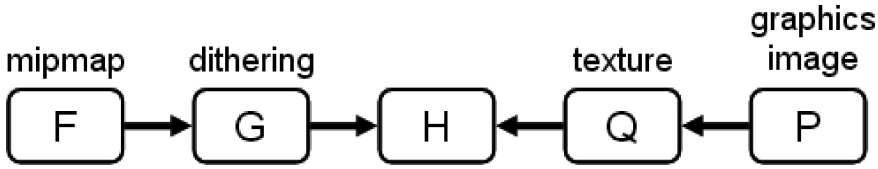


Fig. 1. Example of associations between different functions.

Q will be returned by a search engine that is based on matching query terms to those that are contained in documents. Meanwhile, the function H may be highly relevant to the query, but it is not retrieved since it has no words that match the search terms. In addition, the function G can be called from many other functions since its dithering functionality is generic; however, G is most useful in the context of other functions which implement the complete task described by the user's query. A problem is how to ensure that the functions H and G end up on the list of highly relevant functions?

To remedy this situation we employ an association model that is based on a Spreading Activation Network (SAN) [Collins and Loftus 1975; Crestani 1997]. In SANs, nodes represent documents, while edges specify properties that connect these documents. The edges' direction and weight reflect the meaning and strength of associations among documents. For example, an article about clean energy and a different article about the melting polar ice cap are connected with an edge that is labeled with the common property "climate change". Once applied to SAN, spreading activation computes new weights for nodes (i.e., ranks) that reflect implicit associations in the networks of these nodes. SANs have been used in source-code search before, and in this case edges represented word occurrences in documents [Henninger 1996]. In Portfolio, we view function call graphs as SANs where nodes represent functions, edges represent functional dependencies, and weights represent a strength of associations, which includes the number of shared terms. After the user enters a query, a list of functions is retrieved and sorted based on the score that reflects the match between query terms and terms in functions. Once Portfolio identifies the top matching functions (using word occurrences; see Section 4.1), it computes SAN to propagate the relevance from these functions to others. The result is that every function will have a new score that reflects the associations between relevant functions in these functions and user queries.

2.3. The Combined Model

We compute the PageRank and spreading activation for the functions, and store these scores as ranking vectors of functions and scores. The vector for PageRank π_{PR} and spreading activation π_{SAN} are computed separately and later linearly combined in a single ranking vector, $\pi_C = f(\pi_{PR}, \pi_{SAN})$. PageRank is query independent and is precomputed automatically for a repository function call graph once, while π_{SAN} is computed every time automatically in response to user queries. Assigning different weights in the linear combination of these rankings enables fine-tuning of Portfolio by specifying how each model contributes to the resulting ranking score.

3. PORTFOLIO ARCHITECTURE

The architecture for Portfolio is shown in Figure 2. There is a setup phase, where the different sets of data required for the search are extracted, and a search phase, where the user enters queries and receives functions relevant to these queries. Portfolio is built on top of a Projects Archive, which is a collection of software projects with source code. In this article, we instantiate Portfolio with an archive of 18,203 C/C++ projects from FreeBSD Ports and 13,701 Java projects from Merobase [Hummel et al. 2008].

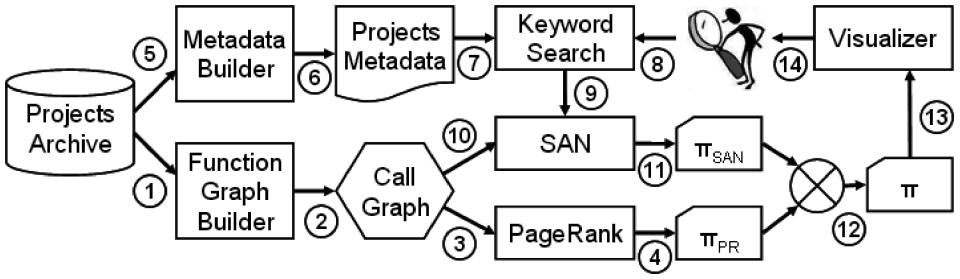


Fig. 2. Portfolio architecture.

The setup phase starts when the Function Graph Builder extracts all of the functional units and functional unit calls in the Projects Archive (1). These functional units are functions in C/C++ or methods in Java. The functional unit calls are the call relationships among these functions or methods, and the Function Graph Builder generates a Call Graph representing all of the extracted functional units as nodes and functional unit calls as edges (2). Next, we run PageRank on the Call Graph (3) and obtain a PageRank value for each functional unit (4). We precompute the PageRank values because the Call Graph is static and PageRank is not query dependent [Langville and Meyer 2006]. The setup phase includes a Metadata Builder, which is responsible for extracting the Projects Metadata (5), which is the textual data about the projects, namely the comments and identifier names present in the functional units (6).

The search phase occurs when a user enters a query into our Keyword Search component (8). The query is a natural language query, and the Keyword Search component matches keywords in this query to keywords from the Projects Metadata (7). The Keyword Search assigns a textual similarity value to the functional units, and SAN propagates this value to other functional units in the Call Graph which are connected to the functional units given by Keyword Search (9). This propagated textual similarity score is the score assigned to the functions located by SAN (11). Then, the SAN score is linearly combined with the PageRank score (12). Details on how the SAN and PageRank values are computed and combined are in Section 4. The combined values are the final scores given by Portfolio to the located functional units. These functional units are sent to the Visualizer (13), which presents to the user the functional units and the calls among these functional units (14).

4. RANKING

This section will discuss the three components that compute different scores in the Portfolio ranking mechanism: first, a component that computes a score based on word occurrences (WOS); second, a component that computes a score based on the random surfer navigation model (PageRank) described in Section 2.1; and finally, a component that computes a score based on SAN connections between these calls based on the association model described in Section 2.2. These three components rely on two sources of information: textual data from comments and identifiers, and dependencies among functions. WOS ranking is used to bootstrap SAN by providing rankings to functions based on query terms. The total ranking score is the weighted sum of the PageRank and SAN ranking scores. Each component produces results from different perspectives (i.e., word matches, navigation, and associations). Our goal is to produce a combined ranking by putting these orthogonal, yet complementary rankings together in a single score.

4.1. WOS Ranking

The purpose of WOS is to enable Portfolio to retrieve functions based on matches between words in queries and words in the source code of applications. This is a bootstrapping ranking procedure that serves as the input to the SAN algorithm.

The WOS component uses the *Vector Space Model* (VSM), which is an algebraic model for representing documents used by search engines to rank matching documents according to their relevance to a given search query. This model is implemented in the Lucene Java framework, which we use in Portfolio. Each document is modeled as a vector of terms contained in that document. In our search engine, the documents are the functions in a software repository, and the terms are the identifier names and comments in these functions. We split the terms using the camel case and underscore conventions [Enslen et al. 2009], and remove all Java reserved words as stop words. The weights of those terms in each document are calculated using the *Term Frequency/Inverse Document Frequency* (TF/IDF) formula. Using a variant of TF/IDF [Manning et al. 2008], the weight for a term is calculated as

$$tf = \frac{n}{\sum_k n_k},$$

where n is the number of occurrences of the term in the document, and $\sum_k n_k$ is the sum of the number of occurrences of the term in all documents. WOS score for a document is the sum of all query terms that are presented in that document.

4.2. PageRank

PageRank is widely described in literature, so here we give its concise mathematical explanation as it is related to Portfolio [Brin and Page 1998; Langville and Meyer 2006]. The original formula for PageRank of a function F_i, denoted $r(F_i)$, is the amortized sum of the PageRanks of all functions that invoke F_i

$$r(F_i) = \sum_{F_j \in B_{Fi}} \frac{r(F_j)}{|F_j|},$$

where B_{Fi} is the set of functions that invoke F_i and $|F_j|$ is the number of functions that the function F_j invokes. This formula is applied iteratively starting with $r_0(F_i) = 1/n$, where n is the number of functions in the project archive indexed by Portfolio (8,557,405; see Section 6). The process is repeated until PageRank converges to some stable values or it is terminated after some number of steps. Functions that are called from many other functions have a significantly higher score than those that are used infrequently or not at all. We used an existing Perl module to implement PageRank, with the default settings of that module.[6]

4.3. Spreading Activation

Spreading activation computes weights for nodes in two steps: pulses and termination checks. Initially, a set of starting nodes is selected using a number of top ranked functions (in this case, the top ten functions) using the WOS ranking. During pulses, new weights for different nodes are transitively computed from the starting nodes using the formula $N_j = \Sigma_i f(N_i \times w_{ij})$, where the weight of the node N_j is equal to the sum of all nodes N_i that are incident to the node N_j (e.g., function j calls function i) with edges whose weights are w_{ij}. This edge weight serves to give a reduced value to nodes further away from the initial nodes. Therefore, the weight is a value between 0 and 1.

[6]http://search.cpan.org/~axiak/Algorithm-PageRank-XS-0.04/lib/Algorithm/PageRank/XS.pm (accessed and verified 5/10/12).

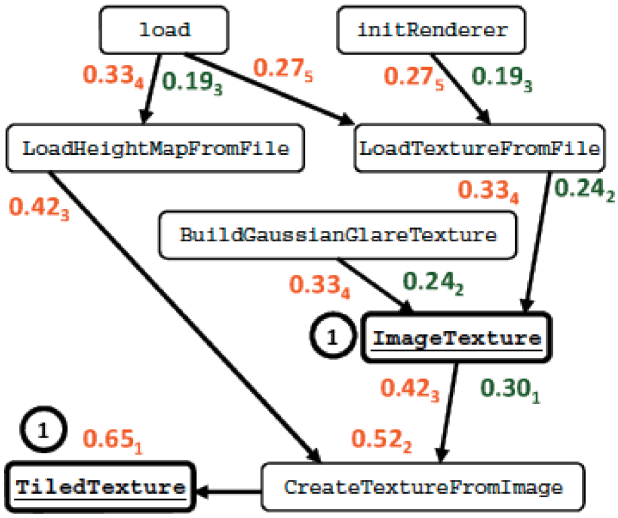


Fig. 3. Example of SAN computation, $w_{ij} = 0.8$. Weights that are computed starting with the function TiledTexture are shown in orange, and weights that are computed starting with the function ImageTexture are shown in green.

In Portfolio, we used a constant edge weight of 0.8; however, this value is configurable and could vary depending on the content of the nodes or other factors. The function f is typically called the threshold function that returns nonzero value only if the value of the argument is greater than some chosen threshold, which acts as a termination check preventing "flooding" of the SAN. We set the threshold in the function f to terminate after 8 iterations.

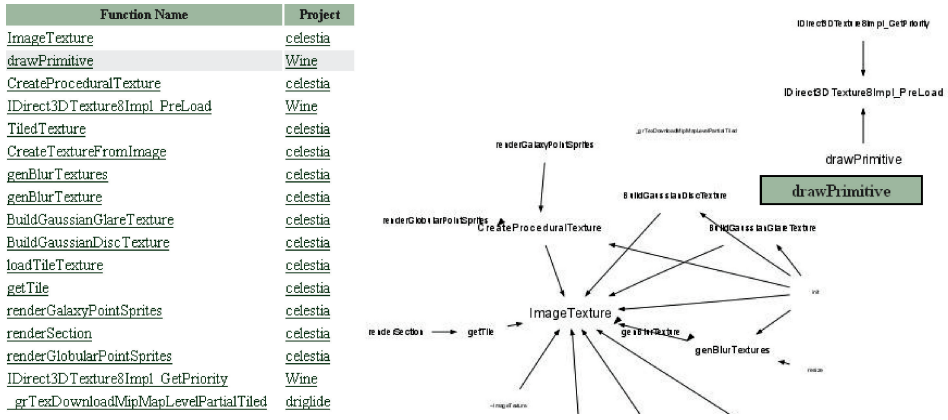
Consider an example of SAN computation that is shown in Figure 3. This example is closely related to the motivating example query "*mip map dithering texture image graphics*." The first ranking component, WOS, assigned the weights 0.65 and 0.30 to the two functions *TiledTexture* and *ImageTexture* correspondingly. We label these functions with 1. All weights are to the right (rounded off to the second digit). Their subscripts indicate the order in which weights are computed from the first function weights. For example, the weight is computed for the function *CreateTextureFromImage* by multiplying the WOS weight for the function *TiledTexture* by the SAN edge weight 0.8. Several functions (e.g., *load*, *initRendered*) get different weights by following different propagation paths from the initial function nodes. In these cases, we use the highest value for each node; the final value assigned to *initRenderer* is 0.27.

4.4. Combining Ranking

The combined rank is $S = \pi_{PR}\lambda_{PR} + \pi_{SAN}\lambda_{SAN}$, where λ is the interpolation weight for each type of the score and π is the value of the score from PageRank or SAN. We precompute the PageRank values, which are independent of queries, unlike the scores WOS and SAN, which are query dependent. Note that we only use the PageRank score to rank the results returned by WOS and SAN. Adjusting the weights (λ) enables experimentation with how underlying structural and textual information in application affects resulting ranking scores. Throughout this article, $\lambda_{PR} = 0.3$ and $\lambda_{SAN} = 0.7$.

5. PORTFOLIO INTERFACES

Portfolio is available as an online tool via a visual Web interface and a SOAP interface for integration into different projects. This section discusses these two interfaces to our search engine.



Function Name	Project
ImageTexture	celestia
drawPrimitive	Wine
CreateProceduralTexture	celestia
IDirect3DTexture8Impl_PreLoad	Wine
TiledTexture	celestia
CreateTextureFromImage	celestia
genBlurTextures	celestia
genBlurTexture	celestia
BuildGaussianGlareTexture	celestia
BuildGaussianDiscTexture	celestia
loadTileTexture	celestia
getTile	celestia
renderGalaxyPointSprites	celestia
renderSection	celestia
renderGlobularPointSprites	celestia
IDirect3DTexture8Impl_GetPriority	Wine
_grTexDownloadMipMapLevelPartialTiled	driglide

Fig. 4. The visual interface of Portfolio. The left side contains a list of ranked retrieved functions for the motivating example query and the right side contains a call graph that contains these functions; edges of this graph indicate the directions of function invocations. Hovering a cursor over a function on the list shows a label over the corresponding function on the call graph. Font sizes reflect the score; the higher the score of the function, the bigger the font size used to show it on the graph. Clicking on the label of a function loads its source code in a separate browser window.

```
#!/usr/bin/perl -w

use SOAP::Lite;

print SOAP::Lite
  -> uri('http://www.cs.wm.edu/portfolio')
  -> proxy('http://www.cs.wm.edu/~cmc/cgi-bin/portfolio/soapserver.pl')
  -> search("record midi file", "10")
  -> result;
```

Fig. 5. Example of using the SOAP interface from Perl. See the Portfolio Web site for more details.

5.1. Portfolio Visual Interface

After the user submits a search query, the Portfolio search engine presents functions relevant to the query in a browser window as shown in Figure 4. The left side contains the ranked list of retrieved functions and project names, while the right side contains a static call graph that contains these and other functions. Edges of this graph indicate the directions of function invocations. Hovering a cursor over a function on the list shows a label over the corresponding function on the call graph. Font sizes reflect the combined ranking; the higher the ranking of the function, the bigger the font size used to show it on the graph. Clicking on the label of a function shows its source code.

5.2. Portfolio SOAP Interface

Programmers can build source-code search directly into their own applications (or even IDEs) by using our SOAP Web service, as shown in Figure 5. We have made three SOAP API calls available. First, *search* takes a query as input and returns a list of results. Second, *code* gives the source code of a specified function from the list. Third, *edges* finds other functions in the repository that call the specified function. All of these SOAP functions are described at our Web site.[7]

[7]http://www.searchportfolio.net/, follow the “Programmer Access” link.

6. FUNCTION GRAPH BUILDER

Our search engine relies on the function call graph of projects. This call graph is a static call graph including all functions in the entire repository of source code. Existing static call graph generators vary in their performance and correctness [Landi 1992; Milanova et al. 2004; Murphy et al. 1998], but rely on complex procedures such as modification of the compilation process. These complex procedures are intended to improve the accuracy of the results, but as a side-effect increase the amount of time required for the analysis. Given our available resources, these existing approaches are too computationally expensive for use over our archives of tens of thousands of projects and hundreds of millions of lines of code.

We introduce our function graph builder, which approximates the call graph of these thousands of projects by matching function invocations to function definitions based on two factors: the name of the function and the location of the function definition. Our graph builder is based on the same approach used by the Linux Cross Reference.[8] First, we locate function definitions and their positions in the source code by using regular expressions. Next, we identify function invocations and extract the name of the function being called. Then, we search different visible scopes in order to match the invocation to the definition (e.g., a function defined within the same class, file, package, or imported class). We avoid computing a full parser or abstract syntax tree in order to reduce the computational expense. Our approach is an approximation of a full lexical and semantic analyzer and introduces imprecision in the resulting call graph. We evaluate the extent of this imprecision in Section 6.1.

Our function graph builder yields a call graph of the entire project archive, including calls from functions in one project to functions in a different project (inter-project calls) as well as among functions in the same project (intra-project calls). We extracted a call graph for 18,203 C/C++ projects from FreeBSD Ports containing 8,557,405 functions and 32,279,450 function calls, 5,252,474 of which are inter-project calls. For Java, we analyzed 13,701 projects from Merobase and extracted 14,499,588 methods and 10,942,126 method calls, of which 3,533,668 were inter-project calls. Note that the call graph for Java is separate from the call graph for C/C++, that is, we instantiated two versions of Portfolio engine: one for C/C++ and one for Java. Our function graph builder is available from our Web site.

6.1. Precision of the Function Graph Builder

The imprecision from the Function Graph Builder may influence the results from Portfolio. Therefore, in this section, we present an evaluation of the correctness of our call graph as extracted by the Builder. The purpose of this evaluation is to determine the level of precision under which the results from Section 7 and 8 can be generalized.

For this evaluation, we randomly selected a representative sample of 25 different C/C++ projects[9] in from FreeBSD Ports and we asked 12 graduate student developers from DePaul University to manually inspect source code of these projects to verify some of the dependencies produced by the Call Graph Builder used in Portfolio. Each participant received five projects, as well as the list of function calls (dependencies) for these projects, extracted by our Function Graph Builder. The participants then evaluated each function call to determine whether the call was correct. The participant was required to read the source code of the function making the call, and then decide

[8]http://lxr.linux.no/ (accessed and verified on 5/10/12).

[9]The 25 projects we sampled ranged from 1,075 to 103,643 LOC, with an average of 17,293 LOC. We extracted on average 3,172 dependencies from each project. The smallest project in terms of dependencies had 105 dependencies, while the largest had 17,965. The projects used in this case study may be downloaded from http://www.cs.wm.edu/semeru/files/ports_subset_01-24-10.tar.gz.

whether the call actually occurs in that function. We asked the participants to provide a brief rationale for each decision they make.

This experiment was performed online via a Web interface where participants submitted their results and the participants had three months to complete the evaluation of the projects. The Web interface randomly selected calls from the assigned projects to each participant. Given the relatively long time span of the experiment, to ensure that the participants carefully examined each function call, we inserted two incorrect function calls into each project. We then confirmed that the participants labeled these calls as incorrect. This embedded check supplied additional confidence in these results.

Since some projects contained a large number of dependencies, several participants did not evaluate every dependency in their assigned projects. In this case, we ignored dependencies that were not evaluated by at least three participants. We then combined the results given by each participant through voting, similar as it was done for building feature location benchmarks in the literature [Robillard et al. 2007]. If a majority of the participants rated the extracted function call as correct, then we count the extracted call as correct. Otherwise, we count the extracted call as incorrect.

Overall, the participants in our experiment evaluating our Function Graph Extractor checked 1,630 function calls in 25 different projects. The participants reported that 76.6% of the functions calls were correct. Moreover, the reported levels of correct calls ranged from 66.2% to 89.85% for any given project. We attribute the difference in correctly extracted calls among different projects to the fact that our Function Graph Extractor relies on matching the function calls to function names within the scope of the function call. Different projects use class inheritance and method overloading differently.

7. EVALUATION

Typically, search engines are evaluated using manual relevance judgments by experts [Manning et al. 2008]. To determine how effective Portfolio is, we conducted an experiment with 49 C/C++ programmers and a follow-up experiment with 19 Java programmers. We designed these experiments to answer the following research questions (RQ).

- RQ_1 For C/C++, does Portfolio return functions that are more relevant than the functions returned by Google Code Search and Koders?
- RQ_2 For Java, does Portfolio return functions that are more relevant than the functions returned by Koders?
- RQ_3 Do the levels of relevance reported by programmers depend on the experience level of the programmers?
- RQ_4 Does our application of PageRank cause relevant functions to be ranked higher than irrelevant functions?

The rationale behind RQ_1 is that we want to compare Portfolio to large-scale code search engines, that are used by tens of thousands of programmers on a daily basis. Therefore, we compare Portfolio to Google Code Search and Koders, which are both commercial-grade code search engines, that search large repositories of code. We dedicated one experiment to studying RQ_1. Because the participants in this experiment were C/C++ programmers, we limited the repositories that each search engine used to those languages. We studied RQ_2 in a second experiment with Java programmers. In this second experiment, our goal was to evaluate Portfolio against the best-performing competitor from the first experiment, over a different programming language.

We analyze the results of the first experiment in more depth in RQ_3. The rationale behind RQ_3 is that programmers with more experience may judge the relevance of the results differently than programmers with less experience, because more experienced

Table I. Designs of (a) the First, C/C++ and (b) Second, Java User Study

Experiment	Group	Search Engine	Task Set
1	G1	Portfolio	T1
	G2	GCS	T2
	G3	Koders	T3
2	G1	Koders	T2
	G2	Portfolio	T3
	G3	GCS	T1
3	G1	GCS	T3
	G2	Koders	T1
	G3	Portfolio	T2

(a) C/C++ user study design

Experiment	Group	Search Engine	Task Set
1	G1	Portfolio	T1
	G2	Koders	T2
2	G1	Koders	T3
	G2	Portfolio	T4

(b) Java user study design

Each group used a different search engine with different sets of tasks in a set of two or three experiments.

programmers are likely to better understand the retrieved source code. Note that we do not combine the results from the first and second experiments for RQ_3. We use only the results from the first experiment because of the potential confounding effects of mixing results from multiple user studies, different programmers, and different programming languages searched. We instead focus on results from the first experiment for three reasons. First, the first study included a much larger number of programmers than the second study. Second, a large majority of the participants from the first experiment were professional programmers from Accenture as opposed to student programmers (the experience level of the professionals was recorded by Accenture's human resources department). Finally, we evaluated three source-code search engines in the first study, compared to two in the second study, which gives our results a greater potential to be generalized.

We pose RQ_4 because we use PageRank to sort the list of functions that are located using textual similarity and spreading activation. Our goal is to determine whether our use of PageRank leads to the relevant functions (as opposed to the irrelevant functions) being located at the top of the list of search results. Recall from Section 4.5 that we only use PageRank to rank the list of results; we do not use it to add functions to the list. Without PageRank, the list will be ranked only by the textual similarity and spreading activation algorithms. While it would also be possible to study the keyword search component (a.k.a. Information Retrieval, IR) in Portfolio as a standalone technique and compare it to IR+SAN and IR+SAN+PageRank, we assume that we accomplish this task while comparing Portfolio to Google Code Search and Koders that solely rely on keyword search.

7.1. First Experiment Design

We used a cross-validation experimental design in which participants were randomly divided into three groups. The experiment was sectioned in three subexperiments (see Table Ia) in which each group was given a different search engine (i.e., Google Code

Search, Koders, or Portfolio) to locate code fragments or functions for given tasks. Each group used a different set of tasks in each subexperiment. The same task was performed by different participants on different engines in each subexperiment. Before each experiment, we gave a one-hour tutorial on using these search engines.

In the course of the experiment, participants translated tasks into a sequence of keywords that described key concepts they needed to find. Once participants obtained lists of code fragments or functions that were ranked in descending order, they examined the functions to determine if the functions matched the tasks. Each participant worked individually, assigning a confidence level, C, to the examined functions using a four-level Likert scale. We asked the participants to examine only the top ten functions that resulted from their searches since the time for each subexperiment was limited to two hours and because recent work shows that users rarely read beyond the first ten results [Granka et al. 2004].

While we aim for ten results evaluated by each participant for each query, in practice there are cases where fewer than ten results were evaluated. Low numbers of results may occur because of time pressures on the participants or because the search engine returned fewer than ten results. Moreover, the participants completed different tasks at different speeds, and participants evaluated different numbers of queries for each engine. These concerns resulted in an unbalanced dataset and form the basis for certain threats to validity (see Section 7.6).

Programmers may interpret a task and form a keyword query, but may decide to refine the query based on new information or better understanding of the task. In this case, we allowed programmers to refine their queries multiple times until they decided on a single query. The programmers then evaluated the search engine results for that final query.

The guidelines for assigning confidence levels are the following.

(1) *Completely irrelevant*. There is absolutely nothing that the participant can use from these retrieved code fragments, nothing in them is related to keywords that the participant chose based on the descriptions of the tasks.
(2) *Mostly irrelevant*. The retrieved function is only remotely relevant to a given task; it is unclear how to reuse it.
(3) *Mostly relevant*. The retrieved function is relevant to a given task and the participant can understand with some modest effort how to reuse it to solve a given task.
(4) *Highly relevant*. The participant is highly confident that the function can be reused and s/he clearly sees show to use it.

7.2. Second Experiment Design

As work subsequent to the first user study of Portfolio for C/C++, we developed a version of Portfolio capable of searching a Java repository (see Section 6). We conducted a user study to evaluate this Java version of Portfolio. The design of this study was similar to the original study, except that we compared Portfolio only to the best-performing search engine as indicated by the results from the first study. In this case, we selected Koders as the competitive approach because Koders outperformed Google Code Search. We split the study into two experiments, and rotated each engine to a different group with different sets of tasks, as shown in Table Ib. The participants then evaluated the top ten results from the assigned engine according to the Likert scale in Section 7.1.

7.3. Participants

Forty-nine C/C++ programmers participated in the first case study, while 19 Java programmers participated in the second study. Forty-four of the participants in the

Table II. Details about the Participants in the Case Studies and Code Search Engine (C.S.E.) Use

Exp.	Participant	Pro	Studs	≥ 2 years experience	Frequently use C.S.E.	Never use C.S.E.	Used G.C.S.	Used Koders
1	49	44	5	33	9	16	10	3
2	19	-	19	13	-	5	2	-

first study were professional programmers from Accenture, while five were computer science graduate students from the University of Illinois at Chicago. For the second study, we recruited 19 computer science graduate students from the College of William & Mary. Further details about these participants are in Table II. All participants have bachelor degrees and a majority have master degrees in technical disciplines.

7.4. Metrics and Statistical Analyses

This section covers the metrics and statistical analyses we used to measure the results of our first experiment. These metrics are derived from the confidence level, C, defined in Section 7.1.

7.4.1. Precision. Two main measures for evaluating the effectiveness of retrieval are precision and recall [Witten et al. 1999]. The precision is calculated as

$$P_r = \frac{\#of retrieved functions that are relevant}{total \#of retrieved functions}.$$

The precision of a ranking method is the fraction of the top r ranked documents that are relevant to the query, where $r = 10$ in this experiment. Relevant code fragments or functions are counted only if they are ranked with the confidence levels 4 or 3. Since we limit the investigation of the retrieved code fragments or functions to the top ten, the recall is not measured in this experiment (e.g., the participants evaluated up to ten results for each query, but there may be more than ten relevant results in the repository).

We created the variable precision P as a categorization of the response variable confidence C (i.e., includes results as relevant if ranked as 3 or 4 by users, and results as irrelevant if ranked as 1 or 2 by users). We did this for two reasons: to improve discrimination of subjects in the resulting data and additionally validate statistical evaluation of results. Precision P imposes a stricter boundary on what is considered reusable code. For example, consider a situation where one participant assigns the level two to all returned functions, and another participant assigns level three to half of these functions and level one to the other half. Even though the average of $C = 2$ in both cases, the second participant reports much higher precision $P = 0.5$ while the precision that is reported by the first participant is zero. Achieving statistical significance with a stricter discriminative response variable will give assurance that the result is not accidental.

7.4.2. Normalized Discounted Cumulative Gain. Normalized Discounted Cumulative Gain (NDCG) is a metric for evaluating search engines [Anquetil and Lethbridge 1998], and has been used to evaluate source-code search engines before [Bajracharya et al. 2010]. Unlike precision or confidence, NDCG considers the *order* of the results given by search engines, such that relevant results at the top of the list of results are rewarded. NDCG is a normalization of the metric Discounted Cumulative Gain, which is computed as

$$G = C_1 + \sum_{i=2}^{10} \frac{C_i}{\log_2 i},$$

where C_1 is the confidence of the result in position 1, and C_i is the confidence of the result in position i. NDCG is computed as $NG = G/iG$, where iG is the ideal DCG (e.g.,

all ten results are highly relevant, and confidence is always 4). For brevity, we refer to NDCG as *NG* throughout the rest of this article.

7.4.3. ANOVA and Randomization Tests. We compute the metrics C, P, and NG based on the results of the first experiment. To answer our first two research questions, we establish the statistical significance of the differences in those metrics using two tests: First, we use one-way ANOVA to establish the statistical significance of differences in the metrics from our experiments. Second, we use randomization tests to establish the directionality of any differences. For RQ_3, we used a two-tailed Student's t-Test to compare the difference of means between levels of C as reported by participants with different experience levels. Finally, for RQ_4, we used a paired two-sample Student's t-Test to compare the mean NG given by our engine, under different configurations, for the same set of queries. One-way ANOVA has been successfully used to evaluate search engine results before [Grechanik et al. 2010; McMillan et al. 2011b]. ANOVA assumes that the population is normally distributed, and we meet this criterion because of the law of large numbers, which states that if a sufficiently large sample is used (minimum 30 participants), then the central limit theorem applies [Sirkin 2006]. We had 49 participants in our first experiment, so we assume that our population is normally distributed. If we determine with ANOVA that the difference of means is statistically significant, our next step is to establish the directionality of the difference of means. Recent work has recommended the use of randomization tests for evaluating the results of IR-based engines [Smucker et al. 2007]. Therefore, we use randomization tests in our studies. We used the default 100 iterations for randomization as given by Smucker et al. [2007]. In determining statistical significance, we used the traditional measure of alpha equals 0.05.

7.4.4. Task Design. We designed 15 tasks for participants to work on during the first experiment in a way that these tasks belong to domains that are easy to understand, and they have similar complexity. The authors of this article visited various programming forums and Internet groups to extract descriptions of tasks from the questions that programmers asked. In addition, we interviewed several programmers at Accenture who explained what tasks they worked on in the past year. Additional criteria for these tasks are that they should represent the diverse real-world programming tasks for which programmers use source-code search engines [Bajracharya and Lopes 2009] and should not be biased towards any of the search engines that are used in this experiment. We avoid bias to any search engine by using engines with large repositories of source code that contain a variety of tasks. These tasks represent real-world programming tasks. They span from specific file (e.g., music files) manipulation (e.g., read/write) to image processing (e.g., rotate image, adjust white balance) to addressing security concerns (e.g., encryption/decryption). The description of these tasks and the results of the experiment are available for download for replication purposes.[10]

The following three tasks are examples from the set of 15 tasks we used in our experiments.

—Implement a module for reading and playing midi files.
—Implement a module that adjusts different parameters of a picture, including brightness, contrast, and white balance.
—Build a program for managing USB devices. The program should implement routines such as opening, closing, writing, and reading from an USB device.

[10]http://www.cs.wm.edu/semeru/portfolio/ExperimentMaterials.tar.gz (accessed and verified on 5/10/12).

7.5. Hypotheses

We introduce the null hypothesis $H_{0\text{-NULL}}$ to evaluate the statistical significance of the difference in means for C, P, and NG as reported by the participants in the first user study, and are designed to answer RQ_1. $H_{1\text{-NULL}}$ is a hypothesis for answering RQ_3, while we answer RQ_4 by evaluating $H_{2\text{-NULL}}$.

$H_{0\text{-}NULL}$ There is no statistical difference in the mean value of *C*, *P*, or *NG* as reported by the case study participants using Portfolio, Google Code Search, or Koders.

$H_{1\text{-}NULL}$ There is no statistical difference in the mean value of *C* as reported by the case study participants who had at least 2 years C/C++ experience and those who had less than 2 years C/C++ experience.

$H_{2\text{-}NULL}$ There is no statistical difference in the mean value of *NG* as calculated when the list of results from Portfolio was ordered with PageRank enabled and when ordered with PageRank disabled.

We also form nine hypotheses (H_3 to H_{11}) that state that *C*, *P*, and *NG* for Google Code Search and Koders are higher than for Portfolio when searching over C/C++. Similarly, three hypotheses (H_{12} to H_{14}) state that *C*, *P*, and *NG* for Koders are higher than for Portfolio when searching over Java. These hypotheses are only evaluated in the case where $H_{0\text{-NULL}}$ is rejected for either C/C++ or for Java.

7.6. Threats to Validity

In this section, we discuss threats to the validity of the experiments and how we address these threats.

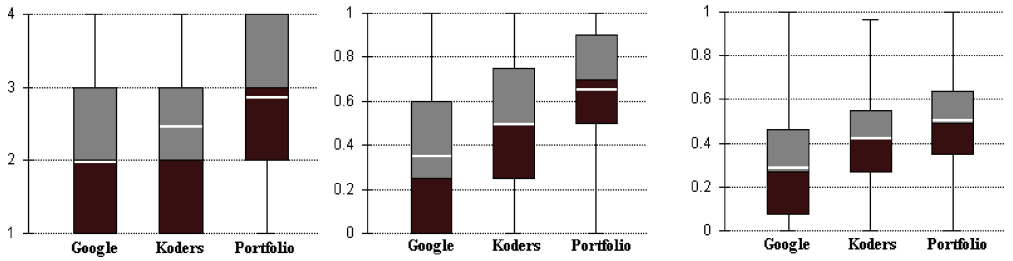
7.6.1. Internal Validity. Internal validity refers to the degree of validity of statements about cause-effect inferences. In the context of our experiment, threats to internal validity come from confounding the effects of differences among participants, tasks, and time pressure.

Participants. Since evaluating hypotheses is based on the data collected from participants, we identify two threats to internal validity: C++ proficiency and motivation of participants. Even though we selected participants who have strong working knowledge of C++ as documented by human resources at Accenture, we did not conduct an independent assessment of how proficient these participants are in C++. This threat is mitigated by the fact that out of 44 participants from Accenture, 31 have worked on successful commercial projects as C++ programmers for more than two years.

The other threat to validity is that not all the participants could be motivated sufficiently to evaluate retrieved code fragments or functions. We addressed this threat by asking participants to explain in a couple of sentences why they chose to assign certain confidence levels to the retrieved source code, and we discarded 27 results for all search engines that were not sufficiently explained (or had no explanations associated with them).

Time pressure. Each experiment lasted for two hours. For some participants, this was not enough time to explore all 50 retrieved code fragments for five tasks (ten results for each of five tasks). Therefore, one threat to validity is that some participants could try to accomplish more tasks by shallowly evaluating retrieved code. To counter this threat we notified participants that their results would be discarded if we did not see sufficient re-ported evidence of why they evaluated retrieved code fragments and functions with certain confidence levels.

7.6.2. External Validity. To make the results of this experiment generalizable, we must address threats to external validity, which refer to the generalizability of a casual



(a) confidence level C (b) precision P (c) normalized discounted gain, NG

Fig. 6. Statistical summary of the results of the first experiment for C, P, and NG. The central box extends from the lower to upper quartile. The thick white line is the mean. The thin black line is the median. The thin vertical line extends from the minimum to the maximum value.

Table III. Results of Hypothesis for One-Way ANOVA

H	P	F	F_{crit}	Decision
$H_{0\text{-NULL}}$	$5e^{-108}$	261.3	3	Reject

relationship beyond the circumstances of our experiment. The fact that supports the validity of this experimental design is that the participants are highly representative of professional C/C++ programmers. However, a threat to external validity concerns the usage of search tools in industrial settings, where requirements are updated on a regular basis. Programmers use these updated requirements to refine their queries and locate relevant code fragments or functions using multiple iterations of working with search engines. We addressed this threat only partially, by allowing programmers to refine their queries multiple times.

Another threat to external validity comes from different sizes of software repositories. Koders.com claims to search more than 3 billion LOC in all programming languages, which is also close to the number of LOC reported by Google Code Search. Even though we populated Portfolio's repository with close to 270 million LOC of C/C++ code and 440 million LOC of Java code, it still remains a threat to external validity.

8. ANALYSIS OF THE USER STUDIES

In this section, we report the results of our experiments and answer our research questions.

8.1. RQ_1 – Portfolio, Google, and Koders Using C/C++

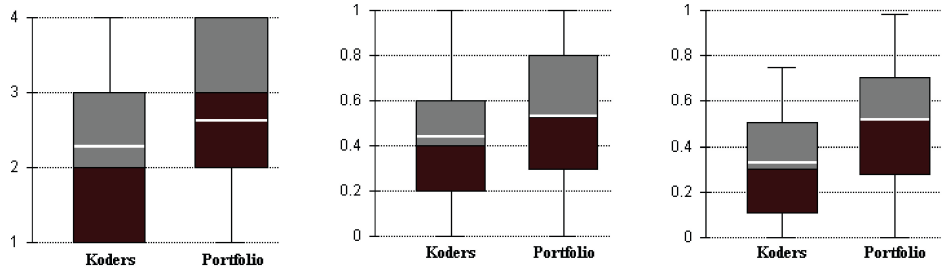
Figure 6 shows a statistical summary of *C*, *P*, and *NG* as reported by participants in our first experiment. The mean confidence level for Portfolio is higher than for Google or Koders, suggesting that Portfolio returns more relevant results than either of those engines. Moreover, the proportion of results that are relevant (measured by precision *P*) is highest for Portfolio, and those relevant results appear nearer to the top of the list of results (measured by discounted gain *NG*).

The results of our ANOVA evaluation of $H_{0\text{-NULL}}$ are in Tables III and IV. The value of *F* exceeds F_{crit}, and *p* is less than 0.05. Therefore, we find evidence to reject the $H_{0\text{-NULL}}$. To evaluate the directionality of the means for each metric, we pose hypotheses H_3 to H_{11} for each of these metrics, as described in Section 7.5. In each case, we reject the hypothesis and conclude that participants who use Portfolio report higher relevance when finding functions in source code than those who use Google Code Search or Koders. In addition, we find that the participants who used Koders report higher relevance in the results than when using Google Code Search.

Table IV. Statistical Summary of the First Experiment for Dependent Variable Specified in the Column Var

Var	Approach	Samples	Min	Max	Median	Mean	StdDev
C	Portfolio	1276	1	4	3	2.86	1.07
	Google	1373	1	4	2	1.97	1.11
	Koders	1486	1	4	2	2.45	1.12
P	Portfolio	184	0	1	0.7	0.65	0.28
	Google	198	0	1	0.25	0.35	0.33
	Koders	208	0	1	0.5	0.49	0.30
NG	Portfolio	184	0	1	0.50	0.51	0.43
	Google	198	0	1	0.27	0.29	0.57
	Koders	208	0	0.97	0.42	0.42	0.42

We report the extremal values, median, mean, and standard deviation.



(a) confidence level C (b) precision P (c) normalized discounted gain, NG

Fig. 7. Statistical summary of the results of the second experiment for C, P, and NG. The central box extends from the lower to upper quartile. The thick red line is the mean. The thin black line is the median. The thin vertical line extends from the minimum to the maximum.

Table V. Results of Hypothesis for One-Way ANOVA

H	P	F	F_{crit}	Decision
$H_{0\text{-NULL}}$	$1.22e^{-6}$	23.9	3.85	Reject

Table VI. Statistical Summary of the Second Experiment for Dependent Variable Specified in the Column Var

Var	Approach	Samples	Min	Max	Median	Mean	StdDev
C	Portfolio	541	1	4	3	2.62	1.06
	Koders	405	1	4	2	2.28	1.06
P	Portfolio	61	0	1	0.53	0.53	0.30
	Koders	62	0	1	0.40	0.44	0.32
NG	Portfolio	61	0	0.98	0.52	0.51	0.25
	Koders	62	0	0.75	0.30	0.33	0.22

We report the extremal values, median, mean, and standard deviation.

8.2. RQ_2 – Portfolio and Koders Using Java

A statistical summary of *C*, *P*, and *NG* for the second experiment is in Figure 7. The mean values of all the metrics are higher for Portfolio than for Koders. Tables V and VI show the results from our ANOVA test to evaluate the significance of these differences. The value of F exceeds F_{crit}, and p is less than 0.05, and we reject $H_{0\text{-}NULL}$. In evaluating the directionality of the means, we found statistically significant differences in the means for *C* and *NG* (i.e., p from our randomization test was less than 0.05 for H_{12} and

Portfolio: Searching Relevant Functions and their Usages in Millions of Lines of Code 37:19

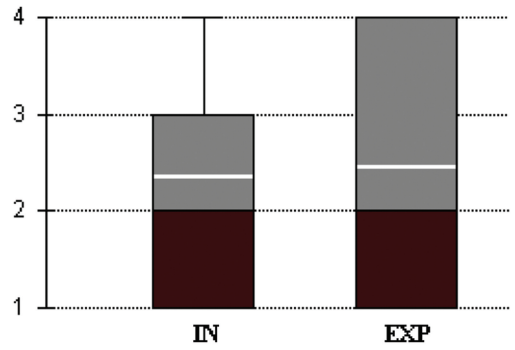


Fig. 8. Confidence level as reported by experienced (EXP) and inexperienced (IN) programmers in the first experiment.

Table VII. Results of Hypothesis for One-Way ANOVA

H	P	t	t_{crit}	Decision
$H_{1\text{-NULL}}$	0.054	1.93	1.96	Not Reject

Table VIII. Statistical Summary of Reported C by Participants' Experience Level

Var	Experience	Samples	Min	Max	Median	Mean	StdDev
C	≥2 years	2883	1	4	2	2.45	1.158
	<2 years	1252	1	4	2	2.37	1.162

H_{14}). On the other hand, we found no statistically significant difference between the values of *P* for Portfolio and Koders.

We conclude that Portfolio outperformed Koders over the Java repository because the participants reported higher levels of relevance according to two of the three metrics. In terms of the proportion of results that were relevant for each query (the metric *P*), Portfolio and Koders are statistically equal. Nevertheless, Portfolio's performance in terms of *C* and *NG* indicates that not only are the results from Portfolio more relevant on average (*C*) than from Koders, but those relevant results occur higher on the list in Portfolio than in Koders.

8.3. RQ_3–Experience Relationships

We divided the reported *C* based on the programming experience of the participants (Figure 8). From the first experiment, we created two groups: one in which participants had at least 2 years of C/C++ experience, and another group that had less than 2 years experience. Tables VII and VIII show the results of our analysis of the null hypothesis for this study. In this case, t is less than t_{crit} and p is greater than 0.05. Therefore, we do not find evidence to reject $H_{1\text{-NULL}}$, and thus conclude that there is no statistically strong relationship between the reported *C* based on experience.

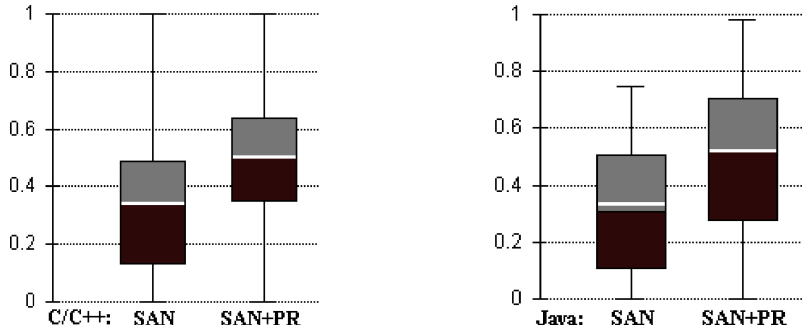
8.4. RQ_4–Effectiveness of PageRank

We use PageRank to rank the list of results returned by the IR engine and SAN. PageRank only contributes to the ranking of the results; it does not add new functions to the list of results in Portfolio. For example, Table IX shows the search results from Portfolio with PageRank enabled and disabled. The functions on the list are the same, and *C* and *P* will be the same whether or not we use PageRank. On the other hand, *NG* measures the order of the list such that *NG* rewards search engines for returning more relevant results higher in the list.

Table IX. The Top Ten Functions Returned by Portfolio for the Query *mscdex emulate*

"mscdex emulate"				
Function Name	SAN	PR	SAN+PR	User Ranking
isoDrive	78.70	60.334	73.19	4
initialisation	73.18	59.481	69.07	4
Activate	71.79	59.481	68.10	3
MSCDEX_AddDrive	63.58	59.240	62.28	3
main	63.13	59.197	61.95	3
AddDrive	63.08	59.325	61.95	1
MSCDEX_Interrupt_Handler	63.05	59.240	61.91	4
MSCDEX_IOCTL_Input	63.08	59.169	61.91	4
DOS_SetupPrograms	62.05	59.240	61.21	1
UpdateMscdex	61.91	59.240	61.11	2
NG	79.50	78.19	79.94	

The first column is the function name. The second column is the spreading activation score given by Portfolio. The third column is the PageRank score for the function. The fourth column is the final score assigned by Portfolio to the function (a combination of SAN and PR; see Section 4.4). The fifth column is the confidence score given by a participant in our experiment when that participant used Portfolio with the query *mscdex emulate*. The rows labeled NG indicate the Normalized Discounted Cumulative Gain for the results when ordering the functions by SAN, PR, and SAN combined with PR. Notice that NG is higher when using the combination of SAN and PR than with SAN or PR alone.



(a) Experiment 1, C/C++ (b) Experiment 2, Java

Fig. 9. Normalized Discounted Cumulative Gain for Portfolio with PageRank enabled ($\text{Portfolio}_{\text{SAN+PR}}$) and disabled ($\text{Portfolio}_{\text{SAN}}$).

We ran Portfolio using every query generated by the participants for each experiment. Then, we matched the C that they reported for the results when using a given query. We ran Portfolio again with the same queries with PageRank disabled, and computed the NG for each case. Figure 9 is a summary of the values we computed for NG. In Table X, we show the results from evaluating $H_{2\text{-NULL}}$. In Table XI, we show that t is greater than t_{crit} and p is less than 0.05 in both cases, and therefore find evidence to reject $H_{2\text{-NULL}}$. Therefore, we conclude that our use of PageRank results in more relevant functions being ranked higher than less relevant functions. Thus, we conclude that PageRank is an important component of our solution, since it helps to promote more relevant functions in the results.

9. QUALITATIVE ANALYSIS OF USER COMMENTS

Forty-five of the participants in our first experiment answered an exit survey related to source-code search activities. The survey included the nine questions in Table XIII

Portfolio: Searching Relevant Functions and their Usages in Millions of Lines of Code 37:21

Table X. Results of Hypothesis for a Paired t-Test for RQ_4

Lang.	P	t	t_{crit}	Decision
C/C++	$<1e^{-4}$	10.6	1.65	Reject
Java	$<1e^{-4}$	4.08	1.67	Reject

Table XI. Statistical Summary of Normalized Gain (NG) with and without PageRank

Lang	Approach	Samples	Min	Max	Median	Mean	StdDev
C/C++	$Portfolio_{SAN+PR}$	181	0	1	0.494	0.503	0.433
	$Portfolio_{SAN}$	181	0	1	0.353	0.341	0.686
Java	$Portfolio_{SAN+PR}$	61	0	0.984	0.525	0.518	0.256
	$Portfolio_{SAN}$	61	0	0.748	0.300	0.332	0.222

Table XII. The Performance for Portfolio

# of users	Average (ms)	Median	90%	Minimum	Maximum
5	2723	2289	4453	2052	4453
10	2948	2797	3761	2390	4501
15	3913	3852	4789	3141	5151
20	5404	5347	6498	3055	7259
25	6655	6477	7896	4437	8539
30	7455	7542	9592	3820	10251
35	9051	8738	11915	5848	12916
40	9272	9066	14454	3334	17728
45	10470	9687	15047	5802	16631
50	10611	9938	17163	5879	18249

The first column is the number of users who access Portfolio simultaneously. The second, third, fourth, fifth, and sixth column are the average, the median, the 90% percentile, the minimum, and the maximum response time in milliseconds correspondingly.

similar to previous surveys [Sim et al. 2011]. Scanned copies of the surveys are available on our Web site for analysis by other researchers, and in this section we focus on three main findings. First, 33 of the 45 respondents found the call graph visualization in Portfolio helpful in understanding the search results. Of the users who did not find it useful, many reported wanting to see a visualization of other data, such as a UML representation or integration with a development environment. These results point to a strong need for source-code search techniques that support visualization of the results.

Our second main finding was that the programmers mostly wanted to find code snippets to directly reuse in their projects as well as API call examples. Forty users answered a question related to the goal of searching, and 30 reported looking for code snippets to directly reuse. Twenty-eight wanted to find examples of API call usage. Sixteen were looking for specific algorithms, and only six wanted some other artifact. Portfolio helps fill the need of programmers looking for relevant source-code snippets as well as for API usage examples (by showing these API calls in the visualization), though some programmers reported wanting further support for exclusively searching API calls. Providing this support is an area of our future work.

Finally, we found that the respondents overwhelmingly turn to general-purpose search engines when looking for source code. Thirty-six programmers used general-purpose search engines to search for source code, and many of the programmers wrote that the context behind the source code was very important. The programmers searched through mailing lists and developer forums to find this context, but overall reported dissatisfaction during source-code search activities. For example, one programmer

Table XIII. The Nine Questions We Asked in the Exit Questionnaire

#	Question
1	Did you find the call-graph visualization in Portfolio useful during your searches?
2	How often do you search for source code on the Internet?
3	What information sources do you use when searching for source code?
4	What search sites do you use to search for source code?
5	Which of the following programming and/or scripting languages have you had some experience with?
6	What do you hope to find when looking for source code online?
7	Please describe one or two scenarios when you were looking for source code on the Internet. Please address details such as: What were you trying to find? How did you formulate your queries? What information sources did you use to find the source code? Which implementation language were you working with? What criteria did you use to decide on the best match?
8	After searching, once you have identified some possible candidates, what are the criteria that guide you to finally select source code that you will use?
9	If you could have the ideal search engine while searching for source code on the Internet, what additional features would you like to have?

commented that he "finds a few [results] via Google search, but was unlikely to find what I [was] looking for." Another developer stated that "my next project needs SQL code, I had to find not the code, but how it was used and its function calls." In Portfolio, we provide usage information about functions by showing how these functions are used in the context of many other functions.

10. PERFORMANCE EVALUATION

We illustrated in Section 8 how Portfolio outperformed Google Code Search and Koders in returning relevant results. In this section, we evaluate the performance of Portfolio in terms of speed of the source-code search. Code search speed is another important metric for a code search engine because it is related to the usability of the search engine. We used Apache JMeter,[11] an open-source load testing tool, to load test Portfolio and measure its performance. We wrote JMeter scripts to mimic multiple users accessing Portfolio simultaneously and record the response time. The results are summarized in Table XII. The results show that the more users access Portfolio at the same time, the higher response time we get. The median response time for 50 users is less than 10 seconds when Portfolio is running on a server with an Intel Xeon X5550 CPU and 24GB of ram.

11. DISCUSSION

We showed in Section 8.1 that Portfolio outperforms both Google Code Search and Koders in terms of relevance of the results. In Section 8.4, we determined that PageRank is a key factor in Portfolio's improvement of the ordering of the results. However, PageRank does not explain why Portfolio outperforms Google and Koders in terms of overall relevance, because Portfolio only uses PageRank to rank the final list of functions; it does not use PageRank to retrieve functions. In this section, we explore how our combination of spreading activation with textual similarity retrieved more relevant results than with textual similarity alone.

Spreading activation propagates the textual similarity score from functions that contain query keywords to other functions that are connected to those functions in the call graph (see Section 4.3). In doing so, Portfolio locates chains of function invocations relevant to the query, and many of these functions do not contain keywords from

[11]http://jmeter.apache.org (accessed and verified 5/10/12).

Portfolio: Searching Relevant Functions and their Usages in Millions of Lines of Code 37:23

Table XIV. A Breakdown of the Results from Portfolio Rated by Participants in the First Case Study

		Proximity to Keyword			
		0	1	2	
Relevance	4	128	95	33	256
	3	110	114	22	246
	2	99	94	20	213
	1	79	68	14	161
		416	371	89	Totals

There are 876 functions in all. These functions were obtained during the first case study by participants using Portfolio. The participants rated each function on a scale from 1 to 4 (Section 7.1). The rows in this table correspond to these ratings (e.g., 256 functions were rated "4", highly relevant, by participants using Portfolio). The columns correspond to the proximity of the function, in the call graph, to another function that contains a keyword from the query. Distance 0 means that a keyword occurred in that function. Distance 1 means that the function did not contain a keyword from the query, but was 1 edge away from a keyword-containing function in the call graph. Note that less than half of all retrieved functions contained a keyword from the query.

the query. Nevertheless, during our first case study, we found that the programmers rated these functions as relevant. Table XIV contains a breakdown of the results from Portfolio from the first case study (see Section 6.1). We divided the results in terms of relevance reported by the case study participants and in terms of proximity to functions that contain keywords.[12]

A recent study found that 100% recall of relevant results can be achieved by looking at functions two edges away from functions containing query keywords [Hill et al. 2011]. We confirm these results in our case study. All of the relevant functions found by Portfolio were within two dependencies of a function containing a keyword from the query. Crucially, however, only 48% of the functions actually contained a query keyword. Over half of all the results, and 53% of all relevant results, did not contain a keyword from the query. Source-code search engines that only consider keyword matches, such as Google Code Search and Koders, miss these relevant functions. Portfolio finds these functions in the context of a chain of function invocations, and displays them to the user.

12. RELATED WORK

In this section, we compare Portfolio to other approaches, different code mining techniques, and tools have that been proposed to find relevant software components. These tools and approaches are shown in Table XV. A main intention of this table is to illustrate how Portfolio differs from other listed related approaches.

To compare Portfolio with other related approaches we introduce different variables that describe different properties of code search approaches. The variable Granularity specifies how search results are returned by different approaches, specifically at the project, function, or unstructured text granularity levels. The variable Usage specifies if an approach provides additional information on how retrieved code units are used. The variable Search Method specifies what kind of search algorithm or technique a given code search approach uses, that is, Pagerank, spreading activation, simple word matching, parameter type matching, or query expansion. Finally, the variable

[12]The proximity is calculated as the number of edges that must be traversed in order to move from the result to a function that contains a keyword. For example, in Figure 1, if the query was "mip map", the function H would be 2 edges away from F.

Table XV. Comparison of Portfolio with Other Related Approaches

Approach	Granularity		Search Method	Result
	Unit	Usage		
AMC [Hill and Rideout 2004]	U	N	W	T
Code Conjurer [Hummel et al. 2008]	F,U	Y	W,T	T
CodeBroker [Ye and Fischer 2002]	P,U	Y	W,Q	T
CodeFinder [Henninger 1996]	F,U	Y	W,Q	T
CodeGenie [Lemos et al. 2007]	P	N	W	T
Dora [Hill et al. 2007]	U	N	W	T
Exemplar [McMillan et al. 2011]	A	Y	W	T
Ferret [de Alwis and Murphy 2008]	F,U	N	W,T	T
Google Code Search	U	N	W	T
Gridle [Puppin and Silvestri 2006]	U	N	W	T
Hipikat [Cubranic et al. 2005]	P	Y	W,Q	T
Koders	U	N	W	T
Krugle	U	N	W	T
MAPO [Zhong et al. 2009]	F	N	W,Q	T
Mica [Stylos and Myers 2006]	U,F	Y	W,Q	T
ParseWeb [Thummalapenta and Xie 2007]	U,F	N	W,Q	T
Portfolio [McMillan et al. 2011]	*F,P*	*Y*	*P,S,W*	*G*
Prospector [Mandelin et al. 2005]	F	N	T	T
S^6 [Reiss 2009]	F,P,U	Y	W,Q	T
SNIFF [Chatterjee, Juvejar and Sen 2009]	F,U	Y	T,W	T
Sourceforge	A	N	W	T
Sourcerer [Bajracharya and Lopes 2009]	F,P,U	Y	P,W	T
SPARS-J [Inoue et al. 2005]	F	Y	P	T
SpotWeb [Thummalapenta and Xie 2008]	U	N	W	T
SSI [Bajracharya, Ossher and Lopes 2010]	F	N	P,W,T	T
Strathcona [Holmes and Murphy 2005]	F	Y	W	T
xSnippet [Sahavechaphan and Claypool 2006]	F	Y	T,W	T

Column Granularity specifies how search results are returned by each approach (*P*rojects, *F*unctions, or *U*nstructured text), and if the usage of these resulting code units is shown (*Y*es or *N*o). The column Search Method specifies the search algorithms or techniques that are used in the code search engine, that is, *P*agerank, *S*preading activation, simple *W*ord matching, parameter *T*ype matching, or *Q*uery expansion techniques. Finally, the last column tells if the search engine shows a list of code fragments as *T*ext or it uses a *G*raphical representation of search results to illustrate code usage for programmers.

Visualization tells if an approach shows code fragments as text or it uses a graphical representation of search results to illustrate code usage for programmers.

Code Conjurer [Hummel et al. 2008] is an Eclipse plug-in that extracts interface and test information from a developer's coding activities and uses this information to issue test-driven searches to a code search engine. It presents components matching the developer's needs as reuse recommendations without disturbing the development work. Unfortunately, there is no comprehensive evaluation of Code Conjurer, and its architectural details are not published. Based on available facts, Code Conjurer conceptually differs from Portfolio in that it does not model how developers search for relevant code; it rather concentrates on not disrupting programmers' work by allowing them to stay within the Eclipse programming environment.

CodeFinder iteratively refines code repositories in order to improve the precision of returned software components [Henninger 1996]. Unlike Portfolio, CodeFinder heavily

depends on the descriptions (often incomplete) of software components to use word matching, while Portfolio uses Pagerank and SANs to help programmers navigate and understand usages of retrieved functions. That is, Portfolio can still find relevant code even if there are no precise matches between words in the code and in queries.

The Codebroker system uses source code and comments written by programmers to query code repositories to find relevant artifacts [Ye and Fischer 2002, 2005]. Similar to CodeFinder, Codebroker depends upon the descriptions of documents and meaningful names of program variables and types, so that it can match words precisely to return relevant code, and it is often the case that precise word matches do not result in relevant code. In contrast, Portfolio goes beyond matching words by using SAN and Pagerank to locate relevant code that cannot be returned using word matching.

The tool Ferret uses different "spheres" of program information as a way of integrating the relationships among program artifacts [de Alwis and Murphy 2008]. These spheres are then used to answer conceptual questions about the artifacts. Like the search model in Portfolio, the sphere model is based on the relationships among various objects. Ferret is implemented with four different spheres including static and dynamic function call data. The main difference between Portfolio and Ferret is that Portfolio locates relevant source code from large repositories of software, whereas Ferret answers questions specific to a particular programming context.

Similar to the Portfolio's model, Robillard et al. proposed a model to represent different requirements in source code, and this model combines different knowledge about program artifacts and their relationships [Robillard and Murphy 2007]. Portfolio expands on modeling of requirements with association and navigation models of functions in the context of source-code search. Locating implementations of these requirements in the context of specific programs is an extensive research area, and these techniques have used information retrieval [Marcus et al. 2004], execution information [Dit et al. 2011; Poshyvanyk et al. 2007], and formal concept analysis [Poshyvanyk et al. 2012].

Dora is a tool for software maintenance to help programmers locate the regions of their programs that relate to a given starting function [Hill et al. 2007]. Similar to Portfolio, Dora determines which functions connect to the starting function, and then filters these functions to determine the "neighborhood" of functions relevant to the programmer's task. Just like Portfolio, Dora also determines a group of relevant functions near a starting point in a call graph. On the other hand, Dora uses textual similarity of functions in order to filter dissimilar functions out of the results. This filtering is unlike Portfolio, which locates relevant functions even if those functions do not contain keywords from the query or the set of starting points.

Portfolio is fundamentally different from SSI, which determines the similarity of source code based on the API calls used in that code, and builds an index of the source code that includes terms from similar source code [Bajracharya et al. 2010]. Unlike Portfolio, SSI searches by only matching keywords from the query to keywords in the expanded index. In this way, SSI can return groups of functions that use the same API calls. In contrast, Portfolio locates functions that are connected to each other in the call graph, thereby finding relevant functions that may otherwise be missed by SSI.

Even though it returns code snippets rather than the code for entire functions, Mica is similar to Portfolio since it uses API calls from the Java Development Kit to guide code search. Mica uses help documentation to refine the results of the search [Stylos and Myers 2006], which may not be always available and it may not have sufficient quality, while Portfolio automatically retrieves functions from arbitrary code repositories and it uses models that rely on functional abstractions to help programmers evaluate the potential of code reuse faster and with higher precision. Possible future work may be to explore a connection between Mica and Portfolio to use external help documentation to improve the quality of code search.

Exemplar, SNIFF, and Mica use documentation for API calls for source-code search [Chatterjee et al. 2009; Grechanik et al. 2010; Stylos and Myers 2006]. SNIFF then performs the intersection of types in these code chunks to retain the most relevant and common part of the code chunks. SNIFF also ranks these pruned chunks using the frequency of their occurrence in the indexed code base. In contrast to SNIFF, Portfolio uses navigation and association models that reflect the behavior of programmers to improve the precision of the search engine. In addition, Portfolio offers a visualization of usages of functions that it retrieves automatically from existing source code, thus avoiding the need for third-party documentation for API calls.

Exemplar is a search engine that returns applications based on the API calls that those applications use [Grechanik et al. 2010; McMillan et al. 2011a]. Exemplar matches keywords in a query to keywords in the documentation of API calls, and then returns applications that use these API calls. The key difference between Exemplar and Portfolio is that Portfolio considers how the search results are related to one another. Exemplar locates software applications that are relevant to the query. In contrast, Portfolio locates chains of function invocations that implement the tasks in user queries. In Portfolio, we use Spreading Activation to propagate the textual similarity of functions to other functions connected via the call graph. Then, we rank these functions by combining the score from Spreading Activation with the PageRank value of the functions as computed in the call graph of thousands of projects.

Web-mining techniques have been applied to graphs derived from program artifacts before [Saul et al. 2007]. Notably, Inoue et al. proposed Component Rank [Inoue et al. 2003, 2005] as a method to highlight the most frequently used classes by applying a variant of PageRank to a graph composed of Java classes and an assortment of relations among them. Quality Of Match (QOM) ranking measures the overall goodness of match between two given components [Puppin and Silvestri 2006; Tansalarak and Claypool 2005], where these components are represented as multidimensional vectors containing words from these components, and then these vectors are coclustered using a measure of similarity between them. Unlike QOM, Portfolio retrieves functions based on surfing behavior of programmers and associations between concepts in these functions. Gridle also applies PageRank to a graph of Java classes [Puppin and Silvestri 2006]. In Portfolio, we apply PageRank to a graph with nodes as functions and edges as call relationships among the functions. In addition, we use spreading activation on the call graph to retrieve chains of relevant function invocations, rather than single fragments of code.

Portfolio is also related to the Sourcerer source-code search engine. Bajracharya et al. combine PageRank and textual similarity to rank source-code components [Bajracharya and Lopes 2009], and find that PageRank significantly improves the quality of the search results relative to using only textual similarity. In this article, we confirm this result in RQ_4, and expand on it with Portfolio, which uses Spreading Activation to locate groups of functions that relate to the developer query. Specifically, with Spreading Activation, Portfolio locates functions which do not contain keywords from the query, but are connected via the call graph to other functions which do contain keyword matches. Sourcerer will not locate relevant functions unless those functions are textually similar to the query.

Programming task-oriented tools like Prospector, Hipikat, Strathcona, and xSnippet assist programmers in writing complicated code [Cubranic et al. 2005; Holmes and Murphy 2005; Holmes et al. 2006; Mandelin et al. 2005; Sahavechaphan and Claypool 2006]. However, their utilities are not applicable when searching for relevant functions given a query containing high-level concepts with no source code. Similarly, in keyword programming, keywords written by a developer are immediately translated into a few lines of code and placed into the developer's project [Little and Miller 2007, 2008]. Portfolio differs from keyword programming because we locate chains of invocations

that implement tasks described by queries, rather than a few lines of code. Given a proper support for keyword programming, Portfolio can be used to enhance its ability to find chains of relevant functions to replace keywords with implementations.

Robillard proposed an algorithm for calculating program elements of likely interest to a developer [Robillard 2005, 2008]. Portfolio is similar to this algorithm in that it uses relations between functions in the retrieved projects to compute the level of interest (ranking) of the project. However, Robillard does not use models that reflect the surfing behavior of programmers and association models that improve the precision of search. We think there is a potential in exploring possible connections between Robillard's approach and Portfolio.

S^6 is a code search engine that uses a set of user-guided program transformations to map high-level queries into a subset of relevant code fragments [Reiss 2009], not necessarily functions. Like Portfolio, S^6 uses query expansion; however, it requires additional low-level details from the user, such as data types of test cases. Adding these low-level details requires programmers to know implementation details before performing code search, which often defeats the purpose of code search to find implementations based only on high-level information from requirements.

13. CONCLUSION

We have presented Portfolio, a source-code search engine, and compared it to Google Code Search and Koders, two large and popular source-code search engines. We found with strong statistical significance that Portfolio outperformed these engines in terms of confidence, precision, and normalized discounted cumulative gain. The experiment we performed for these findings was based on standard information retrieval methodology for evaluating search engines, and the participants in the experiment were professional programmers at a large software company.

We also evaluated the tools that generate the data on which Portfolio relies, and we released these tools to the public. These tools can generate a call graph of tens of thousands of projects and millions of lines of code. We implemented Portfolio with this data and computed the PageRank for over 8.5 million C/C++ functions and 14 million Java methods in call graphs with tens of millions of edges. Portfolio combines textual similarity and Spreading Activation algorithms to retrieve relevant functions, and then uses these algorithms plus PageRank to rank the list of results. We also display a call graph to the user which shows how the functions interact in a specific context.

We have substantiated the findings of other researchers. Specifically, we found that PageRank significantly improves the ranking of the results such that more relevant functions occur near the top of the list of results. We also found that all of the relevant functions in our experiment occurred within two edges of a function containing a query keyword. On the other hand, over half of the relevant functions did not contain a keyword from the query. Our results strongly point to the need for source-code search engines to explore the call graph to find relevant source code. We have built, released, and evaluated Portfolio, a source-code search engine that helps programmers by displaying this relevant source code.

ACKNOWLEDGMENTS

We warmly thank nine graduate students, Luca DiMinervino, Arunrajkumar Dharumar, Rohan Dhond, Sekhar Gopisetty, Hariharan Subramanian, Ameya Barve, Naresh Regunta, Ashim Shivhare, Denzil Rodrigues, from the University of Illinois at Chicago who contributed to Portfolio as part of their work towards the completion of their Master of Science in Computer Science degrees. We also thank Bogdan Dit from the College of William and Mary for his help in building parts of Portfolio. We are grateful to the anonymous TOSEM and ICSE'11 reviewers for their relevant and useful comments and suggestions, which helped us to significantly improve an earlier version of this article.

REFERENCES

ANQUETIL, N. AND LETHBRIDGE, T. 1998. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of the Annual IBM Centers for Advanced Studies Conference (CASCON'98)*. 213–222.

BAJRACHARYA, S. AND LOPES, C. 2009. Mining search topics from a code search engine usage log. In *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*. IEEE Computer Society.

BAJRACHARYA, S., OSSHER, J., AND LOPES, C. V. 2010. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of the 18th International Symposium on the Foundations of Software Engineering (FSE'10)*.

BRIN, S. AND PAGE, L. 1998. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th International Conference on World Wide Web*. 107–117.

CHATTERJEE, S., JUVEJAR, S., AND SEN, K. 2009. SNIFF: A search engine for java using free-form queries. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE'09)*. 385–400.

COLLINS, A. AND LOFTUS, E. 1975. A spreading-activation theory of semantic processing. *Psychol. Rev. 82*, 21.

Corbi, T. A. 1989. Program understanding: Challenge for the 1990s. *IBM Syst. J. 28*, 294–306.

CRESTANI, F. 1997. Application of spreading activation techniques in information retrieval. *Artif. Intell. 11*, 29.

CUBRANIC, D., MURPHY, G. C., SINGER, J., AND BOOTH, K. S. 2005. Hipikat: A project memory for software development. *IEEE Trans. Softw. Engin. 31*, 446–465.

DAVISON, J. W., MANCL, D. M., AND OPDYKE, W. F. 2000. Understanding and addressing the essential costs of evolving systems. *Bell Labs Tech. J. 5*, 44–54. -

DE ALWIS, B. AND MURPHY, G. C. 2008. Answering conceptual queries with ferret. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*. 21–30.

DIT, B., REVELLE, M., GETHERS, M., AND POSHYVANYK, D. 2011. Feature location in source code: A taxonomy and survey. *J. Softw. Maint. Evolut. Res. Pract. 25,* 1, 53–95.

ENSLEN, E., HILL, E., POLLOCK, L., AND VIJAY-SHANKER, K. 2009. Mining source code to automatically split identifiers for software analysis. In *Proceedings of the 6th IEEE Working Conference on Mining Software Repositories (MSR'09)*. 71–80.

FRITZ, T. AND MURPHY, G. 2010. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*. 175–184.

GRANKA, L., JOACHIMS, T., AND GAY, G. 2004. Eye-tracking analysis of user behavior in www search. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM Press, New York.

GRECHANIK, M., FU, C., XIE, Q., MCMILLAN, C., POSHYVANYK, D., AND CUMBY, C. 2010. A search engine for finding highly relevant applications. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*. 475–484.

HENNINGER, S. 1996. Supporting the construction and evolution of component repositories. In *Proceedings of the 18th IEEE/ACM International Conference on Software Engineering (ICSE'96)*. 279–288.

HILL, E., POLLOCK, L., AND VIJAY-SHANKER, K. 2007. Exploring the neighborhood with dora to expedite software maintenance. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*. 14–23.

HILL, E., POLLOCK, L., AND VIJAY-SHANKER, K. 2009. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the 31st IEEE/ACM International Conference on Software Engineering (ICSE'09)*.

HILL, E., POLLOCK, L., AND VIJAY-SHANKER, K. 2011. Investigating how to effectively combine static concern location techniques. In *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*. ACM Press, New York.

HOLMES, R. AND MURPHY, G. C. 2005. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*. 117–125.

HOLMES, R., WALKER, R. J., AND MURPHY, G. C. 2006. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Softw. Engin. 32*, 952–970.

HUMMEL, O., JANJIC, W., AND ATKINSON, C. 2008. Code conjurer: Pulling reusable software out of thin air. *IEEE Softw. 25,* 5, 45–52.

INOUE, K., YOKOMORI, R., FUJIWARA, H., YAMAMOTO, T., MATSUSHITA, M., AND KUSUMOTO, S. 2003. Component rank: Relative significance rank for software component search. In *Proceedings of the 25th IEEE International Conference on Software Engineering (ICSE'03)*. 14–24.

Inoue, K., Yokomori, R., Yamamoto, T., Matsushita, M., and Kusumoto, S. 2005. Ranking significance of software components based on use relations. *IEEE Trans. Softw. Engin. 31*, 213–225.

Krueger, C. W. 1992. Software reuse. *ACM Comput. Surv. 24*, 131–183.

Landi, W. 1992. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst. 1*, 323–337.

Langville, A. and Meyer, C. 2006. *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press.

Lawrance, J., Bellamy, R., and Burnett, M. 2007. Scents in programs: Does information foraging theory apply to program maintenance? In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'07)*. 15–22.

Lawrance, J., Bogart, C., Burnett, M., Bellamy, R., Rector, K., and Fleming, S. 2010a. How programmers debug, revisited: An information foraging theory perspective. *IEEE Trans. Softw. Engin. 39,* 2, 197–215.

Lawrance, J., Burnett, M., Bellamy, R., Bogart, C., and Swart, C. 2010b. Reactive information foraging for evolving goals. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems*. ACM Press, New York, 25–34.

Little, G. and Miller, R. C. 2007. Keyword programming in java. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*. 84–93.

Little, G. and Miller, R. C. 2008. Keyword programming in java. *J. Autom. Softw. Engin. 16*, 37–71.

Mandelin, D., Xu, L., Bodík, R., and Kimelman, D. 2005. Jungloid mining: Helping to navigate the api jungle. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. 48–61.

Manning, C. D., Raghavan, P., and Schütze, H. 2008. *Introduction to Information Retrieval*. Cambridge University Press.

Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J. 2004. An information retrieval approach to concept location in source code. In *Proceedings of the 11th IEEE Working Conference on Reverse Engineering (WCRE'04)*. 214–223.

McMillan, C., Grechanik, M., Poshyvanyk, D., Fu, C., and Xie, Q. 2011a. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Trans. Softw. Engin. 38,* 5, 1069–1087.

McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., and Fu, C. 2011b. Portfolio: Finding relevant functions and their usages. In *Proceedings of the 33rd IEEE/ACM International Conference on Software Engineering (ICSE'11)*. 111–120.

Milanova, A., Rountev, A., and Ryder, B. 2004. Precise call graphs for c programs with function pointers. *Autom. Softw. Engin. 11*, 1, 7–26.

Murphy, G., Notkin, D., Griswold, W., and Lan, E. 1998. An empirical study of static call graph extractors. *ACM Trans. Softw. Engin. Method. 7*, 158–191.

Poshyvanyk, D., Gethers, M., and Marcus, A. 2012. Concept location using formal concept analysis and information retrieval. *ACM Trans. Softw. Engin. Method. 21,* 4,

Poshyvanyk, D., Guéhéneuc, Y. G., Marcus, A., Antoniol, G., and Rajlich, V. 2007. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Softw. Engin. 33*, 420–432.

Puppin, D. and Silvestri, F. 2006. The social network of java classes. In *Proceedings of the ACM Symposium on Applied Computing (SAC'06)*. 1409–1413.

Reiss, S. 2009. Semantics-based code search. In *Proceedings of the 31st IEEE/ACM International Conference on Software Engineering (ICSE'09)*. 243–253.

Revelle, M., Dit, B., and Poshyvanyk, D. 2010. Using data fusion and web mining to support feature location in software. In *Proceedings of the 18th IEEE International Conference on Program Comprehension (ICPC'10)*. 14–23.

Robillard, M. 2005. Automatic generation of suggestions for program investigation. In *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 11–20.

Robillard, M. P. 2008. Topology analysis of software dependencies. *ACM Trans. Softw. Engin. Methodol. 17*, 1–36.

Robillard, M. P., Coelho, W., and Murphy, G. C. 2004. How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Engin. 30*, 889–903.

Robillard, M. P. and Murphy, G. C. 2007. Representing concerns in source code. *ACM Trans. Softw. Engin. Methodol. 16,* 1.

Robillard, M. P., Shepherd, D., Hill, E., Vijay-Shanker, K., and Pollock, L. 2007. *An Empirical Study of the Concept Assignment Problem.* McGill University, Montreal, Quebec.

SAHAVECHAPHAN, N. AND CLAYPOOL, K. 2006. XSnippet: Mining for sample code. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*. 413–430.

SAUL, M. Z., FILKOV, V., DEVANBU, P., AND BIRD, C. 2007. Recommending random walks. In *Proceedings of the 11th European Software Engineering Conference Held Jointly with 15th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'07)*. 15–24.

SILLITO, J., MURPHY, G. C., AND DE VOLDER, K. 2008. Asking and answering questions during a programming change task. *IEEE Trans. Softw. Engin. 34*, 434–451.

SIM, S., UMARJI, M., RATANOTAYANON, S., AND LOPES, C. 2011. How well do search engines support code retrieval on the web? *ACM Trans. Softw. Engin. Methodol. 21*.

SIM, S. E., CLARKE, C. L. A., AND HOLT, R. C. 1998. Archetypal source code searches: A survey of software developers and maintainers. In *Proceedings of the 6th International Workshop on Program Comprehension (IWPC'98)*. 180–187.

SIRKIN, R. 2006. *Statistics for the Social Sciences*. Sage.

SMUCKER, M., ALLAN, J., AND CARTERETTE, B. 2007. A comparison of statistical significance tests for information retrieval evaluation. In *Proceedings of the 16th ACM Conference on Conference on Information and Knowledge Management*.

STARKE, J., LUCE, C., AND SILLITO, J. 2009. Searching and skimming: An exploratory study. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM'09)*.

STYLOS, J. AND MYERS, B. A. 2006. Mica: A web-search tool for finding api components and examples. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. 195–202.

TANSALARAK, N. AND CLAYPOOL, K. 2005. Finding a needle in the haystack: a technique for ranking matches between components. In *Proceedings of the International Symposium on Component-Based Software Engineering*.

WITTEN, I., MOFFAT, A., AND BELL, T. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Fransisco.

YE, Y. AND FISCHER, G. 2002. Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE'02)*. 513–523.

YE, Y. AND FISCHER, G. 2005. Reuse-conducive development environments. *J. Autom. Softw. Engin. 12*, 199–235.

ZAIDMAN, A. AND DEMEYER, S. 2008. Automatic identification of key classes in a software system using web-mining techniques. *J. Softw. Maint. Evolut. Res. Pract. 20*, 387–417.

Received July 2011; revised May 2012; accepted July 2012