

Recommending Random Walks

Zachary M. Saul
saul@cs.ucdavis.edu

Vladimir Filkov
filkov@cs.ucdavis.edu

Premkumar Devanbu
devanbu@cs.ucdavis.edu

Christian Bird
bird@cs.ucdavis.edu

Dept. of Computer Science
University of California, Davis
Davis, CA 95616

ABSTRACT

We improve on previous *recommender systems* by taking advantage of the layered structure of software. We use a *random-walk approach*, mimicking the more focused behavior of a developer, who browses the caller-callee links in the callgraph of a large program, seeking routines that are likely to be related to a function of interest. Inspired by Kleinberg’s work[10], we approximate the steady-state of an infinite random walk on a subset of a callgraph in order to rank the functions by their steady-state probabilities. Surprisingly, this *purely structural approach* works quite well. Our approach, like that of Robillard’s “Suade” algorithm[15], and earlier data mining approaches [13] relies solely on the *always* available current state of the code, rather than other sources such as comments, documentation or revision information. Using the Apache API documentation as an oracle, we perform a quantitative evaluation of our method, finding that our algorithm dramatically improves upon Suade in this setting. We also find that the performance of traditional data mining approaches is complementary to ours; this leads naturally to an evidence-based combination of the two, which shows excellent performance on this task.

Categories and Subject Descriptors: D.2.7 [Distribution, Maintenance and Enhancement]: Documentation

General Terms: Design, Documentation

Keywords: recommender systems, graph theory

1. INTRODUCTION

Software Maintainers spend a lot of time trying to understand the software under maintenance. This problem is especially acute in large software systems [4]. Even well-designed large systems impose steep learning curves on de-

⁰This material is based upon work supported by the National Science Foundation under Grant No. 0613949 (NSF SOD-TEAM) Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

velopers. Over the years, tool builders have sought different approaches to ease this learning task. We are particularly interested here in large, complex, specialized, *application programmer interfaces* (APIs), which constitute the basic substrate, or platform, upon which large software systems are typically built. Carefully architected, long-lived, expensive systems have extensive collections of APIs, which provide common services specialized for application needs, such as storage, synchronization, communication, security and concurrency. While a developer may be conceptually aware of these services, she still has to learn how a particular service is manifest in a specific large system. Documentation for such APIs may be lacking, and more experienced developers may be too busy to answer questions. In the Apache HTTPD web server, for example, there over 300 distinct portability layer API functions that form a “virtual machine” layer to simplify portability. We are concerned with a common discovery task: *given a particular function, find related functions*. For example, a developer may have found the function `apr_file_seek`, guessed that it is associated with file operations and wish to find other related functions. In the absence of documentation, the programmer would have to explore the source code directly, seeking other functions invoked along with `apr_file_seek` that may be related. Our goal is to automatically find and recommend the related API calls to a given call.

This problem of mining related API calls has attracted a lot of interest, and a variety of different approaches have been reported [12, 13, 15, 18, 19]. We now summarize the contributions of this paper.

Task Setting: Given a function, we find the other related functions. We do this using *exclusively the structural information in the call graph*; we don’t use version histories, name similarities or natural language (*e.g.*, API documentation) mining. This is a clear advantage; structural information is always available (and reliable) if the source code is available, but other types of information may not always be available and/or reliable.

New random-walk algorithm: We introduce a fast, simple, accurate algorithm, called FRAN (Finding with RANDOM walks) that is based on the steady state of a random walk on the callgraph neighborhood (inspired by [10]). This approach conceptually generalizes the previous purely structural approach proposed in Robillard’s Suade [15]. The algorithm works by considering a larger set of related items compared to previous algorithms (often too large to explore

manually), but then ranks them using the random-walk algorithm. The larger set increases our likelihood of finding relevant functions; our ranking algorithm increases our ability to narrow in on the most relevant functions in this larger group.

Evaluation: We evaluate this approach on the Apache (C-language) project source code; fortunately, Apache has a large number of well-documented portability layer APIs that can be used for testing. Finding that FRAN returns more answers in more cases, we first conduct case studies to examine whether the greater number of answers returned by our algorithm are relevant. Next, using the Apache documentation as a yardstick to judge correctness of relevant API calls returned by our purely structural approach, we pursue a quantitative comparison of FRAN with a standard mining algorithm and with our own re-implementation of Robillard’s Suade, based faithfully on the description in the published paper. We show first that our algorithm’s ranking, in a significant number of cases, is statistically better than a naive approach which simply returns the random set of nodes from the callgraph-neighborhood of the original call. This indicates that our approach effectively narrows the scope of code that a developer must browse to find related API calls; next, we also show that our approach substantially outperforms both Suade in most cases and (less dramatically, but still in a majority of cases) the traditional mining approach on the traditional F1 measure. Finally, we empirically determine when the mining approach beats FRAN, and present and evaluate an evidence-based combination of the two approaches.

2. MOTIVATION

Programming tasks in large systems are usually contextual. Given any artifact or task, a programmer often needs to know the related tasks or artifacts. Given a specific task, or artifact, a recommender system can suggest related items for the programmer’s consideration. The intent is that the recommendations provided will a) save the programmer time b) ensure that related items are considered, thus potentially avoiding defects c) over time, serve as an educational tool, assisting in learning. There has been quite a large body of work on this topic, which we discuss further below. First, we motivate the problem.

Consider a new developer working on the Apache system, who is writing some multi-threaded code in the server. In her design, she has created a thread, along with memory pool resources for use by the thread and used the thread in her logic. Now she comes to a point in the code where she finds it necessary to kill off a thread. She finds after some searching that the method that will kill a thread is `apr_thread_exit`. She puts the call in. Just to be on the safe side, she views the recommendations from a recommender tool to examine what the related calls are. In this case, the top 5 recommendations returned by our algorithm, FRAN, are: `apr_pool_destroy`, `apr_pool_clear`, `destroy_and_exit_process`, `start_connect` and `clean_child_exit`. Noticing that the destruction of the memory pool was so prominent, she looks at the `apr_pool` functions, and after examining the code, she realizes that the `apr_thread_exit` call has the side effect of de-allocating and clearing out pool memory. She realizes that she had naively made an incorrect assumption that the pool data could be shared with other threads and goes about fixing her code. This example is far

from contrived; the relationship between threads and memory pools in Apache is not trivial and has been the subject of debate.¹

In Apache, as in many large systems, there are a great many such internal library calls, and it is quite a challenge for a newcomer to master their proper use. Apache itself now has good documentation and is reasonably well commented. Using these documents and comments as textual cues, it may, with some effort, be possible to find related code.

Sadly, many large systems do not have good documentation, nor do they have good comments in the code. Even if they did have comments and documents, these may not necessarily reflect the reality of the code, since comments and code often get out of sync. However, any system that has code has internal structure, reflecting dependence between modules; this structure can be mined from the code. This internal structure in a way constitutes an implicit “documentation” of the relationships that exist between the parts of the system. When the code of one module invokes another module, it is an explicit indication of a relevance. If two modules are invoked by the same module, clearly they are related. If two different modules invoke the same module, they are related as well. Some of these relationships may be accidental or uninteresting; however, considered as a whole, the “neighborhood” of a function f in a callgraph has a lot of implicit (but reliable) information about the relevance of various functions in that neighborhood to f .

Our approach, similar to that of [13, 15], is based on the assumption that the dependence structure of software is a reliable, useful way to find related modules. Furthermore, it is always “real” and “code-related” in a way that comments and documents cannot claim to be. Our algorithm essentially assigns an equal probability of the “initial relevance likelihood” to each link, and (mathematically, using linear algebra) does an infinite random walk of an immediate (closed) neighborhood of a given function in the callgraph; the stationary probabilities of this infinite random walk indicate relevance scores.

3. RELATED WORK

Finding related code in systems is a long-standing problem that has attracted a great deal of attention. Space limitations preclude a comprehensive description; we only cover some representative papers here.

Data Mining approaches use a collection of co-occurrences of items (*e.g.*, items in shopping baskets) to find when certain items tend to co-occur, and these *frequent itemsets* are inferred to be strongly associated. Michail [13] proposed the use of this approach to find related calls in reusable libraries. Precision/recall results were not reported. Xie [19] proposes a similar approach, based on sequence mining, using knowledge of the order in which calls occur; he also reports promising, qualitative results. Some research has focused on code changes, judging that method calls added together are strongly related. Other researchers have mined version histories [23, 18]. Ying *et al.* [20] look at co-changes of files as an indication of close ties between files and use this for recommendation. A quantitative evaluation is provided. These approaches work well, but rely on version histories, which may not always be available, or sufficiently rich in a way that is relevant to a given query; thus, if certain func-

¹ See <http://marc.info/?l=apr-dev&m=99525021009667>

tions are not changed frequently enough and/or not strongly associated, then there may be insufficient data to provide good answers. Our approach can work with a single version and provide good results.

Some researchers have used execution trace collections [1, 3] to find patterns of usage. This requires a complete set of test cases and run-time tracing; the results will be useful only if sufficient data on the execution of all the different routines in the different possible ways are available. Static approaches don't require this.

There is another line of work [1, 6, 11] that uses frequent co-occurrence mining to find defects in software. The goal is to mine patterns of calls that *must* occur together in the same procedure or in the same execution trace; when such calls don't occur, it's a heuristic indication of a defect. In contrast, our goal is to find closely related functions whether or not they are called from the same function and without any tracing information.

Concept Location approaches aim to find code modules relevant to a particular concept (*e.g.* "bookmark") in a large source base. These approaches use information from a variety of sources: email messages, cvs logs, bug databases, static analysis, dynamic traces, information retrieval techniques, *etc.* A good survey is available in [17]. Some recent approaches use sophisticated information retrieval methods such as latent semantic indexing (LSI). LSI is used on the source code to find relevant targets. Sometimes, different kinds of information are combined. Sinafl [22] uses a combination of keyword-based retrieval and a special type of callgraph, which preserves control-flow dependencies, to find relevant code. Poshyvanyk *et al* [14] use a combination of information retrieval and function tracing. A related system is Hipikat [5], which attempts to find artifacts related to a specific project artifact. The Hipikat evaluation reports its performance on helping programmers fix bugs. In contrast to this, we are interested in finding functions related to a specific function rather than ones related to a specific feature or a reported bug.

Structural Approaches There are variety of approaches that make holistic use of the structure of the callgraph to extract useful information. Zhang & Jacobsen [21] use a random-walk algorithm to find cross-cutting code (aspects) in Java programs. They use a variant of Google's pagerank algorithm on the entire dependency tree to identify potential aspects. Inoue *et al* [9] describe how *component rank*, a version of pagerank on software dependency graphs, can be used as a "true measure of reuse" to identify valuable components.

Robillard's ACM SIGSOFT distinguished paper on the Suade method [15], which inspired our own, is also structural. The Suade algorithm takes as input a query set of interest I , and returns a suggestion set S . Suade uses two notions: *specificity* and *reinforcement* to rank members of S from among the neighbors of the query set. A method m is a better candidate for the answer S if it is specific to the query set of methods I and is reinforced by it. Method m_s is specific to I if any method m_i in I that is called by m_s is called by few other elements except m_s , and also if m_s calls few other methods; m_r is reinforced by I if most of the methods called by m_r are in I . Robillard presents a fuzzy-set based algorithm for calculating such elements y which is quite efficient. This can be viewed in terms of random walks. Consider a random walker \mathcal{W} (*e.g.*, an in-

experienced (or drunk) programmer randomly browsing a program by traversing call graph edges). Suppose m_s is specific to I . Then, if \mathcal{W} were to, in his ignorance, randomly jump forward from m_s along a random callgraph edge, he would probably end up at a method in I ; if then he jumps backward, he would likely end up at m_s . Likewise, if m_r is reinforced by I , then if \mathcal{W} were to start at the method m in I , then jump forwards to a method called by m and then jump backwards, then he is more likely to end up at m_r than other methods that are not reinforced by I . In such cases, Suade considers m_s and m_r to be possible candidate inclusions into I .

We argue that this algorithm can be generalized by considering the steady state probabilities that a node would be reached after an *infinite random walk* by a naive programmer in a *larger* neighborhood. In other words, if we *infinitely* iterate over specificity and reinforcement of *more* relationships, we would get a better result. Interestingly, standard methods in linear algebra indicate that the probabilities will converge, and can be calculated quickly.

4. TECHNICAL APPROACH

4.1 The FRAN algorithm

The motivation for the FRAN algorithm comes from the observation that there are two distinct types of relevance information readily available in a callgraph. First, if a function, f , calls another, g , it indicates that the functionality of f is related to the functionality of g . Second, assuming some degree of layering, two functions are related if they are in the same layer with respect to their calls (*i.e.*, they call and/or are called by the same functions). Layered structure is quite common; when searching for code related to a target function f , a human programmer typically would make this assumption, and focus her search on the modules in the same layer as f .

Our algorithm consists of two phases, each taking advantage of one type of relevance information. First, based on the query function, FRAN limits the set of all the functions in the program to an enriched set of functions from the same layer as the query function. Next, FRAN ranks this result using an algorithm that calculates the relevance of each function based on the link structure (*i.e.*, which functions call which other functions) of the callgraph. This ranking allows the programmer to consider the most relevant functions first.

FRAN's first step identifies the set of functions in the same layer as the query function by finding two subsets of the callgraph, called the sibling set and the spouse set. First, define the *parent set* for a query function as the set of functions that call the query function, and define the *child set* as the set of functions that are called by the query function. Continuing the family tree metaphor, define the *sibling set* as the set of functions that are called by any function in the parent set and define the *spouse set* as the set of functions that call any function in the child set. Arguably, the functions in the sibling set and the spouse set together form a relevant set of functions in either the same or proximate functional layer.

Therefore, given a query function, FRAN first narrows its search to functions contained in the union of the sibling set, the parent set and the spouse set of the query function. The parent set is included because each function in this set calls the query function, indicating the first type of relevance.

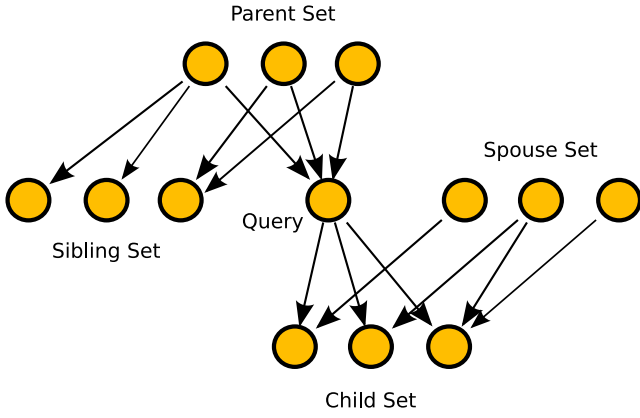


Figure 1: An illustration of the various relationships in the callgraph of the parent set, child set, sibling set and spouse set to the query function.

Inspired by the web search community, we call this union set the *base set*. We’re abusing the term somewhat; our base set is not the same set of nodes (relative to the query node) as the base set in Kleinberg’s well-known paper [10]. For us, the base set is an enriched set of results related to the query function for the programmer to consider. However, the base set, itself, is frequently too large for a human to easily explore. Therefore, it is desirable to rank the functions in the base set based on their relevance to the query.

In order to rank the results in the base set, we first observe that software contains functions that aggregate functionality and functions that largely implement functionality without aggregating, and we note that there is a circular relationship between these two types of functions. Aggregating functions call implementing functions and implementing functions are called by aggregators. A similar relationship exists in the context of world wide web pages where the aggregating pages are called hubs and the implementing pages are called authorities. The Hypertext Induced Topic Selection (HITS) algorithm due to Kleinberg takes advantage of this relationship between hubs and authorities to rank a set of web pages based on degree of authority. In FRAN, we use HITS on the subgraph of the callgraph induced by the base set to assign ranking scores to the nodes in the base set, allowing us to sort the elements of our base set. HITS applied to software callgraphs works as follows.

For a collection of functions in a callgraph assign each function, f , an authority score, $x^{<f>}$, and a hub score, $y^{<f>}$. As noted above, strong hubs (aggregators) call many strong authorities (implementors). To capture this relationship, define two operations \mathcal{I} and \mathcal{O} . \mathcal{I} updates the authority weights based on the hub weights, and \mathcal{O} updates the hub weights based on the authority weights.

$$\mathcal{I} : x^{<f>} = \sum_{\{g|g \text{ calls } f\}} y^{<g>} \quad (1)$$

$$\mathcal{O} : y^{<f>} = \sum_{\{g|f \text{ calls } g\}} x^{<g>} \quad (2)$$

In HITS, these two rules are applied one after the other iteratively:

Algorithm 4.1: HITS ALGORITHM()

```

repeat
   $x_{i+1} \leftarrow \mathcal{I}(y_i)$ , updating the authority scores.
   $y_{i+1} \leftarrow \mathcal{O}(x_{i+1})$ , updating the hub scores.
  Normalize  $x_{i+1}$  and  $y_{i+1}$ 
until  $x_i - x_{i+1} <$  a stopping threshold.

```

Let A be the adjacency matrix of the graph in question. If there exist two functions represented by numerical ids f and g , and if f calls g , then the $(f, g)^{th}$ entry of A is 1; every other entry of A is 0. Kleinberg gives a proof that that the sequence $\lim x_i \rightarrow x^*$ and $\lim y_i \rightarrow y^*$ where x^* is the principal eigenvector of $A^T A$ and y^* is the principal eigenvector of AA^T [10]. Therefore, the HITS algorithm converges and could, in fact, be implemented using any standard eigenvector finding method.

This convergence result has an interesting interpretation in the context of Markov chains. The matrices $A^T A$ and AA^T can be thought of as reachability matrices. The matrix $A^T A$ has a 1 in position (i, j) if j is in the sibling set of i . Another way of saying this is that the i^{th} row of $A^T A$ indicates all of the functions reachable from i by traversing back a function call to a parent, and then, traversing forward to a sibling. Similarly, the i^{th} row in AA^T gives the spouse set of i .

If the rows of $A^T A$ and AA^T are normalized to sum to 1, they can be thought of as transition matrices for Markov chains describing the actions of two programmers randomly attempting to understand a program, and the eigenvectors of these matrices indicate the steady-state probabilities of the Markov chains. Thus, the authority score of a function represents the probability that the random programmer who always investigates sibling functions will end up in the function, and the hub score is the probability that the random programmer who only considers spouse functions will end up in that function. The FRAN algorithm simply returns the top n authorities, where n is selected by the user.

FRAN performs better in this setting than Suade; the Suade algorithm only returns results that are adjacent to the query function in the graph. However, if more data are available, Suade *does* allow graphs other than the callgraph to be used as the basis for the search (e.g. the “member referenced by” graph). In the callgraph, only functions called by or called from the query function are returned. Hence, Suade only finds the results that a programmer might quickly find using “grep”, and in addition it only finds results in the layers above and below the query function rather than finding the most relevant results, which lie *in* that layer.

It is also important to note that FRAN is fast. The implementation for this paper returns query results interactively with no perceptible wait time.

4.2 The FRIAR algorithm

Our second algorithm, Frequent Itemset Automated Recommender (FRIAR) is inspired by the data mining practice Association Rule Mining. Association Rule Mining was developed to analyze purchasing patterns [7]. Define a transaction as a set of items purchased together. Then, given a set of transactions, association rule mining attempts to discover rules of the form $A \implies B$, where A and B are small

sets of items, and the \implies relation indicates that if A is seen in a transaction, then B will also (often) be seen.

A problem related to finding association rules is to list the frequent itemsets. A *frequent itemset* is a set of items that appears in at least s transactions for some threshold, s , and the *support* of a frequent itemset, F , is defined as the fraction of transactions in which F appears.

In FRIAR, we used sets of functions that were commonly called together to predict functions related to a particular query function. To do this, we defined a transaction as the set of functions called by a particular function, and then created a transaction for every function in Apache that calls at least one function. We found 1919 functions that made at least one function call; therefore, we had 1919 transactions.

Using these transactions and the `arules` package [7] in R ,² we found all 56,022 itemsets that have a support of at least 0.001. We found these itemsets so that we could use them as a representation of which functions are called together in the Apache callgraph. Typically in data mining, much higher support thresholds are used. This is because a data miner is interested in statistically significant data trends. However, in related function finding, it is not the trend we are looking for, but, rather, a searchable representation of the “called with” relationship. Therefore, in order to capture most of the instances of this relationship, we use a very low support value, requiring only $2(\lceil 1.919 \rceil)$ out of 1919 transactions contain an instance of a itemset.

To make a query for a particular function, we searched for all of the itemsets that contained that function, returning the union of all these itemsets as the result set. Then, we assigned each result function a score based on the maximum observed support value associated with that function, and we ranked the result set using these scores. For example, if `apr_file_open` was seen in 3 itemsets which had support values 0.1, 0.3 and 0.2, the value 0.3 would be used to rank `apr_file_open`.

4.3 Data Extraction

For the evaluation of our approach, we used a callgraph of the Apache web server. We have downloaded the entire source code repository for the 2.0 version of Apache (`httpd-2.0`). For our analysis, we checked out the source code from October 1st, 2003. The motivation for using this version is that it is near the middle of the life of `httpd-2.0` and thus is fairly mature and stable, but at the time was undergoing continued rapid development. In order to build the web server, we also checked out matching versions of supporting libraries such as the Apache Portable Runtime (`apr`) from other modules within the repository and built the versions of tools (e.g. `gcc`, `as`, `ld`) that were used at the time. The callgraph for this version of the web server source code was generated by using CodeSurfer, a commercial source code analysis and inspection tool from GrammaTech.³ Because CodeSurfer links with `gcc` at build time and accesses its internal symbol tables and data structures, we’re very confident of it’s results. One of the benefits of this tool is that it uses points-to analysis to determine what functions are called indirectly through function pointers. The result of this analysis is a labeled, directed graph, with nodes rep-

resenting functions and edges representing calls from one function to another. The functions in the graph represent all functions calls (including those to `stdlib` such as `strlen`, `printf`, etc.) and are not limited to just those defined within the Apache code base.

5. EVALUATION

Papers on recommender systems that find functions related to a particular function, in the past, have generally used case studies for evaluation. By contrast, systems that recommend files to be changed to fix a reported bug [5] or files whose changes are strongly associated historically with a given file [20, 23] have been evaluated quantitatively using historical data. We seek here to evaluate recommenders that retrieve functions strongly associated with a given function; case studies are *de rigueur* in this setting.

The final arbiter of whether a recommendation is relevant is a human focused on a specific task. As a result, most influential prior papers on systems that recommend related functions have focused on case studies, or small-scale human subject studies, as in Robillard’s recent ACM SIGSOFT Distinguished paper [16] and other similar works [13, 19]. While this type of evaluation is quite useful, there are limitations. First, it is very difficult to scale human experiments to get quantitative, significant measures of usefulness; this type of large-scale human study is very rare. Second, comparing different recommenders using human evaluators would involve carefully designed, time-consuming experiments; this is also extremely rare. Finally controlling for the factors that determine which algorithm performs better would be harder still, since more experimental data would be needed to get sufficient variance in the predictive factors. Quantitative approaches, on the the other hand, *can* allow tests of significance, comparison of performance and determining predictive factors. However, to use quantitative methods, on a statistically significant scale, we need a sufficiently large and varied test example, *and* an oracle to decide which answers are correct. In practice, this is very difficult come by, which perhaps accounts for the rarity of quantitative evaluation. In our work, we have found a specific task where results can be evaluated quantitatively.

For our evaluation, we use the 330 functions that constitute the Apache portability layer. Each of these functions is given as a query to FRAN, FRIAR and Suade, and we simply count the number of answers to see which algorithm gives more answers. Generally speaking, by this metric we found that FRAN outperformed both Suade and FRIAR. Thus, in 239 cases, FRAN retrieves more answers than Suade; in 64 cases, it retrieves exactly the same number; and in 27 cases, Suade retrieves more. When we compared FRIAR and FRAN, FRAN retrieved more answers 304 times; the two methods tied 24 times; and FRIAR retrieved more answers 3 times. However, this is a very crude comparison. Given that FRAN retrieves more answers in a majority of the cases, we should like to know, are these answers relevant, and how often are they relevant? To do this evaluation, we take a two-pronged approach, first using a case study approach to evaluate a number of retrieved answers, by hand. We also then follow with a quantitative approach evaluating a large number of queries on a specific task. The quantitative part of our approach focuses on a very specific, targeted requirement of a recommender: given a function, find the most closely related functions. We were able to do a

² <http://www.r-project.org>

³ See <http://www.grammatech.com/products/codesurfer/overview.html>. We are grateful to GrammaTech for the use of this tool.

fairly thorough quantitative comparison of performance, for this specific task. For the case study part, we focus on a few cases where FRAN retrieves more answers than the other algorithms and examine whether these recommendations are relevant to the given functions.

5.1 Case Study

Given the large number of cases where FRAN retrieves more ersatz related functions than FRIAR, our goal in the case study is to determine a) Are these extra functions really related, or are they just random junk? b) Why does FRAN retrieve more answers? So we focus on cases where where FRAN retrieves more answers than both Suade and FRIAR and critically review the answers. Our cases are all drawn from the Apache portability layer. In each case, we issued the query to all 3 algorithms and examined the retrieved set. The answers were examined by 3 of us (Bird, Devanbu, and Saul). Bird and Devanbu have over the last two years conducted extensive mining and hypotheses testing on the Apache repository. Saul has worked professionally as a Unix systems programmer and has prior experience constructing a web server, so he is familiar with the design of web servers. With our prior experience and careful reading of the Apache documentation, we were able to critically evaluate the answers returned.

Case 1: `apr_collapse_spaces` is a function in the Apache Portability layer, in the String Routines module.⁴ The APR documentation states that the function will “Strip spaces from a string.” For this function, FRIAR returns nothing; Suade returns a function that is called from within the body of a standard C library: this function is normally invisible to C programmers, and is of little value to a developer. FRAN returns several recommendations, ranked by authority scores. Of the top 5, the first is `ap_cfg_getline`, which is described⁵ as “Reads a line from a file, stripping whitespace.” The second is a simple function `read_quoted`, which (from the Apache source code) is readily determined to remove quotes from strings. The rest of the top 5 are irrelevant. Further analysis reveals the reason that FRAN outperforms the other methods is that FRAN includes the spouse set in the set of potential results. The function from the body of the C library is called by `apr_collapse_spaces`, but it is also called by `ap_cfg_getline` and `read_quoted`. Suade and FRIAR do not consider the spouse set so they miss this relationship.

Case 2: `apr_socket_listen` is a function which is described in the documentation⁶ as “Listen to a bound socket for connections.” When this function is given as a query, the top-ranked responses are `apr_socket_opt_set` which is used to set up options for a socket, `apr_socket_close` (closes a socket) and `apr_socket_bind` (binds a socket to an associated port). All of the above are clearly relevant. The fifth one, `ap_log_perror` is an error logging routine that is called often. For this query, Suade retrieves `listen` which is a Unix system call described as “Listen for connections on a call” in the BSD System Calls manual listing (See `listen(2)`). This function is relevant, but clearly belongs in a lower layer of abstraction, and would normally be abstracted

⁴See http://docx.itscales.com/group__apr__strings.html

⁵See http://httpd.apache.org/dev/apidoc/apidoc_ap_cfg_getline.html

⁶See http://docx.itscales.com/group__apr__network_io.html

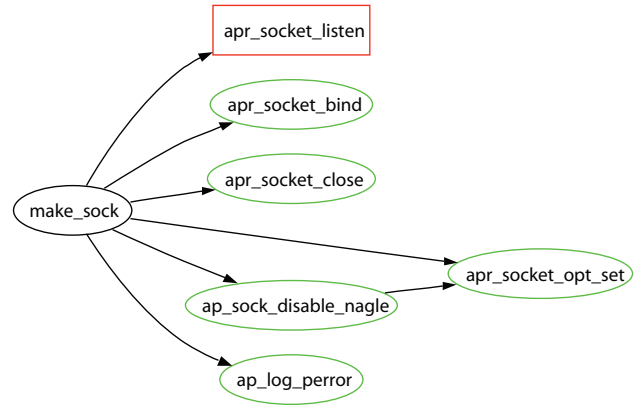


Figure 2: The neighborhood of `apr_socket_listen` with the sibling set highlighted in green.

by the Apache Portability Layer to `apr_socket_listen`; programmers would normally shun functions in the lower layer to avoid platform-specificity. Suade also retrieves the function `make_sock`, which sets up the server’s socket to listen for connections; clearly, this function is also relevant. Interestingly, comments on this code state that this routine is “begging and screaming to be in the portability layer.” This function is also retrieved by FRAN, but it is ranked number 7, after another function `ap_sock_disable_nagle` which is a function to disable a particular network buffering algorithm. FRIAR retrieves nothing in this case. As can be seen in figure 2, the functions returned by FRAN are all siblings, and thus in the same layer. Suade only considers the direct neighbors of a function as possible related functions; since `apr_socket_listen` has only one parent, `make_sock` it finds nothing else but that.

Case 3: `apr_pool_terminate`. Like many large systems, Apache uses a pool-based memory management.⁷ It allocates memory in big hunks, called pools, and uses internal functions to allocate smaller pieces from this pool. `apr_pool_terminate` is a function which will “Tear down all of the internal structures required to use pools.” The first two retrieved functions are `apr_pool_destroy`, which destroys the pool and frees the memory, and `apr_pool_clear`, which clears the memory without de-allocating it. The next function is `destroy_and_exit_process` which shuts down the server. This function actually calls `apr_terminate` (#5 on the retrieved list) to tear down the memory pools. The fourth on this list is `start_connect` which has no obvious relevance. Suade retrieves `apr_terminate` and `apr_pool_destroy`, which are both relevant, but does not retrieve the others. FRIAR retrieves none.

Case 4: `apr_file_eof` is one of the Apache File/IO handling routines.⁸ It checks “Are we at the end of the file.” FRAN retrieves only 3 functions. The first is `apr_file_read`, which is clearly related; it also retrieves `ap_rputs` which⁹ “Output(s) a string for the current request” and `do_emit_plain` which¹⁰

⁷http://apr.apache.org/docs/apr/0.9/group__apr__pools.html

⁸http://docx.itscales.com/group__apr__file_io.html

⁹http://docx.itscales.com/group__a_p_a_c_h_e__c_o_r_e__p_r_o_t_o.html

¹⁰From comments in `mod_autoindex.c` in Apache HTTPD source code

“emit(s) a plain file.” Analysis of the callgraph reveals that `do_emit_plain` calls the other three. We judged all of these are quite relevant, since they are useful when reading from file and writing the contents back to the request. Suede retrieves only one item: the parent function, `do_emit_plain`. As FRIAR looks at frequent itemsets that occur at least twice in the code, it won’t ever see `apr_file_eof` because this function is called only once.

Admittedly, these are a limited set of samples; our focus was also on the cases where FRAN retrieves more items, to check if these extra items were indeed relevant, and to explain the differences in behaviors between the algorithms. As discussed earlier, it is laborious and time-consuming to conduct comparative case studies on a large scale. Therefore, we devote the rest of our evaluation to a more automated, comprehensive quantitative evaluation of the performance of these algorithms on a more specific (but still very core) task of recommender algorithms, as we now describe.

5.2 Quantitative Study

To perform this quantitative evaluation, we focus on a specific task in the Apache system. Apache has a *portability layer* (PL), which consists of 32 separate groupings of functions, or portability layer modules (PLM). Each PLM includes a closely related set of functions that perform tasks such as file operations, socket operations, thread management, locking operations and memory pool management. When a programmer is working on a method that performs a particular operation on the memory pool *e.g.*, she may seek related functions that operate on the memory pool.

In Apache, the directory structure naming conventions of the PL generally allow this task to be done just using file structuring and “grep”. However, this type of a-priori grouping and documentation may not be available to programmers, and, worse, potentially misleading exceptions to established naming conventions can exist. For example, most of the function names in the Apache “File I/O” PLM are prefixed with the string “`apr_file`.” However, some functions from this PLM such as “`apr_temp_dir_get`” and “`apr_dir_make`” do not follow the naming convention.

However, because the extensive Apache PL documentation groups the functions into PLMs, we can use these PLMs as a valuable oracle, quantitatively evaluating the performance of FRAN, FRIAR and Suede on one specific task:

Task: *Given a query function and a callgraph, retrieve other functions in the same PL module.*

It has been suggested that we could evaluate our algorithm’s performance when a set of query functions is given as input (rather than a single query function); however, due to space and time constraints, that task remains for future work.

Also, it can certainly be argued that there are other ways to do the current task in Apache than using FRAN, FRIAR and Suede; in response, we have three rejoinders a) Yes, but the type of extensive documentation available in Apache is rare in large systems b) Even when it exists, documentation is not always in sync with the code, which never lies! c) Finding related functions in the same API module *is* a task that most C programmers have to deal with, since the language is not object-oriented, and d) The fact that the documentation exists makes Apache a useful setting to perform a thorough, and we believe unprecedented, type of quantitative evaluation. For this portion of the study our goal is

	FRAN	FRIAR	Suede
Top-5	206	144	10
Top-10	232	131	9
Top-15	228	129	8

Table 1: Number of queries (out of 330) for which the top-k recommendations from each algorithm pass the 0.05 False Discovery Rate.

to *quantitatively* evaluate both the statistical significance of the results, as well as the recall and precision.

5.2.1 Significance of the Recommendation Sets

Our two algorithms FRAN and FRIAR, as well as Suede, recommend a number of related functions in ranked-order. Based on the interest and resources of the programmer she can choose the top-k of the ranked functions on which to perform her work. We call this top-k set the *recommendation set*.

To objectively quantify the ability of our algorithms to return significant recommendations we tested statistical hypotheses that their efficacy is no better than that of the behavior of a “programmer new to the project” (statistical null-hypothesis). Such a programmer would look at a large *candidate set* of potentially related functions, possibly as small as some well-defined neighborhood in the callgraph around the query function (although still potentially consisting of hundreds of functions), or in some cases even as big as the whole callgraph. Out of those candidate related functions the programmer would uniformly at random choose a smaller set of recommendations.

Using this null-hypothesis programmer model allows us to quantitatively evaluate each set of recommendations of our algorithms as possibly not doing better than chance recommendations. Rejecting the null-hypothesis at a certain significance threshold, say 0.05, would let us believe that the recommendation set would not be guessed by a “programmer new to the project” 95% of the time.

We compared the significances of the three algorithms at three *recommendation set size cutoffs*, top-5, top-10 and top-15, over all 330 query functions. We used the PL module documentation as a Rosetta stone (see above). For each recommendation cutoff k (5,10 and 15), we counted how many of the top-k recommendations for a given query function appear in that function’s PL module. The hypergeometric distribution was used to obtain the chance probability of observing the number of functions from a module within each query recommendation set. More specifically, the probability of observing at least x functions from a PL module within a recommendation set of size k is given by:

$$p = 1 - \sum_{i=0}^{x-1} \frac{\binom{f}{i} \binom{g-f}{k-i}}{\binom{g}{k}}$$

where f is the total number of functions within the PL module and g is the total number of functions in the initial large candidate set (i.e. the population). This is also known as the p-value and is equal to 1– the significance.

In order to assess the relative performance of the algorithms in retrieving related functions it is sufficient to choose a common population set for the functions, for all algorithms (corresponding to the initial candidate set.) In our case we

	$p = 0.05$	$p = 0.1$	$p = 0.2$
Top-5 (out of 272)	13	39	52
Top-10 (out of 228)	35	55	73
Top-15 (out of 209)	50	61	92

Table 2: Number of queries for which the top-k recommendations from FRAN pass a False Discovery Rate threshold.

chose the whole call graph, i.e. $g = 2308$. We submitted each of the 330 unique functions from the PL modules as queries to the algorithms and using the above significance formula counted how many of the recommended sets were significant at 95%, or have a p-value lower than 0.05. Given that each query is in fact a hypotheses, this testing procedure amounts to multiple testing of 330 hypothesis. To maintain an overall false positive rate of below 0.05 (i.e. the False Discovery Rate), the individual p-values during the testings were adjusted (lowered) using the Benjamini-Hochberg adjustment for multiple hypothesis testing [2]. The results of the relative comparison for all three algorithms are given in Table 1.

We also performed a statistical significance study to determine how well FRAN ranks the initial candidate functions. For the population set here we used the FRAN base set as described before in Sec. 4.1. The idea is to compare the performance of FRAN to that of a “programmer new to the project” (null-hypothesis) who is given the same base set of functions in the neighborhood of a given query function and asked to uniformly at random select the top-k as recommendations. Rejecting the null hypothesis at a certain threshold would give us confidence that FRAN’s performance is not based solely on the selection of the base set. We ran the study similarly to the previous one, except we counted how many FRAN recommendation sets had (multiple hypothesis testing adjusted) p-values lower than $p=0.05$, $p=0.1$ and $p=0.2$. The results are given in Table 2. Since not all queries returned recommendations, the total numbers of recommended sets are different from 330 and noted in the table. Clearly FRAN does more than choose randomly from the base sets, even at very stringent p-value of 0.05. The pattern of increasing number of recommendation sets below a given p-value with the increase of their size has been noted and is possibly interesting, but its further exploration was beyond the scope of this paper.

5.2.2 Recall/Precision Comparisons

We sought to compare the effectiveness of the three algorithms in retrieving related functions. In information retrieval, three measures of performance are used repeatedly: *precision*, *recall* and the *F1-measure*. All are defined over the set of documents retrieved by the algorithm, *Retrieved*, and the set of all relevant documents, *Relevant*. Precision is defined as $p = |Relevant \cap Retrieved|/|Retrieved|$ and recall as $r = |Relevant \cap Retrieved|/|Relevant|$. (Precision and recall are also defined in the area of classification theory are known as positive-predictive value and sensitivity, respectively).

The F1-measure (*F-measure*) is the equally-weighted harmonic mean of the recall and precision measures, defined as $F = 2pr/(p+r)$, and is often used as a combined measure of the recall and precision. Note that all three measures have a range of [0..1].

As in the significance study above, we used the functions’ partition into PL modules and compared the performance of the three algorithms at three recommendation set size cutoffs, top-5, top-10 and top-15. We used the F-measure as indicator of performance. To calculate it, for each query function we took the recommended set of functions returned by an algorithm as the *Retrieved* set and the set of functions in the query function’s PL module as the *Relevant* set.

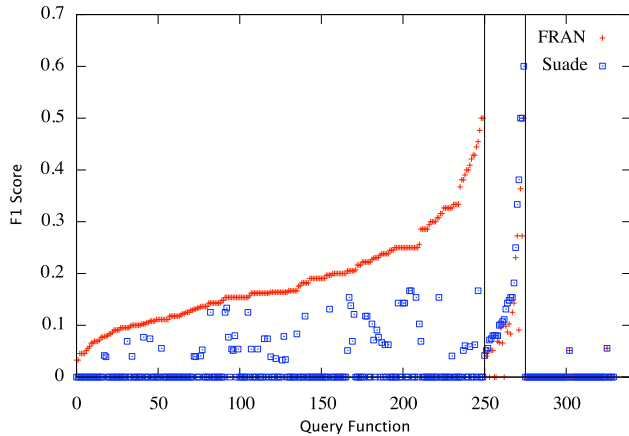
The results of comparing the FRAN and Suade algorithms at the top-10 cutoff are given in Fig. 3(a). The query functions on the x axis are sorted in increasing order of their F-measure values and grouped in three parts, the first when FRAN was favorable, second when Suade was favorable and third when they were tied. It is apparent that FRAN is favorable across the majority of recommendation sets and significantly so. In the small number of cases when Suade is favorable the difference is very small, with only a few (five) FRAN values scoring zeros. The F-measure score comparison for the top-5 and top-15 is omitted here to save space since the story is essentially the same.

We next compared the FRAN and FRIAR algorithms in the same way, and the results are given in Fig. 3(b). Here the situation is more interesting as there are many recommendation sets for which either one is better than the other and there are some for which the two algorithms are tied. A statistical test was performed, Wilcoxon-Mann-Whitney rank sum [8], and we could reject the hypothesis that FRAN is not favorable to FRIAR in more than 50% of comparisons in the top-10 and top-15 cases (p-val of 0.0034 and 0.00034 respectively), but not in the top-5 case (p-val of 0.83). Thus, statistically, FRAN more often than not does better than FRIAR.

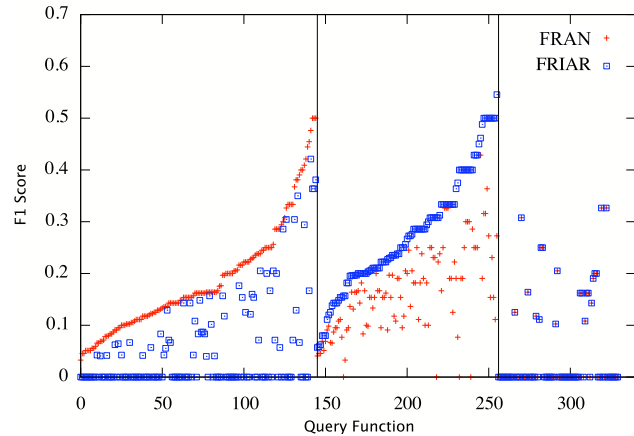
Factors influencing performance It is clear from the above results that the two favorable algorithms FRAN and FRIAR work in complementary way. We suspected that the reasons for the complementarity lie in the core assumptions, the selection of the base set for FRAN and the construction of the frequent item sets for FRIAR. Namely, the FRAN base set choice favors functions called by few other functions, while FRIAR discourages small item sets, especially in the extreme cases of 0 or 1 callers. Thus FRAN would do better for small base set sizes than FRIAR would, which would in turn do better for larger item sets. In trying to explain the cases when one of the algorithms betters the other and vice-versa, we looked at how the difference in the F-measure values (or advantage) between FRAN and FRIAR changes with the base set size of the query function in the call graph. The results Fig. 4(a) show that FRAN wins for small base set sizes while FRIAR wins for large base sets, in support of our complementarity hypothesis.

Combining Algorithms The quantitative approach we undertook in this project allows to both assess how well we are performing with respect to the original task, as well as obtain insight into the factors influencing that performance. Above we illustrated how one such hypothesis, the effect of base set size on FRAN’s performance, can be confirmed. Once there is empirical support for such insight, as well as intuitive or theoretical explanation for it, the algorithms can be bettered by including that knowledge in them.

Here we illustrate this point by combining the FRAN and FRIAR algorithms into an algorithm we call Combined Automated Recommender based on the base set size, or CARB. Given a query function and a call graph, this algorithms



(a) FRAN compared to Suade



(b) FRAN compared to FRIAR

Figure 3: Performance comparisons using the $F1$ measure and top-10 recommendation sets.

simply looks at the size of that function’s base set as generated by FRAN from the call graph, and if it is smaller than 45 (determined from Fig. 4(a) as the switch point) runs FRAN with that function and call graph as input, otherwise runs FRIAR. Fig. 4(b) gives the plot of the cumulative F-measure values for the three previous algorithms and CAR-B. It is apparent from the plot that both FRAN and FRIAR dominate Suade, but are complementary to each other. The advantage of CAR-B is obvious in that it benefits from and combines the advantages of both FRAN and FRIAR.

Threats to validity There are some potential threats to the validity of this study. First, our use of the Suade algorithm is restricted. The Suade algorithm is designed to accept a set of items as the input, but we only pass a set containing only a single item; this certainly helps account for the paucity of results returned by Suade. However, we argue that the problem of finding the set of functions related to a single function is just as important as the more general problem, especially to a developer new to the system, who will not be knowledgeable enough to build a good query set. Additionally, even with a larger query set Suade will still only select neighbors of the query set as potential results, ruling out the important sibling and spouse sets (which are not necessarily direct neighbors of the query nodes).

Also, the Suade algorithm can use several relations (not just the “calls” relation) to find related functions. However, we, assert that the problem of finding related functions given only the callgraph as input is important. There are numerous available tools to find a program’s callgraph, while other relations may be more difficult to obtain.

Finally, the task that we have quantitatively evaluated, that of finding related functions from a given API, is a problem of limited scope. The case studies that we’ve presented on the broader problem of finding any related functions are only qualitative examples. In response, we first note that in spite of its limited scope, the API recommending problem is a very important one. Additionally, we contend that evaluating this type of algorithm is a difficult problem and that we contribute a methodology and an oracle to evaluate it quantitatively. Realistically, further review of the results of our algorithms by outside experts on Apache would be ex-

pensive and time consuming, and beyond the scope of this paper. Such a study is reserved for future work.

6. CONCLUSIONS

In this paper, we propose an approach to finding related functions, based on a random-walk approach. We find that this general, and rather brute-force approach, is surprisingly effective for finding related functions. We evaluated this purely structural approach using the Apache project, whose well-documented APIs provide a convenient evaluation target. The approach generalizes to object-oriented programs, which have many other types of relationships suitable for random walking. We hope in the future to build this into Eclipse.

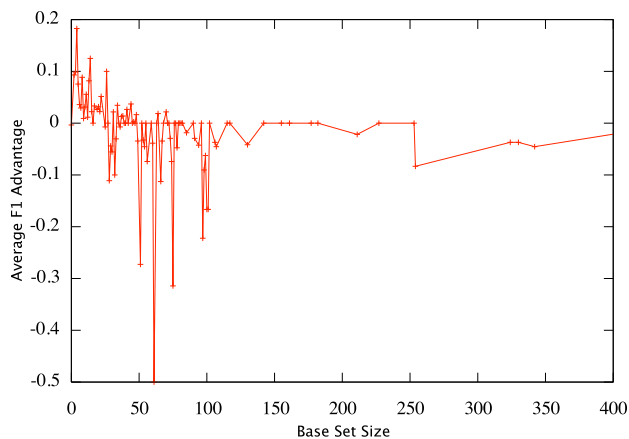
In conclusion, we encourage readers to download our implementations of the FRAN and FRIAR algorithms and try them. The source code for our tools, and also the Apache callgraph data set, are available from <http://macbeth.cs.ucdavis.edu/FRAN/>. Also, the Suade algorithm is available from <http://www.cs.mcgill.ca/~swevo/suade/>.

7. ACKNOWLEDGMENTS

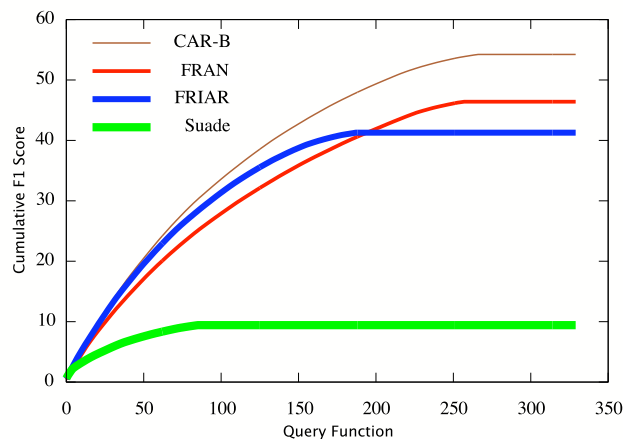
We thank the NSF Science of Design program for supporting this work. Further, we would like to thank the reviewers and Martin Robillard for their comments, but we wish to emphasize that we alone are responsible for the final content of this paper.

8. REFERENCES

- [1] G. Ammons, R. Bodík, and J. Larus. Mining specifications. *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 4–16, 2002.
- [2] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society B*, 57:289–300, 1995.
- [3] W. Cohen. Inductive specification recovery: Understanding software by learning from example



(a) FRAN has advantage over FRIAR for small base set sizes.



(b) A base-set size aware combined algorithm is favorable.

Figure 4: The effect of the base-set size and a cumulative comparison of the methods.

behaviors. *Automated Software Engineering*, 2(2):107–129, 1995.

[4] T. Corbi. Program Understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[5] D. Cubranic, G. Murphy, J. Singer, and K. Booth. Hipikat: a project memory for software development. *Software Engineering, IEEE Transactions on*, 31(6):446–465, 2005.

[6] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 57–72, 2001.

[7] Y. Hahsler, B. Grün, and K. Hornik. A computational environment for mining association rules and frequent item sets. *Journal of Statistical Software*, 14:1–25, 2005.

[8] M. Hollander and D. A. Wolfe. *Nonparametric Statistical Methods*. 2nd edition, 1999.

[9] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component rank: relative significance rank for software component search. *Proceedings of the 25th international conference on Software engineering*, pages 14–24, 2003.

[10] J. Kleinberg. Authoritative sources in a hyperlinked environment. In *Proceedings, 9th SIAM Symposium on Discrete Algorithms*, New York, NY, 1998. ACM, ACM.

[11] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, 2005.

[12] V. B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings of the 13th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-13)*, Sept. 2005.

[13] A. Michail. Data mining library reuse patterns using generalized association rules. *International Conference on Software Engineering*, pages 167–176, 2000.

[14] D. Poshyvanyk, Y. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification. *Proceedings of 14th IEEE International Conference on Program Comprehension (ICPC’06)*, Athens, Greece, pages 137–148, 2006.

[15] M. Robillard. Automatic generation of suggestions for program investigation. *ACM SIGSOFT Software Engineering Notes*, 30(5):11–20, 2005.

[16] M. P. Robillard. Automatic generation of suggestions for program investigation. In *SIGSOFT Symposium on the Foundations of Software Engineering*, 2005.

[17] N. Wilde, M. Buckellew, H. Page, V. Rajlich, and L. Pounds. A comparison of methods for locating features in legacy software. *Journal of Systems and Software*, 65(2):105–114, 2003.

[18] C. Williams and J. K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31, 2005.

[19] T. Xie and J. Pei. MAPO: mining API usages from open source repositories. *Proceedings of the 2006 international workshop on Mining software repositories*, pages 54–57, 2006.

[20] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[21] C. Zhang and H.-A. Jacobsen. Efficiently mining crosscutting concerns through random walks. In *AOSD ’07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 226–238, New York, NY, USA, 2007. ACM Press.

[22] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNI AFL: Towards a static noninteractive approach to feature location. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(2):195–226, 2006.

[23] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *Software Engineering, IEEE Transactions on*, 31(6):429–445, 2005.