Search in Source Code Based on Identifying Popular Fragments

Eduard Kuric and Mária Bieliková

Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Ilkovičova 3, 842 16 Bratislava, Slovakia
{kuric, bielik}@fiit.stuba.sk

Abstract. When programmers write new code, they are often interested in find-ing definitions of functions, existing, working fragments with the same or simi-lar functionality, and reusing as much of that code as possible. Short fragments that are often returned by search engines as results to user queries do not give enough information to help programmers determine how to reuse them. Under-standing code and determining how to use it, is a manual and time-consuming process. In general, programmers want to find initial points such as relevant functions. They want to easily understand how the functions are used and see the sequence of function invocations in order to understand how concepts are implemented. Our main goal is to enable programmers to find relevant func-tions to query terms and their usages. In our approach, identifying popular fragments is inspired by PageRank algorithm, where the "popularity" of a func-tion is determined by how many functions call it. We designed a model based on the vector space model by which we are able to establish relevance among facts which content contains terms that match programmer's queries. The result is an ordered list of relevant functions that reflects the associations between concepts in the functions and a programmer's query.

Keywords: search, source code, reuse, pagerank, ranking, functional dependencies.

1 Introduction

Software development is one of the most creative things a human can do. Every day, a programmer needs to answer several questions for the purpose of finding solutions and making decisions. It requires the integration of different kinds of project (soft-ware system) information, as well as, it depends on the programmer's knowledge, experience, skills and inference. The information retrieval is a key area to obtain suc-cess on reuse initiatives. "*To reuse a software component, you first have to find it*" [13]. There were several studies conducted to find out how programmers comprehend software systems and what information they need to know about source code [10]. The studies deal with identifying and analysing motivations, strategies, and goals, which developers have when they search in

source code. In [11] authors summarize a list of motivations (search forms) for code search. Even though several search forms are used in code search engines, there is still room for innovations.

The structure of code is not conducive to being read in a sequential style. In general, programmers read code selectively. They identify parts of the source relevant to the target task. For example, programmers are looking for use of an existing piece of code, the implementation of some functionality or code with some properties (patterns). Empirical studies indicate that 40% to 60% of source code is reusable within one application and only 15% is unique to a specific application [9]. However in software engineering, there is necessity to adopt systematic reuse code. It requires instruments which facilitate reuse such as source code search tools. The good source code search tool encourages access to the existing code instead of creating new one. The developer's motivation and goals for searches indicate certain functionality, which is required in such search tools.

In general, the process of identifying the parts of the source code that correspond to a specific functionality is called *concept/feature location*. The aim of the process of *concept location* is to find the source code that implements these concepts. The input to the concept location process is a description of a problem (a change task) expressed in natural language and the output is a set of software components (elements) that implement or address the concept. However, there is a difficulty that the input and the output are in different levels of abstraction, i.e., the input is formulated in natural language (domain level) and the output is the source code (implementation level). Therefore, considerable knowledge is required to translate from one level to another.

The complexity and significance of the concept location process increases with the size of the software system. The aim of the methods for concept location is to reduce the search space which the programmer needs to investigate (explore). One common way of various approaches is a decomposition of the source code into units, such as classes and functions (methods) which are enriched with additional information (e.g. relationships between elements of the source code).

Understanding code and determining how to use it is a manual and time-consuming process. In general, programmers try to find initial points such as relevant functions (methods in object oriented parlance). They want easily understand how the functions are used and see the sequence of function invocations in order to understand how some concepts are implemented [10]. When programmers learn about a program (source code), the control flow (execution of function calls) needs to be followed. It means successive jumping from one function to another.

A code query helps to identify locations of interest in the source code. There exist many developer tools and environments that facilitate a developers' work. Short code fragments, which are returned to a programmer's query in current programming environments, do not provide enough "background" to help them reuse the fragments, and programmers usually have to invest considerable effort to understand how to reuse the fragments. When programmers write new code,

they are often interested in finding definitions of functions, existing, working fragments, with the same or similar functionality and reusing as much of that code as possible. It is important that search engines support programmers in finding answers to similar questions and issues.

The most commonly used tools for code queries such as grep are based on purely text-based pattern matching. Even the present development environments, such as Eclipse and Visual Studio, require a great deal of learning effort. When solving a task, programmers usually have particular questions in their mind, such as "*Who implements this interface or these abstract methods?*" [10]. Such question often cannot be answered directly using existing functionality offered by development environments. Even if a particular conceptual query is directly supported, beginning programmers are not often familiar with their development environment, i.e., they are not yet aware of the integrated support features. For example, even though there is a feature "Find references..." in a context menu of Eclipse and we can easily answer to the query such as "*Where is this method called*", novice programmers still need to be aware that the feature - hidden in the context menu - is what they are looking for.

Keyword-based code search tools face the problem of low precision on their results due to the fact that a single word of the programmer's query may not match the desired functionality. It is because no source code content is analyzed or the programmer's needs are not clearly represented in the query.

Our goal is to enable programmers to find relevant functions to query terms and their usages. In our approach, identifying popular fragments is inspired by PageRank algorithm, where the "popularity" of a function is determined by how many functions call it. We designed a model based on the vector space model, by which we are able to establish relevance among facts which content contains terms that match programmer's queries directly. The result is an ordered list of relevant functions that reflects the associations between concepts in the functions and the programmer's query.

This paper is structured as follows. The second section provides an overview of related work. The third section presents processing of the source code repository. In the section 4, searching for relevant functions is presented. Finally, an evaluation is outlined in the section 5.

2 Related work

Current source code search engines are based on information retrieval approaches. Text-based information retrieval systems are successfully used to locate relevant documents. Extraction of keywords from comments, names of functions and variables were often sufficient for finding reusable routines [5]. However, these source code search engines process code as plain text and extracted keywords have unknown semantics. In other words, the search engine compares query keywords to the names of the objects and retrieves matches. It is the most simplistic approach, which does not take into account additional information such as dependencies among objects.

Programs contain functional abstractions, which provide an essential level for code reusing. In other words, programmers define functions once and call them from different places in code. Approaches using functional abstractions to improve code search was proposed in [6, 8]. However, these code search engines do not analyze how functions are used in the context of other functions, despite the fact that understanding the sequence of function invocations is one of the important questions that programmers ask [10].

Some approaches are based on programmer's query refinement. In [12] authors present approach, where queries and restrictions can be formulated in natural language, for example, "*Give me details about the bidding process*". The presented approach is based on the use of domain models containing the objectives, processes, actions, actors and a domain ontology, their definitions, and relationships with other domain-specific terms.

In [14] is presented an approach (CodeBroker) based on tracking what the programmer was doing. Instead of waiting for programmers to explore the reuse source code repository by using explicit queries, information delivery autonomously locates and presents components (code fragments) by using the programmers' partially written programs as implicit queries. Presented tool does the similarity analysis between components based on a concept similarity or constraint compatibility. The concept similarity is identified based on comments in the source code. A constraint similarity is identified based on the function (method) signatures. It further refines the query with inputs from the programmer.

An approach presented in [4] is based on using learning techniques. Authors focus on the task of searching for software in large, complex, and continuously growing libraries. They introduce the concept of active browsing, where an active agent tries to infer programmers' intentions and advise them.

CodeFinder presented in [7] is a tool which uses a query browser to help the programmer construct queries that can be sent to the repository. This helps the programmer make more effective queries. The tool is able to craft the query in a way that can be best used by the source code repository. CodeFinder is based on Spreading Activation algorithm to search sample source code. The advantage of CodeFinder is that it helps the programmer refine and reformulate the query. However, as the repository of sample code increases, Spreading Activation may provide some unrelated results.

Sourcerer [1] is a search engine for open source code that extracts fine-grained structural information from the code as a search basis. This information is used to enable search forms that go beyond conventional keyword-based searches.

Codifier [2] is a programmer-centric search interface (tool), which enable programmers to ask specific questions related to programming languages. It is based on indexing source code using modified compilers (C, C++, C#, VBScript) to extract lexical and syntactic metadata.

Although the mentioned tools are promising, they do not seem to leverage the various complex relationships which are present in the source code and therefore have limited features. Developers have to spend considerable time by

tracking relationships and the tools do not help them effectively organize the relationships.

Our proposed method consists of two phases, namely processing of the source code repository and searching for relevant functions given a programmer's query.

3 Processing of the source code repository

There are two main elements in the phase of the processing of the source code repository, namely an index creator and a function graph creator (see Figure 1). The index creator (A1) creates a document index and a term index (A2) from the source code repository. The purpose of the index creator is to enable to retrieve relevant functions based on matches between terms in programmer's queries and terms in the source code files. The function graph creator (B1) creates a directed graph of functional dependencies (B2). The PageRank process (C) is run on the directed graph of functional dependencies, and it calculates a rank vector, in which every element is a score for each function in the graph.

3.1 Index creator

The index creator creates indexes (document and term indexes) from all source code files (documents D) of projects in the repository. The creator uses the vector space model which is used by search engines to rank matching documents according to their relevance to a search query. By using our parser, from each document , terms t are extracted from comments, names of functions and identifiers. For each term t, NLP (natural language processing) techniques, such as stemming and identifier splitting, are applied. Each document d is modeled as a vector of terms which occur in that document. For storing the indexes, we have adopted the distributed database management system Apache Cassandra (cassandra.apache.org). It is a highly scalable, distributed and structured key-value store with efficient disk access. It is a hybrid between columnoriented DBMS and row-oriented store. Cassandra was especially designed to handle very large amounts of data. The created document index is structured as follows: contains a unique document identifier ; a list of pairs such that tID is a unique identifier of a term which occurs in dj and is a calculated term frequency value for tID and corresponding dj. The created term index is structured as follows: where for each i = 1, ... n, (n is the total number of extracted terms), each row ti contains a unique term identifier ; calculated inverse document frequency value (for calculation see below); the number of documents where the term ti occurs ; and a list of document identifiers dID in which ti occurs. Term frequency (TFi,j) for term ti and document dj is calculated as follows: where ni,j is the number of occurrences of the term ti in the document dj and is the sum of all occurrences of terms nk in the document dj. Inverse document frequency (IDFi) for term ti is calculated as follows: where —D— the total number of documents in the corpus and is the number of documents where the term ti occurs. The corpus represents the set of all source code files (documents) of all projects in the repository. The TF/IDFi,j for term ti and document dj is calculated as follows:

3.2 Function graph creator

The purpose of the function graph creator is to construct a directed graph of functional dependencies. Nodes represent functions names (full signature of functions). A directed edge from the function F to the function G is created if the function G is invoked in the function F. In object-oriented programming languages, such as Java, C# or C++, there are problems, for example, with polymorphism, inheritance and method overloading. For solving this problem we use intermediate representation of source code, i.e., an abstract syntax tree (AST) and control flow graph (CFG) obtained from AST. The AST structures provide such information as nodes representing class definitions, member declarations, function (method) definitions, variable declarations, initialization and assignment statements, and method invocation statements. This allows us clearly to identify method invocations.

The algorithm of creating the graph of functional dependencies by using our parser and AST is as follows:

1. From a document d (source code file), obtain full signature of a defined function f.
2. From the full signature of the function, create unique identifier Fn using function invocation statement obtained from AST.
3. If there do not exist a node called Fn, create a new node called Fn.
4. For each function g which is invoked in the function f :

(a) Execute step 2 and then 3.

(b) Create a directed edge from the node representing the function f to the node representing the function g.

5. If there is an unprocessed function in the document d, execute step 1.
6. If there is an unprocessed document in the repository, select this document and execute step 1.

3.3 Ranking functional dependencies

For ranking of the functional dependencies, we use the PageRank algorithm. Using the PageRank, we are able to determine the "popularity" of a function. The PageRank of a function is defined recursively and depends on how many functions call (invoke) it. The rank value indicates importance of a particular function. On the other hand, following functional dependencies help programmers to understand how to use found functions, i.e. they can see and trace the sequence of function invocations.

The formula for PageRank of a function fi, denoted r, is the sum of the PageRanks of all functions that invoke f:

where is the set of functions that invoke fi and —fj— is the number of functions that the function fj invokes. It is applied iteratively starting with r0(fi) = 1/n, where n is the number of functions. The algorithm is repeated until

the PageRank score converges to some stable values or it is terminated after some number of iterations. Functions called from many other functions, have a significantly higher PageRank score than those that are used infrequently.

There are some other methods which could be used in this phase such as Spreading Activation Network, where the direction of edges and weight reflect the meaning and strength of associations among documents.

4 Searching for relevant functions

The search phase (illustrated in Figure 2) enables programmers to find relevant functions to query terms and subsequently to trace their usages. Searching consists of three main steps. First, (top) relevant documents are retrieved based on a similarity sim(dj, q) between documents (source code files) and programmer's query q. Second, each document dj is divided into subdocuments, where each one contains only one definition of a function Fn contained in the "parent" document dj. For each subdocument , a similarity sim(, q) to the query q is calculated. Finally, an ordered list of relevant functions is obtained so that, for each function Fn (Fn), a final score sc(Fn, q) is calculated as the sum of the similarities and a PageRank score pr(Fn).

4.1 Retrieving relevant documents

When a programmer enters a query (1), for each query term wi, the NLP techniques are applied and subsequently a list of (top) relevant documents (source code files) is retrieved (2). The list contains documents, where at least one query term occurs in each document. A similarity (3) between two documents (query q and a relevant document dj) is calculated (4) using the cosine similarity (distance) as follows:

where da, db are document vectors. Elements of the vectors are pre-calculated TF/IDF weights.

4.2 Subdocument processing

1. Each retrieved relevant document is divided into subdocuments, where each one contains only one definition of a function with surrounded comments (if any).
2. From each subdocument, terms are extracted from comments, function name and identifiers.
3. For the extracted terms, TF/IDF weights are calculated (a subdocument vector).
4. For each subdocument djk, the cosine similarity (5) to the programmer's query q (6) is calculated.

4.3 Ranking of the relevant functions

1. From the relevant (sub)documents, unique identifiers Fn are created using function invocation statements obtained from AST.
2. For each function Fn, a final score sc(Fn, q) is calculated as a sum of:

(a) a similarity sim(dj, q) between the document dj in which the function Fn is defined (Fn dj) and the programmer's query q (4);
(b) a similarity sim(, q) between the subdocument djk in which the function Fn is defined (Fn) and the programmer's query q (6);
(c) a PageRank score pr(Fn) for the function Fn (7)(8) which represents "global popularity" of the function.

Based on our initial experiments we identified a problem with included support libraries (in the project). Consider the following situation when a programmer wants to find a function for compressing texture in the target project and she enters, for example, a query "compress, texture". However, "similar" function can be defined in a source code of an included support library. Moreover, this function can be called from many other functions defined in the library, too. In the case that a function for compressing texture is directly defined in the project we want to prefer this function. We solve this problem using weighting of functions of the supported libraries, i.e. we multiply their calculated score by so-called damping factor = 0.5.

5 Evalution and conclusions

Typically, search engines are evaluated by experts whose task is to determine relevance of the results for a given query. We implemented a plugin into Microsoft Visual Studio which allows programmers to formulate a query and the result is an ordered list of relevant code fragments retrieved by our method. To determine how effective our method is, we conducted an experiment with 6 participants and 2 software projects, namely, ANNOR and The Green Game. These projects are written in C# programming language. ANNOR is an application for automatic image annotation and The Green Game (TGG) is a strategic computer game. Our goal was to evaluate how well these participants could find code fragments that matched given tasks. We divided the participants into 2 groups of 3 members (group#1, groupq#2) and we performed two experiments.

In the first experiment, we created a set of 6 change tasks, i.e., 3 tasks for each project. These participants reformulated the target tasks into a sequence of words that described concepts they needed to find. During performing the target tasks, the participants were asked to use only the search engines, i.e., supporting tools, such as the solution explorer and jumping, were prohibited.

The set of 6 change tasks, which we used in our experiment, is as follows: 1. Find the method for calculating a co-occurrence rank of obtained annotation and change the damping factor to the value 0.3. 2. Find the method for loading an input image and add a log event if loading the image fails. 3. Find the code fragment for extracting local features from training images and change the sigma

parameter to the value 0.45. 4. Find the code fragment for changing color of a selected object (building) and change the current color to yellow. 5. Find the method which starts playing background music automatically in main menu and disable this feature. 6. Find the method for rendering a radial cursor and change the radius parameter to the value 10.

The tasks 1-3 were specified for the ANNOR project (ANNORtasks) and the tasks 4- 6 were specified for the TGG project (TGGtasks). The goal of the participants of the group#1 was to perform ANNORtasks using our search engine. The goal of the participants of group#2 was to perform these change tasks using the built-in search engine. Subsequently, on the contrary, the participants of group#1 were tasked to perform TGGtasks using the built-in search engine and the participants of group#2 were tasked to perform these change tasks using our search engine.

After performing these tasks, we compared the number of participants‘ queries created using our search engine and the built-in search engine. The average number of participants' queries is shown in Table 1. We can see that by using our search engine, the participants had to make less effort for locating target code fragments (methods). It is confirmed by the average number of created queries for the target change tasks.

In the second experiment, we specified 2 implementation tasks. These tasks were focused on reusing code fragments (methods) and they were formulated as follows: 1. ANNOR: Implement a tool for selecting rectangular area in the target image. Reuse the tool for selecting irregular polygon area as much as possible. 2. The Green Game: Implement a module for rendering an elliptic cursor. Reuse functionality for rendering the point cursor as much as possible.

The participants of group#1 were tasked to implement the first task using the builtin search engine whereas the participants of group#2 were tasked to implement the first task using our search engine. For implementing the second task, the participants of group#1 used our search engine and the participants of group#2 used the built-in search engine.

For each query, each participant evaluated relevance of the results. In other words, once the participants obtained lists of code fragments which were ranked in descending order, they examined these code fragments to determine if they matched the tasks. For a query and for each obtained result, a level of confidence, such as complete ly/mostly irrelevant, mostly/highly relevant, was assigned by the participant. Retrieved fragments were evaluated as relevant only if they are ranked with the confidence levels mostly or highly relevant, i.e., a retrieved code fragment is relevant to a task and the participant can understand how to reuse it to solve the task or it can be reused directly.

In our experiment, we used the precision metrics which reflects the accuracy of the programmer's search. The precision is the fraction of the top 5 ranked code fragments which are relevant to the query. The precision P is calculated as follows: Since we limited the number of retrieved code fragments to top 5, the recall was not evaluated in this experiment.

The calculated average precision is shown in Table 2. It illustrates that search results, obtained using our search engine, were more relevant during performing the tasks compared with the use of the built-in search engine.

Programmers often want to locate code fragments which are notable in a software project. Such fragments may represent, for example, internal patterns which should be used or reused during implementing certain functionality, and therefore they do not want to implement their own solutions from beginning. Our approach is able to locate such fragments, because in addition to the relevance of code fragments to a given query, our method establishes "popularity" of code fragments, i.e., it prefers such code fragments which support and motivate programmers in the process of reuse of existing solutions.

Our proposed method could be used in collaborative programming [3], too. For example, programmers can annotate code fragments based on identifying their "popularity" (among several projects). By adding new annotations to the source code such as pattern/exemplar, good example, there could be improved orientation of the programmers in the code through disclosure of the current state. The programmers would be able to see directly which parts of the code are interesting, stable or (often) reused.

Acknowledgement. This work was partially supported by the grants VG1/0675/1/2011-2014, APVV-0208-10 and it is the partial result of the Research & Development Operational Programme for the project Research of methods for acquisition, analysis and personalized conveying of information and knowledge, ITMS 26240220039, co-funded by the ERDF.

References

1. Bajracharya, S., et al.: Sourcerer: a search engine for open source code supporting structure- based search. *In Companion to the 21st ACM SIGPLAN symposium on Objectoriented programming systems, languages, and applications (OOPSLA '06)*, NY, 2006, pp. 681-682.
2. Begel, A.: Codifier: A programmer-centric search user interface. *In Proc. of the workshop on human-computer interaction and information retrieva*l, 2007, pp. 23-24.
3. Bieliková, M., et al.: Collaborative Programming: The Way to "Better" Software. *In Proc. of 6th Workshop on Intelligent and Knowledge Oriented Technologies*. EQUILIBRIA, s.r.o., Košice, 2011, pp. 89-94. (in Slovak)
4. Drummond, Ch., G., et al.: A Learning Agent that Assists the Browsing of Software Libraries. IEEE Trans. Softw. Eng., 2000, pp. 1179-1196.
5. Frakes, W., B., Pole, T., P.: An Empirical Study of Representation Methods for Reusable Software Components. IEEE Trans. Softw. Eng., vol. 20, 1994, pp. 617-630.
6. Grechanik, M., et al.: A search engine for finding highly relevant applications. *In Proc. of the 32nd ACM/IEEE Int. Conf. on Softw. Eng. (ICSE '10)*, ACM, NY, 2010, pp. 475-484.
7. Henninger, S.: Retrieving software objects in an example-based programming environment. *In Proc. of the 14th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR '91)*. ACM, New York, 1991, pp. 251-260.
8. Ossher, J., et al.: SourcererDB: An aggregated repository of statically analyzed and crosslinked open source Java projects. *In Proc. of the 2009 6th IEEE Int. Working Conf. on Mining Softw. Repositories (MSR '09)*. IEEE CS, Washington, 2009, pp. 183-186.
9. . Morisio, M., et al.: Practical Software Reuse. Springer-Verlag, London, 2002.
10. Sillito, J., et al.: Asking and Answering Questions during a Programming Change Task. *IEEE Trans. Softw. Eng.* 34, vol. 4, 2008, pp. 434-451.
11. . Sim, S., E., et al.: Archetypal Source Code Searches: A Survey of Software Developers and Maintainers. *In Proc. of the 6th International Workshop on Program Comprehension (IWPC '98)*. IEEE Computer Society, Washington, 1998, pp. 180-.
12. Sugumaran, V., Storey, V., C.: A semantic-based approach to component retrieval. *SIGMIS Database*, ACM, New York, vol. 34, 2003, pp. 8-24.
13. Prieto-Diaz, R., Freeman, P.: Classifying Software for Reusability. IEEE, 1987, pp. 6-16.
14. Ye, Y., Fischer, G.: Supporting reuse by delivering task-relevant and personalized information. *In Proc. of the 24th International Conference on Software Engineering (ICSE '02)*, ACM, New York, 2002, pp. 513-523.