Spotting Working Code Examples

Iman Keivanloo
Department of Electrical and Computer Engineering
Queen's University
Kingston, Ontario, Canada
iman.keivanloo@queensu.ca

Juergen Rilling
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada
juergen.rilling@concordia.ca

Ying Zou
Department of Electrical and Computer Engineering
Queen's University
Kingston, Ontario, Canada
ying.zou@queensu.ca

ABSTRACT

Working code examples are useful resources for pragmatic reuse in software development. A working code example provides a solution to a specific programming problem. Earlier studies have shown that existing code search engines are not successful in finding working code examples. They fail in ranking high quality code examples at the top of the result set. To address this shortcoming, a variety of pattern-based solutions are proposed in the literature. However, these solutions cannot be integrated seamlessly in Internet-scale source code engines due to their high time complexity or query language restrictions. In this paper, we propose an approach for spotting working code examples which can be adopted by Internet-scale source code search engines. The time complexity of our approach is as low as the complexity of existing code search engines on the Internet and considerably lower than the pattern-based approaches supporting free-form queries. We study the performance of our approach using a representative corpus of 25,000 open source Java projects. Our findings support the feasibility of our approach for Internet-scale code search. We also found that our approach outperforms Ohloh Code search engine, previously known as Koders, in spotting working code examples.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques – *object-oriented programming*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval – *search process*;

General Terms

Algorithms, Performance, Experimentation

Keywords

Source code search, clone detection, working code example

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ICSE '14, June 01 - 07 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2756-5/14/06…$15.00.

1. INTRODUCTION

A source code example is a code snippet, *i.e.*, a few lines of reusable source code, used to illustrate how a programming problem can be solved. Source code examples play an important role in programming, and provide an intrinsic resource for learning [27][29] and reusing [10][18][19]. Code examples can accelerate the development process [23] and increase the product quality [25]. A *working code example* is a code snippet that can be considered for both learning and pragmatic reuse. Such working code examples can spawn a wide range of applications, varying from API usage (*e.g.*, how to use JFreeChart library to save a chart) to basic algorithmic problems (*e.g.*, bubble sort). An ideal working code example should be concise, complete, self-contained, and easy to understand and reuse.

It is not common in software development to explicitly document code examples [15][35][37]. Programmers have to search for code examples through previously written projects and publicly available code repositories on the Internet (*e.g.*, sourceforge.net). Search for source code examples on the Internet is realized using source code search engines [6][10]. These engines are known as *Internet-scale code search engines* [14], such as Ohloh Code (previously known as Koders) and Google code search [13] (discontinued service as of March 2013).

Textual similarity between code snippets and the query is the dominant measure used by existing Internet-scale code search engines. Holmes *et al*. study [16] shows that such similarity is not sufficient for a successful code example search. Buse and Wiemer [10] discuss that the answers of existing code search engines are usually complicated even after slicing. In summary, existing Internet-scale code search engines do not support spotting working code examples well.

To improve Internet-scale code search engines without affecting the way users interact with the engines, any candidate solution must hold three properties. First, the input and output format of the search algorithm must be identical to that of Internet-scale code search engines. Specifically, it must support free-form querying. The output should be a ranked set of code snippets. Second, the time complexity of the search algorithm must be low. Otherwise, the solution cannot be integrated with online engines for search on large-scale data. Third, it should not be limited to API usage queries. Although, recent studies show that a considerable amount of code search queries are related to API usage problems, programmers still use code search engines for other problems [3][28].

Several approaches, known as *pattern-based code search*, have been proposed in the literature. None of these approaches satisfies all of these properties. For example, the earlier approaches hold high time complexity. The time complexity of SNIFF at run time is $O(n^2)$ [12], where n is the number of snippets. Such complexity decreases their applicability when there is a comprehensive corpus covering millions of code snippets. Moreover, the earlier approaches mainly focused on API usage and their output is *abstract programming solutions*. An abstract programming solution is a sequence of API method call fingerprints, *e.g.*, $\mathrm{File.open} \rightarrow \mathrm{File.read} \rightarrow \mathrm{File.close}$.

We propose an approach that improves the performance of Internet-scale code search engines in the context of spotting working code examples. Our research is different from earlier work (*e.g.*, SNIFF [12] and PRIME [26]), as our approach satisfies the three mentioned conditions by exploiting clone detection fundamentals. Our approach combines Baker's *p-strings* [4] with Carter *et al*. [11] vector space model-based approach for clone detection. We extend this idea using frequent itemset mining [7] to detect popular programming solutions. This approach is not limited to API fingerprints; it considers all source code tokens. Furthermore, our approach for spotting working code examples is holding the same complexity as Internet-scale code search engines. This approach answers a free-form query within hundreds of milliseconds on a corpus covering millions of code snippets.

This paper makes the following main contributions:

- We propose an approach that can be used seamlessly by Internet-scale code search engines (*e.g.*, Ohloh Code). Our approach improves their performance in spotting working code examples.
- We evaluate the scalability and feasibility of our approach with a corpus of ~25,000 Java open source projects. We identify a superior ranking schema for spotting working code examples on a noisy comprehensive corpus. Finally, we observe that our approach outperforms Ohloh Code in spotting working code examples.

Organization of the paper. Section 2 outlines our research motivation. Section 3 overviews the background clone detection techniques. Section 4 provides the details of our approach in terms of fact extraction, mining and search. In Section 5, we present the evaluation results and the threats to validity. Related work, conclusions and future work are presented in Sections 6 and 7.

2. MOTIVATING EXAMPLES

In this paper, we propose an approach that finds and ranks working code examples. In this section, we elaborate on our problem statement using two motivating examples. For each example, we discuss the results one would obtain from existing Internet-scale code search engines, previously proposed pattern-based code search approaches (*e.g.*, SNIFF [12]), and our approach. To increase the readability, we discuss our approach separately from the other pattern-based code search approaches.

2.1 Bubble Sort Example

Bubble sort is a classical programming problem. However, bubble sort is not a popular sorting algorithm due to its time complexity. As a result, there is no native or popular API that implements bubble sort in Java. Nevertheless, junior developers and students still search for bubble sort implementation on the Internet [21]. A typical free-form query is either $query = \{bubblesort\}$ or $query = \{bubble, sort\}$.

```
boolean t = true;
while (t) {
    t = false;
    for (int i = 0; i < mas.length - 1; i++) {
        if (mas[i] > mas[i + 1]) {
            int temp = mas[i];
            mas[i] = mas[i + 1];
            mas[i + 1] = temp;
            t = true;
        }
    }
}
```

Figure 1. The bubble sort example

Internet-scale code search engines. The bubble sort query is one of the queries that are well supported by code search engines (*e.g.*, Ohloh Code). A relevance-based search approach [9] finds such snippet easily as it is usually embedded in a method or class that is called bubble sort. Ohloh Code exploits such structural data for term matching. As a result, the first hit returned by Ohloh Code is a correct implementation of the algorithm that is embedded in a method called `bubbleSort`. However, these engines heavily rely on term matching and relevance search that are adopted from Web search engines. Therefore, the second hit returned by Ohloh Code is an empty `if` condition. This snippet is placed as the second hit since it belongs to a file called `sorting-algorithms-bubble-sort-2.java`. This example highlights the challenges faced by any code search approach that depends solely on term matching and textual similarity.

Pattern-based code search approaches. Approaches such as PRIME [26] and SNIFF [12] cannot answer the bubble sort query. This is mainly due to the fact that they are limited to mining API related fingerprints.

Our approach. Figure 1 shows the first hit that our spotting approach returns for the bubble sort query within only 560 milliseconds. The result is based on 5.5 million indexed code fragments that each has at least 5 lines of code extracted from 25,000 open source projects. The spotted snippet is one of the implementations of a bubble sort algorithm. This motivating example also highlights one of the interesting features of our code search approach. A matching answer might not necessarily contain the query terms. We achieve this feature by exploiting ideas from clone detection. In this example, there is no occurrence of the query terms such as bubble, sort, or bubblesort within the spotted fragment; while the code snippet actually implements a bubble sort. It is worth mentioning that our search approach only uses the content of code snippet, and does not consider other resources such as inline comments, Javadocs, and the signature of the owner method, class or file. Moreover, our approach is able to search and mine over both regular and control flow statements with less complexity compared to the graph-based search approaches, *e.g.*, PRIME [26]. In summary, it is essential to support mining and searching of control flow statements for cases such as the bubble sort problem since loops and conditions constitute the cornerstone of their implementations.

2.2 MD5 Example

The second motivating example is a code search problem related to spotting working code examples for MD5 hash value generation in Java. The goal is to acquire MD5 representation of an object (*e.g.*, password) in a string format. MD5 hash code generation is not a trivial programming task using Java native libraries. First, there is no method or class name within the Java libraries called MD5. The actual class and methods responsible for the MD5 generation are `MessageDigest`, `getInstance()`, `update()` and `digest()`. Second, the conversion of the binary representation of hash values to string, has special cases to be handled. Without a proper conversion method, if a generated hash value starts with 0, this leading 0 would be omitted during the conversion from the original format to String (Binary → Numeric → String). **Internet-scale code search engines.** Existing code search engines do not support such queries (*e.g.*, $query = \{messagedigest, md5\}$) well. For example, none of the top 10 answers of Ohloh Code is a complete code example showing how to use `MessageDigest` for MD5 hash value generation. In the best case the result only shows how to create an object of `MessageDigest` using the `getInstance` method.

Pattern-based code search approaches. This query is challenging for recently proposed pattern-based code search approaches. It is not possible to use most of the earlier work for this running example as they do not support free-form queries and they are limited to API names (*i.e.*, cannot handle the MD5 term within the query). Surprisingly, SNIFF that supports free-form querying also failed to answer this query.

Our approach. Figure 2 presents the top ranked hit that our approach returns for the MD5 query. This is a complete answer based on Buse and Weimer [10] discussion since it includes all implementation steps even the error handling cases.

2.3 Summary

In summary, the examples in this section highlight three major features of our approach: (1) spotting working code examples for API usage and algorithmic problems; (2) the ability to provide self-contained examples; and (3) less dependency on term matching. Furthermore, our proposed approach requires only the code snippets content[1]. These features illustrate the potential of our approach for code search in the context of pragmatic reuse. Our approach takes advantage of clone detection fundamentals which allows us to eliminate limitations of earlier pattern-based approaches and existing Internet-scale code search engines.

3. BACKGROUND

Originally, similarity detection in source code has been explored by clone detection researchers in the context of software maintenance. The underlying algorithms (*e.g.*, Koschke [20] and Inoue *et al.* [17]) target detection of major clone types [5][32]. In general, while Type-1 mandates content similarity (*i.e.*, similarity in token names), Type-2 focuses on pattern similarity. Further changes (*i.e.*, additional statements) result in Type-3 similarity.

```java
byte[] unencodedPassword = password.getBytes();
MessageDigest md = null;
try {
    md = MessageDigest.getInstance(algorithm);
} catch (Exception e) {
    log.error("Exception: " + e);
    return password;
}
md.reset();
md.update(unencodedPassword);
byte[] encodedPassword = md.digest();
StringBuffer buf = new StringBuffer();
for (byte anEncodedPassword : encodedPassword) {
    if ((anEncodedPassword & 0xff) < 0x10) {
        buf.append("0");
    }
    buf.append(Long.toString(anEncodedPassword & 0xff, 16));
}
return buf.toString();
```

Figure 2. The MD5 example

[1] Comments, Javadoc and the method signatures are excluded. Although they are invaluable source of information for code search, we excluded them from our study to evaluate the performance of our pattern-mining approach independently.

The clone types are defined based on observable similarity forms in the source code. At source code level, clones share two forms of similarity: (1) pattern and (2) content. For example, "`int temp=0;`" and "`float f=2;`" constitute a Type-2 clone pair since they are following the same pattern. In this example, the shared pattern is observable if we ignore the token names. When substituting the major tokens by β, the output for both statements will be "$\beta\ \beta = \beta$;" which reveals the pattern similarity.

We found this approach useful for spotting working code examples. For example, by representing all code fragments in the corpus using techniques for Type-2 clone detection, we can identify common lines of code implementing bubble sort algorithm (*e.g.*, Figure 1). Since it is also possible that all code fragments implementing bubble sort do not contain exact statements, we also consider Type-3 clone detection techniques to find similar fragments with slightly different statements. Our approach is based on two early ideas proposed for clone detection in software maintenance by Brenda Baker [4] and Carter *et al.* [11]. In the following, we review each of these ideas separately. Finally, we discuss the derived similarity search model based on these two adopted ideas.

Type-2 similarity detection via p-strings. In our research, we adopted the idea of parameterized strings (or *p-strings*) from Brenda Baker's parameterized pattern matching theory [4] for Type-2 source code clone detection. *p-strings* provide us with the ability to match two strings in case of some specific forms of dissimilarity, *e.g.*, using different variable names. In this context, the strings refer to sequences of source code tokens. If two strings contain both ordinary and parameter symbols from alphabets $\sum$ and $\prod$ respectively, they are called *p-strings*. By exploiting this idea, we can detect if two strings are similar, *i.e.*, *p-match*, when a one-to-one transform function exists which translates one of the *p-strings* to the other one by manipulating the parameter symbols (alphabet $\prod$). Since dissimilarity in token names can negatively affect the recall of pattern mining in our research, we use the idea of *p-strings* to improve our recall. For example, this approach helps us to find lines of code common to implement bubble sort (*e.g.*, Figure 1) even if they are implemented with different variable names.

Type-3 similarity measurement via cosine similarity. Originally, Carter *et al.* [11] proposed cosine similarity as a measure that can be used for Type-3 clone detection. They converted code snippets into small-sized vectors. The vectors represent occurrence and frequency of the underlying programming language keywords (*e.g.*, while or return). Carter *et al.* measured the similarity between two code snippets via the angle between corresponding vectors. In our approach we extend the original idea of Carter *et al.* [11] Type-3 similarity search, by redefining the vector elements. Instead of using programming language keywords as the internal elements of vectors, we use *p-strings*.

Our generic similarity search model. Using the adopted clone detection techniques, we are able to (1) mine popular abstract solutions (*e.g.*, programming patterns for bubble sort implementation) even if the input data differs in statements and token names and (2) derive a similarity search model that finds the closest patterns to a subject. We refer to this derived function as the generic similarity search model in this paper. Our search model uses vector space model (VSM) which has been widely used in information retrieval, *e.g.*, [9]. In our case a vector captures *p-strings* of code snippets rather than terms.

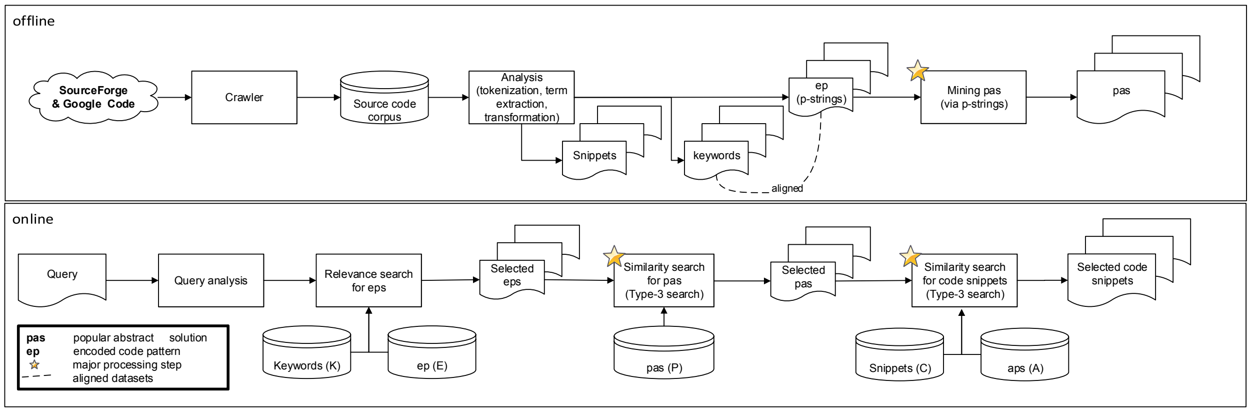


Figure 3. Our approach towards spotting working code examples

The $|x| - dimensional$ space consists of vectors, *e.g.*, $\vec{s_i} = < h_1, h_2, h_3, h_4, \dots, h_x >$, with h_x being the weight (frequency) of a *p-string* x. Similar to traditional information retrieval, we also determine both local and global popularity of an entity by calculating its occurrence frequency. Our weighting function h is derived from TF-IDF [34].

$$h(x,i) = \left(1 + log\left(l_{x,i}\right)\right) \times log\left(\frac{N}{g_x}\right) \quad (1)$$

The local frequency, denoted by $l_{x,i}$, captures the number of occurrences of a p-string x within a particular code snippet i. The global frequency g_x represents the total number of code snippets with at least one occurrence of the p-string where the total number of snippets is denoted by N.

The similarity degree between two patterns is calculated using the cosine similarity function that measures the angle between participating vectors. As a result, we derive a similarity search function that supports Type-2 and 3 pattern similarities. Note the complexity of our search function is similar to existing code search engines on the Internet (*e.g.*, Ohloh Code) since both are using the same underlying search model that is vector space model. The time complexity is $O(n\ logn)$ where *n* is the number of matches. The complexity of our approach is therefore lower than the other similar approaches (*e.g.*, SNIFF $O(n^2)$).

$$cosine\_similarity(\vec{s}_1, \vec{s}_2) = \frac{\vec{s}_1 . \vec{s}_2}{|\vec{s}_1||\vec{s}_2|} \quad (2)$$

$\vec{s_i} = < h_1, \dots, h_x >$*, with* h_x *is the weight (Eq. 1) of p-string* x

4. APPROACH

Figure 3 summarizes the major online and offline processing steps of our proposed approach. The key mining and search steps are marked in Figure 3. Both key similarity search steps are covered by the generic similarity search model (Section 3). The offline processes are responsible to extract the code snippets, generate the *p-strings* and associated keywords. Finally, the popular patterns are detected. At run-time, for a given query, first the most relevant *p-strings* are identified. Then, we use the generic similarity search model two times consecutively, to first find the best candidate popular patterns and second locate the best code examples.

4.1 Definitions

In this section, we define the major concepts in our approach.

Definition 1. A **query** is defined as an unordered set $q = \{t_1, t_2, \dots, t_n\}$, where each term t_i can be a data type (*e.g.*, primitive type, class name or interface), method name, or general concept (*e.g.*, download or bubblesort).

Definition 2. A **code snippet** c is defined as an ordered list of lines of code that are extracted from a method body. We use the code snippet only for final result preview. S denotes the collection of all snippets in the corpus.

Definition 3. An **encoded code pattern** ep_y is a set of all strings that are all mutually *p-match* using the *p-strings* idea: $ep_y = \{p_1, p_2, \dots, p_x\}$. Each ep_y is identified by a unique identifier, y. For each non-empty line of code written in Java, we add a string p_i. By applying identifier splitting techniques on all strings belong to an ep_y, we extract a set of **associated keywords**. The set of associated keywords for the given ep_y are called ep's keywords $epk_y = \{k_1, k_2, \dots, k_y\}$. K and E denote the collections of all epks and eps respectively where corresponding entities are aligned. Figure 4 provides some examples for ep and epk sets.

Definition 4. An **abstract (programming) solution** aps is a (unordered) set of encoded code patterns $\{ep_1, ep_2, \dots, ep_y\}$. In the major processing steps, we use the abstract presentation (*i.e.*, aps_m) of code snippets c_m. aps_m is a (unordered) set of all encoded code patterns corresponding to lines of code within the snippet c_m. Figure 4 presents an example for aps of a snippet related to MD5 hash code generation. Finally, we define a **popular abstract solution** (pas) as an abstract solution that all of its items occur in more than a certain number of apses in the given corpus. A and P denote the collections of all apses and pases. While corresponding items within A and C are aligned, P remains a standalone collection.

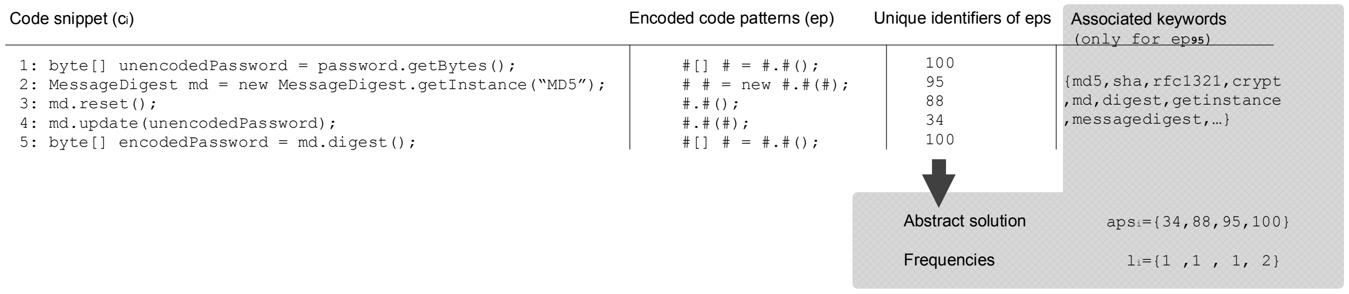
4.2 Data Analysis Steps

In this section, we describe the details of our approach for mining *popular abstract solutions* (*pas*).

4.2.1 Fact extraction

The approach, shown in Figure 3, requires at least two data families: (1) code snippets and (2) popular abstract solutions. While the code snippets can be extracted from extensive web crawling and data gathering, identifying popular abstract solutions require different types of data processing.

The initial processing steps populate all parts of the corpus, except for the popular abstract solutions. The only input data is a source code dataset crawled from the Internet. The dataset only includes source code snippets. There is no need for other information such as content of dependencies (*e.g.*, binaries of libraries). We split each file into code snippets. For each method body, we create one code snippet.


Code snippet (ci)
Encoded code patterns (ep)
Unique identifiers of eps
Associated keywords (only for ep95)
Abstract solution
Frequencies

```
1: byte[] unencodedPassword = password.getBytes();          #[] # = #.#();         100
2: MessageDigest md = new MessageDigest.getInstance("MD5"); # # = new #.#(#);      95
3: md.reset();                                              #.#();                 88
4: md.update(unencodedPassword);                            #.#(#);                34
5: byte[] encodedPassword = md.digest();                    #[] # = #.#();         100

{md5,sha,rfc1321,crypt
,md,digest,getinstance
,messagedigest,…}

Abstract solution    aps1={34,88,95,100}
Frequencies          l1={1 ,1 , 1, 2}
```

Figure 4. An example for abstract programming solution extraction

To create the *abstract programming solutions*, the original content (code snippets) have to be transformed at a higher level of abstraction. Our approach uses the idea of *p-strings* to create this abstract representation, which allows for removal of unnecessary code details (*e.g.*, variable names). We exploit a concept called *encoded code pattern* (ep) in order to map the *p-strings* idea to the vector space model, instead of the original parameterized suffix tree-based search model [4]. In this approach, we assume all *p-strings,* that are *p-match*, share the same *encoded code pattern*. We also assign a unique identifier to each encoded code pattern. Figure 4 provides an example for all internal (*i.e.*, temporary) and persistent extracted facts. In this example, the input is a code snippet denoted by c_i which has five lines of code. First, we create the corresponding ep for each line of code. Second, we map all extracted eps to the corresponding unique identifiers. Type-2 clone detection via the *p-strings* [4] helps us to locate similar code patterns efficiently (*i.e.*, $O(1)$) such as lines 1 and 5 in Figure 4. For each ep_x, we maintain a global set of all associated keywords known as epk_x. That is every time that we meet an occurrence of an ep_x, we extract all keywords from the given line of code and add them to the epk_x. Figure 4 includes an example of associated keywords to ep_{95}. As the example shows, in addition to the current extracted keywords such as MD5, the set also includes other related terms such as SHA that have been added earlier by visiting the same pattern (*i.e.*, ep_{95}) in other usage scenarios and snippets. Finally, the gray area highlights the actual persistent output that constitutes three major sections: (1) updated associate keyword sets epk_x, (2) the abstract solution aps_i, and (3) the frequency information l_i of snippet *i*. These three fact types constitute the baseline search space of our approach. Note, aps_i is a set and it does not include the ordering information of the corresponding code snippet c_i. Using such data presentation (*i.e.*, aps_i and l_i), we can apply the vector space model and cosine similarity for Type-3 similarity search.

4.2.2 Mining popular abstract solutions

Abstract programming solutions and associated keywords are not sufficient to support spotting working code examples. We need to identify the popular solutions in the corpus, in order to improve the quality of our search approach. To identify the popular abstract programming solutions a frequent itemset mining, such as the FPgrowth algorithm [8], can be employed. Since the input for the algorithm are encoded code patterns, *aps*, (not the actual code), the output will be popular abstract programming solutions, pas.

Frequent itemset mining algorithms [7] are capable of extracting popular patterns within a provided record set, with a record being one or more items. In its most simplistic form, the algorithm requires a dataset and a *support* value. The support value determines the minimum number of occurrences of a pattern before it can be considered a frequent itemset.

We adapted a variation of the itemset mining concept referred to as *maximal frequent itemset mining*. This variation has two specific properties: (1) it considers *maximal itemsets* and (2) it has no ordering constraint. The omission of the ordering constraint provides us with a robust mining feature, where re-ordering of code statements does not interfere with the pattern mining process. Moreover, this approach helps us to successfully run the algorithm on large-scale data successfully. The maximal property overcomes some of the challenges of the other itemset mining approaches, such as the possibility of producing an exponential number of frequent sub-itemsets. The occurrence of sub-itemsets in the search space is a threat when answer completeness is required. A maximal itemset is defined as: given m possible elements (*i.e.*, encoded code patterns) in the code base $E = \{e_1, e_2, \dots, e_m\}$ and n code fragments $C = \{c_i \mid c_i \subseteq E, i \in \{1, \dots, n\}\}$, $PAS^{y,x}$ is the set of all possible reputable code patterns defined as ${P_R}^{y,x} = \{e_l \mid \exists A \subseteq C, |A| = y, e_l \in \bigcap_{c_k \in A} c_k\}$ where $|{P_R}^{y,x}| \geq x$. A frequent itemset ${P_R}^{y,x}$ is maximal if $\nexists {P'_R}^{y,x} \in PAS^{y,x}, {P_R}^{y,x} \subset {P'_R}^{y,x}$. x is the minimum size and y is the support (*i.e.*, min popularity of the pattern).

As the final data processing step, we run the mining algorithm to extract popular abstract solutions. The (1) popular abstract solutions, P, along with, (2) original abstract solutions, A, (3) their occurrence frequency information, and (4) associated keywords, K, constitute the complete search space of our approach.

4.3 Searching and Ranking Working Code Examples

Since achieving an optimum result set in our research context is often impractical [26], the alternative is to provide *retrieval* and *ranking* models capable of producing high quality ranked result sets. Popularity of a solution is a key criterion that cannot be ignored to avoid poor quality result set, *e.g.*, [37]. There are other factors affecting the ranking process, *e.g.*, textual similarity. For a free-form query, similarity is continuous (not binary), so solutions other than simple filtering and matching are required. The tradeoff between these factors makes spotting working code examples a challenging task. In the following, we describe how we use our generic similarity search model (Section 3) to satisfy the above mentioned concerns. For a given free-form query $q = < term_1, term_2, \dots, term_n >$, the approach returns a ranked result set of working code examples by finding: (1) the most relevant encoded code patterns to q, (2) the most complete popular abstract solution with regard to the output of the first step, and (3) the most similar code snippets to the output of the second step.

Step 1 – relevance search. The first querying process, in Figure 3, selects the $top\ K'$ relevant encoded patterns, by comparing their associated keywords to the query terms. That is, the data used in this search problem are query terms and ep's keywords, K, while the output consists of encoded patterns. It should be noted that an encoded code pattern ep that shares a keyword with q is not automatically included in the candidate list. Only $top\ K'$ hits are selected, in order to maintain the relevancy between the query and the final spotted code fragments, as query terms are no longer used explicitly in the search process after this step. Moreover, the matching eps are ranked based on their relevancy to the query. This step reuses the well-known relevance search approach as Web search engines, *e.g.*, [9].

Step 2 – similarity search for *pas* selection. In this phase, the $top\ K$ popular abstract solutions are identified using our generic Type-3 similarity search model, where the query is made of the candidate encoded patterns, *i.e.*, the output of Step 1. Due to the clone search-based approach, the $top\ K$ popular abstract solutions can now be ranked based on their similarity to this *intermediate* query. In this greedy approach, we look for the completest popular solutions that are satisfying most of the candidate encoded code patterns.

Step 3 – similarity search for working code examples. In the last step, Figure 3, the spotting of the best working code examples takes place. The ranked result set of the previous step identifies the list of candidate popular abstract solutions sorted based on their completeness and relevancy to the free-query. In this phase, we iterate over the candidate list, and use each *pas* as a separate query to search over abstract solutions of indexed code snippets using our Type-3 similarity search model. For each pas, we find the most similar (*i.e.*, concise and complete) code snippets. The spotted snippets are ranked based on their *abstract solution* similarity to the target pas. Additionally, this step is necessary since it ensures that the results are syntactically and semantically correct, which is crucial as our pas mining and querying model ignores the ordering of the statements. The internal result of this search approach is a two-dimensional hit list for each free-form query. Each row contains the ranked code snippets matching a corresponding poplar abstract solution. Therefore, while the fragments in each row are highly similar, they look different from solutions in the other rows, as they satisfy different popular abstract solutions. Finally, we report the spotted working code examples for a given free-form query by selecting the first hit of each candidate popular abstract solution. Algorithm 1 summarizes the three search steps of our approach.

Algorithm 1. Searching and Ranking algorithm

```
Input: q, K, E, P, A, S
Output: H
1: $E_{topK'} \leftarrow relevancySearch(q, K, E)$
2: $P_{topK} \leftarrow simSearch(E_{topK'}, P)$
3: for all $pas_i \in P_{topK}$
4:     $S^i_{top1} \leftarrow simSearch(pas_i, A, S)$
5:     $H.add(S^i_{top1})$
6: end loop
7: return $(H)$
```

Filtering step. Some queries contain known Java types (*i.e.*, Java classes and interfaces). In this case, we discard any code snippet that is not associated with the types mentioned in the query. This is a heuristic that we derived empirically to improve the chance of returning relevant answers. The heuristic is motivated to compensate issues associated with our Type-2 similarity search that ignores the token names during the actual search process. For each query, we can have correct, incorrect, or no answer. A "No answer" status occurs when the filtering step discarded all of the $top\ K$ snippets and returns "no answer found". The value of k is limited by the number of snippets that our approach can check for filtering at run-time given the available processing resources.

5. CASE STUDY

In order to evaluate the feasibility, scalability, and performance of our approach, we require a reasonably large corpus. We also need a set of measures for assessing the quality of spotted working code examples. In this section, we summarize the details of our corpus. We also review the details of our query set and quality assessment approach. Finally, we report our findings and observations.

5.1 Setup

Corpus. Our corpus includes source code crawled in the first quarter of 2012. This dataset covers approximately 25,000 projects. The dataset is based on source code files that were downloaded from SVN, Git, and CVS repositories from SourceForge and Google Code. To remove high-level duplications in the dataset, only one Java file is selected for each available class name identified by its fully qualified name. During the filtering step, we were biased toward latest revisions such as files appeared in trunk directory. The crawled data, with duplicated files, initially included 12 million Java files. After the filtering step, the data were reduced to 3 million Java files (2.7M regular Java class files and 140K files with default package). We also discarded small-size method blocks, *e.g.*, less than five lines of code. Finally, we extracted 5.5 million code snippets. Table 1 summarizes the key statistics of our corpus.

Deployment. To deploy an instance of the search engine, we use a Linux-based system with a 3.07 GHz CPU (Intel i7) and 24 GB of RAM. The deployed instance of the search engine uses a single process and thread schema, except for the Java virtual machine processes, *e.g.*, garbage collection. The data processing steps, *e.g.*, mining and indexing, finished within a single day.

Query dataset. As part of our performance evaluation, we adopted Mishne *et al.*'s query set [26], since it is not limited to a single domain. The original dataset includes 7 queries from 6 Java libraries. However, we extended it by including additional queries. The additional queries are taken from programming questions posted on StackOverflow. To ensure that the dataset reflects real word search scenarios, we selected our candidate terms from Koders query log dataset [21]. Table 2 summarizes the final query dataset consisting of 15 queries. In addition, to the target library (*i.e.*, domain) and query terms (*i.e.*, free-form queries), Table 2 also includes the description of the queries used to evaluate the relevancy of retrieved code examples.

Quality assessment – features of working code examples. In general, not every code fragment that meets query criteria (*e.g.*, sharing terms) can be considered as a working code example [16][19]. Although there is no formal definition of what constitutes a good working code example, several features are discussed in the literature (Table 3). First, a working code example should be correct and relevant with regard to the query. The code example should include the mandatory steps required to implement the underlying programming problem. Second, a working code example should be concise [10][23][36], self-contained [10], complete [19], and easy to understand and reuse [15][26][37].

Table 1. The features of our corpus

Feature		Value
Raw Data		
Java projects		24,824
Total Java files		12,104,499
Unique~ Java files		2,882,458
LOC		~300 M
Selected fragments+		5,436,638
Selected lines*		65,478,267 (LLOC)
Processed Data		
Unique encoded lines (*ep*)		13,945,442
Observed frequent abstract solutions■	#Solutions (*pas*)	15,856,377
	Size (#encoded lines)	140,410,866
	#Unique items	77,905
	Max. observed support	2,412

~duplicated files are eliminated based on the common fully qualified name heuristic +fragments with at least 5 Logical Line of Code (LLOC) *LLOC after removing duplicated encoded lines within each fragment. Note the number of unique encoded lines can be smaller than the actual unique lines of code. For example int y=0; and int x=0; are counted only once since their encoded patterns are identical. ■maximal frequent itemsets [7] (min:4, max:30, support:20). Thresholds are selected empirically.

Table 2. The extended query set

Domain	Description	Query Terms
Apache Commons CLI [26]	Retrieve arguments from command line	{getOptionValue, CommandLine}
Eclipse UI [26]	Check user selection	{ISelection, isEmpty}
Eclipse GEF [26]	Set up a ScrollingGraphicalViewer	{ScrollingGraphical Viewer}
Eclipse JDT [26]	Create a project	{IProject,monitor}
Apache Commons Net [26]	Successfully login and logout	{FTPClient}
WebDriver [26]	Click an Element	{WebElement}
JDBC [26]	Commit and rollback a statement	{executeUpdate,roll back,PreparedStatem ent}
HTTP	Send a HTTP request via URLConnection	{response,URLConn ection}
Runtime	Redirect Runtime exec() output with System	{read,Runtime}
Memory	Get OS Level information such as memory	{Memory}
SSH	SSH Connection	{ssh}
Download	Download and save a file from network	{download,URLCon nection}
MD5	Generate a string-based MD5 hash value	{md5}
HttpResponse	Read the content of a HttpResponse object line by line	{readLine,HttpResp onse}
Lucene	Search via Lucene and manipulate the hits	{search,IndexSearch er}

Table 3. Features of working code examples

Feature	Measure
Correctness [19]	An answer should address the query expectation (sharing terms with the query is not sufficient).
Conciseness	#irrelevant LOC [19]
Readability	well-chosen variable name [10]
Completeness	#missing statements (well-typed [19], variable initialization, control flow, and exception handling [10])

Table 3 provides a brief summary of the features and measures used for identifying high quality working code examples. In this paper, we evaluate the performance from correctness, conciseness, and completeness points of view. We select these three features as they can be evaluated objectively.

5.2 Case Study Results

This section presents and discusses the results of our two research questions. For each research question, we present the motivation behind the question, the analysis approach and our findings.

RQ1 – What is the best ranking schema for spotting working code examples?

Motivation. In this paper, we focus on the problem of spotting working code examples. Spotting emphasizes on the quality of the first returned answer. Therefore, performance measures such as recall are not the major concerns when evaluating spotting approaches. This requirement distinguishes spotting working code example research from other related code search problems. Thus, we aim to find the best ranking schema for spotting working code examples. An ideal ranking places its best answer at the top of the hit list. Previously, the popularity of solutions has been considered for ranking in the context of code search (*e.g.*, [10][26][37]). The intuition is that the higher the popularity of a solution, the higher the chance of acceptance by end user. In addition to popularity, we explore the impact of similarity and size factors for spotting working code examples.

Approach. For this research question, we study the performance of a set of potential ranking schemas. We can rank code examples based on their similarity to a given query, popularity of the popular abstract solutions, and the size of the popular abstract solutions. In total, we consider five ranking schemas as summarized in Table 4. The similarity is quantified based on the position of the hits within the ranked result set produced by our approach. In other words, *S* is our intrinsic ranking schema discussed in Section 4. Popularity of an abstract solution is derived by the mining step, *i.e.*, support value of *pas*. Size is measured based on the length of the underlying *pas*. As we discussed earlier in Section 5.1 and Table 3, there are several features that high quality working code examples should hold. We evaluate the performance of each ranking schema in terms of correctness, conciseness, and completeness. The measures are selected from Table 3.

Correctness. We consider an answer to be correct only if it satisfies the description of the given query in Table 2. Note that our spotting approach may answer a question incorrectly or return "no answer" due to our filtering step (Section 4). In general, we prefer not to answer a question instead of returning a wrong answer, to avoid a negative effect on the end users' trust. However, excessive occurrences of "no answer" cases in the result sets reduce perceived quality of the search service. In this study, we distinguish between incorrect and "no answer" cases when we evaluate the schemas from the correctness point of view. We calculate the percentage of correct answers with regard to the total number of (1) queries and (2) answered queries. The two related measures are query coverage ($\frac{\#correctly\ answered}{\#total\ queries}$) and precision ($\frac{\#correctly\ answered}{\#answered\ queries}$). A good ranking schema should be able to perform well from both points of view.

Conciseness and Completeness. A correct answer might miss minor tasks such as variable initialization or error handling. Therefore, it is important to measure not only the correctness but also conciseness and completeness.

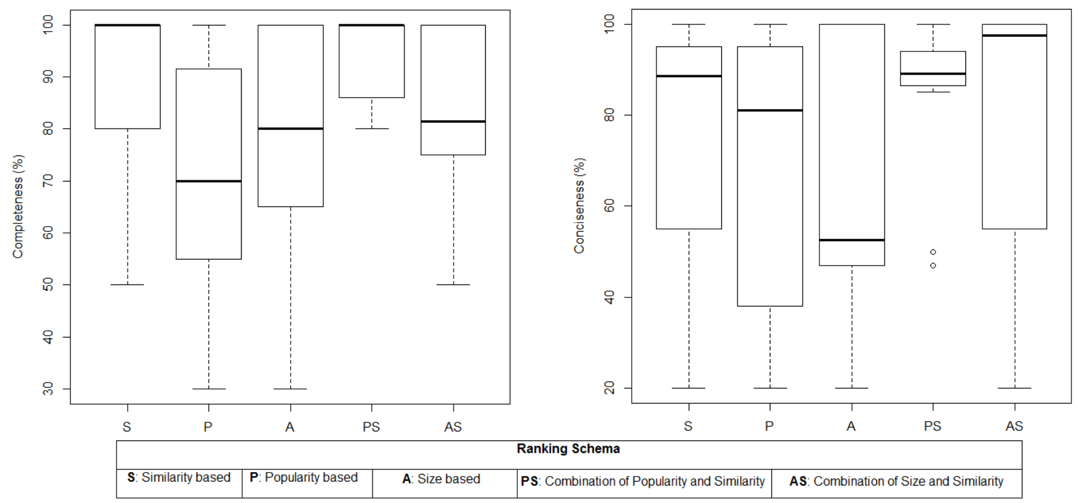


Figure 6. Summary of the completeness and conciseness measures

Table 4. Studied ranking schemas

Ranking Schema	Coverage	Precision
Similarity based (S). The candidate popular abstract solutions are ranked based on their similarity to candidate *eps*.	66%	83%
Popularity based (P). The candidate popular abstract solutions are ranked based on popularity.	80%	80%
Combination of P and S schemas (PS). The top-10 hits of S are re-ranked based on popularity.	73%	91%
Size based (A). The candidate popular abstract solutions are ranked based on their size.	66%	83%
Combination of A and S schemas (AS). The top-10 hits of S are re-ranked based on size.	66%	66%

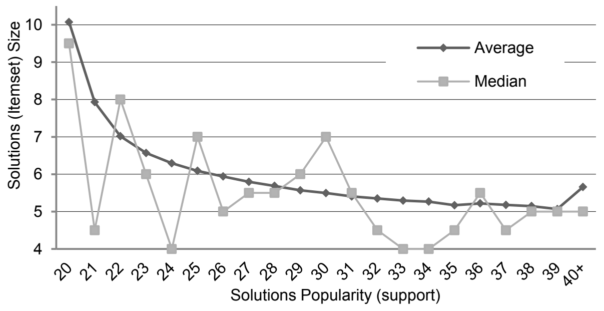


Figure 5. The average size of our popular abstract solutions

To be able to measure conciseness and correctness, we require two lists covering necessary and complementary tasks to be accomplished for each programming problem (a query). These tasks are pre-defined for our query set (Table 2). The necessary tasks are derived based on the description of each query in Table 2. The complementary tasks are related to minor programming steps such as error handling or variable declaration. We extracted these tasks for each query from the best answers provided by the original query set [26] or top ranked answers on StackOverflow. We manually measure the missing minor tasks via completeness, *e.g.*, [10][19]. Completeness is defined as the number of addressed tasks divided by the total number of tasks. Finally, for each correct answer, we manually measure conciseness. Conciseness is defined as the number of irrelevant lines of code divided by the total lines. A line of code which is not related to the identified tasks is irrelevant. We use Wilcoxon signed-rank test, a non-parametric test, to identify if the observations are significantly different.

Results. Table 4 summarizes the result of our correctness study using the coverage and precision measures. The results show that only the ranking scheme based on a combination of size and similarity, *AS*, under performs in both measures. Popularity-based ranking is able to answer more questions successfully while similarity-based and size-based schemas have better precision. The best precision is achieved by the combination of popularity and similarity. This combination also answers more questions than using only similarity, *i.e.*, *S*. During our analysis, we observed that there exists a relation between the popularity and size. Figure 5 summarizes the average size of 15,856,377 popular abstract solutions in our corpus. The abstract solutions are grouped by their popularity, which is measured by the number of occurrences of the solution within the corpus, *i.e.*, the support value. The result shows that the size decreases as the popularity increases. Except for the first few groups, the average size remains in a narrow range between 6 and 5. This behavior partially describes the lower performance of size-based ranking schemas (*e.g.*, AS).

Next, we study the performance of the ranking schemas in terms of completeness and conciseness. Figure 6 presents the summary of our results. Specifically, we focus on the surviving schemas from the correctness evaluation phase, *i.e.*, *P* and *PS*. The results show that the combination of popularity with similarity, *PS*, outperforms the pure popularity ranking schema, *P*, from both completeness and conciseness points of view. The Wilcoxon signed-rank test also confirmed that the improvement is statistically significant at the significance level 0.05.

RQ1. The combination of popularity and similarity leads to a better ranking schema for spotting working code examples.

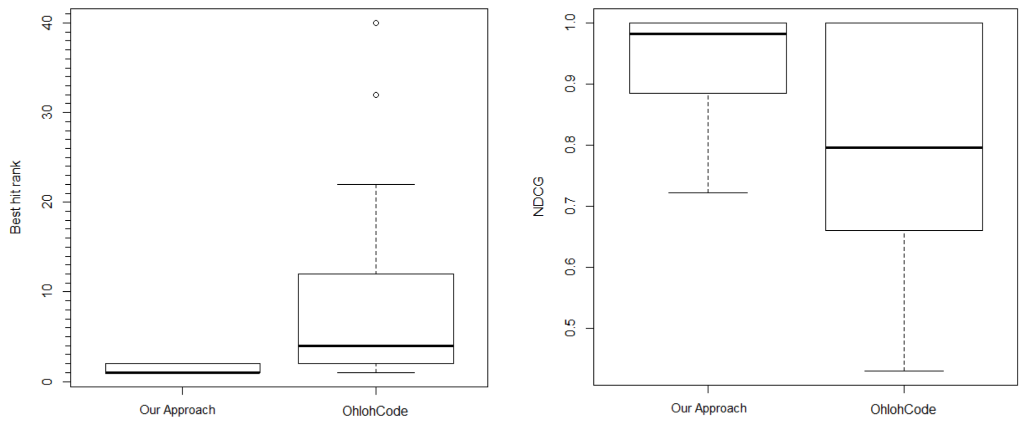


Figure 7. Summary of the best hit rank and NDCG studies - comparing our approach (PS ranking schema) with Ohloh Code

RQ2 – Can our approach outperform Internet-scale code search engines?

Motivation. The ultimate goal of our research is to propose a search approach that improves the performance of Internet-scale code search engines in terms of spotting working code examples. In this research question, we compare our approach with Ohloh Code (previously known as Koders), which is a publicly available Internet-scale source code search engine.

Approach. We evaluate both approaches from the spotting problem point of view. An ideal spotting approach places the best working code example at the top of the list. We study two measures proper for evaluation in the context of spotting working code examples. We compare the ranks of the best answers. Specifically, we find the rank of the best answer within the given top k hits. This evaluation approach reveals how many hits (*i.e.*, code snippets) the end-user should review/browse until she finds the best answer within the ranked result set. A superior approach places its best answer closer to the top of the list. The best answer is selected amongst the top k hits of each approach independently. The best answer is identified using the same measures used in RQ1 for correctness, completeness and conciseness. This is necessary and also reasonable as each approach has its own corpus. Our initial value for k is 10, *i.e.*, Ohloh Code default result set size. However, for the Ohloh Code study, there were some queries with no correct answer within the first 10 hits. In this case we increase k by 10 until we reach to a window with at least one correct answer.

We also study the performance using normalized discounted cumulative gain (NDCG) [24]. NDCG is one of the state of the art measures in information retrieval. Normalized discounted cumulative gain evaluates the ranking capability of a ranking schema, *e.g.*, whether highly relevant answers appear toward the top of the hit list. The major advantage of NDCG over the other popular measures (*e.g.*, MAP [24]) is that its result is comparable when there is heterogeneity in the data under study. It can compare two search engines or ranking schemas that are deployed with different datasets. The environment issues such as data scarcity affects less NDCG comparing to the other measures such as precision at K. We calculate NDCG of each approach for queries in Table 2. We assign relevancy values to the top k hits of each approach by considering the same three measures studied in RQ1 for correctness, completeness and conciseness.

Results. Figure 7 provides a summary of our study on the rank of the best available answer within the top k hits. The results show that our approach outperforms Ohloh code in using the available resources. Our approach places its high quality answers closer to the top of the ranked result set. Our approach ranks its best answer as the second hit in the worst case. Figure 7 also summarizes our NDCG observation. Similar to the previous measure, our approach outperforms Ohloh Code. Finally, Wilcoxon signed-rank test confirmed that the observed improvement is statistically significant.

RQ2. Our analysis shows that our approach outperforms Ohloh Code, the studied Internet-scale code search engines, for spotting working code examples, by consistently placing the best available answer within the two top hits.

5.3 Threats to Validity

There are some threats to validity related to our performance evaluation study. Threats to external validity concern the possibility to generalize our results. We identified two major threats to external validity of our study. The first concern is the differences in the input data. Ohloh Code and our approach are using different datasets. However, both are covering similar number of Java open source projects crawled from the Internet. The second concern is about the query dataset. We chose a recent query dataset that is not limited to a single application domain. Since the number of queries was smaller than earlier studies (*e.g.*, 10 to 20 queries [2][33][36]), we randomly added queries from programming questions posted on Stackoverflow. The final query dataset consists of 15 queries. We also confirmed that the query terms are reflecting real word queries by choosing terms extracted from Koders query log dataset [21]. Although, we tried to address both concerns, still they remain threats to validity of our study. In our study, we considered correctness, conciseness, and completeness measures to evaluate the quality of spotted working code examples. We selected the measures based on earlier studies on the quality of code examples. Nevertheless, these measures do not replace other evaluation approaches such as user studies.

6. RELATED WORK

Various forms of recommendation systems for software engineering exist in the literature [30]. In this section, we review a subset of the research focusing on source code search. We report the related work separately based on their output types.

6.1 Abstract Solution

Diverse mining and search approaches are proposed for recommendation on API usage scenarios. Their main output is a method call sequence. In some cases the sequence includes further information such as object instantiation steps. MAPO (Xie and Pei [38]) establishes the baseline of the frequent pattern mining-based code search in the context of API usage. Acharya *et al.* [1] extended this approach by including API static traces and employing partial orders mining. UP-Miner [37] is presented by Wang *et al.* as a successor of MAPO [38]. It combines clustering and sequence mining to find re-occurred sequences of API fingerprints (*i.e.*, method call tokens). In this area, SNIFF (Chatterjee *et al.* [12]) is one of the few models that directly addresses the free-form querying problem by exploiting longest common subsequence mining and programming documents (*i.e.*, Javadoc). However, SNIFF requires the complete compilation unit (*e.g.*, external libraries and documentations). Mishne *et al.* approach, PRIME [26], follows a lazy method by delaying the pattern mining task to run-time. Lazy mining has been investigated earlier in MAPO, SNIFF, and PARSEWeb [36]. PRIME extracts partial temporal specification from method call sequences to find possible solutions. PRIME is extending PARSEWeb's approach with extra semantic analysis.

Abstract solutions help us to identify high level steps of programming problems, but they cannot replace source code examples [10]. Although, it is possible to report corresponding code snippets for each abstract solution using proactive book keeping, the code snippets are not ranked. This is problematic for large-scale experiments when for an abstract solution numerous code snippets with different quality levels are available (*e.g.*, up to 17000 snippets [26]). Our research addresses this problem, by directly spotting and ranking code examples. Moreover, our approach is not limited to API fingerprints.

6.2 Synthesized Source Code

Another approach is to synthesize source code from abstract solutions. PARSEWeb (Thummalapenta and Xie [36]) answers programming questions formulated as "Source → Destination" template. The answer is a synthesized code snippet that satisfies the given question. To decide about the best approach to answer the given question, PARSEWeb models method invocation sequences including the data flow information as directed acyclic graphs and exploits shortest path discovery algorithms. The final synthesized answers are ranked based on their popularity and size. PARSEWeb improves Prospector (Mandelin *et al.* [23]) approach by considering additional source of information (*i.e.*, code samples) as Prospector relies only on API definitions. Recently, Buse and Wiemer [10] apply mining on graph models created from the data flow and method call sequences. Buse and Wiemer [10] research provides promising results on synthesizing source code examples, however still it cannot outperform human written code examples. Our research proposes a solution that can spot working code examples from source code written by developers.

6.3 Code Snippets

Strathcona (Holmes and Murphy [15]) uses structural matching to find the most similar code examples to API usage problems. The core of the approach is made of six heuristics for structural comparison via similarity measurement between inheritance links, method calls, and type uses. The top-10 code examples are ranked based on their success in surviving more heuristics. XSnippet (Sahavechaphan and Claypool [33]) improves Strathcona by using graph mining techniques. The goal is to decrease the number of irrelevant answers. However, it answers only object instantiation problems. To improve the ranking a combination of popularity, size, and context aspects are employed.

As discussed by Chatterjee *et al.* [12], current approaches that are returning code examples are limited to (1) API usage scenarios and (2) expect the end-user to know what classes and methods she has to use to accomplish the given development task. They rely on specific querying templates such as single API name (*e.g.*, UP-Miner [37]), "Source → Destination" (*e.g.*, PARSEWeb [36]), or incomplete program (*e.g.*, PRIME [26]). Free-form querying [12] relaxes this condition by allowing end-users to express a query using a variety of terms, including class and method names. SNIFF is the closest approach to ours. However, its output is a ranked list of abstract solutions not code examples. Its time complexity is high at runtime (*i.e.*, $O(n^2)$ where n is the number of hits) which decreases its usability for real applications with large-scale corpus. Finally, it is limited to find code example for known APIs and it requires a complete compilation unit including their documentations.

There are also some studies that propose improvement to Internet-scale code search engines. Reiss [31] proposes the idea of semantic-based code search which exploits a variety of resources such as test cases to improve code search experience. McMillan *et al.* [22] approach (Portfolio) supports programmers in finding relevant functions and their usage scenarios. Portfolio search is based on natural language processing and network analysis algorithms such as PageRank on call graphs. A comparison with Koders revealed that Portfolio is better in finding the relevant functions to a given programming task. Our research focuses on finding working code examples. The closest study to our research is done by Bajracharya and Lopez [2]. They proposed a solution that improves the quality of the retrieval approach in terms of recall by exploiting API fingerprints. Diversely, in our study, we improve the quality of the search approach in terms of finding better examples.

7. CONCLUSION

Available Internet-scale code search engines do not support well queries about code examples [10][16][26]. In this paper, we use code clone detection models to support spotting working code examples. Our approach makes it possible to mine programming patterns and search for similarities over all types of statements (*e.g.*, control flow, data flow and API fingerprints) without graph-based models improving scalability and efficiency. Our approach supports free-form querying. This is different from most of the earlier work expecting partial code, API names, or data flow information as the query. The time complexity of our approach is similar to Internet-scale code search engines. We study the feasibility of our approach with a representative corpus of ~25,000 open source Java projects. We first identify the best ranking approach in our research context. We find that the combination of popularity and similarity outperforms the dominant ranking approach in the literature. Second, we study the performance of our approach by comparing it with Ohloh Code, an Internet-scale code search engine. Our study shows that our approach significantly outperforms Ohloh Code in finding working code examples. As a result, our approach can be adapted internally by existing code search engines to improve their performance in spotting working code examples. As the immediate future work, we plan to study the feasibility of our approach for the other popular programming languages. We also plan to release our research as a standalone search engine.

8. ACKNOWLEDGMENTS

We thank Sushil K. Bajracharya for providing us insight about architecture of Internet-scale code search engines such as Ohloh Code. We also thank Chanchal K. Roy, Daniel M. Germán and the anonymous reviewers for their invaluable comments.

9. REFERENCES

[1] Acharya, M., Xie, T., Pei, J., and Xu, J. 2007. Mining API patterns as partial orders from source code: from usage scenarios to specification. In *Proceedings of Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. 25-34.

[2] Bajracharya, S. K., Ossher, J., and Lopes, C. V. 2010. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering.* 157-166.

[3] Bajracharya, S. K., and Lopes, C. V. 2012. Analyzing and mining a code search engine usage log. *Empirical Software Engineering*. Springer US, 17(4-5), 424-466.

[4] Baker, B. S. 1993. A theory of parameterized pattern matching: algorithms and applications (extended abstract), In *Proceedings of ACM Symposium on Theory of Computing*. 71-80.

[5] Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*. 33(9), 577-591.

[6] Brandt. J., Dontcheva, M., Weskamp, M., Klemmer, S. R. 2010. Example-centric programming: integrating web search into the development environment. In *Proceedings of SIGCHI Conference on Human Factors in Computing Systems*. 513-522.

[7] Borgelt, C. 2012. Frequent item set mining. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(6), 437-456.

[8] Borgelt, C. 2005. An implementation of the FP-growth algorithm. In *Proceedings of International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations*. 1-5.

[9] Brin, S., and Page, L., 1998. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1), 107-117.

[10] Buse, R. P., and Weimer, W. 2012. Synthesizing API usage examples. In *Proceedings of the International Conference on Software Engineering*. 782-792.

[11] Carter, S., Frank, R. J., and Tansley, D. S. W. 1993. Clone detection in telecommunications software systems: A neural net approach. In *Proceedings of International Workshop on Applications of Neural Networks to Telecommunications*, 273-287.

[12] Chatterjee, S., Juvekar, S., and Sen, K. 2009. SNIFF: a search engine for Java using free-form queries. In *Proceedings of International Conference on Fundamental Approaches to Software Engineering*. 385-400.

[13] Cox, R. 2012. Regular expression matching with a trigram index or how Google code search worked. Online resource: http://swtch.com/~rsc/regexp/regexp4.html (January 2012).

[14] Gallardo-Valencia, R. E., and Sim, S. E. 2009. Internet-scale code search. In *Proceedings of ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation.* 49-52.

[15] Holmes, R., Walker, R. J., and Murphy, G. C. 2005. Strathcona example recommendation tool. In *Proceedings of European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 237-240.

[16] Holmes, R., Cottrell, R., Walker, R. J., and Denzinger, J. 2009. The end-to-end use of source code examples: An exploratory study. In *Proceedings of International Conference on Software Maintenance.* 555-558.

[17] Inoue, K., Sasaki, Y., Xia, P., and Manabe, Y. 2012. Where does this code come from and where does it go? - integrated code history tracker for open source systems. In *Proceedings of the International Conference on Software Engineering*. 331-341.

[18] Johnson, R. E. 1992. Documenting frameworks using patterns. In *Proceedings of the Conference on Object-oriented Programming Systems, Languages, and Applications.* 63-76.

[19] Kim, J., Lee, S., Hwang, S. W., and Kim, S. 2010. Towards an intelligent code search engine. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 1358- 1363.

[20] Koschke, R. 2012. Large-scale inter-system clone detection using suffix trees. In *Proceedings of European Conference on Software Maintenance and Reengineering* .309-318.

[21] Lopes, C., Bajracharya, S., Ossher, J., Baldi, P. 2010. UCI source code data sets. http://www.ics.uci.edu/~lopes/datasets. Irvine, CA: University of California, Bren School of Information and Computer Sciences.

[22] McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., and Fu, C. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the International Conference on Software Engineering*. 111-120.

[23] 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.* 40(6), 48-61.

[24] Manning, C. D., Raghavan, P., and Schütze, H. 2008. Introduction to information retrieval. Cambridge University Press.

[25] Marri, M. R., Thummalapenta, S., and Xie, T. 2009. Improving software quality via code searching and mining. In *Proceedings of ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*. 33-36.

[26] Mishne, A., Shoham, S., and Yahav, E. 2012. Typestate-based semantic code search over partial programs. In *Proceedings of International Conference on Object-oriented Programming Systems, Languages and Applications*. 997-1016.

[27] Nykaza, J., Messinger, R., Boehme, F., Norman, C. L., Mace, M., and Gordon, M. 2002. What programmers really want: results of a needs assessment for SDK documentation. In *Proceedings of the International Conference on Computer Documentation*, 133-141.

[28] Panchenko, O., Plattner, H., and Zeier, A. 2011. What do developers search for in source code and why. In *Proceedings of International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation.* 33-36.

[29] Robillard, M. P. 2009. What makes APIs hard to learn? answers from developers. *IEEE Software*, 26(6), 27-34.

[30] Robillard, M. P., Walker, R. J., and Zimmermann, T. 2010. Recommendation systems for software engineering. *IEEE Software*. 27(4), 80-86.

[31] Reiss, S. P. 2009. Semantics-based code search. In *Proceedings of International Conference on Software Engineering*. 243-253.

[32] Roy, C. K., Cordy, J. R., and Koschke, R. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*. 74(7), 470-495.

[33] Sahavechaphan, N., and Claypool, K. 2006. XSnippet: mining for sample code. In *Proceedings of Conference on Object-oriented Programming Systems, Languages, and Applications*. 413-430.

[34] Salton, G., and Yang, G. 1973. On the specification of term values in automatic indexing. *Journal of Documentation*, 29(4), 351–372.

[35] Singer, J. 1998. Practices of software maintenance. In *Proceedings of International Conference on Software Maintenance*. 139-145.

[36] Thummalapenta, S., and Xie, T. 2007. Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of International Conference on Automated Software Engineering*. 204-213.

[37] Wang, J., Yingnong, D., Hongyu, Z., Kai, C., Xie, T., and Dongmei, Z. 2013. Mining succinct and high-coverage API usage patterns from source code. In *Proceedings of Working Conference on Mining Software Repositories,* 319-328.

[38] Xie, T., and Pei, J. 2006. MAPO: mining API usages from open source repositories. In *Proceedings of International Workshop on Mining software Repositories*. 54-57.