SNIFF: A Search Engine for Java using Free-Form Queries

Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen
EECS Department, University of California, Berkeley, CA, USA.
{shaunakc,sjuvekar,ksen}@cs.berkeley.edu

Abstract. Reuse of existing libraries simplifies software development efforts. However, these libraries are often complex and reusing the APIs in the libraries involves a steep learning curve. A programmer often uses a search engine such as Google to discover code snippets involving library usage to perform a common task. A problem with search engines is that they return many pages that a programmer has to manually mine to discover the desired code. Recent research efforts have tried to address this problem by automating the generation of code snippets from user queries. However, these queries need to have type information and therefore require the user to have a partial knowledge of the APIs.
We propose a novel code search technique, called SNIFF, which retains the flexibility of performing code search in plain English, while obtaining a small set of relevant code snippets to perform the desired task. Our technique is based on the observation that the library methods that a user code calls are often well-documented. We use the documentation of the library methods to add plain English meaning to an otherwise undocumented user code. The annotated user code is then indexed for the purpose of free-form query search. Another novel contribution of our technique is that we take a type-based intersection of the candidate code snippets obtained from a query search to generate a set of small and highly relevant code snippets.
We have implemented SNIFF for Java and have performed evaluations and user studies to demonstrate the utility of SNIFF. Our evaluations show that SNIFF performed better than most of the existing online search engines as well as related tools.

1 Introduction

Java's evolution and growth over the years has greatly increased the number of APIs available at a programmer's disposal. For example, the Java Standard Library, J2SE, contains thousands of classes and more than 20,000 methods [12]. The Java APIs are often designed in a modular manner using small composable units. A programmer needs to combine these APIs using some (often complicated) sequence of classes and method calls, to correctly use a Java library. This fact, compounded with the sheer number of APIs, makes it difficult for a programmer to discover *what* APIs he/she wants and *how* to use those APIs to perform a given programming task. For example, consider a programmer who wants to read from a file and is unfamiliar with the `java.io` package. Without prior knowledge about the `java.io` package, the programmer will not only

find it difficult to discover what classes he/she needs to use, but also to figure out how to use the methods in those classes to read from a file. Even if the programmer manages to write code to read from a file after digging into the `java.io` API, the code written may not be efficient—the code may only use the class `java.io.FileReader` and ignore the use of the `java.io.BufferedReader` class, which is required for an efficient implementation.

In order to solve the above problem, a programmer generally resorts to one of the following two techniques. He/she might try to explore existing code bases and their documentations, and search for code snippets which perform the desired programming task using some APIs. This "mining" effort becomes quite tedious because existing code bases are often very large, making manual search impossible. Moreover, these code bases may not be well-documented making it difficult to locate the relevant code snippets. Some recent tools, such as Prospector [12], have tried to automate the search process; however, they assume that the programmer knows what classes and object types he/she wants to use. This may not be a realistic assumption if the programmer is unfamiliar with the class names.

Alternately, a programmer might search the web using some search engine such as Google. An advantage of using a search engine is that the programmer posts his/her query in plain English (free-form), such as "read from a file in Java" without any prior knowledge about the required packages, classes, or methods. However, the results returned by the search engines contain relevant code interspersed with irrelevant code and plain English text from the webpage. The programmer needs to determine, potentially involving further searches, what part of the returned code snippet is relevant. More recently, PARSEWeb [18] has tried to combine results from web searches with Prospector to improve the quality of search. However, the tool suffers from the same problem as Prospector—the programmer needs to know what classes and object types he/she has to use.

We propose a novel code search technique called Sniff[1][2], which retains the flexibility of performing a search in English, while obtaining small and relevant code snippets required to perform the desired task. In Sniff, a programmer issues a query expressing the programming task in English and Sniff returns a small set of relevant code snippets. For example, Sniff returns the code snippet in Table 1 for the query "read a line of text from a file".

```
FileReader fr = new FileReader(String fileName);
BufferedReader br = new BufferedReader(FileReader fr);
String line = br.readLine()
```

Table 1. Original code with no useful comments

The key idea of Sniff is *to combine API documentation with publicly available Java code*. Specifically, Sniff takes a large amount of Java source code already available on the web and annotates it by appending each statement containing a method call with the method's Javadoc description (if available). An-

[1] Sniff stands for SNIppet for Free-Form queries
[2] It "sniffs" the code database for the relevant code

notation allows SNIFF to add meaningful comments to otherwise uncommented Java files. SNIFF indexes these annotated Java files in a database. A query to SNIFF looks up this database and collects code chunks that match the query. SNIFF then performs a type-based intersection of these code chunks to retain the most relevant and common part of the code chunks. SNIFF also ranks these pruned chunks using their relevance to the query. We have defined one criterion for the relevance of the code snippets: the frequency of their occurrence in the indexed code base. Ranking in SNIFF is based on this measure of relevance.

SNIFF has several advantages:

1. SNIFF allows free-form English queries about a programming task. This eliminates the need to know the appropriate APIs beforehand.
2. SNIFF facilitates more effective code reuse by eliminating the requirement of much prior knowledge about APIs. Code reuse increases the performance and reliability of the new code.
3. Since SNIFF constructs the most relevant code snippet by performing type-based intersection of several Java code chunks, we get relatively mature and correct code.

We have developed an eclipse plugin for SNIFF and performed a user study that found that programmers could solve the reuse problems 40% faster with SNIFF than with other tools. We have also compared the performance of SNIFF with online code search engines [5, 10, 3] on a set of user queries posted on a Java developer's forum [8]. Our experiments show that SNIFF returned the most relevant code snippet as the top ranked result for about 88% of queries, whereas the online search engines returned the top result for about 50% of queries. Moreover, these results were buried inside hyperlinked source files and required substantial manual inspection to discover the exact snippets. We have also evaluated the importance of our intersection algorithm for the relevance of the snippets. Our experiments show that intersection helps in effective pruning and better ranking of the code snippets.

2 Overview

We give an overview of SNIFF using a simple example. Consider a Java programmer who is unfamiliar with the Java classes `Runtime` and `Process`. The programmer wants to execute a system command such as `ls` from inside a Java program. The required code snippet is shown in Table 2.

```
Runtime r = Runtime.getRuntime();
Process p = r.exec(String command);
```

Table 2. Executing a system command in Java

The code is relatively difficult for the programmer to infer for several reasons. First, it is hard for a programmer unfamiliar with the Java APIs to figure out that the `Runtime` class is required to get the runtime and to execute the command in that runtime. It is even harder for the programmer to discover that the methods `getRuntime` and `exec` should be called in that order to first obtain

the runtime and then to execute the command, respectively. Finally, an object of class `Process` is required to create a separate process and execute the command. In this situation, the programmer has a couple of options to discover the code snippet. (1) The programmer could search the web (e.g. using Google) by posting a query such as "execute command in Java." (2) The programmer could use an existing code synthesizer such as Prospector [12] or PARSEWeb [18]. The results returned by web search would not often give the exact code snippet, but rather some large code fragments that have the relevant statements surrounded by other statements and non-Java text. The programmer then needs to manually examine such code fragments for the exact code snippet. The problem with the code synthesizers is that the programmer needs to know the object types (e.g. `Runtime` and `Process` in this case) that he/she wants to synthesize.

Candidate Codes	**Result of Intersection**

`Runtime r = Runtime.getRuntime();`
~~`String command = "clear";`~~
`Process p = r.exec(command);`
	`Runtime r = Runtime.getRuntime();`
	`Process p = r.exec(String command);`

`Runtime r = Runtime.getRuntime();`
~~`flag = 1;`~~
~~`// flag set if branch entered`~~
`Process p = r.exec("ls");`	

Table 3. Intersection of Code Snippets

Sniff retains the flexibility of performing the search in plain English (as in web search), while obtaining small code snippets (as in Prospector or PARSEWeb) that are relevant to the query. In particular, the programmer will type the query "execute command" in Sniff and Sniff will return a small set of concise and relevant code snippets, that would execute a system command from inside Java.

Sniff works as follows: In a nutshell, Sniff takes a large amount of publicly available Java source code and indexes [2] it in a database. A query to Sniff examines this database and collects code chunks that match the query. Sniff then performs a *Java syntax-aware* intersection of these code chunks and returns a small set of concise and relevant code-snippets. Although the above steps look trivial, each step presents a number of technical challenges. For example, during the indexing phase, what should be considered as keywords? A natural answer would be to consider the words in comments and method names as keywords. However, this straightforward approach does not work, because user codes usually have very few comments and the method names do not always reflect the actual functionality of the method. For example, the code in the first column of Table 3 does not reflect that it is meant for executing a system command.

We address this indexing problem in a novel way. We have observed that although the user of the `Runtime` class does not write any comment about the purpose of the code, the `Runtime` class is well-documented and can be used to annotate the user code to help us understand the purpose of the user-written code.

For example, given the code chunk in Table 1, one can automatically annotate it with comments as shown in Table 4 by inserting the Javadoc description of a method after each statement that contains the method. For code in table 1, these Javadoc descriptions are collected from the `FileReader` and `BufferedReader` classes. This special method of annotating a user-written and possibly uncommented Java source file adds extra useful information about the user code. SNIFF subsequently indexes the annotated user code in a database for the purpose of query.

A query to SNIFF, like "execute command" searches this database and returns the consecutive lines of code from a single source class, that contain both the keywords *execute* and *command* of the query. Assume that the first column of Table 3 lists two such candidate codes returned for the query "execute command". Any such candidate contains the relevant code along with possibly irrelevant code. The irrelevant code might either contain completely irrelevant information, like initialization of some arbitrary variable (e.g. `flag = 1;` in the second code snippet in Table 3) or might contain statements like `Runtime r = Runtime.getRuntime();` which are essential for correctness of the implementation but do not match to any query keyword. Our observation is that the irrelevant statements are dissimilar across the candidates, while relevant statements are similar (syntactically identical) in almost all of them. Therefore, we perform a *type-based intersection* of these candidates to extract the relevant statements out of them. The second column of Table 3 shows the intersection of the candidates from the first column. The comments inserted by SNIFF are not shown in Table 3 for convenience. Note that intersection retains the common statements of the two candidate codes, but removes the statements specific to each individual code. This intersection step is another novel contribution of this work and distinguishes SNIFF from other code search engines. Specifically, our intersection allows SNIFF to return a concise and relevant code snippet instead of a set of code chunks containing some irrelevant statements. We will explain our intersection algorithm in section 3.

Finally, there might be multiple code snippets that achieve the same programming purpose. In order to represent all possible relevant code snippets, we perform clustering to group similar snippets together. These clusters must be meaningfully ranked so that the most relevant snippets are displayed at the top. We have observed that the most relevant snippets are also the ones that are implemented most frequently in our indexed code. Hence, we rank the clusters based on the number of constituent snippets.

3 Our Approach and Algorithm

In this section, we describe in details our approach for generating code snippets from a user query. We begin with formal definitions of user query, code chunk and code snippet.

Definition 1. *A* **user query** *is defined as a sequence* $q = (w_1, w_2, \cdots, w_n)$*, where each* w_i *is a string of characters. We call each* w_i *a keyword.*

An example of a user query is ("execute", "command").

Definition 2. *A* **code chunk** C *is defined as an ordered list of statements* $s_1; s_2; \cdots; s_n$ *that occur in contiguous lines in some Java method body. A* **code snippet** S *is defined as an ordered list of statements* $s'_1; s'_2; \cdots; s'_m$ *that occur in the same order (but not necessarily contiguous) in some Java method body. A code snippet might omit some intermediate statements from a method body.*

Note that every code chunk is a code snippet, but not vice-versa. For example, in Table 3, the first column shows the code chunks while the second column shows a code snippet. We now describe the components of SNIFF.

```
...
FileReader fr = new FileReader(fileName);    // File Reader
                                             // Creates new FileReader given file name read
BufferedReader br = new BufferedReader(fr);  // Buffered Reader
                                             // Creates character-input stream
                                             // uses input buffer specified size
...
```

Table 4. Annotated code generated by SNIFF

3.1 Preprocessing: Parsing Open-Source Java Code

We assume that our codebase has a large collection of Java source code and that these Java classes contain sufficient number of examples on how to use various common Java APIs.

Most often, these Java classes, which we will call *client classes*, contain very few user comments. SNIFF annotates this client code by adding its own comments. This annotation is performed in a novel and automatic way. It first collects the Javadoc descriptions and user comments for every class and its methods defined in the standard Java API as well as in the client code. These comments are pruned to remove common word classes like prepositions, conjunctions and articles (called *stopwords*). SNIFF also performs a preliminary natural language processing in form of *stemming* [14] on all the keywords in the comment.

SNIFF then creates a map $MethToComments: Sig \rightarrow Comment$ from each method signature Sig to its Javadoc comments $Comment$, where a method signature contains the class name containing the method, the method name, the types of the method's arguments and the return type of the method. Finally, SNIFF performs a simplifying transformation of the client code to convert it into an *intermediate representation*. The grammar for this intermediate form is given in Table 5. The purpose of this transformation is to simplify the indexing and retrieval of the code snippets in the latter stages of the system.

```
stmt ::= var = expr | var.field = expr | var.method(var*)
       | var = var.method(var*) | if (expr) goto label
expr ::= constant | var | var.field | var op var
```

Table 5. Intermediate representation for the client code

3.2 Preprocessing: Annotating and Indexing the Client Code

After the previously described transformation, SNIFF performs the actual annotation using the *MethToComment* map as follows. SNIFF parses each client Java source file. For each statement that contains a method call `var.method(var*)` with signature, say `sig`, SNIFF adds the comment *MethToComment*(`sig`) at the end of the statement. Java coding convention recommends the use of descriptive method names, where the first character of every internal word is capitalized. (e.g. `'readLine'`). Therefore, we also break the method name using these capitalizations to identify the breakpoints and add it as a part of the comment to the statement containing the method call. For example, `readLine` is broken up into `read` and `Line` and these two words are added as comments to any statement that contains the method call `readLine`. Thus, the client code in Table 1 gets converted to the commented code shown in Table 4.

SNIFF then indexes the commented code in a database. Our database schema contains two tables. The first table stores the individual statements in the client source files and the second table stores the tuples $(keyword, statementid)$. A tuple for the string *keyword* gives the primary key of the statement whose comments contain *keyword*.

3.3 Responding to a Query

Obtaining Code Chunks using Database: SNIFF takes a user query as an input and applies the same stop-word removal and stemming to it as in the previous section. This modified query will be referred to as q for the remaining portion of the discussion. q is then treated as a *bag of words*, i.e. the ordering between the keywords is ignored. We search for each keyword w_i in q in the database and retrieve the code chunks that contain all keywords w_i in q. We enhance these code chunks by adding a few lines of code located before and after the keyword-matching region in the original source file.

Returning the code chunks "as is" is not useful because (with respect to the user query) these code chunks contain both relevant statements as well as irrelevant ones. We would like to return only those statements that are relevant to the user query. A natural way to return the relevant statements would be to only return those statements whose comments contain at least one keyword. However, we observed that some of the statements that do not contain any keyword in their comments are often the useful glue among the statements containing a keyword; therefore such statements cannot be classified as irrelevant. An example of this is in Table 3. The `Runtime` class and its method does not contain the comments relevant to "execute command", but it is required for actually executing a system command as a `String`. We also observed that these relevant statements, unlike the irrelevant statements, are present in all code chunks. Therefore, to obtain the maximal relevant code snippets, we take an intersection of the code chunks obtained from the previous step.

We next give an algorithm to perform an intersection of any two code snippets. Since each code chunk is also a code snippet, the intersection operation will

apply equally well to code chunks. Every statement in the code chunks returned by the database has the form given in Table 5. We define equivalence $\sim$ of two statements recursively as follows.

- `var1 = expr1` $\sim$ `var2 = expr2` iff *typeOf*(`var1`) = *typeOf*(`var2`) and `expr1` $\sim$ `expr2`.
- `var1.field1 = expr1` $\sim$ `var2.field2 = expr2` iff *typeOf*(`var1`) = *typeOf*(`var2`), `field1` = `field2`, and `expr1` $\sim$ `expr2`.
- `var1.method1(vars1*)` $\sim$ `var2.method2(vars2*)` iff *typeOf*(`var1`) = *typeOf*(`var2`), `method1` = `method2`, and *typeOf*(`vars1*`) = *typeOf*(`vars2*`).
- `if (expr1) goto label1` $\sim$ `if (expr2) goto label2` iff `expr1` $\sim$ `expr2`.
- `label1` $\sim$ `label2`.
- `var1 op1 var1'` $\sim$ `var2 op2 var2'` iff *typeOf*(`var1`) = *typeOf*(`var2`), *typeOf*(`var1'`) = *typeOf*(`var2'`), and `op1` = `op2`.
- `var1` $\sim$ `var2` iff *typeOf*(`var1`) = *typeOf*(`var2`).
- `var1.field1` $\sim$ `var2.field2` iff *typeOf*(`var1`) = *typeOf*(`var2`) and `field1` = `field2`.

Let $S = s_1; s_2; \cdots; s_m$ and $R = r_1; r_2; \cdots; r_n$ be any two code snippets. Note that these snippets contain comments added by SNIFF. We define the intersection of S and R, denoted by $S \sqcap R$, as the *longest common subsequence* (LCS) of S and R defined as follows: The LCS of S and R is the longest subsequence $Q = s_{i_1}; s_{i_2}; \cdots; s_{i_l}$ of S such that there exists a subsequence $T = r_{j_1}; r_{j_2}; \cdots; r_{j_l}$ of R, where $1 \leq i_1 < i_2 \cdots < i_l \leq m$; $1 \leq j_1 < j_2 < \cdots < j_l \leq n$ and $s_{i_k} \sim r_{j_k}$ for all k. The LCS of two code snippets can be computed using a modification of a standard dynamic programming algorithm [4]. The complexity of the algorithm in $O(mn)$ where m and n are the sizes of the two code snippets respectively. All the code chunks (or snippets) ever handled by SNIFF are small in size (less than 10 lines of code) and hence the quadratic running time of the algorithm is not an excessive overhead.

Clustering Code Snippets: The code chunks obtained from the main database are grouped together based on their *similarity*. Informally, two code chunks are *similar* if their intersection is still relevant to the query and does not lose too much information. We next formalize this notion of relevance. We first define *validity* as a measure of similarity of two snippets.

Let $\mathcal{S} = \{S_1, S_2, \cdots, S_m\}$ be a set of code snippets and let $q = w_1 w_2 \cdots w_n$ be a user query. A code snippet S_i is said to be *q-valid* if its code and comments together contain all keywords w_j from the query q. For example, the code chunks returned from the main database in response to q are *q-valid* by definition. We will henceforth use *valid* in place of *q-valid*, when the query q is understood. Similarity of code snippets is defined below:

Definition 3. *Two code snippets S_i and S_j are defined to be **similar** if their intersection $S_i \sqcap S_j$ is **valid**.*

Informally, an initial code chunk matched to a query contains some lines of code which are relevant to the query, and others which are not. It should be noted that the relevant lines of code need not contain any of the keywords. Intersecting two code chunks or snippets attempts to remove the irrelevant lines of code, while

keeping the ones which are relevant to the query (either through the presence of keywords, or because of issues concerning correct implementation). The intuition behind the intersection operation is that the relevant lines would co-occur in both the chunks while the irrelevant lines would not match. If the common lines found by the intersection operation still contain all the keywords, then the two code chunks or snippets are similar.

However, in some cases, there might be multiple method call sequences which correctly perform the task specified by the query. In that case, two such code chunks or snippets (corresponding to different sequences) would not contain all the keywords in their intersection, as the statements in each of them will be different. Thus, these two code chunks or snippets will be dissimilar.

We cluster the code snippets based on this similarity measure. The idea behind clustering is to group similar code snippets in one cluster. The intersection of all code snippets $\{S_{i_1}, S_{i_2}, \cdots S_{i_r}\}$ in a cluster represents the most relevant code to the query, that can be extracted out of those snippets. Formally, clustering is defined as a function $\mathbb{F} : \mathcal{S} \rightarrow \{1, 2, \cdots p\}$, where $\mathcal{S}$ is a set of code snippets and $p \leq |\mathcal{S}|$, satisfying the condition that if $\mathbb{F}(S_i) = \mathbb{F}(S_j)$, then S_i and S_j are *similar*. All snippets mapping to the same integer are said to be in the same cluster, and are represented by a single snippet in the final results. The clustering procedure is detailed in Algorithm 1. The running time of this procedure is quadratic in number of candidate code chunks. Our observation is that the number of code chunks returned from the database is very small (less than 30 on an average), and hence clustering runs very fast on them.

Algorithm 1 Clustering and Ranking algorithm

Input: $\mathcal{C} = C_1, C_2, ..., C_N$, the list of code chunks returned for the query
Output: I, the set of all clusters of code chunks in $\mathcal{C}$.

```
1: I ← C
2: for all C_i ∈ I do
3:     support(C_i) = 1
4: end for
5: for all C_i ∈ I do
6:     for all C_j ∈ I, C_j ≠ C_i do
7:         C ← C_i ⊓ C_j
8:         if isValid(C) then
9:             support(C) = support(C_i) + support(C_j)
10:            I ← I ∪ {C} \ {C_i, C_j}
11:        end if
12:    end for
13: end for
14: return sort(I, support)
```

Ranking Code Snippets: After clustering, SNIFF ranks code snippets before returning them to the user. Ranking should reflect the relevance of the code snippet to the query. We observe that the most relevant code snippet to the query is generally implemented in a large number of client classes and methods, and hence is the most common way of performing the programming task required by the query. We formalize this property of code snippets by using *support* of the snippet defined as follows: For every code chunk C_i returned from the database, $support(C_i) = 1$. The support of an intersection is defined as the sum of the supports of code snippets that participate in the intersection operation. Thus $support(S_1 \sqcap S_2 \sqcap \cdots \sqcap S_k) = \sum_{j=1}^{k} support(S_j)$. That is, support represents the

number of occurrences of the snippet across client source files. Lines 3 and 9 of the Algorithm 1 initialize and update the supports of the clusters respectively. We rank the clusters based on their supports, with the clusters having higher supports receiving higher ranks.

4 Evaluation

We conducted three different experiments using Sniff to show that Sniff is effective in solving programmers' queries and to compare it against the existing tools and search engines. In our first experiment, we compared Sniff against tools such as Prospector [12] and Google Code Search(GCS) [5]. We performed controlled user experiments where a set of programming problems were given to a group of users and their performance (i.e. the time they spent to complete the tasks) using different tools was observed. In the second experiment, we compared Sniff against online code search engines like GCS, Koders and Krugle. We manually collected programming problems posted on a Java user forum [8] and converted them into natural language queries. We then posed these queries to Sniff as well as the online search engines and compared the results. We performed a third experiment to show the effectiveness of our intersection and ranking techniques.

4.1 User Study

Our user study was aimed at evaluating the usefulness of Sniff to developers for real programming tasks which involved reuse of existing APIs. We designed four programming problems and assigned them to a set of users. We had eight participants in our study. Each user was allowed to use Sniff for two of the four problems. Of the remaining two, they were allowed to use Prospector for one problem and Google Code Search Engine for the other. We assigned the problems and the tools to be used to solve them randomly to each user. We recorded users' final answers, the time they spent to complete each problem, the queries they issued, and the rank of the snippet that they used in their code.

Each user was given a brief introduction to Sniff and Prospector. In the introduction we described the tools using a short demo. There was no training phase and none of the users participating in the user study had used Sniff before. The users were graduate students who had moderate to expert programming skills in Java.

The programming problems were designed to approximate real programming tasks. The users were not given any hints about when to use Sniff (or any of the other tools/search engines). However, the tasks were designed in such a way that they involved the usage of some APIs which were not very commonly used. The motivation for this scheme was to ensure that the users would be required to search for some APIs that they did not know about. The four programming problems for the user study were as follows:

1. **Problem 1:** In Eclipse, the visual representation of the editor, i.e. the actual window we see on the editor, is called the *active editor*. The task is to retrieve this active editor from the workbench.

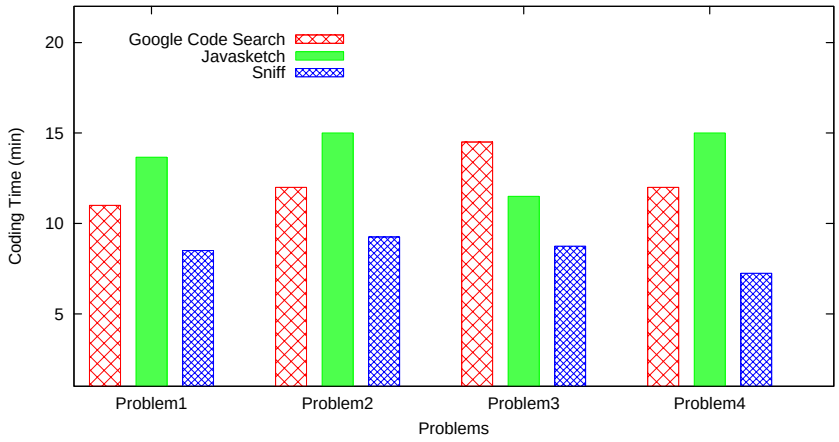

Fig. 1. Average time taken by the users on the problems using each tool

2. **Problem 2:** JDOM is a parser for parsing java source files and identifying method declarations/invocations etc. The task is to use this parser to parse a java source and create and AST.
3. **Problem 3:** JDBC is the Java technology to connect to a remote database and run SQL queries on it. The user is given the url of a remote database server. The task is to connect to the database using JDBC.
4. **Problem 4:** I have a generic Viewer object in Eclipse and I want to pop-up a dialog displaying all directories in the workspace. The task is to open such a directory dialog.

The results of the experiments are given in Figure 1. The plot in the figure gives the average time taken by the users on each problem using different tools. The plot shows that on all problems, Sniff took about 40% less time as compared to Prospector and Google Code Search. In problem 1, the users had difficulty in deducing the desired return type, `IEditorPart`; therefore, they ended up browsing the eclipse API while using Prospector. We observed similar trend in problem 3, where the desired destination class was `java.sql.Connection`. The JDOM API in problem 2 is fairly complicated and Google Code Search failed to return the specific code snippet for parsing the java file. In fact, on all queries, the results returned by Google Code Search required significant manual inspection (most of the time in the associated source code) to arrive at the desired code snippets. Sniff returned the exact code snippet, although the users needed to play around a little with their queries to arrive at this code snippet. Also, the snippets returned by Sniff required the least amount of post-processing at the users' end. Most of this post-processing was renaming variables and importing required packages, which was pretty straightforward.

4.2 Comparison with Online Search Engines

We collected random queries from the ones issued by the users in the previous section and issued them to Sniff as well as online search engines like Google Code Search [5], Koders [10], and Krugle [3]. We also collected some of the most common programming problems from frequently asked questions on a Java user

forum [8] and formulated them as natural language queries. On all of these queries, we compared the results of SNIFF and online search engines with the responses posted on the forum. Table 6 shows the rank of the most relevant snippet (as judged by a human programmer based on the forum response) using different tools. All existing search engines display a set of small code snippets in response to the query. These snippets are hyperlinked to the entire source files that contain them, and clicking on the snippets displays the contents of the files with keywords highlighted in different colors. We have manually searched these returned results and report a match if the returned source file contains the desired code snippet. A - in the table means that the desired code snippet was not present in top 10 hits from the corresponding tool. The last three lines for each tool give the percentage of queries where the most relevant code snippet is among top 1, 5 and 10 hits respectively. Our experiments show that SNIFF returned the most relevant snippet as the top ranked snippet for 87.5% of queries. The same numbers for GCS, Koders and Krugle are 25%, 62.5% and 12.5% respectively. One all the queries, SNIFF returned the most relevant code snippet in top 5 results.

Query	**Rank of the top snippet that matched the forum response**			
	GCS	**Koders**	**Krugle**	**SNIFF**
get active editor window from eclipse workbench	2	2	2	1
parse a java source and create ast	2	3	2	1
connect to a database using jdbc	2	1	-	1
display directory dialog from viewer in eclipse	-	9	2	1
read a line of text from a file	6	1	-	1
return an audio clip from url	1	1	1	1
execute SQL query	1	1	3	2
return current selection from eclipse workbench	5	1	4	1
Relevant snippet is top ranked (%)	25	62.5	12.5	87.5
Relevant snippet is in top 5 hits (%)	75	87.5	75	100
Relevant snippet is in top 10 hits (%)	87.5	100	75	100

Table 6. Results from existing code search engines on user queries.

4.3 Effectiveness of intersection and clustering techniques in Sniff

In order to show the effectiveness of our intersection and clustering algorithms, we first issued the queries given in Table 6 to SNIFF with intersection and clustering turned on. We then issued the same queries after disabling the intersection and clustering. During this phase, we ranked the snippets simply based on their sizes with the smaller snippets getting higher ranks. We compared the ranks and sizes (in LOC) of the most relevant snippets with and without intersection/clustering. We also compared the amount of pruning obtained using intersection.

The results are shown in Table 7. The first column of the table gives the queries. The next two columns give the rank of the most relevant code snippet

with and without intersection and clustering (denoted **I/C** and **No I/C**, respectively.) The next two columns give similar observations for the size of the most relevant snippet. The table shows that the rank and size of the returned snippet is better when intersection and clustering are turned on. In more than half of the cases, the most relevant snippets are poorly ranked when intersection is turned off. Intersection also reduces the size of resultant snippets by 34% on an average. Smaller size of snippet usually means less post-processing time on the returned snippets required by the users.

5 Other Related Work

A large fraction of the previous research, including Prospector [12] has focused on user queries of the form $T_{in} \rightarrow T_{out}$, where T_{in} is a source object and T_{out} is a destination object. These tools return code snippets that convert T_{in} to T_{out}, using a sequence of API method calls. PARSEWeb [18] and XSnippet [16] are two more examples in this research direction.

PARSEWeb [18] gathers the relevant code samples from GCS and performs a static analysis over them to answer the queries of type $T_{in} \rightarrow T_{out}$. They also split the query by introducing intermediate object types. The dynamic database (that of Google Code Search) together with the query splitting results in some reported improvements. XSnippet [16] makes use of context information along with a user query for finding relevant snippets. Their object instantiation queries can be classified as *type-based* (from type T_{in} to type T_{out}) or *parent-based*, where parenthood is defined by the subclass-superclass relation.

Query	**Rank**		**Size(in LOC)**	
	I/C	**No I/C**	**I/C**	**No I/C**
get active editor window from eclipse workbench	1	3	3	4
parse a java source and create ast	1	1	2	4
connect to a database using jdbc	1	6	3	6
display directory dialog from viewer in eclipse	1	1	4	7
read a line of text from a file	1	3	3	5
return an audio clip from url	1	2	6	6
execute SQL query	2	2	2	5
return current selection from eclipse workbench	1	2	2	4

Table 7. Effect of Intersection and Clustering on Results

Although there is variation in the expressiveness of queries allowed by these approaches, the basic structure of the queries is still limited to object instantiation and cannot be generalized to free-form natural language queries. Sniff does not suffer from this limitation since it allows free-form queries.

An alternate approach is the work of Murphy et al [6, 7], which locates example code files relevant to the code under development. The work focuses on structural context matching heuristics. However, this approach does not allow the user to explicitly specify a query and hence, proves to be useful only in special cases, when the code under development is very similar to some example in the repository. Often, the developer might not have enough context present in her program to be guided to an actually relevant example.

SPARS-J [13] is a closely related Java class retrieval system that applies a graph-based model to the programs. It returns a ranked set of classes to a free-form user query using a frequency-of-usage based heuristic. However, the SPARS-J technique is not aware of the functionality of a method or API beyond what is suggested by its name. Several user queries in our experiments were based on functionality. Sniff gathers much more information about the source code from its inline comments or JavaDoc specifications. Moreover, SPARSE-J returns relevant classes and it often requires time and expertise to identify the precise snippet inside a complicated class/API. Robillard et al [15] present another graph-based approach to discover the code relevant to change during code modification. They use topological structure of the program graph to propose and rank program elements that are potentially interesting to a developer modifying the source code.

The role of comments as an aid to program understanding has been a well-studied topic [19]. iComment [17] automatically analyzes comments written in natural language to extract implicit program rules and uses them to automatically detect inconsistencies between comments and source code, indicating either bugs or bad comments. However, extracting and using the information contained in comments can be hard, since comments often convey other information like directions to colleagues (especially in a group development environment) [20].

Besides comments, program invariants is another interesting direction for identifying code snippets. There has also been a lot of work on inferring specifications [1, 11]. However, we believe that in the context of code reuse and searching for relevant APIs and their usage patterns, comments are much more useful than formal specification of invariants.

Our intersection approach has some similarities with DECKARD [9]. However, Sniff performs type-based intersection only at the statement level (since we treat a program as an ordered list of statements), while DECKARD focuses on efficient algorithms for identifying similar subtrees and applying it to tree representations of the source code.

6 Conclusion

We have shown that our approach to locate relevant snippets for a free-form query has a lot of promise. Inserting API comments in the client code at the right places helps in localizing search results. Our current results for free-form queries are already better than several existing code search engines. The intersection performed by Sniff is also critical in coming up with the relevant code snippet(s), while separating out the statements that are irrelevant to the query.

Integrating and indexing search results from online search engines is an important future work on this problem. An interesting extension of current approach would be to deploy the tool in a cloud or compute cluster. Releasing the tool for a larger class of programmers will provide us more valuable feedback on usefulness of techniques used in this paper.

Acknowledgments

We would like to thank Jacob Burnim, Pallavi Joshi, Chang-Seo Park, Christos Stergiou, Raluca Sauciuc, Yamini Kannan, Nicholas Jalbert and Subhransu Maji for their valuable comments on a previous draft of this paper and for participating in the user study. This work is supported in part by the NSF Grants CNS-0720906, CCF-0747390, and CCR-0326577.

References

1. G. Ammons, R. Bodik, and James R. Larus. Mining specifications. In *POPL '02*, pages 4–16, 2002.
2. S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.
3. Krugle inc. `http://www.krugle.com`.
4. T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to algorithms*. MIT press and McGraw-Hill, 2001.
5. Google code search. `http://google.com/codesearch`.
6. R. Holmes and G. Murphy. Using structural context to recommend source code examples. In *ICSE '05*, pages 117–125, 2005.
7. R. Holmes, R. Walker, and G. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, 32(12):952–970, 2006.
8. Java frequently asked questions. `http://www.javafaq.com/`.
9. L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE '07*, pages 96–105, 2007.
10. Koders inc. `http://www.koders.com`.
11. T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: inferring the specification within. In *OSDI '06*, pages 161–176, 2006.
12. D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining : helping to navigate the api jungle. In *PLDI '05*, pages 48–61, 2005.
13. M. Matsushita, K. Inoue, R. Yokomori, T. Yamamoto, and S. Kusumoto. Ranking significance of software components based on use relations. *IEEE Trans. Softw. Eng.*, 31(3):213–225, 2005.
14. M.F. Porter. An algorithm for suffix stripping. In *Readings in information retrieval*, volume 14, pages 130 – 137, 1980.
15. Martin P. Robillard. Automatic generation of suggestions for program investigation. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 11–20, New York, NY, USA, 2005. ACM.
16. N. Sahavechaphan and K. Claypool. Xsnippet: mining for sample code. In *OOPSLA '06*, pages 413 – 430, 2006.
17. L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /*icomment: bugs or bad comments?*/. In *SOSP '07*, pages 145–158, 2007.
18. S. Thummalapenta and T. Xie. PARSEWeb : A programmer assistant for reusing open source code on the web. In *ASE '07*, pages 204 – 213, 2007.
19. S. Woodfield, H. Dunsmore, and V. Y. Shen. The effect of modularization and comments on program comprehension. In *ICSE '02*, pages 215–223, 1981.
20. A. T. T. Ying, J. L. Wright, and S. Abrams. Source code that talks: an exploration of eclipse task comments and their implication to repository mining. In *MSR '05*, pages 1–5, 2005.