to appear in Proceedings of the 4th International Workshop on Search-driven development:
Users, Infrastructure, Tools and Evaluation (SUITE'12)

# Duplication Removal for a Search-Based Recommendation System

Hidehiko Masuhara
*Graduate School of Arts and Sciences*
*University of Tokyo*
*Tokyo, Japan*
*masuhara@acm.org*

Naoya Murakami
*Graduate School of Arts and Sciences*
*University of Tokyo*
*Tokyo, Japan*
*murakami@graco.c.u-tokyo.ac.jp*

Takuya Watanabe
*Edirium K. K.*
*Japan*

***Abstract*—A search-based recommendation system looks, in the code repository, for programs that are relevant to the program being edited. Storing a large amount of open source programs into the repository will make the search results better, but also causes the code clone problem; i.e., recommending a set of program fragments that are almost idential. To tackle this problem, we propose a novel approach that ranks recommended programs by taking their "freshness" count into account. This short paper discusses the background of the problem, and illustrates the proposed algorithm.**

***Keywords*-Code recommendation system; open source programs; code clones**

## I. Introduction

Example programs are useful in remembering usages of libraries that are not so frequently used, discovering useful libraries themselves, and learning algorithms. Among sources of examples, including textbooks, tutorials, and documentation, real programs are superior in volume and variety as we can easily access many open source programs.

It is however not easy to find appropriate examples that suit to a context of a problem that the programmer wants to solve. By using a keyword-based web search, we usually have to spend a lot of time for building reasonable keywords, and for browsing down to appropriate examples.

In order to easily find appropriate examples, several tools, such as CodeBroker [1] and Strathcona [2], search program fragments that have a similar context to the one that the programmer is editing. Roughly speaking, when an example program has more similar context, the rest of the program is more likely to suggest what the programmer should do next.

The authors have also developed such a tool, called Selene, that searches similar program fragments from a repository, and displays subsequent fragments as examples [3]. A couple of characteristics of Selene that should be noted here.

- Selene shows more than one example so that the programmer can find useful information from any of them. In fact, we found that showing 6 or 7 examples works best, given a fixed area on a screen [4].

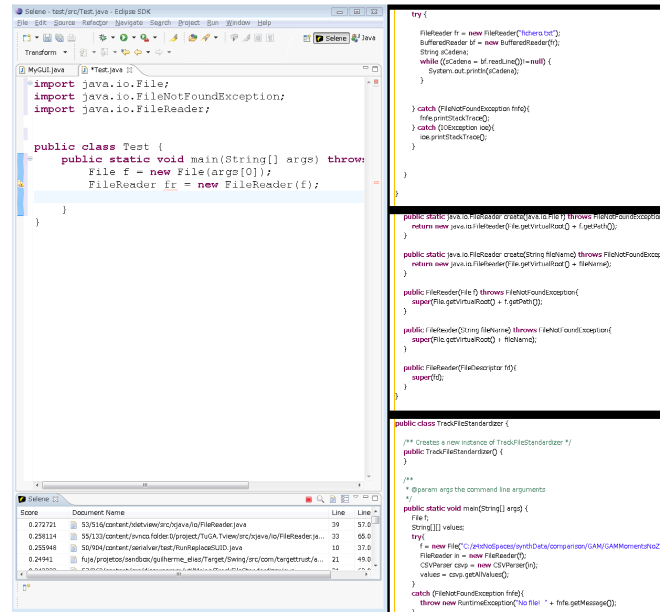


Figure 1. A Screenshot of Selene

- Selene uses a large repository of open source programs, containing more than 2 million files[1].

Figure 1 shows a screenshot of Selene, running as an Eclipse plug-in. The left hand-side is a regular programming editor. The smaller windows stacked on the right-hand side are the example programs shown by Selene.

Selene's activity consists of two search tasks. The first is search based on vector similarity of token frequencies in texts [6] that chooses the top (e.g.,) 10 files from the repository that are similar to the text in the editor. The second search chooses the top (e.g.,) 3 code fragments (approximately 10 lines of code) in those 10 files that are similar to the code fragment around the cursor position in the editor. For each of 3 code fragments, Selene displays a subsequent code fragment as an example. More detailed design and implementation of Selene can be found in the other literature [3].

[1]The open source programs in the repository were collected by the UCI Source Code Data Sets [5].

```
class Calc extends JFrame {                                   1
  Calc() {                                                    2
    JButton ok = new JButton() {                              3
      public void paint(Graphics g) {                         4
        g.setColor(Color.WHITE);                              5
        g.drawString("OK",0,0);                               6
      }                                                       7
    };                                                        8
```

Listing 1. A program written in the editor window.

```
            g2.setColor(Color.YELLOW);                        68
                g2.drawString(freemindVersion,                69
                    xCoordinate , yCoordinate);
            g2.setColor(Color.WHITE);                         70
            g2.drawString(freemindVersion,                    71
                xCoordinate+1 , yCoordinate+1);
          }                                                   72
        };                                                    73
        getContentPane().add(l, BorderLayout.                 74
            CENTER);
        pack();                                               75
        Dimension screenSize = Toolkit.                       76
            getDefaultToolkit().getScreenSize();
```

Listing 2. A code fragment from a version of FreeMindSplash.java distributed in the cpblogclient project.

Assume a programmer is writing a program as shown in listing 1. Selene then finds a code fragment that corresponds to the upper-half (until line 73) of listing 2, and displays the lower-half as an example. The programmer can then easily remember how to put the button on the frame, and other possible operations that would be performed afterward.

## II. Problem: File and Fragment Clones

Since Selene shows more than one example obtained from a large repository of open source programs, it suffers the code clone problem [7]. For example, listing 3 is another file in the Selene's repository. Even though such a file itself would be useful, it is less useful if it is shown along with a very similar file like listing 2. When we are limited in the space of showing examples, code clones would decrease chances of giving useful information to the programmer.

We here classify two types of code clones, namely *file clones* and *fragment clones*. The former are pairs of files that have entirely identical or similar texts. They appear when a repository accidentally includes multiple instances of the same project (of different versions), when a project in the repository contains the source tree of another project also in the repository, when a project adapted some files in another project, and so forth. Listings 2 and 3 are an instance of file clones. From the recorded path names, the file shown in listing 2 seems to be copied from the original project. As the original project evolves, the same file in the original project becomes as shown in listing 3.

The latter, a fragment clone, is a pair of code fragments in two files (or sometimes within one file), that have identical

```
114           g2.setColor(Color.WHITE);
115               g2.drawString(freemindVersion,
                      xCoordinate , yCoordinate);
116           g2.setColor(Color.GREEN);
117           g2.drawString(freemindVersion,
                  xCoordinate+1 , yCoordinate+1);
118         }
119       };
120       getContentPane().add(l, BorderLayout.
              CENTER);
121       mProgressBar = new JProgressBar();
122       mProgressBar.setIndeterminate(true);
123       mProgressBar.setStringPainted(true);
```

Listing 3. A code fragment from FreeMindSplash.java in the FreeMind project.

or similar texts. It is typically made by copying a code fragment from one project to another. Boilerplate code can often become fragment clones.

## III. Existing Techniques to Remove Clones

Even though there are many existing techniques for code clone detection and for clustering textual data, we cannot apply those techniques to our problem for the following reasons. Below, we discuss removing clone either *offline* (i.e., upon constructing a repository from millions of files) or *online* (i.e., when Selene retrieved about 10 files from the repository) as feasibility of those techniques largely differ with respect to the number of elements.

### A. Offline Techniques

One approach is to carefully construct a repository so that it contains no duplicated or similar file. It might be effective for a small scale repository like the one for educational purposes.

Clustering techniques such as the $k$-means algorithm or the hierarchical clustering can group texts into several similar ones. While they are proved to be useful in texts like newspaper articles, it is not clear if these techniques can be used for distinguishing similar files and clones. Moreover, those techniques usually require careful specification of parameters such as the number of clusters or the minimum distance between clusters.

Clone detection tools can visualize candidates of fragment clones for manual detection. CCFinder [8] is one of such tools that is efficient enough to scan over an entire source tree in a large project with thousands of files. However, it is not obvious that the technique can scale to millions of files.

### B. Online Techniques

When we remove clones online, we can apply rather expensive algorithms as we only need to deal with a small amount of code fragments to be shown.

However, since the number of examples is limited, it becomes more difficult to judge whether a pair of code fragments are clones or not. Assume we applied the $k$-means

algorithm to 500 code fragments extracted from the 10 files in the repository. It is not obvious for us the number of clusters that should be specified to the algorithm in advance.

Moreover, we should rank examples in terms of relative usefulness. For example, if we showed lines 74–76 in listing 2 and lines 120–123 in listing 3, the line 120 in the second one is useless as it is identical to the line 74 in the first one. However, as the rest of the lines are different each other, both code fragments can be useful if there are no other good examples.

## IV. Proposal: Ranking by Fresh Token Counts

### A. Overview

To overcome the code clone problem, we propose a novel method that ranks examples by using "freshness" in addition to existing similarity measures. Intuitively, a piece of information in an example is "fresh" if it is unknown to the programmer, and it is not provided by the other examples with higher ranks.

We here regard each token in program code as a piece of information, and assume that the tokens appeared in the editing window are not fresh (because the programmer somehow knew them to type-in). The proposed algorithm therefore displays examples, each of which has the following properties:

1) Its preceding code fragment is similar to the code fragment around the cursor position in the editor window. (*Similarity measure*)
2) It contains as many tokens that do not appear in the entire editor window and the examples that are higher-ranked than the current one. (*Freshness count*)

If we choose the bottom 5 lines of listing 2 as the first example, its freshness count is 10 as all the tokens do not appear in the editing program[2]. The bottom 5 lines of listing 3 then has the freshness count 5 because the tokens in the two lines beginning with `getContentPane()...` already appeared in the first example. As a result, if there is another example that has higher freshness count, it could be ranked higher than listing 3 regardless of a less similar preceding code fragment.

### B. The Algorithm

In order to take both the similarity measure ($s$) and the freshness count ($f$) into account, we assume a function $c(s, f)$ that combines those two measures appropriately. Though we plan to use a linear function optimized by using a mechanical evaluation method[3], it can be any function that is monotonic with respect to $f$; i.e., $f \leq f' \Rightarrow c(s, f) \leq c(s, f')$.

[2]We ignore punctuation, operator symbols and one letter tokens.

[3]The proposed method calculates an average recall ratio of tokens over a problem set that is generated from open source programs [4].

```
1//a code fragment to be shown
2class CF implements Comparable<CF> {
3  Set<Token> tokens; //set of tokens
4  float similarity;  //similarity to the query
5  float score;       //c(similarity,#fresh tokens)
6  void renew(Set<Token> appeared) {
7    score = c(similarity,
8              tokens.diff(appeared).size());
9  }
10}
11
12List<CF> rankFrags(
13    int n,               //#fragments to show
14    SortedSet<CF> cand, //list of fragments
15    Set<Token> appeared) { //set of tokens shown
16
17  List<CF> result = /*an empty list*/;
18  for (/* n times */) {
19    //the first fragment in the candidate list
20    //is the next one to be shown
21    CF f = cand.removeFirst();
22    result.add(f);
23    appeared.addAll(f.tokens);//merge f's tokens
24    SortedSet<CF> updated = /*an empty list*/;
25    //update scores of fragments in cand that
26    //have better score than the best one so far
27    do {
28      f = cand.removeFirst();
29      f.renew(appeared);
30      updated.add(f);    //elements are sorted
31    } while (cand.firstElement().score
32             > updated.firstElement().score);
33    cand.addAll(updated);//merge two sorted lists
34  }
35  return result;
36}
```

Listing 4. The algorithm that ranks code fragments based on the similarity and fresh token counts.

The ranking algorithm assumes that each code fragment is already assigned a similarity score of its preceding fragment, a combined score based on a freshness count with respect to the entire text in the editor window (i.e., no examples are shown yet).

Naively, we can easily select the top $n$ fragments by repeating the following steps.

1) Among the candidate fragments, identify the code fragment that has the highest combined score, and take it out as the next example to be shown.
2) Add tokens in the selected example into the "appeared" token set, which originally contains the tokens in the editor window.
3) Update the combined scores of all the remaining candidate fragments by excluding the appeared token set from freshness count.

The complexity of this algorithm is $O(nm)$ where $m$ is the number of candidate fragments.

Though the naive algorithm would work well as the number of shown examples ($n$) will be limited due to a screen size, we also designed a modified algorithm that avoids unnecessary recalculation of combined scores.

The modified algorithm, shown in listing 4, is written in a Java-like pseudo-language. The first 10 lines show the information assigned to each code fragment. We assume that every fragment has an initialized `score` field with respect to the tokens in the editor window.

The algorithm begins from line 12, which takes the number of fragments to be selected, a list of candidate code fragments sorted in the descending order of combined scores, and a set of tokens in the editor window. As same as the naive algorithm, (step 1) it takes the code fragment that has the highest combined score (lines 21–22), and (step 2) adds tokens into the appeared token set (line 23). Instead of updating all remaining fragments (step 3), it updates fragments that can have better score than the best one that have found so far (lines 27–32). Since the combined scores monotonically decreasing for a larger appeared token set, we do not need to update the scores of fragments that already have lower score than the best one.

The worst time complexity of the algorithm is still $O(nm)$. However, the inner loop of the modified algorithm needs to scan over many elements only when the combined score of a fragment becomes significantly smaller, which can happen with existence of many code clones. If there are few code clones, the inner loop should only scan for a few elements, thus the entire algorithm finishes quickly.

## V. Discussion

We proposed an algorithm that ranks the code fragments by taking the freshness count into account, so as to avoid the code clone problems.

The proposed algorithm is not yet implemented. Since the current implementation of Selene ranks code fragments by calculating scores of the fragments one-by-one, it would require major refactoring in the ranking engine.

Identifying duplications could also suggest about usefulness of the code fragments. As we found that copies of common libraries tend to appear in many projects, duplication could be an indicator of popularity of the code. We would be able to verify this by inspecting correlation between chances of duplication and some popularity measures.


## Acknowledgment

We would like to thank the anonymous reviewers for the insightful comments. One of the reviewers even suggested the relations between existence of code duplications and usefulness. We would like to thank Sushil Bajracharya and Cristina Lopes for their help to access the UCI Source Code Data Sets [5]. This work was partly supported as a Microsoft Research CORE Project. We are grateful to Manabu Toyama for comments on the draft version of the paper.


## References


[1] Y. Ye and G. Fischer, "Supporting reuse by delivering task-relevant and personalized information," in *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, 2002, pp. 513–523.

[2] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, 2005, pp. 117–125.

[3] T. Watanabe and H. Masuhara, "A spontaneous code recommendation tool based on associative search," in *Proceedings of the 3rd International Workshop on Search-driven development: Users, Infrastructure, Tools and Evaluation (SUITE'11)*, May 2011, pp. 17–20.

[4] N. Murakami, H. Masuhara, and T. Watanabe, "Optimizing a search-based code recommendation system," in *Proceedings of the 3rd International Workshop on Recommendation Systems for Software Engineering (RSSE'12)*, Jun. 2012, to appear.

[5] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi, "UCI source code data sets (SDS_source-repo-18k)," Apr. 2010, http://www.ics.uci.edu/~lopes/datasets/.

[6] A. Takano, "Association computation for information access," in *Proceedings of the 6th International Conference on Discovery Science*, ser. Lecture Notes in Computer Science, vol. 2843, 2003, pp. 33–44.

[7] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of the 2nd Working Conference on Reverse Engineering, (WCRE'95)*. IEEE, Jul. 1995, pp. 86–95.

[8] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic tokenbased code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, 2002.