Clustering of Functionally Equivalent Problem Variations for Representing the Similar Semantic Patterns

By
Kamaleshwar Panapakam

Supervised by
Dr. Carlos R. Rivero
Department of Computer Science
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York
May 2016
Abstract

Number of students registering in the Massive Open Online Courses (MOOCs) have increased significantly over the past few years. Several MOOCs provide feedback to the users through Unit Testing, Peer Evaluation and by using Software Verification Tools. As most the uses registering for the courses are novice users, the feedback provided by the above methods does not help the user much in understanding the mistake in the submission. This led to the development of Automated Feedback Systems which enables the instructors/tutors to provide custom feedback to a fragment in the code. The drawback of this approach is that it fails when the submitted program involves different variation of the logic.

To provide the feedback for the correct submission but with different patter variability, we must be able to identify the equivalent submissions for the submitted solution. This project aims exactly at this and provide a small set of equivalent solutions with different variations for a given submission by using a data driven approach.
1. Introduction

1.1 Problem Description

Automated Feedback Systems has gained lot of popularity recently and is there is a lot of research going on this field to provide better feedback to the users with minimal or no programming knowledge and at the same time scale to the growing number of students registering into the MOOCs.

The automated feedback systems generally store some reference solution in the form of graphs and matches it with the students submitted solution. As a problem can be solved in many number of ways, it is not a possible to store all the possible solutions beforehand.

The primary objective of this project is to identify the possible variations of a problem that is semantically equivalent to the current submitted solution form a corpus of previous solutions with the help of a set of predefined test cases. All the code fragments in this paper are extracted from LeetCode.

1.2 Background

In [1], Shasha et al, proposed a method does an approximate tree matching on the Un-ordered tree nodes. However, this approach is bounded to the syntax of the code and does not work well with the problem with different variations.

In [2], Songwen et al, proposed a method that transforms the program while maintaining the semantics. The problem with this approach is that the transformations rules are language specific and more importantly, it does not address the problem of problem variability.
2. System Overview

2.1 Matching algorithm

The matching algorithm implemented in the project is an extension to the Codewebs [3] algorithm that is implemented for Octave. It is a recursive algorithm and is implemented using Java 7 with the help of Redis. The algorithms continue to execute until the size of the equivalent set converges to some value.

2.2 Preparing Knowledgebase

2.2.1 Transforming the submissions

All the submissions in the project are represented in the form of Abstract Syntax Trees. Eclipse Java Development Tool (JDT) is used for extracting the ASTs for a program and to traverse the generated ASTs. Each node in the AST obtained has information such as nodeType, simpleName and getLineNumber etc. For example, the AST obtained for the program in Figure 1 is shown in Figure 2

For this project, only the nodeType information is considered. NodeType represents the type of code fragment that the node is associated with. NodeType is broadly classified into Expressions, Statements, BodyDeclaration, Type. More information about the nodeType can be found in Figure

<table>
<thead>
<tr>
<th>NodeType</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY_CREATION</td>
<td>3</td>
</tr>
<tr>
<td>ARRAY-INITIALIZATION</td>
<td>4</td>
</tr>
<tr>
<td>ASSIGNMENT</td>
<td>7</td>
</tr>
<tr>
<td>BREAK_STATEMENT</td>
<td>10</td>
</tr>
</tbody>
</table>

*Table 1*
simpleName attribute of the node represents the name associated with the variable or method declared in the node. For example, the node with the nodeType value 31 for the program in the Figure 1 will have a simpleName value equals to “func”.

```java
public class Solution {
    public static int func(int n) {
        int oddSum = 0;
        for(int idx=0; idx<=n; ++idx) {
            if(idx%2 != 0) {
                oddSum += idx;
            }
        }
        return oddSum;
    }
}
```

*Figure 1*
Figure 2

Figure 2.1

Figure 2.2
2.2.2 Storing the submissions

Once the AST is obtained for a submission, a Pre-Order traversal is performed on it from the root node and the is stored in a java List. While traversing each node, only the nodeType information which is of Integer type is extracted from each node and is appended to the list.
This information is persisted to the redis Hash data store. For the above AST, the traversal list obtained is Traversallist = [15, 55, 83, 42, 31, 83, 83, 39, 42, 44, 39, 42, 8, 60, 39, 59, 42, 34, 24, 58, 39, 59, 42, 34, 27, 42, 42, 38, 42, 8, 25, 27, 27, 42, 34, 34, 8, 21, 7, 42, 42, 41, 42]

This list along with the source code is persisted to the redis Hash named “all_programs” using the command hset(all_programs, Travserallist.toString(), sourceCode)

Figure 3 shows the “all_programs “data store with two submissions information.

| --- |

**public class** Solution {
    **public static int** func(int n) {
        int oddSum = 0;
        for (int i = 0; i <= n; ++i) {
            if (i % 2 != 0) {
                oddSum += i;
            }
        }
        return oddSum;
    }
}

| --- |

**public class** Solution {
    **public static int** func(int n){
        int a = 1;
        int b = 1;
        while (--n >= 1) {
            a += b;
            b = a - b;
        }
        return a;
    }
}

Figure 3

### 2.2.3 Storing Test Cases

A set of test cases are also stored in the redis hash named by the number of parameters. Each key in the hash represent the data type and the values are the list of test cases in String format. Figure 4 shows a sample test cases data store. While storing the test cases, it is advisable to not to store very large values as one badly implemented submission make the matching
algorithm take very long. Moreover, it is advisable to not to include many edge cases the test cases as we are only testing for similarity not the correct implementation

<table>
<thead>
<tr>
<th>Integer</th>
<th>{0, 34, 18.41, 11}</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>{&quot;&quot;, &quot;abc&quot;, &quot;aa&quot;, null}</td>
</tr>
</tbody>
</table>

*Figure 4*

### 2.3 Running Test Cases

JavaStringCompiler is used for executing the programs that are serialized. JavaStringCompiler is an in-memory compiler that can compile the programs stored in String format without the need to write the String program to a file which would be very inefficient. JavaStringCompiler accepts class name and source code in String format.

```java
compiler.compile(className, sourceCode)
```

Any method information in the code can be obtained using the `getDeclaredMethod()` which accepts the method name and Type of the parameters. The output of this method returns an object on which `invoke()` can be called with the parameters to get the output.

### 2.4 Search

The search operation is a partial hash match functionality that is available in the latest versions of the Redis. Redis hscan scans the entire keys for the given pattern in the hash data store in smaller iterations. On each scan, hscan function returns a cursor to the point of termination in the previous iteration. Generally, hscan starts with a cursor at zero and continues to scan until the cursor reached to zero again. A simple hscan command for the data store “all_programs” is shown below

```
Cursor = hscan(all_programs, 0, pattern)
```
3. **Assumptions**

Most of the online coding platforms provide the skeleton code for the user so that he or she can just focus on solving the problem instead of dealing with the reading and displaying the output. For almost all the problems on the online coding platforms, a student is just asked to fill a method in the skeleton code provided. Based on this, all the programs in the knowledgebase are generalized to have the same class name as “Solution” and method that students need to fill have the name “func”.

4. **Dataset**

The data set consists of submissions from *LeetCode* for the problems listed below. The problems are collected manually from various discussions.

- **GCD of two numbers**
- **Complement of given Number N**
- **Sum of Prime numbers up to N**
- **Repeatedly adding digits of a given N**
- **Sum of Odd numbers up to N**
5. Design and architecture

![Diagram of Design and Architecture]

6. Parsing the submission

The input program is transformed into AST and a traversal list is obtained similarly as explained in the section 2.2.2. But instead of selecting the entire list, the tutor/instructor can select the code fragment by specifying the line numbers. For the program in Figure 6, the user can select

```java
public class Solution {
    public static int func(int n) {
        int result = 0;
        for(int idx=1; idx < n+1;)
            if(idx%2 == 0) {
                ++idx;
            } else{
                result += idx;
                ++idx;
            }
        }
        return result;
    }
}
```

![Figure 6]
line numbers 4 to 11. The initial search pattern is the context [3] of the tree i.e. the part of the tree that is obtained after the removal of the subtree [3]. Here, the context [3] is the entire that is obtained after the subtree that from line 4 to 11. More details about the context is discussed in the later sections of the paper. The pattern is the sub-list of the entire traversal list of the tree. This sub-list represents the context obtained by removing the subtree. For the program in Figure 6, the pattern or the context is *15, 55, 83, 42, 31, 83, 83, 39, 42, 44, 39, 42, 8, 60, 39, 59, 42, 34, *, 21, 7, 42, 42, 21, 38, 42, 41, 42*

The red asterisk in the list represents the code fragment present in between line numbers entered by the tutor. This pattern serves as the initial input to the matching algorithm which will be discussed in the next section.

7. **Matching algorithm**

As discussed in the section 2.1, the matching algorithm is a recursive algorithm. The algorithm keeps adding an equivalent program fragment to a set in each run and continue to run until the size of the set stops to expand.

7.1 **Search for Context**

The algorithm starts by first searching for the context/pattern that is shown in the previous section.
The search will return a set \((\text{ContextSet})\) of submissions from the data store that matched the pattern. For each context \(c_i\) in the \(\text{ContextSet}\) a set of test cases are executed on the program associated with the context and is compared to the \(\text{ExpectedOutput}\) that is passed to the algorithm. If there is a match, then the \(c_i\) is added to the to the \(\text{EquivalentSet}\ S\). In the figures below, the colored part represents the subtree and the non-colored part of the tree represents the context of the tree. As seen in below trees, although the subtrees are different, the context is same. When the test cases match for both trees, it is safe to assume that both the colored subtrees are equivalent.

7.2 Search for Subtree

For each subtree collected in \(\text{EquivalentSet} S\), a search is performed on the data store to obtain a set of trees that contain the subtree. The pattern for the subtree search can be obtained by removing the traversal list of the context from the entire traversal list. For example, if \([15, 55, 83, 42, 31, 83, 83, 39, 42, 44, 39, 42, 8, 60, 39, 59, 42, 34, 24, 58, 39, 59, 42, 34, 27, 42, 42, 7, 42, 34, 8, 21, 7, 42, 42, 41, 42]\) is the entire list, then the subtree, the part highlighted in red represents the subtree.
The search returns a set of trees that has the searched subtree in them. For each tree obtained in the search, we repeat the steps in 7.2

8. Results

8.1 Odd Sum

For the program shown in Figure 6 which is to calculate the sum of Odd numbers up to a given N value and by selecting the code fragment between 4 and 11 when executed, the algorithm returned the below equivalent implementations.

```java
public class Solution {
    public static int func(int n) {
        int sum = 0;
        for (int i=0; i <= n; ++i) {
            if (i % 2 != 0) {
                sum += i;
            }
        }
        return sum;
    }
}

public class Solution {
    public static int func(int n) {
        int ans = 0;
        int start = 0;
        for (start = 0; start <= n; start++) {
            if (start % 2 > 0) {
                ans = ans + start;
            }
        }
        return ans;
    }
}

public class Solution {
    public static int func(int n) {
        int sum = 0;
        for (int i=0; i <= n; ++i) {
            if (i % 2 != 0) {
                sum += i;
            }
        }
        return sum;
    }
}
```
8.2 Climbing Stairs (Nth Fibonacci number)

Input: for the line numbers 5 to 8

Output

```java
public class Solution {
    public static int func( int n){
        int result=0;
        int idx=0;
        while (idx < n + 1) {
            if (idx % 2 != 0) {
                result=idx + result;
            }
            idx=idx + 1;
        }
        return result;
    }
}
```
8.3 Complement of a given N

Input: line numbers between 5 to 11

Output:

```java
public class Solution {
    public static int func( int num){
        int result=0;
        int set=1;
        while (num != 0) {
            if ((num & 1) == 1) {
                result|=set;
            }
            set<<=1;
            num>>=1;
        }
        return result;
    }
}
```
9. Conclusion

The matching algorithm found a set of equivalent programs that are closely matching to the submitted program. For the odd sum program, the algorithm identified the implementation with modulus operator and the one with incrementing the counter twice are equivalent. However, for programs that used recursion or if the program consists of only one line logic as show in Figure 8, the algorithm is returning all the solutions that are relevant to the problem instead of only returning a set of few. This is the expected behavior in some cases but is not a feasible solution for some of the problems. Future work can include the fixing this problem.

```java
public class Solution {
    public static int func(int num) {
        return num<10? num: (num%9==0? 9 : num%9);
    }
}
```

Figure 8

10. References


4. All code fragments are extracted from LeetCode.