CCFinder: A Multilingualistic Token-Based Code Clone Detection System for Large Scale Source Code

Saima Sultana Tithi
03/15/2017
Outline

• Overview
• Duplicated code detection process
• Advantages
• Results
• Discussion
About the paper

• Developed an algorithm to detect duplicated code in a system and implemented a tool named CCFinder (Code Clone Finder)
• Total citations: 1306
• Published in: IEEE Transactions on Software Engineering, Volume - 28, issue - 7
• Publication date: July 2002
• Authors:
  ▪ Toshihiro Kamiya, Osaka University, Japan
  ▪ Shinji Kusumoto, Osaka University, Japan
  ▪ Katsuro Inoue, Osaka University, Japan
Clone detecting process

Source files (input)

 Lexical Analysis

 Token Sequence

 Transformation

 Transformed Token Sequence

 Match Detection

 Formatting

 Clone-pairs (output)
Step 1: Lexical Analysis

Source files (input) → Lexical Analysis → Token Sequence → Transformation → Transformed Token Sequence → Match Detection → Formatting → Clone-pairs (output)
Step 1: Lexical Analysis

- Each line of source files is divided into tokens corresponding to a lexical rule of the programming language

<table>
<thead>
<tr>
<th>Token</th>
<th>Token Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>Identifier</td>
</tr>
<tr>
<td>=</td>
<td>Assignment operator</td>
</tr>
<tr>
<td>3</td>
<td>Integer literal</td>
</tr>
<tr>
<td>+</td>
<td>Addition operator</td>
</tr>
<tr>
<td>2</td>
<td>Integer literal</td>
</tr>
<tr>
<td>;</td>
<td>End of statement</td>
</tr>
</tbody>
</table>

```
sum = 3 + 2;
```
Step 2: Transformation

Source files (input) → Lexical Analysis → Token Sequence → Transformation → Transformed Token Sequence → Match Detection → Formatting → Clone-pairs (output)
Step 2: Transformation

• Transformation has 2 steps
  ▪ *Transformation by Transformation Rules*: The token sequence is transformed based on the transformation rules
  ▪ *Parameter replacement*: After transformation by rules, each identifier related to types, variables, and constants is replaced with a special token
Example of Transformation Rules

<table>
<thead>
<tr>
<th>Transformation Rule</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove namespace attributions</td>
<td><code>std::ios_base::hex</code>  →  <code>hex</code></td>
</tr>
<tr>
<td>Remove template parameters</td>
<td><code>vector&lt;int&gt;</code>  →  <code>vector</code></td>
</tr>
<tr>
<td>Remove accessibility keywords</td>
<td><code>protected void foo()</code>  →  <code>void foo()</code></td>
</tr>
<tr>
<td>Convert to compound block</td>
<td><code>if (a == 1) b = 2;</code>  →  <code>if (a == 1) { b = 2;}</code></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

- The authors developed transformation rules for all programming languages supported by CCFinder, which were C, C++, Java, COBOL
Step 2: Transformation

1. void print_numbers (const set<int>& s) {
2.     int c = 0;
3.     set<int>::const_iterator i = s.begin();
4.     for (; i != s.end(); ++i) {
5.         cout << c << "", "
6.         << *i << endl;
7.     } ++c;
8. }
9. }
10. void print_lines (const vector<string>& v) {
11.     int c = 0;
12.     vector<string>::const_iterator i = v.begin();
13.     for (; i != v.end(); ++i) {
14.         cout << c << "", "
15.         << *i << endl;
16.     } ++c;
17. }
18. }

Sample Code

1. void print_numbers (const set & s) {
2.     int c = 0;
3.     const_iterator i = s.begin();
4.     for (; i != s.end(); ++i) {
5.         cout << c << "", "
6.         << *i << endl;
7.     } ++c;
8. }
9. }
10. void print_lines (const vector & v) {
11.     int c = 0;
12.     const_iterator i = v.begin();
13.     for (; i != v.end(); ++i) {
14.         cout << c << "", "
15.         << *i << endl;
16.     } ++c;
17. }
18. }

Transformed code by transformation rules
Sample Code

Transformed code by transformation rules
Step 2: Transformation

1. void print_numbers (const set & s) {
2.   int c = 0;
3.   const_iterator i = s.begin();
4.   for (; i != s.end(); ++i) {
5.     cout << c <<", "
6.     << *i << endl;
7.     ++c;
8.   }
9. }
10. void print_lines (const vector & v) {
11.   int c = 0;
12.   const_iterator i = v.begin();
13.   for (; i != v.end(); ++i) {
14.     cout << c <<", "
15.     << *i << endl;
16.     ++c;
17.   }
18. }

Transformed code by transformation rules

The code after parameter replacement
Step 3: Match Detection

Source files (input)

Lexical Analysis

Token Sequence

Transformation

Transformed Token Sequence

Match Detection

Formatting

Clone-pairs (output)
Step 3: Match Detection

- Detect similar code segments based on suffix-tree matching algorithm

Transformed Token Sequence:
```
```

Create suffix tree from input sequence

Longest common subsequence:
Step 4: Formatting

Source files (input)

Lexical Analysis

Token Sequence

Transformation

Transformed Token Sequence

Match Detection

Clone Detection

Formatting

Clone-pairs (output)
Step 4: Formatting

- From the output of suffix-tree matching algorithm, all clones are converted to line numbers of the original code
- Here, line 1-9 and line 10-18 is a clone pair

```c++
1.   void print_numbers (const set<int>& s) {
2.     int c = 0;
3.     set<int>::const_iterator i = s.begin();
4.     for (; i != s.end(); ++i) {
5.         cout << c << "", "
6.         << *i << endl;
7.         ++c;
8.     }
9. }
10.  void print_lines (const vector<string>& v) {
11.     int c = 0;
12.     vector<string>::const_iterator i = v.begin();
13.     for (; i != v.end(); ++i) {
14.         cout << c << "", "
15.         << *i << endl;
16.         ++c;
17.     }
18. }
```
Advantages of using transformation step

```java
public class MultiButtonUI extends ButtonUI {
    public static ComponentUI createUI(JComponent a) {
        ComponentUI mui = new MultiButtonUI();
        return MultiLookAndFeel.createUIs(mui,
            ((MultiButtonUI)mui).uis, a);
    }
}

public class MultiColorChooserUI extends ColorChooserUI {
    public static ComponentUI createUI(JComponent a) {
        ComponentUI mui = new MultiColorChooserUI();
        return MultiLookAndFeel.createUIs(mui,
            ((MultiColorChooserUI)mui).uis, a);
    }
}
```
Advantages of using transformation step

```java
public class MultiButtonUI extends ButtonUI {
    public static ComponentUI createUI(JComponent a) {
        ComponentUI mui = new MultiButtonUI();
        return MultiLookAndFeel.createUIs(mui,
                ((MultiButtonUI)mui).uis, a);
    }
}
```

```java
public class MultiColorChooserUI extends ColorChooserUI {
    public static ComponentUI createUI(JComponent a) {
        ComponentUI mui = new MultiColorChooserUI();
        return MultiLookAndFeel.createUIs(mui,
                ((MultiColorChooserUI)mui).uis, a);
    }
}
```
Advantages of using transformation step

```c
for (int i = 0; i < n; i++) {
    if (a == 1) {
        b = 2;
    }
}
```

```c
for (int i = 0; i < n; i++) {
    if (a == 1) {
        if (a == 1)
            b = 2;
    }
}
```
Advantages of using transformation step

```java
for (int i = 0; i < n; i++) {
    if (a == 1) {
        b = 2;
    }
}
```
Implementation

• Implemented in C++
• Supports 4 programming languages: C, C++, Java, COBOL
• Time and space complexity is $O(n)$, where $n$ is total length of source file
Results

• Applied CCFinder on FreeBSD 4.0 (2.2 M lines), Linux 2.4 (2.4 M lines), NetBSD 1.5 (2.6 M lines)

• Time: 108 minutes

<table>
<thead>
<tr>
<th></th>
<th>Clone classes</th>
<th>Coverage(%LOC)</th>
<th>Coverage(%file)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FreeBSD &amp; Linux</td>
<td>1,091</td>
<td>0.8% FreeBSD 0.9% Linux</td>
<td>3.1% FreeBSD 4.6% Linux</td>
</tr>
<tr>
<td>FreeBSD &amp; NetBSD</td>
<td>25,621</td>
<td>18.6% FreeBSD 15.2% NetBSD</td>
<td>40.1% FreeBSD 36.1% NetBSD</td>
</tr>
<tr>
<td>Linux &amp; NetBSD</td>
<td>1,000</td>
<td>0.6% Linux 0.6% NetBSD</td>
<td>3.3% Linux 2.1% NetBSD</td>
</tr>
</tbody>
</table>
**Figure**: Scatter plot of clone pairs having at least 30 same tokens (about 13 lines)
Later Works

• Based on CCFinder, the authors developed another tool AIST-CCFinderX in 2005

• CCFinderX is freely available from:
  http://www.ccfinder.net/ccfinderxos.html,
  https://github.com/petersenna/ccfinderx-core

• Some other tools from the authors of CCFinder:
  • D-CCFinder (distributed CCFinder)
  • Gemini (add GUI to view the output of CCFinder)
  • Aries (refactor code based on clone detection)
  • Agec (clone detection from Java bytecode)
Discussion

• Strengths of this paper
  • Clear explanation of the method
  • Applies the tool on different code bases and shows all the results in terms of
time profile, memory profile, number of clone pairs, and percentage of clones
Discussion

• Weaknesses of this paper:
  • Did not compare CCFinder with other existing tools with respect to running time or memory consumption
  • Did not apply CCFinder on any benchmark data set and calculate the accuracy of the result
Discussion

• How to improve CCFinder?
  • To compute token sequence matching, CCFinder uses suffix-tree based matching algorithm, but suffix-tree is not space efficient for large code bases. According to the authors of SourcererCC, CCFinder runs out of memory for large code bases. As suffix-array based matching algorithm is more space efficient, instead of suffix-tree based matching algorithms, suffix-array based matching algorithm can be used.
Thank You!