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

Saima Sultana Tithi

03/15/2017

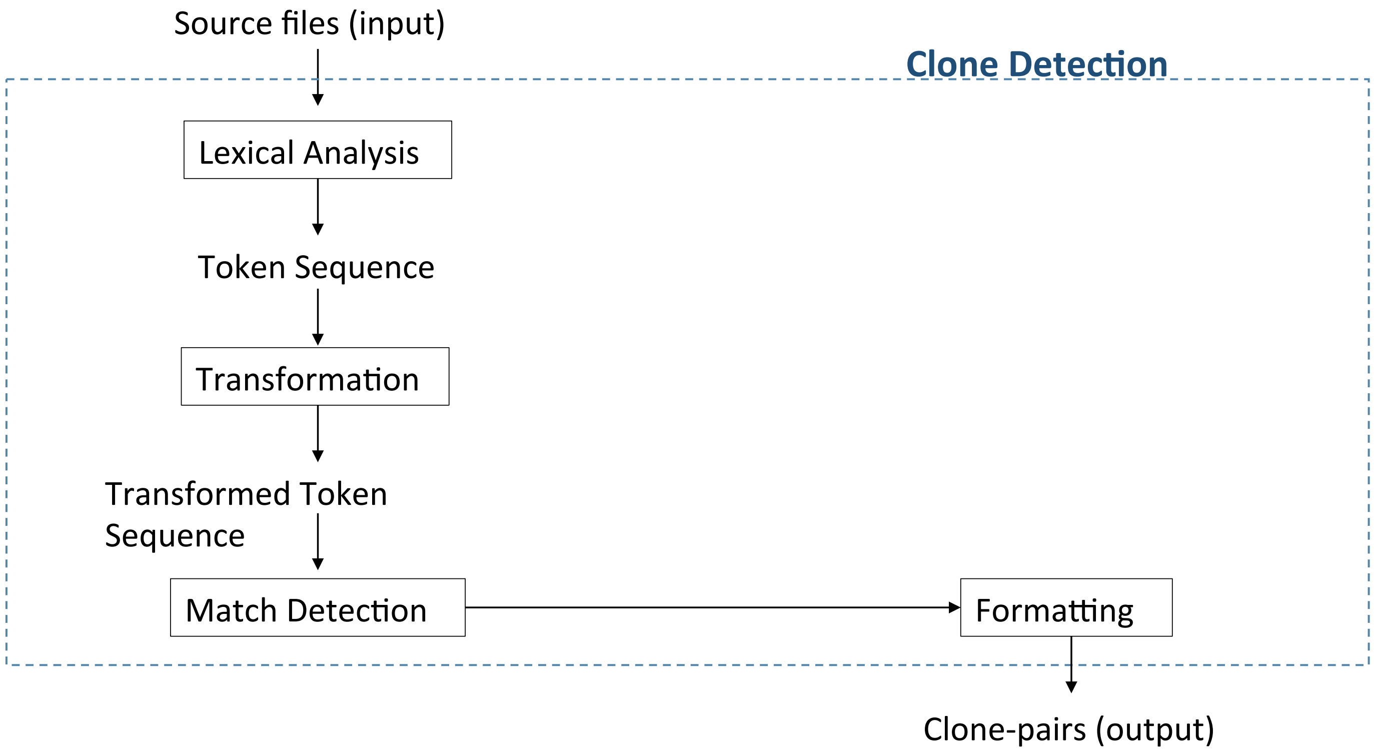
Outline

- Overview
- Duplicated code detection process
- Advantages
- Results
- Discussion

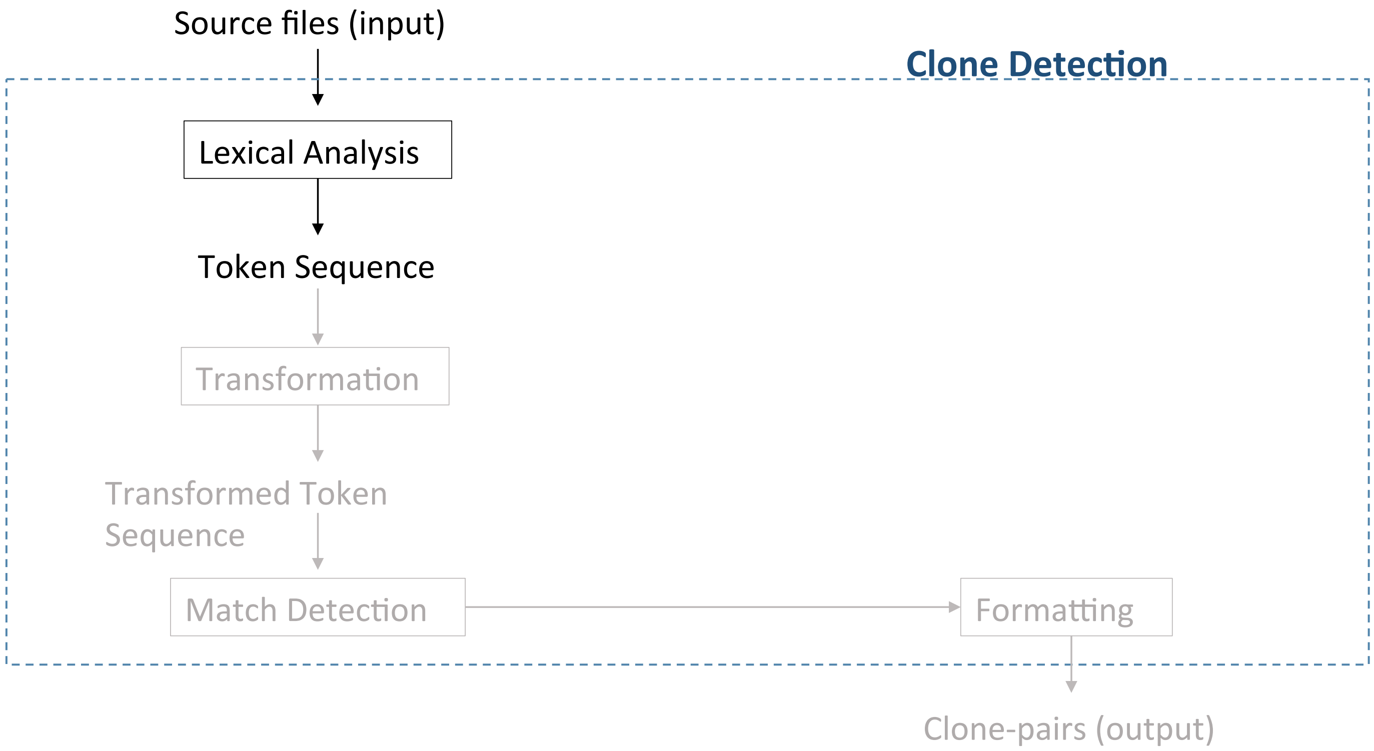
About the paper

- 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



Step 1: Lexical Analysis



Source files (input)

Clone Detection

Lexical Analysis

Token Sequence

Transformation

Transformed Token Sequence

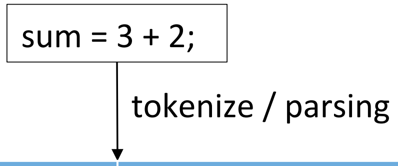
Match Detection

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

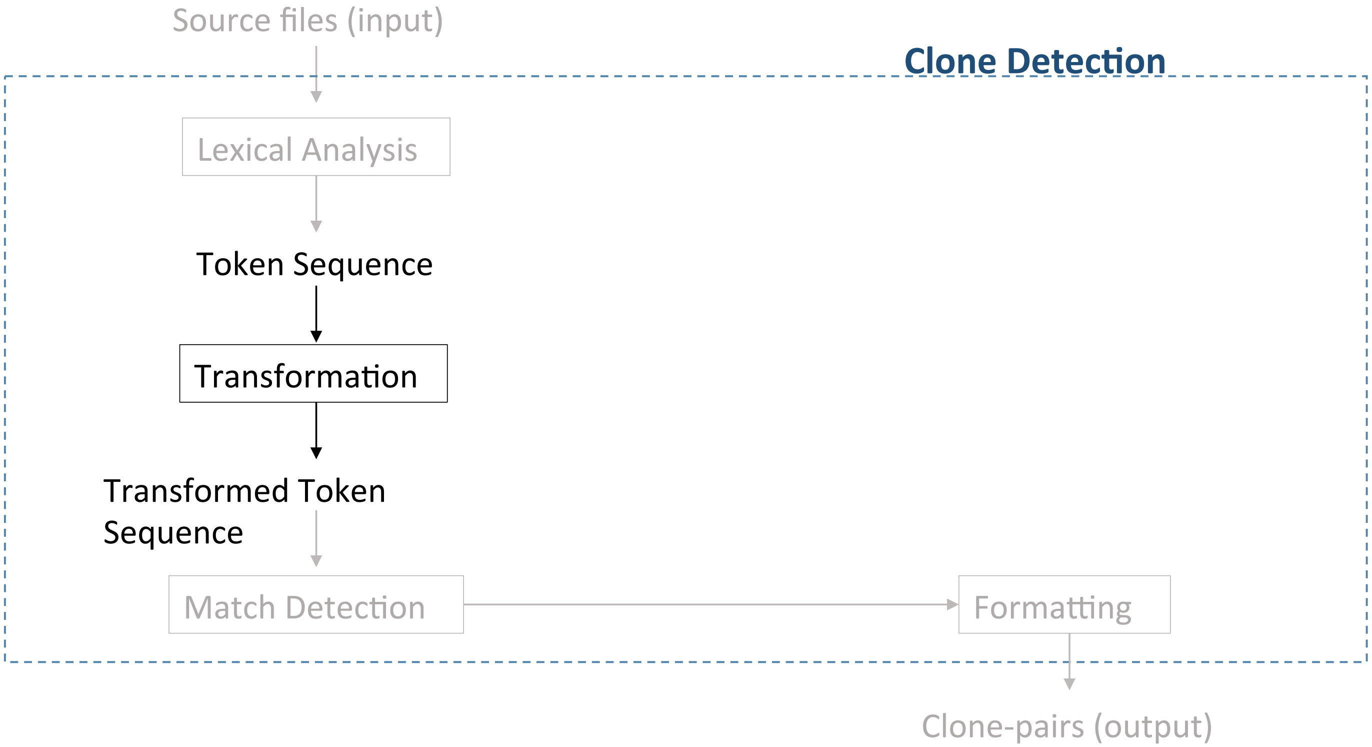


sum = 3 + 2;

tokenize / parsing

Token	Token Category
sum	Identifier
=	Assignment operator
3	Integer literal
+	Addition operator
2	Integer literal
;	End of statement

Step 2: Transformation



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

Remove namespace attributions	std::ios_base::hex → hex
Remove template parameters	vector<int> → vector
Remove accessibility keywords	protected void foo() → void foo()
Convert to compound block	if (a == 1) b = 2; → if (a == 1) { b = 2};
…	

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

Step 2: Transformation

```
void print_numbers (const set<int>& s) {
    int c = 0;
    set<int>::const_iterator i = s.begin();
    for (; i != s.end(); ++i) {
        cout << c << ", "
            << *i << endl;
        ++c;
    }
}
void print_lines (const vector<string>& v) {
    int c = 0;
    vector<string>::const_iterator i = v.begin();
    for (; i != v.end(); ++i) {
        cout << c << ", "
            << *i << endl;
        ++c;
    }
}
```

Sample Code

```
void print_numbers (const set & s) {
int c = 0;
const_iterator i = s.begin();
for (; i != s.end(); ++i) {
cout << c << ", "
<< *i << endl;
++c;
}
}
void print_lines (const vector & v) {
int c = 0;
const_iterator i = v.begin();
for (; i != v.end(); ++i) {
cout << c << ", "
<< *i << endl;
++c;
}
}
```

Transformed code by transformation rules

Step 2: Transformation

```
void print_numbers (const set<int>& s) {
    int c = 0;
    set<int>::const_iterator i = s.begin();
    for (; i != s.end(); ++i) {
        cout << c << ", "
            << *i << endl;
        ++c;
    }
}
void print_lines (const vector<string>& v) {
    int c = 0;
    vector<string>::const_iterator i = v.begin();
    for (; i != v.end(); ++i) {
        cout << c << ", "
            << *i << endl;
        ++c;
    }
}
```

Sample Code

```
void print_numbers (const set & s) {
int c = 0;
const_iterator i = s.begin();
for (; i != s.end(); ++i) {
cout << c << ", "
<< *i << endl;
++c;
}
}
void print_lines (const vector & v) {
int c = 0;
const_iterator i = v.begin();
for (; i != v.end(); ++i) {
cout << c << ", "
<< *i << endl;
++c;
}
}
```

Transformed code by transformation rules

Step 2: Transformation

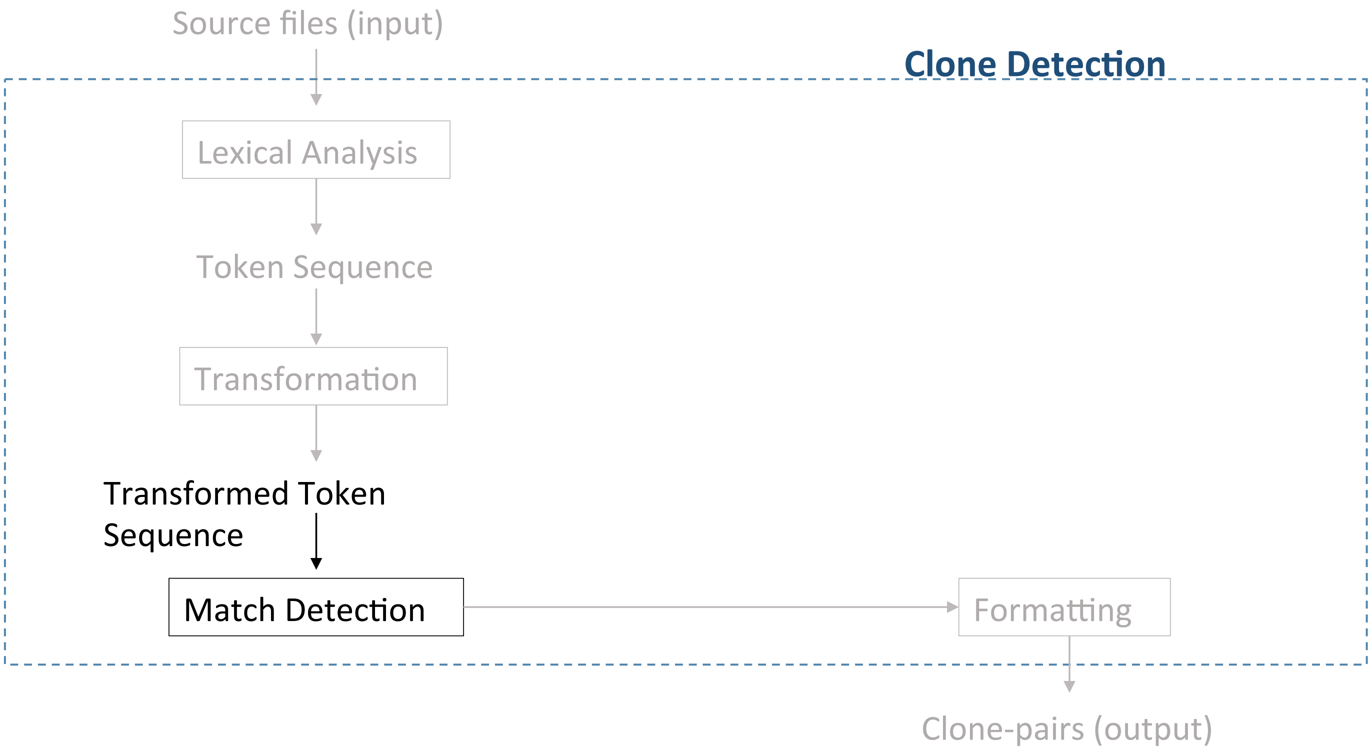
```
void print_numbers (const set & s) {
int c = 0;
const_iterator i = s.begin();
for (; i != s.end(); ++i) {
cout << c << ", "
<< *i << endl;
++c;
}
}
void print_lines (const vector & v) {
int c = 0;
const_iterator i = v.begin();
for (; i != v.end(); ++i) {
cout << c << ", "
<< *i << endl;
++c;
}
}
```

Transformed code by transformation rules

```
$p $p ($p $p & $p) {
$p $p = $p;
$p $p = $p.$p();
for (; $p != $p. $p(); ++ $p) {
$p << $p << $p
<< *$p << $p;
++ $p;
}
}
$p $p ($p $p & $p) {
$p $p = $p ;
$p $p = $p.$p();
for (; $p != $p. $p(); ++ $p) {
$p << $p << $p
<< *$p << $p;
++ $p;
}
}
```

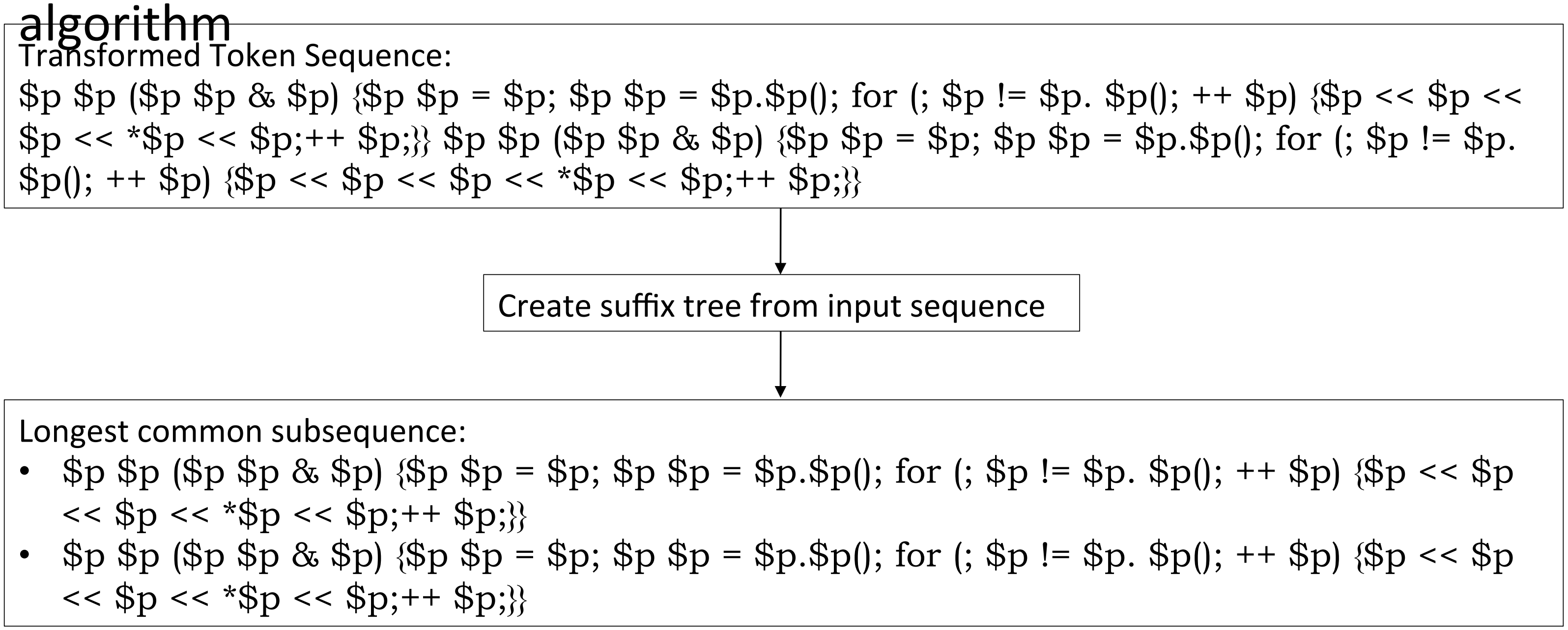
The code after parameter replacement

Step 3: Match Detection



Step 3: Match Detection

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



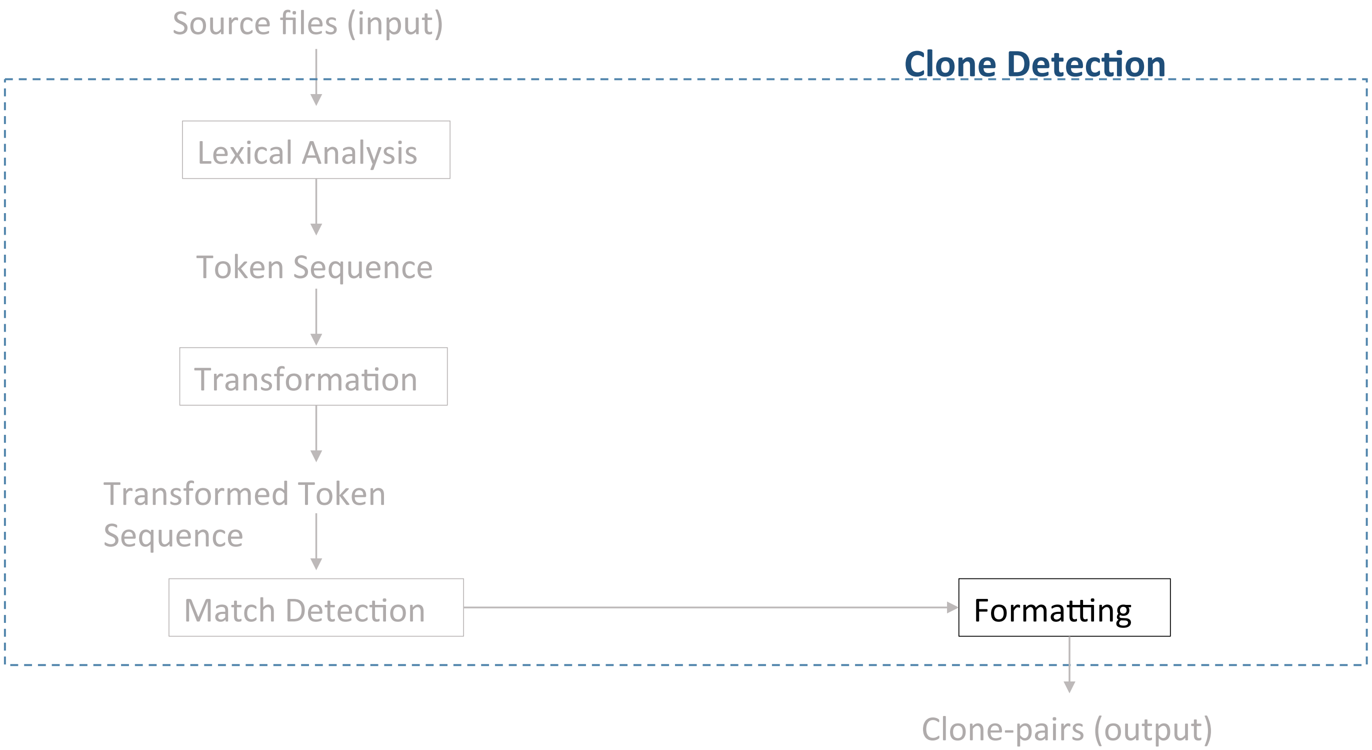
Transformed Token Sequence:
$p $p ($p $p & $p) {$p $p = $p; $p $p = $p.$p(); for (; $p != $p. $p(); ++ $p) {$p << $p << $p << *$p << $p;++ $p;}} $p $p ($p $p & $p) {$p $p = $p; $p $p = $p.$p(); for (; $p != $p. $p(); ++ $p) {$p << $p << $p << *$p << $p;++ $p;}}

Create suffix tree from input sequence

Longest common subsequence:

- $p $p ($p $p & $p) {$p $p = $p; $p $p = $p.$p(); for (; $p != $p. $p(); ++ $p) {$p << $p << $p << *$p << $p;++ $p;}}
- $p $p ($p $p & $p) {$p $p = $p; $p $p = $p.$p(); for (; $p != $p. $p(); ++ $p) {$p << $p << $p << *$p << $p;++ $p;}}

Step 4: Formatting



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

```
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

```
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

```
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

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

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

Advantages of using transformation step

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

```
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

	Clone classes	Coverage(%LOC)	Coverage(%file)
FreeBSD & Linux	1,091	0.8% FreeBSD 0.9% Linux	3.1% FreeBSD 4.6% Linux
FreeBSD & NetBSD	25,621	18.6% FreeBSD 15.2% NetBSD	40.1% FreeBSD 36.1% NetBSD
Linux & NetBSD	1,000	0.6% Linux 0.6% NetBSD	3.3% Linux 2.1% NetBSD

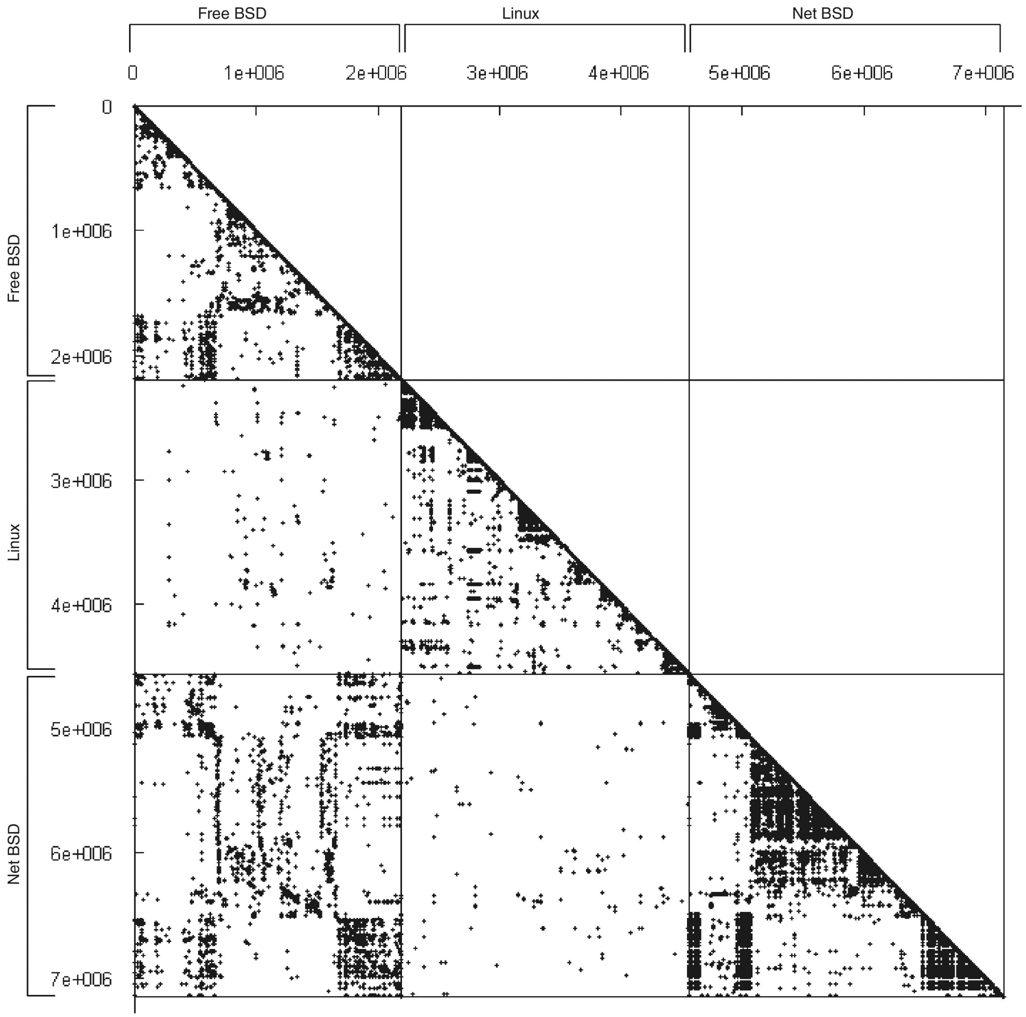


Figure: Scatter plot of clone pairs having at least 30 same tokens (about 13 lines)

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!