Capstone Project Report

Evaluation of Code Clone Detection Techniques in the Context of Computer Science Education

Raj Koushik Reddy Korpally

Advisor: Prof. Carlos R. Rivero
0. Abstract

Code clones are a common phenomenon in computer science educational scenarios. Studies comparing the usefulness of cloning tools are limited. This project aims to compare and evaluate the clone detection tools SourcererCC[5], NiCad[8], JPlag[9], ConQat[10], Deckard[7] using BigCloneBench [1] framework and a clones evaluation system designed using CodeChef programs data. The results from both the systems are analyzed, and the outputs of the individualized tools are studied.

1. Introduction

A piece of code is said to be a code clone when there is a certain similarity pattern match based on some predefined set of rules. The current techniques for code clone detection classify the clones in different categories broadly based on textual, syntactic and semantic similarities and the evaluation of developed tools and techniques is based on detection of code clones in these categories. Code clones can be intentional, unintentional, advantageous, disadvantageous, meaningful or meaningless depending on the context of usage. Several qualitative and quantitative studies exist studying, comparing and evaluating the various code clone detection tools and techniques. However, there is a lack of comparative studies evaluating the current techniques in the field of education.

The goal of the project is the evaluation of clone detection tools in computer science education using two systems, BigCloneBench[1] framework and a CodeChef programs based system. Clone detection tools use different input data formats, processing steps, pattern matching algorithms and have different outcomes. Defining and evaluating results based on common issues is a challenging task. For evaluation purposes, the testing dataset and outputs need to be tuned to match the requirements of these tools. The project aims to use programming submissions from CodeChef[4] database and BigCloneBench’s IJadataset2.0[3] for the evaluation of suitability and appropriateness of each the clone detection techniques. The methods are validated and compared based on metrics like recall, execution time, scalability and other suitable means. The overall goal is a qualitative and quantitative evaluation of the code detection techniques in the context of computer science education. Only Java-based clone evaluation is considered for the project.
2. Setting up Cloning Tools

Figure 1: Initial clone detection tools setup process

The first step in the evaluation process of the tool is to prepare the environment for the different tools that are evaluated. For that purpose, we have to satisfy different requirements needed for running of each tool. For this project, a fresh instance of Linux virtual machine using VirtualBox is created, and the clone detection tools along with its required dependencies are set up on it. The configuration for the VM is as follows:

**VM Configuration:**
VM Name: CloneProject
OS: Ubuntu 16.04 LTE
RAM: 4GB
Hard Disk Space: 30GB
Processors: 2

SourcererCC requires Java 8, Txl parser and make essentials to be installed. Txl v10.6 is installed for satisfying the purpose. NiCad also requires TXl parser, Java environment, the one that is setup for SourcererCC satisfies the purpose. The version of ConQat used for testing is an Eclipse plugin; this tool was tested on real windows environment rather than the setup virtual environment.

Several versions of JPlag exist, for the purpose of the experiments a Linux version is used which provides a greater flexibility inputting custom parameters. One drawback of JPlag is there is no Java 8 parser available for it and programs containing Java 8 code are ignored from evaluation and Deckard requires Linux packages Flex and Bison a pre-requirements.
3. General Clone Detection Process Using Tools

![Flow of Clone Detection Tools](image)

**Figure 2:** Flow of Clone Detection Tools

Any clone detection tool follows a similar generic process for detecting clones from the input data. The figure 2 shows the key steps in process.

**Code Input data:** The test java based programs data used for the project is derived from two sources. A subset of IJaDataset for BigCloneBench evaluations and a CodeChef programs dataset of 1258 code submissions from 154 users for two code problems. The CodeChef programs are derived through a web crawler (the process of extraction is not a part of this project). This common input data is used by all the tools for evaluation.

**Preprocessing Steps:** This step involves converting the input code into the format required for the tool’s input, splitting the code into fragments based on granularity set like functions or block fragments, removing extra spaces and comments.

**Code Transformations:** Based on the clone detection tool transformations are applied like conversion of code into tokens, AST’s, pieces of comparable fragments. Tools txl, flex help to achieve these steps.

**Clone Matching Algorithm:** After the preprocessing and transformation steps a matching algorithm is applied to detect the matched fragments in the code. The table below contains a
summary of clone algorithm used by the tools that are evaluated for the project. The full details can be found in the respective research papers accompanying the tools.

<table>
<thead>
<tr>
<th>Core algorithm used for clone matching by each tool</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SourcercerCC</strong></td>
</tr>
<tr>
<td><strong>NiCad</strong></td>
</tr>
<tr>
<td><strong>Deckard</strong></td>
</tr>
<tr>
<td><strong>ConQat</strong></td>
</tr>
<tr>
<td><strong>JPlag</strong></td>
</tr>
</tbody>
</table>

**Figure 3:** Clone tool and match algorithm used

**Filtering Irrelevant Clones:** Clone detection tools generate irrelevant data that needs to be filtered out. These filters are passed as parameters in the configuration for the tool. Example of filters include filters on type of clones, cutoff similarity level, duplicates and false positives removal.

**Formatting:** The detected clones are normally as transformed fragments, conversion into readable formatting is done. Clone aggregation into classes, converting clone collection into xml, html or text format is common step for the tools.

**Output:** Depending on the clone tool, the clones are output as pairs with line numbers, source files path, similarity level, node numbers or other relevant attributes. Tools are output raw data which can be used to gather more information for further processing. This generated output from the tools is further processed and used in the evaluation process that is described in the further sections.

### 4. Clone Types

There is no standard agreed upon definition for what is a clone and various categories of clones. Most literature on clone studies agree on categorization of clones into four types. Code clones are categorized into four types based on the differences found, the study[6] defines the following types of clones in the below Figure 4. We use these definitions for studying the clone detection tools in our project and calculate the performance of tools based on detection these types.
Targeted clone types

<table>
<thead>
<tr>
<th>Type 1</th>
<th>Exact copy with only differences in whitespace and comments.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 2</td>
<td>Variable renaming + any Type 1 differences</td>
</tr>
<tr>
<td>Type 3</td>
<td>Changing or adding few statements + any Type 2 differences</td>
</tr>
<tr>
<td>Type 4</td>
<td>Semantically identical codes, but not necessarily same syntax</td>
</tr>
</tbody>
</table>

Figure 4: Clone types and definition

4.1 Clone Detection Methods by Tools

Deckard

Clone tool Deckard uses AST creation followed by Vector Matching mechanisms to detect the clones.

The following are the steps used in the process of clone detection:
1. Creating AST’s for the three sample programs (here non-standard AST representation is used to remove unnecessary syntax reducing the size of AST to readable length.)
2. Generating characteristic vector for nodes in the generated AST
   The contents of the vector are the count values of useful labeled nodes (useless nodes like braces are ignored) that occur in the sub trees for the particular node.
   Example Vector’s content fields are specified as follows (not limited length, can be any number)
   <0,0,0,0,0,0,0,0,0,0>

<table>
<thead>
<tr>
<th>id</th>
<th>assignment</th>
<th>operation</th>
<th>array</th>
<th>condition</th>
<th>exp</th>
<th>declaration</th>
<th>While loop</th>
<th>if</th>
<th>literal</th>
</tr>
</thead>
</table>
3. Merging the individual node vectors to generate super vectors using post order traversal
4. Comparing merged vectors using hamming distance metric, thresholds and filtering steps to scale the detection mechanism and avoid useless clones

Clone Detection using example:
Source of programs [1] [12]
Creating AST for Program 1, Program 2, Program 3 with vector characteristics

Program 1 AST with node vectors

Program 2 AST with node vectors
Program 3 AST with node vectors

The above are the generated AST for the three programs. <7,1,1,0,2,0,0,0,0,1,1,0> for node While loop in the AST for program 1 implies that the subtrees of the while loop have 7 id tokens, 1 assignment, 1 operation, 0 array, 1 condition, 0 exp, 0 declarations, 1 while loop, 1 if loop and 0 literals. This is derived by post order merging of the subtree vectors.

The aim of Deckard's algorithm is to compare vector representation of subtrees instead of direct tree comparison as direct tree comparison is computation intensive task. After subgraph vectors are derived they are compared using certain threshold and filtering conditions.

We can see that the vectors generated at the While node in program 1 and program 2 are same and hence said to be clones.

The vector at while loop in the Program 3 significantly differs from the two programs and is not a clone.

Portions of the code are clustered compared by sliding window merging of vectors and applying comparison threshold parameters like fixing token numbers matched Hamming Distance metric is used compare the vectors. It can be described as the number of changes required for one vector to be transformed to another vector. If the transformation changes are less than a certain set threshold value the subtree is detected as a positive clone.

To avoid self matching of child subtree with parent subtrees filtering mechanisms are applied.

There might be small subtree matches like operation node <2,0,1,0,0,0,0,0,0,0,0> in all three programs but setting threshold > 8 tokens for examples filters small fragment code matches that are useless clones. The vectors are searched using Locality sensitive algorithm for faster detection of matches.

Positive output as shown by the tool output for Program 1 and Program 2:

```
000000001 dist:1.0 FILE ../../../program1.java LINE:2:11 NODE_KIND:182 nVARs:3 NUM_NODE:128 TBID:0 TEID:47
```
SourcererCC follows the method of tokenization and indexing to detect clones in an efficient manner. Considering the programs example:

Source of programs [1][12]:

```
Program 1
public static int gcd(int a, int b) {
    while (a != b) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    } return a;
}
```

```
Program 2
public static int gcd(int x, int y) {
    // added comment
    while (x <= y) {
        if (x < y) {
            y = y - x;
        } else {
            x = x - y;
        }
    } return x;
}
```

```
Program 3
public static int gcd(int a, int b) {
    // semantically alternative implementation of gcd
    while (b != 0) {
        int k = a;
        a = b;
        b = k % b;
    } return a;
}
```

**Tokenization output of Program 1:**

```
0,0#@a@@::@@7,b@@::@@6,static@@::@@1,gcd@@::@@1,while@@::@@1,int@@::@@3,public@@::@@1,else@@::@@1,if@@::@@1,return@@::@@1
```

**Tokenization output of Program 2:**

```
1,1#@x@@::@@7,y@@::@@6,static@@::@@1,gcd@@::@@1,while@@::@@1,int@@::@@3,public@@::@@1,else@@::@@1,if@@::@@1,return@@::@@1
```

**Tokenization output of Program 3:**

```
2,2#@a@@::@@4,b@@::@@5,static@@::@@1,gcd@@::@@1,while@@::@@1,int@@::@@4,0@@::@@1,public@@::@@1,t@@::@@2,return@@::@@1
```

**Understanding Tokenization comparison for clone detection:**

First sourcererCC splits the program into small chunks with indexes, they can be blocks or function level granularity. Tokens are calculated in the blocks. The tokens are represented as tuple pair with (name, frequency) format.

In the tokenized output generated by the program 1, the first part is the (parentid, blockid) pair. Pair (0,0) represents that the parent of of the block is file 0 which here is program 1 and block 0 is the first block in the program 1 and the tokenized output is of that part.

And every token is followed by its frequency after the delimiter, static@@::@@1 implies static occurs once in the block.

Clone is detected based on the percentage match between all generated tokens of the blocks.
The tokenized output of program1 is almost identical with program 2 with some identifier renaming and is detected as clone, but output of program 3 doesn’t cross the threshold to be considered as clone. 
Brute force method would be comparing all the tokens of each block with every other block and the complexity would be O(n^2). SourcererCC tokens comparison algorithm reduces the complexity order by using two heuristic improvements by reducing subblocks comparison and creating indexing of important tokens only. 
Another important aspect of token generation step is the ordering of tokens is not the natural ordering of input code but follows a priority token based approach to minimize token comparisons at a later stage of clone matching and also help in the detection of rearranged clones easily.

**JPLag**

JPLag too follows a token based comparison for clone detection like SourcererCC, but the key difference is the logic of comparison algorithm and the rules for token generation arrangement and comparison. The rules for the tokenizer are not clear and assumptions are made while parsing the example programs 1,2,3. 
JPlag uses “Greedy Tile Matching”[11] Algorithm to compare the tokens for clone detection. 
The Tokenization step of the programs can be imagined as follows (approximate rules applies). 
Tokens follow natural order of the code and are not rearranged(unlike SourcererCC where priority based rearranging is done) 
Using same programs from the previous example:

**Program 1 Tokenization:**

```
BEGIN_CLASS,
RETURN_TYPE,BEGIN_METHOD,PARAMETER,PARAMETER,BEGIN_WHILE,IDENTIFIER,COMPARE,IDENTIFIER,COMPARE,IDENTIFIER,BEGIN_IF,IDENTIFIER,ASSIGN,IDENTIFIER,OPERATOR,IDENTIFIER,END_IF,BEGIN_ELSE,IDENTIFIER,ASSIGN,IDENTIFIER,OPERATOR,IDENTIFIER,END_ELSE,END_WHILE,RETURN,IDENTIFIER
END_CLASS
```

**Program 2 Tokenization:**

```
BEGIN_CLASS,
RETURN_TYPE,BEGIN_METHOD,PARAMETER,PARAMETER,BEGIN_WHILE,IDENTIFIER,COMPARE,IDENTIFIER,COMPARE,IDENTIFIER,BEGIN_IF,IDENTIFIER,ASSIGN,IDENTIFIER,OPERATOR,IDENTIFIER,END_IF,BEGIN_ELSE,IDENTIFIER,ASSIGN,IDENTIFIER,OPERATOR,IDENTIFIER,END_ELSE,END_WHILE,RETURN,IDENTIFIER
END_CLASS
```

**Program 3 Tokenization:**

```
BEGIN_CLASS,
RETURN_TYPE,BEGIN_METHOD,PARAMETER,PARAMETER,BEGIN_WHILE,IDENTIFIER,COMPARE
```
The tokens are compared using a repeated two part process. If Program 1 and Program 2 are compared, in first round each token of program 1 is compared with the each token of program 2 to find the longest matching length token strings matched. In second part of the process longest matches are identified and tokens are marked. The process is continued by decreasing max length of match until certain set threshold is reached when there is no match found. The process identifies program 1 and program 2 as match in just one round and all tokens are set and no further process takes place. When comparing program 1 and program 3 the longest matches can seen as 
“BEGIN_CLASS,RETURN_TYPE,BEGIN_METHOD,PARAMETER,PARAMETER,BEGIN_WHILE,IDENTIFIER,COMPARE”, but it fails the set lower threshold limit assuming min 10 tokens match, so it is not a clone as justified by the output.

One problem with JPlag is it can’t identify repeated clones in the programs and each token is only matched once.

Output by JPlag:
Comparing program1.java, Program2.java: 100.0
Comparing program2.java, program3.java: 0.0
Comparing program1.java, program3.java: 0.0

NiCad
NiCad tool uses text comparison algorithm with pre-processing steps like pretty printing, code normalization, gap analysis and using longest common substring to identify the code clones.
Example applying NiCad’s clone detection process using the program 1 and 2 as examples:
1. **Applying Pretty Printing** (remove comments, apply/remove missing braces, spaces)

2. **Applying Code Normalization** (Txl token normalizations applied: only for assignments)

3. **Flexible printing** (Split code into individual lines for improving matching probability) and text line comparison, Applying LCS algorithm (green match and red mismatch)
NiCad tool doesn't detect program 1 and program 2 as clones (NiCad even after normalizations bad tool for renaming changes, even though the end structure similar text comparison hinders clone identification), where as Deckard, SourcererCC, JPlag detect them as clones

**ConQat**

The details of the algorithm for match detection used for ConQat tool is not clearly explained but involves suffix trees and token comparison [10]. The use of suffix trees by the ConQat tool is to detect substring matching like finding LCS or substring matches faster between code pieces. The below example gives the abstract view of ConQat detection process:

First ConQat applies formatting and basic tokenization on the input. The high level tokenized output for the conditional statements inside the while loop for the three programs as:

**Tokens Program 1 inside While { }**:  
If | id | comparator | id | id | assignment | id | else | id | assignment | id |  
1 | 2 | 3 | 2 | 2 | 4 | 2 | 5 | 2 | 4 | 2 |  

**Tokens Program 2 inside While { }**:  
If | id | comparator | id | id | assignment | id | else | id | assignment | id |  
1 | 2 | 3 | 2 | 2 | 4 | 2 | 5 | 2 | 4 | 2 |  

**Tokens Program 3 inside While{ }**:  
If | id | comparator | id | id | assignment | id | else | id | assignment | id |  
1 | 2 | 3 | 2 | 2 | 4 | 2 | 5 | 2 | 4 | 2 |
Constructing a Suffix Tree for Program 1’s if else block using token labels id’s (i.e tree for pattern: 1,2,3,2,2,4,2,5,2,4,2)

Using this suffix tree we can check for substrings from other codes to find clones or merge two suffix trees and do a DFS to find the a LCS match. The LCS match or the substring match is the identifies the clones. For example the complete tokenized ‘if else’ output of program 2 is a substring match when traveled to node 1 from node 0 of suffix tree. So fragment extracts from program 1 and 2 are clones. Whereas program 3’s ‘inside while’ tokens output doesn’t fully exist in the suffix tree, so the fragment is not a clone. But LCS (partial match) can be calculated by common suffix tree between program 1 and program 3 (for inside while loop) and it can be matched as (4,4,2) or (id, assignment, id) which is too small a match to be considered a clone.

The output shown by ConQat:
5. Evaluating Tools

5.1 BigCloneBench Methodology

Overview

BigCloneBench has an extensive collection of manually validated clones from the IJaDataset, and the framework evaluates the clone detection tool based their ability on detecting clones from this reference set. The validated clones dataset contains different variations of various functions like BubbleSort, Hashing Function, Compression Algorithm, Copy function, etc. The final report includes evaluation metrics like recall values for these functions. Due to the enormous size of the IJaDataset, using the complete dataset is not scalable by all tools, the dataset is reduced to a randomized subset of reference clones, and result comparison is made for the subset results.

Required final evaluation input by should be in a csv file in the format [1]:

SubDirectory_Clone_1,FileName_Clone_1,StartLine_Clone_1,EndLineClone_1,SubDirectory_Clone_2,FileName_Clone_2,StartLine_Clone_2,EndLineClone_2.[1]

Setting up BigCloneBench

- Setup the BigCloneBench git project in the prepared Linux machine environment
- Download the the testing IJaDataset (Java reference clones dataset)
- Setup the validation clones database

Benchmarking Steps:

BigCloneBench evaluation works in five steps[1]:

1. Register the tool to be tested to get a tool id number
2. Run the IJadataset on the cloning tool in different configurations.
3. Processing and converting the clones detected to the input format expected by the benchmark.
4. Import formatted clones to the BigCloneBench project.
5. Validate the identified clones with the comparison database in the BigCloneBench and generate a report

<table>
<thead>
<tr>
<th>FUNCTION_ID_ONE</th>
<th>FUNCTION_ID_TWO</th>
<th>TYPE</th>
<th>FILE1</th>
<th>STARTLINE</th>
<th>ENDLINE</th>
<th>TYPE</th>
<th>FILE2</th>
<th>STARTLINE</th>
<th>ENDLINE</th>
<th>SYNTACTIC_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>17321013</td>
<td>19293133</td>
<td>selected</td>
<td>121224.java</td>
<td>56</td>
<td>73</td>
<td>selected</td>
<td>260262.java</td>
<td>27</td>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>17321013</td>
<td>19565950</td>
<td>selected</td>
<td>121224.java</td>
<td>56</td>
<td>73</td>
<td>selected</td>
<td>2437000.java</td>
<td>23</td>
<td>40</td>
<td>3</td>
</tr>
<tr>
<td>41540</td>
<td>654194</td>
<td>default</td>
<td>59964.java</td>
<td>3</td>
<td>21</td>
<td>default</td>
<td>133373.java</td>
<td>3</td>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>41540</td>
<td>389153</td>
<td>default</td>
<td>59964.java</td>
<td>3</td>
<td>21</td>
<td>default</td>
<td>117286.java</td>
<td>3</td>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>41540</td>
<td>45708</td>
<td>default</td>
<td>59964.java</td>
<td>3</td>
<td>21</td>
<td>default</td>
<td>87996.java</td>
<td>3</td>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>356445</td>
<td>654194</td>
<td>default</td>
<td>1501.java</td>
<td>3</td>
<td>21</td>
<td>default</td>
<td>133373.java</td>
<td>3</td>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>356445</td>
<td>389153</td>
<td>default</td>
<td>1501.java</td>
<td>3</td>
<td>21</td>
<td>default</td>
<td>117286.java</td>
<td>3</td>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>45708</td>
<td>368445</td>
<td>default</td>
<td>87199.java</td>
<td>3</td>
<td>21</td>
<td>default</td>
<td>1501.java</td>
<td>3</td>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>6519458</td>
<td>16934481</td>
<td>selected</td>
<td>2594341.java</td>
<td>42</td>
<td>54</td>
<td>selected</td>
<td>753634.java</td>
<td>42</td>
<td>54</td>
<td>3</td>
</tr>
<tr>
<td>2486455</td>
<td>23077160</td>
<td>selected</td>
<td>2514456.java</td>
<td>9</td>
<td>17</td>
<td>sample</td>
<td>BinarySearch.java</td>
<td>32</td>
<td>40</td>
<td>3</td>
</tr>
<tr>
<td>6870108</td>
<td>23077160</td>
<td>selected</td>
<td>229710.java</td>
<td>10</td>
<td>18</td>
<td>sample</td>
<td>BinarySearch.java</td>
<td>32</td>
<td>46</td>
<td>3</td>
</tr>
<tr>
<td>7915389</td>
<td>23077160</td>
<td>selected</td>
<td>2004523.java</td>
<td>86</td>
<td>94</td>
<td>selected</td>
<td>595260.java</td>
<td>80</td>
<td>87</td>
<td>3</td>
</tr>
<tr>
<td>7916231</td>
<td>13704394</td>
<td>selected</td>
<td>2370600.java</td>
<td>164</td>
<td>120</td>
<td>selected</td>
<td>1461211.java</td>
<td>416</td>
<td>432</td>
<td>3</td>
</tr>
<tr>
<td>1703231</td>
<td>1799224</td>
<td>selected</td>
<td>2370600.java</td>
<td>164</td>
<td>120</td>
<td>selected</td>
<td>2279650.java</td>
<td>156</td>
<td>174</td>
<td>3</td>
</tr>
<tr>
<td>13704386</td>
<td>13704387</td>
<td>selected</td>
<td>1461211.java</td>
<td>193</td>
<td>208</td>
<td>sample</td>
<td>BinarySearch.java</td>
<td>32</td>
<td>46</td>
<td>3</td>
</tr>
<tr>
<td>1799239</td>
<td>1799231</td>
<td>selected</td>
<td>2370600.java</td>
<td>86</td>
<td>102</td>
<td>selected</td>
<td>2379600.java</td>
<td>154</td>
<td>120</td>
<td>3</td>
</tr>
<tr>
<td>1799231</td>
<td>13704390</td>
<td>selected</td>
<td>2370600.java</td>
<td>164</td>
<td>120</td>
<td>sample</td>
<td>BinarySearch.java</td>
<td>32</td>
<td>46</td>
<td>3</td>
</tr>
<tr>
<td>7555749</td>
<td>10193605</td>
<td>selected</td>
<td>262852.java</td>
<td>114</td>
<td>134</td>
<td>selected</td>
<td>2089300.java</td>
<td>110</td>
<td>128</td>
<td>3</td>
</tr>
<tr>
<td>7555749</td>
<td>10193607</td>
<td>selected</td>
<td>262852.java</td>
<td>114</td>
<td>134</td>
<td>selected</td>
<td>2289390.java</td>
<td>163</td>
<td>181</td>
<td>3</td>
</tr>
<tr>
<td>5631483</td>
<td>16553114</td>
<td>selected</td>
<td>2480444.java</td>
<td>84</td>
<td>70</td>
<td>selected</td>
<td>873222.java</td>
<td>70</td>
<td>65</td>
<td>3</td>
</tr>
<tr>
<td>1799228</td>
<td>13704387</td>
<td>selected</td>
<td>2370600.java</td>
<td>51</td>
<td>66</td>
<td>selected</td>
<td>1461211.java</td>
<td>220</td>
<td>235</td>
<td>3</td>
</tr>
<tr>
<td>13704387</td>
<td>13704388</td>
<td>selected</td>
<td>1461211.java</td>
<td>220</td>
<td>235</td>
<td>selected</td>
<td>1461211.java</td>
<td>246</td>
<td>263</td>
<td>3</td>
</tr>
<tr>
<td>1561438</td>
<td>11919032</td>
<td>selected</td>
<td>2465371.java</td>
<td>174</td>
<td>191</td>
<td>selected</td>
<td>1907234.java</td>
<td>1225</td>
<td>1242</td>
<td>3</td>
</tr>
<tr>
<td>11919030</td>
<td>11919032</td>
<td>selected</td>
<td>1907234.java</td>
<td>1182</td>
<td>1199</td>
<td>selected</td>
<td>1907234.java</td>
<td>1225</td>
<td>1242</td>
<td>3</td>
</tr>
</tbody>
</table>

**Figure 5:** BigCloneBench Reference Clone Pairs in H2 database

### 5.1.1 Tool Experiments

**SourcererCC**

SourcererCC is run on IJadataset with various configuration settings. SourcererCC operates in three steps:

1. **Tokenization:** From SourcererCC/parser/java/txl in the SourcererCC project and make clean the project, the following command is executed for the tokenization using the InputBuilderClassic.jar file

   Tokenization of input:
   ```
   java -jar InputBuilderClassic.jar "input-data-dir" ./input/dataset/tokens.file ./input/bookkeepings/headers.file functions java
   ```

2. **Indexing:** For indexing copy input folder which contains tokens, headers file and using indexbasedSearchManager.jar index the code fragments

   ```
   cp -r parser/java/input/ .
   java -jar dist/indexbased.SearchManager.jar index 0.7
   ```

3. **Searching Step:** From SourcererCC project execute

   ```
   java -jar dist/indexbased.SearchManager.jar search 0.7
   ```
The output file contains the \( \langle \text{blockid}, \text{blockid} \rangle \) tuple set of identified clones from different data files provided in the input. The tuples are mapped respectively to line numbers in the original codebase present in the header. The file in the bookkeeping folder has a lookup containing source paths and line numbers.

![Image](image_url)

**Figure 6:** Clone Fragments Lookup File

```
6./home/rajkorpally/Desktop/TestingTools/bcb_function35/35/default/53939.java,48,52
1./home/rajkorpally/Desktop/TestingTools/bcb_function35/35/default/53939.java,60,83
2./home/rajkorpally/Desktop/TestingTools/bcb_function35/35/default/53939.java,67,79
3./home/rajkorpally/Desktop/TestingTools/bcb_function35/35/default/53939.java,70,76
4./home/rajkorpally/Desktop/TestingTools/bcb_function35/35/default/53939.java,85,97
5./home/rajkorpally/Desktop/TestingTools/bcb_function35/35/default/53939.java,99,103
6./home/rajkorpally/Desktop/TestingTools/bcb_function35/35/default/53939.java,105,111
7./home/rajkorpally/Desktop/TestingTools/bcb_function35/35/default/53939.java,129,134
8./home/rajkorpally/Desktop/TestingTools/bcb_function35/35/default/53939.java,136,369
9./home/rajkorpally/Desktop/TestingTools/bcb_function35/35/default/53939.java,378,390
10./home/rajkorpally/Desktop/TestingTools/bcb_function35/35/default/53939.java,380,387
11./home/rajkorpally/Desktop/TestingTools/bcb_function35/35/default/53939.java,399,414
12./home/rajkorpally/Desktop/TestingTools/bcb_function35/35/default/53939.java,543,587
13./home/rajkorpally/Desktop/TestingTools/bcb_function35/35/default/53939.java,557,585
14./home/rajkorpally/Desktop/TestingTools/bcb_function35/35/default/53939.java,559,568
15./home/rajkorpally/Desktop/TestingTools/bcb_function35/35/default/53939.java,568,579
16./home/rajkorpally/Desktop/TestingTools/bcb_function35/35/default/53939.java,573,578
```

**Figure 7:** Sourcerer's Clone Pairs Output

A program SourcererCConverter.java is written to convert the tool's output to satisfy BigCloneBench input. After Tokenization, Indexing and Searching on partial IJaDataset the SourcererCC found 38021 pairs of clones with configuration having min tokens size 8, similarity threshold 70%. SourcererCConverter.java takes clone pairs file and lookup file as input and generates the processed output clones file. The output clones are then imported to BigCloneBench project, matched with reference clone pairs and evaluation report is generated after validating the imported clones.
For benchmarking NiCad first input IJaDataset to NiCad, run NiCad clone tool using a bash NicadRunner bash script with a selected configuration file (Min lines: 5, Max lines 2500, rename=blind, abstract = literal, 70% similarity threshold). The detected clones are outputted as formatted xml pairs of clones.

```
<clones>
  <systeminfo processor="nicad3" system="bcb_function35" granularity="functions" threshold="30%" minlines="3" maxlines="100"/>
  <runinfo ncompare="23968257" runtime="2250868"/>
  <clone nlines="100" similarity="98">
    <source file="/bcb_function35/selected/1644882.java" startline="391" endline="490" pcid="98351"/>
  </clone>
  <clone nlines="100" similarity="99">
    <source file="/bcb_function35/selected/1042270.java" startline="269" endline="368" pcid="3339"/>
  </clone>
  <clone nlines="100" similarity="99">
    <source file="/bcb_function35/selected/1894116.java" startline="253" endline="352" pcid="7861"/>
  </clone>
  <clone nlines="100" similarity="99">
    <source file="/bcb_function35/selected/1042270.java" startline="269" endline="368" pcid="3339"/>
  </clone>
  <clone nlines="100" similarity="99">
    <source file="/bcb_function35/selected/1042270.java" startline="269" endline="368" pcid="3339"/>
  </clone>
  <clone nlines="100" similarity="99">
    <source file="/bcb_function35/selected/1042270.java" startline="269" endline="368" pcid="3339"/>
  </clone>
  <clone nlines="100" similarity="99">
    <source file="/bcb_function35/selected/1042270.java" startline="269" endline="368" pcid="3339"/>
  </clone>
  <clone nlines="100" similarity="99">
    <source file="/bcb_function35/selected/1042270.java" startline="269" endline="368" pcid="3339"/>
  </clone>
  <clone nlines="100" similarity="99">
    <source file="/bcb_function35/selected/1042270.java" startline="269" endline="368" pcid="3339"/>
  </clone>
  <clone nlines="100" similarity="99">
    <source file="/bcb_function35/selected/1042270.java" startline="269" endline="368" pcid="3339"/>
  </clone>
```

**Figure 8:** NiCad clone pairs

Then formatting the NiCad output to the BigCloneBench desired input format using a NiCadConverter.java with uses an xml parser for extracting the clones in the required format. Import the converted clones using the correct Input the converted output to BigCloneBench to validate the clones detected by NiCad and generate final report

**JPlag**

The default directory structure of IJaDataset doesn’t support JPlag to generate clones as expected. For our needs first each directory of dataset is run separately to detect the clones within each directory, then JPlag is run on parent directory to get cross directory clones. JPlag is run with configuration of 70% match similarity and min token length of 8. The output generated is a large set of html files which contain the pairs of matching clones. The goal here is not compare the overall percentage match between folders but find the pairs of clones with matched token threshold and see if these exist in the reference clones database.
A JPlagConverter.java is written for extracting clones from these individual HTML files from all compared files and merged the clones to be represented as the input required for BigCloneBench.

JSoup HTML parser library is used in the program for the extraction of clone pairs.

**Deckard**

Deckard was run with the following parameters on the IJadataset: Minimum length of tokens: 30, Similarity threshold: 90%, Token stride: 2, Maximum processes used: 8, Vector generator for used for java: cvecgen

Similarity distance measure used: ‘1.0’.

The clones generated when the similarity threshold reduced if huge, so it was restricted to 90%.

Deckard is a tree based clone detector which generates a lot of a extra information about generated like clones nodes, vector id’s. The final clones are generated as cluster pairs having information of source, similarity distance, line numbers, tree nodes information.

**ConQat**
ConQat is run as an Eclipse plugin. ConQat doesn’t provide configurable parameters like gap ratio, max errors parameter for clone detection. The tool is run with min fragment size of 8, gap ratio 2, max errors 5. As a plugin ConQat provides clone pair visualizations in Eclipse. The total collection of clones generated is output in an xml file as clusters. The clone file assigns ids to each source file, clone fragment and clone class. A ConQatConverter.java program is written to parse the clones in xml and extract clone pairs from the clone classes using the id’s. This generated clone pairs output is passed to the BigCloneBench input for further clones analysis.

Figure 11: ConQat Gapped Clone Analysis form ConQat eclipse plugin [10].

Clone pair format
The final clones pairs output generated by each the tool is converted to a standard format clone pairs from the files are imported to the BigCloneBench database and compared with the reference clones.
5.1.2 Output Summary of Results from BigCloneBench[1]:

SourceerCC

```
-- Recall Per Clone Type (type: numDetected / numClones = recall) --
Type-1: 247 / 249 = 0.9919678714859438
Type-2: 37 / 42 = 0.8809523809523809
Type-2 (blind): 1 / 1 = 1.0
Type-2 (consistent): 36 / 41 = 0.8780487804878049
Very-Strongly Type-3: 336 / 346 = 0.9710982658959537
    Strongly Type-3: 42 / 54 = 0.7777777777777778
    Moderately Type-3: 29 / 493 = 0.058823529411764705
Weakly Type-3/Type-4: 2 / 338716 = 5.904651684597125E-6
```

Deckard

```
-- Recall Per Clone Type (type: numDetected / numClones = recall) --
Type-1: 248 / 249 = 0.9959839357429718
Type-2: 37 / 42 = 0.8809523809523809
Type-2 (blind): 1 / 1 = 1.0
Type-2 (consistent): 36 / 41 = 0.8780487804878049
Very-Strongly Type-3: 336 / 346 = 0.9710982658959537
    Strongly Type-3: 42 / 54 = 0.7777777777777778
    Moderately Type-3: 29 / 493 = 0.058823529411764705
Weakly Type-3/Type-4: 2 / 338716 = 0.002391383932261836
```

JPlag

```
-- Recall Per Clone Type (type: numDetected / numClones = recall) --
Type-1: 249 / 249 = 1.0
Type-2: 27 / 42 = 0.6428571428571429
```
Type-2 (blind): 0 / 1 = 0.0
Type-2 (consistent): 27 / 41 = 0.6585365853658537
Very-Strongly Type-3: 283 / 346 = 0.8179190751445087
   Strongly Type-3: 8 / 54 = 0.14814814814814814
   Moderately Type-3: 1 / 493 = 0.002028397565922921
Weakly Type-3/Type-4: 2 / 338716 = 5.904651684597125E-6

ConQat

-- Recall Per Clone Type (type: numDetected / numClones = recall) --
Type-1: 101 / 249 = 0.40562248995983935
Type-2: 28 / 42 = 0.6666666666666666
Type-2 (blind): 0 / 1 = 0.0
Type-2 (consistent): 28 / 41 = 0.6829268292682927
Very-Strongly Type-3: 0 / 346 = 0.0
   Strongly Type-3: 0 / 54 = 0.0
   Moderately Type-3: 0 / 493 = 0.0
Weakly Type-3/Type-4: 4 / 338716 = 1.180930336919425E-5

NiCad

-- Recall Per Clone Type (type: numDetected / numClones = recall) --
Type-1: 247 / 249 = 0.9919678714859438
Type-2: 29 / 42 = 0.6904761956412131
Type-2 (blind): 1 / 1 = 1.0
Type-2 (consistent): 28 / 41 = 0.6801245875122558
Very-Strongly Type-3: 306 / 346 = 0.9110982658959537
   Strongly Type-3: 47 / 54 = 0.8703703703703703
   Moderately Type-3: 3 / 493 = 0.00608519269768763
Weakly Type-3/Type-4: 0 / 338716 = 0.0

5.1.2 Result Analysis

The objective of BigCloneBench is to evaluate the clone identifying capability of the various tools in detecting various types of clones satisfying the standard definition. The generated results confirm the published results in the BigCloneBench paper and justify the underlying algorithms and mechanisms used by each tool.
Type 1 Analysis

Program 1A

```java
class BSearch {
    int binarySearch(int[] a, int x) {
        int result = -1;
        int mid = 0;
        int left = 0;
        int right = a.length - 1;
        while (result == -1 && left <= right) {
            mid = (left + right) / 2;
            if (a[mid] == x) {
                result = mid;
            } else {
                if (a[mid] > x) {
                    right = mid - 1;
                } else {
                    left = mid + 1;
                }
            }
        }
        return result;
    }
}
```

Figure 12: Type 1 clone pair from BigCloneBench[1][12] database

Program 1B

```java
class BSearch {
    //BinarySearch Method
    int binarySearch(int[] a, int x) {
        int result = -1;
        int mid = 0;
        int left = 0;
        int right = a.length - 1;
        //Check conditions
        while (result == -1 && left <= right) {
            mid = (left + right) / 2;
            if (a[mid] == x) {
                result = mid;
            } else {
                if (a[mid] > x) {
                    right = mid - 1;
                } else {
                    left = mid + 1;
                }
            }
        }
        return result;
    }
}
```

Program 1A and Program 1B only differ in indentation and comments satisfying the conditions of type 1 clone. All tools are well equipped to handle this type of clones and we can see the recall values for tools NiCad, SourcererCC, Deckard, JPlag near 100%.

Figure 13: ConQat output highlighting the type 1 boundary coverage problem

ConQat is the only tool which has low recall values for type 1, this is not the problem of the tool itself but the boundaries displayed by ConQat tool as output. All tools identify clone pair as line 1-22 from program A and line 1-23 from program B, whereas ConQat identifies as 4-20 from Program 1A and 4-22 from Program 1B. This mismatches with the reference clone boundaries in
the database which include whole function as the clone coverage algorithm doesn’t cover the clone pair properly to show it as a true positive.

Type 2 Analysis

Program 2A

```java
/**
 * Returns the next higher value in the domain or current value if none exists
 */
public int getNextHigher(int val) {
    if (values.size() == 0) return val;
    int min = values.get(0);
    if (val < min) return min;
    int max = values.get(values.size() - 1);
    if (val == max) return val;
    int next = val + 1;
    int idx = indexOfValue(next);
    if (idx == 0) return next;
    idx = -idx - 1;
    return values.get(idx);
}
```

Program 2B

```java
/**
 * Returns the next lower value in the domain or current value if none exists
 */
public int getNextLower(int val) {
    if (values.size() == 0) return val;
    int max = values.get(values.size() - 1);
    if (val > max) return max;
    int min = values.get(0);
    if (val == min) return val;
    int prev = val - 1;
    int idx = indexOfValue(prev);
    if (idx >= 0) return prev;
    idx = -idx - 1;
    return values.get(idx);
}
```

Figure 14: Type 2 clone pair from BigCloneBench[1][2] Database

The Program2A and Program2B pair from figure 14 represent type2 clone structure, they differ in variable renaming and reversal of comparative operators and changing of addition and subtraction symbols. SourcererCC, Deckard detects about 89% of such clones, whereas JPlag and ConQat have recall of 65%.

NiCad fails to detect the above clone example and similar clones as it uses text based LCS comparison algorithm whereas token based clone detectors easily identify such clones where identifier renaming is involved.
Figure 15: Failed Type 2 NiCad detection (only 41% similar, cutoff 70%)

As shown in figure 15, NiCad only reports 41% similarity failing the cutoff threshold of 70% to be detected as a clone. JPlag even though can detect type-2 changes very easily, it cannot detect clones from same files, the above clone pair is taken from a same file and detection is missed due that. Hence the lower values for JPlag recall values compared to other tools due to lack of identification of clones from a same file.

**Type 3 Analysis**

**Program 3A**

1.

```java
public static final int binarySearch(int[] a, int key, int length) {
    int x1 = 0;
    int x2 = length;
    int i = x2 / 2;
    while (x1 < x2) {
        if (a[i] == key) {
            return i;
        } else if (a[i] < key) {
            x1 = i + 1;
        } else {
            x2 = i;
        }
        i = x1 + (x2 - x1) / 2;
    }
    return -1 * (i + 1);
}
```

2.

```java
public static final int binarySearch(int[] a, int key, int length) {
    int x1 = 0;
    int x2 = length;
    int i = x2 / 2;
    while (x1 < x2) {
        if (a[i] == key) {
            return i;
        } else if (a[i] > key) {
            x2 = i;
        } else {
            x1 = i + 1;
        }
        i = x1 + (x2 - x1) / 2;
    }
    return -1 * (i + 1);
}
```

**Figure 16**: Type 3 clone pair from BigCloneBench[1][2] database with high structural similarity

**Program 3B**
Figure 17: Type 3 clone pair from BigCloneBench[] Database with moderate structural similarity
public abstract class ArrayLib {
    public static final int binarySearch(int[] a, int key, int length) {
        int x1 = 0;
        int x2 = length;
        int i = x2 / 2;
        while (x1 < x2) {
            if (a[i] == key) {
                return i;
            } else if (a[i] > key) {
                x2 = i;
            } else {
                x1 = i + 1;
            }
            i = x1 + (x2 - x1) / 2;
        }
        return -1 * (i + 1);
    }
}

public class BinarySearch {
    public static int binarySearch1(int arr[], int key, int imin, int imax) {
        //Implementation: Recursive, primitive type
        if(imax < imin)
            return -1;
        int imid = (imin+imax)/2;
        if(arr[imid] > key)
            return binarySearch1(arr,key,imin,imid-1);
        else if (arr[imid] < key)
            return binarySearch1(arr,key,imid+1,imax);
        else
            return imid;
    }
}

public class BinarySearch {
    public static <T extends Comparable<T>> int binarySearch3(T[] arr, T key, int imin, int imax) {
        //Implementation: Recursive, comparable type
        if(imax < imin)
            return -1;
        int imid = (imin+imax)/2;
        if(arr[imid].compareTo(key) > 0)
            return binarySearch3(arr,key,imin,imid-1);
        else if (arr[imid].compareTo(key) < 0)
            return binarySearch3(arr,key,imid+1,imax);
        else
            return imid;
    }
}

Figure 18: JPLag detection of Program 3A 1,2 pair (Detects clone as a whole, matches reference clone in BigCloneBench database)

Figure 19: JPLag detection of Program 3B 1,2 pair (Detects clone as a parts, misses reference clone in BigCloneBench database)

Figure 20: NiCad detects Program 3B pair as a whole (matches reference)
Figure 21: ConQat detects 3A 1,2 clone pair but boundary doesn’t include the same method line

Type 3 clones differ in statement rearrangement, insertion, deletion of in between lines of code and may also include type 2 changes. Strongly type 3 clones have a high level of structural similarity. These types of clones are comparatively easy to detect. Type 3 clones with decreasing similarity become harder to detect, and some fall into type 4 category if the syntactic similarity is below a certain threshold. The recall values of high similarity type 3 clones are good (85%+ recall averaged for strong and very strong type 3) for NiCad, SourcererCC, and Deckard. JPlag too performs well with high similarity type 3 but the performance drastically reduces (81% to 15%) as the similarity level slightly falls from very strong to strong type 3. This can be attributed to gaps formed in the programs and the mechanism of boundary identification. The above JPlag detection of 3A and 3B program pairs from figure 16 and figure 17 show the problem with JPlag in detecting clones with gaps, single color(whole clone) in figure 18 vs multi-color (parts clones) in figure 19. Deckard matches the clone lines in both pairs and positively detecting a large number of strong type clones validating its recall value of about 90% for type 3 clones. One problem with Deckard is the tool generates a large number of clone pairs after de-clustering step. The high recall value is achieved due to this enormous number of clones generated compromising the precision value of the tool, generating a large number of useless clone pairs. SourcererCC parses the code pairs and identifies both the clone pairs of 3A(1,2) and 3B(1,2) as whole functions. The result is verified by doing a lookup in the headers file for the detected clone pair, the finding supports the evaluated high recall value of 87% average for strong and moderate type 3 clones of the tool.

The ConQat detection of 3A program pair figure 21 shows boundary clone is defined by block type and missing the method line, reference clones in the database have function boundary and ignoring of lines costs ConQat recall values when matching the reference clones in the BigCloneBench even after the matching program tries to normalize the boundaries. Type 3 with very less structural similarity are considered as type 4 clones.
Type 4 Analysis

Program 4A

```java
public static int calculateGCD(int a, int b) {
    while (b != 0) {
        int t = a;
        a = b;
        b = t % b;
    }
    return a;
}
```

Program 4B

```java
public static int gcd(int a, int b) {
    while (a != b) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return a;
}
```

Program 4C

```java
public static int gcd(int a, int b) {
    if (b == 0) {
        return 1;
    }
    else {
        return gcd(b, a % b);
    }
}
```

Figure 22: Type 4 Clones from BigCloneBench[1][12] Database

Program 4A, 4B, 4C from figure 22 are semantically similar performing GCD functionality, they represent type 4 clones. All the tools considered for the project fail to detect type 4 clones since the core algorithms are not suited for detecting semantically similar clones. Deckard comparatively performs slightly better (but not significantly better) in detecting Type 4 category compared to others since it uses a tree based comparison algorithm followed by vector matching but still doesn’t detect the above example. The recall values of close to zero from tool reports support this analysis.

Drawbacks of BigCloneBench System

Though BigCloneBench helps in automation of tool evaluation, it is not a system to be fully trusted. Firstly, the input to the system can be only in the form of line numbers of the clones, and this limits the tools that can be benchmarked using it. Modern Graph based, tree based tools which target semantically similar clones are hard to test using the system. The recall values of type 4 clones from all tools are close to zero as the recall for type 4 needs more parameters rather than just only line number comparison to work. One observation is that the type 4 clones (weakly type 3) number dominate the reference
clone database but never detected by any tool rendering the benchmark useless to evaluate type4 clones unless matching algorithm and parameters are improved. Secondly, the reference clones are functional in granularity and this biases the framework towards tools generating functional outputs. The clone coverage matcher can be tweaked but can never be matched with the performance as having as block reference clones. Tools JPLag and ConQat fall victims to this problem and significantly lowers the recall values for these tools. Thirdly, the confidence of clones derived from the benchmark is questionable. The system appears to follow standard definition of clones but when the database is checked for validation using the definitions there exist a significant number of clones which don’t fall into any category specified or fall into the wrong category. Also, knowledge generated from the reports doesn’t give out significant information of the missed clones position and the detected clones, everything is in the form of a statistic measure.

5.2 CodeChef based Evaluation System

To compare the tool results of BigCloneBench with real world data, a system based on CodeChef programming clones as reference is designed and evaluated. The clones in the BigCloneBench target specific functionality and is fine grained where its reference clones are derived using data mining looking for a particular type of functionality. The CodeChef based system targets general clones occurring in programming assignments without any specific targeted functionality. We don’t focus on type 4 clones as tools considered for the project lack the ability to detect them. The system design for the evaluation system is as follows:

![System Design for CodeChef based Tool Evaluation System](image)

**Figure 23:** System Design for CodeChef based Tool Evaluation System
5.2.1 Steps of Evaluating tools on CodeChef data

1. Creating a database of clones from the CodeChef student submitted programs.

For this purpose different types of clones are identified from the corpus of generated clones by the tools from the CodeChef programming data and are then converted to a common format and stored as reference clones. This process of clone identification for database creation is a manual process taking help of the tools. The format of clones stored is similar to clone pair format used previously with extra information on source of clones.

2. Running the tools on the CodeChef data in different configurations and evaluating clones.

The tools are already set up for the evaluation process from the previous experiments, only thing that is changed is running the tools on CodeChef dataset instead of IJaDataset that was previously used in various configurations to generate clones.

3. Converting generated clones from tools in comparable format as the database format clones.

The same processing programs previously written for the BigCloneBench analysis are re used with some minor modification for generating the clones in the required format. Major part of extraction and parsing code and converter logic remains same for the tools as the output transformations do not change.

4. Writing a clone comparator program to compare generated clones with the database clones.

This is the main part of the benchmark for correct detection of clones by the tools. Each tool has its own representation of what a clone should represent, some tools represent line numbers of blocks, some represent clones as fragments within the block. Generalizing matching of tool’s generated clones with reference clones is not a direct comparison task. Progrm CodeChefCloneMatcher.java is written and used for this purpose. Consideration is given to include offsets for inclusion and exclusion of extra lines to detect clones. An algorithm based on cover match metric is used by BigCloneBench to tackle this problem. This is represented by

\[
\text{covers}(f_1, f_2, t) = \frac{\min(f_1, e, f_2, e) - \max(f_1, s, f_2, s) + 1}{f_2, e - f_2, s + 1} \geq t
\]

\[
c - \text{match}(C, R, t) = \text{covers}(C, f_1, R, f_1, t) \land \text{covers}(C, f_2, R, f_2, t)
\]

Algorithm Reference paper [1]

f: fragments in clone pairs, e: end line number, s: start line number, t: is coverage threshold, C is clone in comparison, R is the reference clone from the clone database. [1]

5. Generate evaluation report

The results of the comparator program are collected and result report gives us knowledge of types of clones identified and missed by each tool.
Results for Tools Experiments based on CodeChef System:

Below is the summary of recall values from CodeChef based evaluation system:

<table>
<thead>
<tr>
<th>Tool</th>
<th>Type 1</th>
<th>Type 2</th>
<th>Type 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConQat</td>
<td>0.91</td>
<td>0.46</td>
<td>0.37</td>
</tr>
<tr>
<td>Nicad</td>
<td>0.97</td>
<td>0.66</td>
<td>0.27</td>
</tr>
<tr>
<td>JPlag</td>
<td>0.83</td>
<td>0.71</td>
<td>0.39</td>
</tr>
<tr>
<td>SourcererCC</td>
<td>0.93</td>
<td>0.73</td>
<td>0.34</td>
</tr>
<tr>
<td>Deckard</td>
<td>0.86</td>
<td>0.65</td>
<td>0.54</td>
</tr>
</tbody>
</table>

**Figure 24:** Recall values for the tools (CodeChef clones based)

Based on the clones detected by the each of the tools, overlapping values of the clones between the tools are calculated and the results are shown in the below figure.

<table>
<thead>
<tr>
<th>Tools</th>
<th>ConQat</th>
<th>Nicad</th>
<th>Deckard</th>
<th>JPlag</th>
<th>SourcererCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConQat</td>
<td>1</td>
<td>0.434</td>
<td>0.054</td>
<td>0.122</td>
<td>0.487</td>
</tr>
<tr>
<td>Nicad</td>
<td></td>
<td>1</td>
<td>0.004</td>
<td>0.020</td>
<td>0.734</td>
</tr>
<tr>
<td>Deckard</td>
<td></td>
<td></td>
<td>1</td>
<td>0.031</td>
<td>0.085</td>
</tr>
<tr>
<td>JPlag</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0.093</td>
</tr>
<tr>
<td>SourcererCC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

**Figure 25:** Overlapping clones ratio of clones between tools

<table>
<thead>
<tr>
<th>Tools</th>
<th>ConQat</th>
<th>Nicad</th>
<th>Deckard</th>
<th>JPlag</th>
<th>SourcererCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Times</td>
<td>82000 ms</td>
<td>9251 ms</td>
<td>135684 ms</td>
<td>42150 ms</td>
<td>20444 ms</td>
</tr>
</tbody>
</table>
Figure 26: Time to run each tool on the CodeChef test data for clones identification.

5.2.2 Analysis

The granularity of clones collected from the CodeChef programs is both functional and block type. This enables the tools like JPlag and ConQat to be compared at an equal level with the tools having functional clone outputs and functional reference clones in the database like BigCloneBench. Figure 27 shows the comparison between the recall values of the both the evaluated systems. It can be seen there is a general decrease in the recall values in CodeChef system but the overall trend remains similar with a few observed anomalies. The general decrease could be due to having more generalized and crude nature of clones in the CodeChef system, unlike BigCloneBench where clones are mined based on similar functionality and expert validation of dataset is performed. We can observe from the Type 1 clones comparison in graph in figure 27 that there is increase in the recall values of Type 1 in JPlag and ConQat, one reason is due to having included both granularity clones in the dataset. For evaluating the performance in
detecting type 3 clones from BigCloneBench, the averaged recall values of very strong and strong type 3 clones is taken and lower recall values are ignored. Averaging is done because the cutoff used for most clone tools is set as 70% similarity threshold and clones having any lower similarity are not identified by the tool and so never compared with reference clones having a smaller token or line similarity value. ConQat fails to detect type 3 clones using BigCloneBench, the reason for this anomaly is not figured out (even after varying gapped ratio parameter the results for type 3 did not change). But we can see significant type 3 recall improvement for ConQat on CodeChef based system. JPlag doesn’t have a Java1.8 leading to ignoring of inputs and lessening recall values. Considering performance speed of clone detection NiCad performs fastest and Deckard the slowest as seen from the figure 26. The timing can be justified by the core algorithm used in Deckard AST’s and vector generation as intermediate steps take significantly longer. Also, Deckard generates a large number of clones and clearly has better type 3 clone detection capabilities due to its underlying clone detection algorithm. Figure 24 shows the overlapping clones rations between the clones detected by individual tools. The clones detected by NiCad and SourcererCC are the most overlapping, this can be attributed to the having a common creator for the tool and using similar clone detection principles in both tools. Deckard has the least overlapping clone ratio with other tools.

6. Conclusions

Automated evaluation of cloning tools is a nontrivial task. It requires selecting the right metrics of comparison and configuration parameters while agreeing on the standard definitions of clones. BigCloneBench framework is a step in the right direction but trivializes the clone matching to comparing line numbers and misses out on several important details apart from the drawbacks previously discussed. Reference clone bias having only functional granularity affects the recall of block fragment based clone tools like ConQat and JPlag in systems like BigCloneBench. The recall values of tools are heavily dependent on clone matcher algorithm and the input configuration parameters used for clones extraction using the tools. Using parameters with liberal thresholds increases the number of clones detected improving recall but compromising the precision values. The CodeChef based evaluation system has only a small database of reference clones, and it needs to be enlarged for having a better confidence for the results. Preparing a large reference clones dataset requires considerable effort, often not worth the effort and requires expert validation of clones as an essential part of the process. Focusing on existing clone databases for performing tool evaluation should be considered first. The study shows there is a need for clone benchmarks that cater to the evaluation of graph based or tree-based systems for detecting semantically similar clones which don’t just compare line numbers for clone confirmation. We can conclude based on the project is that all tools are unique and built to satisfy different sets of requirements. JPlag and ConQat even though do not perform great using the
evaluation benchmarks, they are better clone visualization tools which provide interactive examination of clones for practical use. The clones detected from Deckard, SourcererCC are difficult to compare and require considerable user effort. Future scope of the project can be including modern PGD or AST based tools by extending them to suit benchmarking requirements and incorporating more evaluation metrics in tool comparison to have better results.

8. References


[3]. IJadataset2.0 source: https://sites.google.com/site/asegsecold//projects/seclone


[9]. JPLag Tool: https://jplag.ipd.kit.edu/


[12]. IJaDataset2.0: Source of the programs used in the clone detection examples. https://sites.google.com/site/asegsecold//projects/seclone