Making personalized education scalable using Sequence Alignment Algorithm

Submitted by: Lakhan Bhojwani

Advisor: Dr. Carlos Rivero
1. Abstract

There are many ways proposed to detect code clones in the past decade. Some of them are text-based technique with Longest Common Sequence Algorithm, Abstract Syntax Tree based technique, Program Dependencies Graph based technique and metrics-based technique. Some papers also discuss about the hybrid technique which includes metrics-based, Abstract Syntax Tree based and the Program Dependencies Graph based together. All of these methods have some drawbacks such as Longest Common Sequence will find the code clone only if there are duplicate lines in both the codes which are been compared. This paper proposes an algorithm which is a modified version of Smith and Waterman algorithm which is used to come over the drawbacks of the previously proposed techniques [1]. The proposed algorithm works only for Java and can be made generalized to all languages in the future.
2. Introduction

Nowadays the number of students enrolling in computing science are increasing with a tremendous speed and thus providing personalized feedback to all students for all assignments is impossible. With the increase in the numbers of student’s enrollment, the number of submissions to an assignment question has increased. Due to this, it is impossible for a professor to check all the codes manually for the code similarity within these thousands of submissions. Finding code similarity is not only about just searching for matching lines in the two codes, but it is a smart detecting technique which can detect codes which has different variable names, different class names, different method names and different syntax but still has the same semantics. We, humans can manually detect code clones in the two given files but it will take a lot of time to do that. Detecting the code clones by computers will be very fast but it will be not be as tedious as it is for the humans.

Let us compare the following two codes which are taken from the CodeChef website. By human eyes we can notice that both the codes are 100% similar even after there are some changes such as the class name in figure 1.a is ‘PrintNumbers’ and in figure 1.b it is ‘Test’. Similarly, the variables names are different in both the codes. If we perform just the string matching algorithms to this pair of code, we will not get the 100% match result because here the variable names are different. We can also see that the second code has many commented lines which will fail the normal text matching algorithms. We want our application to provide the result for the given example to be 100% similar.
The aim for this project is to find similarities between two codes providing that they are not identical and this project proves that it is more efficient and accurate than the previously proposed method. There are many proposed algorithms that are very well know. Some of them are [2]:

- Text-based technique with Longest Common Sequence Algorithm
- Abstract Syntax Tree based technique
- Program Dependencies Graph based technique
- Metrics-based technique
- Hybrid technique which includes metrics-based, Abstract Syntax Tree based and the Program Dependencies Graph based together.

Most of these techniques can also detect gapped code clone. Among these technologies Abstract Syntax Tree and Program Dependencies Graph technique are accurate but they are time consuming.
as we will have to convert the source files into the trees and graphs and comparing them is a NP hard problem. Algorithms such as Needleman–Wunsch, Hirschberg's algorithm, BLAST, FASTA and Smith–Waterman were explored for the project. Out of these algorithms, BLAST and Smith-Waterman were considered to move forward. Both the algorithms BLAST and Smith-Waterman are to find the sequence. The main difference between BLAST and Smith-Waterman is that BLAST is a heuristic algorithm and the result found from BLAST will not be accurate as it will not find all the matches in the database. Whereas Smith-Waterman will find all the hits in the database and will not miss a single hit and because of it the accuracy will be higher as compared to the BLAST. The only drawback about the Smith-Waterman algorithm is that it takes a lot of time to execute the algorithm and it need more computer usage and space as compared to BLAST [4]. So for now we will move forward with Smith and Waterman as for this project the priority is accuracy. We are now proposing an algorithm which is a modified version of Smith and Waterman algorithm [3]. The proposed method is also a code clone detection technique which consider gapped technology and is accurate and very quick as compared to Abstract Syntax Tree and Program Dependencies Graph techniques. LCS-based techniques works well only when we have to find a large chunk of code such as a method which is totally similar in both the codes. The proposed paper’s algorithm and the Smith and Waterman algorithm has a big difference which is Smith and Waterman algorithm can find only one pair of code clone with the maximum size and with the proposed algorithm we can find as many pairs of code clones available between the two codes. The proposed algorithm will give more recall, precision and f-measure as compared to the Smith and Waterman algorithm [1].
3. THE SMITH-WATERMAN ALGORITHM

Smith and Waterman algorithm is used to identify similarities between two strings. The main benefit of this algorithm is that it can identify common parts of strings even if there is a gap. Let us consider two sequence S1 = “HELOWORK” and S2 = “HELLOWORLD”. Now the Smith and Waterman algorithm is divided into five major steps [1].

I. **Creating a table:** A matrix table is created of size (N+1) X (M+1) where N and M are the size of two sequences respectfully which are been compared. For our example the size of sequence 1 is 9 and for sequence 2 is 10. So the matrix size will be 10 X 11 [1].

II. **Table Initializing:** The matrix table is initialized with 0s for the first row and column of the above created matrix. Following is the matrix result after step I and step 2 [1].
III. **Calculating the Score for each cell in the matrix:** Scores are calculated as following:

\[ C_{i,j} (1 \leq i, 1 \leq j) = \max [C_{i-1,j-1} + \text{match/mismatch value}, C_{i-1,j} + \text{gap}, C_{i,j-1} + \text{gap}] \]

Here,

- \( C_{i,j} \) means the current cell
- \( C_{i-1,j-1} \) means the left above cell to the current cell
- \( C_{i-1,j} \) means cell left to current cell
- \( C_{i,j-1} \) means cell above the current cell.
- match/mismatch value is the score added to current cell score if there is match or mismatch
- gap is the score added to the current cell if the gap is found between two sequences. While calculating the score, if the score becomes less than 0 then we make the score 0 for that cell and also we keep the track of the cells from where the current cell is taking value from. We also keep track of highest values cell, so that it will be easy to trackback. In this example we are taking the gap value as -2, match value as 1 and mismatch as -1 [1].

Here’s the matrix after the step 3:

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>E</th>
<th>L</th>
<th>L</th>
<th>O</th>
<th>W</th>
<th>O</th>
<th>R</th>
<th>L</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>O</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>O</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>R</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>
IV. **Trackbacking the matched sequence:** We will start backtracking from the highest calculated cell in the step III. We will move backwards till we find a cell with the score 0. Following is the matrix with the backtracking solution to the above example shown in gray [1].

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>E</th>
<th>L</th>
<th>L</th>
<th>O</th>
<th>W</th>
<th>O</th>
<th>R</th>
<th>L</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>O</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>O</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>W</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>O</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>R</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

V. **Identify matched subsequence:** While backtracking all the Cell values which are considered in backtracking are said to be match subsequence. So for the above example we will get “HELLOWOR” as the matched subsequence [1].
4. The Proposed Algorithm

The proposed algorithm is the modified version of Smith and Waterman. In this algorithm we will get all the available pair of code clones between the two comparing codes and not only one pair of code clone as in Smith and Waterman. This algorithm can change its output depending on the given minimum number of clones that is minimum number of tokens [1]. So the user of this application can change the minimum clone size depending of its expected result. This algorithm takes in the source folder as input and it can take any number of Java source files in that source folder. The algorithm is divided into eight steps and they are [1]:

A. Perform Lexical Analysis.

B. Perform Normalization on the result of Lexical Analysis.

C. Identify the statements with the Normalized result.

D. Generate hash values for every single token of each line in the source file.

E. Create Hash Sequence with the token’s hash value.

F. Identify same hash codes.

G. Identify Gapped tokens.

H. Map the result to the source file.

Figure 2 shows the overview of the proposed algorithm.
Let us consider following two code snippet from two different codes submission for the explanation of the proposed algorithm.

![Figure 2. Overview of the proposed algorithm [1]](image)

A. Code 1

```java
if(flag == true){
    for(int i = 0; i < arrOfTokens.length; i++){
        arrayListOfTokens.add(arrOfTokens[i]);
    }
    int ans = arrayListofTokens.size();
    for(int j = 0; j < arrOfTokens.length; j++){
        arrayListOfTokens.add(arrOfTokens[j]);
        arrayListOfTokens.add(",");
        int ans = arrayListofTokens.size();
    }
    return ans;
}
```

B. Code 2

```java
ArrayList arrayListofTokens = new ArrayList();
for(int i = 0; i < arrOfTokens.length; i++){
    arrayListofTokens.add(arrOfTokens[i]);
    arrayListOfTokens.add(getComma());
    int ans = arrayListofTokens.size();
    System.out.println(ans);
}
```

![Figure 3. Codes for comparison for proposed method](image)
A. **Perform Lexical Analysis:** Firstly, we are reading the source code files line by line and splitting it by all the white spaces and "." , ";" , '"[", "]"', "(" , ")"", ";" , "+" , "{" , "}" and by removing white spaces and keeping rest of the characters and get an array of strings. This is called Lexical analysis. Following is the example of lexical analysis for the above example [1].

B. **Perform Normalization on the result of Lexical Analysis:** After this we iterate through the above array and check whether that word is in the Java’s keywords arraylist or not. If it is in the arraylist then we keep that word as it is and if it is not then we replace it with a token such as “$”. This step is called normalization. And normalization is done to detect not only identical code fragments but also similar ones as code clones even if they include different variables [1]. All modifiers are deleted for the same reason. Following is the array we are going to get after normalization for the above example.

---

A. Code 1

```
if ( flag = true ) {
    for ( int i = 0 ; i < arrOfTokens.length ; i ++ ) {
        arrayOfTokens.add( arrOfTokens[i] ) ;
    }
    int ans = arrOfTokens.size() ;
}
else {
    for ( int i = 0 ; i < arrOfTokens.length ; i ++ ) {
        arrayOfTokens.add( arrOfTokens[i] ) ;
        arrayOfTokens.add( "--" ) ;
    }
    int ans = arrOfTokens.size() ;
    System.out.println( ans ) ;
}
```

B. Code 2

```
ArrayList arrayListOfTokens = new ArrayList() ;
for ( int i = 0 ; i < arrOfTokens.length ; i ++ ) {
    arrayListOfTokens.add( arrOfTokens[i] ) ;
}
arrayListOfTokens.add( getComma() ) ;
int ans = arrOfTokens.size() ;
System.out.println( ans ) ;
```
C. Identify the statements with the Normalized result: Once the normalization is done, we then identify statements. This step is shown below:

![Diagram of normalized code](image)

![Diagram of identified statements](image)
D. **Generate hash values for every single token of each line in the source file:** We are using the default hashcode function which is already built-in in Java. This hashcode is calculated as \( S[0]*31^{(n-1)} + s[1]*31^{(n-2)} + \ldots + s[n-1] \) where \( s \) is the string whose hashcode we need to calculate [6]. This hascode is calculated for the entire line and not just the tokens. We check for ";”, “{”, “}” tokens because each line ends with one of these characters. We also keep note of the number of tokens in the given line. Each hashcode value has different number of tokens. Following is the example hash values for each token [1].

![Hash values of Individual tokens]

**Figure 6. Hash values of Individual tokens [1]**

E. **Create Hash Sequence with the token’s hash value:** All the hashcodes are stored in an ArrayList where the size of this ArrayList will be equal to the number of tokens in the source file. The hashcode of empty and commented lines is given the default value of 0 for the ease of solving the waterman-smith algorithm.

<table>
<thead>
<tr>
<th>Hash Sequence of Code 1:</th>
<th>1 2 3 4 5 6 7 8 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash Sequence of Code 2:</td>
<td>10 2 3 4 5 11 6 12</td>
</tr>
</tbody>
</table>
F. **Identify same hash codes:** It is done in similar way as the Smith and Waterman algorithm.

We are going to fill up the matrix by following the steps of creating matrix, initializing matrix and filling up the matrix discussed in the section 3.I, 3.II and 3.III. Figure 7 shows the creation and initializing of the matrix. Figure 8 shows the calculation of the matrix.

![Matrix](image)

**Figure 7.** Creation and initializing of the matrix [1]
IV. Trackbacking in the proposed method: In the waterman and smith algorithm, tracebacking is done only one time till the cell value 0 is found. This will give the largest matched sequence but it will ignore other matches sequence which are equally important. Here, we are going to trackback for multiple Cells. We start from the bottom right cell of the matrix. We trackback to the cell till we find the cell value as 0. Once the value is found, we start new trackbacking from the row above the cell with the value 0. In the newly selected row, we find the cell with the maximum score and start the trackbacking from there. This process is done till we reach the 0,0 cell of the matrix. By doing this, we make
sure that not a single line of the code is not missed out. Figure 9 shows the trackbacking of the above example.

![Figure 9. Trackbacking of the proposed method [1]](image)

Once the backtracking is done, we then identify the matched hashcodes. Figure 10, shows the matched tokens.

![Figure 10. Matched hashcodes tokens [1]](image)
G. **Identify Gapped tokens**: While doing the trackbacking, we also check the hashcodes of the two comparing sequence, if the value of hashcodes is different, we make a note of it [1]. In the figure 9, red marked cell is noted as the gapped token. This step we do for till we reach the starting position of the matrix. The number of gapped tokens are calculated so that we can calculate the clone only if our calculated tokens are less than the threshold gapped tokens. These gapped tokens together are marked as gapped statement. After the gapped statements are marked, we perform longest common subsequence on each pair of these statements. The main purpose is to find the token based similarity.

<table>
<thead>
<tr>
<th>Code 1</th>
<th>Code 2</th>
</tr>
</thead>
</table>
| ```java
arrayListOfTokens.add(",");
``` | ```java
arrayListOfTokens.add(getComma());
``` |

While trackbacking we marked cell of row 13 and column 7 as red because it was considered in the common sequence and their hashcodes are not same. Once we find the token with gap we then perform Longest Common Sequence on that gapped tokens and then find out the dissimilarity. In the given example we can notice that code 1 is adding “,” to the arraylist and code 2 is calling a ‘getComma()’ method while adding.

H. **Map the result to the source file**: We will be comparing one java code to every other java code. The representation is done by showing the matched percentage and also by showing the line numbers of the matched lines between the two java codes [1]. In this we will consider everything, the matched line, gapped tokens and also the commented lines. For the above example figure 11 shows the matched tokens and lines. The lines which are highlighted with grey are the matched lines in both the codes.
Figure 11. Matched lines [1].
5. **Results**

Before discussing about the result, we need to discuss the dataset which was taken for this experiment. This data was taken from the codechef website [5]. We selected this website because the data is publicly available. Also, there are thousands of users on this website where hundreds of users solve the same problem and hence there is a great chance that we will find many codes similar and where our algorithm can work properly. The only thing here we had to take care was that users of Codechef solve the problem in different languages such as Java, C++, python and many others. As we are performing this proposed algorithm only on Java source files, we had to segregate the Java files from all other programming language source files. This proposed algorithm is going to be tested on 625 Java files which are taken from the Codechef website. We are going to compare our algorithm with the standard Smith and Waterman algorithm and also we have noticed that Smith and Waterman algorithm works very similarly as the Longest Common Subsequence. We have tested this algorithm for the different match percentage. Let us discuss different scenario of percent match rate.

a. **If the match rate is 65%**: If we keep 65% match rate then we get the best result of the proposed algorithm. Following is the table we are getting for the match rate of 65%.

<table>
<thead>
<tr>
<th></th>
<th>Total number of Files correct clustered</th>
<th>Total number of Files incorrectly clustered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our proposed method</td>
<td>603</td>
<td>22</td>
</tr>
<tr>
<td>Smith and Waterman</td>
<td>530</td>
<td>95</td>
</tr>
<tr>
<td>Algorithm and Longest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Common Subsequence</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Here for the proposed algorithm the total number of files that are incorrectly clustered are 22 in which 15 are the False Negative and rest 7 are the False Positive. In Smith and Waterman Algorithm and Longest Common Subsequence the total number of files that are incorrectly clustered are 95 in which 80 are the False Negative and rest 15 are the False Positive. The accuracy percent is 96.3% and 84.8% for our proposed algorithm and Smith and Waterman algorithm respectfully. Figure 12 shows the graph for the 65% match rate.

\[
\begin{array}{|c|c|c|}
\hline
\text{Method} & \text{Total number of Files correct clustered} & \text{Total number of Files incorrectly clustered} \\
\hline
\text{Our proposed method} & 560 & 65 \\
\hline
\text{Smith and Waterman Algorithm and Longest Common Subsequence} & 437 & 188 \\
\hline
\end{array}
\]

Figure 12. If the match rate is 65%

b. **If the match rate is 80%**: If we keep 80% match rate then we get the best result of the proposed algorithm. Following is the table we are getting for the match rate of 80%.
Here for the proposed algorithm the total number of files that are incorrectly clustered are 65 in which 35 are the False Negative and rest 30 are the False Positive. In Smith and Waterman Algorithm and Longest Common Subsequence the total number of files that are incorrectly clustered are 188 in which 160 are the False Negative and rest 28 are the False Positive. The accuracy percent is 88.3% and 69.9% for our proposed algorithm and Smith and Waterman algorithm respectfully. Figure 13 shows the graph for the 80% match rate.

![Graph showing comparison between proposed method and Smith & Waterman Algorithm for 80% match rate](image)

**Figure 13. If the match rate is 80%**

c. **If the match rate is 90%**: If we keep 90% match rate then we get the best result of the proposed algorithm. Following is the table we are getting for the match rate of 90%.

<table>
<thead>
<tr>
<th></th>
<th>Total number of Files correct clustered</th>
<th>Total number of Files incorrectly clustered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our proposed method</td>
<td>498</td>
<td>127</td>
</tr>
<tr>
<td>Smith and Waterman</td>
<td>293</td>
<td>332</td>
</tr>
<tr>
<td>Algorithm and Longest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Common Subsequence</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Here for the proposed algorithm the total number of files that are incorrectly clustered are 127 in which 78 are the False Negative and rest 49 are the False Positive. In Smith and Waterman Algorithm and Longest Common Subsequence the total number of files that are incorrectly clustered are 332 in which 252 are the False Negative and rest 80 are the False Positive. The accuracy percent is 79.6% and 46.9% for our proposed algorithm and Smith and Waterman algorithm respectfully. Figure 14 shows the graph for the 90% match rate.

![Figure 14. If the match rate is 90%](image)
6. Conclusion

This paper discuss a new modified version of Smith and Waterman algorithm to detect clones with gaps. It is clear from Figure 12, 13 and 14 that the accuracy of the proposed algorithm is way better than just performing Smith and Waterman algorithm. This algorithm can give accuracy similar to Program Dependencies Graph but it’s quick because we don’t have to create any graph or tree before real implementation. It works faster for larger programs. As paper [1] discuss that this algorithm was compared with Bellon’s Benchmark and was applied to eight open source and they have noticed that the proposed algorithm’s recall, precision and f-measure is better than the Bellon’s Benchmark.

7. Future Work

Although this project works gives out good results for the given dataset, we will have to add all the Java key words that are available in the Java language to make it work perfectly for all the code. For this project we have considered keywords which were required for this dataset. In future we will have to take consideration of all the keywords that are possibly used while java programming.

Another future work would be to create a visual graphic of matched lines instead of just showing line numbers that are been matched. This can be done by making an Eclipse plugin so that it will be easy for professors to check plagiarism instead of personally mapping the output line numbers to the source files.
8. Acknowledgement

I would like to take this opportunity to express my gratitude to Professor Dr. Carlos Rivero for his help and support and also for guiding me towards this amazing project.

9. References


