SPL-1 Project Report

Static Code Analyzer

Submitted by

Kamruzzaman Asif

BSSE Roll No. : 1217

BSSE Session: 2019-2020

Submitted to

Dr. KaziMuheymin-Us-Sakib

Professor

Institute of Information Technology

University of Dhaka

Institute of Information Technology

University of Dhaka

[30-05-2022]

Table of Contents

1. Introduction

My software tool named Static Code Analyzer (SCA) analyzes the ‘C’ source codes without executing the program. It determines the software metrics (such as LOC and Halstead Complexity metrics) and detect syntactic code clones.

Static analysis of source code is very important and useful analysis for software quality and security. Software developers use static code analysis for better understanding of the software, maintaining coding standards, identifying potential bugs, detecting code clones and code smells, code simplification and sanitizing, improving application performance, better resource utilization, etc.

The whole project consists of two major parts software metrics generation and clone detection. In metrics part to calculate the LOC metrics(such as physical lines of code, number of comment lines and number of logical statements, etc.), I have followed the counting rules of SLOC according to SEI and the IEEE’s standards. At first, a single C source code file is taken as input from the user. The user can input the file path or choose a file to be processed. Following the counting rules and using several string manipulations (linear searching, substring generation, string matching) the LOC metrics are then generated.

For Halstead metrics (such as distinct operators and operands, the total number of operators and operators) and Halstead measures (such as program vocabulary, length, volume, difficulty, effort, time and number of delivered bugs) generation I have followed several steps. At first the preprocessing of source code (such as removal of blank lines, comments, extra spaces, function declarations and hash directives) is done. Then tokenize the preprocessed code and get tokens of all types (such as character tokens, string tokens, number tokens, identifier tokens, etc.) which are later categorized into two major types of tokens (Operator token and Operator token). Lastly, counting tokens and calculating the Halstead measures we get the Halstead metrics.

Clone detection is another core part of my project. To detect code clones, I have followed a token-based approach with the application of winnowing algorithm. At first the source code is preprocessed such as removal of blank lines, comments, hash directives, punctuations, extra white space, and capitalization of letters. Then from the preprocessed code k-grams are generated. The k-grams are then use for hashing and using rolling hash we get the hash values of the k-grams. Next the k-gram hashes are pushed into winnowing algorithm. The winnowing algorithm generates fingerprints from the hash values. Then comparing the fingerprints using statistical methods (Jaccard Similarity and Dice Similarity Coefficient), we get the clone result.

Through this project we get the static analysis of ‘C’ programs. We get software metrics of LOC and Halstead “software science” which provides objective information throughout the software organization. This reduces the ambiguity that often surrounds complex and constrained software projects. Metrics are the key to objectively representing the progress of project activities and the quality of associated software products across the project life cycle. We also can detect code clones which is very important in software engineering. Applications of code clone detection and management are restructuring of clone codes without any change in the external behavior of the code, duplicate code can be removed, bug detected in copied code leads to identifying bugs in original code or vice versa, code size can be reduced, reuse of existing code, etc.

2. Background of the Project

The whole project consists of the following two major parts:

- Software Metrics
- Code Clone

2.1. Software Metrics

Software metrics are the measures of software characteristics that are quantifiable or countable. Software metrics measure different aspects of software complexity and provide information about external quality aspects of software such as maintainability, reusability, and reliability. Thus, they play a vital role in analyzing and improving software quality.

2.1.1 Types of Software Metrics

Software metrics are often categorized into two primary categories as follows[1]:

- Process Metrics
- Product Metrics

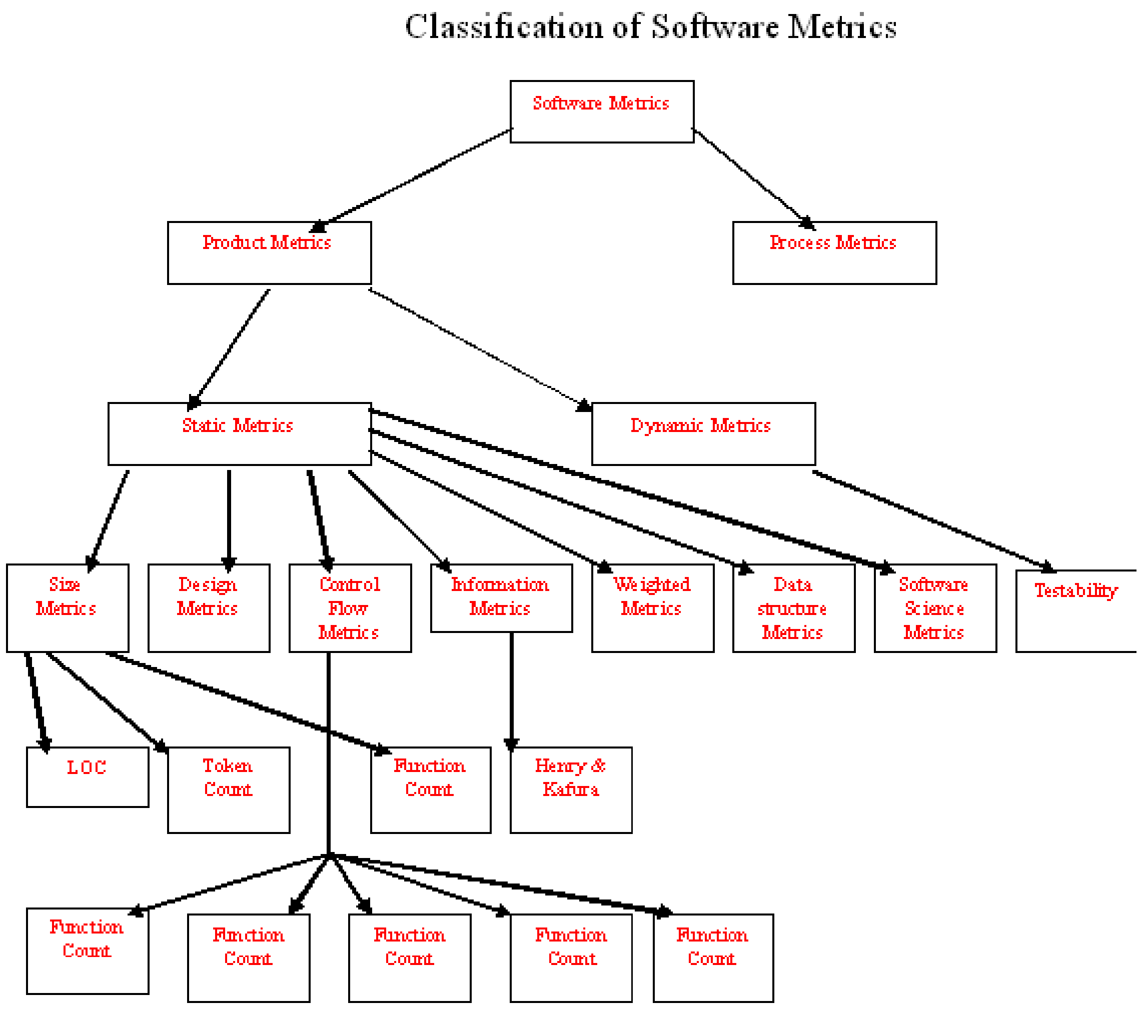


Figure 01: Types of Software Metrics

Source: A Study of Software Metrics, by Gurdev Singh, Dilbag Singh, Vikram Singh, page no. 23

2.1.2 Process Metrics

Process metrics are also known as management metrics that are used to measure the properties of the process which is used to obtain the software. Process metrics include cost metrics, efforts metrics, advancement metrics, and reuse metrics.

2.1.3 Product Metrics

Product metrics are also known as quality metrics that are used to measure the properties of the software. Product metrics include product functionality metrics, performance metrics, usability metrics, cost metrics, size metrics, complexity metrics, and style metrics. Product metrics help to improve the quality of system components & comparisons between existing systems.

2.1.4 LOC Metrics

Lines of Code (LOC) also known as Source Lines of Code (SLOC) is a software metric used to measure the size of a computer program. LOC is the key indicator of software cost and time and also a base unit to derive other metrics for project status and software quality measurement. LOC measures the following:

- Physical Lines of Code
- Logical Lines of Code
- Comment Lines
- Blank Lines

Although the SEI and the IEEE have established SLOC definitions and guidelines to standardize counting practice, inconsistency in SLOC measurements still exists in industry and research. For example, consider the code snippet below:

```
if (a > 0) {
        printf(“a is a positive number”);
}
```

and

```
if (a > 0) printf(“a is a positive number”);
```

The above example shows both code blocks perform the same actions. They should produce the same number of logical LOC. But depending on the interpretation of the SEI framework guidelines, one may end up with a different count for each case. But for this project I tried to follow the standard rules of SEI and IEEE as follows[2]:

Table 01: Logical SLOC Counting Rules for C/C++
Source: A SLOC Counting Standard, Vu Nguyen, Sophia Deeds-Rubin,Thomas Tan Barry Boehm, page no. 9

Structure	Order of Precedence	Logical SLOC Rules
SELECTION STATEMENTS: if, else if, else, “?” operator, try, catch, switch	1	Count once per each occurrence. Nested statements are counted in a similar fashion.
ITERATION STATEMENTS: For, while, do..while	2	Count once per each occurrence. Initialization, condition and increment within the “for” construct are not counted. i.e. for (i= 0; i< 5; i++)… In addition, any optional

		expressions within the "for" construct are not counted either, e.g. for (i = 0, j = 5; i< 5, j > 0; i++, j--)… Braces {…} enclosed in iteration statements and semicolon that follows "while" in "do..while" structure are not counted.
JUMP STATEMENTS: Return, break, goto, exit, continue, throw	3	Count once per each occurrence. Labels used with "goto" statements are not counted.
EXPRESSION STATEMENTS: Function call, assignment, empty statement	4	Count once per each occurrence. Empty statements do not affect the logic of the program, and usually serve as placeholders or to consume CPU for timing purposes.
STATEMENTS IN GENERAL: Statements ending by a semicolon	5	Count once per each occurrence. Semicolons within "for" statement or as stated in the comment section for "do..while" statement are not counted.
BLOCK DELIMITERS, BRACES	6	Count once per pair of braces {..}, except where a closing brace is followed by a semicolon, i.e. };.
COMPILER DIRECTIVE	7	Count once per each occurrence.
DATA DECLARATION	8	Count once per each occurrence. Includes function prototypes, variable declarations, "typedef" statements. Keywords like "struct", "class" do not count.

2.1.5 Halstead Metrics

Halstead metrics, commonly referred to collectively as "software science"[3], are the most widely quoted software measures. For example, researchers have used Halstead's metrics to evaluate student programs[4] and query languages[5], to measure software written for a real-time switching system[6], to measure functional programs[7], to incorporate software measurements into a compiler[8], and to measure to open source software[9].

Halstead considers that a computer program is an implementation of an algorithm considered to be a collection of tokens that can be classified as either operators or operands. In other words, a program can be thought of as a sequence of operators and their associated operands[10].

Halstead's metrics are functions of the counts of these tokens. By classifying and counting the tokens we derive the following four base metrics[3]:

1. the number of distinct operators(n1)
2. the number of distinct operands(n2)
3. the total number of operators(N1)
4. the total number of operands(N2)

All of the Halstead's so called "software science" metrics are defined based on the above collective measures. Halstead defines the following metrics[3]:

- The length(N) of a Program:
  N = N1 + N2
- The vocabulary (n) of a program:
  n = n1 + n2
- Calculated program length(_N):
  _N = n1*log(2)n1 + n2*log(2)n2
- The volume(V) of a program P is defined as:
  a. A suitable measure for the size of any implementation of any algorithm.[]
  b. A count of the number of mental comparisons required to generate a program.[]

  V is computed as

  V = V= N * log(2)n
- Program difficulty(D):
  The difficulty level or error proneness (D) of the program is proportional to the number of unique operators in the program. D is also proportional to the ratio between the total number of operands and the number of unique operands.(i.e. if the same operands are used many times in the program, it is more prone to errors)

  D = (n1/2) * (N2/n2)
- Programming effort (E):
  Programming effort is defined as a measurement of the mental activity required to reduce a preconceived algorithm to a program P. E is defined as the total number of elementary mental discrimination required to generate a program.

  E computed as
  E = D * V
- Programming time(T):
  The time to implement or understand a program (T) is proportional to the effort.

  Time required to program is defined as

  T = E/18 seconds
- Number of Delivered Bugs(B):
  The number of delivered bugs (B) correlates with the overall complexity of the software. Estimate the number of errors in the implementation. Delivered bugs in a file should be less than 2. The number of delivered bugs approximates the number of errors in a module. As a goal at least that many errors should be found from the module in its testing.

  $B = E^{(2/3)} / 3000$ or B = V/3000

Counting rules for C language[12],[13]–

1. Comments are not considered.
2. The identifier and function declarations are not considered
3. All the variables and constants are considered operands.

4. Global variables used in different modules of the same program are counted as multiple occurrences of the same variable.
5. Local variables with the same name in different functions are counted as unique operands.
6. Functions calls are considered as operators.
7. All looping statements e.g., do {…} while ( ), while ( ) {…}, for ( ) {…}, all control statements e.g., if ( ) {…}, if ( ) {…} else {…}, etc. are considered as operators.
8. In control construct switch ( ) {case:…}, switch as well as all the case statements are considered as operators.
9. The reserve words like return, default, continue, break, sizeof, etc., are considered as operators.
10. All the brackets, commas, and terminators are considered as operators.
11. GOTO is counted as an operator and the label is counted as an operand.
12. The unary and binary occurrence of "+" and "-" are dealt separately. Similarly "*" (multiplication operator) are dealt separately.
13. In the array variables such as "array-name [index]" "array-name" and "index" are considered as operands and [ ] is considered as operator.
14. In the structure variables such as "struct-name, member-name" or "struct-name -> member-name", struct-name, member-name are taken as operands and '.', '->' are taken as operators. Some names of member elements in different structure variables are counted as unique operands.
15. All the hash directives are ignored.

2.2 Code Clones

Code clones are fragments of code that are similar[14]. Code cloning occurs by programmers duplicate source code with or without modifications [Roy et al., 2009] and it is a common activity found in software development.

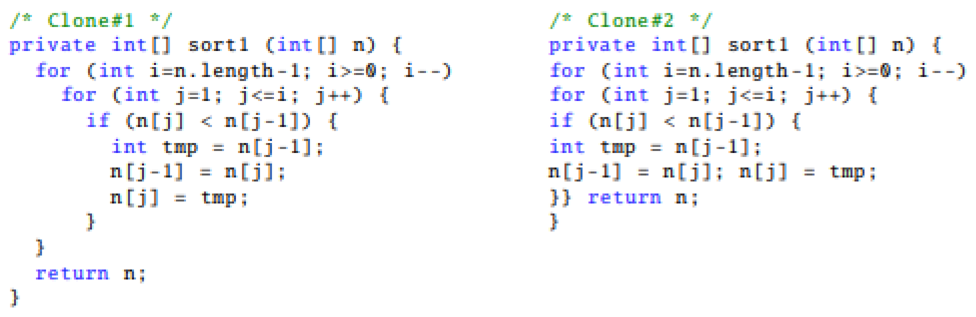
2.2.1 Code Clone Terminology

From "Chaiyong Ragkhitwetsagul" thesis named Code "Similarity and Clone Search in Large-Scale Source Code Data", Doctor of Philosophy University College London, September 30, 2018 we get the following definations[14]:

Code fragment is a segment of code represented by a triple consisting of the source file, the starting and the ending line.
Clone pair is a pair of code fragments and an associated type of similarity, i.e., Type-1, -2, -3 or -4.
Type-1 clones are literally identical code fragments except for differences in formatting such as white spaces, layouts, and comments (as shown in Figure 02).
Type-2 clones are syntactically identical code fragments except for differences in identifiers, literals, types, and formatting (as shown in Figure 03).
Type-3 clones are similar fragments with modified, relocated, added, or removed statements (as shown in Figure 04).
Type-4 clones are code fragments that may not be syntactically similar but share the same semantic. Figure 05 shows a Type-4 clone pair of two sorting algorithms that are implemented independently. They are syntactically different but they share the same semantic based on input and output.

Syntactic clones: We may call Type-1, Type-2, and Type-3 clones as syntactic clones because they mostly preserve the semantic of the originals while differ at syntactic level [Kim et al., 2011]. Syntactic clones are commonly found in software systems as suggested by empirical clone studies.

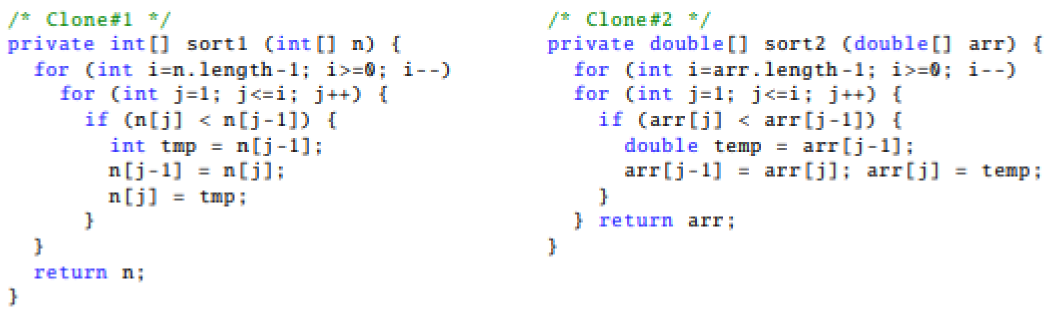
Semantic clones: For Type-4 clone pairs, they may not resemble the same syntax and only contain an equivalent program semantic. Hence, they are sometimes called semantic clones [Funaro et al., 2010].



```java
/* Clone#1 */
private int[] sort1 (int[] n) {
  for (int i=n.length-1; i>=0; i--)
    for (int j=1; j<=i; j++) {
      if (n[j] < n[j-1]) {
        int tmp = n[j-1];
        n[j-1] = n[j];
        n[j] = tmp;
      }
  }
  return n;
}
```

```java
/* Clone#2 */
private int[] sort1 (int[] n) {
for (int i=n.length-1; i>=0; i--)
for (int j=1; j<=i; j++) {
if (n[j] < n[j-1]) {
int tmp = n[j-1];
n[j-1] = n[j]; n[j] = tmp;
}} return n;
}
```

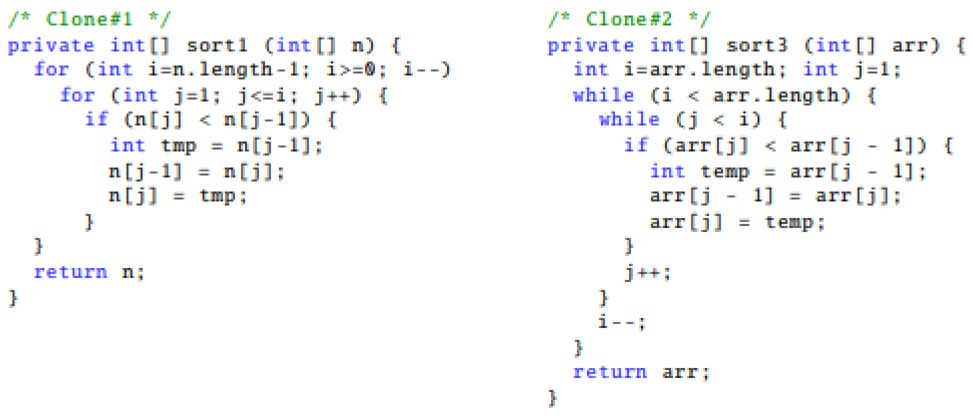
Figure 01: Type-1 clone pair
Source: “Similarity and Clone Search in Large-Scale Source Code Data”, Chaiyong Ragkhitwetsagul, page no. 42



```java
/* Clone#1 */
private int[] sort1 (int[] n) {
  for (int i=n.length-1; i>=0; i--)
    for (int j=1; j<=i; j++) {
      if (n[j] < n[j-1]) {
        int tmp = n[j-1];
        n[j-1] = n[j];
        n[j] = tmp;
      }
  }
  return n;
}
```

```java
/* Clone#2 */
private double[] sort2 (double[] arr) {
  for (int i=arr.length-1; i>=0; i--)
  for (int j=1; j<=i; j++) {
    if (arr[j] < arr[j-1]) {
      double temp = arr[j-1];
      arr[j-1] = arr[j]; arr[j] = temp;
    }
  } return arr;
}
```

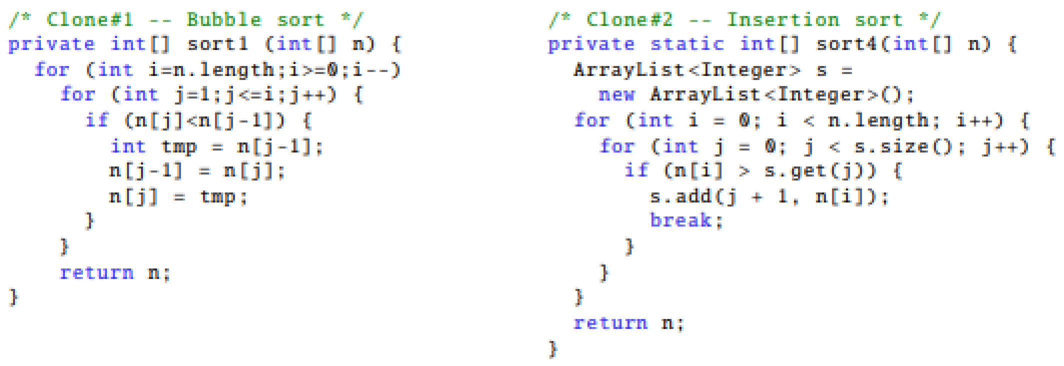
Figure 02: Type-2 clone pair
Source: “Similarity and Clone Search in Large-Scale Source Code Data”, Chaiyong Ragkhitwetsagul, page no. 42



```java
/* Clone#1 */
private int[] sort1 (int[] n) {
  for (int i=n.length-1; i>=0; i--)
    for (int j=1; j<=i; j++) {
      if (n[j] < n[j-1]) {
        int tmp = n[j-1];
        n[j-1] = n[j];
        n[j] = tmp;
      }
  }
  return n;
}
```

```java
/* Clone#2 */
private int[] sort3 (int[] arr) {
  int i=arr.length; int j=1;
  while (i < arr.length) {
    while (j < i) {
      if (arr[j] < arr[j - 1]) {
        int temp = arr[j - 1];
        arr[j - 1] = arr[j];
        arr[j] = temp;
      }
      j++;
    }
    i--;
  }
  return arr;
}
```

Figure 03: Type-3 clone pair
Source: “Similarity and Clone Search in Large-Scale Source Code Data”, Chaiyong Ragkhitwetsagul, page no. 42



```java
/* Clone#1 -- Bubble sort */
private int[] sort1 (int[] n) {
  for (int i=n.length;i>=0;i--)
    for (int j=1;j<=i;j++) {
      if (n[j]<n[j-1]) {
        int tmp = n[j-1];
        n[j-1] = n[j];
        n[j] = tmp;
      }
    }
    return n;
}
```

```java
/* Clone#2 -- Insertion sort */
private static int[] sort4(int[] n) {
  ArrayList<Integer> s =
    new ArrayList<Integer>();
  for (int i = 0; i < n.length; i++) {
    for (int j = 0; j < s.size(); j++) {
      if (n[i] > s.get(j)) {
        s.add(j + 1, n[i]);
        break;
      }
    }
  }
  return n;
}
```

Figure 04: Type-4 clone pair
Source: “Similarity and Clone Search in Large-Scale Source Code Data”, Chaiyong Ragkhitwetsagul, page no. 42

2.2.2 Local Code Similarity Detection with Winnowing

Among various clone detection techniques, the token-based approach is the most popular approach in code similarity because of its simplicity, flexibility, and scalability in code matching. Winnowing is a document fingerprinting algorithm which relies on the concept of n-gram or n-gram tokens.

N-gram (k-gram, q-gram, or k-shingle) is a contiguous sequence of n-size substrings of a given string. Given a string S and a number n, S [i, i + n] is an n-gram of S starting at the i-th character. The size of n can be varied and carefully chosen to suit the task. n-grams are suitable for partial matching in the text [Li et al., 2007] and code [Schleimer et al., 2003]

Winnowing algorithm converts a source code string into n-grams, computes a hash sequence of all n-grams, and creates a sliding window over the sequence to choose a fingerprint. The set of fingerprints is compared with the fingerprints from other programs to get statistical measures for similarity. This algorithm exhibits desirable three properties that are important for measuring similarity[15],[16]:

1. **Whitespace insensitivity:** In matching text files, matches should be unaffected by such things as extra whitespace, capitalization, punctuation, etc. In other domains, the notion of what strings should be equal is different—for example, in matching software text it is desirable to make matching insensitive to variable names.
2. **Noise suppression:** Discovering short matches, such as the fact that the word the appears in two different documents, is uninteresting. Any match must be large enough to imply that the material has been copied and is not simply a common word or idiom of the language in which documents are written.

3. **Position independence:** Coarse-grained permutation of the contents of a document (e.g., scrambling the order of paragraphs)should not affect the set of discovered matches. Adding to a document should not affect the set of matches in the original portion of the new document. Removing part of a document should not affect the set of matches in the portion that remains.

A do run runrun, a do run run
(a) Some text.

Adorunrunrunadorunrun
(b) The text with irrelevant features removed.

Adoru dorun orunr runru unrun nrunr runru
unrun nruna runad unado nador adoru dorun
orunr runru unrun
(c) The sequence of 5-grams derived from the text.

77 74 42 17 98 50 17 98 8 88 67 39 77 74 42
17 98
(d) A hypothetical sequence of hashes of the 5-grams.

(77, 74, 42, 17) (74, 42, 17, 98)
(42, 17, 98, 50) (17, 98, 50, 17)
(98, 50, 17, 98) (50, 17, 98, 8)
(17, 98, 8, 88) (98, 8, 88, 67)
(8, 88, 67, 39) (88, 67, 39, 77)
(67, 39, 77, 74) (39, 77, 74, 42)
(77, 74, 42, 17) (74, 42, 17, 98)
(e) Windows of hashes of length 4.

17 17 8 39 17
(f) Fingerprints selected by winnowing

[17,3] [17,6] [8,8] [39,11] [17,15]
(g) Fingerprints paired with 0-base positional information.

Figure 05: Winnowing sample text.
Source: Winnowing: Local Algorithms for Document Fingerprinting, Alex Aiken, page no. 1

2.3 Rolling Hash(Rabin-Karp algorithm)

Rabin-Karp's algorithm for fast substring matching is apparently the earliest version of fingerprinting based on k-grams. Karp and Rabin propose a "rolling" hash function that allows the hash for the i+1stk-gram to be computed quickly from the hash of the ith k-gram[17]. Treat a k-gram c1 ...ck as a k-digit number in some base b. The hash H (c_1 ...c_k) of c_1 ...c_k is this number:
$c_1 * b^{k-1} + c_2 * b^{k-2} + . . . + c_{k-1} * b + c_k$
To compute the hash of the k-gram c2 ...ck+1, we need only subtract out the high-order digit, multiply by b, and add in the new low-order digit. Thus, we have the identity:
$H (c_2 . . . c_{k+1}) = (H (c_1 . . . c_k) - c_k * b^{k-1}) * b + c_{k+1}$

Since b^{k-1} is a constant, this allows each subsequent hash to be computed from the previous one with only two additions and two multiplications.

2.4 Dice Coefficient

The Sorensen–Dice coefficient is a statistic used to gauge the similarity of two samples. Given any two sets of data A and B it is defined as:

$$\begin{aligned} Dice(A,B) &= \frac{2|AB|}{|A|+|B|} \\ &= \frac{2|AB|}{(|AB|+|A\setminus B|)+(|AB|+|B\setminus A|)} \\ &= \frac{|AB|}{|AB|+\frac{1}{2}|A\setminus B|+\frac{1}{2}|B\setminus A|} \end{aligned}$$

2.5 Jaccard Similarity

The Jaccard index, also known as the Jaccard similarity coefficient, is a statistic used for gauging the similarity and diversity of sample sets. The Jaccard coefficient measures similarity between finite sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets A and B as:

$$\begin{aligned} Jacc(A,B) &= \frac{|AB|}{|A\cup B|} \\ &= \frac{|AB|}{|AB|+|A\setminus B|+|B\setminus A|} \end{aligned}$$

3. Description of the Project

The overall project consists of the following two major parts

- Source code metrics (LOC and Halstead)
- Clone detection

To implement the project I have followed several processes step by step. SCA reads a directory containing 'C' source code files. The detailed process of generating the LOC and Halstead metrics and code clones is described below:

At first, we can see the overview of the full project from figure-06 below:

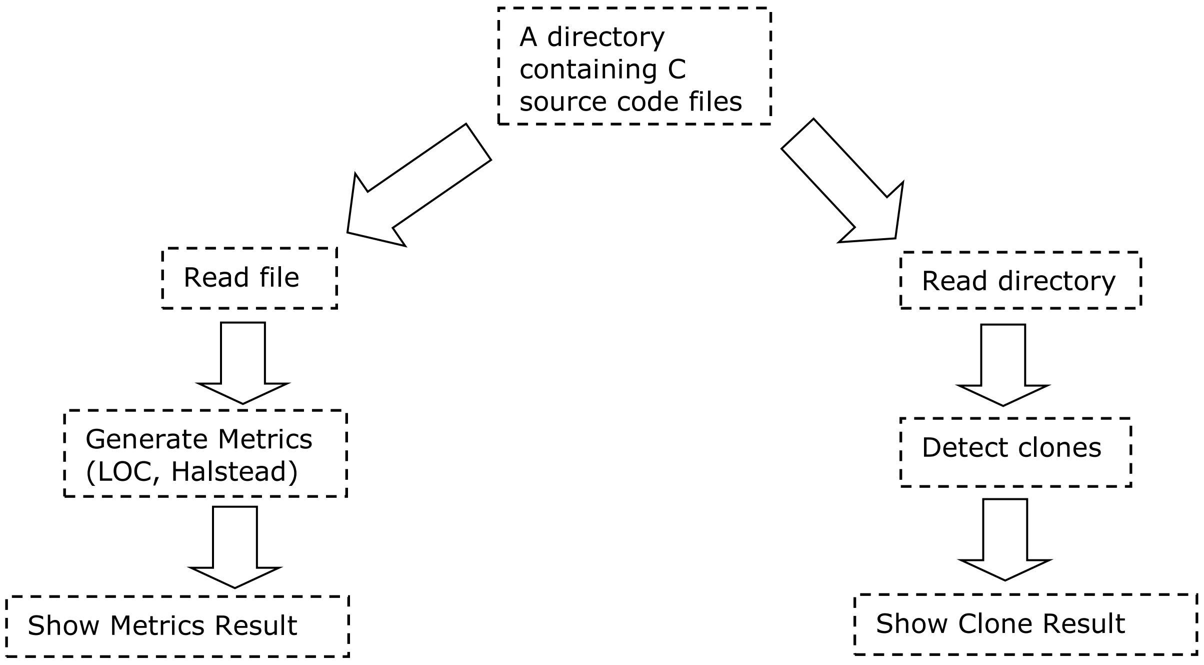

A directory containing C source code files
Read file
Generate Metrics (LOC, Halstead)
Show Metrics Result
Read directory
Detect clones
Show Clone Result

Figure 06: Overview of the Full Project

3.1 LOC metrics

In this feature, I have followed the counting rules of SLOC according to SEI and the IEEE's standards. At first, a single C source code file is taken as input from the user. The user can input the file path or choose a file to be processed. Following the counting rules and using several string manipulations (linear searching, substring generation, string matching) SCA determines the below metrics:

- Number of Blank Lines
- Number of Physical Lines
- Number of Logical Statements
- Number of Comment Lines
- Number of Comment and Statement Lines
- Number of Statement Lines

The processes of calculating LOC metrics is shown in figure-07 below:

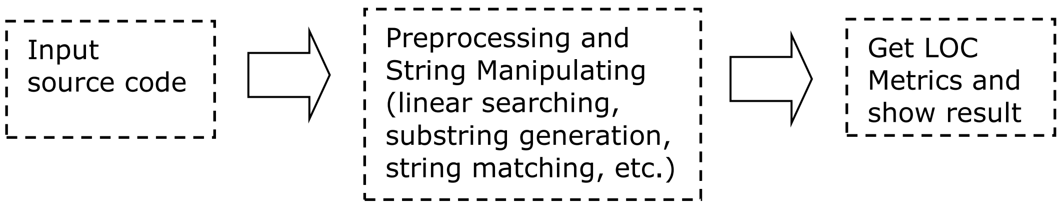

Input source code
Preprocessing and String Manipulating (linear searching, substring generation, string matching, etc.)
Get LOC Metrics and show result

Figure 07: LOC metrics calculation Process

3.2 Halstead Metrics

Halstead metrics generation is a core feature of my project. To implement this feature, I had to go through several steps one by one.

At first, the code is gone through several preprocessing such as the removal of blank lines, single and multiline comments, function declarations, hash directives, etc.

Then the code is passed for tokenization and12 types of tokens are generated from the code and they are categorized into 2 major types of tokens:

- Operator Tokens (keywords, looping statements, control statements, brackets, single and double operators, etc.)
- Operand Tokens (string, character, number, type specifier, identifier, etc.)

Then counting the tokens and doing mathematical calculations we get the following Halstead "software science" metrics:

Halstead metrics:

1. the number of distinct operators(n1)
2. the number of distinct operands(n2)
3. the total number of operators(N1)
4. the total number of operands(N2)

Halstead measures:

1. Program vocabulary
2. Program length
3. Calculated program length
4. Volume
5. Difficulty
6. Effort
7. Time Required to Program
8. Number of delivered bugs

The Halstead metrics generation process flow chart is given below in figure-08:

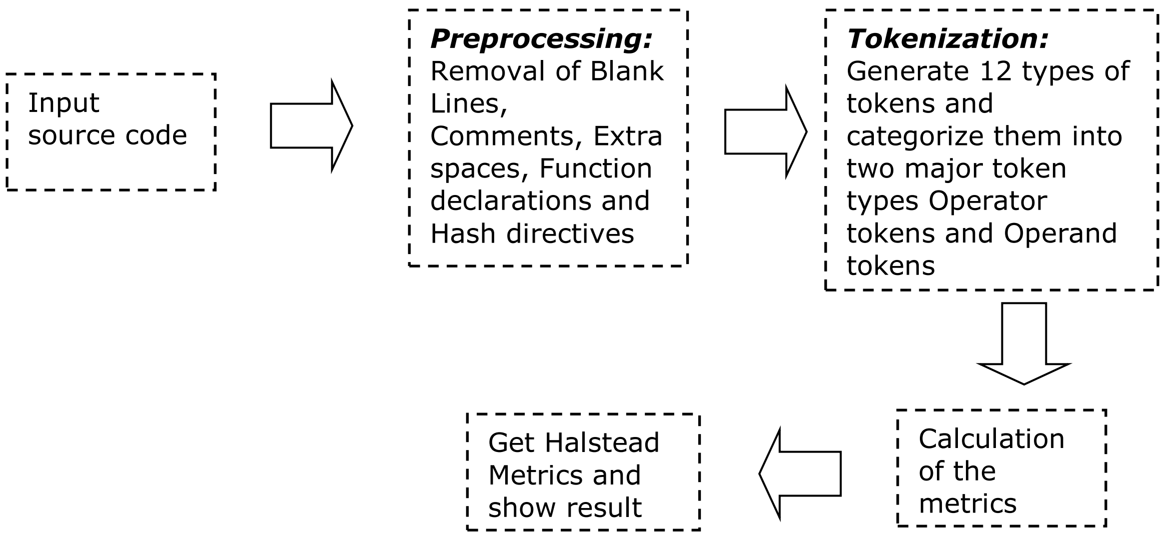

Input source code
Preprocessing: Removal of Blank Lines, Comments, Extra spaces, Function declarations and Hash directives
Tokenization: Generate 12 types of tokens and categorize them into two major token types Operator tokens and Operand tokens
Calculation of the metrics
Get Halstead Metrics and show result

Figure 08: Halstead metrics calculation Process

3.3 Clone Detection

Clone detection is another core feature of my project. I have followed the token-based approach and application of the winnowing algorithm to detect syntactic code clones among source codes. For this I had to go through several processes step by step.

At first the source code is preprocessed such as removal of blank lines, comments, hash directives, punctuations, extra white space, and capitalization of letters. Then from the preprocessed code k-grams are generated. The k-grams are then use for hashing and using rolling hash we get the hash values of the k-grams. Next the k-gram hashes are pushed into winnowing algorithm. The winnowing algorithm generates fingerprints from the hash values. Then comparing the fingerprints using statistical methods (Jaccard Similarity and Dice Similarity Coefficient), we get the clone result. The process is shown below in figure-09:

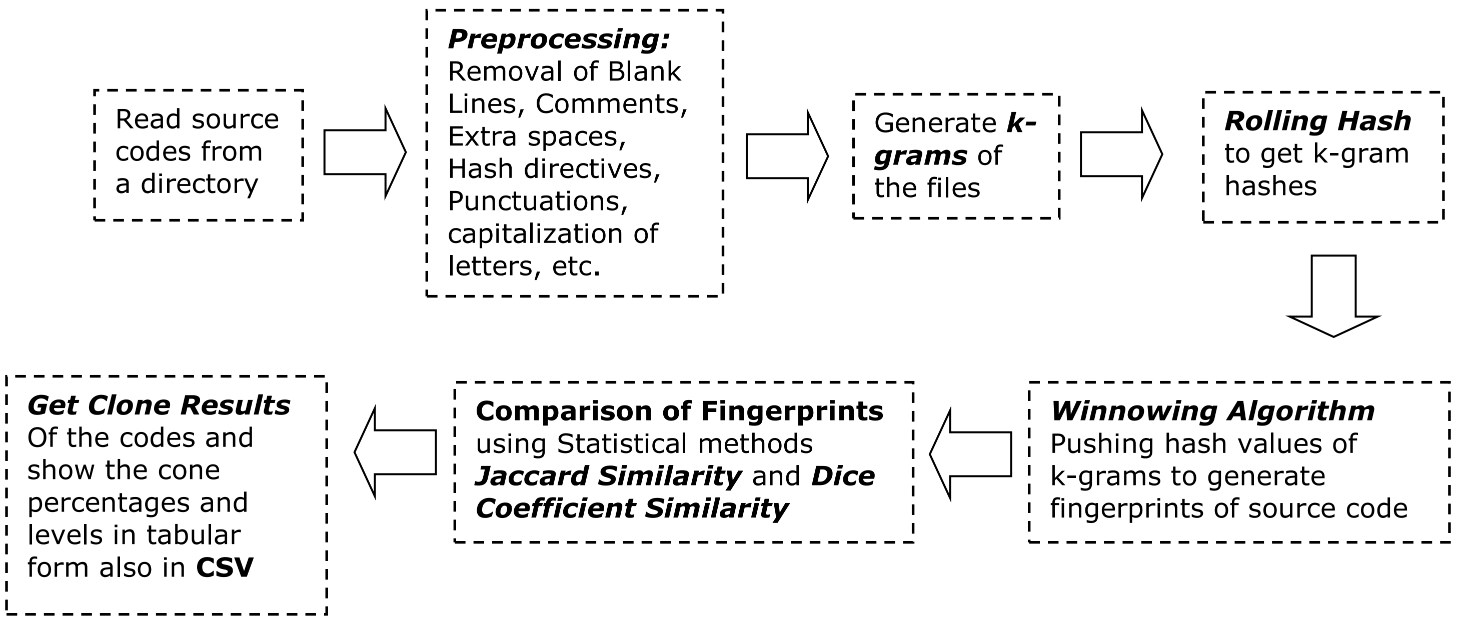
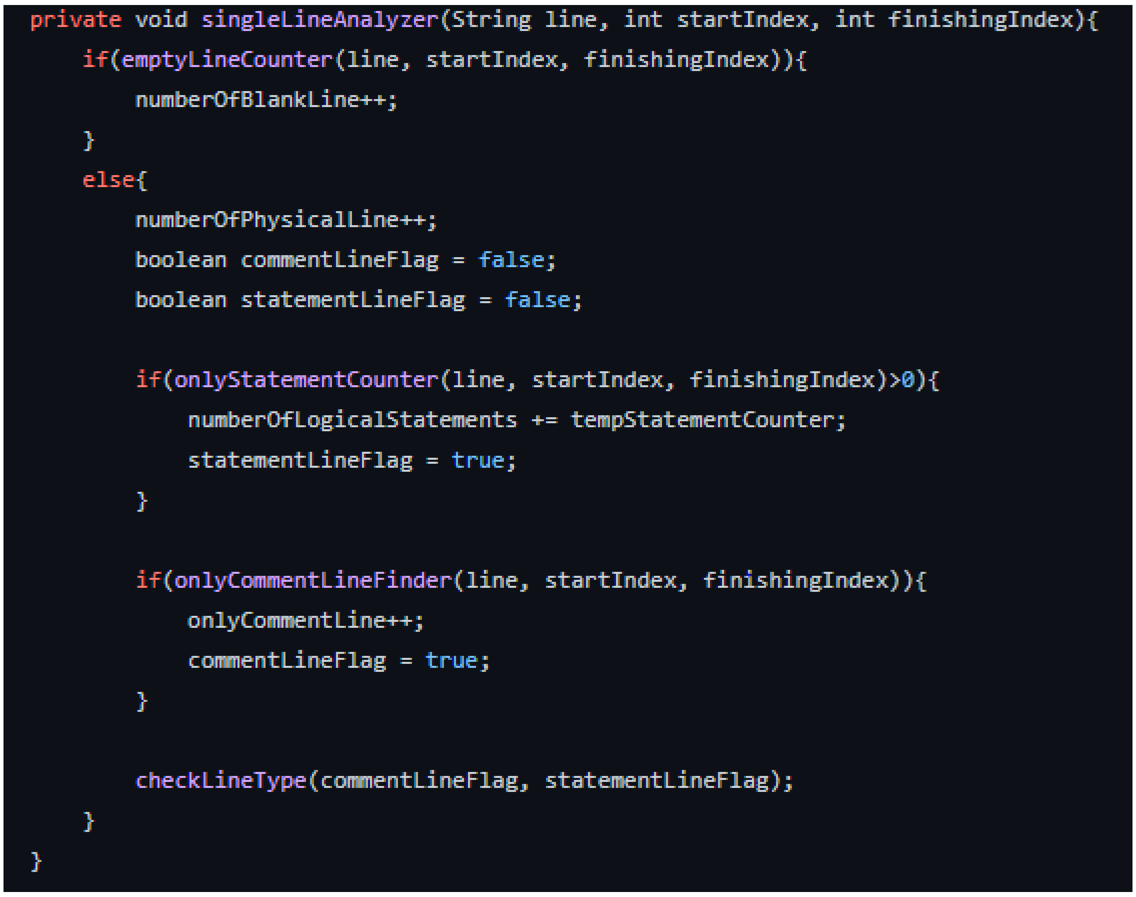

Read source codes from a directory
Preprocessing: Removal of Blank Lines, Comments, Extra spaces, Hash directives, Punctuations, capitalization of letters, etc.
Generate k-grams of the files
Rolling Hash to get k-gram hashes
Winnowing Algorithm Pushing hash values of k-grams to generate fingerprints of source code
Comparison of Fingerprints using Statistical methods Jaccard Similarity and Dice Coefficient Similarity
Get Clone Results Of the codes and show the cone percentages and levels in tabular form also in CSV

Figure 09: Clone Detection Process

4. Implementation and Testing

4.1 LOC Implementation

To implement this feature, I have used various string manipulations such as substring generation, string matching, linear searching and used LOC counting logic to determine the metrics. An important method of this feature is shown below in figure-10 which helps analyzes the source code line by line:

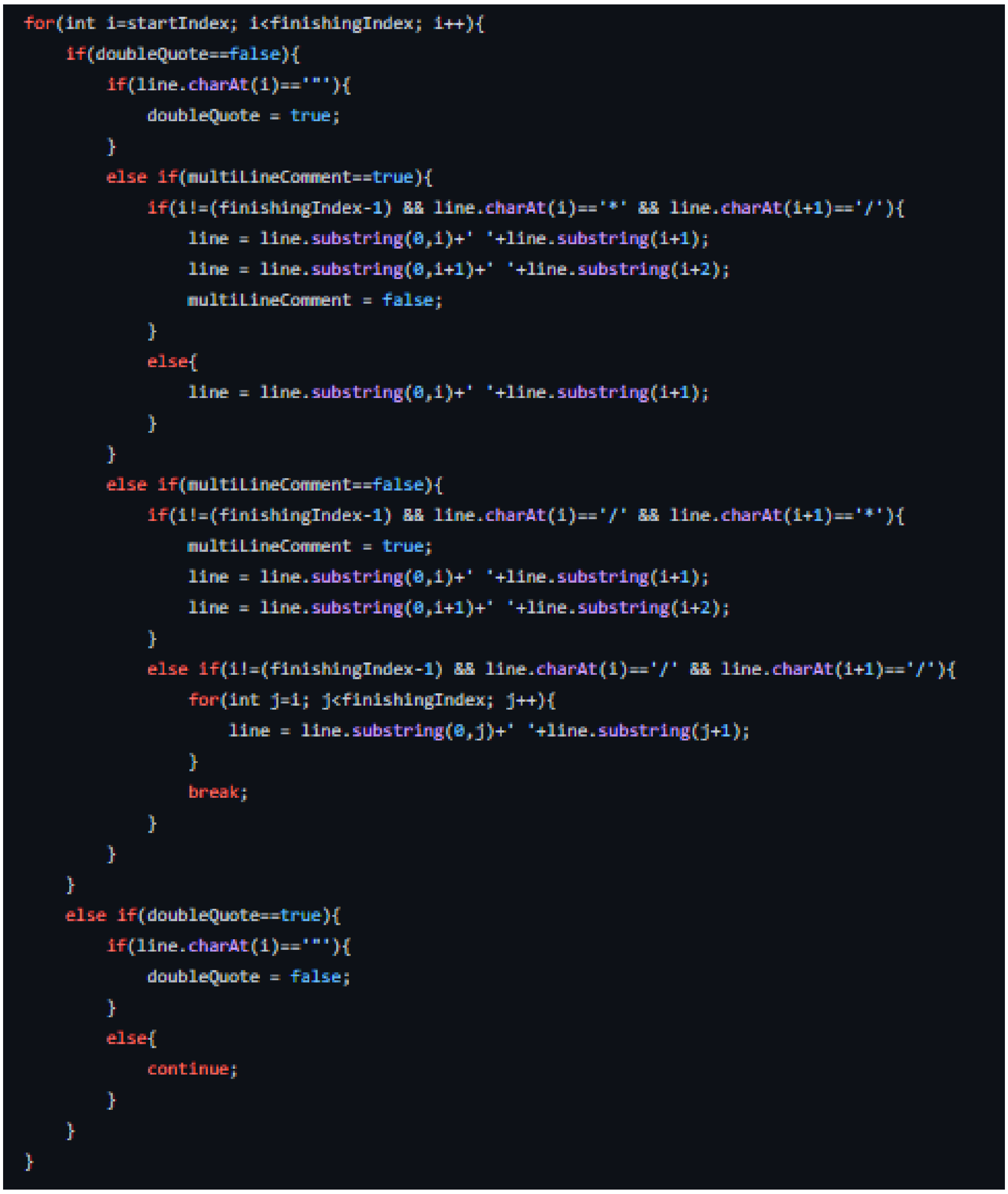


```
private void singleLineAnalyzer(String line, int startIndex, int finishingIndex){
    if(emptyLineCounter(line, startIndex, finishingIndex)){
        numberOfBlankLine++;
    }
    else{
        numberOfPhysicalLine++;
        boolean commentLineFlag = false;
        boolean statementLineFlag = false;

        if(onlyStatementCounter(line, startIndex, finishingIndex)>0){
            numberOfLogicalStatements += tempStatementCounter;
            statementLineFlag = true;
        }

        if(onlyCommentLineFinder(line, startIndex, finishingIndex)){
            onlyCommentLine++;
            commentLineFlag = true;
        }

        checkLineType(commentLineFlag, statementLineFlag);
    }
}
```

Figure 10: LOC singleLineAnalyzer method which analyzes the source code line by line.

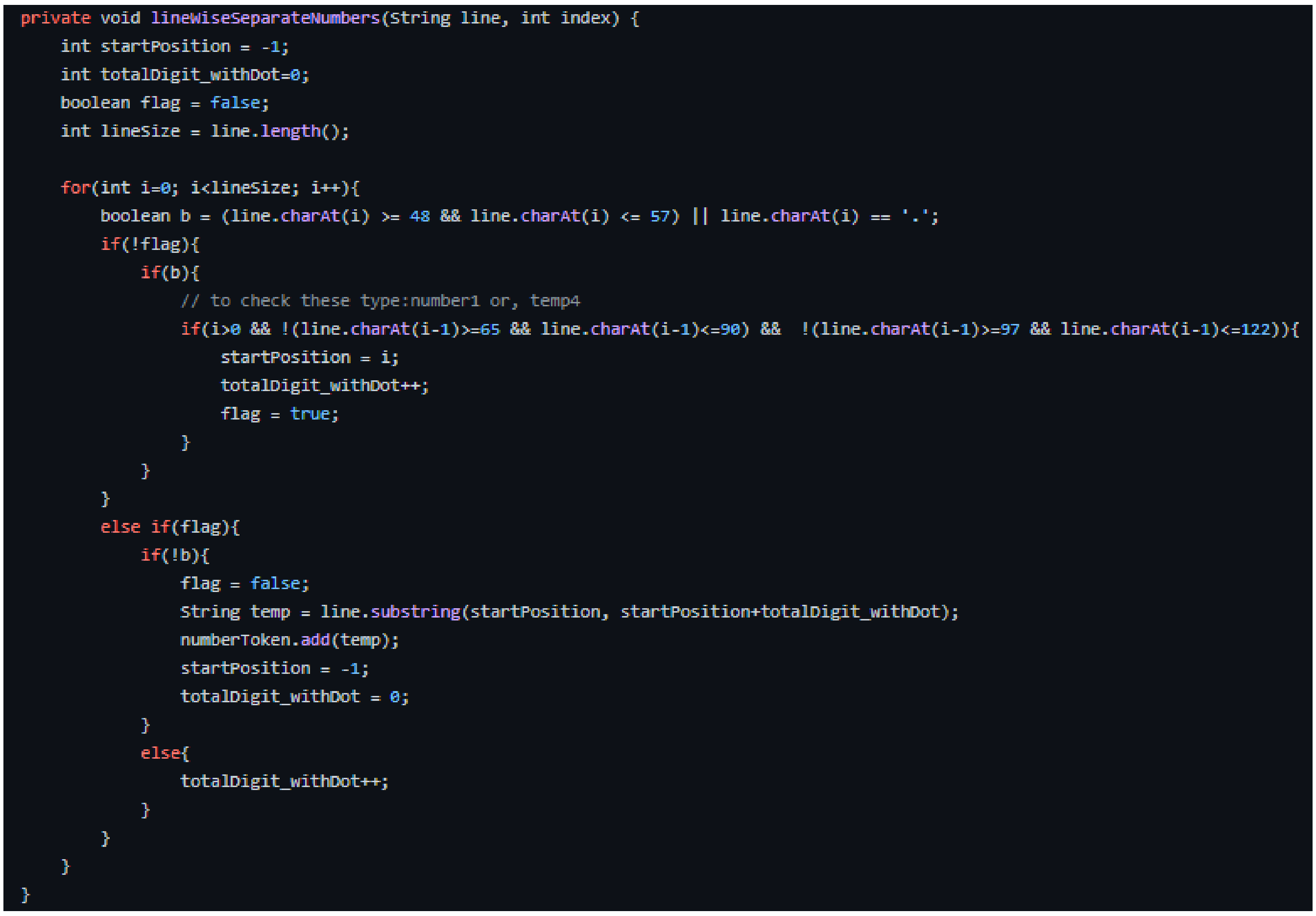
4.2 Halstead Implementation

Implementation of this feature took several preprocessing of source code, tokenization, token counting and mathematical calculations. An example of preprocessing to remove comments is shown in figure-11. An example of tokenization to generate number tokens is shown in figure-12.



```java
for(int i=startIndex; i<finishingIndex; i++){
    if(doubleQuote==false){
        if(line.charAt(i)=='"'){
            doubleQuote = true;
        }
        else if(multiLineComment==true){
            if(i!=(finishingIndex-1) && line.charAt(i)=='*' && line.charAt(i+1)=='/'){
                line = line.substring(0,i)+' '+line.substring(i+1);
                line = line.substring(0,i+1)+' '+line.substring(i+2);
                multiLineComment = false;
            }
            else{
                line = line.substring(0,i)+' '+line.substring(i+1);
            }
        }
        else if(multiLineComment==false){
            if(i!=(finishingIndex-1) && line.charAt(i)=='/' && line.charAt(i+1)=='*'){
                multiLineComment = true;
                line = line.substring(0,i)+' '+line.substring(i+1);
                line = line.substring(0,i+1)+' '+line.substring(i+2);
            }
            else if(i!=(finishingIndex-1) && line.charAt(i)=='/' && line.charAt(i+1)=='/'){
                for(int j=i; j<finishingIndex; j++){
                    line = line.substring(0,j)+' '+line.substring(j+1);
                }
                break;
            }
        }
    }
    else if(doubleQuote==true){
        if(line.charAt(i)=='"'){
            doubleQuote = false;
        }
        else{
            continue;
        }
    }
}
```

Figure 11: A code block of Preprocessing(Comment Removing)

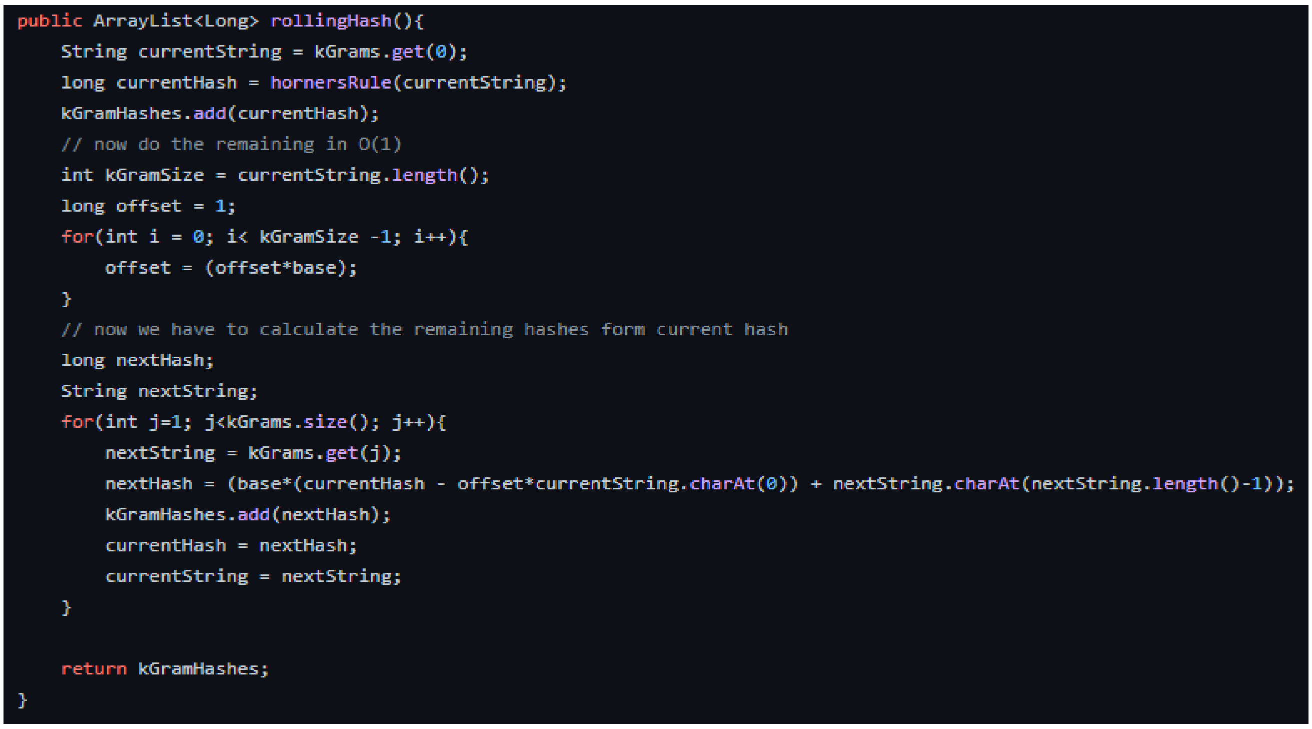


```java
private void lineWiseSeparateNumbers(String line, int index) {
    int startPosition = -1;
    int totalDigit_withDot=0;
    boolean flag = false;
    int lineSize = line.length();

    for(int i=0; i<lineSize; i++){
        boolean b = (line.charAt(i) >= 48 && line.charAt(i) <= 57) || line.charAt(i) == '.';
        if(!flag){
            if(b){
                // to check these type:number1 or, temp4
                if(i>0 && !(line.charAt(i-1)>=65 && line.charAt(i-1)<=90) &&  !(line.charAt(i-1)>=97 && line.charAt(i-1)<=122)){
                    startPosition = i;
                    totalDigit_withDot++;
                    flag = true;
                }
            }
        }
        else if(flag){
            if(!b){
                flag = false;
                String temp = line.substring(startPosition, startPosition+totalDigit_withDot);
                numberToken.add(temp);
                startPosition = -1;
                totalDigit_withDot = 0;
            }
            else{
                totalDigit_withDot++;
            }
        }
    }
}
```

Figure 12: A code block of Tokenization (Number tokens)

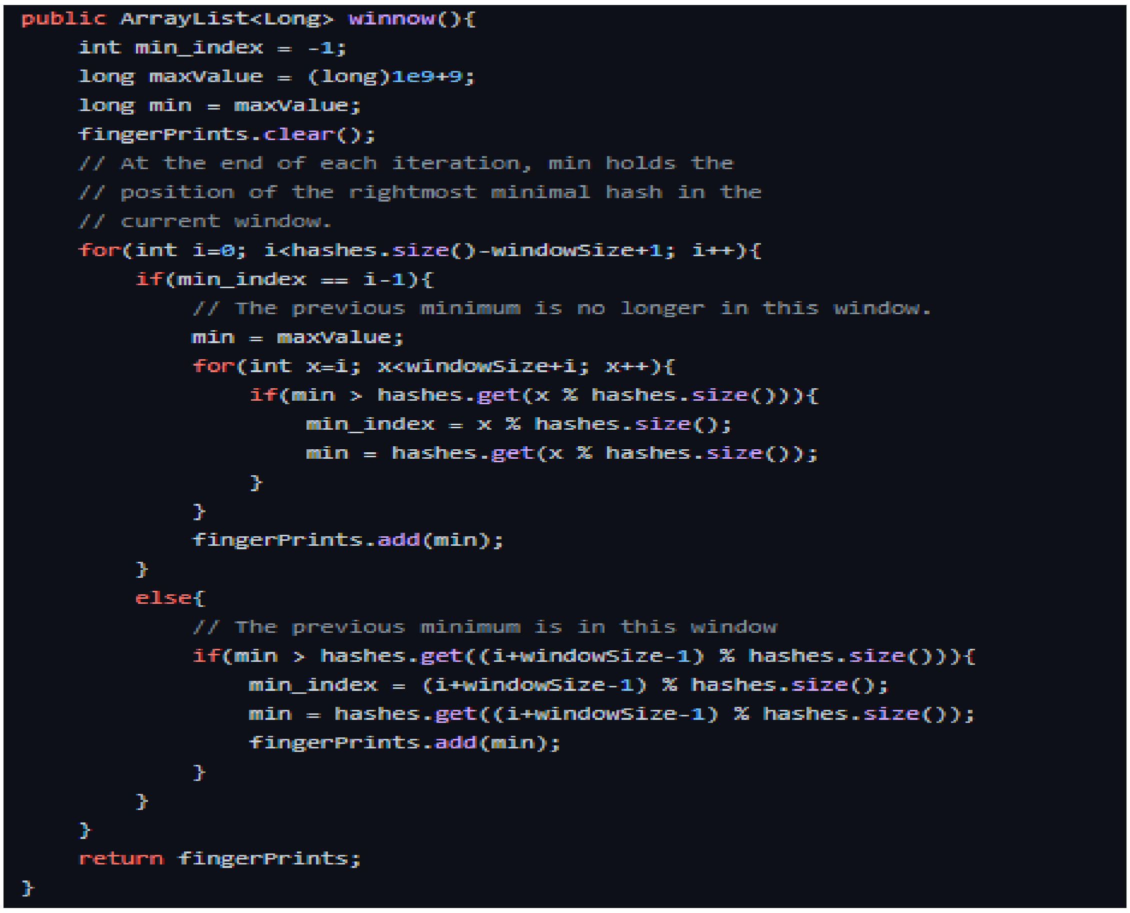
4.3 Clone Detection Implementation

To implement this feature, I had to follow several processes step by step– at first, preprocessing, then k-gram generation, hashing the k-grams, pushing into winnowing algorithm to generate fingerprints, and using Jaccard and Dice coefficient similarity to get clone results. A code block of the rolling hash is shown in figure-13. The implementation of the winnowing algorithm is shown in figure-14.



```java
public ArrayList<Long> rollingHash(){
    String currentString = kGrams.get(0);
    long currentHash = hornersRule(currentString);
    kGramHashes.add(currentHash);
    // now do the remaining in O(1)
    int kGramSize = currentString.length();
    long offset = 1;
    for(int i = 0; i< kGramSize -1; i++){
        offset = (offset*base);
    }
    // now we have to calculate the remaining hashes form current hash
    long nextHash;
    String nextString;
    for(int j=1; j<kGrams.size(); j++){
        nextString = kGrams.get(j);
        nextHash = (base*(currentHash - offset*currentString.charAt(0)) + nextString.charAt(nextString.length()-1));
        kGramHashes.add(nextHash);
        currentHash = nextHash;
        currentString = nextString;
    }

    return kGramHashes;
}
```

Figure 13: Rolling hash (Hashing of k-grams)



```java
public ArrayList<Long> winnow(){
    int min_index = -1;
    long maxValue = (long)1e9+9;
    long min = maxValue;
    fingerPrints.clear();
    // At the end of each iteration, min holds the
    // position of the rightmost minimal hash in the
    // current window.
    for(int i=0; i<hashes.size()-windowSize+1; i++){
        if(min_index == i-1){
            // The previous minimum is no longer in this window.
            min = maxValue;
            for(int x=i; x<windowSize+i; x++){
                if(min > hashes.get(x % hashes.size())){
                    min_index = x % hashes.size();
                    min = hashes.get(x % hashes.size());
                }
            }
            fingerPrints.add(min);
        }
        else{
            // The previous minimum is in this window
            if(min > hashes.get((i+windowSize-1) % hashes.size())){
                min_index = (i+windowSize-1) % hashes.size();
                min = hashes.get((i+windowSize-1) % hashes.size());
                fingerPrints.add(min);
            }
        }
    }
    return fingerPrints;
}
```

Figure 14: Implementation of Winnowing Algorithm

5. User Interface

User Interfaces are shown below:

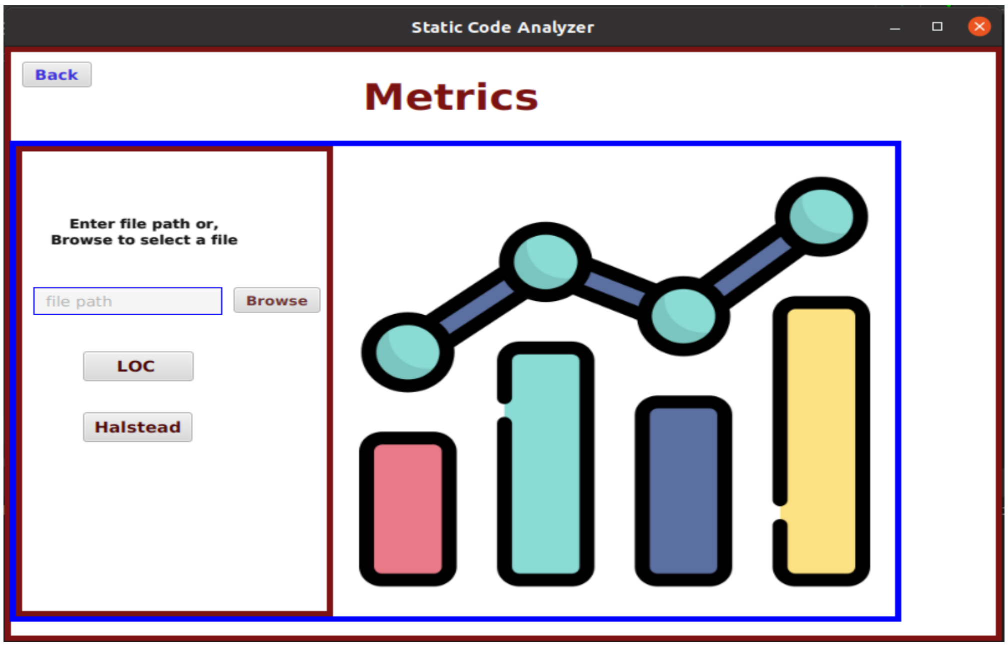
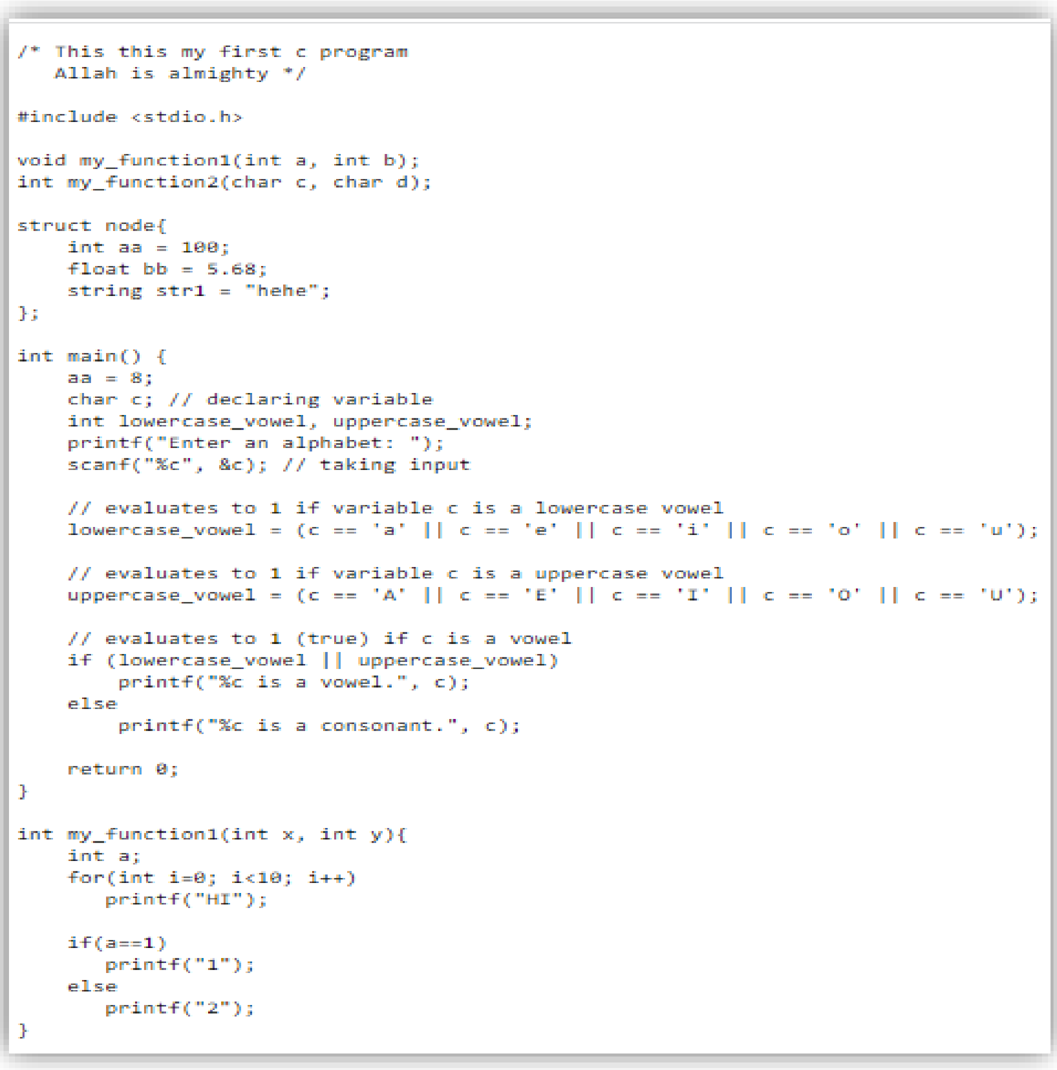
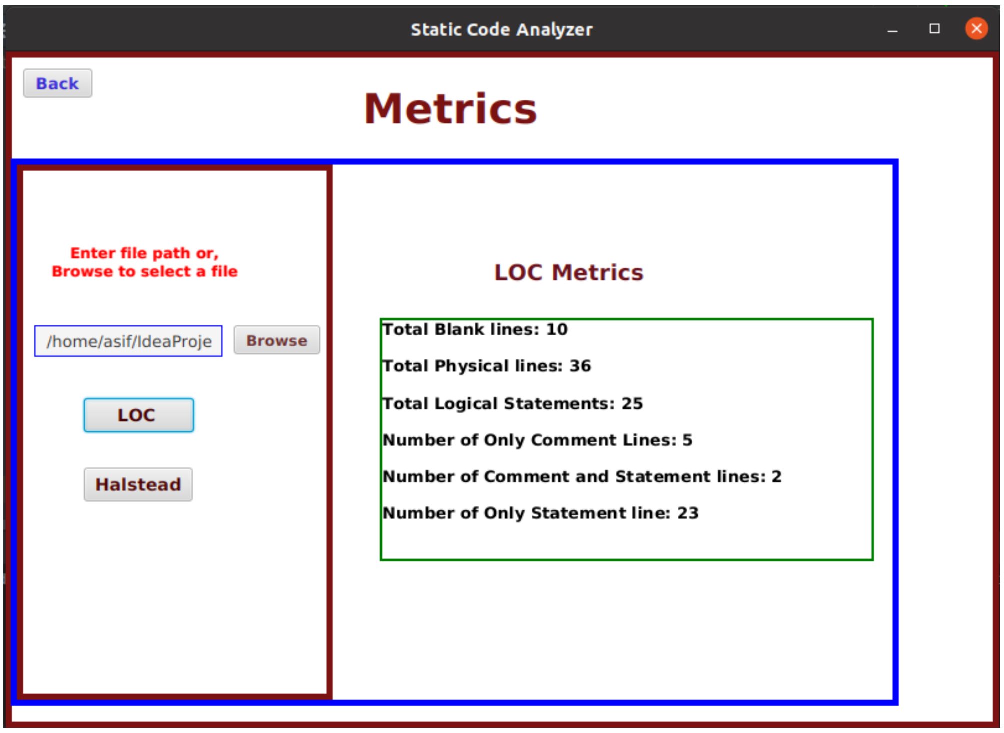


Figure 15: Metrics Main window



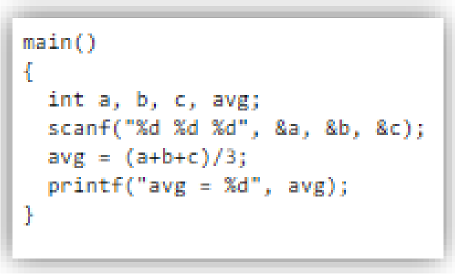
```
/* This this my first c program
   Allah is almighty */

#include <stdio.h>

void my_function1(int a, int b);
int my_function2(char c, char d);

struct node{
    int aa = 100;
    float bb = 5.68;
    string str1 = "hehe";
};

int main() {
    aa = 8;
    char c; // declaring variable
    int lowercase_vowel, uppercase_vowel;
    printf("Enter an alphabet: ");
    scanf("%c", &c); // taking input

    // evaluates to 1 if variable c is a lowercase vowel
    lowercase_vowel = (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u');

    // evaluates to 1 if variable c is a uppercase vowel
    uppercase_vowel = (c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U');

    // evaluates to 1 (true) if c is a vowel
    if (lowercase_vowel || uppercase_vowel)
        printf("%c is a vowel.", c);
    else
        printf("%c is a consonant.", c);

    return 0;
}

int my_function1(int x, int y){
    int a;
    for(int i=0; i<10; i++)
       printf("HI");

    if(a==1)
       printf("1");
    else
       printf("2");
}
```

Figure 16(i): LOC input code LOCtest.c



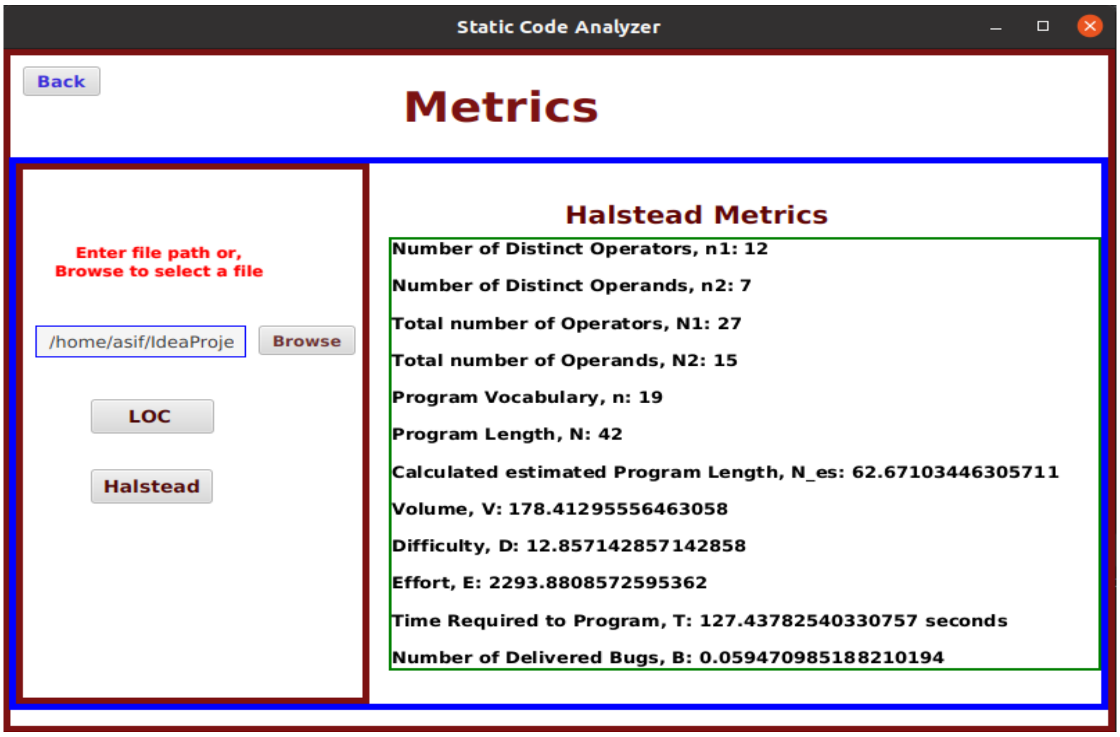
```
Total Blank lines: 10
Total Physical lines: 36
Total Logical Statements: 25
Number of Only Comment Lines: 5
Number of Comment and Statement lines: 2
Number of Only Statement line: 23
```

Figure 16(ii): LOC Output with respect to the input code LOCtest.c



```
main()
{
  int a, b, c, avg;
  scanf("%d %d %d", &a, &b, &c);
  avg = (a+b+c)/3;
  printf("avg = %d", avg);
}
```

Figure 17(i): Halstead input code test1.c



```
Number of Distinct Operators, n1: 12
Number of Distinct Operands, n2: 7
Total number of Operators, N1: 27
Total number of Operands, N2: 15
Program Vocabulary, n: 19
Program Length, N: 42
Calculated estimated Program Length, N_es: 62.67103446305711
Volume, V: 178.41295556463058
Difficulty, D: 12.857142857142858
Effort, E: 2293.8808572595362
Time Required to Program, T: 127.43782540330757 seconds
Number of Delivered Bugs, B: 0.059470985188210194
```

Figure 17: Halstead Output with respect to the input code test1.c

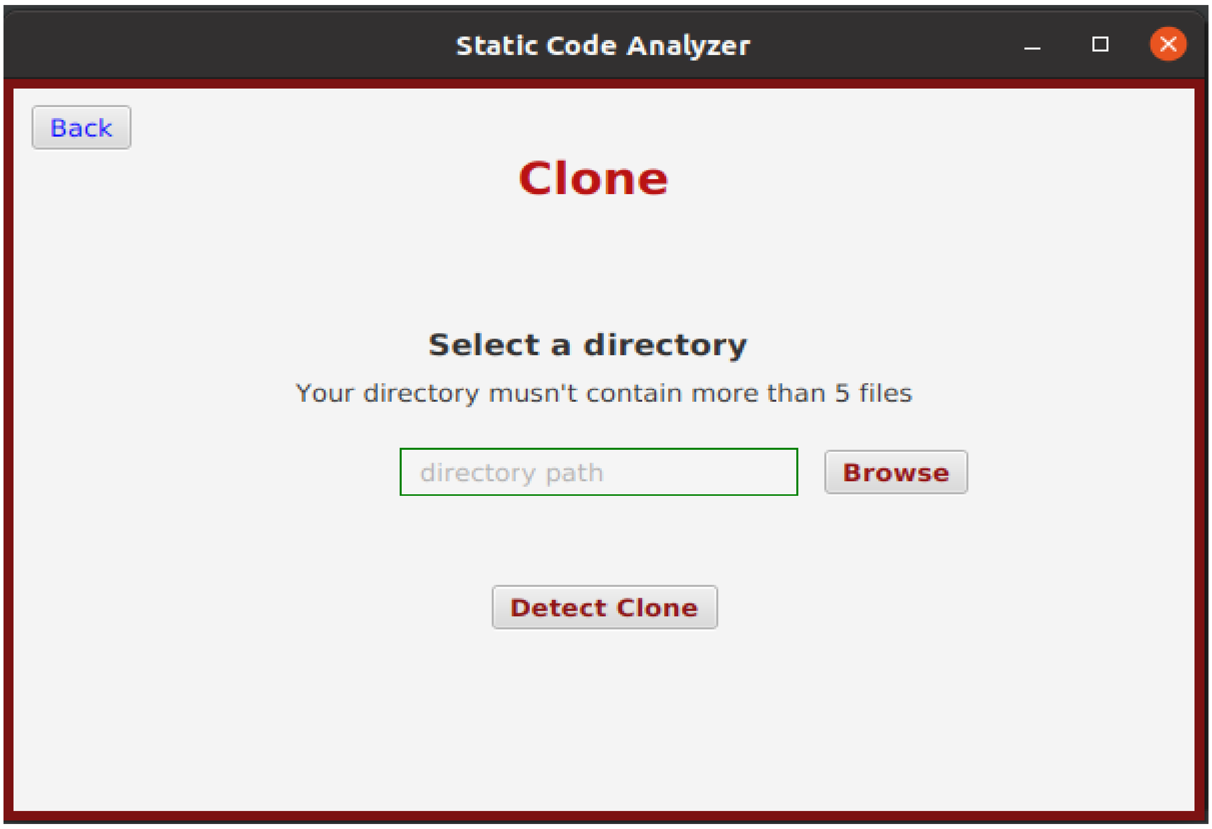
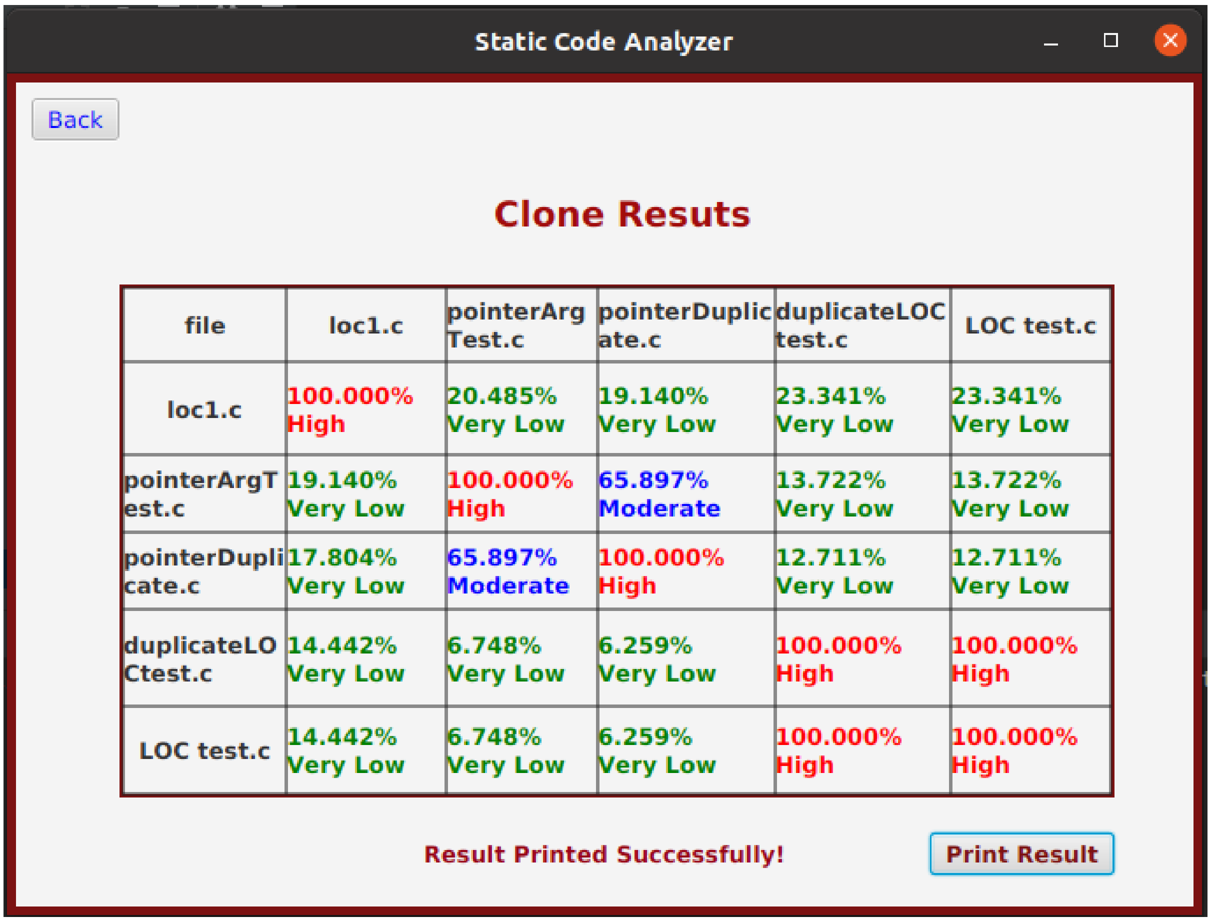


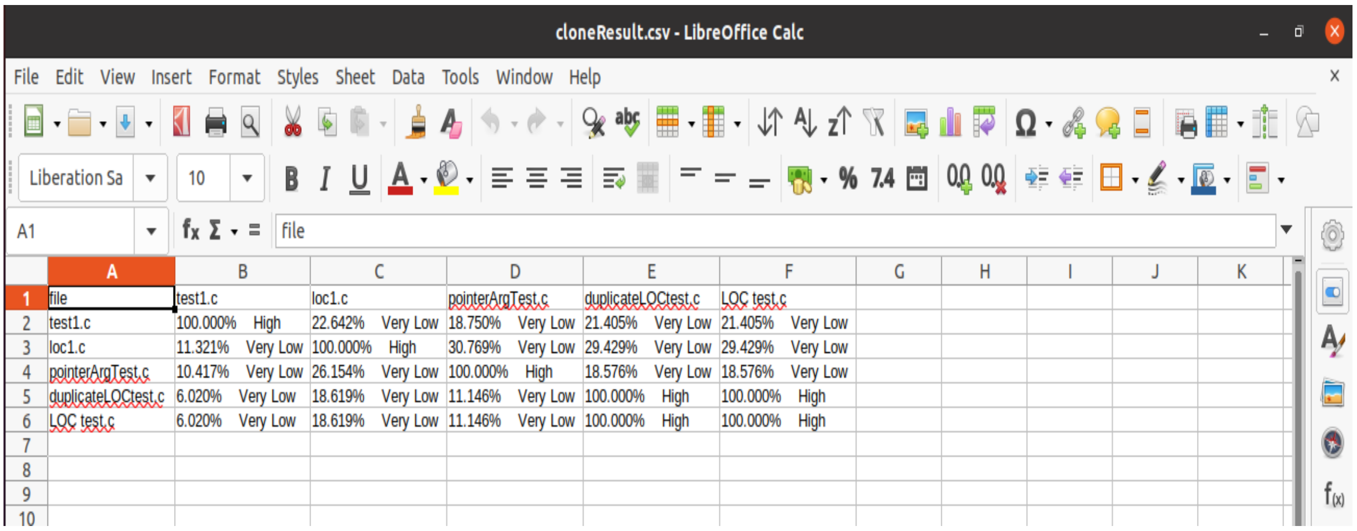
Figure 18: Clone Main Window



file	loc1.c	pointerArgTest.c	pointerDuplicate.c	duplicateLOCtest.c	LOC test.c
loc1.c	100.000% High	20.485% Very Low	19.140% Very Low	23.341% Very Low	23.341% Very Low
pointerArgTest.c	19.140% Very Low	100.000% High	65.897% Moderate	13.722% Very Low	13.722% Very Low
pointerDuplicate.c	17.804% Very Low	65.897% Moderate	100.000% High	12.711% Very Low	12.711% Very Low
duplicateLOCtest.c	14.442% Very Low	6.748% Very Low	6.259% Very Low	100.000% High	100.000% High
LOC test.c	14.442% Very Low	6.748% Very Low	6.259% Very Low	100.000% High	100.000% High

Result Printed Successfully!

Figure 19: Clone Result with respect to the input directory Clone-Test


cloneResult.csv - LibreOffice Calc

file	test1.c	loc1.c	pointerArgTest.c	duplicateLOCtest.c	LOC test.c
test1.c	100.000% High	22.642% Very Low	18.750% Very Low	21.405% Very Low	21.405% Very Low
loc1.c	11.321% Very Low	100.000% High	30.769% Very Low	29.429% Very Low	29.429% Very Low
pointerArgTest.c	10.417% Very Low	26.154% Very Low	100.000% High	18.576% Very Low	18.576% Very Low
duplicateLOCtest.c	6.020% Very Low	18.619% Very Low	11.146% Very Low	100.000% High	100.000% High
LOC test.c	6.020% Very Low	18.619% Very Low	11.146% Very Low	100.000% High	100.000% High

Figure 20: Clone result(printed) in CSV file

6. Challenges Faced

6.1 Challenges faced in determining LOC

While implementing this feature I faced a problem with the counting of logical lines of code. I was getting the wrong output for this. Then I had to study more about logical lines of code and a research paper helps in this issue which describes the SEI and IEEE's standards to count the logical lines of code.

6.2 Challenges faced in Halstead Metrics

Tokenization of the source code was a big challenge for me. As Halstead metrics require about twelve types of tokens for measurement. So, adding the logic and generating the tokens was very hard. I have followed the counting rules of Halstead for tokenizing the source code. Later I also faced problems with logarithmic calculations in java while calculating the Halstead metrics. I got help with this from oracle java[18].

6.3 Challenges faced in Code Clone Detection

The main challenge was to implement the winnowing algorithm. To implement the winnowing algorithm correctly I followed the procedure of Alex Aiken's MOSS. Window handling and generating fingerprints was a hard task. Handling files also was a major problem. The rolling hash function was a bit tricky which generates k-gram hashes. I faced problems in determining offset and indexing. Later I got help from stackoverflow[19] to solve this issue. The problem faced with statistical measurements of fingerprints is what to choose between Jaccard Similarity and Dice Coefficient Similarity to compare fingerprints. Later I studied and understand they are conceptually the same and produce the approximately same result. That's why I decided to get their average value of comparisons.

6.4 Challenges faced in GUI

I have used JavaFX to add GUI. The first challenge was to set up the JavaFX environment. Handling the fxml files and controller classes was very tricky for me. To overcome this problem, I have followed Jenkov's tutorials[20]. Scene swapping and adding subscene was very hard for me. Result showing of LOC and Halstead in listView and adding the benchmarks was tricky. Clone Result showing in Tabular format was very hard to me and I had to study more about this to solve this problem. Clone result printing in CSV format was another big issue. I was getting the wrong output because of wrong indexing. Later I got help from tutorialspoint[21].

7. Conclusion

Throughout the whole project, I have learned many important things about the statistical analysis of source codes. I have learned the differences between dynamic and statistical analysis of source codes. I learned about software metrics and their various important usages in software engineering. I acquired a broad knowledge about LOC and Halstead "software science" metrics and how they play a vital role in software quality assurance and software security management. I also learned about the different types of code clones and their problems in software management. Though I only used a hybrid token-based technique that uses the winnowing algorithm, I learned a little bit about many other clone detection techniques as well like textual approaches, tree-based approaches, graph-based approaches, compile-code-based approaches, etc.

The project can be extended by adding the feature of many other software metrics like cyclomatic complexity, function point metrics, etc. And for clone detection, we can add features of other clone detection techniques. For example, tree-based approaches (AST), graph-based approaches (PDG, CFG), etc. We can also add other features of static analysis of code. For example, bug detection and bug fixing, program control flow, program dependency, etc.

Reference

[1] A Study of Software Metrics, Gurdev Singh, Dilbag Singh, Vikram Singh, IJCEM International Journal of Computational Engineering & Management, Vol. 11, January 2011, page no. 23

[2] A SLOC Counting Standard, Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan Barry Boehm, Center for Systems and Software Engineering University of Southern California {nguyenvu, deedsrub, thomast, boehm}@usc.edu, page no. 9

[3] Halstead, M. H., Elements of Software Science, 1977, New York: Elsevier North-Holland.

[4] Leach, R. J., "Using Metrics to Evaluate Student Programs", ACM SIGCSE Bulletin, Vol. 27, No. 2, 1995, pp. 41-48

[5] Chuan, C. H., Lin, L., Ping, L. L., and Lian, L. V., "Evaluation of Query Languages with Software Science Metrics", in Proceedings of the IEEE Region 10's Ninth Annual International Conference on Frontiers of Computer Technology TENCON'94, 1994, Singapore, pp. 516-520.

[6] Bailey, C. T. and Dingee, W. L., "A Software Study Using Halstead Metrics", in Proceedings of the 1981 ACM Workshop / Symposium on Measurement and Evaluation of Software Quality, 1981, Maryland, USA, pp. 189-197

[7] Booth, S. P. and Jones, S. B., "Are Ours Really Smaller Than Theirs", in Glasgow Workshop on Functional Programming, 1996, Ullapool, Scotland, UK, pp. 1-7

[8] Al Qutaish, R. E., Incorporating Software Measurements into a Compiler, MSc thesis, Department of Computer Science, 1998, Serdang: Putra University of Malaysia

[9] Samoladas, 1., Stamelos, I., Angelis, L., and Oikonomou, A., "Open Source Software Development Should Strive for Even Greater Code Maintainability", Communication of ACM, Vol. 47, No. 10, 2004, pp. 83-8

[10] ISO/IEC, ISO/IEC IS 15939: Software Engineering - Software Measurement Process, 2002, Geneve: International Organization for Standardization.

[11] An analysis of the Design and definaitons of Halstead Metrics, Rafa E. Al Qutaish, Alian Abran, 15th International Workshop on Software Measurement (IWSM 2005), September 12-14, 2005, Monteral, Canada, pp. 337-352

[12] https://www.geeksforgeeks.org/software-engineering-halsteads-software-metrics/, Software Engineering | Halstead’s Software Metrics, 29 May 2022

[13] http://www.verifysoft.com/en_halstead_metrics.html , Measurement of Halstead Metrics, 29 May 2022

[14] “Code Similarity and Clone Search in Large-Scale Source Code Data”, Chaiyong Ragkhitwetsagul, Doctor of Philosophy, University College London, September 30, 2018

[15] MOSS (measure of software similarity), Alex Aiken, https://theory.stanford.edu/~aiken/moss/ , 29 May 2022

[16] Winnowing: Local Algorithms for Document Fingerprinting, Saul Schleimer, Daniel S. Wilkerson, Alex Aiken, SIGMOD 2003, June 9-12, 2003, San Diego, CA.

[17] https://cp-algorithms.com/string/string-hashing.html , String Hashing, 29 May 2022

[18] https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html , Class Math, 29 May 2022.

[19] https://stackoverflow.com/questions/52509368/how-to-efficiently-find-identical-substrings-of-a-specified-length-in-a-collecti , How to efficiently find identical substrings of a specified length in a collection of strings?, 29 May 2022.

[20] https://jenkov.com/tutorials/javafx/index.html , JavaFX Tutorial, 25 May 2022.

[21] https://www.tutorialspoint.com/how-to-write-data-to-csv-file-in-java , How to write data to .csv file in Java?, 29 May 2022.