# NORMALIZER: AUGMENTING CODE CLONE DETECTORS USING SOURCE CODE NORMALIZATION

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Kevin Ly

March 2017

COMMITTEE MEMBERSHIP

| | |
|---|---|
| TITLE: | Normalizer: Augmenting Code Clone Detectors using Source Code Normalization |
| AUTHOR: | Kevin Ly |
| DATE SUBMITTED: | March 2017 |
| COMMITTEE CHAIR: | Aaron Keen, Ph.D. Professor of Computer Science and Software Engineering |
| COMMITTEE MEMBER: | Hugh Smith, Ph.D. Associate Professor of Computer Science and Software Engineering |
| COMMITTEE MEMBER: | John Seng, Ph.D. Professor of Computer Science and Software Engineering |

ABSTRACT

Normalizer: Augmenting Code Clone Detectors using Source Code Normalization

Kevin Ly

Code clones are duplicate fragments of code that perform the same task. As software code bases increase in size, the number of code clones also tends to increase. These code clones, possibly created through copy-and-paste methods or unintentional duplication of effort, increase maintenance cost over the lifespan of the software. Code clone detection tools exist to identify clones where a human search would prove unfeasible, however the quality of the clones found may vary. I demonstrate that the performance of such tools can be improved by normalizing the source code before usage. I developed Normalizer, a tool to transform C source code to normalized source code where the code is written as consistently as possible. By maintaining the code's function while enforcing a strict format, the variability of the programmer's style will be taken out. Thus, code clones may be easier to detect by tools regardless of how it was written.

Reordering statements, removing useless code, and renaming identifiers are used to achieve normalized code. Normalizer was used to show that more clones can be found in Introduction to Computer Networks assignments by normalizing the source code versus the original source code using a small variety of code clone detection tools.

## ACKNOWLEDGMENTS

Thanks to:

- Professor Keen for his unending support and anxiety mitigation
- My family for their patience and patience and patience
- My committee for being good friends during university

TABLE OF CONTENTS

LIST OF TABLES

## LIST OF FIGURES

LIST OF LISTINGS

Listing Page

# Chapter 1

# INTRODUCTION

Code clones are duplicated code fragments that perform the same or a similar task. For example, a piece of code that capitalizes all vowels may be written as in Listing 1.1 or Listing 1.2.

**Listing 1.1: Capitalize vowels in a while loop**

```c
void capitalizeVowels(char *str) {
    char letter = *str;
    while (letter != '\0') {
        if (letter == 'a' || letter == 'e' || letter == 'i' || letter == 'o' ||
            letter == 'u') {
            *str = toupper(letter);
        }
        str++;
        letter = *str;
    }
}
```

**Listing 1.2: Capitalize vowels in a for loop**

```c
void capitalizeVowels(char *str) {
    for (int i = 0; i < strlen(str); i++) {
        char letter = str[i];
        if (letter == 'a' || letter == 'e' || letter == 'i' || letter == 'o' ||
            letter == 'u') {
            str[i] = toupper(letter);
        }
    }
}
```

The main difference in these clones is the choice between the `while` loop and `for` loop, but the loop bodies are very similar to each other and both functions accomplish the same task. In this case, these two code fragments should be identified as clones so a human can inspect them and determine the best course of action, whether to eliminate one of the clones or to leave it as is.

Clones are beneficial to detect because having duplicates increases maintenance efforts. When a change is made to one instance of the clone, it should also be evaluated and performed to all the other instances. If the change is not propagated to all the clones, then bug fixes will also not be propagated, creating inconsistencies. In addition, having clones increases the size of the code base, which opens more opportunities for bugs and also increases the size of the executable. If clones can be abstracted into one function, then the duplicates can be eliminated, resulting in a smaller code base and fewer bugs [9]. If clones cannot be removed, then simply identifying and documenting them will help.

*Normalizer* is a source code mutator that can aid in finding code clones. The way code is written, which is dependent on programming style, can play a role in the detection of code clones. Two code fragments may be written in completely different styles, but still be considered a clone if they perform the same task. To eliminate such differences, *Normalizer* forces the source code to look as similar as possible while preserving the semantics. The transformed source code is to be fed into code clone detection tools with the intent of catching larger and more numerous code clones.

Given the plethora of existing code clone detectors, this tool is not meant to compete in finding clones, but to augment the detectors' performance. The performance of any code clone detector can be improved by normalizing the source file, filtering out uninteresting code sections leaving only minimal differences between clones [15]. How the code is written can make a difference in the effectiveness of the code clone detectors. Due to personal coding styles, a programmer's choice of statement order and identifier names can be inconsistent. These small changes can fool some detectors, but if the input can be massaged so that regardless of how the original source code was written, the output would have a standardized style, thus more consistent findings can be achieved.

*Normalizer* operates on the C language and uses a variety of techniques to normalize the source code. Useless code removal and statement reordering make up the bulk of the tool. Identifier renaming, converting for-loops into while-loops, and splitting source code into multiple files are additional techniques used to achieve a higher clone detection rate, all of which are discussed in greater detail in Chapter 4.

Some code clones detectors are used to evaluate the effectiveness of *Normalizer*. Using the code clone detection tools Simian, JPlag, CloneDR, and Moss, *Normalizer* increases the clone detection rate but occasionally performs worse than without normalization. By reordering and removing code, larger code clones are exposed with fewer interrupted inserted lines. However, reordering code also moves sections of code clones apart and renaming identifiers can result in fewer and smaller clones. Background information surrounding the tools can be found in Chapter 3.

This paper discusses the design, implementation, and performance of *Normalizer*. In addition, a comparison of the code clone detection tools is presented detailing their strengths and weaknesses while using *Normalizer*. The goal of this thesis is to demonstrate that the simple task of formatting the input can change the detector's results and possibly improve it by making the code appear as similar as possible.

# Chapter 2

# BACKGROUND

Code clones are duplicated code fragments that perform the same or similar task. They tend to have lexical similarity between them, but may be slightly altered to fit their own application. These clones can be abstracted into a separate function or at least be identified as a code clone. This chapter explains why identifying code clones is important and the measures already taken to expose them.

## 2.1 Code Clones

Code clones can be created in several ways, commonly by copy-and-pasting code sections, by duplication of effort, or by intentional choice in performance critical systems [13]. Copy-and-pasting code sections from one file to another can occur when a programmer does not have the means to call the function from their own code. The easy solution is to copy the code over into their own file and use it. The copied code may also be slightly altered to fit the programmer's application, creating a near-identical clone. Duplication of effort can take different forms, but does not have to involve any irresponsible intent. Sometimes, two independent developers may coincidentally write a commonly useful function without knowing one already existed. Code can also be duplicated in performance critical systems to avoid the overhead of function calls [1]. The mere identification of code clones is enough for the programmer to inspect and decide whether to abstract the clones into a function or leave them be.

### 2.1.1 Types of Code Clones

There are varying levels of code clones, from literal copy-and-pastes to alterations that achieve the same result but are implemented differently. Roy, who authored a paper comparing code clone techniques, defined four types of clones [2]:

1. **Type-1**: These are the easiest clones to catch, which involve varying whitespace and comments as these do not affect the program at all.

2. **Type-2**: These clones are Type-1 clones but also have changed identifier names, literal values, and types.

3. **Type-3**: These are everything Type-2 includes but also has inserted or removed statements. The code's functionality may be slightly different, but it is still considered a clone.

4. **Type-4**: Lastly, Type-4 clones have the same semantics as each other, but have different implementation details to achieve that [16] [18].

Code clone detection tools have differing levels of effectiveness on code clone types depending on the methods used to find clones. Type-4 clones are not always possible to detect; when possible, doing so requires complex techniques and large amounts of computer resources.

## 2.2 Code Clone Detection Tools

Tools already exist to detect code clones. They analyze the source files and commonly report line numbers, file names, and the sizes and similarity score of the code clones. The standard reporting medium is through HTML, although textual output is also used. These tools are marketed towards industry codebases in order to analyze the unwieldy, large codebases that would take prohibitively long for humans to analyze manually.

### 2.2.1 Textual and Token-Based Clone Tools

Each tool uses varying levels of complexity to identify clones. The most basic tools compare source files on a character-by-character basis [16]. This approach is easily defeated by spurious whitespace and comments. A step-up is a token-by-token basis. The tool is able to discern between keyword and identifier, allowing for some flexibility in identifier names. If the texts have a close enough match, then a clone is detected.

Such a tool that uses token-based clone detection is Simian. Given a programming language, Simian can compare source code. It can be run with options to ignore certain tokens such as identifiers, variable names, or literals. Compared with all the tools used in this paper's validation, this is considered the most basic [6]. A more detailed overview of Simian is provided in Section 3.1.1.

Another tool, called JPlag, compares token strings to assess whether a clone exists or not. Once parsed, tokens from one clone are matched with tokens from the other clone as greedily as possible. A feature of JPlag is that the tokens may appear in any order, which lepts to detect clones that reorder statements [12]. Section 3.1.2 covers JPlag in more detail.

### 2.2.2 Tree-Based Clone Tools

More complex tools can parse the source code and obtain a better understanding of the code when provided a programming language grammar. The tool constructs a representation of the program called an abstract syntax tree (AST), whose nodes represent different coding constructs such as statements and functions [16]. The tool then compares the subtrees from the AST with other subtrees and if two subtrees match, then a clone has been identified. Finding the largest matching subtree is a resource intensive task but generally gives better results than a token-based tool. Compared with a token-based tool, this technique uses information on the syntactic structure of the code to help base its judgement. An example of an AST for an `if` statement in Listing 2.1 is shown in Figure 2.1.

**Listing 2.1: A small `if` statement**

```
if (strlen(str) < size) {
    return size;
}
```

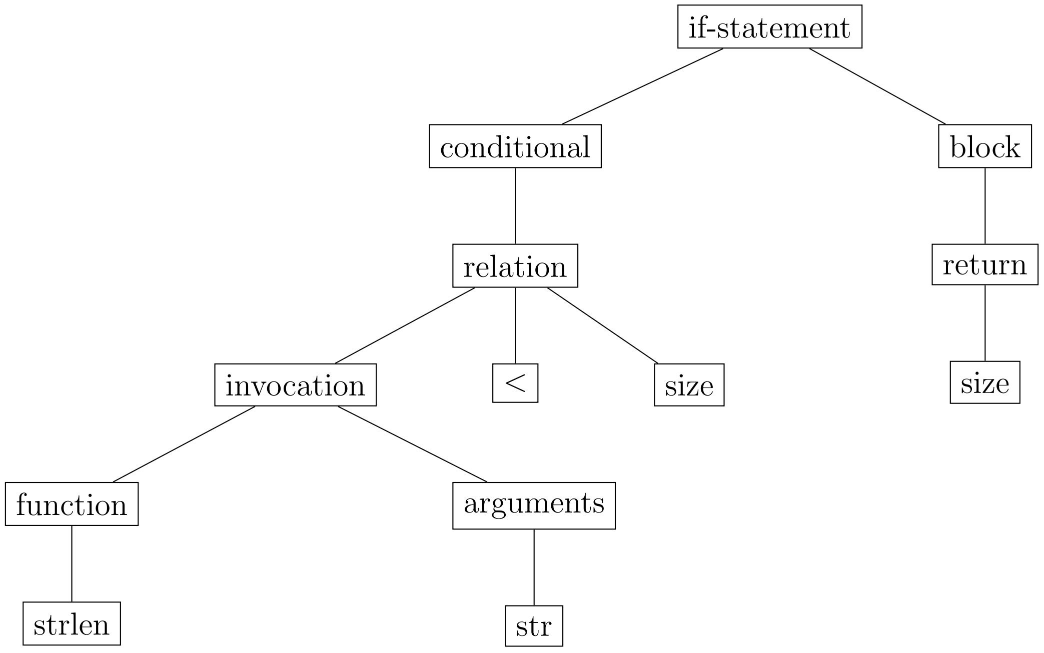


**Figure 2.1: AST for Listing 2.1**

A code clone detector called CloneDR uses this approach. Trees are compared with one another by comparing their hashed values. If the hashed values collide or are similar within a threshold, then the trees can be considered a clone [1]. More information on CloneDR is given in Section 3.1.3.

### 2.2.3 Program Dependence Graph Clone Tools

Once an internal representation of the program is created as an AST, the program can be further transformed into a program dependence graph (PDG) [16]. A PDG is a representation of the program where statements are nodes and dependencies between statements are represented as edges. If one statement depends on the result of another statement, then a directed edge is drawn between those nodes. Regardless of how the two similar code sequences are ordered, the program dependence graph remains the same. Tools can compare the graphs and identify isomorphic subgraphs which would suggest a possible code clone [1]. A sample of a PDG derived from Listing 2.2 is shown in Figure 2.2.

**Listing 2.2: `strlen` code sample for PDG**

```
int strlen(const char* str) {
    int len;
    len = 0;
    while (*str != '\0') {
        str++;
        len++;
    }
    return len;
}
```

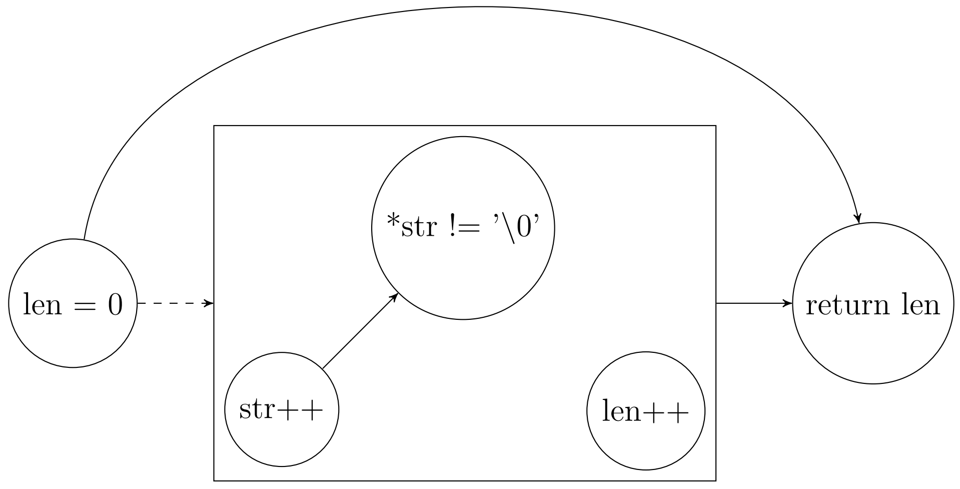


**Figure 2.2: Sample PDG for Listing 2.2**

This type of analysis is more robust than a tree-based clone tool against statement reordering and code insertion, however, it is more compute intensive. PDGs are used in *Normalizer* as a way to reorder statements and to remove useless code. Constructing PDGs and their use in *Normalizer* is explained in Section 4.3.

Among all the different algorithms to detect clones, some may perform better than others depending on various factors. Those factors include but are not limited to: programming language used, size of source code, and the nature of the code clones [4]. *Normalizer* aims to be a general booster to all code clone detectors, and each tool may witness varying degrees of effectiveness from source code normalization.

## 2.3 Code Normalizing

*Normalizer* is not given the responsibility of detecting clones, but instead its goal is to mutate the source code into a format that is more likely to expose code clones. Achieving normalized code will involve using similar techniques to the advanced tools, but instead of exposing clones, *Normalizer* will output source code with renamed identifiers and reordered statements with the intent that any code clone detector will find more clones than without normalization.

# Chapter 3

# RELATED WORKS

Detecting code clones is a challenge and many works have been produced to explore this problem. This chapter discusses some of the works focusing on code clone detectors and techniques involving code normalization.

## 3.1 Code Clone Detection Tools

Code clone detection tools produce a report detailing any caught clones that the tool has found. These tools use techniques to find clones that are related to how *Normalizer* normalizes source code. The main distinction between the two is that *Normalizer* outputs source code while a code clone detector outputs a clone report. Despite the difference, these two ideas are meant to benefit the cause of catching code clones. Some of the tools are explained below.

### 3.1.1 Simian

Simian is a code clone detector that operates on tokens. Simian splits the text into tokens and can selectively ignore or consider certain types of tokens. It can operate on many languages such as Java, C, Ruby, JavaScript, or just plain text. By specifying a programming language, Simian is better able to tokenize the input instead of using a whitespace delimiter. Some options are available to certain languages, such as the ability to ignore identifier names, literal values, and modifiers such as `static`. Since Simian is a token-based clone detector, it will not be able to understand program structure, but it serves as a simple and fast detector [6].

Simian outputs clones via the terminal. A sample output is given in Listing 3.1.

Listing 3.1: Sample Report from Simian

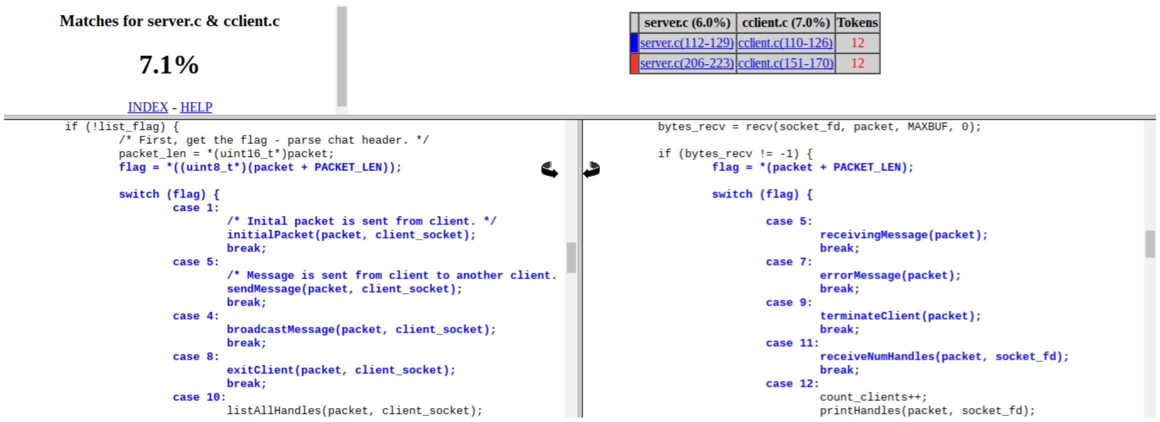
```
Similarity Analyser 2.4.0 - http://www.harukizaemon.com/simian
Copyright (c) 2003-2015 Simon Harris.  All rights reserved.
Simian is not free unless used solely for non-commercial or evaluation purposes.
{failOnDuplication=true, ignoreCharacterCase=true, ignoreCurlyBraces=true,
    ignoreIdentifierCase=true, ignoreModifiers=true, ignoreStringCase=true,
    reportDuplicateText=true, threshold=6}
Found 6 duplicate lines in the following files:
 Between lines 115 and 121 in /home/ooee/Thesis/cclient.c
 Between lines 103 and 109 in /home/ooee/Thesis2/cclient.c
    packet = makePacketMssg(CLIENT_MESSAGE, destLen, dest, srcLen, myHandle, toSend);
    if(sendPacket(packet, socket) < 0) {
      perror("Packet Message");
      exit(1);
    }
    free(packet);
=====================================================================
Found 8 duplicate lines in the following files:
 Between lines 92 and 102 in /home/ooee/Thesis2/cclient.c
 Between lines 53 and 63 in /home/ooee/Thesis2/cclient.c
  int mssgNum = (strlen(mssg) + 1) / maxMssgLen;  int consumed = 0;
  /* add mssgNum because that's how many nulls we'll end up with
     which we need to account for in the packet*/
  int theRest = ((strlen(mssg) + 1) % maxMssgLen) + mssgNum;

  toSend = malloc(maxMssgLen);
  while(mssgNum > 0) {
    memcpy(toSend, mssg + consumed, maxMssgLen - 1);
    toSend[maxMssgLen - 1] = 0;
    consumed += maxMssgLen - 1;
=====================================================================
Found 28 duplicate lines in 6 blocks in 1 files
Processed a total of 543 significant (926 raw) lines in 5 files
Processing time: 0.054sec
```

The metric Simian uses to quantify code clone sizes is line numbers. The default minimum threshold for Simian code clones is six lines of duplicated code. Other defaults are to ignore capitalization cases and not to ignore identifier names.

### 3.1.2 JPlag

JPlag is another code clone detection tool that also uses tokens. JPlag converts the text into token strings, whose intent is to characterize the essentials of a program. By capturing only the important tokens, those tokens would be likely to be preserved if the code was cloned and modified. Some examples of important tokens are `IF`, `ASSIGN`, and `FUN`, for function [11]. Identifiers and common operators such as the arithmetic operators or relational operators are not included as tokens because they are not considered as important or essential to the program. The token strings produced from each clone candidate can be compared against each other disregarding order. If the two token strings match closely enough, then a clone is encountered [12]. The numeric metric used to measure clone size is by tokens matched.

JPlag outputs results as an HTML page. A sample JPlag output is given in Figure 3.1.



**Matches for server.c & cclient.c**

**7.1%**

INDEX - HELP

| | server.c (6.0%) | cclient.c (7.0%) | Tokens |
|---|---|---|---|
| 🟦 | server.c(112-129) | cclient.c(110-126) | 12 |
| 🟥 | server.c(206-223) | cclient.c(151-170) | 12 |

```
if (!list_flag) {
        /* First, get the flag - parse chat header. */
        packet_len = *(uint16_t*)packet;
        flag = *((uint8_t*)(packet + PACKET_LEN));

        switch (flag) {
                case 1:
                        /* Inital packet is sent from client. */
                        initialPacket(packet, client_socket);
                        break;
                case 5:
                        /* Message is sent from client to another client.
                        sendMessage(packet, client_socket);
                        break;
                case 4:
                        broadcastMessage(packet, client_socket);
                        break;
                case 8:
                        exitClient(packet, client_socket);
                        break;
                case 10:
                        listAllHandles(packet, client_socket);
```

```
bytes_recv = recv(socket_fd, packet, MAXBUF, 0);

if (bytes_recv != -1) {
        flag = *(packet + PACKET_LEN);

        switch (flag) {

                case 5:
                        receivingMessage(packet);
                        break;
                case 7:
                        errorMessage(packet);
                        break;
                case 9:
                        terminateClient(packet);
                        break;
                case 11:
                        receiveNumHandles(packet, socket_fd);
                        break;
                case 12:
                        count_clients++;
                        printHandles(packet, socket_fd);
```

**Figure 3.1: Clones found in JPlag**

### 3.1.3 CloneDR

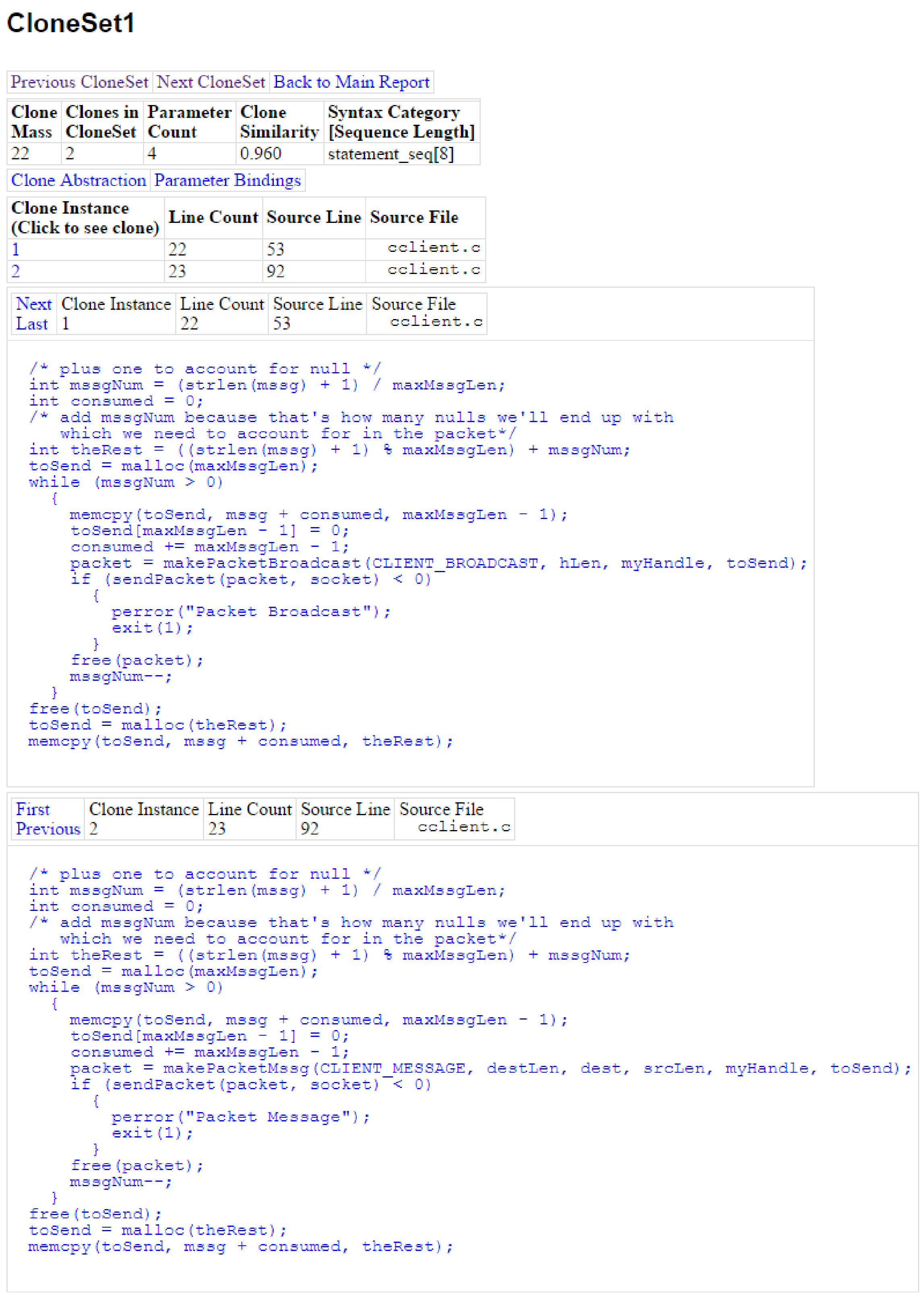
CloneDR is a clone detection tool that converts a program into an AST then compares the subtrees with each other. It uses a hashing function that hashes trees and its children into a value. The authors decided upon using a poor hashing function so that similar subtrees would hash into similar values. The hashed values are then binned so they can be compared with others in the same bin which lends to its computational performance. If the hashes are similar enough, then a clone has been detected [1].

Figure 3.2 is a screenshot of CloneDR's reporting.

### 3.1.4 Moss

Moss is a plagiarism detector web service where instructors submit student code and reports of copied code are returned. Moss uses fingerprinting to obtain a unique signature of a section of code. To obtain a fingerprint, Moss partitions the code into sections by using a sliding window, then each section is hashed into a value. The algorithm then picks every $n$ hashes as a component of the code's signature, where $n$ is a constant. A database of fingerprints is generated and compared with one another, and if any two fingerprints match closely enough, then a clone has been detected [17]. The numeric metric used to measure clone size is by lines matched.

Moss produces a report in HTML given by a unique URL that lasts ten days. A list of clones is presented with each clone listed side-by-side. A sample Moss output is given in Figure 3.3.



## CloneSet1

| Previous CloneSet | Next CloneSet | Back to Main Report |
|---|---|---|

| Clone Mass | Clones in CloneSet | Parameter Count | Clone Similarity | Syntax Category [Sequence Length] |
|---|---|---|---|---|
| 22 | 2 | 4 | 0.960 | statement_seq[8] |

| Clone Abstraction | Parameter Bindings |
|---|---|

| Clone Instance (Click to see clone) | Line Count | Source Line | Source File |
|---|---|---|---|
| 1 | 22 | 53 | cclient.c |
| 2 | 23 | 92 | cclient.c |

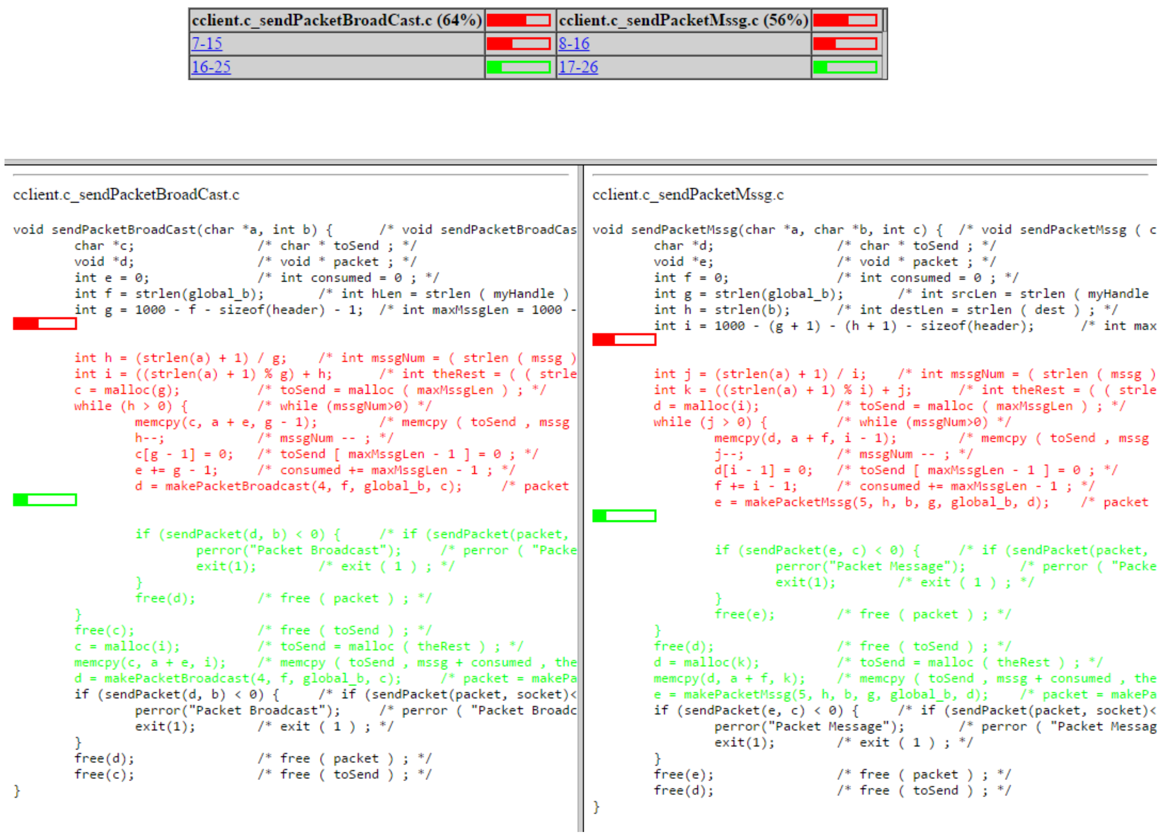
| Next Last | Clone Instance | Line Count | Source Line | Source File |
|---|---|---|---|---|
| | 1 | 22 | 53 | cclient.c |

```
/* plus one to account for null */
int mssgNum = (strlen(mssg) + 1) / maxMssgLen;
int consumed = 0;
/* add mssgNum because that's how many nulls we'll end up with
   which we need to account for in the packet*/
int theRest = ((strlen(mssg) + 1) % maxMssgLen) + mssgNum;
toSend = malloc(maxMssgLen);
while (mssgNum > 0)
  {
    memcpy(toSend, mssg + consumed, maxMssgLen - 1);
    toSend[maxMssgLen - 1] = 0;
    consumed += maxMssgLen - 1;
    packet = makePacketBroadcast(CLIENT_BROADCAST, hLen, myHandle, toSend);
    if (sendPacket(packet, socket) < 0)
      {
        perror("Packet Broadcast");
        exit(1);
      }
    free(packet);
    mssgNum--;
  }
free(toSend);
toSend = malloc(theRest);
memcpy(toSend, mssg + consumed, theRest);
```

| First Previous | Clone Instance | Line Count | Source Line | Source File |
|---|---|---|---|---|
| | 2 | 23 | 92 | cclient.c |

```
/* plus one to account for null */
int mssgNum = (strlen(mssg) + 1) / maxMssgLen;
int consumed = 0;
/* add mssgNum because that's how many nulls we'll end up with
   which we need to account for in the packet*/
int theRest = ((strlen(mssg) + 1) % maxMssgLen) + mssgNum;
toSend = malloc(maxMssgLen);
while (mssgNum > 0)
  {
    memcpy(toSend, mssg + consumed, maxMssgLen - 1);
    toSend[maxMssgLen - 1] = 0;
    consumed += maxMssgLen - 1;
    packet = makePacketMssg(CLIENT_MESSAGE, destLen, dest, srcLen, myHandle, toSend);
    if (sendPacket(packet, socket) < 0)
      {
        perror("Packet Message");
        exit(1);
      }
    free(packet);
    mssgNum--;
  }
free(toSend);
toSend = malloc(theRest);
memcpy(toSend, mssg + consumed, theRest);
```

**Figure 3.2: Clones found in CloneDR**



| cclient.c_sendPacketBroadCast.c (64%) | 🟥 | cclient.c_sendPacketMssg.c (56%) | 🟥 |
|---|---|---|---|
| 7-15 | 🟥 | 8-16 | 🟥 |
| 16-25 | 🟩 | 17-26 | 🟩 |

cclient.c_sendPacketBroadCast.c

```
void sendPacketBroadCast(char *a, int b) {      /* void sendPacketBroadCas
        char *c;                        /* char * toSend ; */
        void *d;                        /* void * packet ; */
        int e = 0;                      /* int consumed = 0 ; */
        int f = strlen(global_b);               /* int hLen = strlen ( myHandle )
        int g = 1000 - f - sizeof(header) - 1;  /* int maxMssgLen = 1000 -

        int h = (strlen(a) + 1) / g;    /* int mssgNum = ( strlen ( mssg )
        int i = ((strlen(a) + 1) % g) + h;      /* int theRest = ( ( strle
        c = malloc(g);                  /* toSend = malloc ( maxMssgLen ) ; */
        while (h > 0) {                 /* while (mssgNum>0) */
                memcpy(c, a + e, g - 1);        /* memcpy ( toSend , mssg
                h--;                    /* mssgNum -- ; */
                c[g - 1] = 0;   /* toSend [ maxMssgLen - 1 ] = 0 ; */
                e += g - 1;     /* consumed += maxMssgLen - 1 ; */
                d = makePacketBroadcast(4, f, global_b, c);     /* packet

                if (sendPacket(d, b) < 0) {     /* if (sendPacket(packet,
                        perror("Packet Broadcast");     /* perror ( "Packe
                        exit(1);        /* exit ( 1 ) ; */
                }
                free(d);                /* free ( packet ) ; */
        }
        free(c);                        /* free ( toSend ) ; */
        c = malloc(i);                  /* toSend = malloc ( theRest ) ; */
        memcpy(c, a + e, i);    /* memcpy ( toSend , mssg + consumed , the
        d = makePacketBroadcast(4, f, global_b, c);     /* packet = makePa
        if (sendPacket(d, b) < 0) {     /* if (sendPacket(packet, socket)<
                perror("Packet Broadcast");     /* perror ( "Packet Broadc
                exit(1);                /* exit ( 1 ) ; */
        }
        free(d);                        /* free ( packet ) ; */
        free(c);                        /* free ( toSend ) ; */
}
```

cclient.c_sendPacketMssg.c

```
void sendPacketMssg(char *a, char *b, int c) {  /* void sendPacketMssg ( c
        char *d;                        /* char * toSend ; */
        void *e;                        /* void * packet ; */
        int f = 0;                      /* int consumed = 0 ; */
        int g = strlen(global_b);               /* int srcLen = strlen ( myHandle
        int h = strlen(b);              /* int destLen = strlen ( dest ) ; */
        int i = 1000 - (g + 1) - (h + 1) - sizeof(header);      /* int max

        int j = (strlen(a) + 1) / i;    /* int mssgNum = ( strlen ( mssg )
        int k = ((strlen(a) + 1) % i) + j;      /* int theRest = ( ( strle
        d = malloc(i);                  /* toSend = malloc ( maxMssgLen ) ; */
        while (j > 0) {                 /* while (mssgNum>0) */
                memcpy(d, a + f, i - 1);        /* memcpy ( toSend , mssg
                j--;                    /* mssgNum -- ; */
                d[i - 1] = 0;   /* toSend [ maxMssgLen - 1 ] = 0 ; */
                f += i - 1;     /* consumed += maxMssgLen - 1 ; */
                e = makePacketMssg(5, h, b, g, global_b, d);    /* packet

                if (sendPacket(e, c) < 0) {     /* if (sendPacket(packet,
                        perror("Packet Message");       /* perror ( "Packe
                        exit(1);        /* exit ( 1 ) ; */
                }
                free(e);                /* free ( packet ) ; */
        }
        free(d);                        /* free ( toSend ) ; */
        d = malloc(k);                  /* toSend = malloc ( theRest ) ; */
        memcpy(d, a + f, k);    /* memcpy ( toSend , mssg + consumed , the
        e = makePacketMssg(5, h, b, g, global_b, d);    /* packet = makePa
        if (sendPacket(e, c) < 0) {     /* if (sendPacket(packet, socket)<
                perror("Packet Message");       /* perror ( "Packet Messag
                exit(1);                /* exit ( 1 ) ; */
        }
        free(e);                        /* free ( packet ) ; */
        free(d);                        /* free ( toSend ) ; */
}
```

**Figure 3.3: Clones found in Moss**

### 3.1.5 Summary

This work does not compare these clone detectors against each other, but rather uses each as the detection engine for the evaluation of the normalization techniques described in Section 4. As a summary of all the tools used in this thesis, Table 3.1 compares the clone matching technique and the reporting metric of each tool.

**Table 3.1: Tool Comparison**

| Tool Name | Matching Technique | Size Metric |
|---|---|---|
| Simian | Textual tokens | Lines |
| JPlag | Important tokens such as keywords | Tokens |
| CloneDR | AST and subtree matching | Lines |
| Moss | Fingerprinting from tokens | Lines |

## 3.2 Code Normalization for Fighting Self-Mutating Malware

Finding similarities by normalizing code is not limited to the software engineering field, but also can be used against malware. Self-mutating malware is a problem since their executable signature is always changing in order to avoid anti-virus detection. By perturbing the binary within the malware just slightly without modifying the functionality, the byte-per-byte comparison between it and an established signature will not yield a match. The techniques that malware uses to self-mutate include instruction substitution, instruction reordering, dead-code insertion, variable substitutions, and control flow alterations. By normalizing the machine code, it can undo the process of the self-mutation, leaving only the core functionality of the malware behind. The core functionality would be the same across different mutations of the malware, which can be compared with an existing anti-virus signature and be detected as malware [3].

The mutations that the malware takes are similar to the clones that programmers may write. Programmers may tweak the source code of a clone to better fit their own application or write a function slightly differently from one that already exists out of forgetfulness. These clones may have reordered statements, extra variables, and even extra code. Just as normalizing source code may expose similarities to other clones, normalizing malware binary may expose similarities to the malware signature.

## 3.3 NICAD

Accurate Detection of Near-miss Intentional Code (NICAD) is a tool that uses pretty-printing and pattern matching to detect code clones. It parses the file to eliminate any noise caused by spacing or comments and formats the source code into a specific way, such as placement of braces. This would allow clone detection by line comparison easier. Pattern matching targets a statement of a certain selected type. Then it compares against others of the same statement type, matching whether it contains a similar body or expression type. An `if` statement for example, could match for any conditional, but will match the entirety of the body. If any other `if` statement exhibits the same body, then those two code sections are good candidates for a clone. The algorithm chosen to detect clones is by longest common substring, where each token is considered to be an element in the string [15].

Similar to *Normalizer*, NICAD uses pretty-printing to format the code in a consistent way. Both *Normalizer* and NICAD removes comments and applies uniform spacing to the entire code, removing noise which would distract from the source code. By cleaning up the source code, both NICAD and *Normalizer* aim to detect more code clones. An added benefit over NICAD is that *Normalizer* also reorderes statements and removes useless code.

# Chapter 4

# IMPLEMENTATION

This section goes over the design of *Normalizer* and the techniques used to achieve the source code normalization. First, the requirements of *Normalizer* are stated.

## 4.1 Requirements

In order for *Normalizer* to best produce results, it must output C code that:

1. is able to be parsed by the code clone detection tool
2. should perform as similar to the original code as much as possible but be as written as consistently as much as possible. Some alterations are:
   (a) Statement reordering
   (b) Identifier renaming
   (c) For-loop conversions
   (d) Useless code removal
3. can be tracked back to original code

To accomplish this set of tasks, the following steps are taken to process and output normalized code.

## 4.2 Input

The input is a C source code file which *Normalizer* will load into memory. Some steps are taken to prepare the source code.

### 4.2.1 Preprocessing

Before reading the source file, a typical C source file contains preprocessor directives and comments. Comments are removed and preprocessor directives are resolved by the GCC compiler using the `-E` argument. The result is all `#include` directives are included directly into the file, all `#define` directives expanded and substituted, and all comments are replaced with white space. The GCC preprocessor leaves `#include` markers identifying the sections of included code in its place. These markers which start with a `#` can safely be removed and thus leaves source code readily acceptable by a parser.

### 4.2.2 ANTLR

ANTLR is a parser generator that can generate code to parse a language and construct an abstract syntax tree (AST) [10]. Using the ANTLR provided C grammar, a parser is generated in the Java language that can accept C source code. The preprocessed code can be lexed and parsed by the generated parser to create the internal representation of the program as an AST. During the transformation from token stream to AST, some unimportant information is lost. Modifiers such as `const`, `static`, and `volatile` are not recorded in the AST. The reason being that those keywords are relatively less interesting than other tokens.

The parser however does not have a symbol table, making recording typedef names and function names infeasible during the parsing process. One outcome of this inability is the ambiguity of variable declarations and function invocations. Variables in C can be declared by *type*`(`*identifier*`)`. The code `int(a)` is a legal variable declaration for the `int` called `a`. The code for `run(a)` is ambiguous because it could be a function invocation for the function `run` or a variable declaration for the typedef name of `run`. Since it is uncommon to use the parenthetical style to declare variables, that feature is removed from the grammar and `run(a)` can only be interpreted as a function invocation. Other grammar modifications were made to accept complicated code included by C standard library header files.

### 4.2.3 Trackability

In order to report results to the user, outputted source code must have a crumb trail so that through any transformation performed on the code, it can be traced back to the original source. Each statement from the AST contains a string of tokens from the original source. The tokens are concatenated into a string and preserved throughout the program run. A sample output of Normalizer showing the original code is shown in 4.1.

**Listing 4.1: Comments added as a breadcrumb to original code**

```
int square(int *a) { /* int square(int *number) */
    if (a == ((void*) 0)) { /* if (number == NULL) { */
        return 0; /* return 0 */
    } else { /* } else { */
        return * a * * a; /* return * number * * number */
    }
}
```

## 4.3 PDG

After the program has been loaded as an AST, a further transformation of the internal representation into a program dependence graph (PDG) can be performed. A PDG is a way of representing a program where each vertex in the graph represents a statement in the program. Directed edges are placed to represent dependencies. If two vertexes are connected, then one statement is dependent on the result of the other. The PDG is used for two features, *statement reordering* and *useless code removal.*

### 4.3.1 Constructing the PDG

From the AST, a PDG is constructed. Each statement is represented as a node. Statements can form relationships with other nodes, either *control dependent* or *data dependent*.

Data Dependencies - A statement is data dependent if it depends on the data of a variable from another statement. Take Listing 4.2 for example.

**Listing 4.2: Data dependency example**

```
int a, b;
a = 5 + 2;
b = a * 2;
```

The statement at `b = a * 2` is dependent on `a = 5 + 2` because the variable `a` is modified on line 2 and is being read on line 3. Therefore, the PDG for Listing 4.2 is shown in Figure 4.1. Data dependencies are marked as a solid arrow in the PDG.

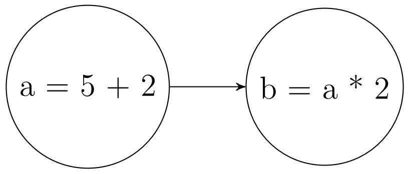


**Figure 4.1: PDG for Listing 4.2**

There are two variations on how statements can modify a variable.

1. Guaranteed Modified

   A guaranteed modification is a statement that with 100% certainty will assign a value into a variable by the end of the statement. For example, `a = 5 + 2`, the variable `a` will be modified. Guaranteed modifications occur when assignments are not in any conditional or are under both branches of a conditional.

2. Potentially Modified

A potential modification is a statement that is not guaranteed to assign a value into a variable. This is best observed in an if-statement as in Listing 4.3.

**Listing 4.3: Variable `a` has the potential of being set to 0**

```
1 int a, b, c;
2 a = 6;
3 if (c < 5) {
4     a = 0;
5 }
6 b = a;
```

Because `a` can either take the value at line 2 or line 4, the read on line 6 depends on both statements `a = 6` and `a = 0`. Since `a = 0` is inside a conditional, the `if` statement can potentially modify `a`. Potential modifications compound on previous modifications until a guaranteed modification is encountered. The constructed PDG for Listing 4.3 is shown in Figure 4.2. Notice the two in-edges going into `b = a`.

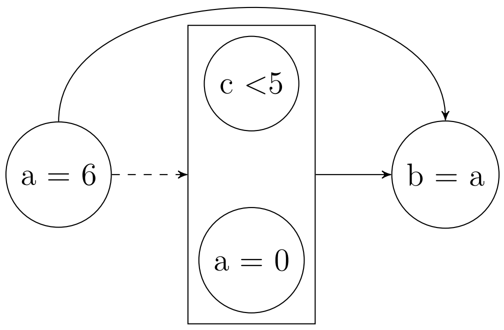


**Figure 4.2: PDG for Listing 4.3**

Control Dependencies - Control dependencies are classified when two statements cannot reverse their order. Of the two dependency types, control dependencies are a weaker form of relationship than data dependencies. Listing 4.4 has an example.

**Listing 4.4: Control dependency example**

```
int a, b;
```

```
a = 5;
a = 7;
b = a;
```

Here, `b = a` is dependent on the data from `a = 7`. Also implicitly, `a = 5` must happen before `a = 7`. If `a = 5` came immediately after `a = 7`, then `b` will be assigned a different value. Therefore there must be a control dependency between both assignments. The constructed PDG is shown in Figure 4.3. Control dependencies are marked as a dashed arrow. Control dependencies are placed when two statements contain assignments to the same variable, also called a “write after write.”

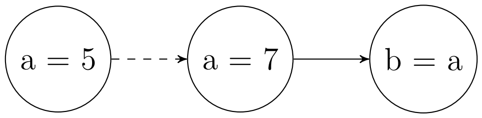


**Figure 4.3: PDG of Listing 4.4**

Using both variable dependencies and control dependencies, a PDG can be generated to represent the program.

### 4.3.2 Pointer Analysis

To obtain a more accurate PDG, simple pointer analysis is performed in order to better detect changed variables. If a pointer is dereferenced and modified, then any variables declared before it could potentially be modified by that statement. In addition, if a pointer type has been dereferenced and read, the dereferencing depends on any variable prior to that statement.

Take Listing 4.5 and its PDG in Figure 4.4 as an example of a read from a pointer.

**Listing 4.5: Variable `p` is dereferenced and read**

```
int func() {
    int a, b, *p;
    a = 1;
    b = 2;
    p = &a;
    b = *p;
    return b;
}
```

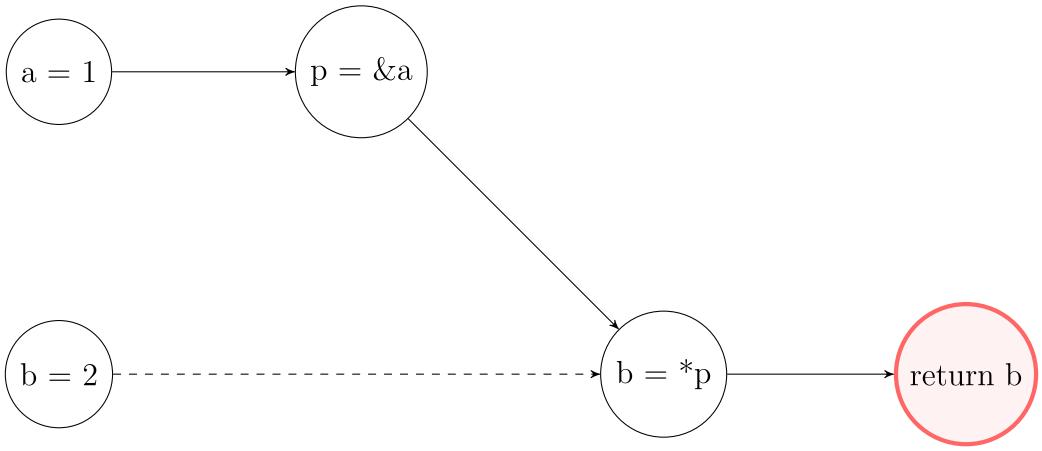


**Figure 4.4: PDG for Listing 4.5**

The statement of `b = *p` tells that `p` was dereferenced and its referenced value was assigned into b. Since the assignment of `p = &a` also depends on `a`, then the chain of dependencies is preserved.

Listing 4.6 and Figure 4.5 showcase a scenario where a value is written into a dereferenced pointer.

**Listing 4.6: Variable `p` is dereferenced and written to**

```c
int func2() {
    int a, c, *p;
    a = 1;
    b = 2;
    p = &a;
    *p = 5;
    return b;
}
```

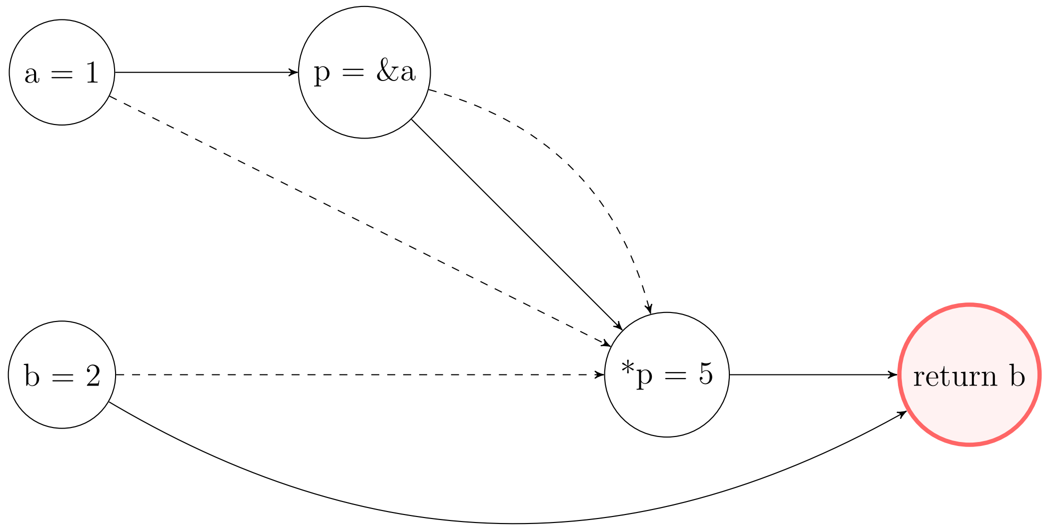


**Figure 4.5: PDG for Listing 4.6**

The dereference of `*p = 5` could potentially modify any of the variables, so all nodes have a control dependency on `*p = 5`. Also by dereferencing `p`, the value of `p` was read, so a variable dependency was drawn from `p = &a` to `*p = 5`.

### 4.3.3 Structs and Unions

Structs and unions provide another opportunity for dependencies to occur in the PDG.

Structs - In the C language, the members of a struct each occupy their own space in memory. The members of a struct can be treated as their own variables, where an assignment into one of the members does not affect their siblings. However, an assignment into a member will affect its children and the parent. An example is shown in Listing 4.7.

**Listing 4.7: Variable `p` is dereferenced and written to**

```c
typedef struct {
    int ssn;
    int age;
} Person;

typedef struct {
    int salary;
    Person person;
} Employee;

typedef struct {
    int averageScore;
    int yearsPlayed;
} BowlingStats;

typedef struct {
    Employee employee;
    BowlingStats bowlingStats;
} CompanyBowler;

int main() {
    CompanyBowler bowlerA = {{80000, {123456789, 23}}, {180, 2}};
    // A CompanyBowler who makes $80000 annually, has an ssn of 123456789, is 23 years old, and has an average score of 180 over 2 years of playing

    Employee newEmployee = {90000, {222222222, 26}};
    // An Employee who makes $90000 annually, has an ssn of 222222222 and is 26 years old

    bowlerA.employee = newEmployee;
    printf("%d", bowlerA.employee.salary);
    printf("%d", bowlerA.employee.person.ssn);
    printf("%d", bowlerA.employee.person.age);
    printf("%d", bowlerA.bowlingStats.averageScore);
    printf("%d", bowlerA.bowlingStats.yearsPlayed);
    return 0;
}
```

The assignment into `employee` causes all members under `employee` to be modified. In addition, the parent struct `CompanyBowler` is also modified since one of its members has been modified. The siblings of `employee` however are not modified, so `bowlingStats` is left untouched. The resulting PDG is shown in Figure 4.6.

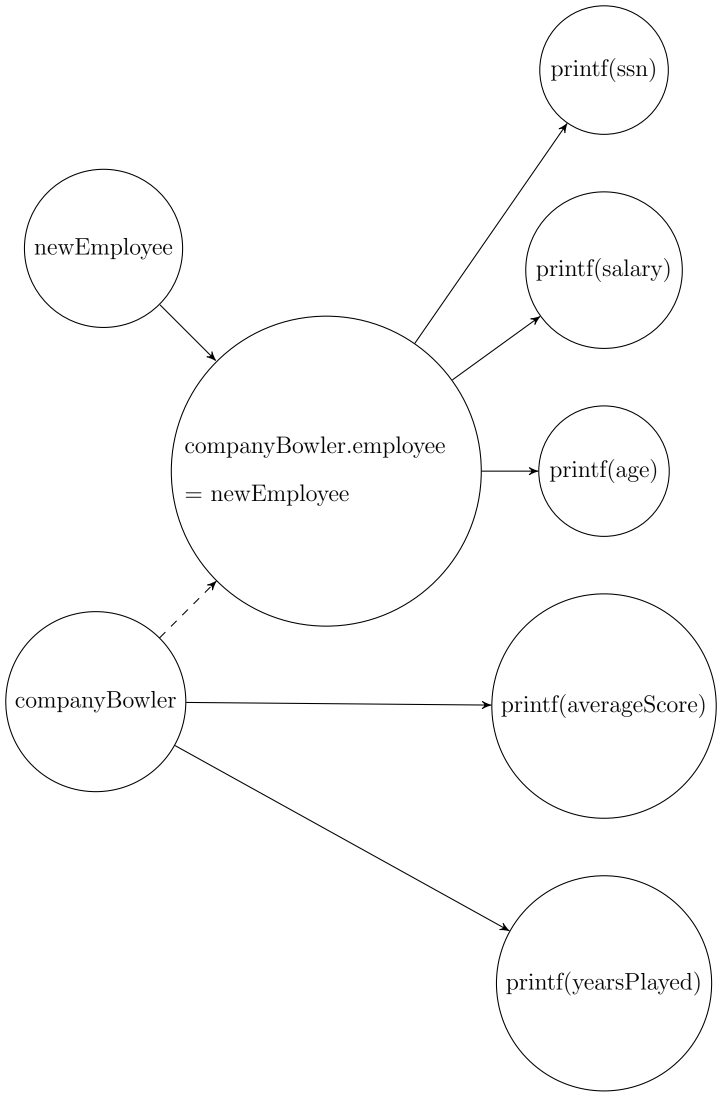


**Figure 4.6: PDG for Listing 4.7**

Unions - Unions are similar to structs except each member occupies the same memory space. If one member is modified, then all of its siblings could also be potentially modified. If a member of a union is modified, then their parent and children are also modified, the same way as structs. An example is given in Listing 4.8.

Listing 4.8: **Variable `p` is dereferenced and written to**

```c
typedef struct {
    int WPM;
    int floor;
} Programmer;

typedef struct {
    int injuries;
    int weight;
} Chef;

typedef union {
    Programmer programmer;
    Chef chef;
} Occupation;

typedef struct {
    int salary;
    Occupation occupation;
} Employee;

int main() {
    Employee employee;
    employee.salary = 80000;
    employee.occupation.programmer.WPM = 100;
    employee.occupation.programmer.floor = 2;

    employee.occupation.chef.injuries = 11;

    printf("%d", employee.occupation.programmer.WPM);
    printf("%d", employee.occupation.chef.injuries);
    printf("%d", employee.salary);
    return 0;
}
```

Although the assignment into the field `injuries` is inside a struct, The member `chef` is inside the union `Occupation` and has a sibling of `programmer`. So by modifying `injuries` which modifies `chef`, the members of `programmer` can also be potentially modified, namely `WPM` and `floor`. Therefore printing `WPM` depends on the original assignment into `WPM` and the potential modification into `injuries`. The PDG is shown in Figure 4.7.

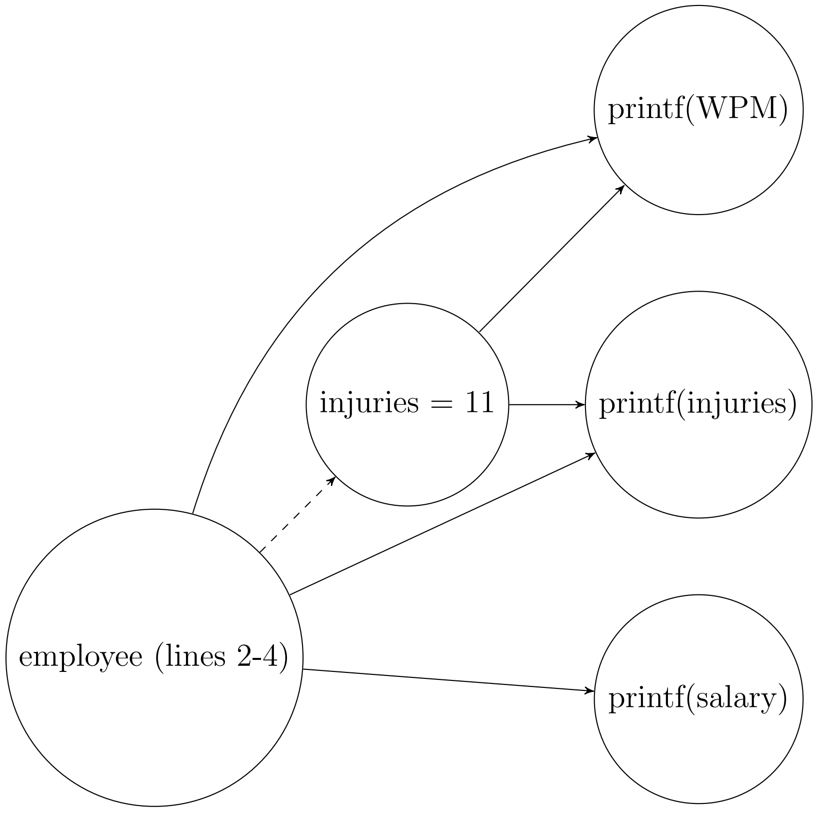


**Figure 4.7: PDG for Listing 4.8**

### 4.3.4 Side-effecting Functions

Some functions produce side-effects, for example C's `printf` outputs content to `stdout`. User-defined functions may also modify global variables or modify any input parameters passed into it. Due to this, analysis is performed on a per-function basis to give a profile on the capabilities of each function. If a function does modify global variables or modifies any input parameters, then *Normalizer* will treat each call to that function as one that also modifies those variables. If the function body is not provided, such as those in the C standard library, then it takes the conservative approach and assumes that it produces side-effects, potentially affecting any pointers passed in.

An example of a side-effecting function is shown in Listing 4.9 and its corresponding PDG in Figure 4.8.

**Listing 4.9: Variable `square` produces a side-effect to what was passed in**

```c
void square(int *a) {
    *a = *a * *a;
}

int main() {
    int a;
    a = 7;
    square(&a);
    return a;
}
```

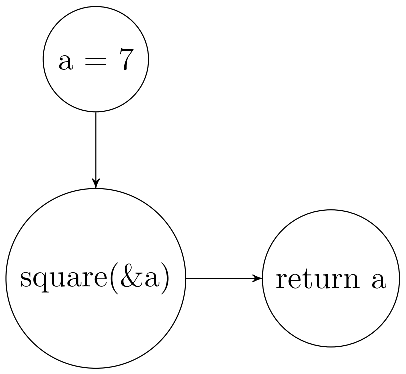


**Figure 4.8: PDG for Listing 4.9**

Since *Normalizer* identifies that `square` is guaranteed to modify whatever was passed in, there is a data dependency from `square(&a)` to `return a` and not one from `a = 7` to `return a`.

### 4.3.5 Nested Statements

C constructs such as for loops and while loops contain nested statements. PDGs are constructed for each opening brace `{` and closing brace `}`, which is called a code block. The code block collectively can be aggregated so that at a whole, the code block has variables it depends on, can potentially modify, and will modify. An example of nested statements is shown in Listing 4.10.

**Listing 4.10: `if` statements can read, write, and potentially write into variables**

```
on = 1;
num = 6;
name = get();
pts = 0;
if (on) {
    pts = num;
    name[0] += pts;
} else {
    name = "Jo";
}
printf("%d", on);
printf("%d", pts);
printf("%d", num);
printf(name);
```

Let us call the statements not inside any braces as the “first-level statements” and call each statement inside any brace as the “second-level statements” and so forth for every nested statement. The `if` which encapsulates the second-level statements is a first-level statement. The PDG for Listing 4.10 is shown in Figure 4.9.

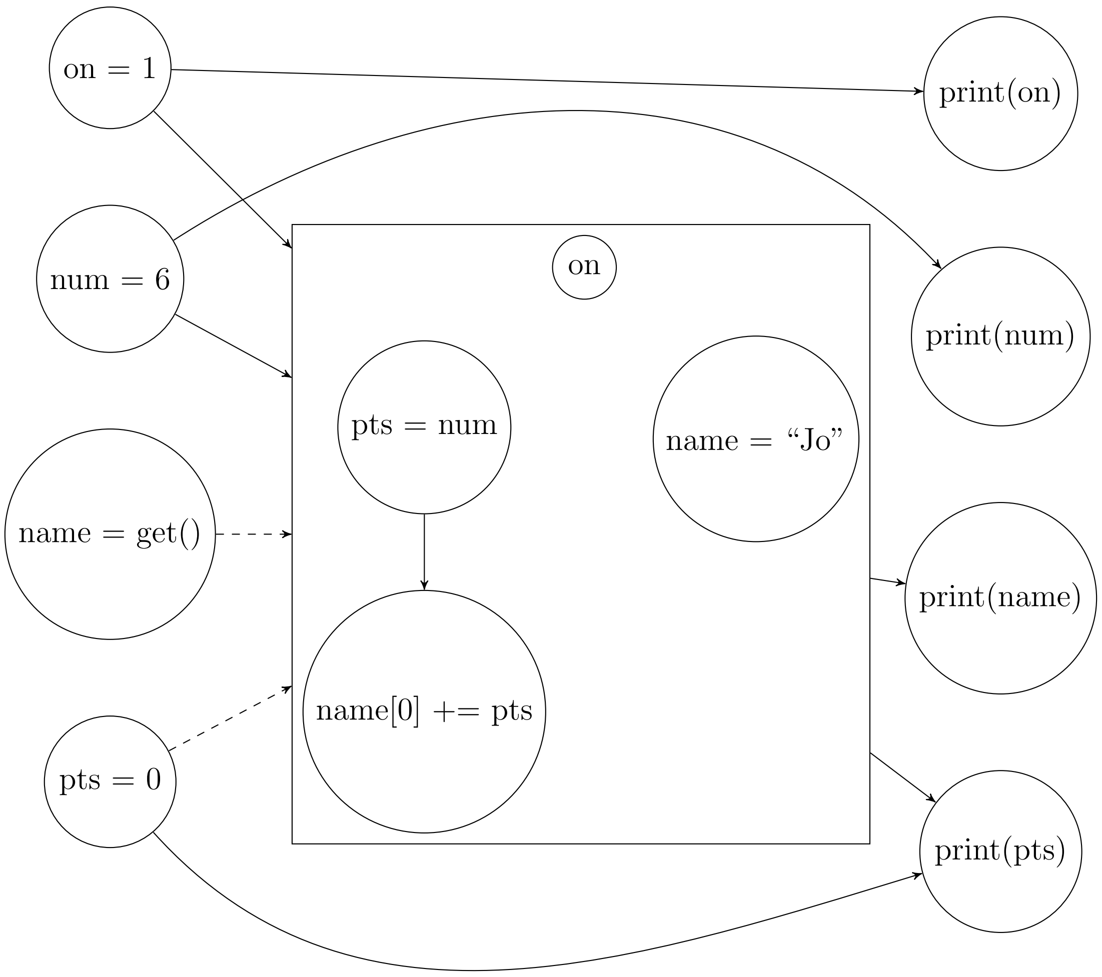


**Figure 4.9: PDG for Listing 4.10**

PDG nodes only have relationships with other PDG nodes on the same level. So `on = 1`, `num = 6`, `name = get()`, and `pts = 0` all have relationships with the `if` statement, but not any statement in second-level statements, even if those statements directly have those dependencies. Second-level statements can have relationships with each other localized within their own code block. Statements with nested statements take on the aggregate of their nested statements' dependencies. Therefore for the entirety of the `if` statement, it reads from the variables `on`, `num`, potentially modifies `pts`, and will modify `name`.

### 4.3.6 Transitive Reduction

After the PDG is constructed, there may be more edges than required. A transitive reduction of the PDG is performed to reduce the number of edges in the graph, while still retaining the same reachability. In addition, if a node has both a data dependence and control dependence to another node, the control dependence is removed because data dependence is a stronger relationship than control dependence and already implies order. Every graph has a unique transitive reduced graph [5]. By reducing the amount of edges, the computations become less numerous when sorting the PDG in the future. In addition, since the transitive reduction is unique, it provides an opportunity to compare the graphs between different functions. Figure 4.10 and Figure 4.11 show the result of a transitive reduction on a graph.

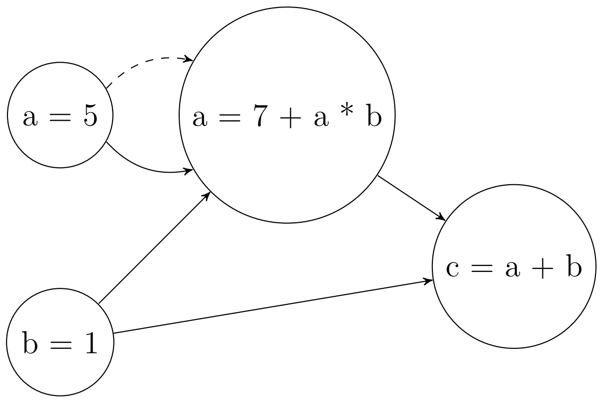


**Figure 4.10: Before transitive reduction**

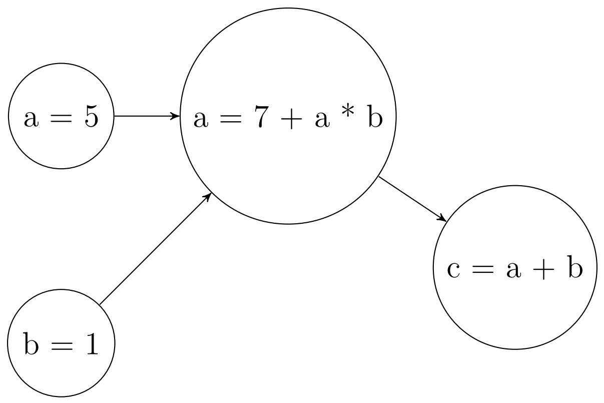


**Figure 4.11: After transitive reduction**

At this point, the PDG for the source has been created and is ready to be sorted under normalization.

## 4.4 Normalization

Source code normalization involves mutating the source code to appear as similar as possible while preserving the logic of the program. Clones that look alike to other clones are more likely to be detected as a clone. In this section, the techniques used by *Normalizer* are discussed.

### 4.4.1 Statement Reordering

Programs are written in a linear fashion, but the statements are not necessarily restricted to run in the order in which they are coded. Independent statements can be moved around if they do not depend on each other and the surrounding code does not depend on the reorder. By transforming the program into a PDG, the programmer's choice of ordering is taken out of consideration. The PDG can then be serialized back into source code using a topological sort. By applying the same topological sorting algorithm to each function, the order of code may appear as similar to each other and additional code clones can be detected.

A topological sort places nodes which have no in-edges first. Stated differently, it will prioritize statements which do not have any dependencies not yet placed. However, multiple nodes may have zero in-edges, creating ties. A good sorting algorithm for *Normalizer* is one that minimizes ties so that the code will appear in as consistent an order as possible. Consider Listing 4.11 and its PDG in Figure 4.12.

**Listing 4.11: Both `a = 8` and `b = 144 / 4` are independent of each other**

```
int a, b, c;
a = 8;
b = 144 / 4;
c = a + b;
```

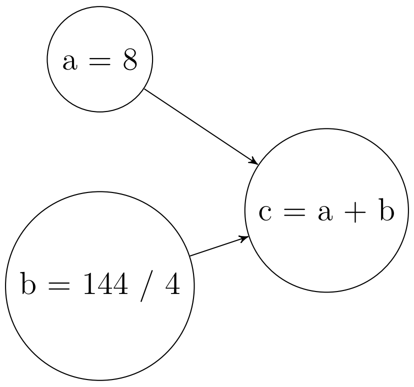


**Figure 4.12: PDG for Listing 4.11**

Both `a = 8` and `b = 144 / 4` are candidates for next in the topological sort since picking either of them to go first will not alter the value of `c`. This tie should be broken to force a strict ordering. *Normalizer* breaks ties by determining which statement is more "simple." This heuristic decision observes that `a = 8` is a simpler statement than `b = 144 / 4`, because no arithmetic operations take place in `a = 8`. So `a = 8` is picked before `b = 144 / 4`. Regardless of how the original code ordered those two statements, it will always output `a = 8` then `b = 144 / 4`. By enforcing a strict sorting using set rules, the source code ordering is consistent.

The example above gives an order on two `EXPRESSION_STATEMENT`s. To sort two different statement types, an order was predetermined based on their statement types. The ordering is listed below.

1. TYPEDEF_DECLARATION
2. ENUM_DEFINITION
3. STRUCT_DEFINITION
4. VARIABLE_DECLARATION
5. DECLARATION
6. EXPRESSION_STATEMENT
7. SELECTION_STATEMENT_IF
8. SELECTION_STATEMENT_SWITCH
9. ITERATION_STATEMENT_FOR
10. ITERATION_STATEMENT_DECLARE_FOR
11. ITERATION_STATEMENT_WHILE
12. ITERATION_STATEMENT_DO_WHILE
13. LABELED_IDENTIFIER_STATEMENT
14. LABELED_CASE_STATEMENT
15. LABELED_DEFAULT_STATEMENT
16. COMPOUND_STATEMENT
17. JUMP_BREAK_STATEMENT

18. JUMP_CONTINUE_STATEMENT

19. JUMP_RETURN_STATEMENT

If the PDG has two candidates, an `if` statement and an expression statement `p = 0`, the sorting algorithm will favor `p = 0` since `EXPRESSION_STATEMENT` is closer to the top than `SELECTION_STATEMENT_IF`.

If two statements are the same type, such as two `EXPRESSION_STATEMENT`s, `a = 55 / b` and `len = strlen("hello")`, a score is computed for each statement. The score is determined by the composition of the statement. Each expression statement at its heart is an assignment expression. Then the right hand value is a multiplicative expression `55 / b` and an invocation expression `strlen("hello")` respectively. To break this tie, the distance from those expression types to an `expression` as determined by the C grammar is used. The multiplicative expression is closer to an `expression` than the invocation expression, so `a = 55 / b` happens first.

There are some ties that cannot be broken, an example is `a = 0` and `b = 0`. Both are an assignment with the constant value of `0`. It would not make sense to break the tie by comparing the identifier names because identifier names are also up to the programmer. This pair of statements is marked as equal, and the sort picks one arbitrarily.

### 4.4.2 Useless Code Removal

Useless code removal is an optimization to remove code that has no effect on the program. Useless code is unintentionally left in the program typically by forgetting to remove debugging code or by including unnecessary statements. In the context of code clones, useless code is inserted code, which can be removed to potentially reveal a larger contiguous code clone instead of fragments of duplicated code.

Identifying useless code is possible by constructing the PDG. Useless code may form a tree which is not critical in the forest. Certain nodes in the PDG are marked as critical, such as `return` statements and function calls and global variable assignments. Then every node that the critical statements has a data dependency on is also marked as critical. The critical property propagates throughout the PDG until there are no more nodes to propagate to. All nodes not marked as critical are considered useless code and safe to remove from the program.

Take for an example Listing 4.12 and its PDG in Figure 4.13.

**Listing 4.12: A function with some useless code**

```
int square(int x) {
    int n, a, b;
    a = 0;
    n = 2;
    a = x * x;
    b = a * x;
    return a;
}
```

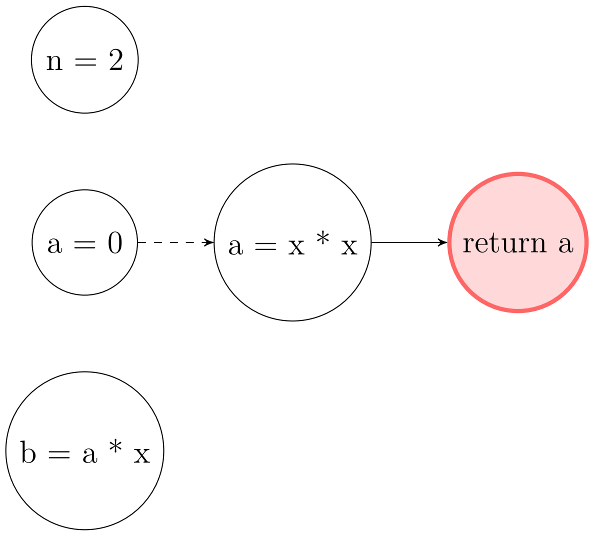


**Figure 4.13: Critical state start**

The statement with the `return` is marked as critical, so all nodes that the return depends upon are also marked as critical. Then all nodes that `a = x * x` depends on are also marked as critical and so forth. Note that `a = 0` is not marked critical because critical propagation only propagates across data dependencies and not control dependencies. The final result of useless code removal is shown in Figure 4.14. The unmarked nodes identified as `n = 2`, `a = 0`, and `b = a * x` have no effect on the program, and can be safely removed.

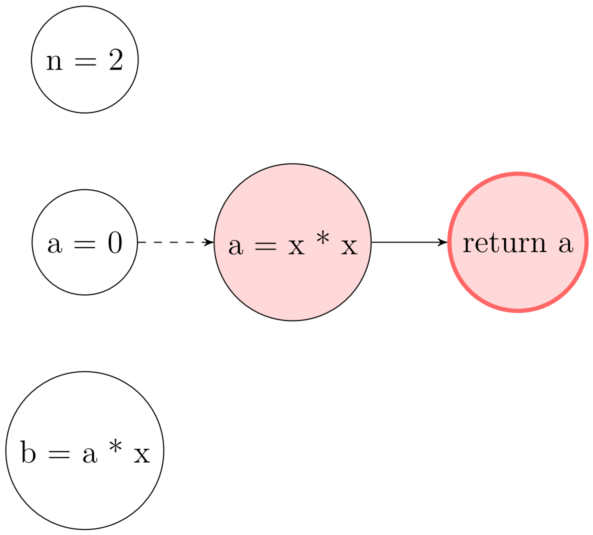


**Figure 4.14: Critical property propagated**

### 4.4.3 Identifier Renaming

Identifiers are the names a programmer chooses for their variables, struct types, and functions. Since they are simply labels, the choice of identifiers will not matter in the overall program meaning. *Normalizer* will rename the identifiers that the programmers have chosen into standardized names. Once an AST is obtained, either by sorting the PDG or by using the original AST, *Normalizer* assigns the identifier names on a first-used, first-assigned basis. The first identifier is assigned the variable `a` to `z` then `aa` to `zz` and so forth. Typically, function parameters are first assigned then local variables.

An example of identifier renaming is shown in Listing 4.13 and the result in Listing 4.14.

**Listing 4.13: Before identifier renaming**

```c
int power(int num, int power) {
    int i = 0;
    int product = 1;
    while (i < power) {
        product = product * num;
        i++;
    }
    return product;
}
```

**Listing 4.14: After identifier renaming**

```c
int power(int a, int b) {
    int c = 0;
    int d = 1;
    while (c < b) {
        d = d * a;
        c++;
    }
    return d;
}
```

Variable shadowing is also considered in identifier renaming. Variable shadowing is a declaration of a variable inside an inner scope whose name already exists in the outer scope. It is not a common occurrence in source code, but it is part of the C language. Take into consideration Listing 4.15.

**Listing 4.15: Before identifier renaming with variable shadowing**

```c
int func(int num) {
    int i = 1;
    while (num < 100) {
        int i = num \% 10;
        num += i;
    }
    return i + num;
}
```

Variable `i` is declared inside the `while` loop, but `i` is already declared in the outer scope. This is legal C, and `i` within the `while` loop occupies a different memory location than the `i` declared as a function local variable. *Normalizer* recognizes variable shadowing and the resulting identifier renaming in shown in Listing 4.16.

**Listing 4.16: After identifier renaming with variable shadowing**

```
int func(int a) {
    int b = 1;
    while (a < 100) {
        int c = a \% 10;
        a += c;
    }
    return b + a;
}
```

### 4.4.4 For-Loop Transformation

For-loops can be transformed into a while loop with relative ease. Given a for-loop of:

```
for (initial; condition; iteration) {
    body
}
```

An equivalent while-loop is:

```
initial
while (condition) {
    body
    iteration
}
```

By restricting the number of ways loops can be written, it is more likely that code will appear the same, therefore increasing the chances of detecting code clones.

### 4.4.5 Function Splitting

Sometimes, the granularity of clone detection is too large. Tools like Moss will only look for clones between files, rather than looking for clones within a file. Simply by fragmenting each function into its own file, Moss can be enabled to look for clones between functions that originally belonged in the same file.

# Chapter 5

# RESULTS

This chapter evaluates the effectiveness of *Normalizer* and compares the quantity and quality of clones caught using and not using *Normalizer*. If the combination of *Normalizer* and the code clone detectors produces more clones than the detector by itself, then code normalization can be considered a beneficial component in detecting code clones.

As input, we used student code submitted in the Introduction to Computer Networks class. The assignment chosen is a chat program, where students write a chat server and a chat client. Both the server and client perform similar tasks, such as setting up a socket and creating or reading packets. These similar functions can best be placed in a common C source file, but can also be placed as clones in both the server and client source files separately. This assignment has a likely chance that code clones will exist, proving to be a good sample for the validation. It is not a goal of students to create code clones, but by the pressures of deadlines, students are prone to write clones. Although the test sample is comprised of students, code clones can be written by anyone.

In this chapter, we select three students and provide an in-depth look at each of them using the various code clone detection tools. Their results are compared for each tool with and without *Normalizer*. We consider the following versions of normalized code that *Normalizer* can output:

1. **original**: This is the original source code but with a slight modification. Since the other source code versions are void of comments and empty lines and each statement occupies exactly one line, *original* must be as well. In order to

give each version a fair assessment, *original* must also be formatted to look like the rest. To remove comments, the GCC preprocessor is invoked with the `-fpreprocess`, `-dD`, and `-E` flags. Removing empty lines and preprocessor directives is achieved by using the Unix program `sed` and deleting empty lines or lines that begin with `#`. The Unix program `indent` is used to enforce that each statement occupies one line, no matter how many characters the line has. `indent` is invoked with the following flags: `-linux`, `--braces-on-func-def-line`, and `-l9999` which forces braces on the same line as `if`, `while`, and `for` statements and function definitions and does not enforce line wrapping until 9999 characters.

2. **noRename_PDG**: This version disables all normalization features, but the source code is still fed through *Normalizer*. Some normalization takes place including C preprocessing, whitespace formatting, and removing modifiers such as `const`, `static`, `volatile`.

3. **normalized**: This is the full normalization which includes statement reordering, identifier renaming, and useless code removal. This version enables all normalization features.

4. **noRename**: This is the full normalization except for identifier renaming. Some tools perform better without renaming identifiers.

5. **noPDG**: This version does not build a PDG. Therefore statement reordering and useless code removal are not performed, but identifier renaming is still performed.

6. **split**: This is like the *normalized* version, but every function is separated into its own file. Some tools perform better when the input is composed of individual files each containing a single function.

7. **noRename_split**: This is like *split*, except the normalization omits identifier renaming.

## 5.1 Student A

The first student is anonymized as Student A.

### 5.1.1 Simian

Simian performed best using the *noRename* version, as it detected the greatest clone mass between any of the versions. In one case, by normalizing the code, a clone was split into two separate clones, whose combined mass is larger than the original clone. The original clone is shown in Listing 5.1 and Listing 5.2.

Listing 5.1: Single clone in *original*

```
while (mssgNum > 0) {
   memcpy(toSend, mssg + consumed,
       maxMssgLen - 1);
   toSend[maxMssgLen - 1] = 0;
   consumed += maxMssgLen - 1;
   packet = makePacketMssg(
       CLIENT_MESSAGE, destLen, dest,
       srcLen, myHandle, toSend);
   if (sendPacket(packet, socket) < 0)
       {
      perror("Packet Message");

      exit(1);
   }
   free(packet);
   mssgNum--;
}
free(toSend);
toSend = malloc(theRest);
memcpy(toSend, mssg + consumed, theRest
    );

memcpy(toSend, mssg + consumed, theRest
    );
packet = makePacketMssg(CLIENT_MESSAGE,
     destLen, dest, srcLen, myHandle,
    toSend);
if (sendPacket(packet, socket) < 0) {
   perror("Packet Message");
   exit(1);
}
free(packet);
free(toSend);
```

Listing 5.2: Single clone in *original*

```
while (mssgNum > 0) {
   memcpy(toSend, mssg + consumed,
       maxMssgLen - 1);
   toSend[maxMssgLen - 1] = 0;
   consumed += maxMssgLen - 1;
   packet = makePacketBroadcast(
       CLIENT_BROADCAST, hLen, myHandle,
        toSend);
   if (sendPacket(packet, socket) < 0)
       {
      perror("Packet Broadcast");

      exit(1);
   }
   free(packet);
   mssgNum--;
}
free(toSend);
toSend = malloc(theRest);
memcpy(toSend, mssg + consumed, theRest
    );

packet = makePacketBroadcast(
    CLIENT_BROADCAST, hLen, myHandle,
    toSend);
if (sendPacket(packet, socket) < 0) {
   perror("Packet Broadcast");
   exit(1);
}
free(packet);
free(toSend);
```

By normalizing the code, the clone has fragmented into two pieces, but each fragment itself is a clone. This phenomena is shown in Listing 5.3 and Listing 5.4. The clones are within the same function and two clones are highlighted in yellow and orange.

**Listing 5.3: Clones within the same function in *noRename***

```
1  while (mssgNum > 0) {
2     memcpy(toSend, mssg + consumed,
          maxMssgLen - 1);
3     mssgNum--;
4     toSend[maxMssgLen - 1] = 0;
5     consumed += maxMssgLen - 1;

6     packet = makePacketMssg(5, destLen,
          dest, srcLen, myHandle, toSend);
7     if (sendPacket(packet, socket) < 0)
          {
8        perror("Packet Message");
9        exit(1);
10    }
11    free(packet);
12 }
13 free(toSend);

14 toSend = malloc(theRest);
15 memcpy(toSend, mssg + consumed, theRest
       );

16 packet = makePacketMssg(5, destLen,
       dest, srcLen, myHandle, toSend);
17 if (sendPacket(packet, socket) < 0) {
18    perror("Packet Message");
19    exit(1);
20 }
21 free(packet);
22 free(toSend);
```

**Listing 5.4: Clones within the same function in *noRename***

```
1  while (mssgNum > 0) {
2     memcpy(toSend, mssg + consumed,
          maxMssgLen - 1);
3     mssgNum--;
4     toSend[maxMssgLen - 1] = 0;
5     consumed += maxMssgLen - 1;

6     packet = makePacketBroadcast(4, hLen
          , myHandle, toSend);
7     if (sendPacket(packet, socket) < 0)
          {
8        perror("Packet Broadcast");
9        exit(1);
10    }
11    free(packet);
12 }
13 free(toSend);

14 toSend = malloc(theRest);
15 memcpy(toSend, mssg + consumed, theRest
       );

16 packet = makePacketBroadcast(4, hLen,
       myHandle, toSend);
17 if (sendPacket(packet, socket) < 0) {
18    perror("Packet Broadcast");
19    exit(1);
20 }
21 free(packet);
22 free(toSend);
```

The only difference between the two versions is the movement of `mssgNum--` upwards from line 11 to line 3. This potentially made the original code clone too small to count as a clone. As a result, a different code clone has been identified. However, when taking both functions into consideration, the entire function should be considered a clone to each other. Simian has failed to detect the clone to that extent, but it has detected a higher clone mass.

Another clone that Simian in *noRename* is a one line difference from *original*. The clone from *noRename* is shown in Listing 5.5 and Listing 5.6.

**Listing 5.5: Clone in *noRename***

```c
int mssgNum = (strlen(mssg) + 1) /
    maxMssgLen;
int theRest = ((strlen(mssg) + 1) %
    maxMssgLen) + mssgNum;
toSend = malloc(maxMssgLen);
while (mssgNum > 0) {
   memcpy(toSend, mssg + consumed,
       maxMssgLen - 1);
   mssgNum--;
   toSend[maxMssgLen - 1] = 0;
   consumed += maxMssgLen - 1;

   packet = makePacketMssg(5, destLen,
       dest, srcLen, myHandle, toSend);
   if (sendPacket(packet, socket) < 0)
       {
      perror("Packet Message");
      exit(1);
   }
   free(packet);
```

**Listing 5.6: Clone in *noRename***

```c
int mssgNum = (strlen(mssg) + 1) /
    maxMssgLen;
int theRest = ((strlen(mssg) + 1) %
    maxMssgLen) + mssgNum;
toSend = malloc(maxMssgLen);
while (mssgNum > 0) {
   memcpy(toSend, mssg + consumed,
       maxMssgLen - 1);
   mssgNum--;
   toSend[maxMssgLen - 1] = 0;
   consumed += maxMssgLen - 1;

   packet = makePacketBroadcast(4, hLen
       , myHandle, toSend);
   if (sendPacket(packet, socket) < 0)
       {
      perror("Packet Broadcast");
      exit(1);
   }
   free(packet);
```

The *original* version is shown in Listing 5.7 and Listing 5.8.

**Listing 5.7: Listing 5.5 as written in *original***

```c
int mssgNum = (strlen(mssg) + 1) /
    maxMssgLen;
int consumed = 0;
int theRest = ((strlen(mssg) + 1) %
    maxMssgLen) + mssgNum;
toSend = malloc(maxMssgLen);
while (mssgNum > 0) {
   memcpy(toSend, mssg + consumed,
       maxMssgLen - 1);
   toSend[maxMssgLen - 1] = 0;
   consumed += maxMssgLen - 1;

   packet = makePacketMssg(
       CLIENT_MESSAGE, destLen, dest,
       srcLen, myHandle, toSend);
   if (sendPacket(packet, socket) < 0)
       {
      perror("Packet Message");
      exit(1);
   }
   free(packet);
   mssgNum--;
```

**Listing 5.8: Listing 5.6 as written in *original***

```c
int mssgNum = (strlen(mssg) + 1) /
    maxMssgLen;
int consumed = 0;
int theRest = ((strlen(mssg) + 1) %
    maxMssgLen) + mssgNum;
toSend = malloc(maxMssgLen);
while (mssgNum > 0) {
   memcpy(toSend, mssg + consumed,
       maxMssgLen - 1);
   toSend[maxMssgLen - 1] = 0;
   consumed += maxMssgLen - 1;

   packet = makePacketBroadcast(
       CLIENT_BROADCAST, hLen, myHandle,
        toSend);
   if (sendPacket(packet, socket) < 0)
       {
      perror("Packet Broadcast");
      exit(1);
   }
   free(packet);
   mssgNum--;
```

The clone size is the same in *noRename* and *original*, but by statement reordering, `int consumed = 0` is moved away from the clone and `mssgNum--` is moved into the clone. By gaining and losing one line, the clone sizes remain the same, but shows that a clone size of nine is possible.

When performing full normalization including identifier renaming, Simian performed worse than *noRename*. The same clone found in Listing 5.3 and Listing 5.4 from *noRename* is not identified in *normalized*, which is shown in Listing 5.9 and Listing 5.10.

**Listing 5.9: Listing 5.3 clone lost by identifier renaming**

```c
int j = (strlen(a) + 1) / i;
int k = ((strlen(a) + 1) % i) + j;
d = malloc(i);
while (j > 0) {
   memcpy(d, a + f, i - 1);
   j--;
   d[i - 1] = 0;
   f += i - 1;
```

**Listing 5.10: Listing 5.4 clone lost by identifier renaming**

```c
int h = (strlen(a) + 1) / g;
int i = ((strlen(a) + 1) % g) + h;
c = malloc(g);
while (h > 0) {
   memcpy(c, a + e, g - 1);
   h--;
   c[g - 1] = 0;
   e += g - 1;
```

The different identifier names have caused Simian to fail catastrophically. Between the two listings from *normalized*, the identifier names diverged. The extra variable causes all subsequent variable declarations to take a different name, causing a shift in identifier names. Despite perfectly preserving the structure of the clones, the clone is undetected. Clones that exist within the same function, such as from Listing 5.3 and Listing 5.4 are unaffected by identifier renaming since the clones use the same renamed identifiers.

The overall performance of Simian on Student A is shown in Figure 5.1. The total number of clone lines found is written on top of the solid bar and the total number of clones found is scribed in white on top of the shorter bar.

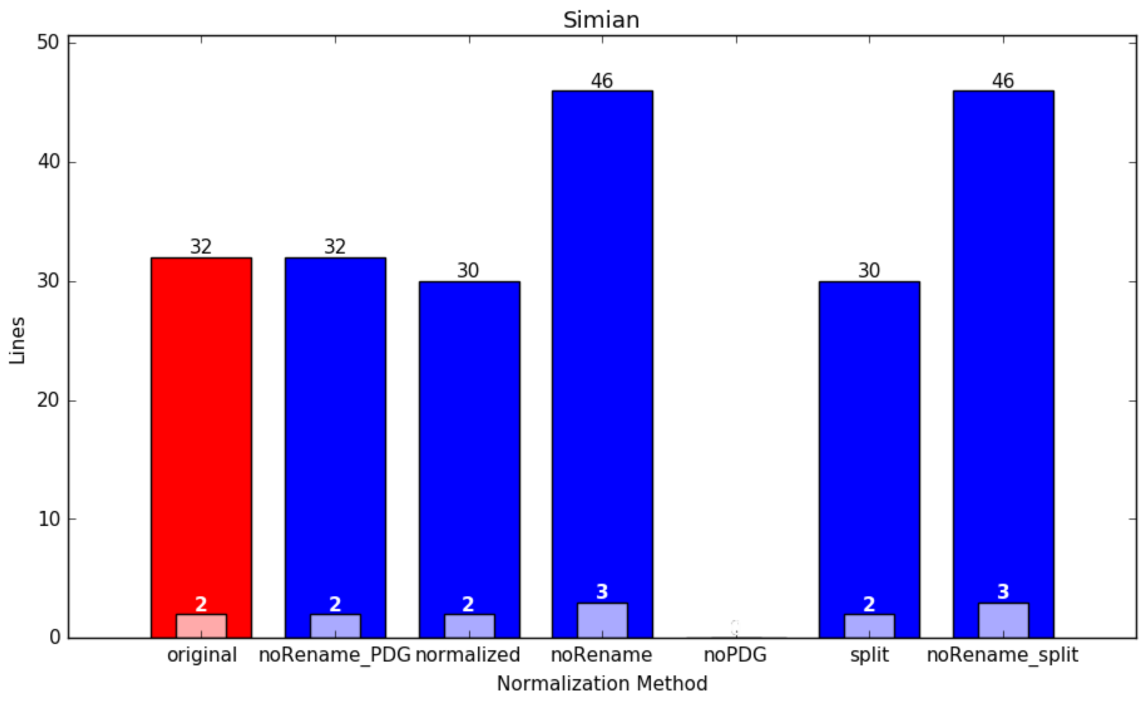


**Figure 5.1: Summary performance of Simian on Student A**

The performance of *noPDG* is zero because this mode will change identifier names but will not reorder the statements. As already seen, changing identifier names causes Simian to fail. Using *noRename* performs better than *original*. For the remainder of this chapter, when comparing Simian runs, *noRename* will be considered instead of *normalized* as the best candidate for normalized code.

### 5.1.2 JPlag

Through normalizing the code, JPlag was able to catch an entirely new clone. The new clone is shown in Listing 5.11 and Listing 5.12.

Listing 5.11: New *normalized* clone found

```
if (global_c > 0) {
   c = global_d[global_c - 1] + 1;
}
for (b = 0; b < global_c; b++) {
   FD_SET(global_d[b], &a);
}
if (mySelect(c, (fd_set *) & a, ((void
    *)0), ((void *)0), ((void *)0)) ==
    -1) {
   perror("Error selecting FD to read
       from");
   exit(1);
}
if (FD_ISSET(global_b, &a)) {
   return global_b;
} else {
```

Listing 5.12: New *normalized* clone found

```
if (global_a) {
   FD_SET(fileno(stdin), &a);
}
if (mySelect(b, (fd_set *) & a, ((void
    *)0), ((void *)0), ((void *)0)) ==
    -1) {
   perror("Error selecting FD to read
       from");
}
if (FD_ISSET(global_c, &a)) {
   return global_c;
}
```

To investigate how this new clone was discovered, the original code which the normalized code is derived from is shown in Listing 5.13 and Listing 5.14.

Listing 5.13: Undiscovered clone in *original*

```
for(i = 0; i < clientNum; i++) {
    FD_SET(clientFDs[i],&fdvar);
}
if (clientNum > 0) {
    maxFD = clientFDs[clientNum-1] + 1;
}
if(select(maxFD,(fd_set *) &fdvar, NULL
    , NULL, NULL) == -1) {
    perror("Error selecting FD to read
        from");
    exit(1);
}
if(FD_ISSET(serverFD, &fdvar)) {
    return serverFD;
} else {
```

Listing 5.14: Undiscovered clone in *original*

```
if(isConnected) {
    FD_SET(fileno(stdin),&fdvar);
}
if(select(maxFD,(fd_set *) &fdvar, NULL,
     NULL, NULL) == -1) {
    perror("Error selecting FD to read
        from");
    exit(1);
}
if(FD_ISSET(socketFD, &fdvar)) {
    return socketFD;
}
```

The difference is the movement of `if (clientNum > 0)` in Listing 5.13 from line 4 to line 1. In the original, it was inserted in the middle of the detected clone. *Normalizer* is able to determine that the `if` statement is free to move anywhere before the `select` call. Since `if` statements are given a higher priority in statement reordering than a `for` loop, so it was moved upwards. By moving it up, the clone appears as one contiguous unit and thus was detected.

A limitation of JPlag is its inability to find clones within the same file. So by splitting each function into its own file and inputting those files into JPlag, a significant improvement in clone detection has been shown. The result is that a large clone was exposed which originally belonged in the same file. The clone is shown in Listing 5.15 and Listing 5.16.

Listing 5.15: A clone found in *split*

```
int f = 0;
int g = strlen(global_b);
int h = strlen(b);
int i = 1000 - (g + 1) - (h + 1) -
    sizeof(header);
int j = (strlen(a) + 1) / i;
int k = ((strlen(a) + 1) % i) + j;
d = malloc(i);
while (j > 0) {
   memcpy(d, a + f, i - 1);
   j--;
   d[i - 1] = 0;
   f += i - 1;
   e = makePacketMssg(5, h, b, g,
       global_b, d);
   if (sendPacket(e, c) < 0) {
      perror("Packet Message");
      exit(1);
   }
   free(e);
}
free(d);
d = malloc(k);
memcpy(d, a + f, k);
e = makePacketMssg(5, h, b, g, global_b
    , d);
if (sendPacket(e, c) < 0) {
   perror("Packet Message");
   exit(1);
}
free(e);
free(d);
```

Listing 5.16: A clone found in *split*

```
int e = 0;
int f = strlen(global_b);
int g = 1000 - f - sizeof(header) - 1;
int h = (strlen(a) + 1) / g;
int i = ((strlen(a) + 1) % g) + h;
c = malloc(g);
while (h > 0) {
   memcpy(c, a + e, g - 1);
   h--;
   c[g - 1] = 0;
   e += g - 1;
   d = makePacketBroadcast(4, f,
       global_b, c);
   if (sendPacket(d, b) < 0) {
      perror("Packet Broadcast");
      exit(1);
   }
   free(d);
}
free(c);
c = malloc(i);
memcpy(c, a + e, i);
d = makePacketBroadcast(4, f, global_b,
    c);
if (sendPacket(d, b) < 0) {
   perror("Packet Broadcast");
   exit(1);
}
free(d);
free(c);
```

The clone discovered is a larger clone of what Simian discovered. Even when the identifiers have been renamed, JPlag was able to detect the clone, something that Simian is not able to do.

By finding other clones within the same source file including the recently discussed clone in Listing 5.15 and Listing 5.16, JPlag caught a total mass of at least two times more than the original source code. A summary is shown in Figure 5.2.

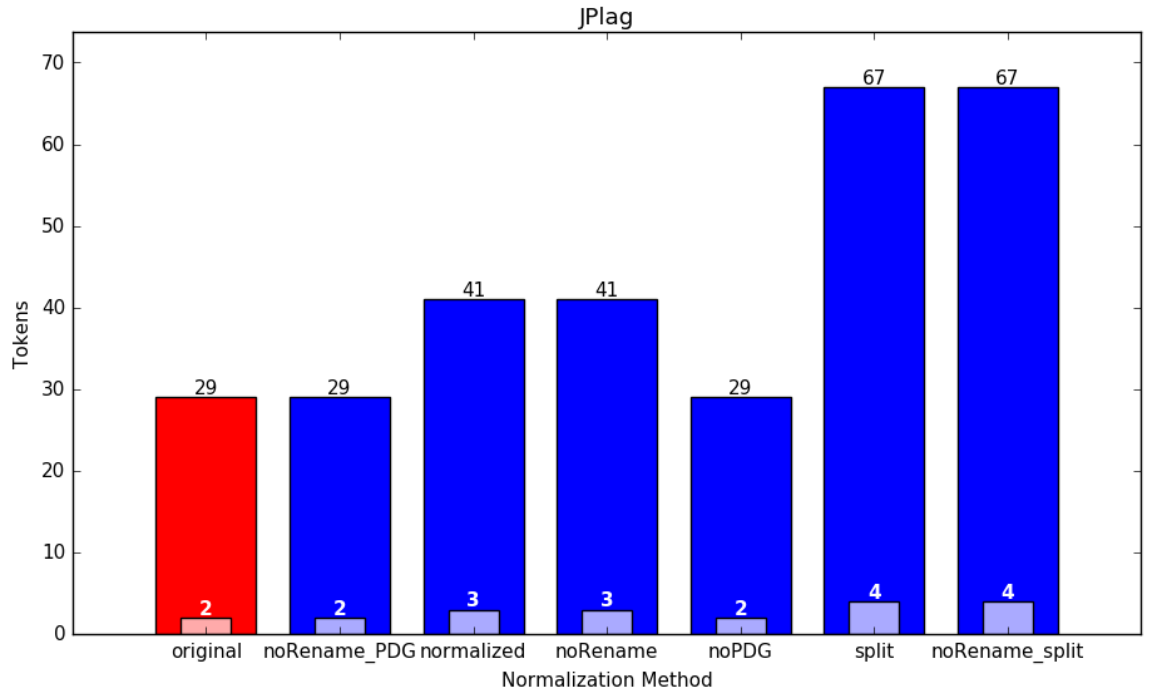


**Figure 5.2: Summary performance of JPlag on Student A**

### 5.1.3 CloneDR

CloneDR is also used to find code clones. However, preprocessor directives left in the input are considered code. The reason the *original* version is stripped of preprocessor directives is because CloneDR would report a series of `#include`s as a clone. Listing 5.17 is a real CloneDR reported clone on unmodified original source code.

**Listing 5.17: Preprocessor directives counted as clones**

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/uio.h>
#include <sys/time.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <strings.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
```

Since `#include`s do not contain any coding logic, it would be unfit for it to be considered as a code clone.

Aside from nonsensical clones, CloneDR performs poorly with identifier renaming. It was able to detect the same clone as JPlag in *original*, which is shown in Listing 5.18 and Listing 5.19.

Listing 5.18: Clone from the *original* code

```c
int hLen = strlen(myHandle);
int maxMssgLen = MAXBUF - hLen - sizeof
    (header) - 1;
int mssgNum = (strlen(mssg) + 1) /
    maxMssgLen;
int consumed = 0;
int theRest = ((strlen(mssg) + 1) %
    maxMssgLen) + mssgNum;
toSend = malloc(maxMssgLen);
while(mssgNum > 0) {
    memcpy(toSend, mssg + consumed,
        maxMssgLen - 1);
    toSend[maxMssgLen - 1] = 0;
    consumed += maxMssgLen - 1;
    packet = makePacketBroadcast(
        CLIENT_BROADCAST, hLen, myHandle
        , toSend);
    if(sendPacket(packet, socket) < 0)
        {
        perror("Packet Broadcast");
        exit(1);
    }
    free(packet);
    mssgNum--;
}
free(toSend);
toSend = malloc(theRest);
memcpy(toSend, mssg + consumed, theRest
    );
packet = makePacketBroadcast(
    CLIENT_BROADCAST, hLen, myHandle,
    toSend);
if (sendPacket(packet, socket) < 0) {
   perror("Packet Broadcast");
   exit(1);
}
free(packet);
free(toSend);
```

Listing 5.19: Clone from the *original* code

```c
int srcLen = strlen(myHandle);
int destLen = strlen(dest);
int maxMssgLen = MAXBUF - (srcLen + 1)
    - (destLen + 1) - sizeof(header);
int mssgNum = (strlen(mssg) + 1) /
    maxMssgLen;
int consumed = 0;
int theRest = ((strlen(mssg) + 1) %
    maxMssgLen) + mssgNum;
toSend = malloc(maxMssgLen);
while(mssgNum > 0) {
    memcpy(toSend, mssg + consumed,
        maxMssgLen - 1);
    toSend[maxMssgLen - 1] = 0;
    consumed += maxMssgLen - 1;
    packet = makePacketMssg(
        CLIENT_MESSAGE, destLen, dest,
        srcLen, myHandle, toSend);
    if(sendPacket(packet, socket) < 0)
        {
        perror("Packet Message");
        exit(1);
    }
    free(packet);
    mssgNum--;
}
free(toSend);
toSend = malloc(theRest);
memcpy(toSend, mssg + consumed, theRest
    );
packet = makePacketMssg(CLIENT_MESSAGE,
     destLen, dest, srcLen, myHandle,
    toSend);
if (sendPacket(packet, socket) < 0) {
   perror("Packet Message");
   exit(1);
}
free(packet);
free(toSend);
```

However, when including identifier renaming in *normalized*, CloneDR was unable to report the clone from *original* in the mutated code. A portion of the normalized code is shown in Listing 5.20 and Listing 5.21.

**Listing 5.20: Listing 5.18 in *normalized***

```c
int e = 0;
int f = strlen(global_b);
int g = 1000 - f - sizeof(header) - 1;
int h = (strlen(a) + 1) / g;
int i = ((strlen(a) + 1) % g) + h;
c = malloc(g);
while (h > 0) {
   memcpy(c, a + e, g - 1);
   h--;
   c[g - 1] = 0;
   e += g - 1;
   d = makePacketBroadcast(4, f,
       global_b, c);
   if (sendPacket(d, b) < 0) {
       perror("Packet Broadcast");
       exit(1);
    }
   free(d);
}
free(c);
c = malloc(i);
memcpy(c, a + e, i);
```

**Listing 5.21: Listing 5.19 in *normalized***

```c
int f = 0;
int g = strlen(global_b);
int h = strlen(b);
int i = 1000 - (g + 1) - (h + 1) -
    sizeof(header);
int j = (strlen(a) + 1) / i;
int k = ((strlen(a) + 1) % i) + j;
d = malloc(i);
while (j > 0) {
   memcpy(d, a + f, i - 1);
   j--;
   d[i - 1] = 0;
   f += i - 1;
   e = makePacketMssg(5, h, b, g,
       global_b, d);
   if (sendPacket(e, c) < 0) {
      perror("Packet Message");
      exit(1);
   }
   free(e);
}
free(d);
d = malloc(k);
memcpy(d, a + f, k);
```

The normalized code clone exhibits very similar structure, but the identifier names have been shifted by one. Upon suspicion that CloneDR may use identifier names, a further test is performed using *noRename*, which is shown in Listing 5.22 and Listing 5.23.

Listing 5.22: Listing 5.18 in *noRename*

```
int consumed = 0;
int hLen = strlen(myHandle);
int maxMssgLen = 1000 - hLen - sizeof(
    header) - 1;
```

```
int mssgNum = (strlen(mssg) + 1) /
    maxMssgLen;
int theRest = ((strlen(mssg) + 1) %
    maxMssgLen) + mssgNum;
toSend = malloc(maxMssgLen);
while (mssgNum > 0) {
   memcpy(toSend, mssg + consumed,
       maxMssgLen - 1);
   mssgNum--;
   toSend[maxMssgLen - 1] = 0;
   consumed += maxMssgLen - 1;
   packet = makePacketBroadcast(4, hLen
       , myHandle, toSend);
   if (sendPacket(packet, socket) < 0)
       {
      perror("Packet Broadcast");
      exit(1);
   }
   free(packet);
}
free(toSend);
toSend = malloc(theRest);
memcpy(toSend, mssg + consumed, theRest
    );
packet = makePacketBroadcast(4, hLen,
    myHandle, toSend);
if (sendPacket(packet, socket) < 0) {
   perror("Packet Broadcast");
   exit(1);
}
free(packet);
free(toSend);
```

Listing 5.23: Listing 5.19 in *noRename*

```
int consumed = 0;
int srcLen = strlen(myHandle);
int destLen = strlen(dest);
int maxMssgLen = 1000 - (srcLen + 1) -
    (destLen + 1) - sizeof(header);
```

```
int mssgNum = (strlen(mssg) + 1) /
    maxMssgLen;
int theRest = ((strlen(mssg) + 1) %
    maxMssgLen) + mssgNum;
toSend = malloc(maxMssgLen);
while (mssgNum > 0) {
   memcpy(toSend, mssg + consumed,
       maxMssgLen - 1);
   mssgNum--;
   toSend[maxMssgLen - 1] = 0;
   consumed += maxMssgLen - 1;
   packet = makePacketMssg(5, destLen,
       dest, srcLen, myHandle, toSend);
   if (sendPacket(packet, socket) < 0)
       {
      perror("Packet Message");
      exit(1);
   }
   free(packet);
}
free(toSend);
toSend = malloc(theRest);
memcpy(toSend, mssg + consumed, theRest
    );
packet = makePacketMssg(5, destLen,
    dest, srcLen, myHandle, toSend);
if (sendPacket(packet, socket) < 0) {
   perror("Packet Message");
   exit(1);
}
free(packet);
free(toSend);
```

The clone has been rediscovered, but it is smaller due to the movement of `int consumed = 0` away from the main clone towards the top as its own fragment. Since the clone has been rediscovered, CloneDR does rely on identifier names. Therefore, for all subsequent analysis for CloneDR, *noRename* will be considered instead of *normalized* since *noRename* will give better results than *normalized*. The overall performance of CloneDR on Student A is shown in Figure 5.3.

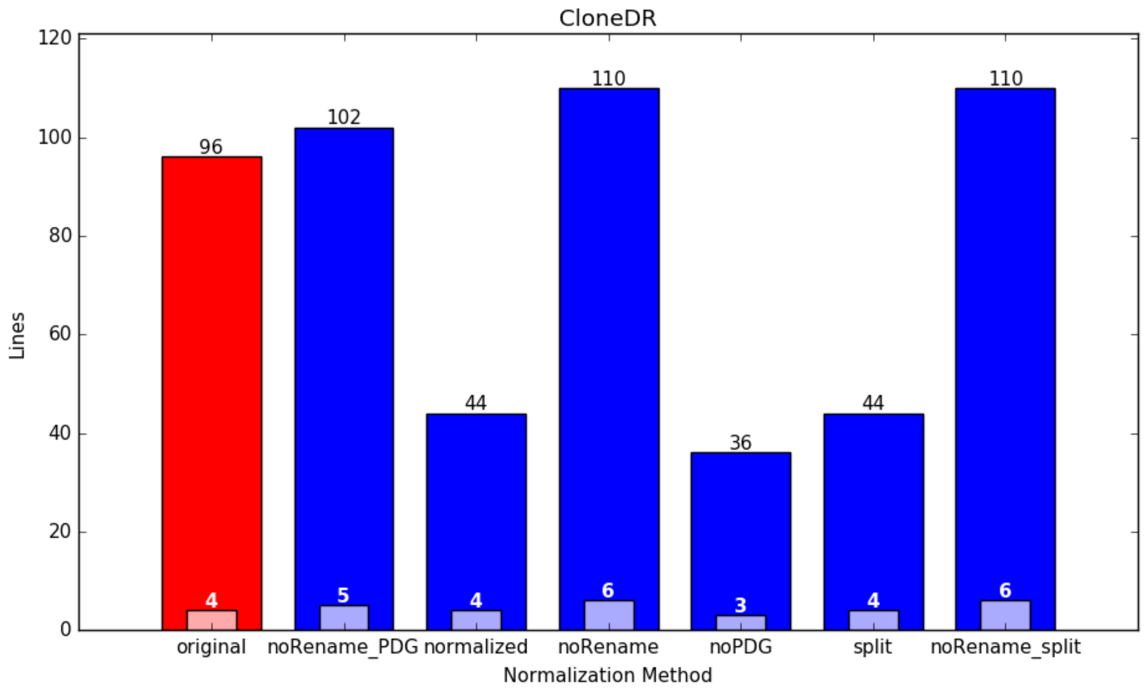


**Figure 5.3: Summary performance of CloneDR on Student A**

There are some versions that perform better than *original*. *noRename_PDG* found a total of six more cloned lines, which are shown in Listing 5.24 and Listing 5.25.

**Listing 5.24: Clone in *noRename_PDG***

```
if (mySelect(maxFD, (fd_set *) & fdvar,
    ((void *)0), ((void *)0), ((void *)
   0)) == -1) {
   perror("Error selecting FD to read
       from");
   exit(1);
}
```

**Listing 5.25: Clone in *noRename_PDG***

```
if (mySelect(maxFD, (fd_set *) & fdvar,
    ((void *)0), ((void *)0), ((void *)
   0)) == -1) {
   perror("Error selecting FD to read
       from");
   exit(1);
}
```

*noRename_PDG* is a version that does not perform any normalization, but the clone mass has increased. A possible reason behind this is preprocessing. The *original* code is presented in Listing 5.26 and Listing 5.27.

**Listing 5.26: Original code of Listing 5.24**

```
if (select(maxFD, (fd_set *) & fdvar,
    NULL, NULL, NULL) == -1) {
   perror("Error selecting FD to read
       from");
   exit(1);
}
```

**Listing 5.27: Original code of Listing 5.25**

```
if (select(maxFD, (fd_set *) & fdvar,
    NULL, NULL, NULL) == -1) {
   perror("Error selecting FD to read
       from");
   exit(1);
}
```

Through normalization, `NULL`s have been expanded into `((void *)0)`, which creates a bigger overall AST. It could be the case that the larger trees are just enough to be a clone over the size threshold, therefore marking this as a clone.

Version *noRename* performed the best numerically. By reordering, despite a one line decrease of the clone in Listing 5.22 and Listing 5.23, another clone was found which raised the total clone mass.

Normalization had a small beneficial impact on CloneDR for this student.

### 5.1.4 Moss

As a plagiarism detection tool, Moss is a conservative clone detector. In order to use Moss however, the source code must be broken into individual files because Moss can only find clones between files but not within files. The version *noRename_PDG_split* will serve as the control version for Moss as it does not perform any normalization on the code besides the GCC preprocessing. For Student A, Moss demonstrated poorer performance using normalized code than using original code. A clone that was detected in *noRename_PDG_split* but not in *split* is shown in Listing 5.28 and Listing 5.29.

**Listing 5.28: Clone in *noRename_PDG_split***

```c
uint8_t *packet = malloc(1000);
uint8_t *toFree = packet;

int numBytes = 0;
if ((numBytes = myRecv(socketFD, packet
    , 1000, 0)) < 0) {
   perror("recv call");
   exit(-1);
}
if (numBytes == 0) {
   printf("Server terminated\n");

   close(socketFD);
   exit(1);
}
```

**Listing 5.29: Clone in *noRename_PDG_split***

```c
uint8_t *packet = malloc(1000);

int numBytes = 0;
if ((numBytes = myRecv(fd, packet,
    1000, 0)) < 0) {
   perror("recv call");
   exit(-1);
}
if (numBytes == 0) {
   removeClient(fd);

}
```

The normalized code is displayed in Listing 5.30 and Listing 5.31.

**Listing 5.30: Listing 5.28 as normalized in *split***

```
uint8_t *a = malloc(1000);
int b = 0;
uint8_t *c = a;
if ((b = myRecv(global_c, a, 1000, 0))
     < 0) {
    perror("recv call");
    exit(-1);
}
if (b == 0) {
    printf("Server terminated\n");
    close(global_c);
    exit(1);
}
```

**Listing 5.31: Listing 5.29 as normalized in *split***

```
uint8_t *b = malloc(1000);
int c = 0;
if ((c = myRecv(a, b, 1000, 0)) < 0) {
    perror("recv call");
    exit(-1);
}
if (c == 0) {
    removeClient(a);
}
```

Evidently, `int b = 0` was moved away from the clone in the normalized code in Listing 5.30, where `b` is the renamed variable for `numBytes`. Now, `uint8_t *c = a` is inserted into the middle of the clone but no matching statement is present in Listing 5.29. Since the code clone has been fragmented, the clone size was too small to trigger the threshold, causing the non-detection.

An improvement that could be used is to sort statements to place variable assignments just before they are used. In Listing 5.28, variable `c` is assigned a value in between the assignment to `b` and the `if` statement. However, `c` will not be used for several more lines. If sorting would pick statements that are in immediate need by another statement, then this problem could be avoided.

A summary of Moss is shown in Figure 5.4. *split* and *noRename_split* performed the same as Moss does not take the names of identifiers into account. Moss is expected not to consider identifier names because renaming identifiers is an easy way to plagiarize code. Under Student A, normalization had a negative impact on detection.

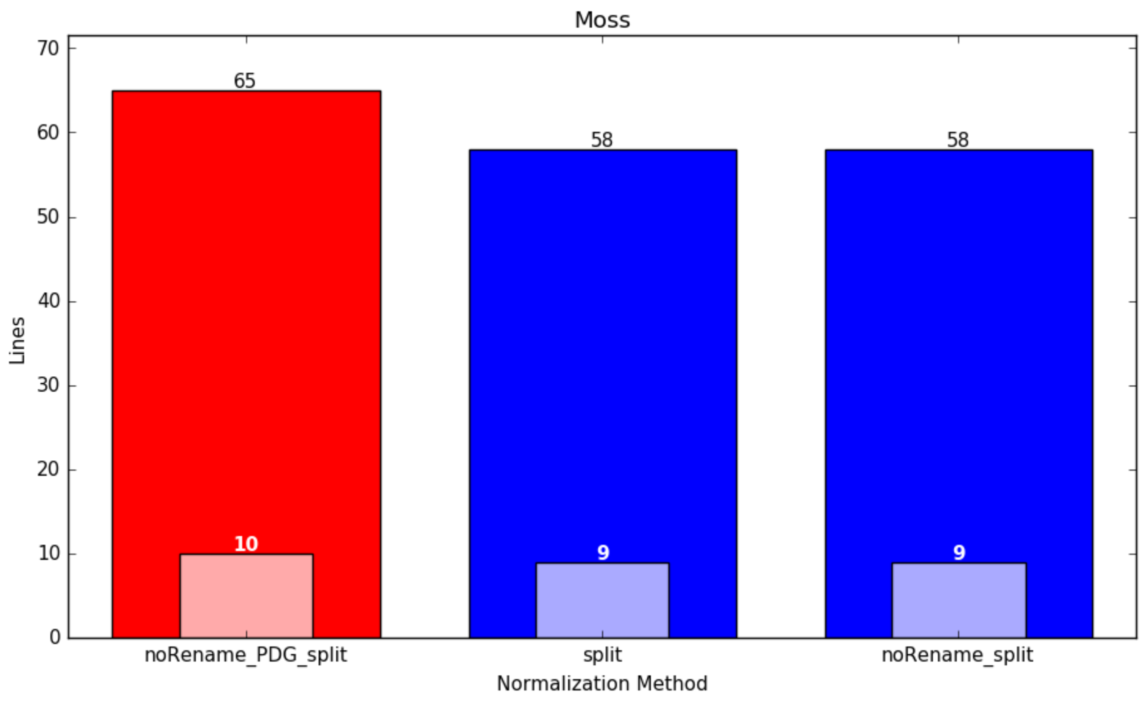


**Figure 5.4: Summary performance of Moss on Student A**

## 5.2 Student B

This section examines the submission of a second student, Student B, in depth. This submission features code clones that appear in more than two areas.

### 5.2.1 Simian

Normalization does not increase clone detection using Simian. As with Student A, identifier renaming has a negative effect on Simian, therefore the versions that include identifier renaming, i.e. *normalized*, *noPDG*, and *split* have reduced clone findings. The rest, *noRename_PDG*, *noRename*, and *noRename_split* all report the same findings.

For this student, Simian reports there is a common clone as it appears a total of four times. The clone taken from *original* is shown in Listing 5.32.

**Listing 5.32: Clone from *original* duplicated four times**

```
handle_len = *data_buf;
data_buf += sizeof(handle_len);
for (i = 0; i < handle_len; i++) {
   handle[i] = *data_buf;
   data_buf++;
}
handle[i] = '\0';
```

This clone is identically duplicated in three other locations. Two of the clone instances are featured in their own clone as a part of a larger clone. The featured larger clone, which is identical to each other is shown in Listing 5.33.

**Listing 5.33: Larger clone with Listing 5.32**

```
handle_len = *data_buf;
data_buf += sizeof(handle_len);
for (i = 0; i < handle_len; i++) {
   handle[i] = *data_buf;
   data_buf++;
}
handle[i] = '\0';
printf("\n%s: %s\n", handle, data_buf);
```

This is the same clone as Listing 5.32 with the addition of a `printf` statement at the bottom. In fact, each of the first four original clones of Listing 5.32 has a `printf` statement, but only two have the same format string in Listing 5.33. Since there are four instances of the small clone present, it is a strong hint that it should be abstracted out. Good candidates to replace the clone are `memcpy` and `strcpy`.

Simian's performance on Student B is shown in Figure 5.5.

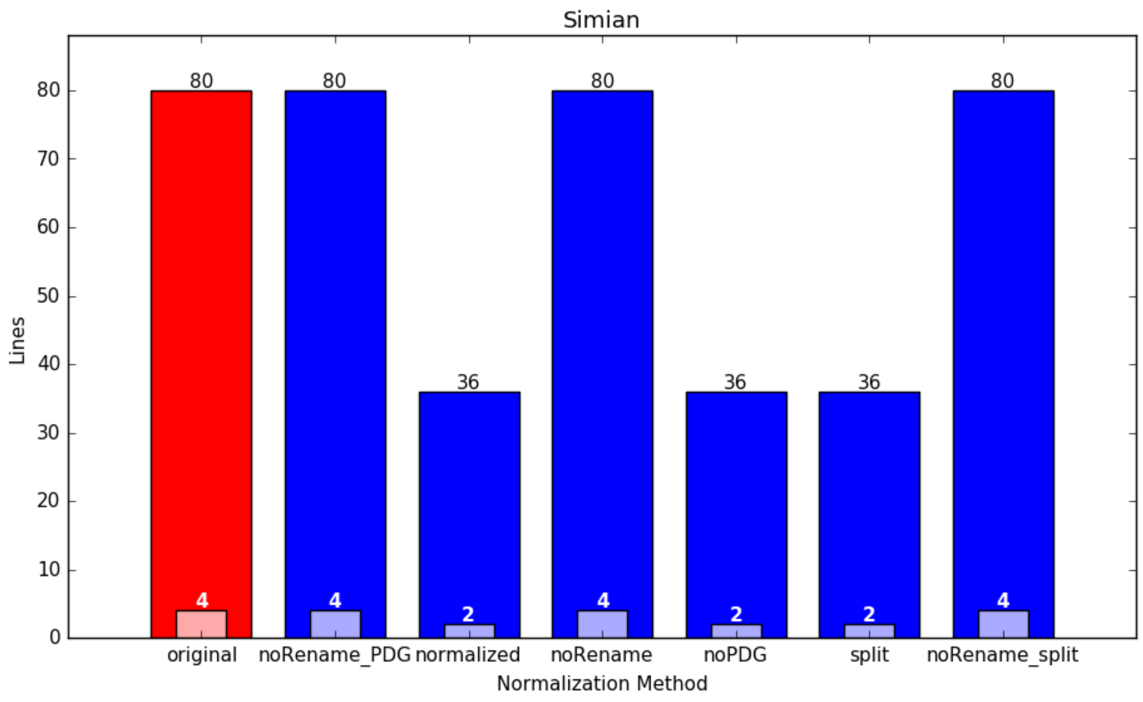


**Figure 5.5: Summary performance of Simian on Student B**

Clone detection degraded if normalization included identifier renaming. There are cases where some clones were were preserved even when the identifiers were renamed. Some functions were small enough that they did not have differing number of variables. However, it is more likely that a clone would get lost through identifier renaming. Such a clone is given in Listing 5.34 and Listing 5.35.

**Listing 5.34: Lost clone in *normalized* by identifier renaming**

```
if ((b = recv(a, c, 1259, 0)) <= 0) {
   if (b < 0) {
      perror("recv call");
      exit(-1);
   }
   if (close(a) < 0) {
      perror("close call");
      exit(-1);
```

**Listing 5.35: Lost clone in *normalized* by identifier renaming**

```
if ((f = recv(a, c, 1259, 0)) <= 0) {
   if (f < 0) {
      perror("recv call");
      exit(-1);
   }
   if (close(a) < 0) {
      perror("close call");
      exit(-1);
```

There is a variable shift which causes the difference in the name of the variable in the `if` statement's conditional. That was enough for Simian to not identify each of these two code sections as a clone, even when evidently they have the same structure.

### 5.2.2 JPlag

JPlag sees significant improvement from normalization. The majority of the increased clone matches comes from splitting the program into separate functions, but also still benefits from statement reordering and useless code removal. An example of useless code removal is shown in Listing 5.36 and Listing 5.37. First, the *original* version is shown.

**Listing 5.36: Clone found in *original***

```
uint16_t pack_len, pack_len_net;
uint8_t flag = 0;
char *temp = data_buf;
memcpy(&pack_len_net, temp, sizeof(
    pack_len_net));
pack_len = ntohs(pack_len_net);
temp += sizeof(pack_len);
flag = *temp;

if (flag == 4) {
   Broadcast(data_buf, *cli_sk_head,
       pack_len, fd_src);
} else if (flag == 5) {
   DirectMessage(data_buf, *cli_sk_head
       , pack_len);
} else if (flag == 10) {
   SendList(data_buf, *cli_sk_count,
       fd_src, *cli_sk_head);
} else if (flag == 8) {
   EndConnection(data_buf, fd_src,
       cli_sk_head, cli_sk_count);
}
```

**Listing 5.37: Clone found in *original***

```
uint16_t pack_len, pack_len_net;
uint8_t flag;
char *temp = data_buf;
int isExit = 0;
pack_len_net = *temp;
pack_len = ntohs(pack_len_net);
temp += sizeof(pack_len);
flag = *temp;
temp += sizeof(flag);

if (flag == FLAG4) {
   InterpreteFlag4(temp);
} else if (flag == FLAG5) {
   InterpreteFlag5(temp);
} else if (flag == FLAG7) {
   InterpreteFlag7(temp);
} else if (flag == FLAG11) {
   InterpreteFlag11(temp, handle_numb);
} else if (flag == FLAG12) {
```

The *normalized* version of the code is shown in Listing 5.38 and Listing 5.39.

**Listing 5.38: Larger clone in *normalized***

```
uint16_t e, f;
uint8_t g = 0;
char *h = a;
memcpy(&f, h, sizeof(f));

e = ntohs(f);
h += sizeof(e);
g = *h;
if (g == 4) {
    Broadcast(a, *b, e, c);
} else if (g == 5) {
    DirectMessage(a, *b, e);
} else if (g == 10) {
    SendList(a, *d, c, *b);
} else if (g == 8) {
    EndConnection(a, c, b, d);
}
```

**Listing 5.39: Larger clone in *normalized***

```
uint8_t d;
uint16_t e, f;
int g = 0;
char *h = a;
f = *h;

e = ntohs(f);
h += sizeof(e);
d = *h;
if (d == 4) {
    InterpreteFlag4(h);
} else if (d == 5) {
    InterpreteFlag5(h);
} else if (d == 7) {
    InterpreteFlag7(h);
} else if (d == 11) {
    InterpreteFlag11(h, c);
} else if (d == 12) {
```

Compared against *original*, the code clone in *normalized* has increased in size by three lines. In Listing 5.37, the assignment `temp += sizeof(flag)` is not needed because `temp` is a local variable not used in the remainder of the function. By removing that line, each clone has the same three statements preceding it: the assignments into `pack_len`, `temp`, and `flag`. These three lines augment the already existing clone, creating a larger clone.

A better analysis could be performed if `memcpy(&f, h, sizeof(f))` was rewritten as `*f = *((uint16_t*)h)`. If that statement was rewritten, a total of two more lines would be matched into the clone, shown in Listing 5.40 and Listing 5.41.

Listing 5.40: Modified assignment from Listing 5.38

```
uint16_t e, f;

uint8_t g = 0;
char *h = a;
*f = *((uint16_t *) h);
e = ntohs(f);
h += sizeof(e);
g = *h;
if (g == 4) {
   Broadcast(a, *b, e, c);
} else if (g == 5) {
   DirectMessage(a, *b, e);
} else if (g == 10) {
   SendList(a, *d, c, *b);
} else if (g == 8) {
   EndConnection(a, c, b, d);
}
```

Listing 5.41: Same code from Listing 5.39

```
uint8_t d;
uint16_t e, f;

int g = 0;
char *h = a;
f = *h;
e = ntohs(f);
h += sizeof(e);
d = *h;
if (d == 4) {
   InterpreteFlag4(h);
} else if (d == 5) {
   InterpreteFlag5(h);
} else if (d == 7) {
   InterpreteFlag7(h);
} else if (d == 11) {
   InterpreteFlag11(h, c);
} else if (d == 12) {
```

If `memcpy` is replaced by an assignment, both clone instances can now match the assignment into `g` and `f`, creating a larger clone. How the code was originally written plays a factor in the effectiveness of normalization on detecting clones, and this example shows that if the implementation is contrived enough, it could evade clone detection. However, if programmers can write code using as consistently a style as possible, code clone detection will be more accurate.

When the input is split into separate files for each function, JPlag saw a four times clone mass increase. The clones found are similar to Simian's analysis. JPlag reported an exact match in Listing 5.42 and Listing 5.43.

Listing 5.42: Perfect match by JPlag

```
void InterpreteFlag7(char *a) {
    int b;
    char c[255 + 1];
    uint8_t d;
    d = *a;
    a += sizeof(d);
    for (b = 0; b < d; b++) {
        a++;
        c[b] = *a;
    }
    c[b] = '\0';
    printf("Client with handle %s does
        not exist\n", c);
}
```

Listing 5.43: Perfect match by JPlag

```
void InterpreteFlag4(char *a) {
    int b;
    char c[255 + 1];
    uint8_t d;
    d = *a;
    a += sizeof(d);
    for (b = 0; b < d; b++) {
        a++;
        c[b] = *a;
    }
    c[b] = '\0';
    printf("\n%s: %s\n", c, a);
}
```

JPlag does not consider the arguments to `printf` as different, therefore marking the entire function as a clone to each other. JPlag also identified the other two clone instances in `InterpreteFlag5` and `InterpreteFlag12` just like Simian. The rest of the differences between *split* and *normalized* is attributed to splitting the source file into a file per function for JPlag. The summary of JPlag on Student B is shown in Figure 5.6.

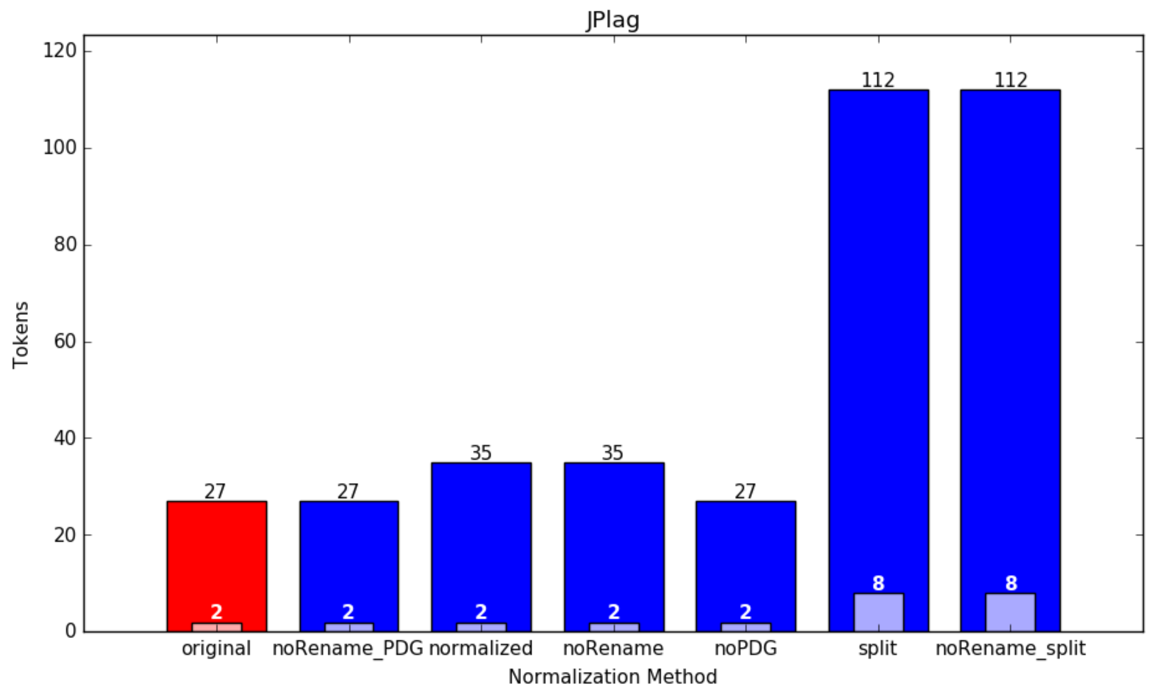


**Figure 5.6: Summary performance of JPlag on Student B**

### 5.2.3 CloneDR

CloneDR has mixed results from using normalized code. CloneDR reports a clone with four instances of the clone across the source files in *original*. All four instances of the clone found in *original* are shown in Listing 5.44, Listing 5.45, Listing 5.46, and Listing 5.47.

Listing 5.44: InterpreteFlag4 in *original*

```
void InterpreteFlag4(char *data_buf) {
   uint8_t handle_len;
   char handle[MAX_HANDLE + 1];
   int i;
   handle_len = *data_buf;
   data_buf += sizeof(handle_len);
   for (i = 0; i < handle_len; i++) {
      handle[i] = *data_buf;
      data_buf++;
   }
   handle[i] = ’\0’;
   printf("\n%s: %s\n", handle,
       data_buf);
}
```

Listing 5.45: InterpreteFlag7 in *original*

```
void InterpreteFlag7(char *data_buf) {
   uint8_t handle_len;
   char handle[MAX_HANDLE + 1];
   int i;
   handle_len = *data_buf;
   data_buf += sizeof(handle_len);
   for (i = 0; i < handle_len; i++) {
      handle[i] = *data_buf;
      data_buf++;
   }
   handle[i] = ’\0’;
   printf("Client with handle %s does
       not exist\n", handle);
}
```

Listing 5.46: InterpreteFlag5 in *original*

```
void InterpreteFlag5(char *data_buf) {
   uint8_t handle_len;
   char handle[MAX_HANDLE + 1];
   int i;
   handle_len = *data_buf;
   data_buf += sizeof(handle_len);
   data_buf += handle_len;
   handle_len = *data_buf;
   data_buf += sizeof(handle_len);
   for (i = 0; i < handle_len; i++) {
      handle[i] = *data_buf;
      data_buf++;
   }
   handle[i] = ’\0’;
   printf("\n%s: %s\n", handle,
       data_buf);
}
```

Listing 5.47: InterpreteFlag12 in *original*

```
void InterpreteFlag12(uint32_t
    handle_numb, char *data_buf) {
   uint8_t handle_len;
   char handle[MAX_HANDLE + 1];
   int i, j;
   for (j = 0; j < handle_numb; j++) {
      handle_len = *data_buf;
      data_buf += sizeof(handle_len);
      for (i = 0; i < handle_len; i++)
          {
         handle[i] = *data_buf;
         data_buf++;
      }
      handle[i] = ’\0’;
      printf("\t%s\n", handle);
   }
}
```

Simian found the same clone in Listing 5.32 and JPlag also successfully detected this clone as well. CloneDR found the same clones in *noRename*, however when performing both statement reordering and identifier renaming in *normalized*, CloneDR does not catch all four instances, instead catching only two instances of a larger clone. The clone from *normalized* is shown in Listing 5.48 and Listing 5.49 and the missed clones are shown in Listing 5.50 and Listing 5.51.

**Listing 5.48: InterpreteFlag4 in *normalized***

```
void InterpreteFlag4(char *a) {
    int b;
    char c[255 + 1];
    uint8_t d;
    d = *a;
    a += sizeof(d);
    for (b = 0; b < d; b++) {
        a++;
        c[b] = *a;
    }
    c[b] = '\0';
    printf("\n%s: %s\n", c, a);
}
```

**Listing 5.49: InterpreteFlag7 in *normalized***

```
void InterpreteFlag7(char *a) {
    int b;
    char c[255 + 1];
    uint8_t d;
    d = *a;
    a += sizeof(d);
    for (b = 0; b < d; b++) {
        a++;
        c[b] = *a;
    }
    c[b] = '\0';
    printf("Client with handle %s does
        not exist\n", c);
}
```

**Listing 5.50: InterpreteFlag5 in *normalized***

```
void InterpreteFlag5(char *a) {
   int b;
   char c[255 + 1];
   uint8_t d;
   d = *a;
   a += sizeof(d);
   a += d;
   d = *a;
   a += sizeof(d);
   for (b = 0; b < d; b++) {
      a++;
      c[b] = *a;
   }
   c[b] = '\0';
   printf("\n%s: %s\n", c, a);
}
```

**Listing 5.51: InterpreteFlag12 in *normalized***

```
void InterpreteFlag12(uint32_t a, char
    *b) {
   int c, d;
   char e[255 + 1];
   uint8_t f;
   for (d = 0; d < a; d++) {
      f = *b;
      b += sizeof(f);
      for (c = 0; c < f; c++) {
         b++;
         e[c] = *b;
      }
      e[c] = '\0';
      printf("\t%s\n", e);
   }
}
```

Compared with *noRename*, using *normalized* yields a lower net clone mass. The summary of normalization on CloneDR for Student B is shown in Figure 5.7.

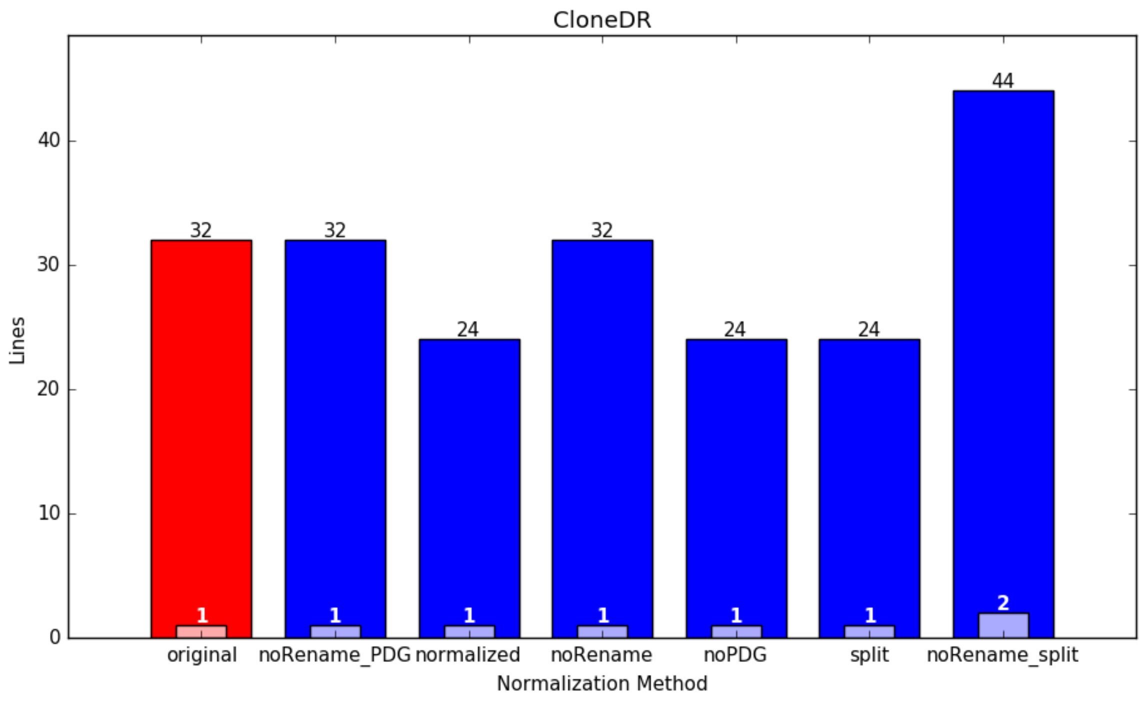


**Figure 5.7: Summary performance of CloneDR on Student B**

There is an increase from *original* to *noRename_split*. CloneDR reported exactly one more clone in *noRename_split*, which is shown in Listing 5.52 and Listing 5.53.

**Listing 5.52: Discovered clone in *noRename_split***

```
void AddChatHeader(char *data_buf, int
    message_len, int header_len, uint8_t
     flag) {
  uint16_t pack_len, pack_len_net;

  char *temp = data_buf;
  pack_len = message_len + header_len;
  pack_len_net = htons(pack_len);
  memcpy(temp, &pack_len_net, sizeof(
      pack_len));
  temp += sizeof(pack_len);
  memcpy(temp, &flag, sizeof(flag));

}
```

**Listing 5.53: Discovered clone in *noRename_split***

```
void EndConnection(char *data_buf, int
    src_fd, node ** cli_sk_head, int *
    cli_sk_count) {
  uint16_t pack_len, pack_len_net;
  uint8_t flag = 9;

  char *temp = data_buf;
  pack_len = sizeof(pack_len) + sizeof
      (flag);
  pack_len_net = htons(pack_len);
  memcpy(temp, &pack_len_net, sizeof(
      pack_len));
  temp += sizeof(pack_len);
  memcpy(temp, &flag, sizeof(flag));

  if (send(src_fd, data_buf, pack_len,
       0) < 0) {
    perror("send call");
    exit(-1);
  }
  if (close(src_fd) < 0) {
    perror("close call");
    exit(-1);
  }
  removeNode(cli_sk_head, src_fd);
}
```

Comparing both versions *noRename* and *noRename_split*, the functions in both versions are identical, yet this clone is not identified in *noRename*. The reason for this discrepency could be that when split, the clone occupies a higher percentage of the source file, resulting in marking it as a clone. Overall, normalization has a neutral effect on Student B with CloneDR.

### 5.2.4 Moss

Moss performed poorer using normalized code than non-normalized code. Some clones shrank in size while other clones were not detected due to statement reordering. A clone that was caught in the control *noRename_PDG_split* but not in the experimental *noRename_split* is shown in Listing 5.54 and Listing 5.55.

**Listing 5.54: Clone in *noRename_PDG_split* but not in *noRename_split***

```
uint8_t flag = 7;
uint8_t handle_len = strlen(handle);
pack_len = 3 + sizeof(handle_len) +
    handle_len;

pack_len_net = htons(pack_len);
memcpy(temp, &pack_len_net, sizeof(
    pack_len));
temp += sizeof(pack_len);
memcpy(temp, &flag, sizeof(flag));
temp += sizeof(flag);
memcpy(temp, &handle_len, sizeof(
    handle_len));

memcpy(temp, &handle_len, sizeof(
    handle_len));
temp += sizeof(handle_len);
memcpy(temp, handle, handle_len);
return pack_len;
```

**Listing 5.55: Clone in *noRename_PDG_split* but not in *noRename_split***

```
node *nodeCur = cli_sk_head;
pack_len = sizeof(pack_len) + sizeof(
    flag) + sizeof(handle_numb);
flag = 11;

pack_len_net = htons(pack_len);
memcpy(temp, &pack_len_net, sizeof(
    pack_len));
temp += sizeof(pack_len);
memcpy(temp, &flag, sizeof(flag));
temp += sizeof(flag);
memcpy(temp, &handle_numb, sizeof(
    handle_numb));
```

When the source code is normalized by statement reordering, the clone is no longer detected. The reordered code is shown in Listing 5.56 and Listing 5.57.

Listing 5.56: Lost clone from Listing 5.54

```
1  uint8_t flag = 7;
2  char *temp = data_buf;
3  uint8_t handle_len = strlen(handle);
4  pack_len = 3 + sizeof(handle_len) +
       handle_len;
5  pack_len_net = htons(pack_len);
6  memcpy(temp, &pack_len_net, sizeof(
       pack_len));
7  temp += sizeof(pack_len);
8  memcpy(temp, &flag, sizeof(flag));
9  temp += sizeof(flag);
10 memcpy(temp, &handle_len, sizeof(
       handle_len));
11 temp += sizeof(handle_len);
12 memcpy(temp, handle, handle_len);
13 return pack_len;
```

Listing 5.57: Lost clone from Listing 5.55

```
1  node *nodeCur = cli_sk_head;
2  pack_len = sizeof(pack_len) + sizeof(
       flag) + sizeof(handle_numb);
3  flag = 11;
4  pack_len_net = htons(pack_len);
5  memcpy(temp, &pack_len_net, sizeof(
       pack_len));
6  temp += sizeof(pack_len);
7  memcpy(temp, &flag, sizeof(flag));
8  pack_len = 0;
9  temp += sizeof(flag);
10 memcpy(temp, &handle_numb, sizeof(
       handle_numb));
```

The difference is the insertion of `pack_len = 0` into line 8 of the clone in Listing 5.57. *Normalizer* found freedom to move `pack_len = 0` upwards and placed it indiscriminately in the middle of a clone even though `pack_len` would not be accessed for a while. Since the other instance does not have a corresponding variable assignment, the clone is not detected. Through reordering, nine clones out of 21 marked by Moss had been lost. Other clones in *noRename_split* have diminished in size. To demonstrate, first the original clone example is given in Listing 5.58 and Listing 5.59.

Listing 5.58: Original sized clone in *noRename_PDG_split*

```
void InterpreteFlag12(uint32_t
    handle_numb, char *data_buf) {
  uint8_t handle_len;
  char handle[255 + 1];
  int i, j;
  for (j = 0; j < handle_numb; j++) {
    handle_len = *data_buf;
    data_buf += sizeof(handle_len);
    for (i = 0; i < handle_len; i++)
        {
      handle[i] = *data_buf;
      data_buf++;
    }
    handle[i] = '\0';
    printf("\t%s\n", handle);
  }
}
```

Listing 5.59: Original sized clone in *noRename_PDG_split*

```
void DirectMessage(char *data_buf, node
    * cli_sk_head, int pack_len) {
  char *temp = data_buf;
  char handle_dest[255];
  char handle_src[255];
  uint8_t handleDest_len,
      handleSrc_len;
  int i, dest_fd, src_fd;
  temp += 3;
  handleDest_len = *temp;
  temp += sizeof(handleDest_len);
  for (i = 0; i < handleDest_len; i++)
      {
    handle_dest[i] = *temp;
    temp++;
  }
  handle_dest[i] = '\0';
  handleSrc_len = *temp;
  temp += sizeof(handleSrc_len);
```

For an unknown reason, Moss removes two lines from each clone when the source code is reordered in *noRename_split*. The smaller code clone is shown in Listing 5.60 and Listing 5.61.

**Listing 5.60: Smaller clone in *noRename_split* from Listing 5.58**

```c
void InterpreteFlag12(uint32_t
    handle_numb, char *data_buf) {
   int i, j;
   char handle[255 + 1];
   uint8_t handle_len;
   for (j = 0; j < handle_numb; j++) {
      handle_len = *data_buf;

      data_buf += sizeof(handle_len);
      for (i = 0; i < handle_len; i++)
          {
         data_buf++;
         handle[i] = *data_buf;
      }
      handle[i] = '\0';

      printf("\t%s\n", handle);
   }
}
```

**Listing 5.61: Smaller clone in *noRename_split* from Listing 5.59**

```c
void DirectMessage(char *data_buf, node
     * cli_sk_head, int pack_len) {
   int i, dest_fd, src_fd;
   char handle_dest[255];
   char handle_src[255];
   uint8_t handleDest_len,
       handleSrc_len;
   char *temp = data_buf;
   temp += 3;
   handleDest_len = *temp;

   temp += sizeof(handleDest_len);
   for (i = 0; i < handleDest_len; i++)
       {
      temp++;
      handle_dest[i] = *temp;
   }
   handle_dest[i] = '\0';

   handleSrc_len = *temp;
   temp += sizeof(handleSrc_len);
```

Line 6 and Line 13 from Listing 5.58 and Line 8 and Line 15 from Listing 5.59 are not included in the clone, but when the function is compared textually with *noRename_PDG_split*, the logic is the same. The only difference is the reordering of the variable declarations which is not part of the clone in the first place and the reorder of the `for` loop body statements which is consistently reordered in both clones. This issue occurs for six other clones where lines are missing from clones for no apparent reason.

The summary of Moss on Student B is shown in Figure 5.8.

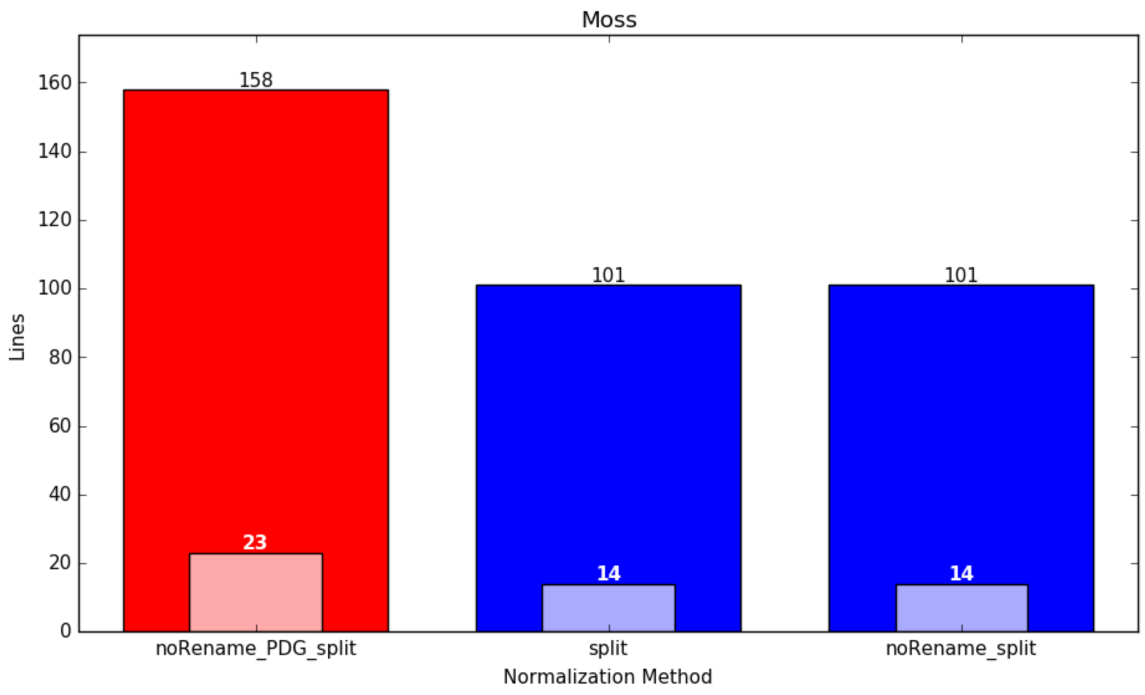


**Figure 5.8: Summary performance of Moss on Student B**

Overall, normalization causes Moss to detect fewer and smaller clones.

## 5.3 Student C

Student C's code features some functions with variable declarations in the middle of them. This proves to be a good sample for *Normalizer* to increase the clone detection rate.

### 5.3.1 Simian

Simian did not detect any clones on Student C's original code, but discovered a new one in *noRename*, which is shown in Listing 5.62 and Listing 5.63.

Listing 5.62: Discovered clone in *noRename*

```c
char msg[1001];
char srcName[51];
uint16_t size;
printf("\n");
memcpy(&size, rcvBuff, sizeof(uint16_t));
size = ntohs(size);

int srcLen = *(rcvBuff + sizeof(struct packetHeader));
int msgLen = sizeof(struct packetHeader) + 1 + srcLen;
memcpy(srcName, rcvBuff + sizeof(struct packetHeader) + 1, srcLen);
srcName[srcLen] = '\0';
```

Listing 5.63: Discovered clone in *noRename*

```c
char msg[1001];
char srcName[51];
uint16_t size;
printf("\n");
memcpy(&size, rcvBuff, sizeof(uint16_t));
size = ntohs(size);

int dstLen = *(rcvBuff + sizeof(struct packetHeader));
char *srcHandle = rcvBuff + sizeof(struct packetHeader) + 1 + dstLen + 1;
int srcLen = *(rcvBuff + sizeof(struct packetHeader) + 1 + dstLen);
int msgLen = sizeof(struct packetHeader) + 2 + dstLen + srcLen;
```

The original code before normalization is shown in Listing 5.64 and Listing 5.65.

Listing 5.64: Listing 5.62 in *original*

```c
printf("\n");
uint16_t size;
memcpy(&size, rcvBuff, sizeof(uint16_t));
size = ntohs(size);
int srcLen = *(rcvBuff + sizeof(struct packetHeader));
int msgLen = sizeof(struct packetHeader) + 1 + srcLen;
char msg[1001];
char srcName[51];
memcpy(srcName, rcvBuff + sizeof(struct packetHeader) + 1, srcLen);
srcName[srcLen] = '\0';
```

Listing 5.65: Listing 5.63 in *original*

```c
printf("\n");
uint16_t size;
memcpy(&size, rcvBuff, sizeof(uint16_t));
size = ntohs(size);
int dstLen = *(rcvBuff + sizeof(struct packetHeader));
int srcLen = *(rcvBuff + sizeof(struct packetHeader) + 1 + dstLen);
char *srcHandle = rcvBuff + sizeof(struct packetHeader) + 1 + dstLen + 1;
int msgLen = sizeof(struct packetHeader) + 2 + dstLen + srcLen;
char srcName[51];
char msg[1001];
```

There are similar lines in *original*, but the fragments are not large enough to be considered a clone. Variable declarations are put in the middle of code, but since they can be declared anywhere before first use, they can be aggregated together to possibly form part of a clone. By moving the declarations to the top of the function, Simian is able to detect a clone. The clone however can still be improved. The clones are identical for the first six lines; at line 7 the logic is the same but the variable names are different. Line 7 should be considered a part of the clone. Using identifier renaming in *normalized* creates a larger clone by including line 7. The improved clone is shown in 5.66 and 5.67.

**Listing 5.66: Improved clone by identifier renaming**

```
1  char b[1001];
2  char c[51];
3  uint16_t d;
4  printf("\n");
5  memcpy(&d, a, sizeof(uint16_t));
6  d = ntohs(d);
7  int e = *(a + sizeof(struct
       packetHeader));

8  int f = sizeof(struct packetHeader) + 1
        + e;
9  memcpy(c, a + sizeof(struct
       packetHeader) + 1, e);
10 c[e] = '\0';
```

**Listing 5.67: Improved clone by identifier renaming**

```
1  char b[1001];
2  char c[51];
3  uint16_t d;
4  printf("\n");
5  memcpy(&d, a, sizeof(uint16_t));
6  d = ntohs(d);
7  int e = *(a + sizeof(struct
       packetHeader));

8  char *f = a + sizeof(struct
       packetHeader) + 1 + e + 1;
9  int g = *(a + sizeof(struct
       packetHeader) + 1 + e);
10 int h = sizeof(struct packetHeader) + 2
        + e + g;
```

By assigning the identifier names as first-used first-assigned, `srcLen` and `dstLen` are both renamed to `e`. The statements are now also identical to each other and therefore, added to the clone, overcoming Simian's weakness of unmatched identifier names. This is a case where identifier renaming produces a better detection. The identifier renaming process can be improved to more likely increase other clone sizes too.

This is the only clone found by Simian with *Normalizer*. The summary of Simian on Student C is shown in Figure 5.9. Overall, normalization has a positive effect on clone detection using Simian with Student C.

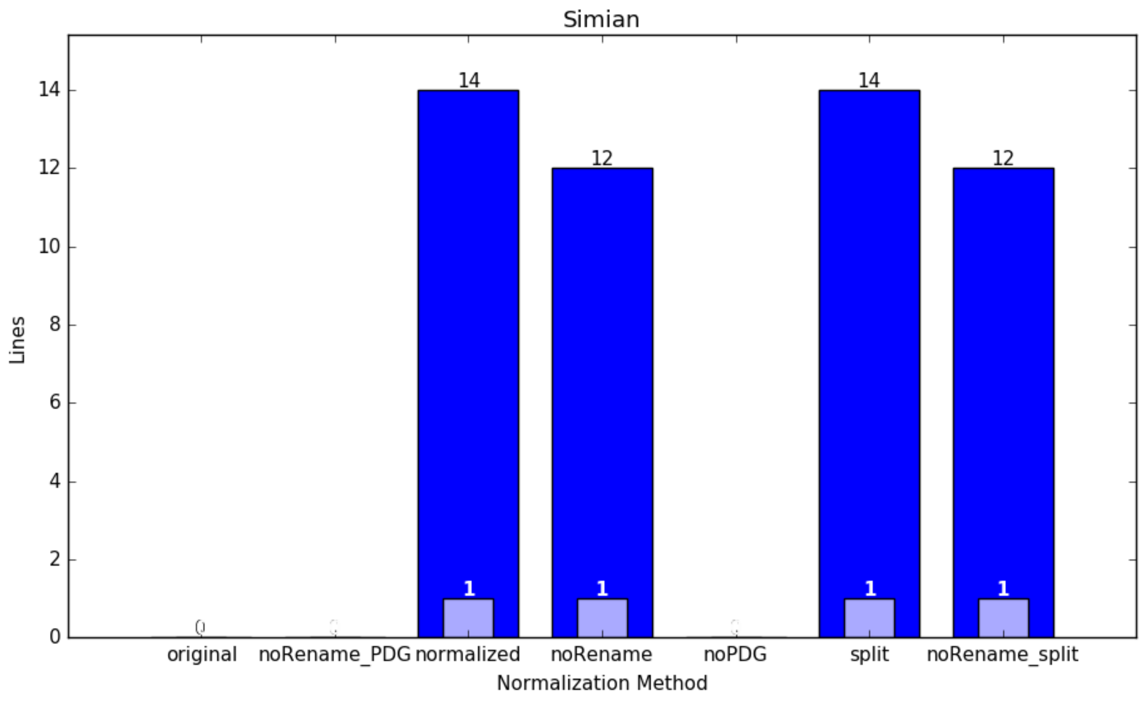


**Figure 5.9: Summary performance of Simian on Student C**

### 5.3.2 JPlag

JPlag reports more clones in *original* than *noRename.* However, a questionable clone found in *original* is shown in Listing 5.68 and Listing 5.69.

**Listing 5.68: Dubious clone in *original***

```
list.curSize = 0;
list.first = calloc(1, sizeof(struct
    handle));
if (!list.first) {
   perror("Calloc Call Err");
   exit(1);
}

list.maxSize = 10;
FD_ZERO(&fds);
FD_SET(servSock, &fds);
while (TRUE) {
   curFds = fds;
   if (select(FD_SETSIZE, &curFds, NULL
       , NULL, NULL) < 0) {
      perror("Select Call Err");
      exit(1);
   }
   if (FD_ISSET(servSock, &curFds)) {
```

**Listing 5.69: Dubious clone in *original***

```
int stopSig = 1, numClients;
fd_set curFds, fds;
FD_ZERO(&fds);
FD_SET(socketNum, &fds);
FD_SET(STDIN_FILENO, &fds);
printf("$: ");
fflush(stdout);
while (stopSig) {
   curFds = fds;
   if (select(FD_SETSIZE, &curFds, NULL
       , NULL, NULL) < 0) {
      perror("Select Error");
      exit(1);
   }
   if (FD_ISSET(socketNum, &curFds)) {
```

The quality of this clone is dubious, since only three statements from Listing 5.68 matches to five statements and two declarations from Listing 5.69. The `while` loop is an acceptable portion of the clone. Upon normalization however, JPlag does not find this as a clone in *noRename*. The code snippet from *noRename* is shown in Listing 5.70 and Listing 5.71.

**Listing 5.70: Listing 5.68 in *noRename***

```
list.curSize = 0;
list.maxSize = 10;
list.first = calloc(1, sizeof(struct
    handle));
if (!list.first) {
   perror("Calloc Call Err");
   exit(1);
}
FD_ZERO(&fds);
FD_SET(servSock, &fds);
while (1) {
   curFds = fds;
   if (mySelect(1024, &curFds, ((void
       *)0), ((void *)0), ((void *)0)) <
        0) {
      perror("Select Call Err");
      exit(1);
   }
   if (FD_ISSET(servSock, &curFds)) {
```

**Listing 5.71: Listing 5.68 in *noRename***

```
fd_set curFds, fds;
int stopSig = 1, numClients;
FD_ZERO(&fds);
FD_SET(socketNum, &fds);
FD_SET(0, &fds);
printf("$: ");
fflush(stdout);
while (stopSig) {
   curFds = fds;
   if (mySelect(1024, &curFds, ((void
       *)0), ((void *)0), ((void *)0)) <
        0) {
      perror("Select Error");
      exit(1);
   }
   if (FD_ISSET(socketNum, &curFds)) {
```

The difference is the movement of `list.maxSize = 10` upwards in Listing 5.70. However, the `while` loop still should have been detected as a part of the clone. *Normalizer* does not know the dependencies of `FD_SET`, `FD_ZERO`, and `printf` so it assumes they cannot be reordered. To human programmers however, we recognize that lines 1 through 7 in Listing 5.68 can be rearranged in a way to preserve dependencies and help expose a larger clone. A logically equivalent code is shown in Listing 5.72 and Listing 5.73.

**Listing 5.72: Manual rearranging of Listing 5.70**

```
list.curSize = 0;
list.maxSize = 10;
list.first = calloc(1, sizeof(struct
    handle));
if (!list.first) {
   perror("Calloc Call Err");
   exit(1);
}
FD_ZERO(&fds);
FD_SET(servSock, &fds);
while (1) {
   curFds = fds;
   if (mySelect(1024, &curFds, ((void
       *)0), ((void *)0), ((void *)0)) <
        0) {
      perror("Select Call Err");
      exit(1);
   }
   if (FD_ISSET(servSock, &curFds)) {
```

**Listing 5.73: Manual rearranging of Listing 5.71**

```
fd_set curFds, fds;
int stopSig = 1, numClients;
printf("$: ");
fflush(stdout);
FD_ZERO(&fds);
FD_SET(socketNum, &fds);
FD_SET(0, &fds);
while (stopSig) {
   curFds = fds;
   if (mySelect(1024, &curFds, ((void
       *)0), ((void *)0), ((void *)0)) <
        0) {
      perror("Select Error");
      exit(1);
   }
   if (FD_ISSET(socketNum, &curFds)) {
```

The `FD_ZERO` and `FD_SET`s are grouped together and are positioned right before the `while` loop. This manual arrangement would flag a human reader's attention as a code clone.

As shown by other students, JPlag greatly detects more clones when the input is split per function. A clone caught in *noRename_split* is shown in Listing 5.74 and Listing 5.75.

**Listing 5.74: Clone found by splitting in *noRename_split***

```
      if (mySend(socketNum, sendBuff,
          1024, 0) < 0) {
         perror("Send Call");
         exit(1);
      }
   }
}
size += length + sizeof(struct
    packetHeader) + 2 + handleLen +
    locHandLen;
header->packLength = htons(size);
memcpy(sendBuff, header, sizeof(struct
    packetHeader));
if (mySend(socketNum, sendBuff, 5 +
    handleLen + locHandLen + length, 0)
    < 0) {
   perror("Send Call");
   exit(1);
}
printf("$: ");
fflush(stdout);
```

**Listing 5.75: Clone found by splitting in *noRename_split***

```
      if (mySend(sockNum, sendBuff,
          1024, 0) < 0) {
         perror("Send Call");
         exit(1);
      }
   }
}
size = length + sizeof(struct
    packetHeader) + 1 + handleLen;
header->packLength = htons(size);
memcpy(sendBuff, header, sizeof(struct
    packetHeader));
if (mySend(sockNum, sendBuff, size, 0)
    < 0) {
   perror("Send Call");
   exit(1);
}
printf("$: ");
fflush(stdout);
```

The clone is an almost exact clone, both of which originally existed in the same file. The summary performance of JPlag on Student C is shown in Figure 5.10.

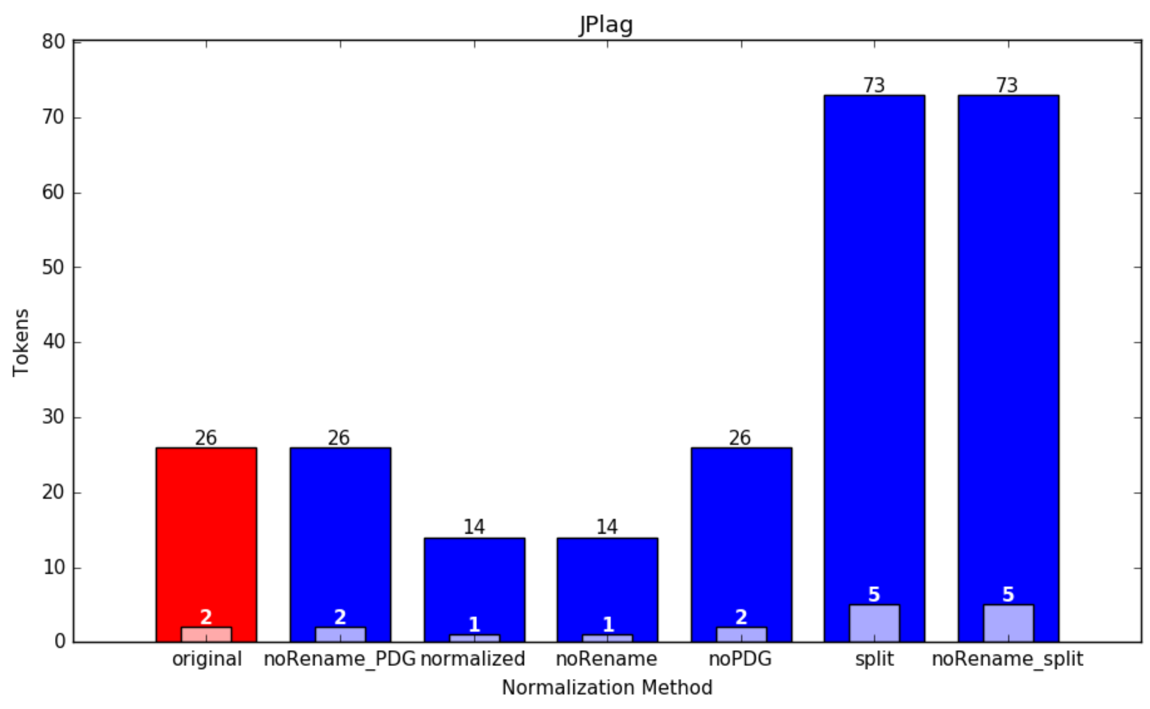


**Figure 5.10: Summary performance of Simian on Student C**

### 5.3.3 CloneDR

CloneDR detects more clones using normalized source code than the original source code. A clone that CloneDR detected in *noRename* but not in *original* is shown in Figure 5.76 and Figure 5.77.

**Listing 5.76: Same clone as Simian in *noRename***

```
char msg[1001];
char srcName[51];
uint16_t size;
printf("\n");
memcpy(&size, rcvBuff, sizeof(uint16_t)
    );
size = ntohs(size);
int srcLen = *(rcvBuff + sizeof(struct
    packetHeader));
```

**Listing 5.77: Same clone as Simian in *noRename***

```
char msg[1001];
char srcName[51];
uint16_t size;
printf("\n");
memcpy(&size, rcvBuff, sizeof(uint16_t)
    );
size = ntohs(size);
int dstLen = *(rcvBuff + sizeof(struct
    packetHeader));
```

This is the same clone as the first Simian clone in Listing 5.66 and Listing 5.67, except with one more line at the bottom. Another clone that CloneDR detected in *noRename* is shown in Listing 5.78 and Listing 5.79.

**Listing 5.78: Another clone in *noRename***

```
void printHandles(int sockNum, char *
    handleName) {
  char sendBuff[1000];

  struct packetHeader temp;
  temp.flag = 10;
  temp.packLength = sizeof(struct
      packetHeader);
  memcpy(sendBuff, &temp, sizeof(
      struct packetHeader));
  if (mySend(sockNum, sendBuff, sizeof
      (struct packetHeader), 0) < 0) {
    perror("Send Error");
    exit(1);
  }

}
```

**Listing 5.79: Another clone in *noRename***

```
void execExitCmd(int socketNum, char *
    handleName) {
  char sendBuff[1000];

  struct packetHeader temp;
  temp.flag = 8;
  temp.packLength = sizeof(struct
      packetHeader);
  memcpy(sendBuff, &temp, sizeof(
      struct packetHeader));
  if (mySend(socketNum, sendBuff,
      sizeof(struct packetHeader), 0) <
       0) {
    perror("Send Call");
    exit(1);
  }

  printf("$: ");
  fflush(stdout);
}
```

This clone is a good clone to parameterize and abstract out. `printHandles` can be parameterized and be called `execExitCmd`. There is a lost clone, however, through normalization. The clone only found in *original* is shown in Listing 5.80 and Listing 5.81.

Listing 5.80: Clone only found in *original*

```
if ((1 + length + handleLen + sizeof(
    struct packetHeader)) == BUFFER_SIZE
    ) {
   header->packLength = 1 + length +
       handleLen + sizeof(struct
       packetHeader);
   header->packLength = htons(header->
       packLength);
   printf("Message is %d bytes, this is
        too long.", size);
    printf("Message truncated to 1000
        bytes.");

   memcpy(sendBuff, header, sizeof(
       struct packetHeader));
   sendBuff[BUFFER_SIZE] = '\0';
   if (send(sockNum, sendBuff,
       BUFFER_SIZE, 0) < 0) {
      perror("Send Call");
      exit(1);
   }
   length = 0;

}
```

Listing 5.81: Clone only found in *original*

```
if (curPackSize + sizeof(struct
    packetHeader) + list->first[i].len
    >= BUFFER_SIZE) {
   sending->packLength = htons(
       curPackSize);

   memcpy(sendBuff, sending, sizeof(
       struct packetHeader));
   sendBuff[curPackSize] = '\0';
   if (send(curClient, sendBuff,
       BUFFER_SIZE, 0) < 0) {
      perror("Send Call");
      exit(1);
   }
   curPackSize = 0;

}
```

When normalized by statement reordering in *noRename*, the clone is lost. *noRename*'s code is shown in Listing 5.82 and Listing 5.83.

**Listing 5.82: Listing 5.80 clone lost in *noRename***

```
if ((1 + length + handleLen + sizeof(
    struct packetHeader)) == 1024) {
  length = 0;
  header->packLength = 1 + length +
      handleLen + sizeof(struct
      packetHeader);
  header->packLength = htons(header->
      packLength);
  printf("Message is %d bytes, this is
       too long.", size);
  printf("Message truncated to 1000
      bytes.");
  memcpy(sendBuff, header, sizeof(
      struct packetHeader));
  sendBuff[1024] = '\0';
  if (mySend(sockNum, sendBuff, 1024,
      0) < 0) {
    perror("Send Call");
    exit(1);
  }
}
```

**Listing 5.83: Listing 5.81 clone lost in *noRename***

```
if (curPackSize + sizeof(struct
    packetHeader) + list->first[i].len
    >= 1024) {
  sending->packLength = htons(
      curPackSize);
  memcpy(sendBuff, sending, sizeof(
      struct packetHeader));
  sendBuff[curPackSize] = '\0';
  curPackSize = 0;
  if (mySend(curClient, sendBuff,
      1024, 0) < 0) {
    perror("Send Call");
    exit(1);
  }
}
```

By reordering the statements, `length = 0` and `curPackSize = 0` are moved upwards. `length = 0` is moved away from the clone, fragmenting the clone. `curPackSize = 0` does not have a corresponding match anymore and is also inserted in the middle of the already existing clone. By fragmenting the clones, they are not big enough to pass the size threshold and are not detected as clones.

Overall however, CloneDR performs better with normalization. A summary of normalization on CloneDR for Student C is shown in Figure 5.11.

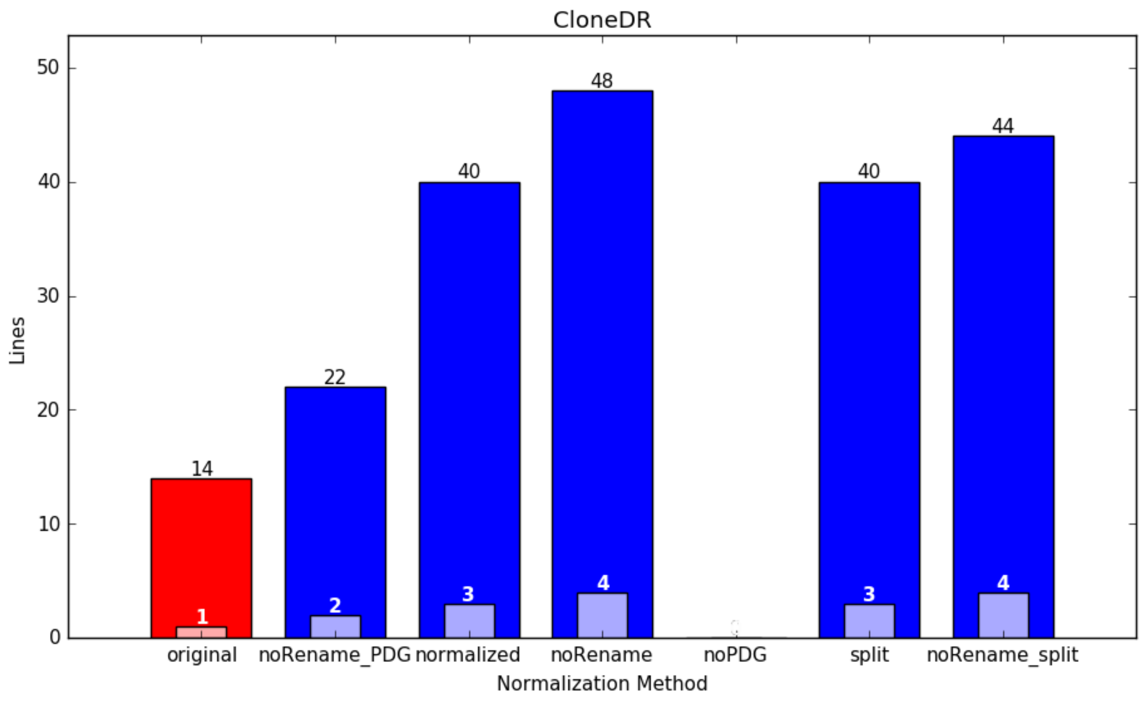


**Figure 5.11: Summary performance of CloneDR on Student C**

### 5.3.4 Moss

Moss performed favorably with code normalization. A clone was detected by using statement reordering in *noRename_split* that was not found in the original. The clone is shown in Listing 5.84 and Listing 5.85.

**Listing 5.84: Clone found by statement reordering in *noRename_split***

```
void execExitCmd(int socketNum, char *
    handleName) {
   char sendBuff[1000];
   struct packetHeader temp;
   temp.flag = 8;
   temp.packLength = sizeof(struct
       packetHeader);
   memcpy(sendBuff, &temp, sizeof(
       struct packetHeader));
   if (mySend(socketNum, sendBuff,
       sizeof(struct packetHeader), 0) <
        0) {
      perror("Send Call");
      exit(1);
   }
   printf("$: ");
   fflush(stdout);
}
```

**Listing 5.85: Clone found by statement reordering in *noRename_split***

```
void printHandles(int sockNum, char *
    handleName) {
   char sendBuff[1000];
   struct packetHeader temp;
   temp.flag = 10;
   temp.packLength = sizeof(struct
       packetHeader);
   memcpy(sendBuff, &temp, sizeof(
       struct packetHeader));
   if (mySend(sockNum, sendBuff, sizeof
       (struct packetHeader), 0) < 0) {
      perror("Send Error");
      exit(1);
   }
}
```

This is the same clone that CloneDR caught in Listing 5.78 and Listing 5.79.

Normalizing the source code provided some improvements over clones found without normalized code. An interesting case that Moss detected is shown in Listing 5.86 and Listing 5.87. There are two clones, one clone is highlighted in yellow and the other in orange.

**Listing 5.86: 2 clones found in *noRename***

```
void processBroadCast(char *rcvBuff) {
   char msg[1001];
   char srcName[51];
   uint16_t size;
   printf("\n");
   memcpy(&size, rcvBuff, sizeof(
       uint16_t));
   size = ntohs(size);
   int srcLen = *(rcvBuff + sizeof(
       struct packetHeader));

   int msgLen = sizeof(struct
       packetHeader) + 1 + srcLen;
   memcpy(srcName, rcvBuff + sizeof(
       struct packetHeader) + 1, srcLen)
       ;

   srcName[srcLen] = '\0';
   memcpy(msg, rcvBuff + msgLen, size -
        msgLen);
   msg[size - msgLen] = '\0';
   printf("%s:%s\n", srcName, msg);
}
```

**Listing 5.87: 2 clones found in *noRename***

```
void processMsg(char *rcvBuff) {
   char msg[1001];
   char srcName[51];
   uint16_t size;
   printf("\n");
   memcpy(&size, rcvBuff, sizeof(
       uint16_t));
   size = ntohs(size);
   int dstLen = *(rcvBuff + sizeof(
       struct packetHeader));

   char *srcHandle = rcvBuff + sizeof(
       struct packetHeader) + 1 + dstLen
        + 1;
   int srcLen = *(rcvBuff + sizeof(
       struct packetHeader) + 1 + dstLen
       );
   int msgLen = sizeof(struct
       packetHeader) + 2 + dstLen +
       srcLen;
   memcpy(srcName, srcHandle, srcLen);

   srcName[srcLen] = '\0';
   memcpy(msg, rcvBuff + msgLen, size -
        msgLen);
   msg[size - msgLen] = '\0';
   printf("%s:%s\n", srcName, msg);
}
```

The non-normalized version featuring a smaller clone in *noRename_PDG_split* is shown in Listing 5.88 and Listing 5.89.

Listing 5.88: Listing 5.86 in *noRename_PDG_split*

```
void processBroadCast(char *rcvBuff) {
   printf("\n");
   uint16_t size;
   memcpy(&size, rcvBuff, sizeof(uint16_t));
   size = ntohs(size);
   int srcLen = *(rcvBuff + sizeof(struct packetHeader));
   int msgLen = sizeof(struct packetHeader) + 1 + srcLen;

   char msg[1001];
   char srcName[51];
   memcpy(srcName, rcvBuff + sizeof(struct packetHeader) + 1, srcLen);

   srcName[srcLen] = '\0';
   memcpy(msg, rcvBuff + msgLen, size - msgLen);
   msg[size - msgLen] = '\0';
   printf("%s:%s\n", srcName, msg);
}
```

Listing 5.89: Listing 5.87 in *noRename_PDG_split*

```
void processMsg(char *rcvBuff) {
   printf("\n");
   uint16_t size;
   memcpy(&size, rcvBuff, sizeof(uint16_t));
   size = ntohs(size);
   int dstLen = *(rcvBuff + sizeof(struct packetHeader));
   int srcLen = *(rcvBuff + sizeof(struct packetHeader) + 1 + dstLen);

   char *srcHandle = rcvBuff + sizeof(struct packetHeader) + 1 + dstLen + 1;
   int msgLen = sizeof(struct packetHeader) + 2 + dstLen + srcLen;
   char srcName[51];
   char msg[1001];
   memcpy(srcName, srcHandle, srcLen);

   srcName[srcLen] = '\0';
   memcpy(msg, rcvBuff + msgLen, size - msgLen);
   msg[size - msgLen] = '\0';
   printf("%s:%s\n", srcName, msg);
}
```

The difference is the variable declarations of `msg` and `srcName` in the middle of the code in *noRename_PDG_split* for both clone instances. By shifting the variable declarations upwards to the top, `char msg[1001]` and `char srcName[51]` are included in the clone. However the declaration for `msgLen` in Listing 5.86 is no longer included in the clone and the declaration for `srcLen` in Listing 5.87 is also not part of the clone anymore. Overall, there is a net gain of one line from normalizing this clone. Normalization can fragment and augment already existing clones.

A summary of normalization on Moss for Student C is given in Figure 5.12.

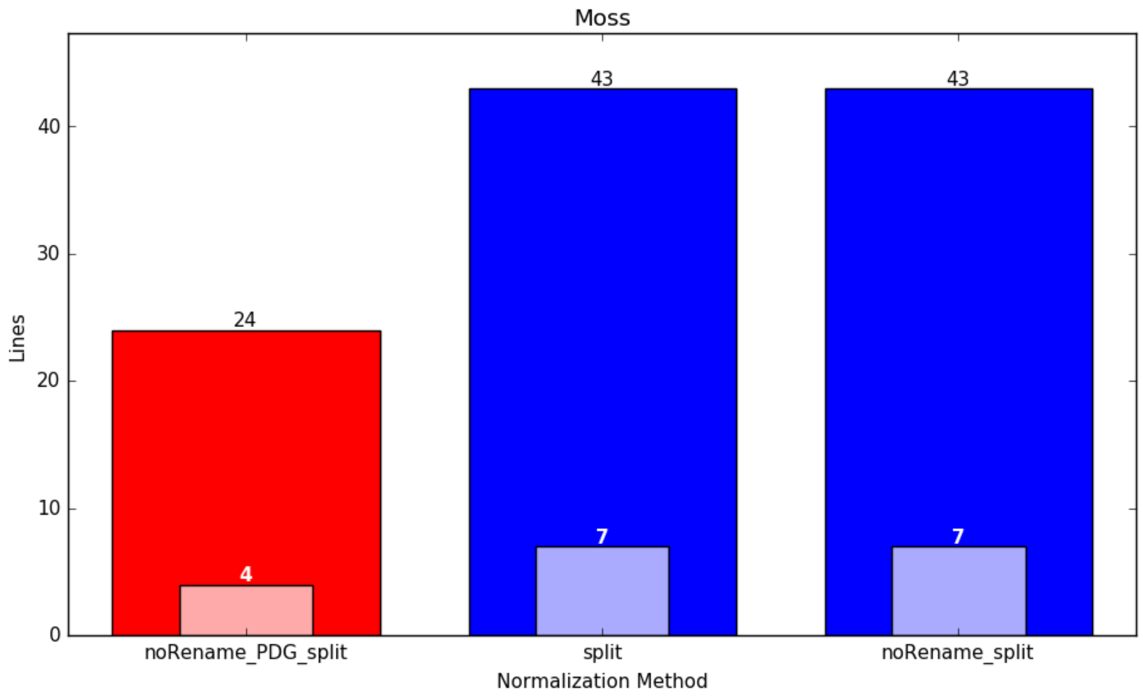


**Figure 5.12: Summary performance of Moss on Student C**

## 5.4 Aggregates

*Normalizer* is used on student submissions from three sections of the Introduction to Computer Networks class. A number of submissions used C++ as their programming language which *Normalizer* does not support so these are excluded. Some students have code that cannot be parsed. The unparseable code is originally from a C standard library struct definition which includes `__attribute__` modifiers and GCC specific syntax. A best effort attempt was made to handle these, but the effort was abandoned due to the small number of affected submissions. The remaining pool of students totaled 49 who chose the C language and had code that was able to be parsed by *Normalizer*. Across 49 students, the total number of clones and cloned lines and tokens per tool

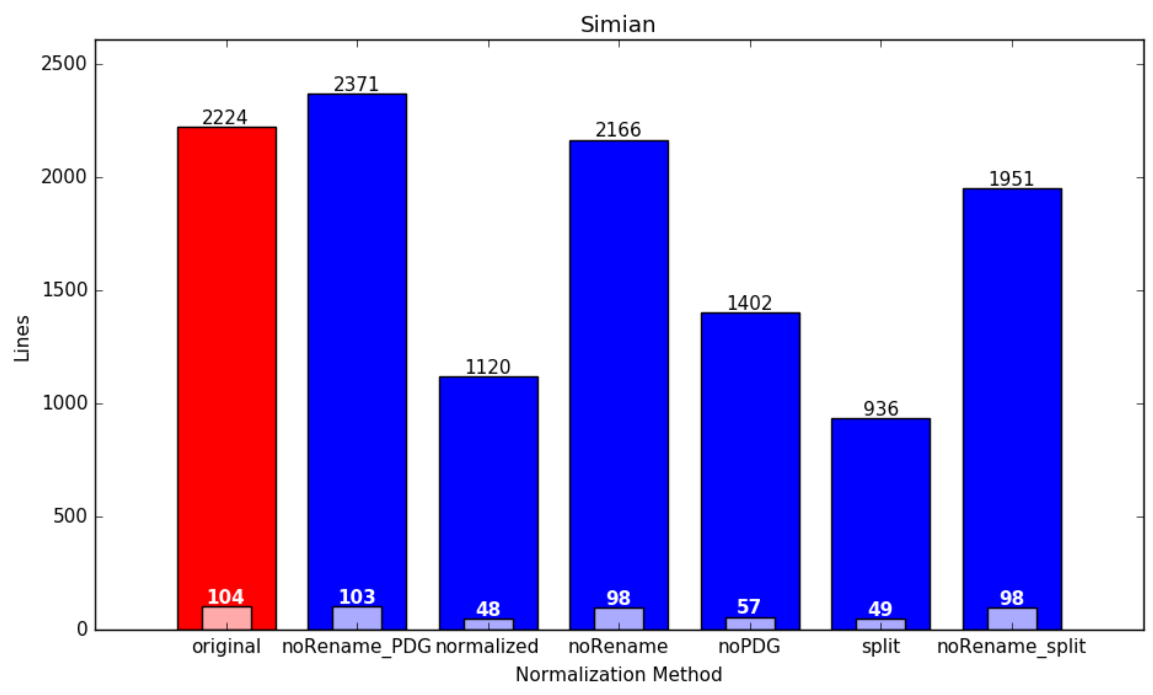


**Figure 5.13: Aggregate results of Simian**

is shown in Figures 5.13, 5.14, 5.15, and 5.16. The number of clones is shown in the narrow white bar and the total clone mass is shown as the tall and wide bar.

In Figure 5.13, Simian using *noRename* does nearly as well as *original*. Surprisingly, *noRename_PDG* does better than *original* when no normalization occurs. All versions that renames identifiers do worse than *original*, as exemplified by Students A, B and C.

Shown in Figure 5.14, JPlag finds many more clones with any *split* version. All versions performed better than *original*, including *noRename_PDG*, which is expected to have little to no difference.

In Figure 5.15, CloneDR's general performance matches with Simian's relative performance per normalization version. Same as Simian and JPlag, *noRename_PDG* has a higher clone mass than *original*.

Finally, Moss in general found fewer clones by normalizing the input shown in

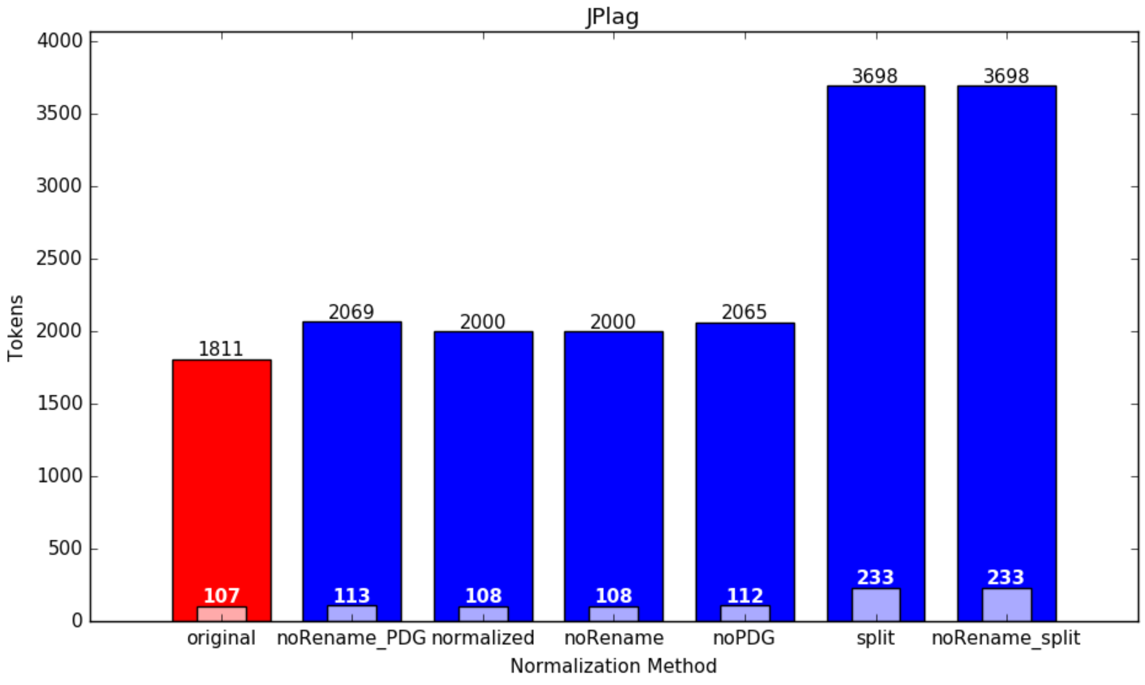


Figure 5.14: Aggregate results of JPlag

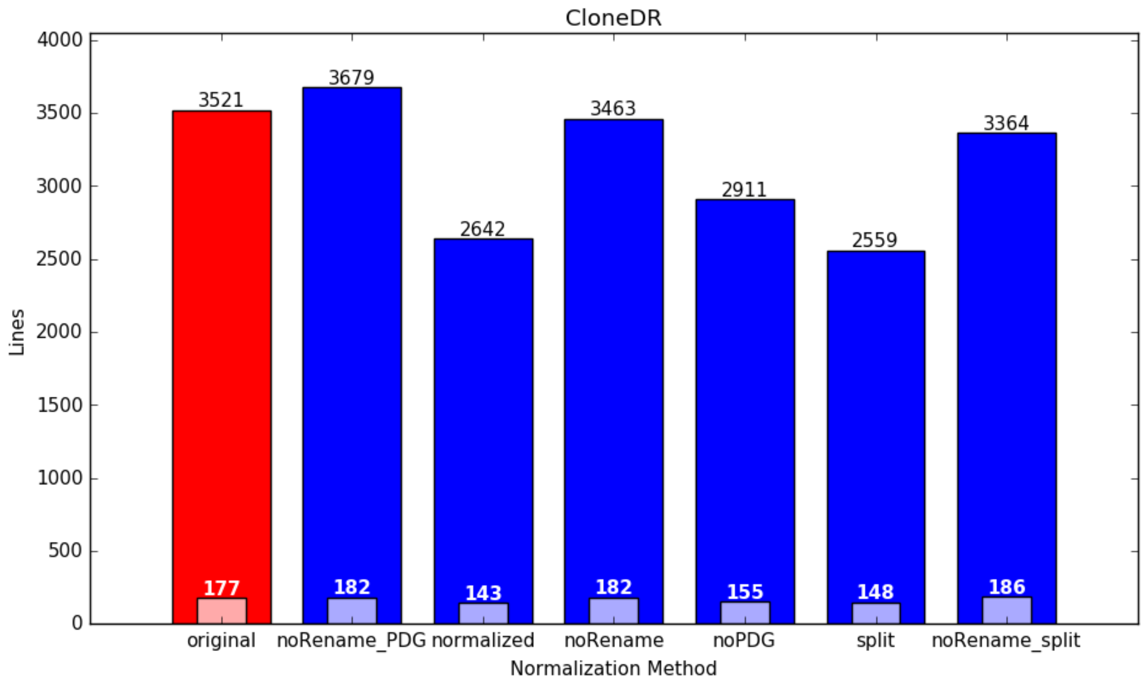


Figure 5.15: Aggregate results of CloneDR

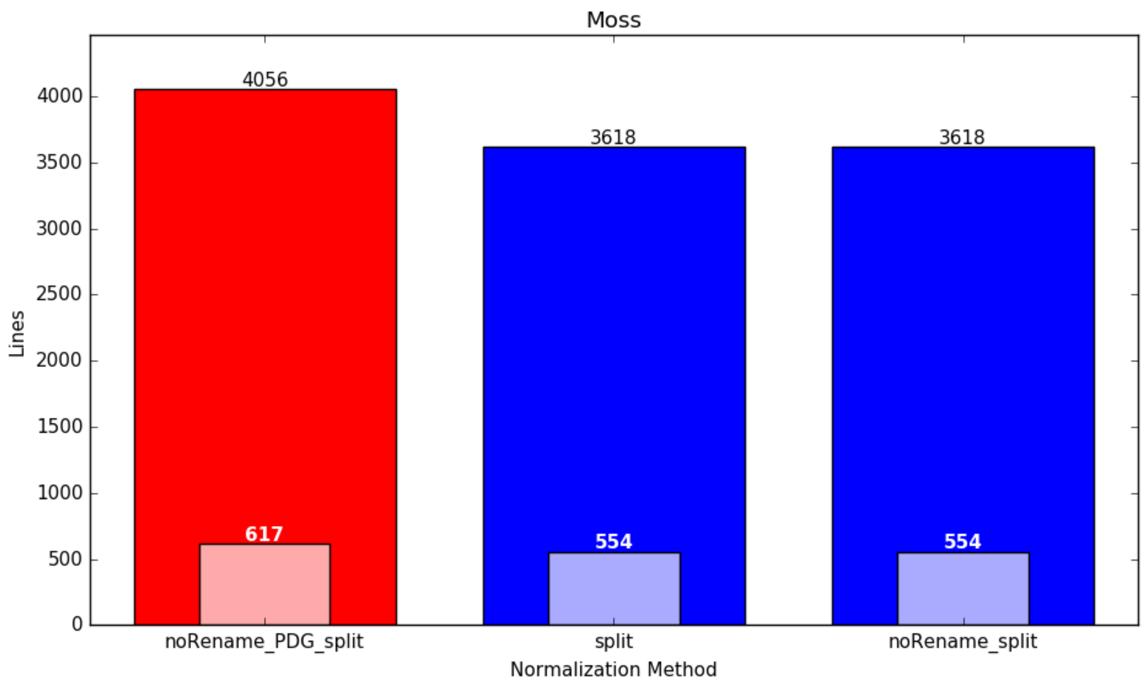


**Figure 5.16: Aggregate results of Moss**

Figure 5.16. Identifier renaming does not affect Moss's clone detection algorithm, but statement reordering will.

Curiously in Simian, JPlag, and CloneDR, *noRename_PDG* shows a larger clone mass than *original* when it is expected to have no difference since no normalization occurred. To investigate, a histogram is generated where each data point is the number of clone lines changed from *original* to *noRename_PDG* per student. The histogram is shown in Figure 5.17.

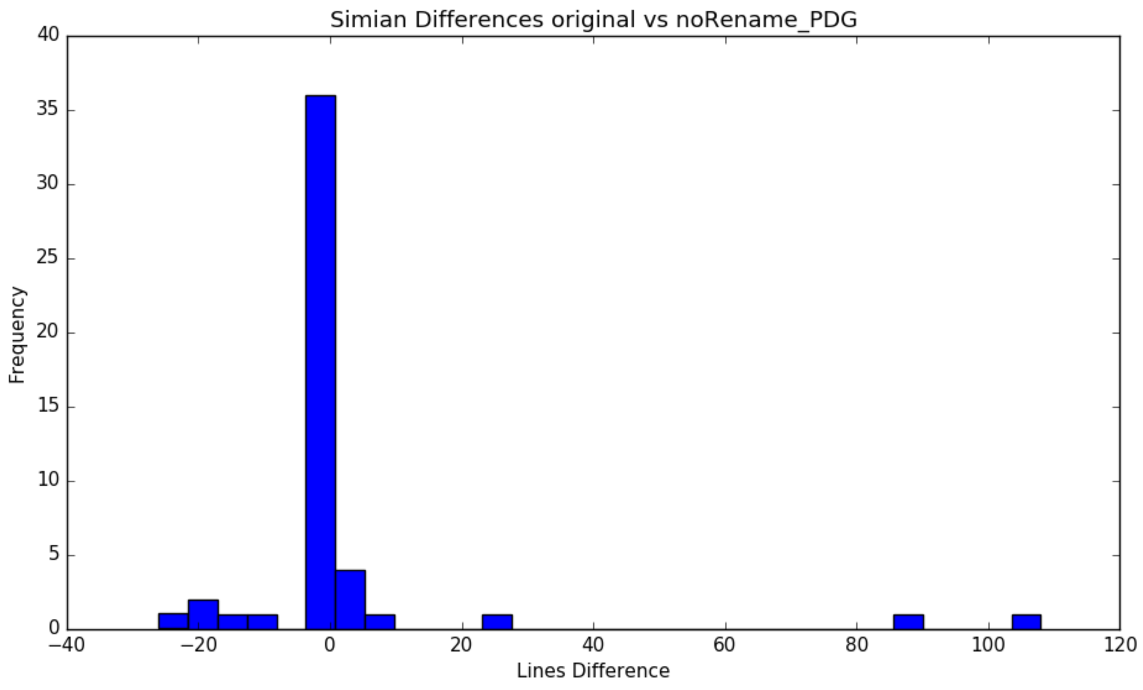


**Figure 5.17: Histogram of *original* to *noRename_PDG***

By running through *noRename_PDG*, two students' submissions saw an increase of 87 and 108 lines of clones. Upon inspecting these two submissions, the students have effectively put code into their C header files, either by having function definitions in their header files or by importing a C source file. By preprocessing the `#include` directives for each file that imports the header file, the code is inserted into each file, creating a code clone. These code clones do not make multiple versions of source code, but it does increase the size of the executable. If these two special case submissions are excluded, then across 47 students the aggregate results are shown in Figure 5.18, 5.19, 5.20, and 5.21.

In Figure 5.18, by removing the two outliers, *noRename_PDG* has dropped to a similar level as *original*. Simian's highest scoring normalized version is *noRename*, but still scored lower than *original*. The outlier correction is also shown for JPlag shown in Figure 5.19.

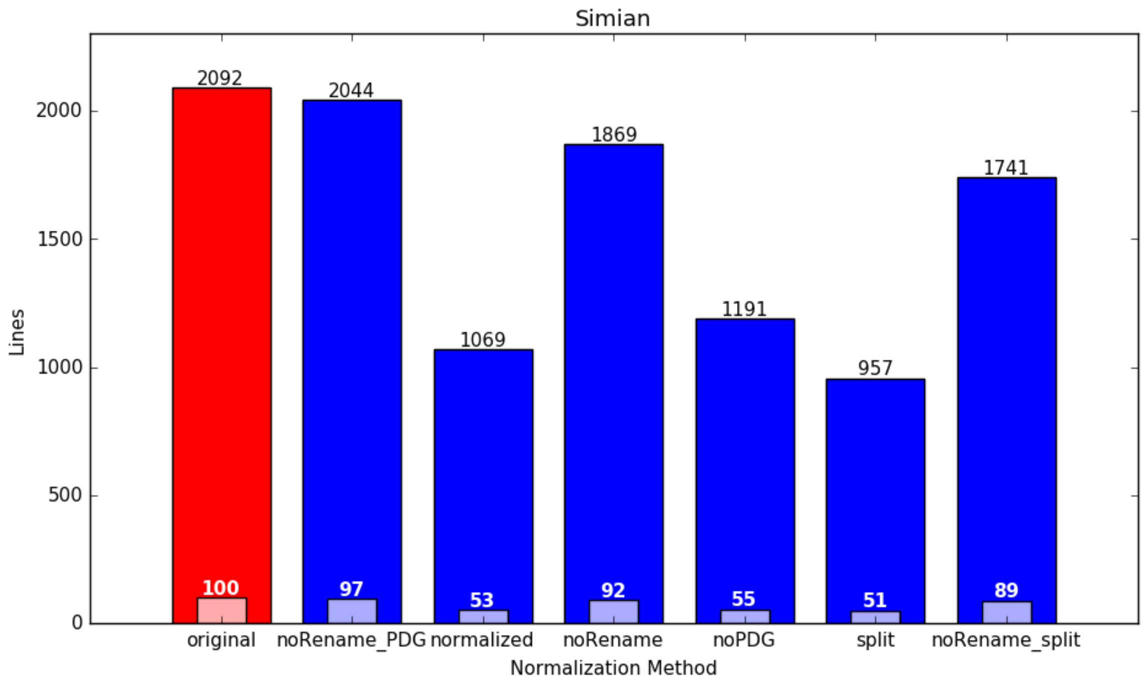


**Figure 5.18: Aggregate results of Simian excluding outliers**

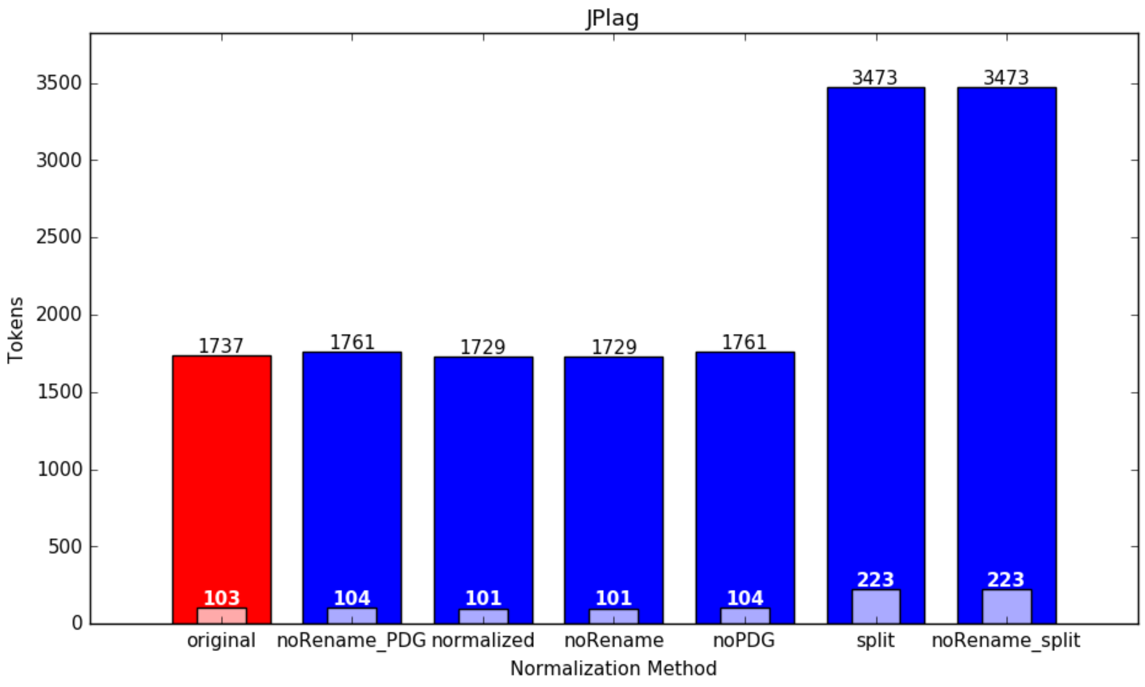


**Figure 5.19: Aggregate results of JPlag excluding outliers**

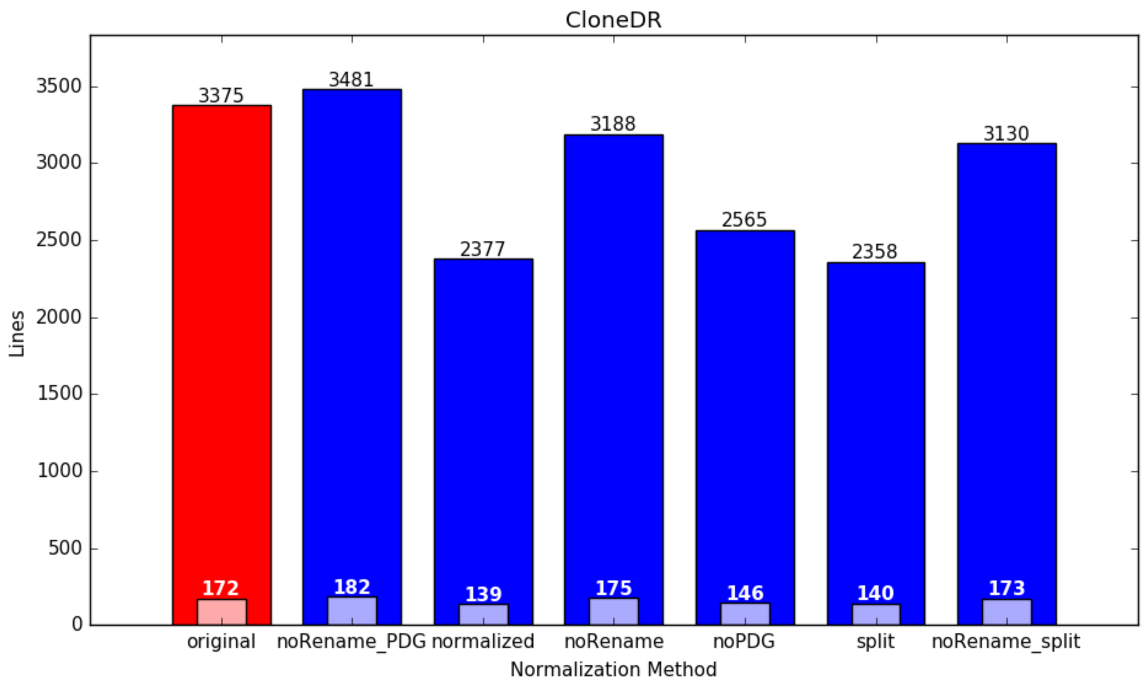


**Figure 5.20: Aggregate results of CloneDR excluding outliers**

In Figure 5.19, JPlag does approximately equally in all non-split versions now that preprocessing does not introduce clones. Also, identifier renaming does not make any impact at all for JPlag. JPlag truly shines in split versions of the source code.

In Figure 5.20, CloneDR's highest normalized version is *noRename*, finding a larger number of clones than *original*, but a total smaller clone mass.

Moss shows the same picture in Figure 5.21, as normalization has an overall negative effect on Moss. The results shows on average a slight decrease in clone masses caught for all normalization methods. Each submission has a different reaction to source code normalization and the aggregates show that normalization on average lowers clone sizes. An exception is JPlag, which shows a substantial increase in clones detected by splitting the input files into functions.

Supplemental box plots are generated as another means to graph aggregate results. These box plots are placed in Appendix A, which show the distribution of each

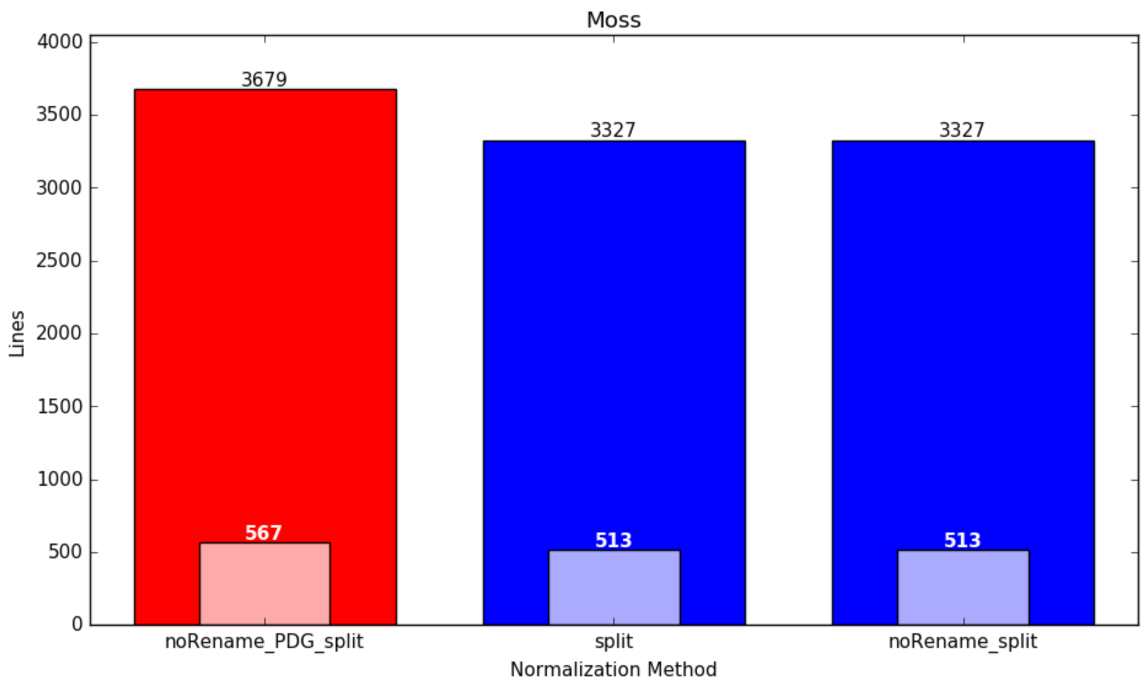


**Figure 5.21: Aggregate results of Moss excluding outliers**

student's detected clone masses per tool and normalization version. These charts are useful for seeing the magnitude of clones written by students.

A closer look into *Normalizer*'s effect can be observed by only comparing between *original* and another normalized version. In the ideal case, *normalized* will show the greatest improvement over *original*. The comparison histogram for Simian from *original* to *normalized* is shown in Figure 5.22.

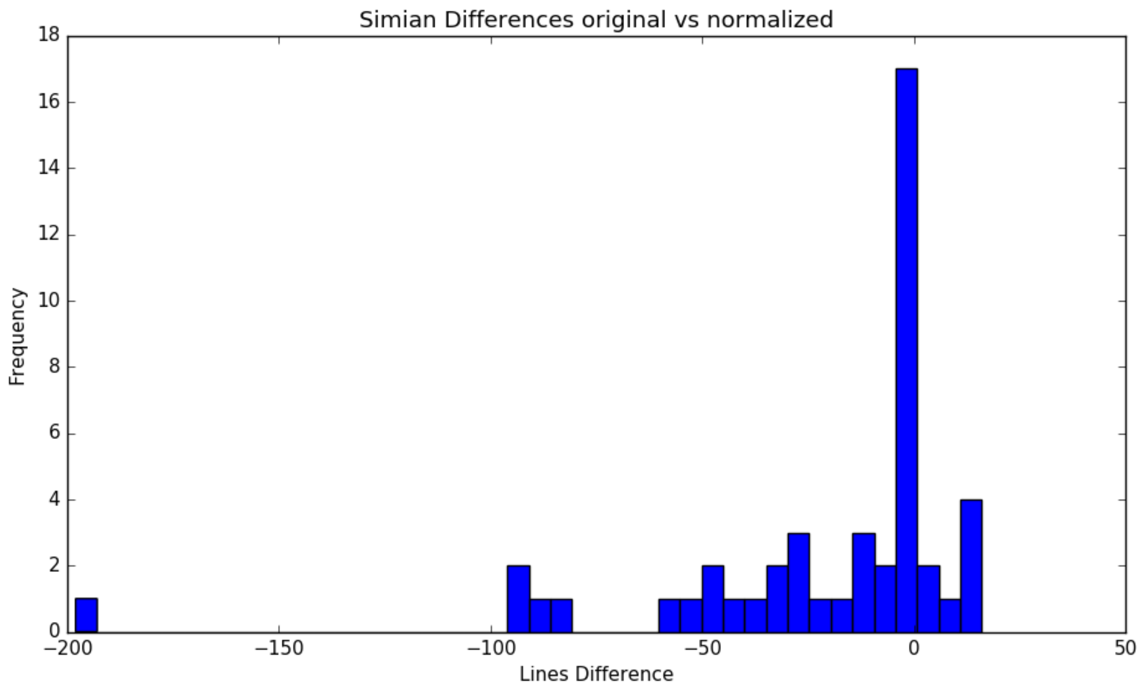


**Figure 5.22: Histogram of *original* to *normalized* using Simian**

There are minor improvements for a few submissions but for many students, their clones become hidden by normalizing. One student even showed dramatically less detected clones when reordered and renamed. As already explored, *noRename* generally performs better than *normalized* for Simian. The comparison histogram from *original* to *noRename* is shown in Figure 5.23.

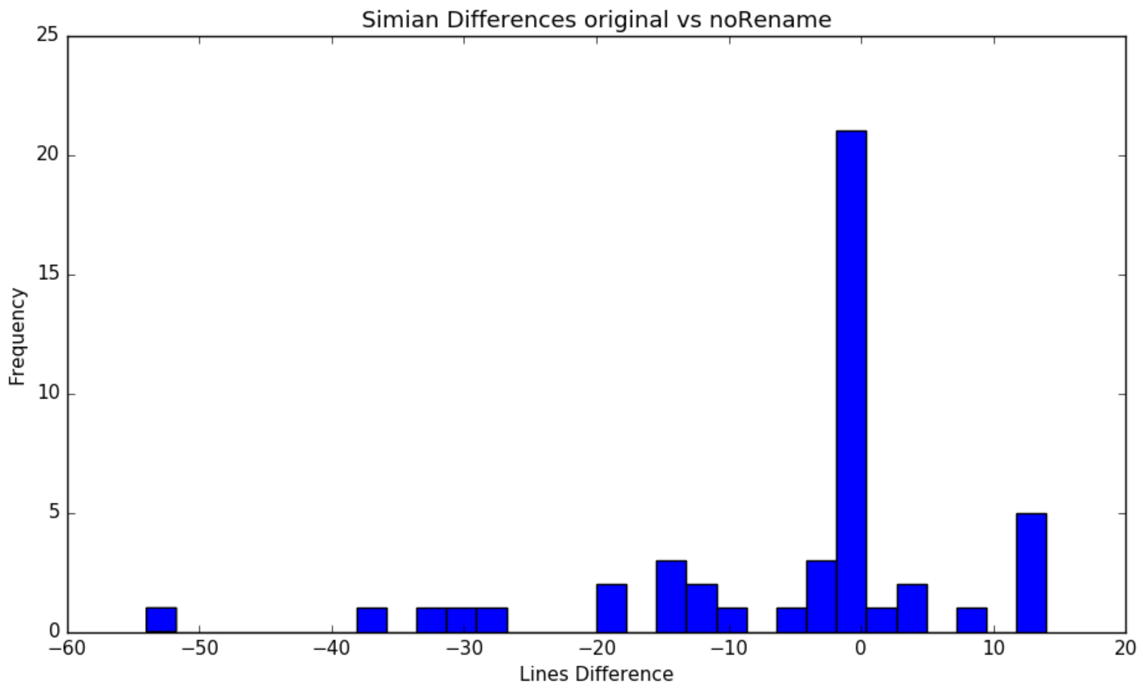


**Figure 5.23: Histogram of *original* to *noRename* using Simian**

Most submissions do not observe any difference with the usage of *Normalizer*. This could be because there are no hidden clones to be discovered. Compared against *normalized*, by omitting identifier renaming, submissions have increased clone masses and the outliers from Figure 5.22 have disappeared. This shows that identifier renaming has a negative effect on Simian. Statement reordering still can cause negative effects and positive effects. As already demonstrated, reordering statements can sometimes break clones apart or move fragments together. JPlag tells the same story; the comparison between *original* and *normalized* is shown in Figure 5.24.

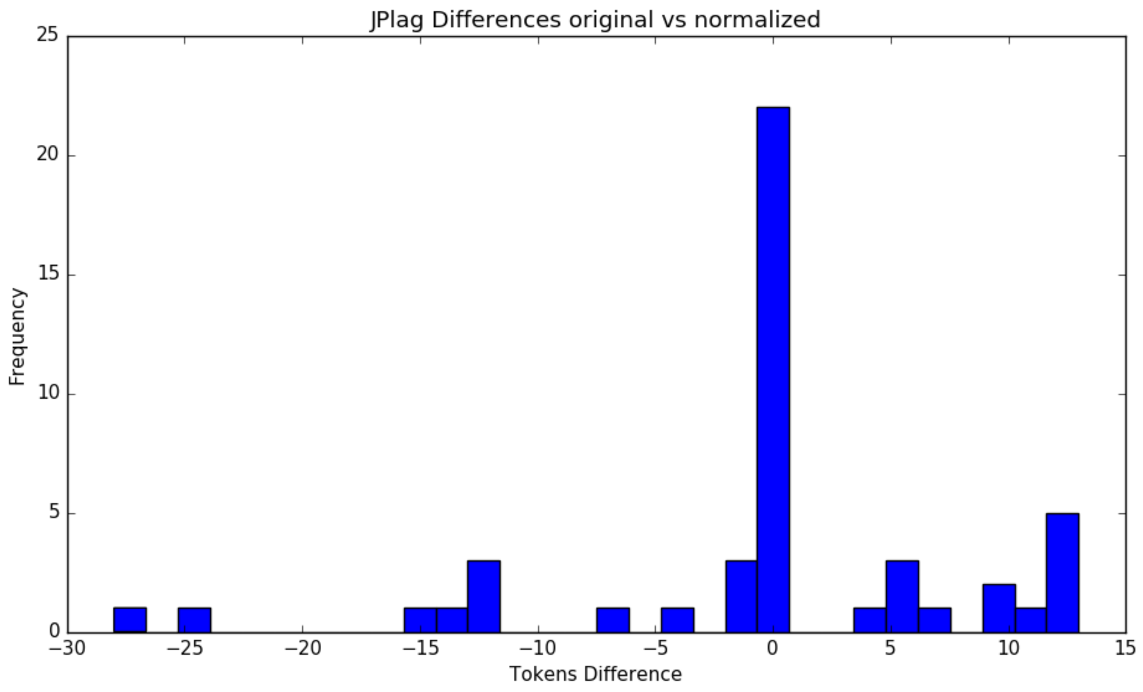


**Figure 5.24: Histogram of *original* to *normalized* using JPlag**

JPlag does a good job disregarding order within clones and identifier names resulting in a generally higher benefit from normalization. Figure 5.24 also does not include splitting the input into a file per function. When comparing *original* to *split*, the histogram is shown in Figure 5.25.

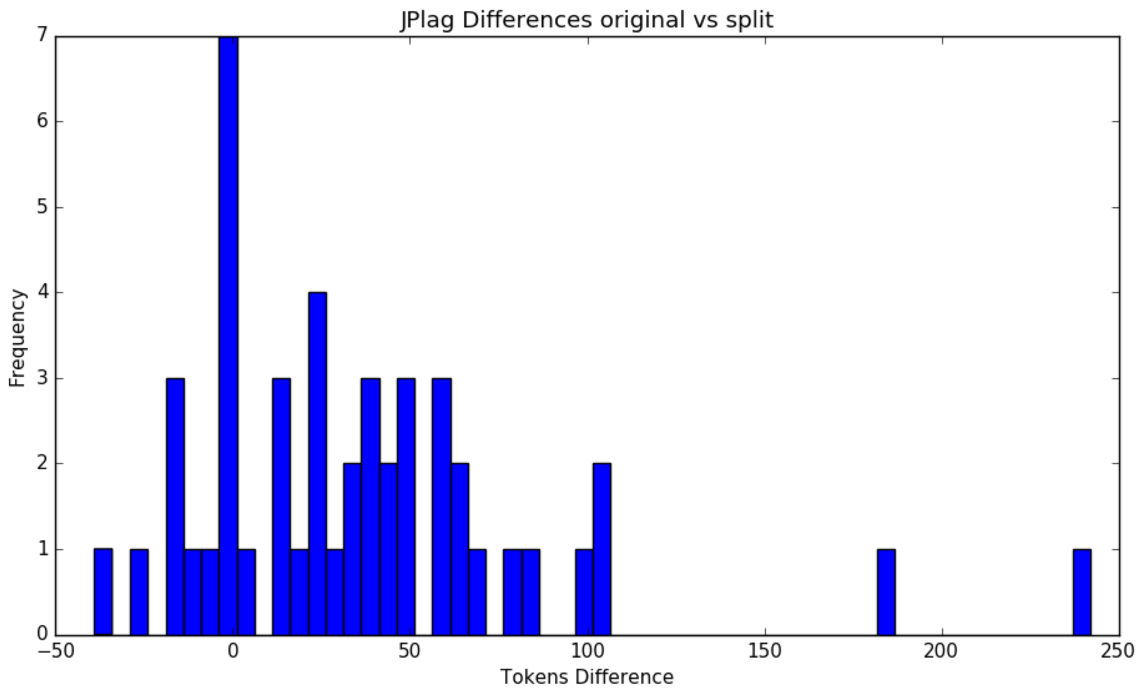


**Figure 5.25: Histogram of *original* to *split* using JPlag**

By splitting the input file, a majority of students saw a benefit from using *Normalizer*. For the case of two students, large clones exist within the same file. CloneDR shares the same picture as JPlag without splitting. The histogram of CloneDR comparing *original* and *noRename* is shown in Figure 5.26.

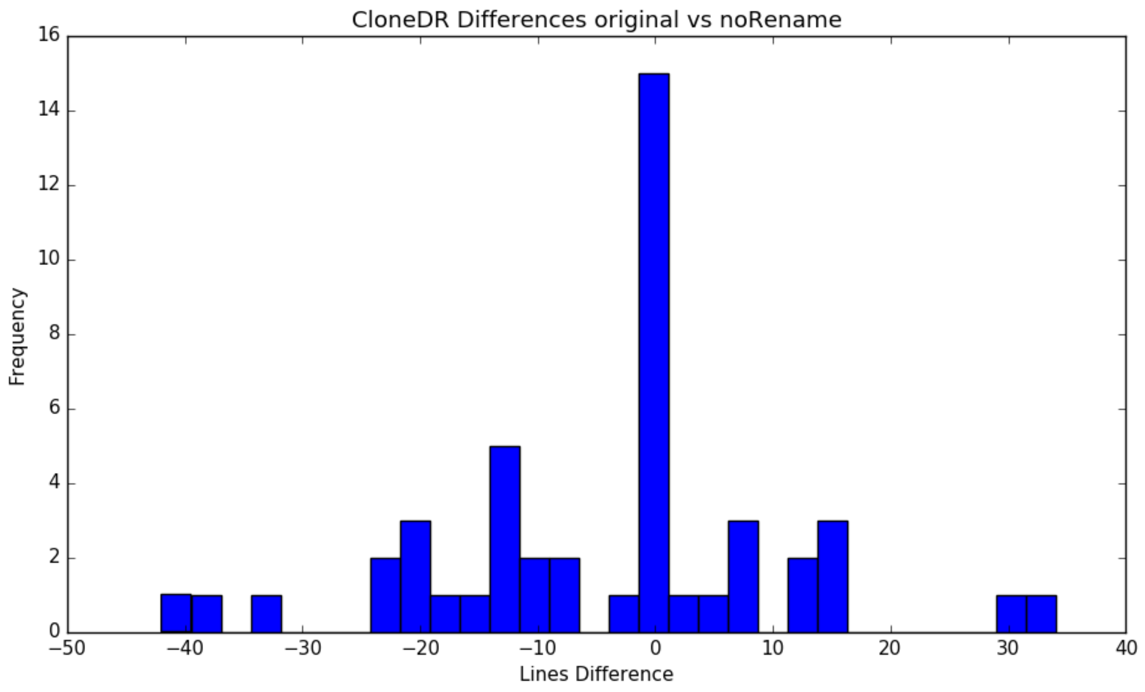


**Figure 5.26: Histogram of *original* to *noRename* using CloneDR**

Source code normalization provides more extreme results for CloneDR than JPlag, giving higher gains and higher losses. However, it is important to see that gains are still to be had which normalization provides.

Lastly, even though Moss on the average lowers clone masses, it still can show detected clones. The comparison histogram between *noRename_PDG_split* and *split* in Moss is shown in Figure 5.27.

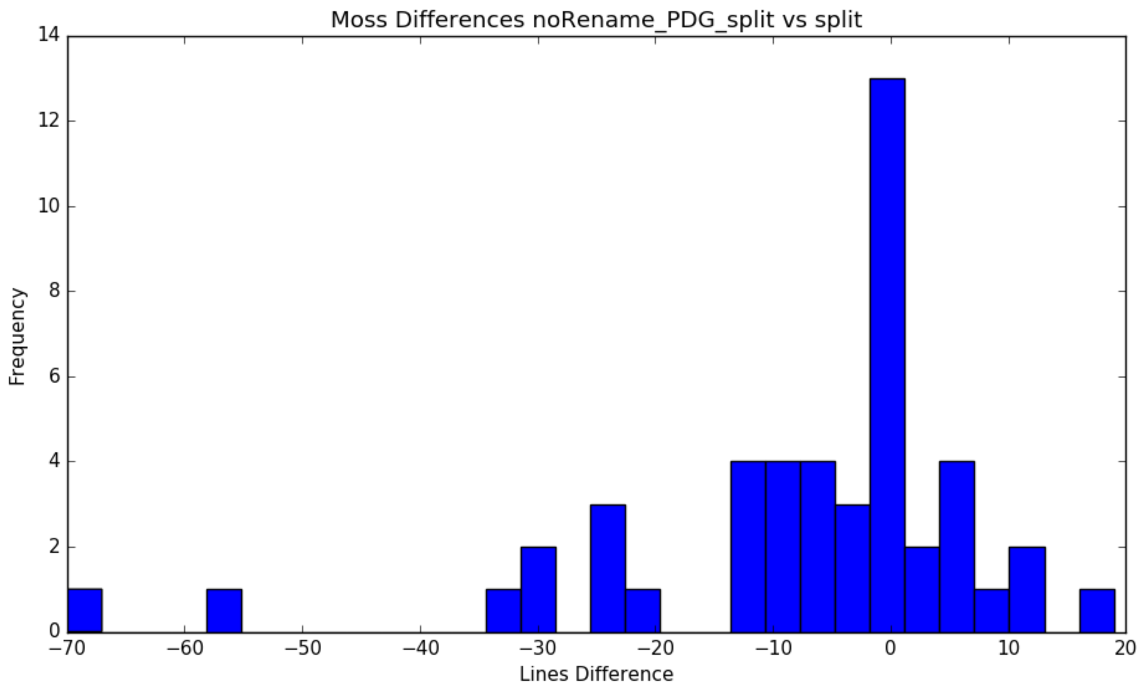


**Figure 5.27: Histogram of *noRename_PDG_split* to *split* using Moss**

Opportunities to increase the rate of finding clones are discussed in Chapter 6.

## 5.5 Same Identifiers

A small experiment was conducted where all identifiers were renamed as the same identifier `id`. This would no longer make the code semantically similar to the original, but it would remove the effect identifier names have on code clone detection. By naming all identifiers alike, the structure and logic of the code remains, which the tools will use as their basis for clone detection. Since Simian and CloneDR uses identifiers as part of their analysis, only their performance will be examined. JPlag and Moss will see no measurable difference from naming the identifiers the same.

For this section, instead of renaming each variable, all variables will be renamed to `id`. Therefore, in *noPDG*, the only normalization feature in effect is the renaming of all variables to `id`. The aggregate of Simian's performance is shown in Figure 5.28

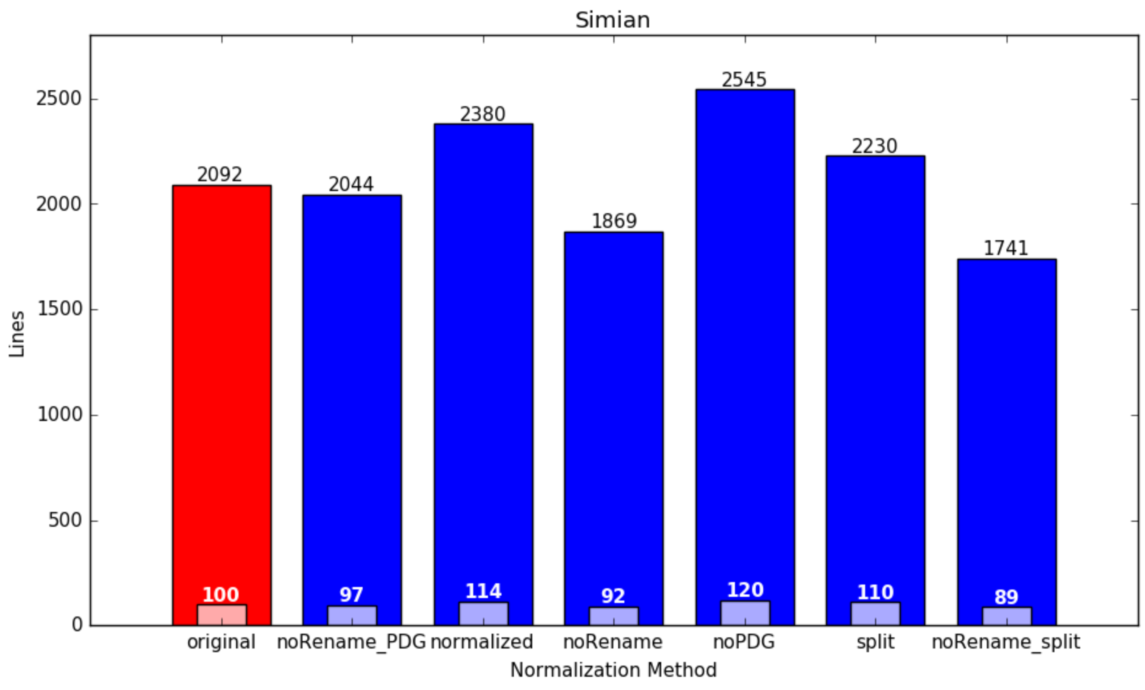


**Figure 5.28: Aggregate results of Simian with same identifier names**

By renaming all identifiers to `id`, *noPDG* has risen considerably, finding the most among all the versions. Just by renaming all the identifiers the same name and not reordering the statements provided the best overall clone mass detected. A breakdown of the distribution of clone mass changes from *original* to just identifier renaming in *noPDG* is shown in Figure 5.29.

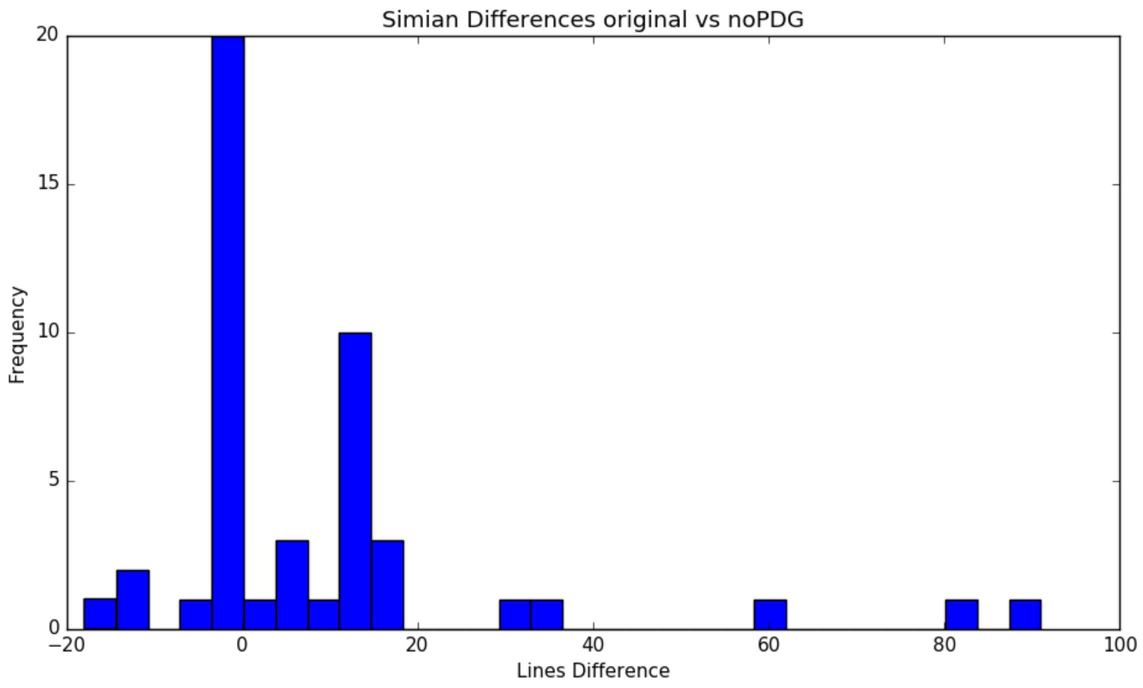


**Figure 5.29: Histogram of *original* to *noPDG* using Simian same identifier**

Some submissions still see a negative impact. This is due to preprocessing and how it slightly perturbs the source. Some submissions saw improvements of over 50 lines, and only a few submissions saw decreases. By naming all identifiers the same name, this is the upper bound of what identifier renaming can do. Version *normalized* fared in between *original* and *noPDG*. However, by including statement reordering, some new clones have been discovered and some clones got larger. The distribution of the difference from *noPDG* to *normalized* is shown in Figure 5.30.

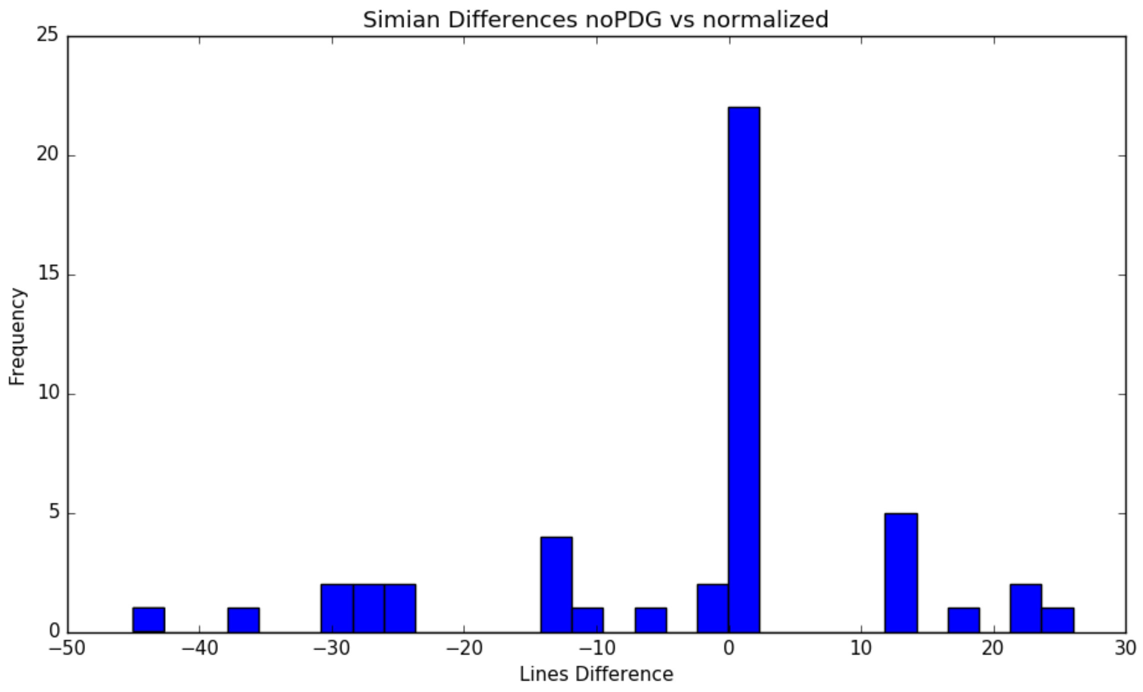


**Figure 5.30: Histogram of *noPDG* to *normalized* using Simian same identifier**

Statement reordering provided additional gains on top of same identifier renaming, but also causes some submissions to report fewer clones. CloneDR also has a positive effect from uniform identifier renaming, whose aggregate result is shown in Figure 5.31.

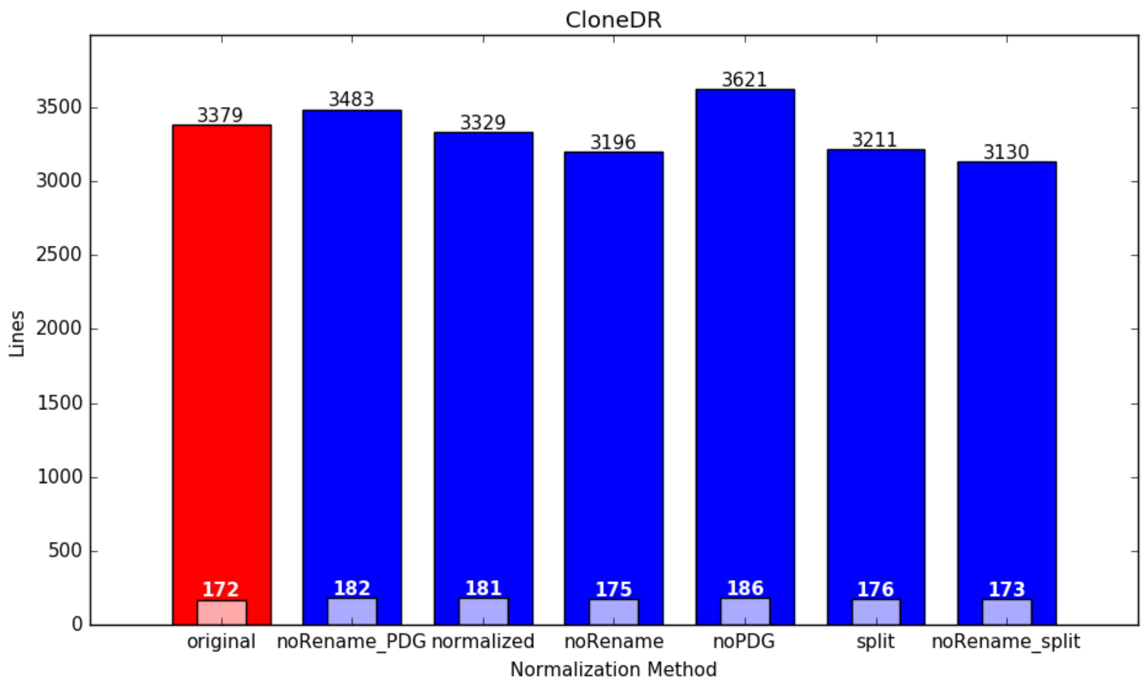


**Figure 5.31: Aggregate results of CloneDR with same identifier names**

Version *noPDG* performs the best just as in Simian's case, but *normalized* detected less than *original* for CloneDR. Some submissions still suffer from uniform identifier renaming although not as drastic as regular identifier renaming, which is shown in the comparison from *original* to *noPDG* in Figure 5.32.

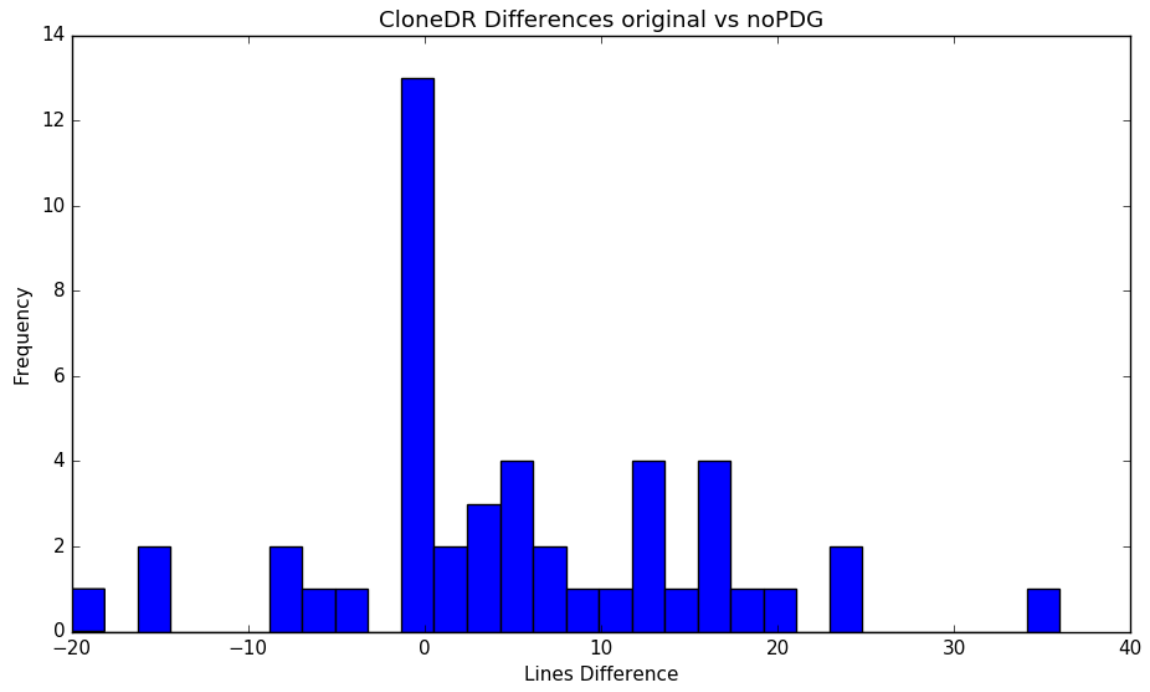


**Figure 5.32: Histogram of *original* to *noPDG* using CloneDR same identifier**

Chapter 6

# FUTURE WORK

*Normalizer* serves as a proof-of-concept that a singular tool can bolster the effectiveness of multiple different code clone detectors. Although results have been positive, there are improvements that can be made to *Normalizer*.

## 6.1 Pointer Analysis

Pointer analysis in C has always been a challenge. Although out of the scope of this thesis, a more in-depth pointer analysis would benefit the construction of the PDG. Knowing which variables may be modified or read instead of a sweeping generalization that memory has been written to or read from would allow a more surgically accurate PDG and grant greater freedom to reorder statements.

A specific area of plausible improvement is determining which variables can be modified. If the assignment is into an `int*` type, then it can be reasonable to assume that only `int`s are potentially modified, and not any other type. In the C language, this is not always held true, but if the code was written well enough in a non-convoluted fashion, then there is reason to believe that the assumption can be held true. If the goal of the code clone is to circumvent detection, such as academic cheating, then this improvement may be detrimental.

## 6.2 Different Identifier Normalization Namespaces

Identifiers are renamed based on when they are declared in the function, with the first identifier named as “a”, and subsequent identifiers named along each letter of the alphabet. Therefore even if two functions perform a similar task and one function has an extra variable to store a temporary value, then the naming of identifiers for that function will be shifted by one. By the mismatched variable names, the clone will not be detected by clone detectors that rely on identifier names.

A potential improvement could be to name identifiers based on their type and usage instead of first-come-first-serve. If the variable is declared as an `int`, then the name of the variable could be `int_a`. This way, each variable type occupies their own namespace and extra variables do not encroach on another type's namespace. Another way to name variables is that variables that are used more often than others should be named as a frequently used variable. Therefore, popularly used variables would have the same name across functions.

## 6.3 Depth-First Statement Reordering

Once a PDG has been created, the graph has to be serialized back into an AST to be printed as a C program. There are many ways to sort a topological graph, but determining the best way to expose clones is hard. Students write code in logical sections, where one idea occupies a contiguous block of code. Those contiguous blocks of code can appear in other clones too.

A potential idea for a better topological sort is to prefer a depth-first topological sort. In those logical section, one statement may be a direct prerequisite for the statement immediately after. Even though other statements may be eligible by the topological sort, the sort would favor statements that become available from the most recently sorted element. An example is shown in Listing 6.1 and its PDG is shown in Figure 6.1.

**Listing 6.1: Program with two logical code sections**

```
1  double func() {
2      double pi, rad, h, rCube, sph, sph3, circle, cyl;
3      pi = 3.14;
4      rad = 7;
5      h = 3;
6
7      rCube = rad * rad * rad;
8      sph = pi * rCube;
9      sph3 = sph + sph + sph;
10
11     circle = pi * rad * rad;
12     cyl = h * circle;
13
14     return sph3 + cyl;
15 }
```

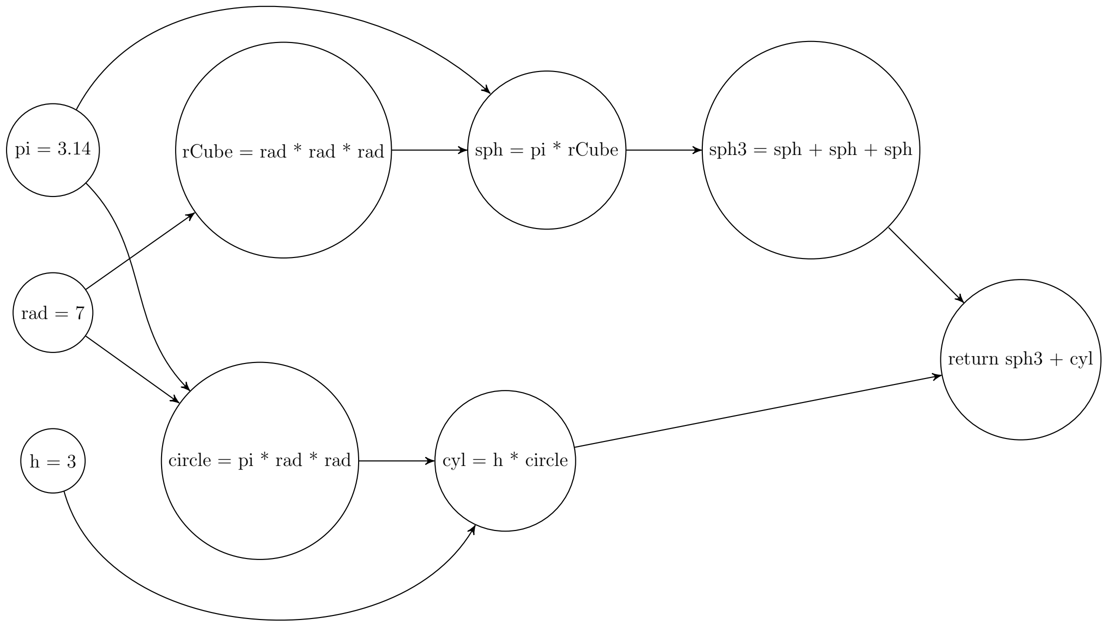


**Figure 6.1: PDG for Listing 6.1**

The code is split into 2 separate logical sections, one to compute the volume of three spheres in lines 7-8 and one to compute the volume of a cylinder in lines 11-

12 in Listing 6.1. If a depth-first topological sort were to take place, then it could arbitrarily pick to fulfill `sph3 = sphere * 3` and all of its prerequisites first. So to fulfill the prerequisites, the order would be `pi = 3.14`, `rad = 7`, `rCube = rad * rad * rad`, `sph = pi * rCube` then `sph3 = sph + sph + sph`. Then to fulfill `cyl = h * circle`, the sort would pick `circle = pi * rad * rad`, `h = 3`, then `cyl = h * circle`. Finally the return `return sph3 + cy1` would be chosen, concluding the sort. The resulting resorted program would look like Listing 6.2.

**Listing 6.2: Depth-first topological sort**

```
double func() {
    double pi, rad, h, rCube, sph, sph3, circle, cyl;
    pi = 3.14;
    rad = 7;
    rCube = rad * rad * rad;
    sph = pi * rCube;
    sph3 = sph + sph + sph;
    circle = pi * rad * rad;
    h = 3;
    cyl = h * circle;
    return sph3 + cyl;
}
```

Notice how the logical units are still contiguous for computing the three spheres and cylinder. Even if the input was jumbled, tangling the computation for the spheres and the cylinder such as Listing 6.3, it would result in the same PDG and the same sorted program.

**Listing 6.3: Jumbled program still has the same PDG as Figure 6.1**

```
double func() {
    double pi, rad, h, rCube, sph, sph3, circle, cyl;
    pi = 3.14;
    rad = 7;
    h = 3;
    rCube = rad * rad * rad;
    circle = pi * rad * rad;
    sph = pi * rCube;
    cyl = h * circle;
    sph3 = sph + sph + sph;
    return sph3 + cyl;
}
```

A way to summarize this sorting algorithm is to pick nodes that are immediately used by the next node.

## 6.4 Traceback

As a hint for the end user, the original code is written as a comment besides the normalized code to allow for backtracking. Support can be increased by providing line numbers corresponding to the original non-preprocessed code. This would require parsing the original source code with the preprocessor directives unresolved. This does not help in detecting more clones, but it makes it easier for humans to find where the clones are in the original source.

## 6.5 Code Clone Detection

*Normalizer* has constructed the internal representation of the input source and has the necessary information to detect clones by itself. Abandoning the existing tools and creating a code clone detection algorithm is a potential avenue for improvement. Since PDG construction is already done in *Normalizer*, PDG comparisons is another logical step besides serializing the PDG and outputting source code. The results from PDG comparison can be more accurate than AST or token based approaches. Program order is not as significant a factor in PDG comparisons than AST comparisons. By serializing the PDG into a program, the PDG is forced to take on one of many possible linear representations of the program, which makes detection harder.

Comparing two graphs to check if one graph contains an isomorphic graph equal to the other is an NP-complete problem [8]. Graph comparisons take a lot of time, but with small PDGs, a brute force appraoch can be feasible in an acceptable amount of time. Among many PDGs, some of which could potentially be large, a heuristic approach can be adopted to find code clones. The transitive reduction of a PDG discussed in Section 4.3.6 will help in PDG comparisons.

# Chapter 7

# CONCLUSION

Code clones can appear in any codebase, intentionally and unintentionally. Detecting code clones can lead to increasing the maintainability of the code. Numerous tools exist to report clones, but they do not augment the performance of another [18]. *Normalizer* is not a code clone detector, but instead is a source code formatter that rewrites the source code so clones look as similar to each other as possible. The formatted code is then used as input for code clone detectors.

*Normalizer* can help detect clones that would not have been detected without normalizing the code. The benefit of using *Normalizer* is dependent on many variables, such as coding style, the tool used to detect clones, and the heuristics *Normalizer* uses to reorder statements or rename variables.

There are many instances where, by using *Normalizer*, new or larger clones can be detected that the original source code did not expose. At the same time, *Normalizer* can also make clones shrink or disappear. However, in the intent to catch as many clones as possible, a viable option is to run the code clone detection tools both with and without *Normalizer* and gather the clones from both executions.

It is shown that the input source can have a dramatic effect on code clone detection. Even if two programs perform the same logical task, depending on something as trivial as the programmer's coding style, those logical clones may not be detected. *Normalizer* attempts to remove the programmer's style out of the code by rewriting the code in its own style while preserving the logic. By consistently rewriting the code, more clones may be exposed. From this proof-of-concept, source code normalization can be used to strengthen the effectiveness of code clone detection tools.

# BIBLIOGRAPHY

[1] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–, Washington, DC, USA, 1998. IEEE Computer Society.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, Sept 2007.

[3] D. Bruschi, L. Martignoni, and M. Monga. Code normalization for self-mutating malware. *IEEE Security and Privacy*, 5(2):46–54, Mar. 2007.

[4] R. Gauci. Smelling out code clones: Clone detection tool evaluation and corresponding challenges. *arXiv preprint arXiv:1503.00711*, 2015.

[5] D. Gries, A. J. Martin, J. L. van de Snepscheut, and J. T. Udding. An algorithm for transitive reduction of an acyclic graph. *Science of Computer Programming*, 12(2):151 – 155, 1989.

[6] S. Harris. Simian - similarity analyser, Feb 2017.

[7] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[8] S. Kijima, Y. Otachi, T. Saitoh, and T. Uno. Subgraph isomorphism in graph classes. *Discrete Mathematics*, 312(21):3164 – 3173, 2012.

[9] R. Komondoor and S. Horwitz. Eliminating duplication in source code via procedure extraction. *UW-Madison Dept. of Computer Sciences, Technical Report*, 1461, 2002.

[10] T. Parr. Antlr, Feb 2017.

[11] L. Prechelt. Jplag cpp tokens, mar 2017.

[12] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with jplag. *J. UCS*, 8(11):1016, 2002.

[13] D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[14] C. K. Roy. *Detection and Analysis of Near-miss Software Clones*. PhD thesis, Kingston, Ont., Canada, Canada, 2009. AAINR65337.

[15] C. K. Roy and J. R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 172–181, Washington, DC, USA, 2008. IEEE Computer Society.

[16] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, May 2009.

[17] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 76–85, New York, NY, USA, 2003. ACM.

[18] A. Sheneamer and J. Kalita. A survey of software clone detection techniques. *International Journal of Computer Applications*, pages 0975–8887, 2016.

# Appendix A

# BOX PLOTS

These box plots are the distribution of clone masses detected by a code clone tool per student. These plots give a sense of the range of code clones produced in a typical class. Some students do not make any clones and a few students produce relatively much higher counts of clones. In addition, these box plots also provide an idea of how much a code clone detector can vary in its performance from student to student. However, each point is also subject to the amount of clones that exist in that submission.

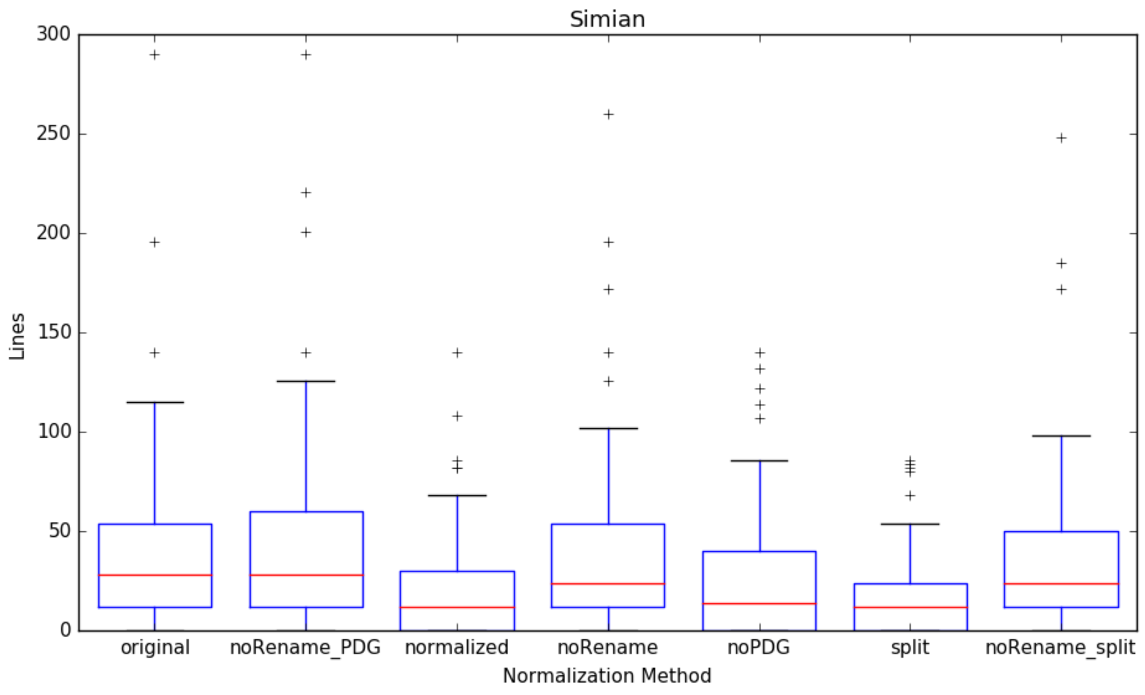


**Figure A.1: Box Plot of Simian's Results**

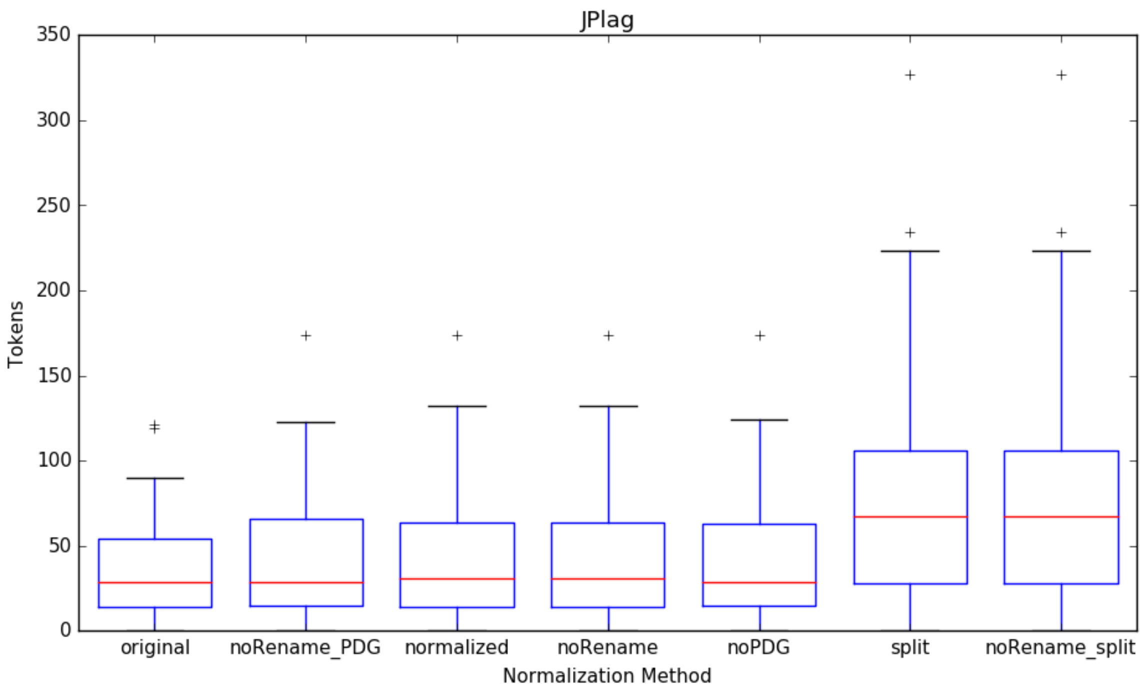


**Figure A.2: Box Plot of JPlag's Results**

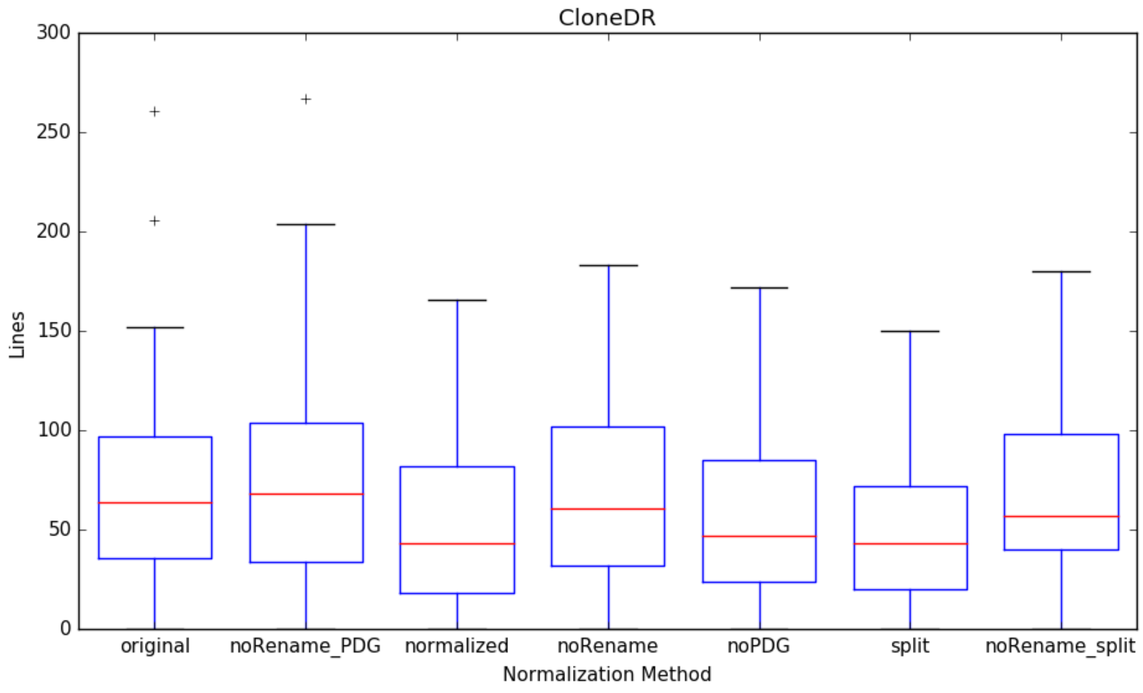


**Figure A.3: Box Plot of CloneDR's Results**

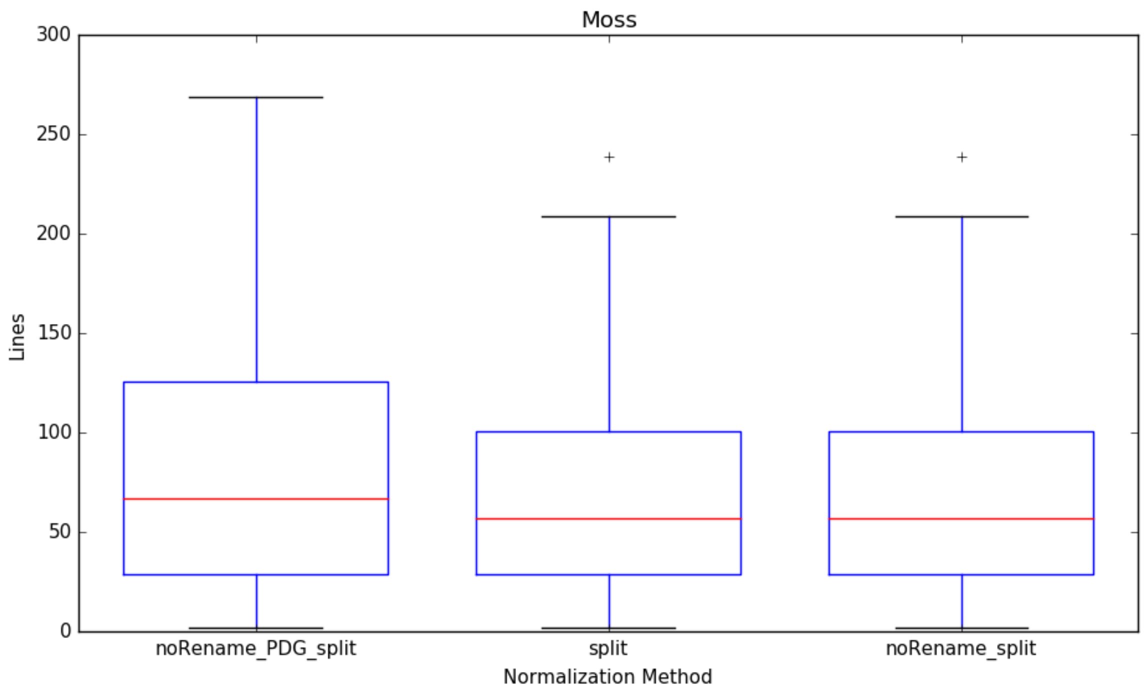


**Figure A.4: Box Plot of Moss's Results**