CLONELESS: CODE CLONE DETECTION VIA PROGRAM DEPENDENCE GRAPHS WITH RELAXED CONSTRAINTS

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

Thomas Simko

June 2019

COMMITTEE MEMBERSHIP

TITLE: Cloneless: Code Clone Detection via Program Dependence Graphs with Relaxed Constraints

AUTHOR: Thomas Simko

DATE SUBMITTED: June 2019

COMMITTEE CHAIR: Aaron Keen, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Theresa Migler, Ph.D.
Assistant Professor of Computer Science

COMMITTEE MEMBER: David Janzen, Ph.D.
Professor of Computer Science

ABSTRACT

Cloneless: Code Clone Detection via Program Dependence Graphs with Relaxed Constraints

Thomas Simko

Code clones are pieces of code that have the same functionality. While some clones may structurally match one another, others may look drastically different. The inclusion of code clones clutters a code base, leading to increased costs through maintenance. Duplicate code is introduced through a variety of means, such as copy-pasting, code generated by tools, or developers unintentionally writing similar pieces of code. While manual clone identification may be more accurate than automated detection, it is infeasible due to the extensive size of many code bases. Software code clone detection methods have differing degree of success based on the analysis performed. This thesis outlines a method of detecting clones using a program dependence graph and subgraph isomorphism to identify similar subgraphs, ultimately illuminating clones. The project imposes few constraints when comparing code segments to potentially reveal more clones.

ACKNOWLEDGMENTS

Thanks to:

- My family and friends for their unconditional support
- Aaron Keen for all of his help and reading the many drafts
- Andrew Guenther, for uploading this template

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Code clones are segments of code that perform the same action. Code clones may be introduced as a code base evolves. These clones can either be exact copies of each other or can even be expressed with syntactic differences and still have the same functionality. If two segments of code can be refactored into one, then they qualify as a clone. Many times duplication issues arise in large corporations where two people can write similar pieces of code without having knowledge of another implementation's existence. Other times, code clones can be willingly introduced to prevent overhead from function calls [16]. Ultimately, this results in both pieces of code needing to be maintained when only one is necessary. Due to the high monetary investment in maintaining code, shrinking the code base by removing duplicates is very important [20].

Code bases are subject to various influences that may impact the detection of clones. Since code is in a linear form within a file, a developer's style contributes to the structure of the code. Certain expressions can be rearranged in different orders without changing the functionality of the code.

1.1 Types of Clones

Despite the simple definition of a code clone, two code segments having the same functionality, Roy, et al., identified four different types of duplicate code [17].

1.1.1 Type I Clones

Type I clones are segments of code that are direct copies of one another. These code clones may be introduced through copy-and-pasting code. Due to the high similarity between these clones, they are the easiest to identify. In order to find these clones, no knowledge of the language is required; they can be found purely through textual comparison.

1.1.2 Type II Clones

Type II clones are almost exact clones, but with minor naming changes. The structure of the clones is exactly the same. The differences between Type II clones are comments, different variable names or types, or reordered methods. These clones are commonly found by tools that have minimal knowledge about the language.

1.1.3 Type III Clones

Type III clones are segments of code with a different order, inserted statements, or removed statements, but the same overall functionality. These clones may arise through copy-and-pasting code, then modifying it to suit one's needs.

1.1.4 Type IV Clones

Type IV clones are pieces of code that are syntactically different, but still have the same underlying functionality as illustrated in Listing 1. These clones are often unknowingly introduced into the code base by different individuals.

```
public int func1(int num) {
    int i = 1;
    int sum = 0;
    while (i <= num) {
        sum = sum + i;
        i = i + 1;
    }
    return sum;
}
```

```
public int func2(int num) {

    int sum = 0;
    for(i = 1; i <= num; i++) {
        sum += i;

    }
    return sum;
}
```

Listing 1: Example of Type IV Code Clones

In Listing 1, both functions sum the numbers from 1 to *num* but in two different ways. One uses a for-loop and the other uses a while-loop. This syntactic difference is enough to fool many code clone detectors, even though they are semantically the same. Although these may be easy for a human to identify, these clones are much more difficult for software to discover.

```
public int func3(int num) {
    if(num <= 0)
        return 0;
    return num + func2(num - 1);
}
```

Listing 2: Recursive Type IV Code Clone for Listing 1

Listing 2 shows a less obvious clone of those in Listing 1. The recursive approach to the same problem obfuscates the clone, making it harder for both humans and software to detect. Furthermore, if the clone is mixed with other code, the problem worsens. Detecting these clones requires the software to have in-depth knowledge of

the language and variable usage. By performing complex analysis on the code, clone detection tools can more accurately find clones than would be reported by purely textual comparisons.

However, not all clones are interesting to report. For example, Listing 3 shows two snippets of code that are technically clones, but refactoring them into one implementation may reduce readability, making it harder to maintain. Therefore, it is important for tools to only report clones of interest (i.e., those that can be meaningfully refactored) to a developer.

```
public class Class1 {
  int num;

  public Class1(int num) {
    this.num = num;
  }
  ...
}
```

```
public class Class2 {
  int num;

  public Class2(int num) {
    this.num = num;
  }
  ...
}
```

Listing 3: Clones Not of Interest

Detecting code clones is important in properly maintaining a code base. When a developer is tasked with changing an instance of a clone, the others should be potentially changed as well. By not propagating the fix to all instances of a clone, bugs can be introduced [19]. The preferred method for resolving a code clone is to move the duplicate logic into a shared function to reduce the number of sites to consider for updates. Furthermore, by extracting the duplicate code into a function, the size of the code base is decreased, potentially reducing the size of the executable and the learning curve for new developers.

The goal of this thesis is to identify potential clones of interest using a program dependence graph to relax the unintended structure introduced by writing code in a linear form. Certain constraints are lifted from the graph to reduce the number of edges, potentially illuminating more clones by creating more similar graphs. Ultimately, once a clone has been detected, it is the developer's choice whether the clone should be eliminated or not. Therefore, this thesis is primarily focused on reporting a large number of clones of interest, leaving the final removal decision to the developer.

Chapter 2

BACKGROUND

This chapter defines the algorithms and data structures used to detect code clones using program dependence graphs.

2.1 Abstract Syntax Tree

An abstract syntax tree (AST) is a tree-based representation of code where each node represents an element, such as an operand, expression, or variable, from the original source code. It focuses on the constructs of the language (e.g. statements, expressions, etc.) rather than its concrete syntactic structure. To create the AST, the code is first passed through a tool like ANTLR [15], which parses and builds a parse tree. Performing minimal modifications to the ANTLR parse tree creates an abstract syntax tree. The AST format accounts for the order that operations are applied to an element; items that are closer to the leaves of the tree are evaluated first. Figure 1 shows a graphical representation of the abstract syntax tree for a small function.

2.2 Control Flow Graph

A control flow graph (CFG) represents the flow of control through a function where nodes represent basic blocks of code and edges represent the transfer of control. It is a directed graph such that two blocks are connected if one follows the other during a potential runtime scenario.

After transforming the code into this representation, all of the possible routes of a

```
int factorial(int num) {
    int fact = 1;
    while(num > 0) {
        fact = fact * num;
        num = num - 1;
    }
    return fact;
}
```

Listing 1: Small Factorial Function

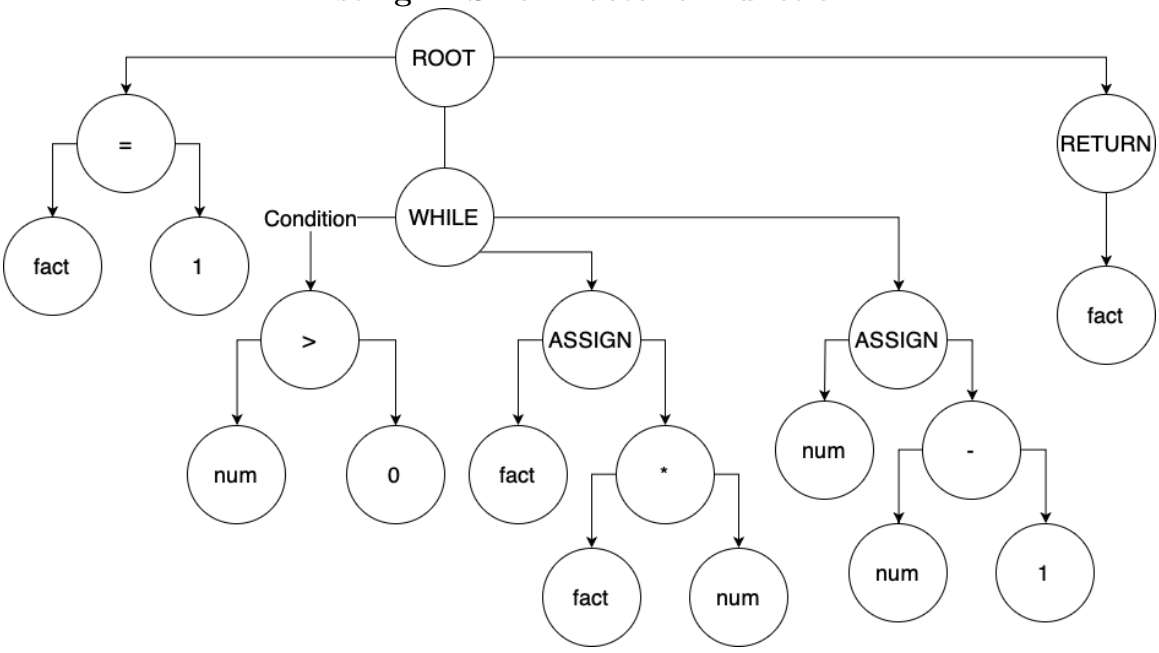


Figure 1: Abstract Syntax Tree for Function Body in Listing 1

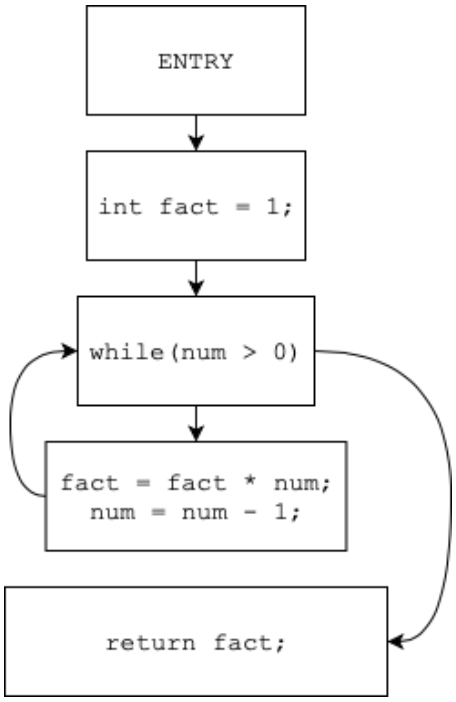


Figure 2: Control Flow Graph for Function Body in Listing 1

function can be traced. Loops are expressed in this format by an edge pointing to a predecessor of a given node. If a subgraph of blocks cannot be reached from the entry block, then the code is dead and has no impact on the execution of the function. The dead code can then be removed, leaving behind only the core code of the function.

Figure 2 shows a CFG for the factorial function in Listing 1. Starting from the $ENTRY$ node, there is only one successor. Since there is only one successor, the initialization of $fact$ will always occur. Proceeding to the next block, different actions may occur based on the predicate in the while-statement. If the predicate evaluates to true, the code continues with the block within the while-statement's curly braces, which eventually loops back to the while-statement's predicate. But if the predicate is false, that block will be skipped and the code will proceed to the final return statement block.

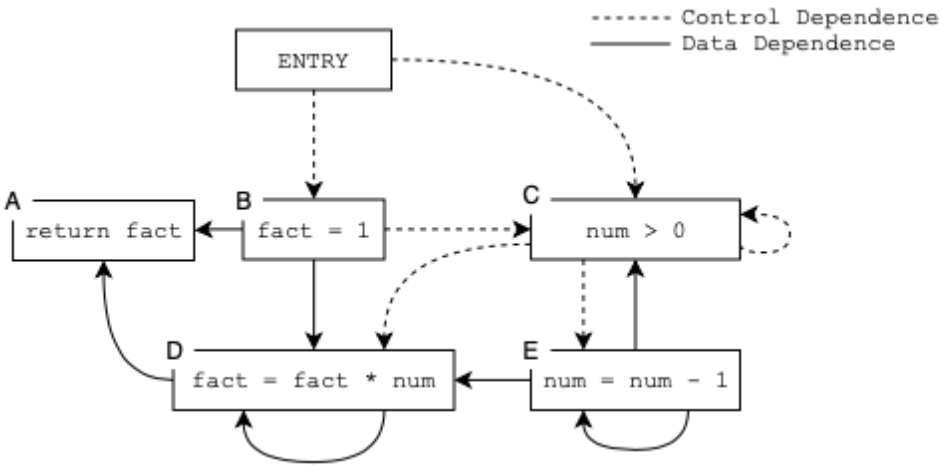


Figure 3: Program Dependence Graph for Listing 1:
solid arrows represent data dependencies, dashed arrows represent control dependencies.

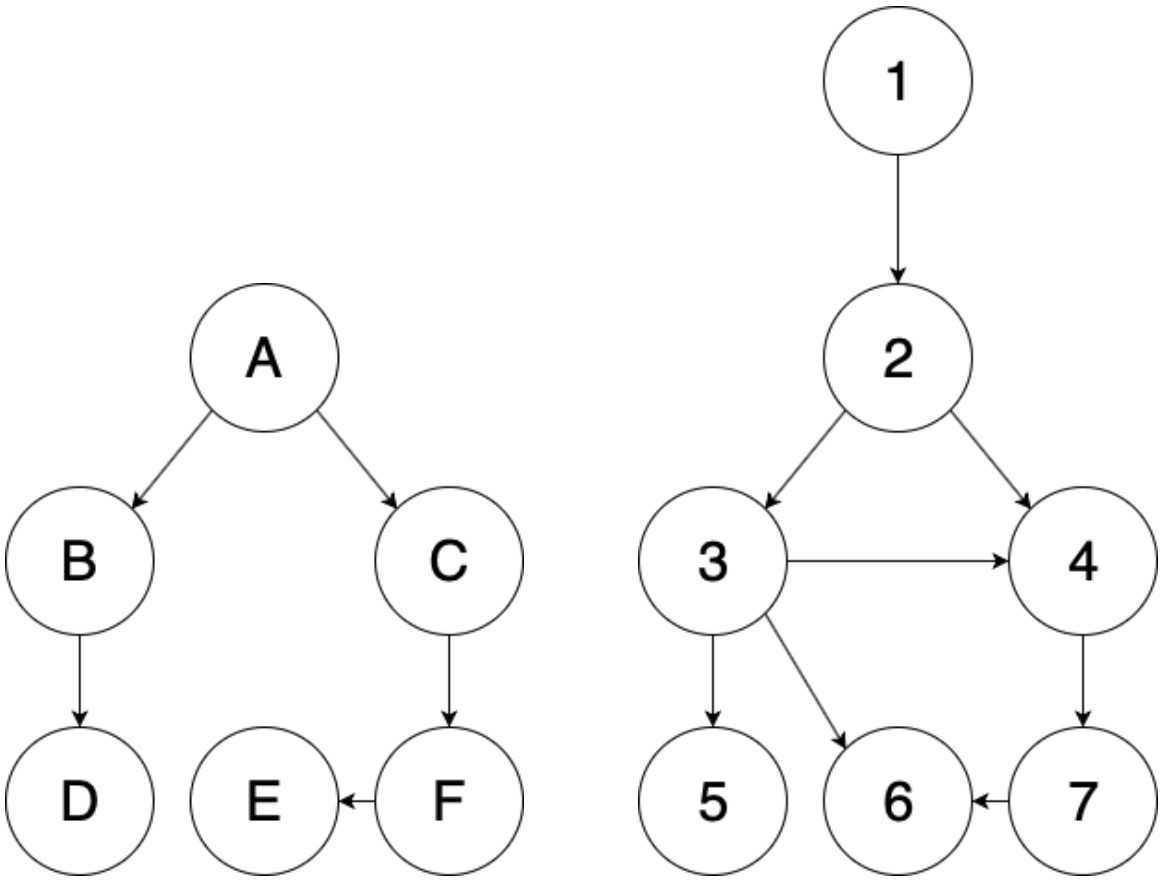
2.3 Program Dependence Graph

The program dependence graph (PDG) is a structure where each vertex represents an expression. A directed edge connects two expressions when one expression depends on the execution of the other. Dependencies include both data dependencies, when a variable is used after being defined, and control dependencies, the ordering of expressions where one expression's execution is conditionally guarded by another. A control dependence can be introduced by expressions residing in different basic blocks in the CFG or from a write-after-write (changing a variable after a prior assignment).

Figure 3 shows the PDG for the factorial function in Listing 1. Starting from the $ENTRY$ node, two control dependencies exist; however vertex B must be computed first because vertex C depends on it. This control dependence comes from the expressions residing in different blocks of the CFG. The expression $num > 0$ has a data dependence from vertex E because when num is decremented, the predicate is evaluated again with the updated value. The vertex D depends on the identifiers $fact$ and num and also has a control dependence from the while-loop's condition, C.

2.4 VF2

VF2 is a graph isomorphism algorithm which can be modified to find common subgraphs. The algorithm mimics a depth-first search approach at each stage to determine potential matchings from a small graph, G_1, to a large graph, G_2. Beginning with an initial state where no vertices have been matched, candidate pairs are computed from the two graphs. These pairs are defined by feasibility rules that compare two nodes, one from each of the graphs. Two nodes are considered candidate pairs if they satisfy feasibility rules. These rules are characterized by checks on the *in* and *out* degrees of the vertices in question and their successors. The algorithm performs a 1-look-ahead and 2-look-ahead by checking the neighborhood surrounding the vertices in question. If the neighborhood of the small graph does not match that of the large graph, then that vertex is not a match. The goal of these look-aheads is to streamline the depth-first search by checking if a vertex's neighborhood is compatible with the larger graph, saving time and memory. In addition to structural feasibility rules, semantic rules can be added to match vertices on their attributes [4].



(a) Small Graph (b) Large Graph

Figure 4: Sample Graphs for VF2 Algorithm

For example, in order to determine whether the small graph is a subgraph of the large graph in Figure 4 the algorithm proceeds as follows (as illustrated in Figure 5). To begin, the matching is empty. Then, an arbitrary vertex is selected with a valid candidate pair, such as $A \rightarrow 2$ as shown in Figure 5 Graph B. For each of the current pair's successors, its candidate pair is computed and the feasibility rules are applied. Selecting vertex C, a match may be found with vertex 3 (see Figure 5 Graph C). The successors of vertex C are then matched with the successors of vertex 3, eventually failing to match all of its successors because vertex F has a successor vertex E while vertex 5 does not have a successor (Figure 5 Graph D). The algorithm backtracks to the last valid match with not visited successors, $A \rightarrow 2$. A different

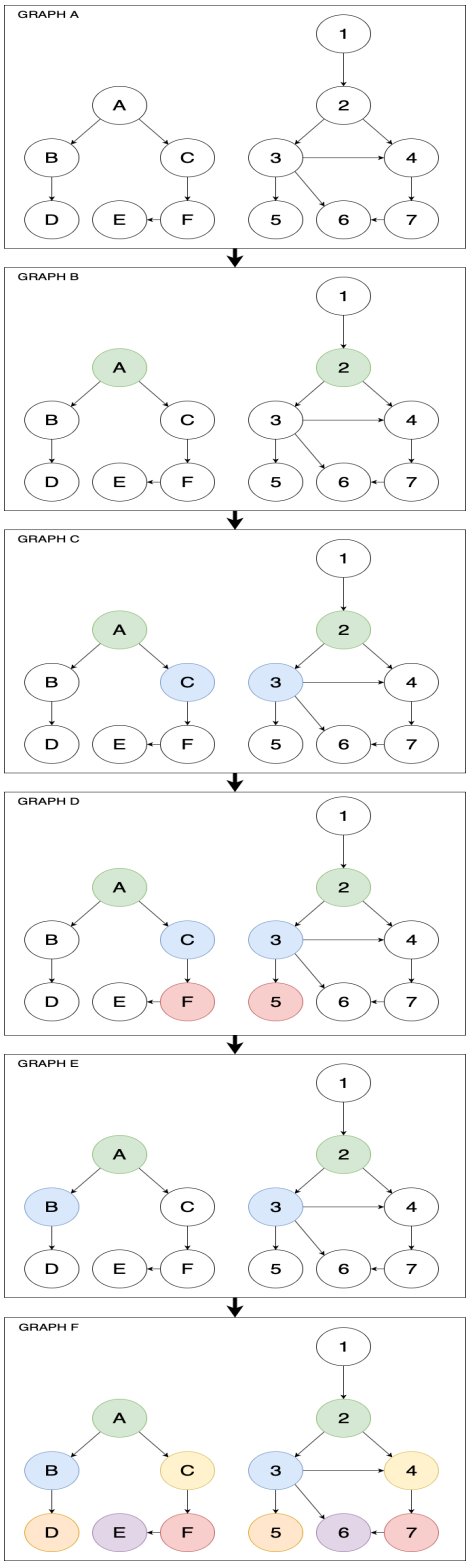


Figure 5: Sample VF2 Algorithm Walkthrough for Figure 4

matching is selected among its successors, $B \rightarrow 3$ (see Figure 5 Graph E). Continuing the matching, $D \rightarrow 5$, $C \rightarrow 4$, $F \rightarrow 7$, and $E \rightarrow 6$ are computed, resulting in the matching shown in Figure 5 Graph F. Since all of the subgraph's vertices are used, the algorithm terminates with a valid matching.

2.5 Reaching Definitions

The Reaching Definition analysis [5] is a method to determine which definitions are available at each block in a program. When a variable is defined and is not redefined along a path to some block, then the definition 'reaches' that block (i.e., the value at the definition site is candidate for use in the block).

Listing 2 shows how multiple definitions can reach an expression. The return-statement on line 4 could have a definition from the assignment on line 1 or line 3. The assignment that reaches the expression is dependent on the if-statement's condition.

```
1 int num = 4;
2 if(condition)
3     num = 7;
4 return num;
```

Listing 2: Example Code Where Two Definitions Reach the Same Expression

Chapter 3

RELATED WORKS

This chapter presents different tools that are used to aid in the detection of clones.

3.1 Simian

Simian[6] is a code clone detection tool that operates on a wide variety of languages, focusing on textual comparisons. The goal of this tool is to quickly identify potential duplicates within large files. Simian allows users to indicate the source language so that it can process the tokens better; ultimately allowing it to disregard variable names, modifiers, and literals. By ignoring these tokens, Simian can more accurately identify clones because these items do not impact the overall structure of the code. It also allows for many configurations, including but not limited to, ignoring case and setting a line number threshold before a duplicate is reported. At the time of this thesis, Simian is available for download and is used for validation of Cloneless.

3.2 CloneDr

CloneDr[2] goes one step further than Simian by building an abstract syntax tree for the analyzed source code. Using the AST representation, CloneDr calculates the similarity between code segments based on the number of shared nodes and shared subtree nodes. In order to improve efficiency, CloneDr hashes the trees using an artificially bad hashing function. The intentionally subpar hash function gives CloneDr the ability to ignore certain data that a good hashing function might include, such

as variable names. Hashed values are then placed in buckets and compared. If the similarity threshold is met, then they are added to a clone list. By hashing the trees, CloneDr minimizes the number of tree comparisons, but it also leads to loss in data. While attempting to use CloneDR to detect clones, the application throws an exception, preventing it from being further analyzed in this thesis.

3.3 SimScan

SimScan[1] is a tree-based code clone detection tool that performs comparisons purely off the parser's output. SimScan was used within IDEs such as IntelliJ and Eclipse where code clones could be caught as they were being created. This tool had many configurable options, such as the volume, similarity, and speed, that affect the performance and accuracy of the tool. Although this tool was widely used at one point, it has been removed from the aforementioned IDEs and its website is no longer operational.

3.4 Duplix

Duplix[12] is a code clone detection tool that uses a program dependence graph to evaluate code blocks. By using a program dependence graph, semantics and grammar are accounted for, potentially highlighting code segments with different structure but similar functionality. After creating these graphs, Duplix performs graph comparisons, trying to identify similar subgraphs. Due to the nature of these comparisons, it has quadratic complexity, requiring the need for a limit placed on the number of comparisons. The tool is no longer available online for use.

3.5 Scorpio

Scorpio[10], similar to Duplix, compares subgraphs in the program dependence graph. For its comparisons, it performs a three-step process:

1. Find pairs of clones by placing all expressions into equivalence classes.
2. Remove subsumed clones.
3. Combine pairs of clones into larger groups.

During its research, it was discovered that some clones were presented multiple times because the subgraphs of similar subgraphs are also similar. In addition to this, it accounts for when there are more than two clones. For instance, if code segment A is a clone of segment B and segment B is a clone of segment C, then segments A, B, and C are all grouped together because they are all clones of each other. Scorpio does not identify code clones that cross loops if the loop itself does not match. The performance of Scorpio could not be analyzed because the project would not build.

3.6 Normalizer

Normalizer[13] is a code clone detector enhancement tool that formats the code in a consistent manner for use as input to clone detection tools. The goal of modifying the code in this fashion is to remove the programmer's style from the code and to remove useless code. Using this tool alongside a clone detection tool has a tradeoff of hiding some clones and revealing others.

Running code through Normalizer modifies it into a standard form using a variety of options, such as renaming variables, removing useless code, and reordering state-

ments. In addition, it can perform all combinations of the above actions, storing each result to its own file. In order to maximize its chances of finding clones, Normalizer changes certain structures into others, such as changing a for-loop into a while-loop.

```
int func(int num) {
  for(int i=1; num<100;num+=j) {
    int j = num % 10;
  }
  return i + num;
}
```

(a) Before Normalizing

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

(b) After Normalizing

Listing 1: Sample of Normalizer's Output

Listing 1 shows how Normalizer sets a standard form for code by renaming all of the variables and transforming the for-loop. Normalizing code helps reveal code that might have the same functionality but different structure.

Normalizer creates a graphical representation of the code, but then converts it back into a linearized form, potentially obscuring clones. Textual-based code clone detection tools, like Simian, do not necessarily benefit from sorting the code. Normalizer uses a topological sorting algorithm that orders instructions on certain attributes, such as expression complexity. Since the ordering of expressions matters with textual comparison tools, the sorting can hide clones by interweaving other code blocks with a potential clone. Normalizer was not used in the analysis of this thesis because it was written to normalize C source code, while this thesis targets Java.

3.7 NICAD

NICAD[18] is a hybrid approach to code clones. The goal of NICAD is to be lightweight, like a textual-based comparison tool, and as accurate as graphical comparison tools. It parses code using TXL[7] to create a parse tree and perform normalization on the code as defined by the configuration. NICAD then gathers all statements of a given type to be stored as potential clones using TXL's *extract* function. NICAD performs post-processing on the potential clones to split and sort the clones into clusters. It finds the spanning clones by using the longest common subsequence algorithm, counting the number of unique items in the spanning clone. A percentage similarity is computed and if it meets a certain threshold, the clones are reported. NICAD is readily available and is used during the analysis of this thesis.

Chapter 4

IMPLEMENTATION

Cloneless takes a Java file or directory containing Java files as input and then searches for clones within. The entire implementation was coded from scratch except for the use of ANTLR and VF2. The current implementation requires syntactically valid input. Section 4.5 will discuss further syntactic limitations.

4.1 Initial Processing

The input is first parsed and a parse tree is built. The parse tree is then transformed into an AST (as discussed in Section 2.1) and information, such as filename and operands, is added to each node. In addition, for-statements are transformed into while-statements to maximize the similarity between like structures. Switch-statements and ternary expressions may also be converted at this time to if-statements; however, Cloneless does not support them in its current form. Prefix and postfix unary expressions are also transformed to assignment expressions that save the values into temporary variables.

To form the CFG (as discussed in Section 2.2), blocks are created for the given statements, and each statement's expressions are added to their corresponding blocks. At this stage, statements are no longer needed since they are encapsulated in the structure of the graph, leaving just expressions within the blocks. Each block of expressions has pointers to its successors and predecessors for more efficient processing.

4.2 Creating the PDG

In order to convert the CFG into the PDG (as discussed in Section 2.3), relationships between the expressions need to be formed. Iterating through each expression from top-to-bottom, the Reaching Definitions (Section 2.5) for each source identifier are added as data dependencies. Control dependencies in the scope of this project only pertain to write-after-write dependencies to minimize the number of edge constraints that are placed on the graph. These dependencies are added from a target variable in an expression and to each of the variable's reaching definitions. If an expression is both data and control dependent on another expression, only the data dependence is added because it is a stronger constraint. The current implementation also assumes there are no side-effecting functions, meaning functions do not modify any variables outside of their local state. Therefore, there are no control dependencies between method calls.

```
1 public int func1(int num) {
2     int index = 1;
3     int sum = 0;
4     while (index <= num) {
5         sum = sum + index;
6         index = index + 1;
7     }
8     return sum;
9 }
```

Listing 1: Sample Function

Listing 1 shows sample code to consider for PDG construction. Beginning at the expressions on lines 2 and 3, variables *index* and *sum* are defined. The next statement

has a data dependence on *index* as shown in Figure 6(a). This statement also depends on *num*, but *num* is not defined within the function body (num is a parameter and, thus, initialized before any expression within the function).

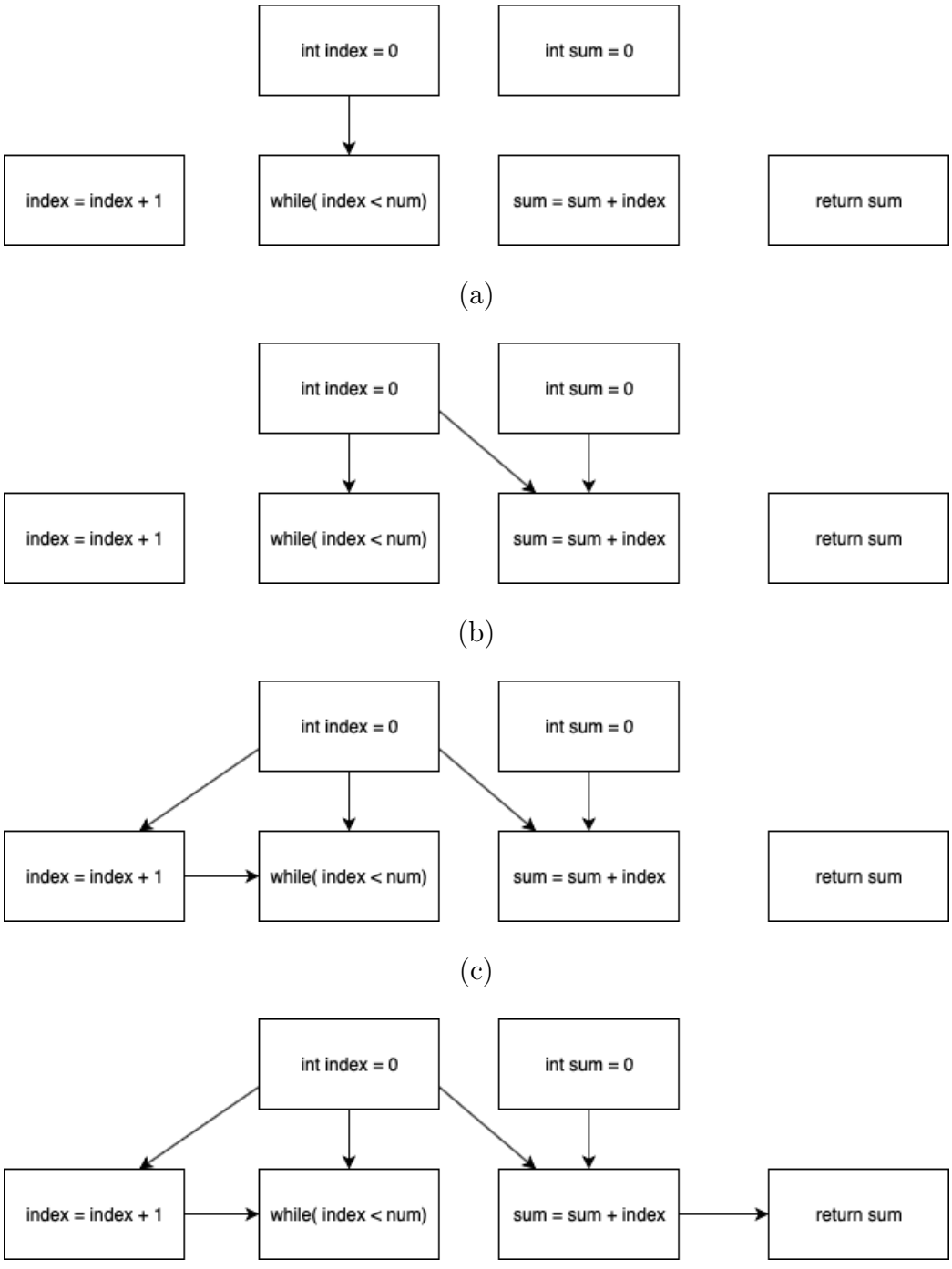


Figure 6: Sample Graphs for PDG Construction

A control dependence is not added because there is no modification of index (a write-after-write dependence). Next, the assignment expression on line 5 has a data depen-

dence from the definitions on lines 2 and 3. Note, a control dependence is not added because the data dependence is a stronger constraint (see Figure 6(b)). The expression on line 6 gets a data dependence from the definition on line 2. Additionally, the while-loop's condition gains a data dependence from the assignment on line 6. The current state can be found in Figure 6(c). Finally, *sum* gains data dependencies from lines 3 and 5 due to the assignments into *sum*. Figure 6(d) shows the final PDG for the sample function. At this stage, each PDG represents a function in the source code.

4.3 Subgraph Isomorphism

From the PDG, subgraph isomorphism is performed using the VF2 algorithm. The algorithm requires a small graph that the program attempts to match to a large graph. Since the number of large graphs is relatively small for a program, they can be computed first and stored in memory. The large graphs are created by computing all of the component connected graphs in a program. Component connected graphs are subgraphs where one component is defined by the set of vertices that have a path connecting them. These graphs are necessary because the clones should be localized to a region in a function. Otherwise, clones would span functions throughout a program and lose meaning.

For each expression in the graph, every possible connected subgraph containing that expression is created. This begins by constructing a subgraph containing only the expression itself. Additional subgraphs are constructed from existing subgraphs by traversing the incoming or outgoing edges of members of the existing subgraph. Each of the subgraphs are checked for isomorphisms within the large graphs.

Since there is high overlap between subgraphs, it is important to filter out subgraphs

that have already been checked for isomorphism. At the end of the algorithm, the matchings are in the Priority Queue.

4.4 Matching Filtering

At this stage, the matchings still contain many duplicate entries as shown in Listing 3, so filtering is performed. The matchings are stored in a Priority Queue that is sorted on the size of each matching from largest to smallest. The algorithm filters out unique matchings that are not a subset of another saved matching, resulting in the final set. This ensures that the longest clones are saved. The final matchings are output to the user through the terminal with a verbose option that describes which expressions match with one another.

```
public int func1(int num) {
    int i = 1;
    int sum = 0;
    while (i <= num) {
        sum = sum + i;
        i = i + 1;
    }
    return sum;
}
```

Listing 2: Sample Function for Matching

Listing 3 shows the sorted unfiltered matchings for Listing 2. The instructions in this matching overlap and are all a subset of the first matching in Listing 3(a). By filtering these matchings, the result is simply the matching in Listing 4.

```
          int i = 1          →  int sum = 0
          int sum = 0        →  int i = 0
(a)       i <= num           →  i <= num
          sum = sum + i      →  sum += i
          i = i + 1          →  i++
          return sum         →  return sum
-------------------------------------------------------
          int i = 1          →  int sum = 0
(b)       int sum = 0        →  int i = 0
          i <= num           →  i <= num
          sum = sum + i      →  sum += i
-------------------------------------------------------
          sum = sum + i      →  sum += i
(c)       i = i + 1          →  i++
          return sum         →  return sum
-------------------------------------------------------
(d)       int i = 1          →  int sum = 0
          int sum = 0        →  int i = 0
```

Listing 3: Sample Matchings before Filtering for 2

```
int i = 1          →  int sum = 0
int sum = 0        →  int i = 0
i <= num           →  i <= num
sum = sum + i      →  sum += i
i = i + 1          →  i++
return sum         →  return sum
```

Listing 4: Sample Matching after Filtering Listing 3

4.5 Syntactic Limitations

The current implementation has a few limitations on the input files. The input must compile and follow the following rules:

- The source code must not use lambda functions.
- The source code must not use switch statements.

Chapter 5

RESULTS

In this chapter Cloneless is compared against Simian and NICAD using small test cases and about 90 sample Java projects from introductory college students.

5.1 Test Cases

To evaluate Cloneless, a test set was created to determine how well it performs on known clones. The test set contains a variety of clones, ranging from Type I to Type IV. Each test is contained within a file that has two functions which are tested for clones. Table 1 shows the summarized results for the following test cases.

Exact Clones

This test case involves two sections of code that are the exact same. It is expected that textual comparison tools as well as graph-based tools are able detect clones of this nature.

Variable Names Changed

This test case involves methods that are the exact same but only the variable names have been changed. By changing variable names, textual-based tools that have no knowledge of the language are expected to fail. Since all tools that were tested have a basic understanding of the language, they were all able to detect this clone.

Table 1: Test Case Results, x means the tool detected a clone

#	Test Case	Simian	NICAD	Cloneless
1	Direct Clones	x	x	x
2	Variable Names Changed	x	x	x
3	Variable Types Changed	x	x	x
4	Post-Unary Increment vs. Addition		x	x
5	Changed Method Call	x	x	x
6	For-loop vs. While-loop		x	x
7	Two-step addition vs. One-step addition		x	x
8	Change Addition Order			x
9	Reverse Condition		x	x
10	Reverse Condition with Not		x	
11	Auto initialization vs. Manual initialization			x
12	Recursive vs. Iterative			
13	Independent vs. Dependent Declarations			
14	Not A Clone			x
15	Reorder Instructions		x	x
16	Dead Code Insertion			x

Variable Types Changed

This test case refers to changing the types of the returned values from 32-bit to 64-bit to see how clone detection tools fair. Changing a variable from float to double or vice-versa has little impact on the execution of the code except for edge cases where precision becomes an issue. Regardless, code segments with these minor changes should be marked as clones. All of the tools were able to identify clones of this kind.

Unary Post-Increment vs. Binary Addition Expression

The expressions $x++$ and $x = x + 1$ have the same functionality. Therefore, code clone tools should mark these as duplicates. These Type III clones often evade textual comparison tools, such as Simian, due to the underlying functionality of the expression. Cloneless was able to identify this clone by transforming the unary expression into a binary addition expression.

Method Call Change

Calling methods has been greatly impacted by the addition of lambda functions in Java 8. Two pieces of code that do the same thing but have two separate method calls can be refactored by replacing the method call with a lambda expression. Syntactic and graphical tools can be configured to ignore method calls as shown by this test case being discovered by all the tools tested.

Listing 1 shows an example of this type of clone. The only difference between the two functions are the calls to methods *doSomething*1 and *doSomething*2. These method calls can be transformed into lambda expressions and the methods can be combined into one as shown in Listing 1.

```
double test1(int[] arr){
  int sum=0;
  int count = 0;
  for(int i=0;i< 3;i++){
    sum=sum+arr[i];
    count=doSomething1(count);
  }
  return (sum*1.0)/count;
}
```

```
double test2(int[] arr){
  int sum = 0;
  int count = 0;
  for(int i=0;i< 3;i++){
    sum=sum+arr[i];
    count=doSomething2(count);
  }
  return (sum*1.0)/count;
}
```

Listing 1: Example of Method Call Clones

```
double test3(int[] arr, IntFunction<Integer> func){
  int sum = 0;
  int count = 0;
  for(int i=0;i< 3;i++){
    sum=sum+arr[i];
    count=func.apply(count);
  }
  return (sum*1.0)/count;
}
```

Listing 2: Refactored Function for Listing 1

For-Loop vs. While-Loop

Loops in Java share many of the same characteristics with one another, leading them to be high candidates for clones. For-loops and while-loops accomplish the same tasks

in two different ways. This test case evaluates whether the tools can identify these Type III clones. Simian is unable to detect such a clone, while Cloneless and NICAD are able to detect it by converting the for-loop into a while-loop.

Two-Step Addition vs. One-Step Addition

This test case considers complex expressions versus those split over multiple statements. For instance, within the loop of Listing 3, one code block uses two different assignments to add numbers together and the other uses one in lines 5 and 6. Mathematically, these operations produce the same result. Simian is unable to detect the similarity. Cloneless is able to detect the duplicate code; however, it did not use the mathematical similarity. Instead during the subgraph isomorphism stage, Cloneless identifies the clone without the second addition step because of the similarity in the initial expressions.

```
double test1(int[] arr) {
  int sum = 0;
  int count = 0;
  for(int i = 0; i < 3; i++) {
    sum = sum + arr[i] + 3;

    count++;
  }
  return (sum * 1.0) / count;
}
```

```
double test2(int[] arr) {
  int sum = 0;
  int count = 0;
  for(int i = 0; i < 3; i++) {
    sum = sum + arr[i];
    sum = sum + 3;
    count++;
  }
  return (sum * 1.0) / count;
}
```

Listing 3: Example of Two-Step Addition vs. One-Step Addition Clones

Change Addition Order

This test case is built off the commutative property. Changing the ordering of certain operations has no impact on the results of a mathematical expression. Therefore, clone detection tools should be able to recognize the ordering change. Both tools were configured to ignore identifiers to maximize the chances of detecting a clone. Surprisingly, NICAD did not detect this clone. Cloneless was able to detect this clone, while Simian was not.

Reverse Condition

The two expressions $i < 3$ and $3 > i$ are equivalent but written in the opposite order. This Type III clone requires the tool to have some knowledge about the operators expressed in Java. Cloneless is able to detect this kind of clone by checking the reverse of any binary expression to see if they are equal. Simian does not perform this analysis and therefore is unable to detect the clone.

Reverse Condition with Not

The two expressions $i < 3$ and $!(i >= 3)$ are equivalent expressions. Both Cloneless and Simian are unable to detect these clones purely on its own. Cloneless can be modified to catch this test case by performing more pre-processing of the code, putting it into an ideal form. However, if the settings are tweaked and the minimum compared graph size is changed, the code around these expressions can be marked as a duplicate, drawing attention to the region. NICAD was able to detect the clone even though it marks the different conditions as unique because it was still over the threshold.

Auto-Initialization vs Manual Initialization

Upon an identifier's declaration in Java, its value is initialized regardless of whether that value was specified by the programmer or not. For example, integers are automatically initialized to zero. Alternatively, the programmer can manually specify the value to be zero. In both instances, the value stored in the variable is zero and there is potential for code clones. Cloneless, unlike Simian and NICAD, was able to detect this clone. When creating the AST, if a variable was not initialized, then Cloneless adds the assignment expression to initialize it to its default value. This helps to illuminate more clones due to a programmer's style.

Recursive vs. Iterative Solutions

A classic Type IV test case is the iterative versus the recursive approach. These two implementations can have the same functionality but look drastically different. None of the tools were able to detect this type of duplicate as expected. This test requires a lot of interpretation and complex analysis to detect the clone.

Independent vs. Dependent Declarations

This test case was specifically defined to display a downside of Cloneless. Since Cloneless relies on a minimum size graph to compare, if the expressions are all independent of one another but perform the same operation (see Listing 4), the clone is not detected.

```
public double test1() {
    int num1 = 0;
    int num2 = num1;
    int num3 = num2;
    int num4 = num3;
    int num5 = num4;
    int num6 = num5;

    return num6;
}
```

```
public double test2() {
    int num1 = 0;
    int num2 = 0;
    int num3 = 0;
    int num4 = 0;
    int num5 = 0;
    int num6 = 0;

    return num6;
}
```

Listing 4: Independent vs. Dependent Declarations Test Case

Though these functions aren't particularly useful and some might not define them as clones, it shows a potential downside of using a graphical tool. None of the tools were able to detect this clone.

Not a Clone

Because Cloneless only possesses write-after-write dependencies in the PDG, the control dependence from a block's condition might not divide expressions in the PDG. Listing 5 shows a case where in the PDG the if-statement's condition is entirely detached from the connected subgraph containing $newNum$. This causes Cloneless to potentially misidentify clones, causing false positives. While this example could still be refactored, the changes would not be beneficial to a developer. Cloneless was the only tool to falsely identify a clone in this case.

```
int foo(int in, int[] arr) {
  int newNum = 4;
  for(int num : arr) {
    in += num;
  }
  if(in % 2 == 0) {
    newNum = 10;
  }
  return newNum;
}
```

(a) Function A

```
int bar() {
  int newNum = 4;
  newNum = 10;
  return newNum;
}
```

(b) Function B

Listing 5: Independent vs. Dependent Declarations Test Case

Reorder Instructions

One of the key features of a graph-based algorithm is to remove the ordering set by the developer. Switching two instructions can fool basic textual tools, while there is no difference for their graphical counterparts. Because of this, Cloneless was able to detect reordered instructions while Simian was not. NICAD was able to detect this clone due to its normalization.

Dead Code Insertion

Another key feature of using the PDG is ignoring dead code, or code that has been inserted into a program but has no impact on the program's functionality. For example, Listing 6 shows the base method in Function A and the method with dead code inserted in Function B. The cloned expressions in Function B are highlighted. The

variables *num*, *anotherNum*, and *finalNum* add no functionality into the method. By using the PDG, Cloneless is able to identify code that is not used and perform comparisons solely on the code that is used in the program. Simian and NICAD are easily fooled by adding in useless statements, while Cloneless is not.

```
double test1(int[] arr) {
  int sum = 0;
  int count = 0;

  for(int i = 0; i < 3; i++) {

    sum = sum + arr[i];
    count++;

  }
  return (sum * 1.0) / count;
}
```

(a) Function A

```
double test2(int[] arr) {
  int num = 4;
  int sum = 0;
  int count = 0;
  num = num + 12;
  num = num == 3 ? sum + 32
      : foo(arr[2] + "");
  for(int i = 0; i ¡ 3; i++) {
    num++;
    num--;
    sum = sum + arr[i];
    count++;
    int anotherNum =
        Integer.parseInt("43");
    anotherNum = anotherNum / num;
  }
  int finalNum = num % 4 == count
      ? num : 432;
  return (sum * 1.0) / count;
}
```

(b) Function B

Listing 6: Dead Code Insertion Test Case

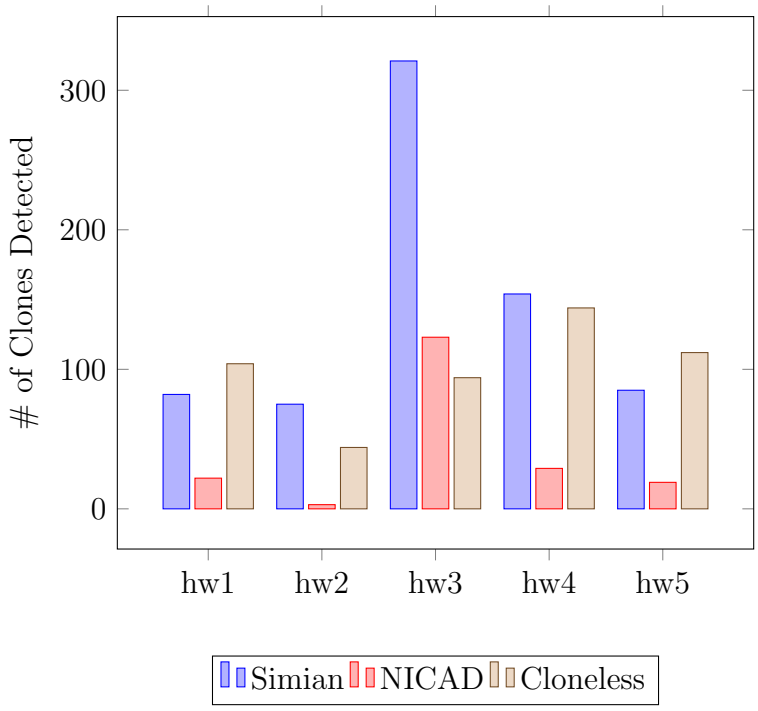


Figure 7: Number of Clones Detected on the Student Dataset for Each Tool

5.2 Sample Student Code

Further testing was performed on code gathered from an introductory Computer Science course. The projects have been trimmed down to eliminate lambda expressions and code that does not compile. The testing set consists of 1,183 files with about 54,000 lines of code.

Figure 7 presents the number of clones detected for each tool; Simian detected more clones than any of the other tools, and Cloneless on average reported more clones than NICAD. However, looking more in-depth into these clones, some reported clones are not what a developer would deem a clone. Additionally, some reported clones have no method of refactoring them into a single instance.

Taking a sampling of 20% of the submissions from each project, the reported clones

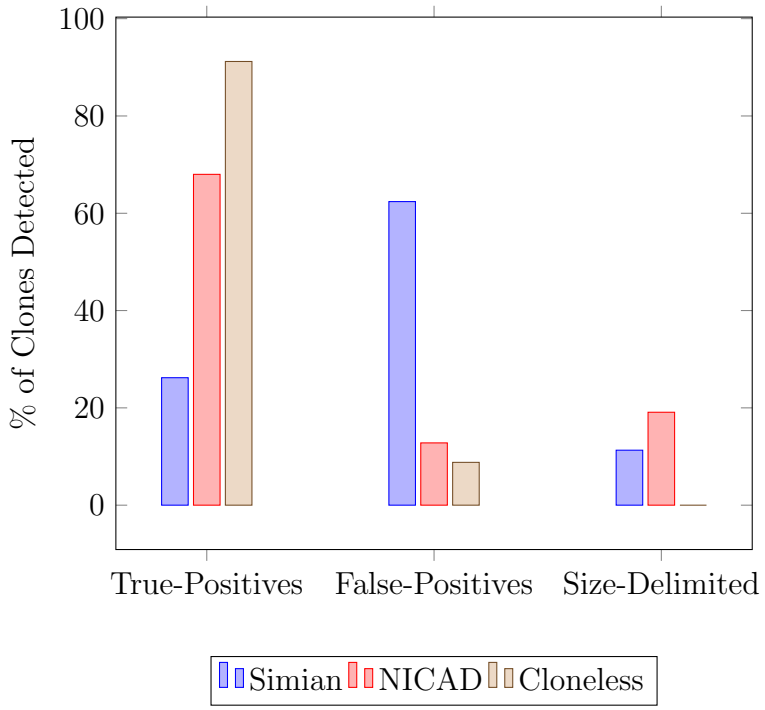


Figure 8: Types of Reported Clones on a 20% Sampling

were categorized as reported in Figure 8. The Figure shows three categories of clones: true-positives, false-positives, and size-delimited clones. True-positive and false-positive clones were defined through manual inspection of the reported clones. If a clones is not refactorable into a single instance, then it is a false-positive. Since, NICAD reports near-miss clones, it is difficult to define the code segments as false-positives. Size-delimited clones are potential clones reported by the other tools, but that do not create a PDG of sufficient size for Cloneless to detect. These clones were split into their own category because given different input parameters to Cloneless, the clones are detected.

Cloneless, unlike the others, reports clones in groupings of only two. This means that if three pieces of code are clones, then Cloneless reports these as six separate clones while the other tools report them as one. Since these code segments are still clones, they are not considered false-positives even though they may inflate the total count.

In Figure 8, however, these clones were reduced to count as only one clone to mimic the other tools as closely as possible.

While Simian reports a large number of clones, a high percentage are false-positives which may overwhelm the developer and hide real clones. Simian was prone to detecting clones for variable declarations, constructors, imports, and even detecting clones across methods. For example Listing 7 shows a clone that one may consider a false-positive. While this example could be refactored into an abstract class, doing so would reduce the clarity gained by using domain appropriate identifiers.

```
public class PolarPoint
{
   private double r;
   private double theta;

   public PolarPoint(
     double radius,
     double angle)
   {
      this.r = radius;
      this.theta = angle;
   }
   ...
}
```

(a) Snippet from Class A

```
public class CartesianPoint
{
   private double x;
   private double y;

   public CartesianPoint(
     double x_coord,
     double y_coord)
   {
      this.x = x_coord;
      this.y = y_coord;
   }
   ...
}
```

(b) Snippet Class B

Listing 7: Sample Detected Clone by Simian for Not Refactorable Clone

```
public interface RecursiveOONode
{
   String getString();
   public RecursiveOONode getNext();
   ROONode addToEnd(String element);
   ROONode add(int index, String element);
   RecursiveOONode remove(int index);
   String get(int index);
   int indexOf(String element);
   int size();
   StringList toLowerCase();
   StringList toUpperCase();
   StringList startsWith(String prefix);
   StringList hasSubstring(String substring);
}
```

(a) Snippet from Class A

```
public interface StringList
{
   void addToEnd(String element);
   void add(int index, String element);
   void remove(int index);
   String get(int index);
   int indexOf(String element);
   StringList toLowerCase();
   StringList toUpperCase();
   StringList startsWith(String prefix);
   StringList hasSubstring(String substring);
}
```

(b) Snippet Class B

Listing 8: Sample Detected Clone by Simian for Spanning Methods

Simian also has a tendency to report clones that span functions. Listing 8 shows a case where Simian checked the methods of the interfaces and detected that these two interfaces were clones. While they may have some of the same functionality, these interfaces may serve completely different purposes, and it may not make sense to combine like functions into another interface. Additionally, the functions for one of the interfaces returns objects for each operation, while the other may return nothing.

Cloneless detected some out of order clones that other tools did not due to its lack of reliance on the code's linear structure. Listing 9 shows an interesting segment of

```
...
strDate = scanner.nextLine();
amount = scanner.nextLine();
d = dateConverter(strDate);
dollars = strToDollar(amount);
cents = strToCents(amount);
...
```

(a) Code Segment A

```
...
dateStr = scanner.next();
date = dateConverter(dateStr);
amountStr = scanner.next();
dollarsPaid = strToDollar(amountStr);
centsPaid = strToCents(amountStr);
...
```

(b) Code Segment B

Listing 9: Sample Detected Clone by only Cloneless

a project where Cloneless was able to detect 24 similar lines of code while the other tools did not discover any duplicated lines. After analyzing these code segments, the blocks are quite similar and perform many of the same actions. Therefore, it is correct to classify them as clones.

Table 2: Time Results Over All Projects

Simian	NICAD	Cloneless
34.7s	45.3s	12m 54.8s

5.2.1 Running Times

Table 2 shows the running times of all the tools on the student programs dataset. As expected, Simian and NICAD performed better since they opt for a textual comparison approach. Cloneless runs significantly slower than the others due to its subgraph isomorphism algorithm.

5.2.2 Simian

While running the student code through Simian, it was apparent that Simian does not ignore standard Java conventions at the beginning of files, such as class declarations and constructors. These items should most likely not be reported as clones because the language requires them in order to compile. During testing, Simian was run using the *ignore identifiers* flag, causing it to report more clones than the other tools. By ignoring identifiers, some code snippets that perform operations with different identifiers that match the same pattern could be falsely detected as clones.

5.2.3 NICAD

While running the test cases, NICAD performed well due to the high similarity in the functions. If the functions are widely different with the same functionality, then it does not detect the clone. This is shown in the dead code insertion test case. By adding many useless instructions into the function, NICAD is unable to detect clones

because the function is not inflated to reach the duplicate threshold.

Since NICAD detects potential clones up to a threshold, they may not be refactored into a single instance and be deemed a clone. Listing 10 shows code that NICAD reports as a clone. These segments come from a comparator class. Due to their high structural similarity, NICAD sees that the code segments are comparable to one another and declares them a clone. However, this example is both size-delimited and a false-positive. There are 5 key expressions in the snippet of code, so it is not detected by the other tools. This case shows a drawback of using NICAD. The near-miss clones that the tool finds may not be actual clones, flooding the user with unnecessary information.

5.2.4 Cloneless

Cloneless was able to more accurately identify some clones than the other tools tested; however, it also required much more running time. Its slower performance is expected due to the complex operations performed on the graphs.

By comparing the code purely off the graphical format, Cloneless hid some clones that were detected by the other tools. Listing 11 shows a clone that was detected by the other tools but not Cloneless. Lines 1 to 5 are a clear code clone of lines 7 to 11. Cloneless was unable to detect this because it requires a minimum graph size threshold to be met before a code segment is considered for a clone. Additionally, the print statements are not relevant in the PDG due to the reduction in control dependencies, shrinking the size of the graph. Therefore, the PDG only had a vertex set of size 3 for each of the code clones, which is below the default threshold.

Another drawback of Cloneless is that sometimes the clones are difficult to map to the original code. Since the lines of code do not necessarily need to be close to each

```
if(e1.getDate().after(e2.getDate())) {
  return 1;
}
else if(e1.getDate().equals(e2.getDate())) {
  return 0;
}
else {
  return -1;
}
```

(a) Snippet from Class A

```
if(e1.getCents() > e2.getCents()) {
  return 1;
}
else if(e1.getCents() == e2.getCents()) {
  return 0;
}
else {
  return -1;
}
```

(b) Snippet Class B

Listing 10: False-Positive Clone Reported by NICAD

```
1  System.out.print("Enter x-coordinate: ");
2  double x1 = input.nextDouble();
3  System.out.print("Enter y-coordinate: ");
4  double y1 = input.nextDouble();
5  p1 = new CartesianPoint(x1, y1);
6  ...
7  System.out.print("Enter x-coordinate: ");
8  double x2 = input.nextDouble();
9  System.out.print("Enter y-coordinate: ");
10 double y2 = input.nextDouble();
11 p2 = new CartesianPoint(x2, y2);
```

Listing 11: Sample Clone Missed by Cloneless

other or in order, the range of the cloned area can be quite large. In some situations, the large range could reveal clones that are more hidden. However, it may also reveal clones that may not be refactorable for a developer.

Chapter 6

FUTURE WORK

In order to make this approach more usable in the future, it should be extended to encompass the full Java language. In its current state, it only supports a subset of Java, disregarding lambda functions and switch statements entirely. Additionally, small changes can be made to the formation of the AST, such as transforming switch statements and ternary expressions to if statements. Changing these structures into one standardized construct may illuminate more Type III clones. Furthermore, generating all possible subgraphs is a computationally intensive task, accounting for a large portion of the running time. Therefore, developing a more efficient algorithm would be very beneficial to the execution times. Finally, more advanced static code analysis methods, such as constant propagation, could be used to potentially reveal more clones.

Chapter 7

CONCLUSION

Code clones are segments of code with the same functionality that inflate the size of a code base, increasing maintenance costs. Cloneless is a tool that discovers clones through subgraph isomorphism on the PDG. While creating the PDG, Cloneless implements fewer control dependencies than a traditional PDG to remove constraints on the graph to help reveal more clones. It is able to detect some clones better than other tools due to its normalization of some structures and extraction of code into a graphical form. Since Cloneless has a much higher running time than the other tools, it is more practical to run Cloneless a few times a day while a textual-based approach runs more frequently. This ensures that many types of clones are detected and has minimal impact on developer productivity.

BIBLIOGRAPHY

[1] Simscan. *Last accessed November*, 2008. Available at `https://web.archive.org/web/20080308002006/http://blue-edge.bg/download.html`.

[2] I. Baxter, A. Yahin, and L. Moura. Clone detection using abstract syntax trees. *Proceedings. International Conference on Software Maintenance*, 1998.

[3] P. Bulychev and M. Minea. Duplicate code detection using anti-unification. *Spring Young Researchers Colloquium on Software Engineering*, 2008.

[4] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, pages 149–159, 2001.

[5] D. Grune, H. Bal, C. Jacobs, and K. Langendoen. *Modern compiler design*. John Wiley Sons, 2000.

[6] S. Harris. Simian - similarity analyser. Available at `https://www.harukizaemon.com/simian/`.

[7] C. James. Txl - a language for programming language tools and applications, May 2006.

[8] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. *29th International Conference on Software Engineering (ICSE07)*, 2007.

[9] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic

token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654670, 2002.

[10] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. *Static Analysis Lecture Notes in Computer Science*, pages 40–56, 2001.

[11] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. *Static Analysis Lecture Notes in Computer Science*, page 4056, 2001.

[12] J. Krinke. Identifying similar code with program dependence graphs. *Proceedings Eighth Working Conference on Reverse Engineering*, 2004.

[13] K. Ly. Normalizer. Master's thesis, California Polytechnic State University San Luis Obispo, Mar 2017.

[14] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 2010.

[15] T. Parr, S. Harwell, and K. Fisher. Adaptive ll(*) parsing. *ACM SIGPLAN Notices*, 49(10):579598, 2014.

[16] D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Science Direct Information and Software Technology*, pages 1165–1199, 2013.

[17] C. Roy, J. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, May 2009.

[18] C. K. Roy. Detection and analysis of near-miss software clones. *Queens University*, Aug 2009.

[19] S. Sharma and P. Mehta. To enhance type 4 clone detection in clone testing. *International Journal of Computer Science and Information Technologies*, 7:967–971, 2016.

[20] H. M. Sneed. Estimating the costs of software maintenance tasks. *Proceedings of International Conference on Software Maintenance*, 1995.