Sisyphus - Crawling up the Library to Prevent Code Duplication

Research Project

Joey Eremondi
The University of British Columbia
jeremond@cs.ubc.ca

Fabian Ruffy
The University of British Columbia
fruffy@cs.ubc.ca

Itrat Akhter
The University of British Columbia
iaakhter@cs.ubc.ca

ABSTRACT

When learning a new language, library, or framework, programmers are frequently unaware of all the methods that are available to them. As result, they will often write code that duplicates functionality of existing methods and may be less safe, efficient, or robust.

We present Sisyphus, a tool that statically analyzes Java source code, detects functions that are already provided by Java standard libraries and recommends them to the programmer. This tool will help the programmer explore all the capabilities of Java as they learn it, enabling them to write cleaner code.

Sisyphus is based on a novel application of clone detection in recommending Java library functions. We investigate two different categories of clone detection methods - semantic and syntactic. By using it to compare computer science and engineering students' implementation of multiple Java Math and Arrays methods with the methods in the *OracleJDK*1.8.0_121 library, we are able to evaluate the accuracy and usefulness of our tool. Evaluation results reveal that Sisyphus is able to detect approximately 40% of duplicates, with a 25% false positive rate.

CCS CONCEPTS

• **Applied computing** → *Computer-assisted instruction*;

KEYWORDS

clone-detection, linters, software-engineering, computer-science-education

1 INTRODUCTION

Modern libraries and frameworks provide a wealth of methods that can perform common tasks, and can greatly reduce the amount of code which a programmer needs to write for a project. However, because these libraries are often large, newcomers to a language or framework may not be aware of all of the functions provided for them. As a result, they may attempt to implement these methods themselves, increasing the size of their task, and possibly introducing more bugs.

Our tool Sisyphus analyzes Java source code, detects methods that are already implemented by Java standard libraries and recommends them to the user. It parses both the source code of the Java libraries being considered, and the code written by the user, constructing an abstract syntax tree (AST) for each method. The trees are transformed to remove superficial differences and make them more comparable with each other. Abstract syntax trees of user implemented methods are compared with those of Oracle JDK1.8.0_121 library methods using a clone detection algorithm. If the algorithm decides that there exist similar methods, the relevant library methods are recommended to the user. We talk about related work in Section 2 and give a more detailed description of our tool in Section 3. In Section 4, we elaborate on how we evaluated our tool, and in Sections 5 and 6, we discuss the results and their implications. We list some threats to validity in section 7 and then finally we conclude the paper with future work and conclusion in sections 8 and 9 respectively.

The main contributions of this work are:

- Presenting a novel application of clone detection, in recommending standard library functions to beginner coders
- An adaption of existing clone detection methods to this use-case
- An evaluation of the suitability of different clone detection methods to this use-case

2 RELATED WORK

2.1 Clone Detection

Clone detection has been quite a popular area of research in software engineering. Traditionally, clone detection techniques have mostly catered to detecting code fragments that have been copied from some other parts of the software. According to Juergens et al., 2009 [5] clones can lead to difficulties in maintaining the code and cause unexpected behavior in software. In this work we incorporated clone detection techniques in a different context - to identify Java library methods that can replace the user's implementation of the same methods. Johnson, 1993 [4] identified textual clones by using an incremental hash function and identifying sequences of line with the same hash values as clones. Komondoor and Horwitz, 2001 [9] detect clones by finding isomorphic program dependency subgraphs. We incorporated clone detection techniques in Sisyphus that are both syntactic and semantic. A comprehensive overview of clone-detection methods can be found in [14, 15].

2.1.1 Code Representation. Clone detection algorithms can represent code in various ways, depending on the specific use case. These approaches are by no means mutually exclusive and are often used in combination to achieve the maximum possible accuracy.

Plain Text. Text-based parsing splits code into simple text fragments of arbitrary granularity which may be stripped of whitespace and standardized into a pretty-printed form. The output can be analyzed using efficient string-matching algorithms which are commonly applied in plagiarism detection. These algorithms include Rabin-Karp [7] or Knuth-Morris-Pratt [8]. Plain text analysis is fast and simple but falls short in detection of code that is heterogeneous but semantically identical.

Tokens. Token parsers generate a more fine-grained view of the code by performing lexical analysis. A grammar is fed into the analyzer which will generate a list of classifications of the individual code elements (e.g., "+" will be considered an "addition operator"). As opposed to plain text analysis, the representation of the method is generalized and abstracted from the specific implementation. Tokens may also be used to generate a suffix tree which aids in identifying common token patterns in the code. This allows for powerful analysis, most prominently utilized in the tool CCFinder. [6]

Abstract Syntax Trees. The Abstract Syntax Tree is a very common method in code analysis. Albeit slow, they accurately represent the code hierarchy and are capable of storing arbitrary information related to the source code. A syntax tree root may either be the entire class or an individual method in the compilation unit, which itself may be decomposed into declarations, expressions, statements, or any other possible syntactic category. One major advantage of ASTs over tokens and plain text is the child-parent relationship of nodes. For example, a method declaration may contain If-statements, which in turn are composed of a conditional-block, a then-block, and an optional else-block.

On the basis of an AST it is possible to restructure and generalize code, as well as generate many different types of graph structures to analyze the program. Common graph types include the data and control flow graphs, discussed in Section 3.3.

Metrics. A variant of the abstract syntax tree analysis is measuring a vector of features and characteristics as abstraction of the code. A metrics parser traverses the previously acquired AST to store and classify information. This information may include the number of types, declarations, control flow operations, or even subgraphs.

2.2 Control and Data-flow analysis

Control-flow and Data-flow graphs [11] are a staple of program analysis, and have been used in many compiler optimizations. The program dependence graph (PDG) [2] combines a control-dependency graph (an acyclic representation of control flow) and the data-flow graph into a single multigraph, which can be analyzed. Originally intended for applications in optimization and slicing, the PDG has been used to perform semantics-aware clone detection [9, 10]. We used the PDG in a similar fashion.

3 DESCRIPTION OF THE TOOL

Our tool has been designed to be highly modular with support for multiple parsing and clone detection techniques. It is composed of several exchangeable and extensible building blocks (see Figure 1). This design allows us to easily switch techniques or compare results of different parsers for experimentation, comparison and optimization purposes.

We chose Java as primary tool language for several reasons. First, various parsing and code analysis libraries already exist which we could potentially integrate in our tool. Second, the OOP paradigm provides the advantage of inherent categorization of syntax tree elements which we can leverage for efficient parsing and normalization. Third, Java is a common business development language heavily relying on IDE use. As we intend for our tool to ultimately be integrated into a text editor, choosing Java will simplify the integration process into Eclipse or IntelliJ.

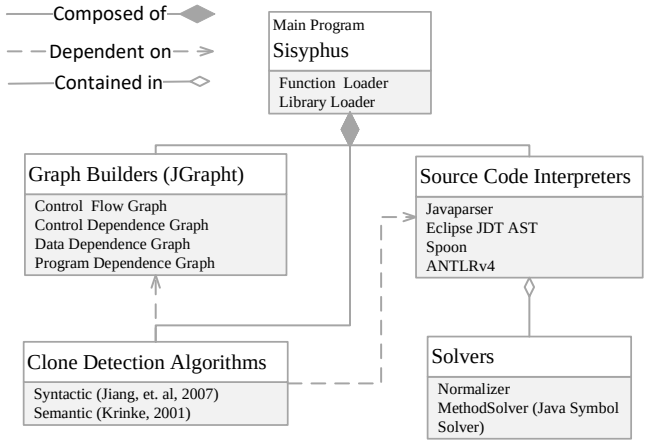

Figure 1: The modular design in Sisyphus.

The tool itself currently consists of around 4000 lines of code and utilizes three different open-source libraries. The input is a compilation unit (CU) which may either be a single method declaration, a full class file, or entire folder. The CU is parsed by a syntax parser which generates an abstract syntax tree (AST) and normalizes the CU into a near-canonical form. As an additional feature, the actual method being called at a given point may be found, to provide a more expressive graph structure. The canonical form of the method is stored in a method data structure and can be reused to generate various graph types for clone detection (see Figure 2).

We utilized two different clone detection techniques, which we drew from Jiang et. al, 2007 [3] and Krinke, 2001 [10]. After computing the corresponding graphs our program will apply the detection algorithm to a database containing previously normalized method declarations of an arbitrary library. In our current implementation we are matching against a collection of the Oracle JDK1.8.0_121 source files, which are parsed in parallel and persistently stored. The output of the program is a collection of all potential matches; we select the match with the highest confidence.

Detailed description of specific parts of our tool are given in the following sections.

3.1 Code Representation

Each clone-detection representation has its advantages and drawbacks in terms of accuracy, performance, and coverage. As we decided to prioritize accuracy and coverage, we opted for an AST parsing framework. Our decision was based on the fact that a function could be implemented in many different ways. The only way to approximate the different variations of the functionally same implementation is to capture high level structure. Abstract syntax trees can paint a comprehensive picture and enable the use of perform syntactic and semantic clone detection.

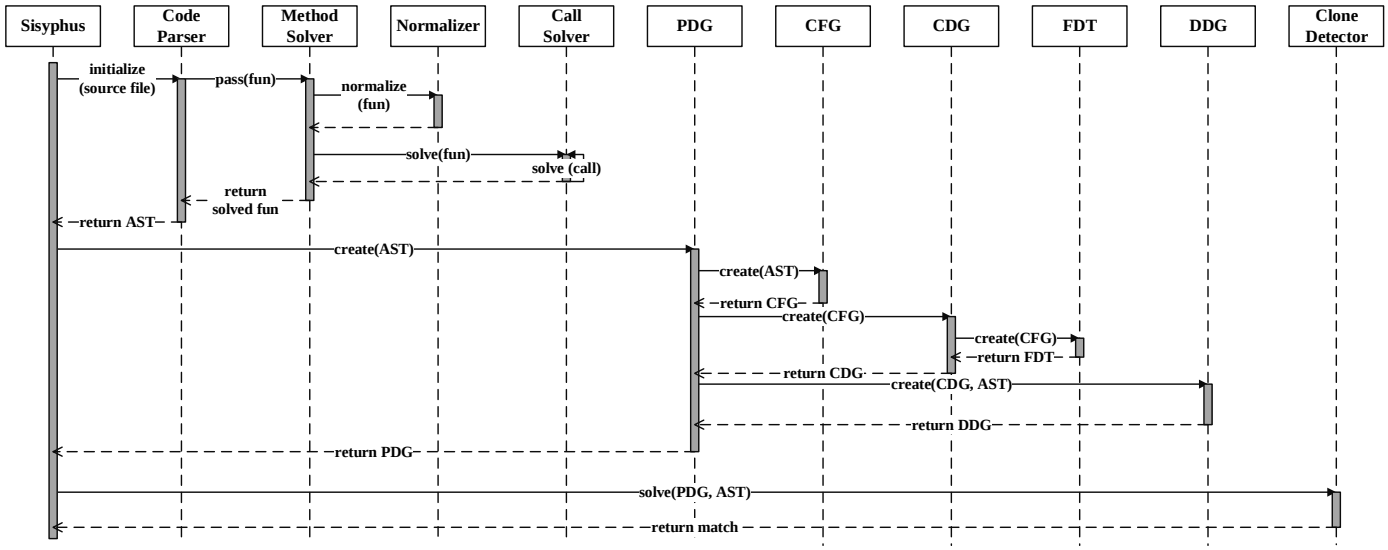

Figure 2: The typical program flow in Sisyphus.

To parse Java, there was no need to develop our own framework, since in this space, a wealth of potential solutions is available. Options we considered were Javaparser [18], the default Eclipse JDT AST parser, Spoon [13], and ANTLRv4 [12]. Despite the lack of language flexibility we went with Javaparser as our parsing tool of choice. It is a lean and extensible library to which we aim to contribute back to. In addition, it provides the Java-Symbol-Solver as extension which offers interesting capabilities to dereference function calls in static text based files. It is able to find the corresponding declaration without the use of reflection. After feeding Javaparser a file or a list of files, it will generate a compilation unit containing a syntax tree for the corresponding input. We segment the compilation unit into method declarations which we then process further.

3.2 Standardizing Methods

In order to achieve a more accurate matching behavior and a closer approximation to functionally similar code we perform several transformations and analysis on the generated method declarations.

3.2.1 Normalizing Individual Methods. We wanted to identify code pieces which are semantically similar, even if they are superficially different. To achieve this, we ran code through several *normalizers*, transformers which attempt to put code in a standard (normal) form.

The main normalizers used in the tool are:

- A variable renaming normalizer, which renames parameters and local variables solely based on their type and the order in which they are declared.
- A loop normalizer, which desugars For, Do-While and For-each loops into While loops.
- A Block-merging normalizer, which takes adjacent blocks of statements and merges them into a single block, ensuring no naming conflicts occur. This is particularly useful for cleaning up after the loop normalizer.

Pseudocode for the renaming normalizer is found in Algorithm 1. An example of code before and after normalization can be seen in Listing 1.

Algorithm 1 Variable Renaming

```
1:  procedure RENAME(node, env, parentDecls)
2:      if node is a variable and env contains node.name then
3:          node.name ← lookup(node.name, env)
4:      for each variable x declared by node do
5:          i ← highest index of string(type(x)) in env
6:          x′ ← "_" +string(type(x))+ "_" +string(i + 1)
7:          insert(env, (x, x′))
8:          insert(declsForParent, (x, x′))
9:      childDecls ← {}
10:     newEnv ← env
11:     for child ∈ children(node) do
12:         RENAME(child, newEnv, childDecls)
13:         newEnv ← newEnv ∪ childDecls
14:     if node is not a block with its own scope then
15:         declsForParent ← declsForParent ∪ childDecls
```

3.2.2 Resolving Method Calls. In many cases, methods are a sequence of nested operations consisting of further method calls. Typical syntax tree parsers are not capable of performing a deep inspection, so the underlying control flow and information contained in the method call are masked. This may impact the accuracy of clone detection. as well as increase the number of false negatives. Therefore, we utilized the Java-Symbol-Solver library to resolve

and integrate method calls into our current AST to provide an encompassing view. The Java-Symbol-Solver is capable of traversing a reference library and returning the corresponding declaration object of a method call. This object contains the body of the declaration, a feature which is not supported by conventional parsers as they only provide the signature of the method.

We built an extension which traverses all method calls in a declaration, removes the call, and inserts a syntactically correct version of the call AST back into the original body. As the acquired body may contain subsequent method calls the tool iterates over the body until all methods are resolved or a previously specified threshold is hit. This threshold also serves to mitigate the effect of mutually dependent loops in recursion. Early experiments showed that the amount of true positives as well as overall matches increased. However, as opposed to Javaparser, Java-Symbol-Solver is experimental and not yet mature. The tool has initially not been designed to solve the entire JDK source library. As the scope and comprehensiveness of Sisyphus increased we were facing more and more bugs and problematic behavior related to the tool. We are currently providing continuous feedback to the developers in the hopes of making the tool more robust in the future. As a result, we have decided to not include the symbol solver in our evaluation until the problematic aspects are fixed. Nonetheless, it remains a key feature of our program.

Listing 1: Code before and after normalization. Parameters and local variables are renamed in a deterministic way, and for-loops are unrolled into while loops.

```
public static double power(double val1, double val2) {
  double accum = 1;
  if (val2 >= 0) {
    for (int i = 0; i < val2; i++) {
        accum *= val1;
    }
    } else {
        for (int i = 0; i > val2; i--) {
            accum /= val1;
        }
    }
  return accum;
  }

public static double powerNormalized(
  double _param_double_0,
  double _param_double_1) {
  double _var_double_0 = 1;
  if (_param_double_1 >= 0) {
      int _var_int_0 = 0;
      while (_var_int_0 < _param_double_1) {
        _var_double_0 *= _param_double_0;
          _var_int_0++;
      }
  } else {
      int _var_int_0 = 0;
      while (_var_int_0 > _param_double_1) {
          _var_double_0 /= _param_double_0;
          _var_int_0--;
      }
  }
  return _var_double_0;
}
```

3.3 Building Graphs

In order to match programs that are similar but not syntactically identical, we wish to abstract away from its syntax, into a representation that is aware of its semantics. One way to do this is using dependence graphs. The program is broken into blocks, and the dependencies between those blocks are expressed as directed edges.

Our semantic clone detection algorithm is based on the matching of the program dependence graph of each method. The program dependence graph requires the construction of the control flow graph, dominator tree, control dependence graph and data dependence graph. To store and operate on graphs we used the open source library JGraphT [1]. In addition to providing a well tested set of common graph structures and algorithms, JGraphT encapsulates the objects that are held in the graph. Clone detection algorithms do not have to be account for the semantics of the parser and are able to operate on the JGraphT graph only.

3.3.1 Control Flow Graph. The first and most basic graph is the control flow graph (CFG), it is the foundation to all other graphs which will be generated in Sisyphus. CFGs describe all the possible paths a process may take when running through a program. It does not necessarily record state or nature of the edge in the control flow. For example, an if-statement has an edge to its then and else branches, but the edges are not marked with true or false, and both edges are present, even if one branch is never reached.

Although control flow graphs are well defined they can differ in terms of concrete implementation or personal customization. A CFG could either be a directed and acyclic graph (DAGs) or a pseudo graph which contains cycles as well as multiple edges. While- and for-loops add cycles to the graph, which are necessary for the construction of dependence graphs, but may turn out to be undesirable in later operations. In our implementation, we added a back-edge to facilitate the detection of a potential cycle.

3.3.2 Dominator Tree. In the control flow graph, a node B is "dominated" by a node A if and only if every path to B needs to pass A. In addition, A immediately dominates B if no node after A dominates B. As this is a unique property for every node, it is possible to define a forward dominator tree (FDT). For example, in Figure 3, $val < 0$ immediately dominates $val = -val$. However, $val < 0$ does not immediately dominate $return\ val$ as it is possible that $return\ val$ is accessed over $val = -val$. The only possible immediate dominator is thus $Entry$. The corresponding graph has the same representation as (3).

Dominator trees alone are not expressive enough for clone detection, but are an intermediate computation for dependence graphs. Our implementation leverages an efficient open-source solution by Martin Schoeberl [17].

3.3.3 Control Dependence Graph. The control dependence graph is an abstract representation of code dependencies in the program. Drawing from the forward dominator tree, a control dependence graph categorizes nodes into immediate dominators and dependent children. The only predecessor of dominators is the entry node, while the children are attached to the corresponding dominator (See Figure 3). This structure facilitates the quick identification of similar dependent code in the system which indicate a method clone. A CDG is created by merging the FDT with the corresponding control

```
1 public int absoluteValue(int val) {
2   if (val < 0) {
3         val = -val;
4   }
5   return val;
6 }
```

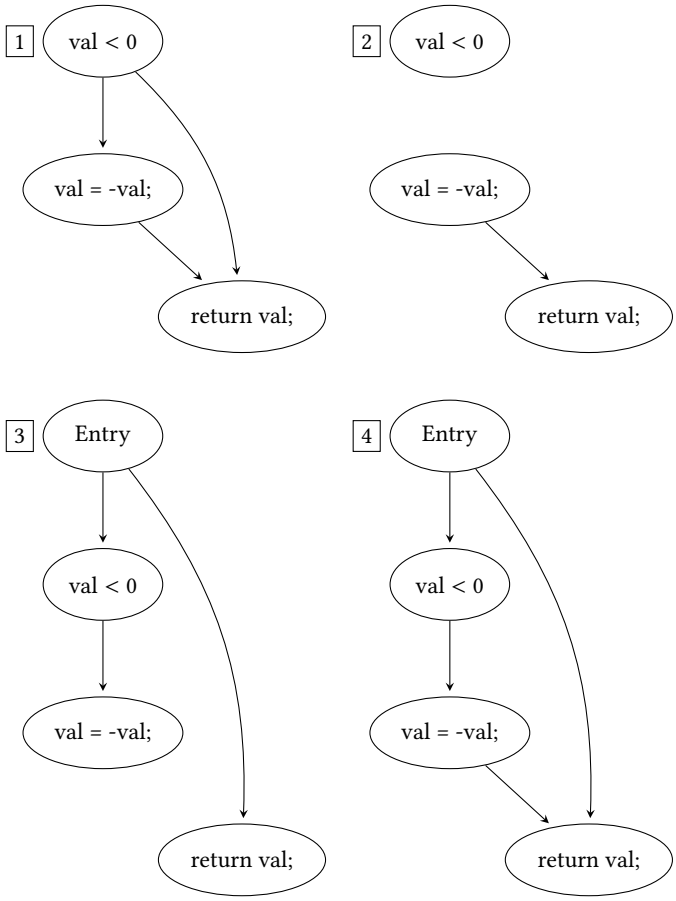

Figure 3: Graphs of an implementation of Math.abs method. (1): Control Flow Graph. (2): Data Dependence Graph. (3): Control Dependence Graph. (4): Program Dependence Graph

flow graph. If a node does not have any children in the forward dominator tree, it is considered to be dependent. An edge between child and its dominator is added. Algorithm 2 contains a simplified description of the merge process.

3.3.4 Data Dependence Graph. In addition to characterizing control, it is useful to distinguish data dependencies of a method: which parameters might possibly affect which values.

To compute the data dependency graph, we used a work-list algorithm to iteratively find a fixed-point of dataflow equations for reaching definitions [11]. This tells us which assignments of a variable could possible have caused its value at a given use.

Equations for dataflow are given in Figure 4. The equations for *Entry* and *Exit* are mutually recursive, so an iterative worklist

Algorithm 2 Control Dependence Graph

```
1:  procedure MERGE
2:      fdt ← the forward dominator tree
3:      cfg ← the control flow graph
4:      cdg ← init(entryNode)
5:      cdg ← addVertex(vertexSet(cfg))
6:      for node in vertexSet(cfg) do
7:          if (fdtOutedgesOf(node) = 0) then
8:              cdg ← addEdge(dominator(node), node)
9:          else
10:             cdg ← addEdge(entryNode, node)
```

$$
\begin{aligned}
Kill(n) &= \{(x, n') \mid n = (x = e), e \in \textsc{Expr}, n' \in \textsc{Block}\} \\
Gen(n) &= \{(x, n) \mid n = (x = e), e \in \textsc{Expr}\} \\
Entry(n) &= \bigcup \{Exit(n') \mid (n', n) \in CFG \\
&\quad \cup \{(x, ?) \mid x \text{ free in method, } n \text{ initial}\} \\
Exit(n) &= (Entry(n) \setminus Kill(n)) \cup Gen(n) \\
DDG &= \{(n, n') \mid (x, n') \in Entry(n), x \in \textsc{FreeVars}(n)\}
\end{aligned}
$$

Figure 4: Data flow equations for reaching definitions [11]. The data dependency graph between nodes contains an edge from each block defining a variable to each block using that variable reached by the definition.

algorithm can be used to find sets satisfying the equations. Each set starts empty, and each *Exit* set begins as the *Gen* set for each block. All nodes begin on the worklist. We repeatedly pop blocks off the worklist, calculate *Entry* by joining all *Exits* from blocks who can reach this block in the CFG, then finally calculate our Exit from our entry. If this expanded our *Exit* set, then all nodes that the block had an edge to in the CFG are added to the worklist. The process is repeated until the worklist is empty.

The result of the worklist algorithm tells us which assignment statements of a variable can possibly reach each block. To translate this into a graph of data dependencies between blocks, we created a graph where each assignment has an edge to each reached use of the assigned variable.

As is common in program analysis, this analysis is conservative, showing which definitions could possibly reach use of a variable, possibly including more than what will actually happen dynamically. For example, the definitions that reach the block after an if-else branching statement will be the union of all branches, regardless of which branches are actually taken.

Since our dataflow graph is used only for measuring code similarity, as opposed to actual program optimization or transformation, we took several liberties with our analysis, such as ignoring side-effects on parameters of method calls, and treating each struct field as a unique variable. Likewise, we performed no alias analysis, so if two references point to the same object, a change to one is not marked as a change to the other.

A very simple example of a data-flow graph can be seen in Figure 3. The assignment to *val* reaches the return statement, as well as whatever value it holds at the start of the method (not pictured).

3.3.5 Program Dependence Graph. The program dependence graph [2] combines the control-dependence and data-dependence graph into a single graph as seen in 3. This graph is semantically aware, yet abstracts away many details of the program, making it well suited for comparing method bodies.

3.4 Clone Detection

We incorporated syntactic and semantic clone detection in Sisyphus to capture the possible variations in the user implementations of the same method. The AST representation of the source code was used to detect syntactic clones. Semantically detecting all clones is an undecidable problem. However, approximations can still be made by using a graph similarity algorithm on the PDG constructed from the AST [15]. Note that both clone detection techniques focused only on the body of the method in our tool. In order to reduce false positives, before we applied clone detection on any two methods we asserted that the return types and the types of the parameters of the methods were the same. If they were not, we did not consider the methods to be a match.

3.4.1 Abstract Syntax Tree Matching. Our AST matching algorithm is a variation on methods used by Jiang et al., 2007 [3] in their clone detection tool Deckard. The algorithm is based on characterizing the AST's with characteristic numerical vectors in the Euclidean space. The characteristic vector $[c_1, ...c_n]$ represents a point where each value *ci* in the vector is the number of occurrences of a certain kind of nodes in the AST. For example, consider an abstract syntax subtree which has an *if statement* node as a parent and a *condition* node and a *then assignment* node as children. The characteristic vector of each of these nodes will be of the type $[c_1, c_2, c_3, c_4, c_5, \ldots c_n]$ where c_1, c_2 and c_3 hold the number of occurrences of *the if statement* node, the *condition* node and the *then assignment* node respectively in the subtree starting from the node. Values $c_4, c_5 ... c_n$ hold the number of occurrences of other kinds of nodes in the subtree.

Suppose that there are no other *if statements*, *condition* or *then assignment* nodes in the subtree being considered. The characteristic vector of the *then assignment* node will be $[0, 0, 1, c_4, c_5, \ldots c_n]$. Similarly, the characteristic vector of the *condition* node will be $[0, 1, 0, c_4, c_5, \ldots c_n]$ and the characteristic vector of the parent *if statement* node will be $[1, 1, 1, c_4, c_5, \ldots c_n]$. Thus, the characteristic vector of an AST is generated by summing the vectors of the children with the vector of the parent as we do a post order traversal of the tree. When two methods are compared, the $l1$ distance between the characteristic vectors of the corresponding AST's are calculated. If the distance is below a certain threshold, the methods fall under the category of probable matches. For every user implemented method, a list of probable matches from the Java library will be returned. The method from the list with the least distance is considered to be the actual match. If there are no probable matches, then the user's method does not have a match from the Java library.

3.4.2 Program Dependence Graph Matching. Methods that have similar program dependence graphs are considered to be clones of each other. The difficulty arises in defining "similarity" between two graphs. The most obvious solution would be to relate similarity to isomorphism. In isomorphic graphs, every edge can be bijectively matched to an edge in the other graph, and the edge attributes and corresponding vertices are the same. However, determining if two graphs are isomorphic has no known polynomial algorithm, and determining if one graph is isomorphic to a subgraph of another is an *NP*-hard problem.

Instead, we used the graph similarity measure defined by Krinke et al., 2001 [10] to compare program dependence graphs. Two graphs G and G' are considered to be similar if for every path that starts from vertex v_0 and ends with vertex v_n in G, there exists a path from v'_0 to v'_n in G' and the edges and vertices have the same attributes in both the paths. This algorithm was implemented by constructing a maximal similar graph M. We started with vertex v_0 from graph G and vertex v'_0 from graph G'. If the vertices were equal, we added v_0 to M. For each child c of v_0, if there was a child c' of v'_0 such that c equaled c', we added c to M. We continued doing this until we reached the end vertex v_n, v'_n of each graph. If we were unable to reach the end vertex of either graph, then the graphs were not considered to be similar and hence the methods were not a match. Otherwise the methods were considered to be probable matches. Similar to the abstract syntax graph matching, for every user implemented method, a list of probable matches from the Java library will be returned. Out of those matches, the one with which the highest number of paths is shared is chosen as the actual match. For our tool, we considered the *Entry Statement* of each program dependence graph to be the start vertex v_0 and any leaf vertex of program dependence graph to be the end vertex v_n.

4 EVALUATION METHODOLOGY

There are numerous libraries in Java. In order to test our tool in the context of the simplest case, we evaluated it on two Java static libraries - *Math* and *Arrays*. We extracted 12 methods from these libraries that a novice Java coder would most likely attempt to implement (for example, Math.abs or Arrays.copyOf). We created a template containing the signatures of these 12 methods and asked participants to implement the methods in the template. We sent the template to and received the completed template from each participant electronically.

Each method signature in the template was accompanied with Javadoc style documentation where details about the parameters and return type of the method were included. A basic description of what the method should do was also included. We took care not to mention any of the corner cases that the Java library method takes care of, as our goal was to simulate how a novice Java coder would implement these methods.

There were 12 participants in our study. All of our participants were computer science and engineering students of varying degrees of experience in Java ranging from using Java for the first time to being completely adept at Java. Two of the participants were undergraduate students and the rest were graduate students.

We accumulated all the participants' implementations in one source file and compared each of them with all the methods in

Matching Type	TP	FP	TN	FN
ASTControl	3	0	3506832	141
AST	60	60	3506772	84
PDG	58	19	3506813	86

Table 1: Summary of true positives, false positives, true negatives and false negatives returned by our tool

Algorithm	Matches	TP/Matches	TP/Ideal Matches
ASTControl	3	100%	2.08%
AST	120	50%	41.67%
PDG	77	75.32%	40.28%

Table 2: Performance of tool comparing participants' code with Java jre library. ((%)TP/Matches) is the percentage of true positives in relation to the total number of matches returned by our tool. ((%)TP/Ideal Matches) is the percentage of true positives in relation to the total number of matches we were expecting

.

the Oracle JDK1.8.0_121 library. We used this extensive library for comparison to get an approximation of the false positives incurred by our tool and to increase confidence in our results. We used both our AST and PDG matching algorithms to return the list of matches between the user implemented methods and the Java library methods. Our goal for the tool was as many true positives as possible and as few false positives as possible, because we hypothesize that too many inaccurate matches would annoy the user, but a false negative will likely go unnoticed. The data that we collected reflects this intention of our evaluation.

5 EVALUATION RESULTS

Our tool returns a list of matches between the participants' code and the Oracle JDK1.8.0_121 library for both the syntactic (AST) clone detection and semantic (PDG) clone detection. We have also incorporated a control clone detection (ASTControl) where we compare methods by matching the abstract syntax trees directly. A summary of the raw data - number of true positives, false positives, true negatives and false negatives returned by all the different types of clone detection are included in Table 1.

Table 2 gives a more intuitive analysis of the performance of our tool. ((%)TP/Matches) gives a measure of false positives. The higher this number is the lower the proportion of false positives. We want this value to be as high as possible because too many false positives might annoy the user. The PDG clone detection performed the best out of all the clone detection incorporated in our tool in regards to returning low false positives, with ((%)TP/Matches) of 75.32%. The AST comparison was able to correctly match more of the participants' code with the library methods than PDG, but at the cost of a higher false-positive rate. As we can see from Listing 2, our tool is able to match methods that are implemented quite differently by the participants to the correct Java library method.

6 DISCUSSION

6.1 Exact matching

While there were no false positives with exact matching, it matched very few true positives. It seems then that for a recommender to be successful, it must use approximate methods.

Listing 2: Examples of methods that are correctly matched with the Java library method (in this case Arrays.copyOfRange) by our tool using the PDG clone detection

```
1
2  public static int[] returnCopyRangeJ(
3    int[] array,int startIndex,int endIndex) {
4    int[] newArr = new int[endIndex-startIndex];
5    for (int i = 0; i < newArr.length; i++) {
6          newArr[i] = (startIndex + i
7     < array.length) ?
8     array[startIndex + i] : 0;
9    }
10   return newArr;
11 }
12
13 public static int[] returnCopyRangeAJ(
14   int[] array, int startIndex, int endIndex) {
15   //YOUR CODE HERE
16   // determine the length of the new array
17   if (endIndex >= array.length) {
18         endIndex = array.length;
19   }
20   assert (startIndex <= array.length);
21   int newLength = endIndex - startIndex;
22
23   int[] newArray = new int[newLength];
24
25   int j = 0;
26   for (int i = startIndex; i < endIndex; i++) {
27         newArray[j] = array[i];
28         j++;
29   }
30   return newArray;
31 }
```

6.2 Critical Features

Our results are generally positive, but show a moderate proportion of false positives.

Clone detection was originally meant for large code-bases, and specialized in detecting code that had been copied, and possibly modified. While semantic methods exist, it is no surprise that it is best at recognizing methods that are structurally similar.

In our results, many false positives came from structurally similar code. Both syntactic and semantic clone detection had difficulty distinguishing *maximum* and *minimum* functions. Additionally, methods implemented in a similar style, such as *maximum* and *sum* functions that both iterate through an array, led to false positives.

In order to reduce the number of false positives, future method recommenders may need to develop a model of which parts of the code are structural boilerplate, and which are critical to the semantics of the operation performed, so that the difference between < and > is seen as significant in *max* and *min* functions.

6.3 Implementation Style

As can be seen in our results, more than half of actual matches were missed in all methods. One factor in this is the style in which Java library functions are implemented, often taking advantage of more

obscure language features in order to gain performance. Because of this, semantically identical functions end up looking syntactically very different, beyond what basic normalizers can convert, and even having radically different control and data flow structures. While more advanced normalizers could provide improvement, another approach would be to make a reference implementation of important standard-library functions against which to perform the comparison, written in a more straightforward style.

7 THREATS TO VALIDITY

7.1 Internal Threats To Validity

Our implementation of clone detection algorithms and program dependence graphs may contain defects. Participants' implementation of Java methods may contain defects. Additionally, our program analysis components took several liberties. For example, the participants implemented only the methods which were already implemented in the Java library. We did not evaluate false positives in the context of methods that are not in the Java library.

7.2 Generalizability

We evaluated Sisyphus by using it to compare participants' implementations of a few simple Java Arrays and Math methods, but our tool may not scale well to other more complicated Java methods. Most participants were graduate students and adept at using Java. Hence, their implementation of the Java methods may not be representative of how a novice Java user would implement the same methods.

8 FUTURE WORK

8.1 IDE Integration

The tool we developed takes a Java file containing a set of methods, and reports which standard library methods it contains clones of. In order to be useful to developers and students, our tool could be integrated as a plugin to an IDE, such as Eclipse or IntelliJ. Every time a file is saved (or committed, depending on the desired frequency checking), our tool could be run on it, and the user could be notified of which standard library function they should use instead.

In addition to adding this interactive component, future iterations of the tool could allow the user to choose which libraries they check against, allowing for exploration of more APIs.

8.2 Advanced Clone Detection

Our analysis focused on clone detection of procedural code. More advanced features of Java, such as object methods and lambda-closures, were widely ignored by our analysis. These would require more advanced analysis and normalization, such as converting all object methods into static methods taking an object parameter.

Likewise, our analysis focused on methods, but clone detection research on classes has also been performed [16]. This could allow for recommending commonly duplicated data structures, such as pairs or linked-lists.

8.3 Performance

An important aspect of making our tool practical is improving its performance. While some low-hanging fruit remains, such as caching analyses of library functions, comparing to every library function may still be too slow to be performed every time the user saves. In addition, memory usage of Sisyphus is very high, as it is currently loading the entire solved library into memory. More advanced methods for reducing the number of comparisons will likely be needed for the tool to see adoption.

8.4 Modularity

One of Sisyphus' major advantages in design is the flexibility and adaptability of the framework. The tool is capable of integrating many different detection algorithms, graph builders, and parsers which we can leverage to analyze the effectiveness of any individual combination. By integrating a configuration file describing the specific tools to be used we can either perform comprehensive analysis or as quickly switch to a more effective detection algorithm in the context of the source code that is being analyzed.

8.5 Empirical Human Evaluation

While our evaluation measures the performance of our tool, the metric is somewhat artificial. Future work could include a detailed empirical study on programmers who are learning Java for the first time. Such a study could measure:

- The frequency of matches in code newcomers actually write.
- The proportion of matches which are false positives or negatives.
- The frequency of use of recommended methods (as a measure of how well the tool assists in learning a new library).
- The reduction in development time provided by the tool. (This would need to involve measurement over a medium to large sized project, as a project that is too small would not have enough code reuse to see any benefit to from our tool.)
- Developers' opinion of the tool (helpful, bothersome, etc.)

9 CONCLUSION

Our work provided an initial exploration into the application of clone detection to recommending library functions to beginner programmers. We developed such a recommender, based on existing clone detection techniques.

Our evaluation showed promising results, particularly for the semantic approach. However, some key deficiencies of clone detection were identified with regards to this specific application. In particular, structurally similar methods led to false positives, while standard-library code style led to many false negatives. Non-approximate approaches were found to be infeasible.

This work lays the foundation for syntactic and semantic library recommendation tools, exhibiting both their potential, and key weaknesses, providing groundwork for more elaborate experiments and more practically oriented tools. Our code will be published on https://github.com/FRuffy/CPSC_507.

REFERENCES

[1] 2017. *JGraphT – a free Java graph library.* http://jgrapht.sourceforge.net

[2] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. DOI: http://dx.doi.org/10.1145/24039.24041

[3] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 96–105.

[4] J. Howard Johnson. 1993. Identifying Redundancy in Source Code Using Fingerprints. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1 (CASCON '93)*. IBM Press, 171–183. http://dl.acm.org/citation.cfm?id=962289.962305

[5] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. 2009. Do Code Clones Matter?. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 485–495. DOI: http://dx.doi.org/10.1109/ICSE.2009.5070547

[6] T. Kamiya, S. Kusumoto, and K. Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (Jul 2002), 654–670. DOI: http://dx.doi.org/10.1109/TSE.2002.1019480

[7] Richard M. Karp and Michael O. Rabin. 1987. Efficient Randomized Pattern-matching Algorithms. *IBM J. Res. Dev.* 31, 2 (March 1987), 249–260. DOI: http://dx.doi.org/10.1147/rd.312.0249

[8] Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt. 1977. Fast Pattern Matching in Strings. *SIAM J. Comput.* 6, 2 (1977), 323–350. DOI: http://dx.doi.org/10.1137/0206024

[9] Raghavan Komondoor and Susan Horwitz. 2001. Tool Demonstration: Finding Duplicated Code Using Program Dependences. In *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*. 383–386. DOI: http://dx.doi.org/10.1007/3-540-45309-1_25

[10] Jens Krinke. 2001. Identifying Similar Code with Program Dependence Graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01) (WCRE '01)*. IEEE Computer Society, Washington, DC, USA, 301–. http://dl.acm.org/citation.cfm?id=832308.837142

[11] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[12] Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Bookshelf.

[13] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. SPOON: A library for implementing analyses and transformations of Java source code. *Software: Practice and Experience* 46, 9 (2016), 1155–1179. DOI: http://dx.doi.org/10.1002/spe.2346

[14] Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *QueenâĂŹs School of Computing TR* 541, 115 (2007), 64–68.

[15] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Sci. Comput. Program.* 74, 7 (May 2009), 470–495. DOI: http://dx.doi.org/10.1016/j.scico.2009.02.007

[16] Tobias Sager, Abraham Bernstein, Martin Pinzger, and Christoph Kiefer. 2006. Detecting Similar Java Classes Using Tree Algorithms. In *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR '06)*. ACM, New York, NY, USA, 65–71. DOI: http://dx.doi.org/10.1145/1137983.1138000

[17] Martin Schoeberl. 2008. A Java Processor Architecture for Embedded Real-time Systems. *J. Syst. Archit.* 54, 1-2 (Jan. 2008), 265–286. DOI: http://dx.doi.org/10.1016/j.sysarc.2007.06.001

[18] Danny van Bruggen. 2017. JavaParser. https://github.com/javaparser/javaparser. (2017).