AN INVESTIGATION OF ROUTINE REPETITIVENESS IN OPEN-SOURCE PROJECTS

Mohd Arafat

A Thesis
Submitted to the Graduate College of Bowling Green
State University in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE

August 2018

Committee:

Robert Dyer, Advisor
Robert C. Green II
Raymon Kresman
Robert Dyer, Advisor

Many programming languages contain a way to provide a sub-portion of the source code that performs a specific and often independent behavior. Depending on the language this is called a (sub-)routine, method, function, procedure, etc. One of the main purposes of creating a routine is to enable re-use. As devised, routines are intended to be called from multiple places within a program. Sometimes, however, the same code is repeated within a project or across projects. In this work, we investigate how often such routines are repeated in a large-scale corpus of open source software. This work attempts to independently reproduce a prior research result by Nguyen et al., building from the ground up the analysis framework and analyzing a different and very large set of open source software projects. In this work, we use the Boa infrastructure to investigate routine repetitiveness by analyzing over 300k open source projects from GitHub. Similar to the prior work, we first compute the program dependence graphs (PDGs) for each routine in the dataset, perform normalization on the PDGs, and look for repetitions both within and across projects. Our experiment shows that about 16.4% of routines repeat within a project and approximately 11% of routines repeat across at least two different projects. We then perform static program slicing on the PDGs, slicing the graph on each routine argument to obtain subroutines and look for repetitiveness once again. We observe that approximately 17% of all subroutines repeat within a project and 11% repeat across projects. Finally, we investigate if the size of the PDG or the number of control nodes has any impact on the repetitiveness of routines. Overall, our results confirm the trends shown in the prior study, though with differences in the size of the results.
I would like to express my gratitude to a number of people for their support during my time at BGSU. First and foremost, I would like to thank my advisor Dr. Dyer for being a source of ideas and for providing valuable guidance throughout my master’s journey. I have benefited immensely from his thoughtful suggestions and technical discussions. His insights on programming languages and writing clean and efficient code have made me a much better researcher and a programmer. I also have to thank him for being patient with me throughout my masters and especially in the last few weeks.

The rest of my thesis committee members, Dr. Kresman and Dr. Green also helped me greatly during my years as a graduate student. I would like to thank Dr. Kresman for teaching me the building blocks of algorithm design and databases. I was fortunate to have taken Dr. Green’s artificial intelligence class which introduced me to the new world of heuristics and meta heuristics algorithm. I thank both of them for their valuable suggestions and feedbacks to improve the thesis.

Lastly, this work was supported by the National Science Foundation (NSF) under grant CCF-1518776.
TABLE OF CONTENTS

CHAPTER 1  INTRODUCTION 1

CHAPTER 2  RELATED APPROACHES 4

2.1  Control Flow Graphs 4

2.2  Post Dominator Trees 7

2.3  Control Dependence Graphs 8

2.4  Data Dependence Graphs 9

2.5  Program Dependence Graphs 11

2.6  Program Slicing 12
    2.6.1  Forward Static Slicing 13

CHAPTER 3  RELATED EMPIRICAL STUDIES 15

3.1  Code Repetition Studies 15

3.2  Code Uniqueness Studies 16

CHAPTER 4  METHODOLOGY 17

4.1  Augmented Graphs 17

4.2  Normalization 17

4.3  Per-Argument Slicing subGraphs (PASG) 21

4.4  Implementation in Boa 21
    4.4.1  Boa Graphs API 23

4.5  Boa Queries 24

CHAPTER 5  EVALUATION 29
5.1 Dataset ................................................................. 29

5.2 Post-processing ......................................................... 29

5.3 RQ1: Repetitiveness of Routines ................................. 33

5.3.1 RQ1a: Repetitiveness Within a Project ....................... 33

5.3.2 RQ1b: Repetitiveness Across Projects ......................... 34

5.3.3 RQ1c: Repetitiveness by Graph Size ......................... 35

5.3.4 RQ1d: Repetitiveness by Number of Control Nodes .......... 36

5.4 RQ2: Repetitiveness of PASGs ..................................... 37

5.4.1 RQ2a: Repetitiveness Within a Project ....................... 37

5.4.2 RQ2b: Repetitiveness Across Projects ......................... 37

5.4.3 RQ2c: Repetitiveness by Graph Size ......................... 39

5.4.4 RQ2d: Repetitiveness by Number of Control Nodes .......... 39

5.5 Threats to Validity ....................................................... 40

CHAPTER 6 CONCLUSION ................................................... 41

BIBLIOGRAPHY ............................................................... 42
LIST OF FIGURES

2.1 An example Java method to iteratively compute factorials. ................. 4
2.2 A control flow graph for the example code shown in Figure 2.1. .......... 5
2.3 The CFG in Figure 2.2 augmented with an ENTRY node. .................... 6
2.4 The data-flow equations for computing dominators. .......................... 7
2.5 A post dominator tree for the augmented CFG shown in Figure 2.3. ....... 8
2.6 A control dependence graph for the PDT shown in Figure 2.5. ............. 9
2.7 The data-flow equations for computing def-use chains. ....................... 10
2.8 A data dependence graph for the augmented CFG shown in Figure 2.3. .... 10
2.9 A program dependence graph for the augmented CDG in Figure 2.6 and DDG in Figure 2.8. ................................................................. 11
2.10 An example forward static slice criteria, with all statements contributing to the slice highlighted. ................................................................. 13
2.11 A forward static slice using criteria $\text{<fact, 2>}$ on PDG shown in Figure 2.9. ................................................................. 14
4.1 The augmented CFG based on Figure 2.3. ........................................ 18
4.2 The augmented PDG based on Figure 2.9. ........................................ 19
4.3 The normalized PDG based on Figure 4.2. ...................................... 20
4.4 The PASG for the PDG in Figure 4.2. ............................................. 21
4.5 The normalized PASG based on Figure 4.4. .................................... 22
4.6 The Boa query for RQ1a (repetition within project). ........................ 24
4.7 The Boa query for RQ1b (repetition across projects). ....................... 25
LIST OF TABLES

5.1 Analyzed Dataset Statistics ........................................... 29
CHAPTER 1. INTRODUCTION

The word repetition or duplication refers to occurrence of something which has already been said or done. Repetition occurs in all kind of systems and depending on the context can be good or bad. For example, repetition codes are used as an error correction mechanism in communication systems and therefore would constitute a good use of repetition. A bad example would be copying somebody else’s work without citation. Software systems are also not immune to duplication and tend to repeat both in terms of architectural design and code syntax and semantics. Software repetition, though mostly considered undesirable [19], is sometimes needed to enhance the performance or some parts of the system.

In this work, we are going to deal with only code repetitions \[14, 17, 31\] in software systems which are maintained in repositories. Code repetition is not always intentional and could also be sometimes accidental especially if it is occurring across different projects. The are many reasons why source code repetition occurs. First of all, some people find it easier to copy and paste a working or complex piece of code. Automation tools or compilers which translate source code from one language to another can also produce similar looking code. Sometimes efficient and reusable APIs or routines are copied because of performance reasons. Other reasons could be implementation of similar algorithms, naming and style conventions [31], and use of similar design patterns. There are also cases like distributed Source Code Management systems (SCM) such as git whose philosophies are based on allowing clones or copies of projects so that people can experiment with and customize their individual copies as per their needs.

There are many disadvantages associated with code repetition specifically if it is a result of copy and paste. If the copied code has defects, the defects also get replicated and if the code is further copied, it may lead to defect propagation. Repeated code, especially within a project, can cause maintenance issues and drive up its cost. This is because if a change is required in the code, it has to be carried out at all the clone locations. This will require more time and effort and can
result in messy code. Though mostly discouraged but repeated code in some cases can provide certain advantages. A repeated method with specifications can help in synthesizing specifications for rarely used similar APIs using some fine grained slicing techniques. Repetition can also help code suggestion tools by aiding in generating better models for suggestions.

The repetition of code can occur at lexical level, syntactical level and semantic level [4, 14, 17]. Lexical techniques like token for token similarity and syntax techniques statement for statement similarity have been used quite extensively in previous researches but there has been a recent increase in semantic based techniques. Nguyen et al. [31] published the first semantic analysis based large scale study on repetitiveness, composition, and, containment of source code. They used a program dependence graphs and slicing based technique to find the percentage semantically similar routines or methods. Their work answered five research questions such as repetition rate of routines within as well as across projects, repetition of routines as part of other routines, repetition of part of routines from other places, repetition of multiple routines together and, the repetitiveness of routines (or part of) routines from common libraries. The data used for the research is a snapshot of SourceForge repository consisting of under 10K projects and just over 17M methods. The program dependence graphs are used to compare routines for similarity and their slices are used to answer questions on routines composability which is defined as the number of per variable subroutines (PVSG) composing a method.

In this work, we repeat the work of Nguyen et al. [31] and answer two of their five research questions which are as follows.

RQ1. How likely a routine for a task is repeated exactly elsewhere as an entirety?

RQ2. How often portions of a routine are repeated or repeated together; what is the unique set of all of such portions?

One problem with the earlier approach is that it is difficult to replicate or extend the research because of the heavy lifting required to set up the infrastructure which can mine on a scale of 100K projects. In this work we leverage the existing Boa [11] infrastructure which is capable of mining ultra large software repositories. Boa infrastructure consists a domain specific language,
a compiler for translating it, multiple datasets such as GitHub, SourceForge, a back-end based on map-reduce for parallel distributed processing of datasets and a front-end to write boa queries. For this research we have used GitHub-2015 dataset. To carry out the analysis we have extended the boa infrastructure with graph analysis capabilities. Boa now supports construction of control flow graphs, dominator and post-dominator trees, control dependence graphs, data dependence graphs, program dependence graphs and also enables slicing control flow graphs and program dependence graphs. With the enhanced infrastructure it is as easy as writing 30-40 lines boa queries to perform highly scalable research on code repetitiveness and other similar areas which require large scale analysis.
CHAPTER 2. RELATED APPROACHES

This chapter describes various graphs and trees which are intermediate data structures needed for building a program slicer. The subsections present different graph representations of the example method shown in Figure 2.1.

```
void factorial(int arg) {
  int count = 1;
  int fact = 1;
  while (count <= arg) {
    fact = fact * count;
    count = count + 1;
  }
  return fact;
}
```

Figure 2.1: An example Java method to iteratively compute factorials.

This method is a simple Java method that takes an integer as input and returns its factorial. The method initializes two local variables, *count* and *fact* to 1. The variable *fact* is used to store the result of the factorial of *arg*. The variable *count* generates integers from 1 to *arg* which are multiplied with *fact* one at a time. When all the integers have been multiplied, the control comes out of the loop and the factorial is returned to the caller.

2.1 Control Flow Graphs

A control flow graph (CFG) [5] is a directed graph that represents all possible execution paths in a program. A node can represent a single expression/statement or a basic block which consists of a straight line sequence of expressions/statements with no branching. A basic block representation is more compact because of reduced number of nodes and edges and therefore has less memory footprint. An expression/statement representation can simplify analysis in some cases but has higher memory footprint and is less efficient to build. In this work, we refer to a single expression/statement as node for convenience. An edge represents a possible control flow path from
one node to another node in the CFG. A CFG can be constructed from any suitable intermediate representation of the source code, such as an abstract syntax tree (AST).

Figure 2.2: A control flow graph for the example code shown in Figure 2.1. The nodes START and STOP in the figure are special kinds of nodes added to the CFG to show entry into and exit from the method, respectively. The numbers inside brackets in each box show the node ids, which can be mapped to the statement number in the method by decreasing one from it. The labels T and F correspond to the true and false branches of a predicate node. The label B represents a back-edge of a loop. Before constructing a post dominator tree from the CFG, we augment it
with a new predicate node named ENTRY [13]. The ENTRY node models the condition which triggers the method call i.e entry into the method. It has two branches, the true branch going to the START node and the false branch going to the STOP node. The newly augmented CFG is shown in Figure 2.3.
2.2 Post Dominator Trees

A post dominator tree (PDT) is a tree representation in which the parent of a node represents its immediate post dominator. To understand PDT it is important to first understand the concept of post dominance which can be defined as follows: a node \( n_2 \) post dominates a node \( n_1 \) if all paths from \( n_1 \) to the STOP node go through \( n_2 \). The post dominator set for each node can be computed by applying the data-flow equations in Figure 2.4 to the reverse CFG (a reverse CFG is the original CFG with all its edges reversed and control enters from the STOP node and exits through the START node).

\[
\begin{align*}
\text{Dom} (n_0) &= n_0 \\
\text{Dom} (n) &= \left( \bigcap_{p \in \text{pred}(n)} \text{Dom}(p) \right) \cup \{n\}
\end{align*}
\]

Figure 2.4: The data-flow equations for computing dominators.

From the post dominator set of a node, it is easy to determine its immediate post dominator, which post dominates only the concerned node and no other node from the set. A PDT can then be constructed by connecting each node to its immediate post dominator and since each node has exactly one immediate post dominator (except the START node), the resulting connections will form a tree. Alternatively, there is a direct and more efficient algorithm by Lengauer and Tarjan [27] for building a PDT. Due to its complexity, we chose not to use this alternate algorithm.

Figure 2.5 shows the PDT for the augmented control flow graph in Figure 2.3. Given a tree, it is easy to find post dominators and immediate post dominators of a node. To get the post dominator set for a node, traverse from the parent of the given node all the way up to the STOP node. For example, the post dominator set of node 2 is \( \{3, 6, 7\} \). As can be seen in the figure, the START node does not post dominate any node and the STOP node post dominates all other nodes in the graph.
2.3 Control Dependence Graphs

A control dependence graph (CDG) represents control dependency among CFG nodes. The labels of control edges correspond to the predicate being true (T) or false (F) in the control node. A CDG can be built by walking up a PDT from the destination node to the source node for certain control edges in the augmented CFG. For example, consider the control edge 3 → 4 in Figure 2.3 where the source node is 3 and the destination node is 4. On traversing the PDT in Figure 2.5 from node 4 to node 3, we get nodes 4 and 5 as the dependent nodes. We do not include those control edges where the source node is post dominated by the destination node. For example, we skip the edge 3 → 6 because 3 is post dominated by 6. The construction of CDGs is explained in much more detail by Ferrante et al. [13].
The CDG presented in this work differs from that given in [13] as it does not include the concept of region nodes. A CDG can also be built by using other approaches such as building a post dominance frontier [10] and then reversing it to get the control dependence. Both approaches are quite easy to follow but in this work we have used Ferrante’s approach.

![Control Dependence Graph](image)

Figure 2.6: A control dependence graph for the PDT shown in Figure 2.5.

Figure 2.6 shows a CDG constructed from the PDT in Figure 2.5. For constructing a CDG, only the true branch from the ENTRY node is considered and the false branch is ignored because the STOP node post dominates the ENTRY node. Every node in a CDG is either directly or transitively dependent on the ENTRY node because it models the external condition which triggers the method. In Figure 2.6 we can see the dependency of different nodes on the two predicate nodes 0 and 3. In a CDG, the successors of each node show its dependent nodes. For example, nodes 4 and 5 are control dependent on node 3. In other words, the execution of nodes 4 and 5 depends on node 3. Notice also that node 3 is control dependent on itself.

2.4 Data Dependence Graphs

A data dependence graph (DDG) represents data dependency among CFG nodes. A DDG can be built by connecting variable definitions to their uses and the connections are known as def-use chains. Def-use chains can be computed using the same data-flow equations that are used for
live variable analysis, with one modification to keep track of the node ids of each set of the out variables. So for live variable analysis out[n] and in[n] are set of variables but in this case they are set of pairs of variable and node i.e set of (variable, node). A DDG is usually a disconnected graph.

\[
\begin{align*}
\text{in}[n] &= \text{use}[n] \cup \left( \text{out}[n] - \text{def}[n] \right) \\
\text{out}[n] &= \bigcup_{s \in \text{succ}[n]} \text{in}[s]
\end{align*}
\]

Figure 2.7: The data-flow equations for computing def-use chains.

Figure 2.8 shows a DDG constructed by applying the data-flow equations in Figure 2.7 to the augmented CFG from Figure 2.3. The label of each edge shows the dependent variable for the successor node. Here the ENTRY node is added just to be consistent with the previous graphs.
The node however does not represent any data dependence for the successor nodes which is also evident from the empty label of the edges originating from it.

2.5 Program Dependence Graphs

A *program dependence graph* (PDG) represents both control and data dependency among CFG nodes. A PDG is a composition of a CDG and a DDG. To construct a PDG, first a CDG is constructed and then edges representing data dependencies (def-use chains) are overlaid on it. A PDG is a very sophisticated data structure and is used in various complex analyses and compiler optimizations.

Figure 2.9: A program dependence graph for the augmented CDG in Figure 2.6 and DDG in Figure 2.8.
Figure 2.9 shows a PDG constructed from the augmented CDG and DDG of Figure 2.6 and Figure 2.8 respectively. Each edge shows either a control or data dependency which is made explicit from the labels. The total edges in a PDG is the sum of the edges in the CDG and the DDG which are used to compose it. As mentioned earlier, we do not include edges originating from the ENTRY node in the DDG and therefore we only see control edges emanating from the ENTRY node of the PDG.

2.6 Program Slicing

Program slicing is a method of filtering out certain statements of interest from a program based on some slicing criteria. The term was first defined in 1981 by Weiser [41] who compared it to the mental abstraction of relevant code statements that programmers make when debugging a piece of code. Program slicing can be classified as intra-procedural, when a graph does not cross the boundary of the method or inter-procedural [18, 41], when a graph of the entire program or project is constructed. The graphs can be of different kinds such as control flow graphs, program dependency graphs, system dependency graphs, etc and some of these are explained in the previous sections. The other major classifications include forward and backward slices, static, quasi-static and dynamic slices, and executable and closure slices [38].

Forward slicing is used to determine the effect of certain selected variables (aka the slicing criteria) on other statements in the program whereas backward slicing produces all the statements which can affect the variables at the slicing criteria location. The slicing criteria of dynamic slicing [1, 7, 15, 23, 43], unlike static slicing, takes into account the input values of the variables which enables dynamic slicing to generate significantly smaller slices by excluding unnecessary branches. The dynamic slicing technique has been applied to a number of problems and has shown promising results especially in the field of software testing and debugging [24, 29, 39]. When the sliced statements form an executable subgraph then the slice produced is said to be executable. On the other hand if the subgraph is not executable then the slice is called a closure slice. While most of the earlier work on program slicing focused on approaches which preserved the syntactic struc-
ture of the program, Harman and Danicic presented a technique for amorphous slicing [6, 12, 16] which is semantic preserving but performs necessary transformations of the source code statements to reduce the size of the slices. In this work we only deal with forward static slicing.

2.6.1 Forward Static Slicing

*Forward static slicing* is used to generate statements which are affected by the variables in the slicing criteria. For this reason it is also called ripple effect slicing. The slicing criteria for forward static slicing is a set of the form `<variable, statementNum>` and unlike dynamic slicing, does not include input values of the variables. The method in Figure 2.1 is sliced using a slicing criteria `<fact, 2>` and the slice obtained is shown highlighted in Figure 2.10. The slice shows all the statements which are affected by the local variable `fact`.

```c
void factorial(int arg) {
    int count = 1;
    int fact = 1; // criteria: <fact, 2>
    while (count <= arg) {
        fact = fact * count;
        count = count + 1;
    }
    return fact;
}
```

Figure 2.10: An example forward static slice criteria, with all statements contributing to the slice highlighted.

To compute slices, it is more convenient to first translate the code into a suitable intermediate representation such as PDG and then slice it with a given slicing criteria to obtain the desired slice. Figure 2.11 shows the slice obtained by slicing the PDG from Figure 2.9 with the slicing criteria `<fact, 2>`. The slice obtained with PDG captures both control and data dependencies unlike the slice on CFG which only captures control dependencies.
Figure 2.11: A forward static slice using criteria $<\text{fact}, \ 2>$ on PDG shown in Figure 2.9.
CHAPTER 3. RELATED EMPIRICAL STUDIES

*Code repetitiveness* and *code uniqueness* have been studied quite extensively in previous research. In this chapter we discuss prior research in these fields.

3.1 Code Repetition Studies

Techniques for finding repetition in source code across software systems include tree based, graph based, token or n-gram based, heuristic based, and machine learning approaches. Here we overview a few of these approaches.

There have been numerous studies on the identification of source code clones [21, 22, 25, 26] across software systems which resulted in different taxonomies and classification of clones such as true clones and accidental clones [2]. Jiang and Su [20] presented an algorithm to search for similar code fragments of random sizes at the functional level using the technique of random testing. Researchers have employed syntax based, semantics based as well as many hybrid techniques for searching similar codes. Reiss [36] presented a semantic-based code search technique to generate user requested functions or classes using a set of program transformations. Several heuristics and meta-heuristics [8, 40] have also been tried for the large scale detection of repetitiveness in source codes.

Hindle et al. [17] introduced an n-gram based approach to find repetitive nature of software when broken into its constituent pieces at the source level. Since the n-grams model alone fails to capture the effect of localness of software, Tu et al. [37] came up with an approach that augments the n-gram model with a cache component to exploit locality of the software. Nguyen et al. [32] described a syntax similarity based approach to find repetitive code changes in software evolution using difference in sub-AST structures for a pair of code pieces. They later presented an approach which uses semantic information to detect source code repetitiveness [33]. This is different from an n-gram model which uses lexemes or syntactic similarity to determine uniqueness of source code.
White et al. [42] employed a deep learning based technique for training neural networks on code fragments to identify code clones. The technique exploits both lexical and syntactic structures to link patterns. Cesare and Xiang [9] presented a framework for finding package level clones using pattern classification techniques and proposed a number of systems based on it. Mahmoud and Bradshaw [30] described an approach that takes into account domain-specific terms in the source code to augment the semantic-based code analysis techniques. The authors used the Normalized Software Distance (NSD) method which captures semantic relatedness in the source code.

Finally, Nguyen et al. [31] presented a large scale study on open source projects to find code repetitiveness by comparing PDGs and subgraphs extracted from PDGs using slicing. This work is the focus of our reproduction study.

3.2 Code Uniqueness Studies

The techniques used for identifying code uniqueness are similar to those used for identifying repetitiveness. Gabel and Su [14] presented the first study on code uniqueness which answers questions about software uniqueness at a given level of granularity. Ray et al. [35] present the definition of unique changes and proposes a methodology to identify them. The study establishes how prevalent unique code is and investigated where they occur most commonly. Petke et al. [34] performed a comprehensive survey on the improvement of software using genetic techniques. It covers various techniques such as parameter tuning, slicing etc that have been used to improve or transform software. Alexandru et al. [3] designed a generic framework called LISA (Lean Language-Independent Software Analyzer) which represents and analyzes multi-revisioned software artifacts and employs a unique, redundancy-free, multi-revision representation to avoid re-computation of the same artifacts.
CHAPTER 4. METHODOLOGY

This work reproduces two of the five research questions from the previous work by Nguyen et al. [31] but using the Boa infrastructure and dataset. Similar to their study, this work constructs program dependence graphs and then uses them to compare either the entire methods or the per-argument slicing subgraph (PASG) obtained by slicing a method’s PDG on each argument. The latter is different from per-variable slicing subgraphs (PVSG) used in the prior work, where a PDG is sliced on every local variable. This is the main difference between our approach and their approach, meaning our RQ2 is slightly modified to use PASGs instead of PVSGs.

4.1 Augmented Graphs

To enable PASGs in Boa, we augment control flow graphs with new argument assignment statements that are placed at the top immediately after the \texttt{START} node. Since all remaining graphs in Boa are built starting from the control flow graph, these new nodes cascade to all other graph kinds. An example of the new control flow graph with augmented nodes for the CFG in Figure 2.3 is shown in Figure 4.1.

As can be seen in the code example in Figure 2.1, the argument \texttt{arg} of the method \texttt{factorial} is now a node in the control flow graph. This argument is assigned an arbitrary value ($k$). Since we are doing static analysis the choice of arbitrary value does not affect the results. We skip showing the other augmented intermediary graphs as they are similar and directly show the final augmented PDG in Figure 4.2.

4.2 Normalization

To normalize a program dependence graph or a PDG slice, the normalizing algorithm traverses the graph in a depth first order. At each node it sorts all the node’s successors and all its out edges using their respective node/edge id. The normalized program dependence graph and slices also
Figure 4.1: The augmented CFG based on Figure 2.3.
Figure 4.2: The augmented PDG based on Figure 2.9.
have all their variable names replaced with the pattern var$(number)$. For example the normalized version of the expression $x = y + 5$ will be $\text{var$1 = var$2 + 5}$. The normalization algorithm keeps a map of already seen variables and their normalized names. For the previous expression, the map will contain \{x: var$1, y: var$2\}.

The algorithm traverses the program dependence graph (or its slice) and when it reaches nodes with expressions it first looks up to see if the variables in the expression have already been assigned names. If a variable is present in the map, the algorithm replaces the variable with its normalized name and if the variable is not present then it generates the next sequential value which it uses to replace the variable and also stores the mapping. For example, the normalized version of the
4.3 Per-Argument Slicing subGraphs (PASG)

A per-argument slicing subgraph (PASG) is obtained when a program dependence graph is sliced on one of the method’s arguments. The subgraph obtained is used to answer research questions related to containment and composability of routines and its slices. The PASG of the program dependence graph of Figure 4.2 on its only argument \( \text{arg} \) is shown in Figure 4.4 and its normalized version is shown in Figure 4.5.

4.4 Implementation in Boa

Boa [1] is an enabling language and infrastructure for mining ultra large scale software repositories. It consists of a domain-specific language, a compiler for the language, multiple datasets.
with each containing almost a million open-source projects, a back-end based on map-reduce for parallel distributed processing of datasets and a front-end to write programs in boa language. Boa’s domain-specific language is a statically typed query language which tightly integrates with the Boa infrastructure. The compiler compiles the query to a Java program which acts as a map in the map-reduce model and each map program operates on one open source project. The number of maps running concurrently depends on the back-end and can be adjusted as per the core count of the boa’s back-end cluster. So depending on the back-end core count, it is possible to achieve a very high degree of parallelism. Boa datasets are based on open-source repositories like GitHub, SourceForge, etc. Each project’s code in the repository is parsed and made available to users for analysis using Boa queries. Boa gets new datasets each year while preserving the datasets from the previous years so that they are available to researchers for experimenting and verifying previous results.
4.4.1 Boa Graphs API

The Boa graph API provides access to several kinds of graphs and trees. It includes control flow graphs, control dependence graphs, data dependence graphs, program dependence graphs, dominator trees, post dominator trees, and two slicers for program dependence graphs and another for control flow graphs. This Boa graph library was implemented as part of this research.

To get a graph object, one of several builtin functions is called. For example, to get a control flow graph for a method a user calls `getcfg(Method)`. Here, `Method` is a domain specific type representing a Java method in the abstract syntax tree in Boa. Similarly there are other functions for other graphs kinds:

- `getcfg(Method)`: returns a control flow graph
- `getcdg(Method)`: returns a control dependence graph
- `getddg(Method)`: returns a data dependence graph
- `getpdg(Method)`: returns a program dependence graph

Some functions take optional arguments for normalizing the graph. Currently, only program dependence graphs and program dependence graph slices can be normalized in this way. We also provide a `normalize()` function to normalize any graph object.

There are also custom traversals available which can be used to traverse the graphs using standard traversal algorithms such as depth first, post order, worklist, etc. Boa also provides a way to visualize the graphs. The `dot(Graph)` function returns a string representation of the `Graph` object. This representation uses GraphViz DOT, a graph descriptive language. GraphViz provides tools to turn a DOT description into a graph image. All graphs in this paper were produced using this Boa function and GraphViz.
```plaintext
out: output sum[string][string] of int;

visit(input, visitor { # last snapshot only
  before c: CodeRepository -> {
    if (len(c.revisions) >= 50) {
      snapshot := getsnapshot(c, "SOURCE_JAVA_JLS");
      if (len(snapshot) >= 50)
        foreach (i: int; def(snapshot[i]))
          visit(snapshot[i]);
    }
    stop;
  }
  before m: Method -> {
    if (len(m.statements) > 0) {
      pdg := getpdg(m, true);
      normalize(pdg);
      hash := getcrypthash(pdg, "SHA-1");
      out[input.id][hash] << 1;
    }
  }
});
```

Figure 4.6: The Boa query for RQ1a (repetition within project).

### 4.5 Boa Queries

The Boa language can be used easily for writing queries to mine software repositories. For a single run only one dataset can be queried. The language provides standard programming constructs such as loops (called quantifiers in Boa), conditionals, functions, and a function library for writing time efficient queries which are also very readable. The output of the query can be combined using one of the several Boa aggregators such as top, bottom, sum, collection, etc.

In this work we created one query for each research sub-question.

For example, Figure 4.7 shows the query for answering the first part of RQ1 that are repetition of methods within project. The query has globals out and seen at the top where out is an aggregator used to collect the results and seen is a set used to hold PDGs that have already been seen in the project. Lines 2-20 show a custom visitor which is called from line 2. The visitor is used to perform custom traversals on the project source code represented as an abstract syntax.
Figure 4.7: The Boa query for RQ1b (repetition across projects).

tree. The visitor provides access to each node in the tree with several domain specific types. The two visitor constructs `before` and `after` perform operations specified at each node of the tree before visiting the children of the node and after visiting the children of the node, respectively.

In this work, we are only considering Java projects and among those we filter out projects with less than 50 revisions or less than 50 files as shown in the figure from lines 3–11. Line 13 filters out the abstract methods (a method with no body) and emits a SHA1 cryptographic hash corresponding to a method’s PDG but takes care not to emit hashes for similar methods twice. The queries for the remaining RQs and their parts are shown in Figure 4.7 to Figure 4.13.
Figure 4.8: The Boa query for RQ1c (repetition by graph size).

Figure 4.9: The Boa query for RQ1d (repetition by number of control nodes).
Figure 4.10: The Boa query for RQ2a (repetition within project).

```java
out: output sum[string] of int;

visit(input, visitor {
    before c: CodeRepository -> { # last snapshot only
        ...
        before m: Method -> {
            if (len(m.statements) > 0) {
                foreach (i: int; def(m.arguments[i])) {
                    pdg := getpdg(m, true);
                    slice := getpdgslice(pdg, i, true);
                    slicehash := getcrypthash(slice, "SHA-1");
                    out[input.id][slicehash] << 1;
                }
            }
        }
    });
```

Figure 4.11: The Boa query for RQ2b (repetition across projects).

```java
out: output sum[string] of int;

seen: set of string

visit(input, visitor {
    before c: CodeRepository -> { # last snapshot only
        ...
        before m: Method -> {
            if (len(m.statements) > 0) {
                foreach (i: int; def(m.arguments[i])) {
                    pdg := getpdg(m, true);
                    slice := getpdgslice(pdg, i, true);
                    slicehash := getcrypthash(slice, "SHA-1");
                    if (!contains(seen, slicehash)) {
                        add(seen, slicehash);
                        out[slicehash] << 1;
                    }
                }
            }
        }
    });
```
out: output sum[string][string] of int;

visit(input, visitor {
  before c: CodeRepository -> { # last snapshot only
    ...
  }
  before m: Method -> {
    if (len(m.statements) > 0) {
      foreach (i: int; def(m.arguments[i])) {
        pdg := getpdg(m, true);
        slice := getpdgslice(pdg, i, true);
        graph_size := gettotalnodes(slice) + gettotaledges(slice);
        slicehash := getcrypthash(slice, "SHA-1");
        out[graph_size][slicehash] << 1;
      }
    }
    out[graph_size][slicehash] << 1;
  }
});

Figure 4.12: The Boa query for RQ2c (repetition by graph size).

out: output sum[string][string] of int;

visit(input, visitor {
  before c: CodeRepository -> { # last snapshot only
    ...
  }
  before m: Method -> {
    if (len(m.statements) > 0) {
      foreach (i: int; def(m.arguments[i])) {
        pdg := getpdg(m, true);
        slice := getpdgslice(pdg, i, true);
        num_control_nodes := gettotalcontrolnodes(slice);
        slicehash := getcrypthash(slice, "SHA-1");
        out[num_control_nodes][slicehash] << 1;
      }
    }
    out[num_control_nodes][slicehash] << 1;
  }
});

Figure 4.13: The Boa query for RQ2d (repetition by number of control nodes).
CHAPTER 5. EVALUATION

In this section we present our results and compare them with the results of Nguyen et al. [31]. In this work, we answered two research questions RQ1 and RQ2 which corresponds to RQ1 and RQ4 of Nguyen et al. [31]. To help with comparisons, we kept captions of the figures the same as in Nguyen et al. [31].

5.1 Dataset

The Boa data set used is 2015 September/GitHub. The important statistics of the dataset are shown in the Table 5.1. The query to generate this data is shown in Figure 5.1. This data is after filtering out toy projects. Almost 38k projects are analyzed.

<table>
<thead>
<tr>
<th>Total Projects</th>
<th>37,955</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Classes</td>
<td>12,572,376</td>
</tr>
<tr>
<td>Total Methods</td>
<td>77,035,440</td>
</tr>
<tr>
<td>Total Statements</td>
<td>559,100,406</td>
</tr>
<tr>
<td>Total extracted PDGs</td>
<td>161,560,232</td>
</tr>
<tr>
<td>Total extracted PASGs</td>
<td>154,487,449</td>
</tr>
</tbody>
</table>

5.2 Post-processing

The post-processing script takes data produced by the Boa scripts (about 15GB total) and converts it to the Sframe [28] format. Sframe is a mutable dataframe object which can scale to big data. We used it because it stores data on disk and is not constrained by memory size. Once we have the data in the required format we perform a number of transformations on the Sframe object.

Figure 5.4 shows the script used to generate the graphs from the data. Figure 5.2 shows the portion of the script used to compute parts A and B for both RQ1 and RQ2. Figure 5.3 shows the portion of the script used to compute parts C and D for both RQ1 and RQ2. Finally, Figure 5.5 shows the main portion of the script.
has := false; st: stack of bool;

has2 := false; st2: stack of bool;

# filter out projects that cause memory issues
bad := { "10455994", "4858268", "4857247", "2051480"};
ifall (i: int; input.id != bad[i])
visit(input, visitor {
    before node: CodeRepository -> {
        if (len(node.revisions) >= 50) {
            snapshot := getsnapshot(node);
            if (len(snapshot) >= 50) {
                foreach (i: int; def(snapshot[i]))
                    visit(snapshot[i]);
                p << 1;
            }
        }
    }
    stop;
}
before Declaration -> {
    push(st2, has2);
    has2 = false;
}
after Declaration -> {
    if (has2) c << 1;
    has2 = pop(st2);
}
before Method -> {
    push(st, has);
    has = false;
}
after Method -> {
    if (has) {
        m << 1;
        has2 = true;
    }
    has = pop(st);
}
before Statement -> {
    has = true;
    s << 1;
}
});

Figure 5.1: Boa query to generate dataset statistics.
def rq_AorB(sf, rq, x_lower=1, x_upper=20, x_step=1, y_step=0.25, x_label='Number of Repetition'):
    sf_rq = sf[sf['hash'].contains(rq)]
    total_methods = sf_rq.num_rows()
    occurance_count = sf_rq.groupby(key_columns='repeat', operations={'counts': graphlab.aggregate.COUNT()}).sort('repeat')
    rep_count = occurance_count.to_dataframe().set_index('repeat').to_dict(orient='dict')['counts']
    rep_percent = dict()
    for k, v in rep_count.iteritems():
        if k > x_upper+1: break
        if k > x_lower: rep_percent[k - 1] = v * 100.00 / total_methods
    for i in range(x_lower, x_upper+1):
        if i not in rep_percent.keys(): rep_percent[i] = 0
    y_max = max(rep_percent.iteritems(), key=operator.itemgetter(1))[0]
    plot_graph(rep_percent, [x_lower, x_upper], [0, rep_percent[y_max] + 1], x_step, y_step, x_label, rq, 'o')

Figure 5.2: Python script to compute parts a/b from RQ1 and RQ2.

def rq_CorD(sf, rq, x_lower=3, x_upper=123, x_step=10, y_step=5, pad=5, x_label='Size'):
    sf_rq = sf[sf['hash'].contains(rq)]
    sf_rq['temp'] = sf_rq.apply(lambda x: (x['hash'].lstrip(rq).strip('[').strip(']')))
    sf_rq['graph_property'] = sf_rq.apply(lambda x: int(re.sub('.*',' , x["temp"])))
    sf_rq.remove_columns(['temp','hash'])
    gproperty_rcount = sf_rq.groupby(key_columns='graph_property', operations={'counts': graphlab.aggregate.COUNT()}).to_dataframe().set_index('graph_property').to_dict(orient='dict')['counts']
    sf_rq = sf_rq[sf_rq['repeat'] > 1].groupby(key_columns='graph_property', operations={'counts': graphlab.aggregate.COUNT()}).to_dataframe().set_index('graph_property').to_dict(orient='dict')['counts']
    rep_percent = OrderedDict()
    for k, v in sorted(sf_rq.iteritems()):
        if int(k) > x_upper: break
        rep_percent[k] = v * 100.00 / gproperty_rcount[k]
    for k in range(0, x_lower):
        if k in rep_percent: del rep_percent[k]
    y_max = max(rep_percent.iteritems(), key=operator.itemgetter(1))[0]
    plot_graph(rep_percent, [x_lower, x_upper], [0, rep_percent[y_max] + pad], x_step, y_step, x_label, rq)

Figure 5.3: Python script to compute parts c/d from RQ1 and RQ2.
import re
import numpy
import operator
import graphlab
import graphlab.aggregate
import matplotlib.pylab
from collections import OrderedDict
from matplotlib.ticker import PercentFormatter

def plot_graph(repitition_percent_map, x_range=[0,20], y_range=[0,10],
                x_step=1, y_step=1, x_label='Number', fig_name='Repetition', mark='',
                col='blue'):
    lists = repitition_percent_map.items()
    x, y = zip(*lists)
    matplotlib.pylab.xlim(xmin=x_range[0], xmax=x_range[1])
    matplotlib.pylab.ylim(ymin=y_range[0], ymax=y_range[1])
    matplotlib.pylab.xticks(numpy.arange(x_range[0], x_range[1]+1,
                                          x_step))
    matplotlib.pylab.yticks(numpy.arange(int(y_range[0]), y_range[1]+1,
                                          y_step))
    matplotlib.pylab.xlabel(x_label, fontsize=15)
    matplotlib.pylab.ylabel('Percentage', fontsize=15)
    matplotlib.pylab.gca().yaxis.set_major_formatter(PercentFormatter(
        xmin=100, decimals=0))
    matplotlib.pylab.grid(True)
    matplotlib.pylab.plot(x, y, linewidth=3, color=col, marker=mark)
    matplotlib.pylab.savefig(fig_name + '.pdf')

sf = graphlab.SFrame.read_csv('full.out.txt', delimiter=' ', usecols=[
    'hash', 'equal', 'repeat']).remove_column('equal')
rq_AorB(sf, 'rq1a', 1, 20, 1, 1, 'Number of Repetitions(Repeated Routines)')
rq_AorB(sf, 'rq1b', 1, 20, 1, 1, 'Number of Repetitions(Repeated Routines)')
rq_CorD(sf, 'rq1c', 3, 123, 10, 10, 10, 'Graph Size')
rq_CorD(sf, 'rq1d', 0, 20, 2, 10, 10, 'Number of Control Nodes')
rq_AorB(sf, 'rq2a', 1, 20, 1, 1, 'Number of Repetitions(Repeated Subroutines)')
rq_AorB(sf, 'rq2b', 1, 20, 1, 1, 'Number of Repetitions(Repeated Subroutines)')
rq_CorD(sf, 'rq2c', 3, 53, 10, 10, 10, 'Graph Size')
rq_CorD(sf, 'rq2d', 0, 20, 2, 10, 10, 'Number of Control Nodes')

Figure 5.4: Python script to plot graphs.

Figure 5.5: Main portion of Python script to perform post-processing.
5.3 RQ1: Repetitiveness of Routines

The first research question investigates the repetitiveness of entire routines within and across projects. The question has four parts which are evaluated in the following sub-sections.

5.3.1 RQ1a: Repetitiveness Within a Project

The repetitiveness of routines within a project is shown in Figure 5.6. The figure shows the percentage of repeating routines decreases with the increase in repetition count. Our experiment shows 16.4% routines repeat once, 4.2% twice and 2.2% thrice in a project. The corresponding repeat percentages for Nguyen et al. [31] are 6.7%, 2% and 1.1% respectively. For routines with more than 7 repetitions, the repeat percentage is less than 0.1% in our case as well as theirs. About 24% of routines are repeated up to 7 times in our case which is higher than the 12.01% reported by Nguyen et al. [31].

Notice that the general trend we observed is the same as the prior work. However, while the
shape of the curve is the same, the percentages are roughly double and thus the line is shifted upward.

5.3.2 RQ1b: Repetitiveness Across Projects

The repetitiveness of routines across projects is shown in Figure 5.7. This figure, like the last one, shows a sharply decreasing trend in the repetition percentage as the number of repetition increases. The repeat percentages for one, two and three repetitions are 10.8%, 3.8% and 1.83% respectively. The corresponding repeat percentages for n Nguyen et al. [31] are 2%, 0.5% and 0.3% respectively. Our result shows that for routines with more than 12 repetitions, the repeat percentage is less than 0.1%. About 19% of routines are repeated between 1 to 12 times across different projects.
5.3.3 RQ1c: Repetitiveness by Graph Size

In this part we measure the repetitiveness based on the size of PDG. The size of a PDG is taken as the sum of nodes and edges. The repetition is calculated for each group based on PDG size. For example, in Figure 5.8 repetition for methods with PDG of size 4 is about 58%. Methods with smaller PDG sizes in the range of 3-5 tend to repeat more, primarily because getters and setters fall into that size range. We have not included sizes 1 and 2 because they represent methods with empty bodies having only entry and exit nodes. Both our and Nguyen et al. [31] results show initially a decreasing trend with PDG size but their decrease is very sharp between the size range of 3-5. As can be seen in the figure the repeat percentage is much less affected by the graph size after 40. The Nguyen et al. [31] result confirms the trend.
5.3.4 RQ1d: Repetitiveness by Number of Control Nodes

The number of control nodes in a program is a measure of its complexity. A program with more control nodes will be considered having higher complexity. In this part of RQ1 we calculate the repetitiveness based on the number of control nodes in a method. The PDGs of the methods in the entire corpus are first grouped by the number of control nodes and then repetition is found in different groups. The result in Figure 5.9 shows that methods with no control nodes repeat about 43% of the times. There is a decreasing repetition trend for the first six control nodes and then repetition becomes effectively indifferent which is same as Nguyen et al. [31]. The comparison between our graph and Nguyen et al. [31] shows that both graphs have a similar trend but our repetition percentage average for all nodes is comparatively higher.
5.4 RQ2: Repetitiveness of PASGs

As explained in the approach section, RQ2 is not exactly same as that of Nguyen et al. [31] due to using PASG instead of PVSG. For these questions we first slice each routine’s PDG on each argument, which we call as PASG. Then we find the repetitiveness of slices in the source code dataset. These questions answer the same four parts as RQ1 but looking at repetition for subroutines instead.

5.4.1 RQ2a: Repetitiveness Within a Project

The trend for subroutines repetitiveness shown in Figure 5.10 is very similar to the trend of RQ1’s first part. The figure shows an exponentially decreasing trend for repetitiveness with the increase in repetition count. Our experiment shows about 17% subroutines repeat once, 4.3% twice and 2.6% thrice in a project. The corresponding repeat percentages for Nguyen et al. [31] are not reported in their paper. About 24% of subroutines are repeated between 1 and 8 times and the repeat percentage for subroutines with more than 8 repetitions, is less than 0.1%.

5.4.2 RQ2b: Repetitiveness Across Projects

The repetitiveness of subroutines across projects is shown in Figure 5.11. This figure also shows decreasing trend with increasing repetition count. However, the percentage for subroutines which repeat once across projects is 10.9% which is less than the percentage for the same repeat count within a project. The trend is similar to what we observed in entire routines case in RQ1 where the percentage within project is more than across projects for single repetition. Again most of the repetitions occur for repetition count less than equal to 13 and beyond that repetition percentage approaches zero. From this we can conclude that repetition trends for routines and subroutines are quite similar with only little variations.
Figure 5.10: % of entire subroutines realized elsewhere within a project.

Figure 5.11: % of subroutines realized elsewhere in other projects.
5.4.3 RQ2c: Repetitiveness by Graph Size

Subroutines with size three repeat more than 45% of the time as shown in Figure 5.12. The subroutines with smaller sizes are more likely to be repeated as can be seen in the figure. As the size increases the repetition percentage does not vary much though the decrease and variations are more than what we observed in the case of routines in RQ1. When compared to Nguyen et al. [31], our decrease is slightly less varying and has less spikes or sudden variations. The maximum repetition for both occurs when the size of graph is under 10 nodes.

5.4.4 RQ2d: Repetitiveness by Number of Control Nodes

The trend for repetition by the number of control nodes is smoother than the repetition by Graph size. The decrease after 8 control nodes is also much less than before it as can be seen in Figure 5.13. The subroutines with no control nodes tend to repeat more with as high as 44% repetition. From this we can conclude that less complex subroutines are more likely to get repeated.
but after a certain complexity level repetition almost becomes independent of the number of control nodes in the method.

5.5 Threats to Validity

The GitHub dataset we used includes data from branches other than master. As a result it may lead to higher repetition percentage than there actually is. This may also skew the final results slightly. A second threat to results may come from hashing algorithm, we have used SHA1 which is 160 bits and has a collision probability which can affect repetition percentage, though the chances of that happening are minuscule.
CHAPTER 6. CONCLUSION

In this work we presented a large scale study on routine repetitiveness in open source projects. The study is a partial reproduction of a prior study by Nguyen et al. [31] and looks at how often (sub)routines repeat both within a project and across projects. Our results show that routines are more likely to be repeated within a project than across projects. A total of 24% routines repeat within a project while a total of 19% repeat across projects. We also performed a similar study on subroutines, obtained by slicing routine PDGs on their corresponding arguments which we call per argument slicing subgraph. The trend of subroutine repetition is similar to full routine repetition. A total of 24% subroutines repeat within a project while only 21% repeat across projects. Some results based on graph size and number of control nodes also present interesting insights about repetition of source code.

When compared with Nguyen et al. [31], we see similar trends. The exact repetition percentage for both RQs is higher in our case. For RQ1 about 24% and 19% of routines repeat 2–7 times within and across projects in our case. The similar percentages from Nguyen et al. [31] are 12.1% and 4% respectively. The repetition percentage for RQ1 (excluding trivial cases) based on graph size and number of control nodes is 35% in our case and 9% in their case. For RQ2 Nguyen et al. [31] only have the result for repetition by graph size and is 4% on average. For us the same is close to 35%. These differences seen between the two studies indicates the need for further exploration.

A direct application of these results could be in the synthesis of behavioral specifications of API methods. If an API method is composed of subroutines and those subroutines already contain specifications in another project, perhaps they could be re-used for the target API. In the future we plan to use this insight to automatically generate specifications for such API methods.
BIBLIOGRAPHY


