Sampling Code Clones from Program Dependence Graphs with GRAPLE

Tim A. D. Henderson
tadh@case.edu

Andy Podgurski
podgurski@case.edu

Department of Electrical Engineering and Computer Science
Case Western Reserve University
Cleveland, Ohio, United States of America

ABSTRACT

We present GRAPLE, a method to generate a representative sample of recurring (frequent) subgraphs of any directed labeled graph(s). GRAPLE is based on frequent subgraph mining, absorbing Markov chains, and Horvitz-Thompson estimation. It can be used to sample any kind of graph representation for programs. One of many software engineering applications for finding recurring subgraphs is detecting duplicated code (code clones) from representations such as program dependence graphs (PDGs) and abstract syntax trees. To assess the usefulness of clones detected from PDGs, we conducted a case study on a 73 KLOC commercial Android application developed over 5 years. Nine of the application's developers participated. To our knowledge, it is the first study to have professional developers examine code clones detected from PDGs. We describe a new PDG generation tool jpdg for JVM languages, which was used to generate the dependence graphs used in the study.

CCS Concepts

•**Mathematics of computing** → **Markov processes;** *Graph enumeration;* •**Social and professional topics** → *Software management;* •**Software and its engineering** → *Software maintenance tools; Designing software;*

Keywords

frequent subgraph mining, program dependence graphs, sampling estimation, clone detection, bug mining, Markov chains

1. INTRODUCTION

Code clones are similar fragments of program code [35]. They can arise from copying and pasting, using certain design patterns or certain APIs, or adhering to coding conventions, among other causes. Code clones create maintenance hazards, because they often require subtle context-dependent adaptation and because other changes must be applied to each member of a clone class. To manage clone evolution the clones must first be found. Clones can be detected using any program representation: source code text, tokens, abstract syntax trees (ASTs), flow graphs, dependence graphs, etc. Each representation has advantages and disadvantages for clone detection.

PDG-based clone detection finds *dependence clones* corresponding to recurring subgraphs of a program dependence graph (PDG) [22, 23]. Since PDGs are oblivious to semantics preserving statement reorderings they are well suited to detect *semantic* (functionally equivalent) clones. A number of algorithms find clones from PDGs [22, 23, 25, 5, 11, 29, 30, 19, 33]. However, as Bellon [3] notes, "PDG based techniques are computationally expensive and often report non-contiguous clones that may not be perceived as clones by a human evaluator." Most PDG-based clone detection tools are biased, detecting certain clones but not others.

The root cause of scalability problems with PDG-based clone detection is the number of dependence clones. Section 3 illustrates this with an example in which we used an unbiased frequent subgraph mining algorithm [24] to detect all dependence clones in Java programs. In programs with about 70 KLOC it detected around 10 million clones before disk space was exhausted. Processing all dependence clones is impractical even for modestly sized programs.

Instead of exhaustively enumerating all dependence clones, an unbiased random sample can be used to statistically estimate parameters of the whole "population" of clones, such as the prevalence of clones exhibiting properties of interest. For these reasons, we developed a statistically unbiased method for *sampling* dependence clones and for *estimating* parameters of the whole clone population.

We present GRAPLE (GRAph samPLE)[1], a method to generate a representative sample of recurring subgraphs of any directed labeled graph(s). It can be used to sample subgraphs from any kind of program graph representation. GRAPLE is not a general purpose clone detector but it can answer questions about dependence clones that other PDG-based clone detection tools cannot. We conducted a preliminary case study on a commercial application and had its developers evaluate whether the sampled subgraphs represented code duplication. To our knowledge, it is the first study to have professional programmers examine dependence clones.

Contributions:

1. GRAPLE: a framework for unbiased sampling of frequent subgraphs of large graphs such as PDGs and for esti-

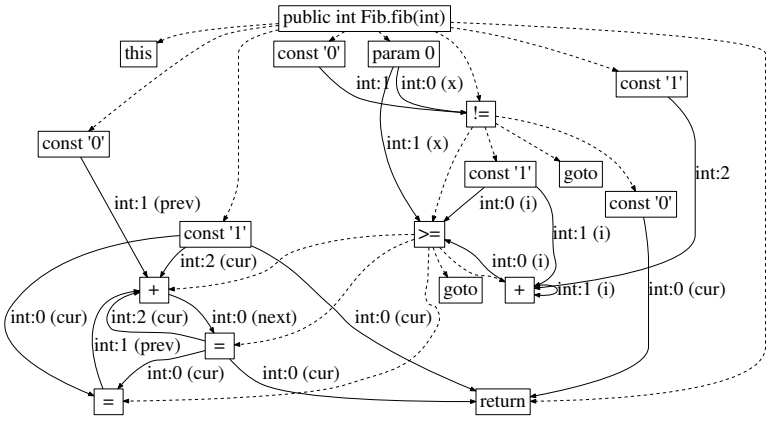
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

SWAN'16, November 13, 2016, Seattle, WA, USA
ACM. 978-1-4503-4395-4/16/11...
http://dx.doi.org/10.1145/2989238.2989241

[1] https://github.com/timtadh/graple

```
1  public class Fib {
2    public int fib(int x) {
3      int prev = 0;
4      int cur = 1;
5      if (x == 0) {
6        cur = 0;
7      } else {
8        for (int i = 1; i < x; i++) {
9          int next = prev + cur;
10         prev = cur;
11         cur = next;
12       }
13     }
14     return cur;
15   }
16 }
```

(a) Program to compute terms from the Fibonacci sequence.



(b) Procedure dependence graph of Figure 1a.

Figure 1: Example procedure and procedure dependence graph. Solid lines are data dependencies. Dashed lines are control dependencies. The labels on the data dependencies indicate the type of the data and its usage context (ie. its parameter number). The data dependencies are also annotated with variable names to aid in readability. ©Tim Henderson

mating statistics characterizing the whole population of frequent subgraphs. (Sec. 3.3, 3.5)

2. A case study in which `GRAPLE` was applied to a commercial Android application and in which its output was examined by developers. (Sec. 4)
3. `jpdg`: a new procedure dependence graph generator for JVM languages. (Sec. 2)

`GRAPLE` has applications in bug mining, test case selection, and bioinformatics. The sampling algorithm also applies to frequent item sets, subsequences, and subtrees allowing code clone sampling from tokens and ASTs.

2. A REVIEW OF DEPENDENCE GRAPHS

A *program dependence graph* (PDG) [10] represents possible dependence relationships between statements in a program, with vertices representing statements and directed edges representing control and data dependences. Informally, a statement s_1 is *control dependent* on a statement s_2 if s_2 is a branch predicate that controls the execution of s_1. A statement s_1 is *data dependent* on a statement s_2 if a value assigned to a variable x at s_2 can later be accessed from x at s_1. (This requires that all control flow paths from s_2 to s_1 do not assign a new value to x.)

PDGs approximate semantic dependencies between statements. They are not affected by reordering statements in ways which preserve the semantics. Horwitz *et al.*[16] showed that, under certain assumptions, if the PDGs of two programs are isomorphic then the programs are equivalent. Given Horwitz's result and related results from Podgurski [31, 32], the PDG is a good representation to detect code clones with renamed variables, semantics-preserving statement reorderings, and unrelated code insertions.

An important variant of the program dependence graph is the *system dependence graph* (SDG) [17], which consists of *procedure dependence graphs* (pDGs), each representing an individual subprogram, connected by *inter-procedural dependence edges* representing subprogram calls. The case study described in Section 4 involves mining pDGs. Figure 1 shows an example procedure dependence graph. The dotted lines indicate control dependencies and the solid lines indicate data dependencies.

To compute the pDGs used in this paper a prototype tool named `jpdg` was built. A successor to JavaPDG [38], `jpdg` was created to improve the PDG representation for code mining purposes. For instance, most dependence graphs place the arguments to operations in the vertices of the graph. In the graphs produced by `jpdg` these are associated with the edges (see Figure 1). `jpdg` is built on top of the Soot optimization framework [12][2], and non-constant vertices map to Jimple instructions. Control dependencies are computed using Cytron's method [8]. Data dependences are computed using Upward Exposed Uses analysis [26].

3. SAMPLING DEPENDENCE CLONES

A *dependence clone* is duplicated code detected from the program dependence graph (PDG). To detect dependence clones, identify subgraphs appearing in multiple locations in the PDG. The problem of finding recurring subgraphs is called *Frequent Subgraph Mining* (FSGM) [7]. Frequent subgraph miners search for subgraphs which recur k times with $k > 1$.

Applying standard mining algorithms to program graphs is not straightforward. Software engineers are potentially interested in subgraphs with very few repetitions. Small frequency thresholds are uncommon in the applications typically considered in the data-mining literature. Our experiments on `jgit`[3] ($\sim$ 72 KLOC) have found that with a minimum frequency setting of 5, there are over 11.8 million frequent subgraphs. We were unable to completely mine `jgit` as we exhausted our disk space storing the patterns ($\sim$ 1 TB). This mining attempt, which used the vSiGraM algorithm [24], took over 12 days. We made another attempt where the patterns were simply logged to the console instead of stored. During this attempt, over 350 million patterns were discovered before the process was killed after 10 days. The application considered in Section 4 had similar results ($>$ 10 million frequent patterns before disk space exhaustion and $>$400 million patterns after 10 days of mining).

[2]http://sable.github.io/soot/
[3]https://github.com/eclipse/jgit

So, not only is it impractical in time and space to use an algorithm like vSiGraM to do code clone detection it would not be possible for all patterns to be stored and individually examined.

3.1 How Frequent Subgraph Mining Works

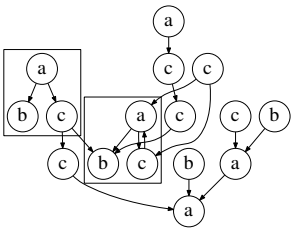
Standard frequent subgraph miners search for subgraphs that recur at least a specified number of times in a graph database, which may contain one very large graph or many smaller graphs [7]. Conceptually, miners work by enumerating the subgraphs of the graphs in the database by traversing *frequent connected subgraph lattice* (see Fig. 2b). As each subgraph is found its *support* must be computed. Informally, the support of a subgraph is the number of *embeddings* that it has in the graph database. There are a few complications to this definition which will be explained in Section 3.4.

Using subgraph-isomorphism checks to count support is expensive. A faster way to count support is to store the embeddings of each subgraph. The stored embeddings can also be used during the subgraph enumeration process to reduce the number of candidate patterns. As each subgraph is produced it is "canonicalized." The canonicalization process always puts isomorphic graphs into the same form and thus neatly solves the graph isomorphism problem. We use Bliss [21] for canonicalization.

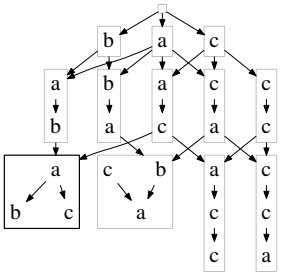
3.2 From Mining to Sampling

Large PDGs may have a huge number of frequent subgraphs, but in applications of clone detection it may be unnecessary to consider them all. We focus on two use-cases: (1) developers want to manually examine mined clones, e.g., to propose refactorings, and (2) developers and researchers want to answer questions about the whole population of clones that *could* be mined given enough time and storage space.

In the first use-case, developers will have limited time to examine mined clones. Thus, they will generally prefer to consider a small, diverse set of clones. In the second use-case, the questions posed could be either objective (e.g., "What percentage of our code base is covered by one or more frequent subgraphs?") or subjective (e.g., "What proportion of potential dependence clones do our programmers want to refactor?"). Both kinds of questions can often by answered by examining a representative sample of frequent



(a) A labeled directed graph



(b) 2-frequent connected subgraph lattice

Figure 2: Figure 2b is a connected subgraph lattice of Figure 2a including only subgraphs with 2 or more embeddings in Figure 2a. The boxed nodes in the graph show the embeddings of the boxed subgraph in the lattice. (See Section 3.4) ©Tim Henderson

```
1  # param G : the graph being mined
2  # param bottom : lattice node for the empty subgraph
3  # param min_support : int, minimum number of embeddings
4  # returns : leaf node of the frequent connected subgraph
5  #           lattice which is a maximal frequent subgraph
6  def walk(G, bottom, min_support):
7    v = u = bottom
8    while v is not None:
9      u = v
10     v = rand_select(get_children(G, u, min_support))
11   return u
12
13 # param u : a lattice node
14 # returns : a list of lattice nodes which are 1 edge
15 #           extensions of u
16 def get_children(G, u, min_support):
17   exts = list()
18   for emb in u.embeddings:
19     for a in embedding.V:
20       for e in G.edges_to_and_from(a):
21         if not emb.has_edge(e):
22           exts.append(emb.extend_with_edge(e))
23   groups = group_isomorphs(exts)
24   return [ LatticeNode(lbl, group)
25            for lbl, group in groups.iteritems()
26            if len(group) >= min_support ]
27
28 # param subgraphs : a list of subgraphs of G
29 # returns : map label -> list of isomorphic subgraphs.
30 def group_isomorphs(subgraphs):
31   isomorphs = dict()
32   for sg in subgraphs:
33     label = bliss.canonical_label(sg)
34     if label not in isomorphs:
35       isomorphs[label] = list()
36     isomorphs[label].append(sg)
37   return { label: minimum_image_supported(group)
38            for label, group in isomorphs.iteritems() }
```

Listing 1: `GRAPLE`'s sampling procedure

subgraphs. If care is taken in designing the method to select the sample, then *statistical estimation* [39] techniques can be used to estimate unbiased answers.

We developed `GRAPLE` to address both use-cases. It samples randomly from the space of *maximal frequent subgraphs* (a frequent subgraph is maximal if no larger frequent subgraph can be constructed from it). The sampling procedure is well defined and allows us to compute *selection probabilities* for subgraphs, which can be used in statistical estimators such as the *Horvitz-Thompson (HT) unequal probability estimator* [39]. Furthermore, developers can use `GRAPLE` to collect small, diverse sets of potential dependence clones from an entire code base or from parts of interest. The same basic algorithm can also be applied to item-set, sub-sequence, and sub-tree mining.

3.3 Sampling Maximal Frequent Subgraphs

All frequent pattern miners traverse the frequent pattern lattice, which for subgraph miners means the frequent connected subgraph lattice (see Figure 2). Each node in the lattice represents a frequent subgraph, with the directed edges connecting A to B if adding one edge to A produces a graph isomorphic to B (see Section 3.4). Miners traverse the lattice in either a breadth first or depth first manner to find all of the frequent subgraphs.

Since we seek a random sample of the maximal frequent subgraphs, it is unnecessary to traverse the whole lattice. Instead, we make n partial traversals where n is the desired sample size. Each traversal is an unweighted random walk over the lattice, which proceeds from smaller frequent sub-

graphs to larger ones. The walk terminates when it reaches a maximal frequent subgraph and that pattern and its embeddings are added to the sample. This procedure is outlined in Listing 1.

The most expensive part of frequent subgraph mining is extending a pattern with e edges to patterns with $e+1$ edges and computing support of each of those extensions. Extensions are computed on lines 17-22 by considering all possible extensions for each embedding. The lattice nodes hold a list of their embeddings, making this computation relatively cheap. After all possible extensions are computed they are grouped by their canonical labeling (lines 30-36) as computed by Bliss [21]. The groups are then minimized (lines 37-38) to remove many of the overlapping embeddings using minimum image support [4]. If a group contains enough embeddings to be considered frequent then a lattice node is created and it is returned as a child (lines 24-26).

Generalizing results from the sample of maximal frequent patterns to the population of all maximal frequent pattern requires taking into account *unequal sample inclusion probabilities*. In order to do this a correction factor or weight is applied to each sampled value. The weight for the value y_i of the i^{th} population unit is the inverse of the unit's probability π_i of inclusion in the sample. With these weights, the *Horvitz-Thompson (HT) estimator* [15], denoted $\hat{\tau}_\pi$, can be used to make an unbiased estimate of the population total τ for the study variable, and a simple variant $\hat{\mu}_\pi$ can be used to unbiasedly estimate the population mean μ. Let ν be the number of distinct units in the sample. Then

$$\hat{\tau}_\pi = \sum_{i=1}^{\nu} \frac{y_i}{\pi_i} \tag{1}$$

Observe that units that are rarely sampled will have their values boosted substantially by the weights $1/\pi_i$, while units which are commonly sampled have their values boosted less. Thompson [15] provides formulas for the HT estimator's mean and variance and for computing confidence intervals for estimates.

When sampling n units with replacement, the probability π_i that the i^{th} population unit is included in the sample can be computed from the probability p_i that unit i is selected on a particular random walk:

$$\pi_i = 1 - (1 - p_i)^n \tag{2}$$

3.4 Formal Definitions

Before describing how to compute the selection probabilities (the p_i's above) a few definitions are needed.

A *directed labeled graph* (labeled digraph) G is a set of vertices V, a set of edges $E = V \times V$, and a labeling function which maps vertices (or edges) to labels $l : V|E \to L$. E can be represented by a matrix $\mathbf{E}$. $\mathbf{E}_{i,j} = 1$ if and only if there is an edge from vertex v_i to vertex v_j, otherwise it is 0.

H is a subgraph of G ($H \sqsubseteq G$), if and only if an injective mapping $m : V_H \to V_G$ exists such that:

1. All vertices in H map vertices in G with the same label:
   $\forall\ v \in V_H\ [l_H(v) = l_G(m(v))]$
2. All edges in H are in G:
   $\forall\ (u, v) \in E_H\ [(m(u), m(v)) \in E_G]$
3. All edge labels match:
   $\forall\ (u, v) \in E_H\ [l_H(u, v) = l_G(m(u), m(v))]$

Such a mapping m is known as an *embedding*. A digraph A is *isomorphic* to another digraph B, $A \cong B$, if $A \sqsubseteq B$ and $B \sqsubseteq A$. The *isomorphism class* of a subgraph H is the set of all of the subgraphs of G isomorphic to H with distinct mappings, denoted $[\![H]\!] = \{H' \sqsubseteq G : H' \cong H \wedge m_{H'} \neq m_H\}$.

The subgraph relation $\cdot \sqsubseteq \cdot$ induces a *connected subgraph lattice* $\mathcal{L}_G$ representing all possible ways of constructing G. $\mathcal{L}_G$ can itself be viewed as a directed graph where each vertex u represents a unique connected[4] subgraph of G. An edge exists between u and v if adding one edge to u creates a subgraph $u + \epsilon$ which is isomorphic to v, $v \cong u + \epsilon$. A *k-frequent connected subgraph lattice* k-$\mathcal{L}_G$ contains only those subgraphs which have at least k embeddings in G, see Figure 2. Finally, a *pattern* refers to the isomorphism class $[\![H]\!]$ of a subgraph H.

One definition of the *support*, or *frequency*, of a pattern $[\![H]\!]$ in G is the number $|[\![H]\!]|$ of unique embeddings it has in G. However, many graphs have *automorphisms*, which occurs when a graph is isomorphic to itself, and each automorphism results in a unique embedding. Including automorphisms in support overstates the true frequency of many graphs. A tractable solution is *minimum image support* (MIS) [4], which is an upper bound on the size of the maximum independent set of non-overlapping embeddings. MIS is found by computing the unique embeddings for each vertex individually. The vertex with the fewest unique embeddings determines the support of the pattern.

3.5 Computing the Probability of Selecting a Maximal Subgraph

In order to use the HT estimator outlined in the previous section, it is necessary to determine the probability p_i that the i^{th} maximal frequent pattern $[\![H_i]\!]$ is selected on a random walk of the k-frequent connected subgraph lattice (k-$\mathcal{L}_G$). We compute these probabilities using the theory of Markov chains.

A *finite-state Markov chain* [13] consists of a finite set of states, $S = \{s_1, \ldots, s_n\}$, and a matrix $\mathbf{P}$, called the *transition matrix*, where $\mathbf{P}_{i,j}$ gives the probability of a state transition from s_i to s_j. A Markov chain moves from state to state according to the probabilities in the transition matrix. A random walk in a graph G can be viewed as a Markov chain whose set of states S corresponds to the vertex set V_G. An *absorbing Markov chain* [13] is a special type of Markov chain which always ends in a state that cannot be exited, called an *absorbing state*.

To construct an absorbing Markov chain from the lattice k-$\mathcal{L}_G$, let the states of the chain be the vertices of the lattice (i.e., the frequent patterns $[\![H_i]\!]$). To model how the algorithm in Listing 1 transitions from one lattice node to the next by uniformly selecting a neighboring node, let the transition probability for an edge $v_i \to v_j$ be the reciprocal of the out-degree of v_i:

$$\mathbf{P}_{i,j} = \begin{cases} \frac{1}{\sum_k \mathbf{E}_{i,k}} & \text{if } \mathbf{E}_{i,j} = 1 \\ 1 & \text{if } i = j \wedge v_i \text{ is maximal} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

The selection probability p_i of $[\![H_i]\!]$ is the probability that state s_i absorbs the Markov process starting at the bottom lattice node. To compute p_i, arrange the transition matrix $\mathbf{P}$ into *canonical form* such that the transient states come

[4]In this paper, *connected* ignores edge direction (see Fig. 2).

before the absorbing states:

$$\mathbf{P} = \begin{array}{c} \\ \text{TR.} \\ \text{ABS.} \end{array} \begin{array}{c} \begin{array}{cc} \text{TR.} & \text{ABS.} \end{array} \\ \begin{bmatrix} \mathbf{Q} & \mathbf{R} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \end{array} \quad (4)$$

$\mathbf{Q}_{i,j}$ is the probability of transitioning from a transient state s_i to transient state s_j. $\mathbf{R}_{i,j}$ is the probability of transitioning from transient state s_i to absorbing state s_{t+j} where t is the number of transient states. $\mathbf{I}$ is the identity matrix and $\mathbf{0}$ is the zero matrix, as once a Markov process enters an absorbing state it never leaves. The probability of a process starting at the bottom of the lattice s_0 and being absorbed by state s_i with zero or more transitions ($\xrightarrow{\star}$) is [13]:

$$p_i = \Pr[s_0 \xrightarrow{\star} s_i] = (\mathbf{P}^{\infty})_{0,i} = ((\mathbf{I} - \mathbf{Q})^{-1}\mathbf{R})_{0,(i-t)} \quad (5)$$

3.5.1 Computing p_i with a submatrix of $\mathbf{P}$

To compute p_i using Equation 5, the entire matrix $\mathbf{P}$ needs to be constructed. It turns out only a *submatrix* of $\mathbf{P}$ is needed to compute the probability p_i of selecting a frequent pattern $[\![H_i]\!]$ using the algorithm in Listing 1. The required rows and columns of $\mathbf{P}$ correspond to the vertices of k-$\mathcal{L}_G$ which are subgraphs of H_i.

THEOREM 1. *Let $[\![H_i]\!]$ be a maximal k-frequent pattern sampled from k-$\mathcal{L}_G$. Let s_i be the corresponding state in the Markov chain formed from k-$\mathcal{L}_G$. The selection probability of $[\![H_i]\!]$, $p_i = ((\mathbf{I}-\mathbf{Q})^{-1}\mathbf{R})_{0,(i-t)}$, can be computed from the submatrix of $\mathbf{P}$ that includes only the rows and columns that correspond to subgraphs of H_i.*

PROOF. See supplementary materials[5] □

Computing the submatrix of $\mathbf{P}$ for H_i requires finding every connected subgraph of H_i which is much less work than mining all frequent subgraphs of G. Unfortunately, computing the submatrix is only tractable for subgraphs with fewer than 20 edges (which can induce submatrices as large as $2^{20} \times 2^{20}$). In future work, we intend to estimate p_i rather than compute it exactly, in order to handle much larger subgraphs. Note that our sampling procedure does not have a size limitation (it has found frequent subgraphs with over 100 edges); only the probability computation has this limitation.

4. CASE STUDY: ASSESSMENT OF CLONE RELEVANCE

We conducted a preliminary case study to see if GRAPLE could help us assess the usefulness of PDG-based code clone detection to developers. Our study assessed the relevance of dependence clones in a commercial Android application with $\sim$ 73 KLOC that had been under continuous development for 5 years. Nine developers participated in the study. The study was conducted as a survey which asked a set of 10 questions about each group of mined clones (two questions are in Figure 3). The clones were displayed as both graphs and highlighted regions of source code. The goal was to estimate the proportions of frequent subgraphs that represent duplicate code and that developers would act upon.

The survey involved 104 dependence clone groups sampled using GRAPLE, which was configured to require frequent subgraphs to have minimum support 5 and to contain at least 8 vertices. As sampling was done *with replacement*, 415 patterns were sampled, with 122 unique patterns. Of the 122, 104 with fewer than 20 edges were retained as discussed in Section 3.5.1. The sampling was done on an computer with an 8 core Intel Xeon processor, 64 GB of main memory and a 250 GB hard drive. It took 138 seconds to collect the samples and 23.2 hours to compute the selection probabilities used in the HT estimator. Computing the selection probabilities was made possible by using SuiteSparse[6] for large sparse matrix inversion.

At the beginning of the study, the participants were given a presentation on code clones. Code clones were somewhat familiar to these developers, as they utilize a commercial static analysis tool, SonarQube, which makes use of a clone detector based on Hummel *et al.*'s algorithm [18]. SonarQube detected none of the clones the developers reviewed in the study. In an ideal study each clone group would have been reviewed by each participant, but in order to maximize the number of clones reviewed, each clone group was only reviewed by a single participant.

4.1 Study Results

All of the 104 clone groups were reviewed by the participating developers. Despite the approximately unbiased nature of the modified Horvitz-Thompson (HT) estimator we used [36], its results can still be skewed if there is high variance in the inclusion probabilities. The best way to address such skew is to collect more data. As this was not possible the next best option of removing the outliers was taken. Thus two of the clone groups with outlying selection probabilities were discarded.

For survey question 1, the estimate of the proportion of all mineable dependence clones for which the answer would be **Yes** if they were all examined was 61%, with a 95% confidence interval of $42\% - 78\%$. For survey question 2, the estimate of the proportion of all mineable clones for which one of the "action answers" (2.a, 2.b, 2.c, and 2.e) would be given if all of the clones were examined was 14% (which is smaller than the sample proportion of 33%), with a 95% confidence interval of $0\% - 33\%$.

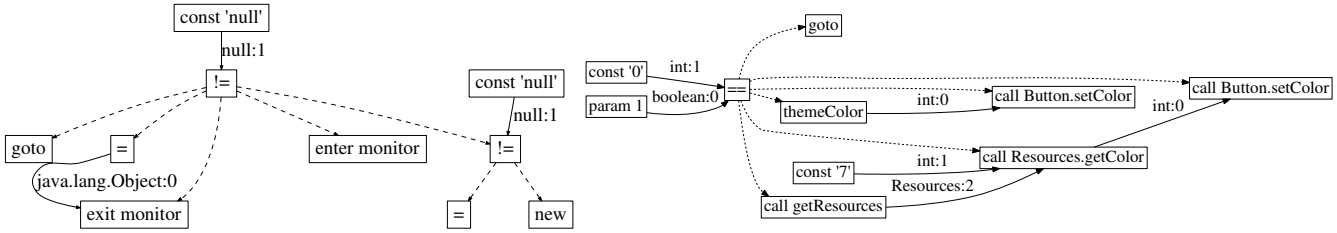
It should be emphasized that the results for survey question 1 do not actually mean that our frequent subgraph miner produced erroneous results 39% of time. It did in fact identify only 5-frequent PDG subgraphs. However, the

[5] http://hackthology.com/pdfs/swan-2016-supplemental.pdf

[6] http://faculty.cse.tamu.edu/davis/suitesparse.html. See Davis' 2004 paper for details [9].

1. Do the highlighted portions of the code fragments, in conjunction with the associated graph, represent duplicated, similar or cloned code?
   (a) Yes
   (b) No
2. If you answered **Yes** to question 1, would you:
   (a) Create a story card to refactor this code?
   (b) Add a comment to consider refactoring on next change?
   (c) Add a note about duplicate code even if it cannot be refactored?
   (d) Ignore it?
   (e) Take some other action?

Figure 3: The two critical survey questions



(a) Double Checked Locking. Used to create singletons. Not considered a code clone by the reviewer.

(b) Setting a button color depending on theme choice. Considered a code clone by the reviewer.

Figure 4: Two clones discovered in the study. Application details have been obfuscated. ©Tim Henderson

developers had their own subjective criteria for deciding whether the corresponding code was duplicated. For example, Figure 4 shows two simplified clones sampled from the application. Figure 4a was identified by the developer who reviewed it as the *double-checked locking* pattern [37]. Although widely used throughout the code base, the reviewer indicated that it was too general to be a code clone, because each instance exhibited context-specific specialization. Figure 4b represents a class of different clones involving user-interface state modifications. In this clone, the color of a button is changed based on the current theme. As all elements of the application are themed to support multiple brands of the program, this code was duplicated in many locations (often non-contiguously). The reviewers recommended it for refactoring to centralize the theming decision.

Thus, the results for survey questions indicate that (a) about 61% of mineable dependence clones in the project would be judged by developers to be duplicate code and (b) the developers would want to take action for only about 14% of mineable clones. Assuming that the developers' judgments about the clones were justified, the difference between these two estimates suggests that for this project, additional filtering is needed to eliminate clones that are not of interest to developers. Further study of the sampled dependence clones and discussions with the developers about them might suggest what filtering criteria are needed. Note that in developers' responses to a followup questionnaire in our study, they indicated that they felt the exercise was useful and that they would like to periodically review new findings from a PDG-based tool.

Further investigations are needed to fully understand these results in the context of other clone detection techniques. Unlike many detectors ours did not employ any filtering or normalization heuristics, making direct comparisons to previous results difficult. In future studies, `GRAPLE` could be applied to AST and token representations, allowing a direct comparison. The effect of filtering and normalization can also be estimated using `GRAPLE`.

5. RELATED WORK

Code clone detection and management have been rich areas of research and there are several recent surveys available [3, 34, 33, 35]. Krinke's Duplix algorithm [23] and Komondoor and Horwitz's algorithm [22] are early examples of detecting code clones from PDGs. Gabel et al. [11] introduced an alternate formulation by mapping PDG's to abstract syntax trees and detecting clones with DECKARD [20]. Higo and Kusomoto extended Komondoor's algorithm to detect contiguous clones [14]. ModelCD from Pham et al. [30] detect code clones in Matlab/Simulink models by converting them to labeled directed graphs and finding code clones with vSiGraM [24]. Nguyen et al. [28, 27] created two code completion tools based on the code mining tool in [29]. Both tools work by first building a graph database of *groums* which are object/object interaction graphs.

Many algorithms have been developed for frequent subgraph mining [7]. Our technique aligns most directly with approaches which have employed Markov Chain based sampling strategies. Musk [1] constructs a Markov process which samples the maximal subgraphs *uniformly* given enough time. Al Hasan et al. proposed a Metropolis-Hastings approach [2] to uniformly sample all frequent subgraphs (as opposed to just the maximal subgraphs). Unfortunately, we have found both the Metropolis-Hastings approach and Musk to be unworkable on large program dependence graphs. ORIGAMI [6] uses a random walk on the connected subgraph lattice to collect a sample of the frequent maximal subgraphs in a similar manner to `GRAPLE`. It then prunes the sample to include only the most *representative* subgraphs. However unlike `GRAPLE`, ORIGAMI does not provide a means of computing sampling probabilities.

6. CONCLUSION

We have presented `GRAPLE`, a framework for randomly sampling unique frequent subgraph from directed labeled graphs. Our sampling method enables unbiased estimation of statistics characterizing the whole population of frequent subgraphs (without enumerating it). The results of our case study suggest that `GRAPLE` will prove useful to software engineering researchers and to developers who apply advanced analytical methods to better understand large code bases. In future work, we plan to estimate the sample inclusion probabilities needed for Horvitz-Thompson estimation rather than computing them exactly to enable studies involving larger patterns.

7. REFERENCES

[1] M. Al Hasan and M. Zaki. Musk: Uniform Sampling of k-Maximal Patterns. In *ICDM*, pages 650–661. SIAM, apr 2009.

[2] M. Al Hasan and M. J. Zaki. Output Space Sampling for Graph Patterns. *VLDB*, 2(1):730–741, aug 2009.

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, sep 2007.

[4] B. Bringmann and S. Nijssen. What is frequent in a single graph? In *LNAI*, pages 858–863, 2008.

[5] R.-Y. Chang, A. Podgurski, and J. Yang. Discovering Neglected Conditions in Software by Mining Dependence Graphs. *IEEE Transactions on Software Engineering*, 34(5):579–596, sep 2008.

[6] V. Chaoji, M. Al Hasan, S. Salem, J. Besson, and M. J. Zaki. ORIGAMI: A Novel and Effective Approach for Mining Representative Orthogonal Graph Patterns. *Stat. Anal. Data Min.*, 1(2):67–84, jun 2008.

[7] H. Cheng, X. Yan, and J. Han. Mining Graph Patterns. In *Frequent Pattern Mining*, pages 307–338. Springer International Publishing, 2014.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, oct 1991.

[9] T. A. Davis. Algorithm 832: UMFPACK V4.3—an Unsymmetric-pattern Multifrontal Method. *ACM Transactions on Mathematical Software*, 30(2):196–199, jun 2004.

[10] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[11] M. Gabel, L. Jiang, and Z. Su. Scalable Detection of Semantic Clones. In *ICSE*, pages 321–330, 2008.

[12] E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java Bytecode Using the Soot Framework : Is It Feasible? *Lecture Notes in Computer Science*, 1781:18–34, 2000.

[13] C. M. Grinstead and J. L. Snell. *Introduction to Probability*. American Mathematical Society, Providence, RI, 2 edition, 1997.

[14] Y. Higo and S. Kusumoto. Code Clone Detection on Specialized PDGs with Heuristics. In *CSMR*, pages 75–84, mar 2011.

[15] D. G. Horvitz and D. J. Thompson. A Generalization of Sampling Without Replacement From a Finite Universe. *Journal of the American Statistical Association*, 47(260):pp. 663–685, 1952.

[16] S. Horwitz. Identifying the Semantic and Textual Differences Between Two Versions of a Program. *SIGPLAN Notes*, 25(6):234–245, jun 1990.

[17] S. Horwitz, J. Prins, and T. Reps. On the Adequacy of Program Dependence Graphs for Representing Programs. In *POPL*, pages 146–157, 1988.

[18] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: Incremental, distributed, scalable. In *ICSM*, 2010.

[19] B. Hummel, E. Juergens, and D. Steidl. Index-based Model Clone Detection. In *IWSC*, pages 21–27, 2011.

[20] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105, 2007.

[21] T. Junttila and P. Kaski. Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs. In *Workshop on Algorithm Engineering and Experiments*, pages 135–149, jan 2007.

[22] R. Komondoor and S. Horwitz. Using Slicing to Identify Duplication in Source Code. *Lecture Notes In Computer Science*, 2126:40–56, 2001.

[23] J. Krinke. Identifying similar code with program dependence graphs. In *WCRE*, pages 301–309, 2001.

[24] M. Kuramochi and G. Karypis. Finding Frequent Patterns in a Large Sparse Graph*. *Data Mining and Knowledge Discovery*, 11(3):243–271, nov 2005.

[25] C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In *KDD*, pages 872–881, 2006.

[26] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.

[27] A. T. Nguyen and T. N. Nguyen. Graph-based Statistical Language Model for Code. In *ICSE*, pages 858–868, Piscataway, NJ, USA, 2015.

[28] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based Pattern-oriented, Context-sensitive Source Code Completion. In *ICSE*, pages 69–79, 2012.

[29] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *ESEC/FSE*, page 383, 2009.

[30] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Complete and accurate clone detection in graph-based models. In *ICSE*, pages 276–286, 2009.

[31] A. Podgurski and L. Clarke. The Implications of Program Dependencies for Software Testing, Debugging, and Maintenance. In *TAV3*, pages 168–178, 1989.

[32] A. Podgurski and L. A. Clarke. A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, sep 1990.

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

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

[35] C. K. Roy, M. F. Zibran, and R. Koschke. The vision of software clone management: Past, present, and future. In *CSMR-WCRE*, pages 18–33, 2014.

[36] C.-E. Särndal, B. Swensson, and J. Wretman. *Model Assisted Survey Sampling*. Springer-Verlag, 1992.

[37] D. C. Schmidt and T. Harrison. Double-Checked Locking: An Optimization Pattern for Efficiently Initializing and Accessing Thread-safe Objects. In *PLOP*, chapter 7, pages 363–376. 1998.

[38] G. Shu, B. Sun, T. A. D. Henderson, and A. Podgurski. JavaPDG: A New Platform for Program Dependence Analysis. In *International Conference on Software Testing, Verification and Validation*, pages 408–415. IEEE, 2013.

[39] S. K. Thompson. *Sampling*. John Wiley & Sons, New York, 2 edition, 2002.