Rethinking Dependence Clones

Tim A. D. Henderson[†], and Andy Podgurski[‡]
Dept. of Electrical Engineering and Computer Science
Case Western Reserve University
Cleveland, Ohio, USA 44106
Email: [†]tadh@case.edu, [‡]podgurski@case.edu

***Abstract*—*Semantic code clones* are regions of duplicated code that may appear dissimilar but compute similar functions. Since in general it is algorithmically undecidable whether two or more programs compute the same function, locating all semantic code clones is infeasible. One way to dodge the undecidability issue and find potential semantic clones, using only static information, is to search for recurring subgraphs of a *program dependence graph* (PDG). PDGs represent control and data dependence relationships between statements or operations in a program. PDG-based clone detection techniques, unlike syntactically-based techniques, do not distinguish between code fragments that differ only because of dependence-preserving statement re-orderings, which also preserve semantics. Consequently, they detect clones that are difficult to find by other means. Despite this very desirable property, work on PDG-based clone detection has largely stalled, apparently because of concerns about the scalability of the approach. We argue, however, that the time has come to reconsider PDG-based clone detection, as a part of a holistic strategy for clone management. We present evidence that its scalability problems are not as severe as previously thought. This suggests the possibility of developing integrated clone management systems that fuse information from multiple clone detection methods, including PDG-based ones.**

I. Introduction

Fragments of similar code are typically scattered throughout large code bases [1]. These repeated fragments or *code clones* often result from programmers copying and pasting code. Code clones (or just *clones*) may also result from limitations of a programming language, use of certain APIs or design patterns, following coding conventions, or a variety of other causes. Whatever their causes, existing clones need to be managed. When a programmer modifies a region of code that is cloned in another location in the program, they should make an active decision whether or not to modify the other location. Clearly, such decisions can only be made if the programmer is aware of the other location.

In general, there are 4 types of code clones [1]:

Type-1 Clones – Identical regions of code (excepting whitespace and comments).

Type-2 Clones – Syntactically equivalent regions (excepting names, literals, types, and comments).

Type-3 Clones – Syntactically similar regions (as in Type-2) but with minor differences such as statement additions or deletions.

Type-4 Clones – Regions of code with functionally equivalent behavior but possibly with different syntactic structures.

Much of the research on code clone detection and maintenance has been geared toward Type-1 and Type-2 clones [1]–[4], as they are easier to detect and validate than Type-3 and Type-4 clones. The two most popular detection methods involve searching for clones in *token streams* [5], [6] and *abstract syntax trees* (ASTs) [7], respectively.

An alternative approach to clone detection is to search for them in a *Program Dependence Graph* (PDG) [8], which represents the control and data dependences between statements or operations in a program. Recurring subgraphs in PDGs represent potential *dependence clones* (see Figure 1 on page 3, which is examined in Section II). Some of the previous work [9]–[11] on PDG-based clone detection used forward and backward path-slicing to find clones. This method can detect matching slices, but it cannot detect all recurring subgraphs. The latter can be identified using *frequent subgraph mining* (FSM) [12]. However, for low frequency thresholds, the number of PDG subgraphs discovered by FSM may be enormous. For example, we found that for a Java program with 70,000 lines of code (LOC), over 700 million PDG subgraphs with 5 or more instances were discovered by FSM.

Since it is infeasible for developers to examine so many subgraphs, we previously developed GRAPLE [13], an algorithm to select representative samples of maximal frequent subgraphs. In this paper, the core sampling process remains the same as in GRAPLE but we present a new algorithm for traversing the k-frequent subgraph lattice (see Section III). One tricky aspect of FSM is how to define exactly what "frequency" means in a large connected graph [14]. In order to handle pathological cases that occur in real programs, we introduce a new metric to measure subgraph frequency (or "support"), called the *Greedy Independent Subgraphs* (GIS) measure. Section IV details the first empirical examination of the scalability and speed of sampling dependence clones from large programs. The study showed that our new system can quickly sample from programs with 500 KLOC of code and successfully sample from programs with perhaps 2 MLOC. Finally, since at times the sampling algorithm may return several potential clones, which are quite similar to each other, we evaluate the performance of a density-based clustering algorithm on the samples collected.

II. A Motivating Example

Figure 1 shows an example *dependence clone* extracted from jGit, a Java implementation of the Git version control system. The clone was identified from the PDG of jGit produced by our PDG-generator `jpdg` [13]. PDG-based clone detection techniques, unlike techniques based on syntactic representations, do not distinguish between code fragments that differ only because of dependence-preserving statement re-orderings, which also preserve semantics [15]. Hence, using only static information, they detect semantically-equivalent clones that are not detected by syntactically-based techniques. (Of course, they cannot detect all semantically-equivalent code fragments.)

Three functions are shown in Figure 1 (two of them are partial) that parse PATCH files. The PATCH file format is a plain text format describing the differences (line by line) between two versions of a piece of software. There are several varieties of PATCH files including: “Traditional”, “Combined”, and “GIT”, all of which jGit can parse. Both the Combined and GIT formats are supersets of the Traditional format, which leads to some amount of duplication, especially involving headers. Figure 1(d) shows a graph pattern “explaining” the highlighted duplication in the three functions in terms of a subgraph of jGit’s PDG, which represents code detecting the start of a Traditional PATCH header.

However, the duplication in Figure 1 isn’t due to a simple copy-paste. Each function contains unique context-specific code interleaved with portions that detect the start of a Traditional header. Figure 1(a) contains the function `parseTraditionalHeaders`, which uses the pattern to drive the parser to extract the changed filenames. Function `parseFile` in Figure 1(b) drives the parsing process for all file types and uses the pattern to detect the start of each “hunk.” In contrast to the function in 1(a), `parseFile` reorders some of the statements and interleaves significant new functionality between statements. Finally, function `parseHunks` in Figure 1(c) exhibits a third statement ordering distinct from the first two. It uses the pattern to detect when it should stop parsing the current hunk.

Figure 1 illustrates that dependence-based clone detection can discover subtle programming patterns that are difficult to detect using other program representations due to differences in the ordering of statements within pattern instances and to the interleaving of statements from within the pattern with other statements.

III. Sampling Dependence Clones

Dependence clones are found by identifying recurring subgraphs of a program dependence graph (PDG), which is a kind of labeled directed graph. Formally, a *labeled directed graph* or *labeled digraph* $G = (V, E, l)$ consists of a set V of vertices, a set of edges $E \subseteq V \times V$, and a labeling function l that maps vertices and edges to labels.

A. Program Dependence Graphs

A *program dependence graph* (PDG) [8], [15]–[17] is a labeled digraph where the vertices represent statements or operations and the edges represent control or data dependences between them. A statement s is *control dependent* on another statement t if t is a conditional statement that decides whether s executes. For instance, the statements in the body of an **`if`** statement are each control dependent on the conditional expression of the **`if`** statement. A statement s is *data dependent* on a statement t if s reads a value v defined by t such that there is a control flow path between t and s that does not redefine v. Since PDGs represent dependence relationships instead of control flow relationships, they are not affected by semantics-preserving changes to the ordering of operations. The PDGs considered in this paper were constructed using our PDG generator `jpdg` [13].

Frequent subgraph mining is defined in terms of subgraph isomorphism.

Definition 1 (Subgraph Isomorphism)**.** *Given labeled digraphs* $H = (V_H, E_H, l_H)$ *and* $G = (V_G, E_G, l_G)$*, an injective mapping* $m : V_H \to V_G$ *is a* subgraph isomorphism *when:*

1) *m maps each vertex in H to a vertex in G with the same label:* $\forall\ v \in V_H\ [l_H(v) = l_G(m(v))]$
2) *m preserves edges:*
   $\forall\ u, v \in V_H\ [(u, v) \in E_H \Leftrightarrow (m(u), m(v)) \in E_G]$
3) *m preserves edge labels:*
   $\forall\ (u, v) \in E_H\ [l_H(u, v) = l_G(m(u), m(v))]$

If there is a subgraph isomorphism m between H and G then H is a *subgraph* of G, denoted $H \sqsubseteq G$. A subgraph isomorphism from H into G is called an *embedding* of H in G. The set of all of subgraph isomorphisms from H into G is denoted by $[\![H]\!]_G$. The problem of locating all of the unique embeddings of H in G is known as the *subgraph matching* problem. Frequent subgraph mining finds all subgraphs H that have at least k *supported* embeddings in a graph G, for a specified k. The number of supported embeddings is typically fewer than the number of unique embeddings [14].

B. Recurring Subgraphs

Dependence clones can be found by searching for “frequent” subgraphs of a PDG. An advantage of using FSM, rather than some other data mining techniques, is that lower frequency thresholds (e.g., 2-5) can be used without increasing the number of irrelevant patterns that are found [18]. This is because FSM helps ensure that the statements in a pattern are semantically related – not just co-occurring.

Definition 2 (k-Frequent Subgraph)**.** *Given a support measure (support counting function)* $\sigma : [\![\cdot]\!] \to \mathbb{N}^+$*, a minimum frequency level* $k \in \mathbb{N}$*, and a labeled digraph G, subgraph H of G is k-*frequent *if* $\sigma([\![H]\!]_G) \geq k$.

The process of finding frequent subgraphs of G can be viewed as a traversal of a *lattice* of subgraphs. The subgraph relation $\cdot \sqsubseteq \cdot$ induces a *Connected Subgraph Lattice* $\mathcal{L}_G$ representing all the possible ways of constructing G from the empty subgraph by adding one edge at a time. $\mathcal{L}_G$ is a digraph where each vertex u represents a unique connected (ignoring edge direction) subgraph of G. There is an edge from u to v in $\mathcal{L}_G$ if adding some edge ϵ to u creates a subgraph $u + \epsilon$ of G that is isomorphic to v, $v \cong u + \epsilon$. The *k-Frequent*

```
int parseTraditionalHeaders(int ptr, final int end) {
  while (ptr < end) {
    final int eol = nextLF(buf, ptr);
    if (isHunkHdr(buf, ptr, eol) >= 1) {
      // First hunk header; break out and parse them later.
      break;
    } else if (match(buf, ptr, OLD_NAME) >= 0) {
      parseOldName(ptr, eol);
    } else if (match(buf, ptr, NEW_NAME) >= 0) {
      parseNewName(ptr, eol);}
    } else {
      // Possibly an empty patch.
      break;
    }
    ptr = eol;
  }
  return ptr;
}
```

(a) Example from org.eclipse.jgit.patch.FileHeader (line 496)

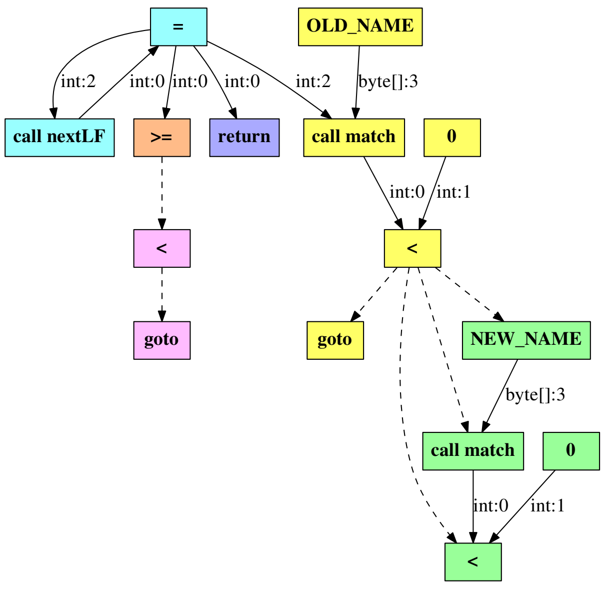
```
int parseFile(final byte[] buf, int c, final int end) {
  while (c < end) {
    if (isHunkHdr(buf, c, end) >= 1) { //Needs header
      error(buf, c,
          JGitText.get().hunkDisconnectedFromFile);
      c = nextLF(buf, c);
      continue;
    }
    // OMITTED: Valid git style patch?
    final int n = nextLF(buf, c);
    if (n >= end) { // Patches cannot be one line long
      return end;
    }
    if (n - c < 6) { // A valid header is >= 6 bytes
      c = n;
      continue;
    }
    if ( match(buf, c, OLD_NAME) >= 0
        && match(buf, n, NEW_NAME) >= 0) {
      // Probably a traditional patch. check "@@ -"
      final int f = nextLF(buf, n);
      if (f >= end)
        return end;
      if (isHunkHdr(buf, f, end) == 1)
        return parseTraditionalPatch(buf, c, end);
    }
    c = n;
  }
  return c;
}
```

(b) Example from org.eclipse.jgit.patch.Patch (line 172)

```
int parseHunks(final FileHeader fh, int c, final int end) {
  final byte[] buf = fh.buf;
  while (c < end) {
    // If we see a file header at this point, we have all of the
    // hunks for our current file. We should stop and report back
    // with this position so it can be parsed again later.
    if (match(buf, c, DIFF_GIT) >= 0)
      break;
    if (match(buf, c, DIFF_CC) >= 0)
      break;
    if (match(buf, c, DIFF_COMBINED) >= 0)
      break;
    if (match(buf, c, OLD_NAME) >= 0)
      break;
    if (match(buf, c, NEW_NAME) >= 0)
      break;

    if (isHunkHdr(buf, c, end) == fh.getParentCount()) {
      // OMITTED: Parses the hunk
      continue;
    }
    final int eol = nextLF(buf, c);
    if (fh.getHunks().isEmpty()
        && match(buf, c, GIT_BINARY) >= 0) {
      fh.patchType = FileHeader.PatchType.GIT_BINARY;
      return parseGitBinary(fh, eol, end);
    }
    if (fh.getHunks().isEmpty() && BIN_TRAILER.length < eol - c
        && match(buf, eol - BIN_TRAILER.length, BIN_TRAILER) >= 0
        && matchAny(buf, c, BIN_HEADERS)) {
      // The patch is a binary file diff, with no deltas.
      fh.patchType = FileHeader.PatchType.BINARY;
      return eol;
    }
    c = eol;  // Skip this line and move to the next.
  }
  // OMITTED: Check for empty patch which might be binary.
}
```

(c) Example from org.eclipse.jgit.patch.Patch (line 272)



(d) Graph pattern explaining the code duplication

Fig. 1: The highlighted regions above illustrate semantic code duplication in jGit (commit `efd91ef8a`), which was not found by the clone detector CCFinderX [5]. jGit is a Java implementation of the Git version control system. The three functions shown parse portions of PATCH files. The function in (a) parses the header of traditional PATCH files. The function in (b) (which has portions removed) parses all types of PATCH files used by Git (and "drives" the rest of the parsing functions). The function in (c) (which has portions removed) parses the hunks from the PATCH file looking for headers to signal the start of the next hunk. The graph in (d) is a frequent subgraph found in the PDG of jGit (produced by `jpdg` from JVM bytecode) and is a subgraph of the procedure dependence graphs of these functions. In the graph, dotted lines represent control dependences. Solid lines represent data dependences and are annotated with their types and usage context.

Connected Subgraph Lattice k-$\mathcal{L}_G$ is a Connected Subgraph Lattice containing only those subgraphs that are at least k frequent in G according to some support measure σ.

The most natural definition of the support measure is $\sigma(\llbracket H \rrbracket_G) = |\llbracket H \rrbracket_G|$, i.e., the number of unique embeddings in H's isomorphism class. Unfortunately, this definition does not satisfy an important property of suitable support measures, called the *Downward Closure Property* [14]. A measure that is commonly used instead is *Minimum Image Support* (MNI) [14]. However, some pathological cases involving patterns with *automorphisms* (nontrivial isomorphisms from a subgraph to itself) and with overlapping embeddings can cause exponential-time computations when MNI is used. We have found that these actually occur in real programs. To circumvent this problem we use *an unsound support measure* called *Greedy Independent Subgraphs* (GIS).

```
 1 def sample(N, graph, min_support):
 2     for _ in xrange(N):
 3         yield unweighted_random_walk(G, min_support)
 4
 5 def unweighted_random_walk(G, min_support):
 6     prev = cur = G.root_lattice_node(min_support)
 7     while cur is not None:
 8         n = randchoose(cur.children()); prev = cur; cur = n
 9     return prev
10
11 class LatticeNode(object):
12     def __init__(self, G, sg, embs, exts):
13         self.graph = G
14         self.subgraph = sg
15         self.supported_embeddings = embs
16         self.extensions = exts
17     def children(self):
18         G = self.graph
19         for sg_ext in self.extend():
20             support, exts, embs = exts_and_embs(G, sg_ext)
21             if support >= min_support:
22                 yield LatticeNode(G, sg, embs, exts)
23     def extend(self):
24         exts = set()
25         for ext in self.extensions:
26             exts.add(self.subgraph.extend(ext))
27         return exts
28
29 def exts_and_embs(G, sg):
30     embs = list(); exts = set(); seen = set()
31     for emb in find_embeddings(G, sg, gis_pruner(seen)):
32         for emb_idx in embs.idxs:
33             for edge in G.children_of(emb_idx):
34                 add_ext(G, exts, emb, edge, emb_idx, -1)
35             for edge in G.parents_of(emb_idx):
36                 add_ext(G, exts, emb, edge, -1, emb_idx)
37         embs.append(emb)
38     return gis_support(embs), exts, embs
39
40 def find_embedddings(G, sg, prune_fn=None):
41     edges = spanning_tree(G, sg)
42     for edge in sg.edges:
43         if edge not in edges: edges.append(edge)
44     stack = list()
45     vembs = vertex_embeddings(G, sg, edges[0].src)
46     for emb_idx in vembs:
47         stack.append((ListNode(start_idx, emb_idx), 0))
48     while len(stack) > 0:
49         cur, eid = stack.pop()
50         if prune_fn is not None and prune_fn(cur):
51             continue
52         if eid >= len(spanning_edges):
53             yield embedding_from_ids(cur)
54         else:
55             for n in extend_embedding(G,sg,cur,edges[eid])
56                 stack.append((n, eid+1))
57
58 def gis_pruner(seen):
59     def gis_prune(cur):
60         for n in cur:
61             if n.emb_idx in seen:
62                 for m in cur: seen.add(m.emb_idx)
63                 return True
64         return False
65     return gis_prune
```

Listing 1: Sample N k-frequent subgraphs.

An embedding n is *directly connected* to another embedding m if any vertex in n is also used in m. The embedding n is *connected* (possibly indirectly) to another embedding x if there is some sequence of embeddings $y_1 ... y_n$ such that: the embedding n is directly connected to the embedding y_1, y_1 is directly connected to y_2, and so on, until y_{n-1} is directly connected to y_n and y_n is directly connected to x. An embedding n is said to be *independent* of an embedding x if they are not connected by any sequence of embeddings. Recall that automorphic patterns with overlapping embeddings cause problems for MNI since there may be millions of unique embeddings. However, one can short-circuit this computation by computing the number of independent groups of embeddings in a greedy fashion. This yields the metric GIS, which is implemented by the function `gis_pruner` in Listing 1.

C. Sampling Frequent Subgraphs

Sampling N frequent subgraphs can be accomplished by an unweighted forward random walk on the connected frequent subgraph lattice [13]. The process is outlined in Listing 1 and is based on our previous work on GRAPLE [13]. Note the listing is in Python for brevity but the actual implementation is in the Go programming language. The core sampling process (lines 1-9) remains the same as in GRAPLE but there is a new algorithm for traversing the k-frequent subgraph lattice. The new algorithm incorporates the GIS support metric and pruning strategy into its subgraph matching algorithm `find_embeddings`. The algorithm also merges the support and candidate extension computations in `exts_and_embs`.

In Listing 1, the function `sample` does N walks over the frequent subgraph lattice by calling `unweighted_random_walk` repeatedly. Each walk starts at a `LatticeNode` representing the empty subgraph. In every step of the walk a call is made (line 8) to `LatticeNode.children()`, which computes frequent super-graphs of the graph represented by the current `LatticeNode`. One super-graph is selected at random to use for the next step. The walk terminates when there are no frequent super-graphs. The `children` method computes candidate frequent super-graphs using the `extend` method. Each candidate is computed from an "extension", which is obtained by adding a single edge to the graph represented by the `LatticeNode`. After the candidates are computed, each one must be checked to see if it is frequent. This is done by the `exts_and_embs` function, which computes the frequency (support), the candidate one-edge extensions, and the supported embeddings for the candidate frequent subgraph.

To compute the support for a subgraph, all of its embeddings need to be found. This is implemented by the

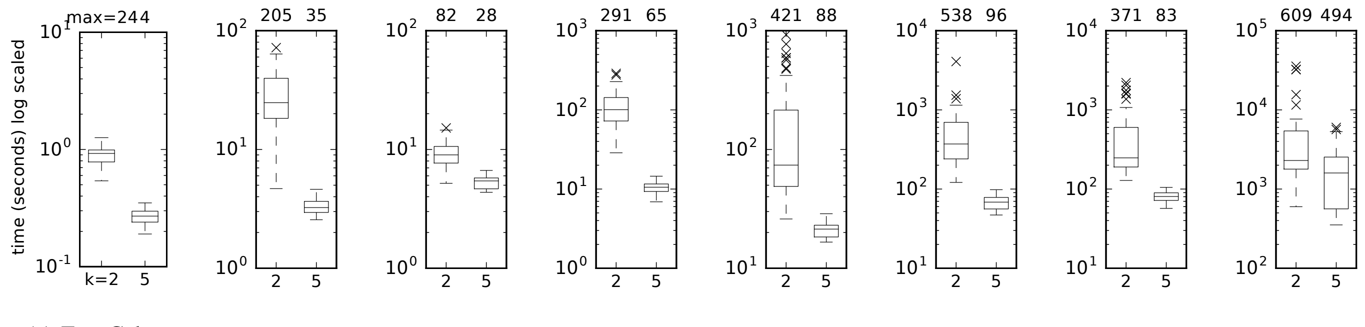


Fig. 2: Execution time to sample 100 dependence clones from the subject programs. Box and whisker diagrams of execution time in seconds of 50 runs of the Unweighted Random Walk sampling algorithm; each run collected a sample containing 100 frequent subgraphs. The number on the top of the axis is the size (in # of edges) of the largest dependence clone found. The number on the bottom axis is the minimum support (frequency) used.

`find_embeddings` method, which solves the subgraph matching problem. This method implements a back-tracking tree search procedure. The nodes in the search are partial subgraph isomorphism mappings. The search tree for a subgraph H has height $|E_H|$, where $|E_H|$ is the number edges in H. The search starts at a node mapping a single vertex in H to some vertex with the same label in the graph G. It proceeds edge by edge, building up a mapping until a full mapping is obtained or back-tracking is performed by discarding the current mapping and considering another partial mapping. If the `gis_pruner` is supplied as the pruning function, `prune_fn`, to `find_embeddings` then on line 50 there is a chance to discard the current mapping. GIS discards the mapping if any of the mapped vertices in it have already appeared in a completed mapping.

While not shown in the listing, the implementation contains several other optimizations to the embedding search. For instance, it uses a lightweight index to ensure that candidate vertices used to extend the current embedding have at least the degree of the matching subgraph vertex.

IV. Evaluation

A new dependence-clone sampling system was implemented using the ideas described in Section III-C, and it was evaluated in an empirical study. The study examined samples of 100 potential code clones from the eight subject programs described in Table I. The sample size was chosen to represent the maximum number of clones a single developer would likely be able to review in a day. The study considered code cloned in at least five locations as well as code cloned in at least two locations, because code that is duplicated in more locations is potentially more relevant to the programmer and can be found more quickly with pattern mining methods. Since every sample collection run collects a different selection of 100 potential clones, there is variance in the amount of time it takes to collect the sample. The variance was measured by collecting 50 independent samples with 100 frequent subgraphs in each sample (see Figure 2).

To look at the scalability of the new implementation on larger programs, OrientDB was examined with the addition of *all* of its library dependencies – including generated code. Thus, the OrientDB PDG includes code from both the application and all of the non-application libraries. Since the libraries were delivered as JVM bytecode no line count could be determined. OrientDB generated around four times the number of vertices as the next largest program – so the total line count of OrientDB and its dependencies may be as large as 2 MLOC.

The PDGs used in the study were generated by our tool `jpdg` [13]. The evaluation was conducted on a dual-socket server with 2010 Intel Xeon X5650 CPUs and 96 GB of memory. Despite the large amount of memory on the server used to collect the timing data, all computations ran well on a laptop with 16 GB of memory. All used under 4 GB of RAM except for those involving the OrientDB dataset, which required around 10 GB of RAM. The memory usage was dominated by storage for the graph index and varied only slightly during the actual sampling procedure.

It took under 2 minutes to collect a sample of 100 five-frequent subgraphs for all of the programs except OrientDB (Figure 2). For the smaller programs, collecting a sample required less than 10 seconds – making our new implementation suitable for desktop usage. For the smallest program, ExprCalc, we were able to find all of the frequent subgraphs ($\sim$2.5 million at minimum support 2). The rest of the programs had too many frequent subgraphs ($>$100 million) for us to mine them all at minimum support 2 or 5. At support level 5 it yields only frequent subgraphs containing at most 4 edges.

TABLE I: The datasets used in the study. KLOC – Kilo Lines of Code.

Dataset	KLOC	Nodes	Edges	Description
ExprCalc	0.8	1,110	2,162	Arithmetic calculator
Zookeeper	32.4	17,028	32,691	Distributed KV store
DDH	19.3	36,384	65,874	Anonymized proprietary application
BCEL	28.6	52,731	108,542	JVM bytecode lib.
jGit	72.1	136,716	300,550	GIT in Java
Tomcat	220.7	377,657	806,824	Web server
hBase	561.4	442,063	981,577	Database
OrientDB	N/A	2,022,640	3,476,158	Database & all dependencies.

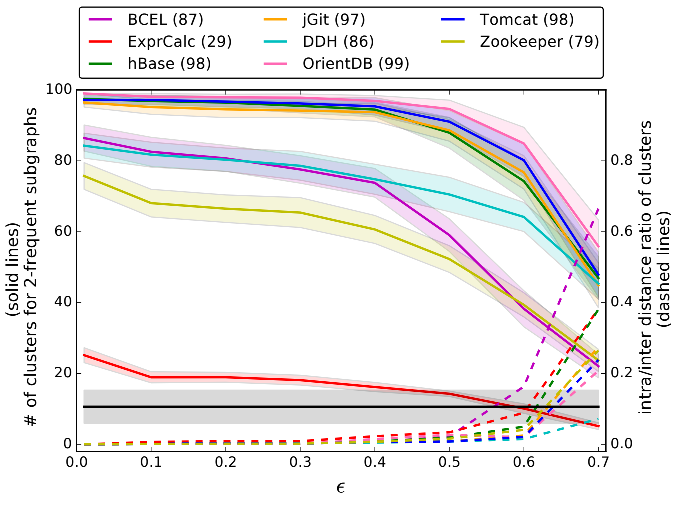


Fig. 3: The mean number of clusters for samples of 2-frequent subgraphs (solid lines; variances are shown as shaded regions), for various choices of ϵ. Items α and β cluster together if $\delta(\alpha, \beta) < \epsilon$ where δ is a distance metric between the items. The number next to each dataset name is the average number of unique items sampled when sampling (with replacement) 100 items total. The dashed lines show the intra/inter cluster distance ratio – the closer it is to 0 the better the clustering. The solid black line shows this ratio for random clustering (the shaded area indicates the variance). If a dashed line is below the black line it is better than random.

Larger 5-frequent subgraphs were found for rest of the subject programs, ranging in size from 35 edges for Zookeeper to 96 edges for Tomcat to 494 edges for OrientDB (see the top axis in Figure 2).

Programmers are potentially interested in regions of code duplicated in just one other location. As shown in Figure 2, the execution time required to collect a sample of 100 2-frequent subgraphs was greater than the time for 5-frequent subgraphs. The longest amount of time spent collecting a sample for any program except OrientDB was 67 minutes, while the highest mean time was $8\frac{3}{4}$ minutes. OrientDB took much longer — as long as 10 hours. Recall that OrientDB is four times larger than the next largest program, hBase, which has 500 KLOC. Clearly, there is work left to be done on the scalability of PDG-based clone detection. In the future, when detecting clones for code review only the ones related to changed code need to be found. Performance may also be improved by using a better indexed subgraph matching algorithm [19] and incrementally updating the PDGs [20], [21].

Another issue with code clones in general, but especially with clones found from graphical representations such as PDGs, is reporting multiple clones that are very similar to each other. These "clone families" arise naturally since many frequent subgraphs share common subgraphs with other frequent subgraphs. When reviewing code clones, programmers do not want to see very similar clones over and over again. Future PDG-based clone detection systems should address this problem. Towards that end, Figure 3 evaluates a simple density based clustering algorithm, DBSCAN [22] (with no minimum number of items allowed in a cluster). In density based clustering, items α and β cluster together if the distance between them, according a metric δ, is less than ϵ: $\delta(\alpha, \beta) < \epsilon$. In Figure 3 the δ metric is the Jaccard set similarity coefficient applied to the sets of vertex labels of two subgraphs. Thus, subgraphs that contain the same combination of operators, method calls, and constants are placed together.

DBSCAN identified sizable "tight" clusters (as measured by the intra/inter cluster distance ratio) of clones for 4 of the subject programs (ExprCal, Zookeepr, BCEL, and DDH in Figure 3). These clusters indicate the presence of clone families, which we confirmed with visual inspection. Identifying these clusters reduces the programmer effort needed to review the set of potential clones, since only a representative from the cluster needs to be reviewed by the programmer. For the other 4 programs (jGit, Tomcat, hBase, and OrientDB) most of the sampled frequent subgraphs were distinct from each other, and cluster quality was poor when they were grouped together (as can be seen in the figure, for the higher settings for ϵ). Future work could integrate an online clustering technique into the sampling procedure to ensure adequate diversity in the output. In addition, there are many other similarity measures for graphs [23] and some of them such as graph kernels can take into account the structure of graphs. Such measures may perform better for this application and their utility for PDG-based clone-detection should be evaluated. Semi-supervised kernel graph clustering [24], which can exploit user feedback, also merits consideration.

V. Related Work

There are several recent surveys on finding and managing code clones [1]–[4]. Recent work by Sajnani *et al.* on SourcererCC [6] showed that clone detection can scale to 100 MLOC when programs are represented as bags-of-tokens. It took SourcererCC only $1\frac{1}{2}$ days to find all of the clones from an artificially constructed code base consisting of 100 MLOC. In the SourcererCC study, the latest version of CCFinder [5] (CCFinderX) was competitive with SourcererCC on most benchmarks in terms of time, precision, and recall.

Krinke's Duplix algorithm [9] and Komondoor and Horwitz's algorithm [10] were the first attempts at detecting code clones from PDGs. Komondoor's algorithm found pairs of clones by slicing backwards and then forwards from matched starting vertices. However, the forward slicing operation is only applied when matching control vertices are discovered. Komondoor also applies a variety of heuristics to filter out certain types of clones. After pairs are identified, ones that include the same locations are grouped together and subgraphs are discarded in favor of their super-graphs. Higo and Kusomoto [11] created Scorpio, which extends Komondoor's algorithm to detect contiguous clones by adding links into the PDG. Krinke's algorithm Duplix is similar to Komondoor's but restricts itself to forward slicing up to a limit of k edges. Both of these algorithms, unlike the one presented in this paper, tend to find long paths through the PDG instead of general subgraphs (containing any edge structure) such as the one in Figure 1. Given the reported scalability problems with Komondoor's algorithm [2], Gabel *et al.* [25] proposed mapping PDG nodes onto abstract syntax trees, which allowed them to use a high performance clone detector for ASTs [7]. ModelCD [26] used the vSiGraM algorithm [12] to find clones in Matlab/Simulink models. Nguyen *et al.* [27] introduced *groums* for finding graph-based object-usage patterns using a novel frequent subgraph miner. Jia *et al.* presented KClone [28], which extends contiguous clones (found from tokens) using local data dependence information. Finally, Higo *et al.* [20] demonstrated an approximate incremental approach for detecting clones from evolving PDGs.

Many algorithms have been developed for frequent subgraph mining [29]. The algorithm presented here is based on GRAPLE [13], which is strongly related to the ORIGAMI algorithm [30]. There are two other sampling procedures for frequent subgraphs: Musk [31] and a Metropolis-Hastings approach [32]. However, our experiments indicate that neither can find patterns larger than around 15 edges.

VI. Conclusion

We have presented a new algorithm for sampling potential code clones from program dependence graphs, using unweighted random walks over the frequent connected subgraph lattice. The algorithm uses the greedy independent subgraphs measure to prune the subgraph-matching search space, which reduces the computation costs for difficult-to-mine program graphs. Empirical results were presented that demonstrated that the algorithm is capable of mining large programs. For programs with at least 500,000 LOC it can sample clones fast enough to be used either on the desktop or in a continuous integration system for use during code review. The algorithm presented does not use any heuristics or limit the size of frequent subgraph found. Results were also presented for the effectiveness of using density based clustering on the returned clones. For half of the programs significant clusters were found. The time has come to reconsider PDG-based clone detection as part of a holistic strategy of clone management and to develop clone management systems that integrate multiple detection strategies.

References

[1] C. K. Roy, M. F. Zibran, and R. Koschke, "The vision of software clone management: Past, present, and future," in *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering*, 2014, pp. 18–33.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, sep 2007.

[3] 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*, vol. 74, no. 7, may 2009.

[4] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[5] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, jul 2002.

[6] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling Code Clone Detection to Big-code," in *Int. Conference on Software Engineering*, 2016, pp. 1157–1168.

[7] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Int. Conference on Software Engineering*, 2007, pp. 96–105.

[8] 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*, vol. 9, no. 3, pp. 319–349, July 1987.

[9] J. Krinke, "Identifying similar code with program dependence graphs," in *IEEE Working Conference on Reverse Engineering*, 2001.

[10] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," *Int. Symposium on Static Analysis*, vol. 2126, 2001.

[11] Y. Higo and S. Kusumoto, "Code Clone Detection on Specialized PDGs with Heuristics," in *European Conference on Software Maintenance and Reengineering*. IEEE, mar 2011, pp. 75–84.

[12] M. Kuramochi and G. Karypis, "Finding Frequent Patterns in a Large Sparse Graph*," *Data Mining and Knowledge Discovery*, vol. 11, no. 3, pp. 243–271, nov 2005.

[13] T. A. D. Henderson and A. Podgurski, "Sampling Code Clones from Program Dependence Graphs with GRAPLE," in *Int. Workshop on Software Analytics*. ACM, 2016.

[14] B. Bringmann and S. Nijssen, "What is frequent in a single graph?" in *Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, 2008, pp. 858–863.

[15] 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*, vol. 16, no. 9, pp. 965–979, sep 1990.

[16] S. Horwitz, J. Prins, and T. Reps, "On the Adequacy of Program Dependence Graphs for Representing Programs," in *Symposium on Principles of Programming Languages*. ACM, 1988, pp. 146–157.

[17] A. Podgurski and L. Clarke, "The Implications of Program Dependencies for Software Testing, Debugging, and Maintenance," in *Symposium on Software Testing, Analysis, and Verification*. ACM, 1989, pp. 168–178.

[18] R.-Y. Chang, A. Podgurski, and J. Yang, "Discovering Neglected Conditions in Software by Mining Dependence Graphs," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 579–596, sep 2008.

[19] K. Zhu, Y. Zhang, X. Lin, G. Zhu, and W. Wang, "NOVA: A Novel and Efficient Framework for Finding Subgraph Isomorphism Mappings in Large Graphs," in *Int. Conference on Database Systems for Advanced Applications*, 2010, pp. 140–154.

[20] Y. Higo, U. Yasushi, M. Nishino, and S. Kusumoto, "Incremental Code Clone Detection: A PDG-based Approach," in *Working Conference on Reverse Engineering*, oct 2011, pp. 3–12.

[21] A. Ray, L. Holder, and S. Choudhury, "Frequent Subgraph Discovery in Large Attributed Streaming Graphs," in *Int. Workshop on Big Data, Streams and Heterogeneous Source Mining*, vol. 36, 2014, pp. 166–181.

[22] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A Density-based Algorithm for Discovering Clusters a Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," in *Int. Conference on Knowledge Discovery and Data Mining*, 1996.

[23] K. Riesen, X. Jiang, and H. Bunke, *Exact and Inexact Graph Matching: Methodology and Applications*. Springer, 2010, pp. 217–247.

[24] B. Kulis, S. Basu, I. Dhillon, and R. Mooney, "Semi-supervised graph clustering: a kernel approach," *Machine Learning*, vol. 74, no. 1, 2009.

[25] M. Gabel, L. Jiang, and Z. Su, "Scalable Detection of Semantic Clones," in *Int. Conference on Software Engineering*, 2008, pp. 321–330.

[26] 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 *Int. Conference on Software Engineering*, 2009.

[27] 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 *European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. ACM Press, 2009, p. 383.

[28] Y. Jia, D. Binkley, M. Harman, J. Krinke, and M. Matsushita, "KClone: A Proposed Approach to Fast Precise Code Clone Detection," in *Int. Workshop on Software Clones*, 2009.

[29] H. Cheng, X. Yan, and J. Han, "Mining Graph Patterns," in *Frequent Pattern Mining*. Springer, 2014, pp. 307–338.

[30] 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," *Statistical Analysis and Data Minining*, vol. 1, no. 2, pp. 67–84, jun 2008.

[31] M. Al Hasan and M. Zaki, *Musk: Uniform Sampling of k-Maximal Patterns*. SIAM International Conference on Data Mining, 2009.

[32] M. Al Hasan and M. J. Zaki, "Output Space Sampling for Graph Patterns," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 730–741, aug 2009.