Code Relatives: Detecting Similarly Behaving Software

Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey,
Simha Sethumadhavan, Gail Kaiser and Tony Jebara
Columbia University
500 West 120th St, MC 0401
New York, NY USA
{mikefhsu, jbell}@cs.columbia.edu, kh2333@columbia.edu
{simha, kaiser, jebara}@cs.columbia.edu

ABSTRACT

Detecting "similar code" is useful for many software engineering tasks. Current tools can help detect code with statically similar syntactic and–or semantic features (code clones) and with dynamically similar functional input/output (simions). Unfortunately, some code fragments that behave similarly at the finer granularity of their execution traces may be ignored. In this paper, we propose the term "*code relatives*" to refer to code with similar execution behavior. We define code relatives and then present DyCLINK, our approach to detecting code relatives within and across codebases. DyCLINK records instruction-level traces from sample executions, organizes the traces into instruction-level dynamic dependence graphs, and employs our specialized subgraph matching algorithm to efficiently compare the executions of candidate code relatives. In our experiments, DyCLINK analyzed 422+ million prospective subgraph matches in only 43 minutes. We compared DyCLINK to one static code clone detector from the community and to our implementation of a dynamic simion detector. The results show that DyCLINK effectively detects code relatives with a reasonable analysis time.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: [Distribution, Maintenance, and Enhancement]; I.1.5 [**Pattern Recognition**]: [Clustering]

General Terms

Algorithms, Experimentation, Design

Keywords

Code relative, runtime behavior, link analysis, subgraph match, code clone

1. INTRODUCTION

Code clones [45], which represent textually, structurally, or syntactically similar code fragments, have been widely adopted to detect similar pieces of software. However, code clone detection systems typically focus on identifying static patterns in code, so relevant code fragments that behave similarly at runtime, though with different structures, are ignored. Detecting code fragments that accomplish the same tasks or share similar behavior is pivotal for understanding and improving the performance of software systems. For example, with such functionality, it may be possible to automatically replace an old algorithm in a legacy system with a new one or to detect commonly repeated tasks to create APIs (semi)automatically. It may allow quick search and understanding of large codebases, and de-obfuscation of code.

Towards detecting similarly behaving code, previous work observed code fragments that yield the same output for the same input [14, 22] or that share similar identifiers and structural concepts [5, 39, 41]. A significant challenge in detecting *similar* but not equivalent code fragments by comparing input and output pairs, a technique also known as finding *simions* [23], is judging how similar two outputs need to be for the two code fragments to be considered simions. Particularly, with object-oriented languages, this problem may be more complex: the same data can be designed with different project-specific data types between projects [11].

Our key insight is to shift this similarity comparison to study how each code fragment computes its result, rather than simply comparing those output results or comparing what that code looks like. That is, we can gauge how similar two code fragments are without even looking at the respective inputs and outputs. To represent runtime similarity (i.e., how the code fragment computes its result), we introduce the term *Code Relatives*. Code relatives are continuous or discontinuous code fragments that exhibit similar behavior, but may be expressed in structurally or even conceptually different ways. The key relationship between these code fragments is that they are performing a similar computation regardless of how similar or dissimilar their outputs may be.

Our key contribution is an efficient system for detecting these code relatives that is agnostic to the output format or identifiers used in the code. Our system, DyCLINK, traces each program's execution creating a dynamic dependency graph that captures behavior at the instruction level. These dependency graphs encode rich and dense behavioral information, more than would be found simply observing the outputs of parts of a program or obtained from a static analysis of that program. Code relatives are common (sub)graphs via fuzzy matching that occur repeatedly between and within these profiled execution graphs. DyCLINK detects code relatives at any granularity: a code relative may be a part of

ACMXXX United States

a single method or instead be composed of several methods that are executed in a sequence.

The resulting graphs are large: containing a single node for every instruction, plus edges representing dependencies. Hence, typical approaches for detecting common subgraphs (subgraph isomorphism) are time prohibitive — requiring expensive comparisons between each potential set of code relatives. In our evaluation, we examined 118 projects for code relatives, containing a total of 1,244 different dynamic dependence graphs, which represented a total of over 422 million subgraphs that would be compared for similarity. To efficiently identify code relatives in these graphs, we have developed a new algorithm, *LinkSub*, that leverages the PageRank [33] algorithm to compare subgraphs and to reduce the number of pairwise comparisons needed between subgraphs to efficiently detect code relatives (in our evaluation, filtering away over 99% of the comparisons).

We built DyCLINK, targeting Java programs, but our methodology applies to most high level languages. We evaluated DyCLINK on a corpus of Java programs that were known to contain clusters of similar programs. DyCLINK effectively reconstructed the clusters of programs with very high precision (94%). We compared DyCLINK with one state-of-the-art static clone detector plus one dynamic simion detector (input-output similarity checker), finding it to be more effective at clustering similarly behaving software.

2. BACKGROUND

Before discussing the details of DyCLINK, we first define the key terms used in this paper and discuss some use cases of code relatives.

2.1 Basic Definitions

When discussing the notion of *similar* code, it is important to have a clear definition of what *similar* means. For our purpose, two code fragments are similar if they produce similar instruction-level runtime behavior, which is witnessed by execution traces (dynamic dependency graphs) that are roughly equivalent.

- **Code fragment:** Either a continuous or discontinuous set of code lines.
- **Code clone:** "A code fragment CF_2 is a clone of another code fragment CF_1 if they are similar by some given definition of similarity" [45]. We express this as follows. CF_1 and CF_2 are code clones if:
$$f_{sim}(CF_1, CF_2) \geq \theta_{stat} \quad (1)$$
where f_{sim} is a similarity function and θ_{stat} is a predefined threshold for static code fragments.
- **Code relative:** An execution of a code fragment generates a trace of that execution, $Exec(CF)$. We denote the set of a code fragment's traces as $\{Exec(CF)\}$. Given a similarity function f_{sim} and a threshold θ_{dyn} for code execution, two code fragments, CF_1 and CF_2, are code relatives if:
$$f_{sim}(\{Exec(CS_1)\}, \{Exec(CS_2)\}) \geq \theta_{dyn} \quad (2)$$

In this work, we capture execution traces as dynamic program dependency graphs, and we model the similarity between two code fragments as a subgraph isomorphism problem described further in §4.

Code relatives are distinct from "simions" in that simions are code fragments that show similar outputs given an input, while code relatives show similar behavior, regardless of their outputs [23]. Moreover, code relatives may consider discontinuous code fragments and include cases in which their intermediate results (but not outputs) are similar. Code relatives are not tied to a particular programming abstraction: a code relative may be a portion of a method, or may represent computation that is performed across several methods. All code relatives are behavioral code clones given that the definition of "similarity" is limitless for clones in general. We use the term code relative rather than a variant of code clone or simion to make their distinctions clear and avoid ambiguity.

2.2 Motivation

Detecting similar programs is beneficial in supporting several software engineering tasks, such as helping developers understand and maintain systems [41], identifying code plagiarism [37], and enabling API replacement [30]. Although code clone detection systems can efficiently detect structurally similar code fragments, they may still miss some cases for optimizing software and–or hardware that require information about runtime behavior [12]. Programs that have syntactically similar code fragments usually have similar behavior; however programs can still have similar behavior even if their code is not alike [22, 23].

Moreover, programs may have similar behavior even if their outputs for the same or equivalent inputs are not identical. In fact, in many cases, it may be difficult to judge that two outputs are equivalent, or even similar, due to differences in data structures. On detecting functionally equivalent code fragments in Java, Deissenboeck et al. reported that 60-70% of the studied program chunks across five open-source projects referred to project-specific data types [11]. Hence, it is impossible to directly compare inputs and outputs for equivalence across many projects. To get around these dissimilar data types, developers would have to specify adapters to convert from project-specific datatypes to abstract representations that could be compared. By ignoring the outputs of code fragments and observing only their behavior, we can avoid this output equivalence problem.

Consider, for example, the two code examples shown in Figure 1, taken from the libraries *Apache Commons Math*[1] and *Jama*[2], both of which perform the same matrix decomposition task. In the case of Figure 1a, all computation is done in a single method and the result is stored as instance fields of the object being constructed. In the case of Figure 1b, computation is split between several methods: `solve`, which invokes several methods to compute the result, which is returned as a `Matrix` object (a type defined by the library). A simion detector (comparing inputs and outputs) would have difficulty to compare the inputs and outputs when the data structures do not match exactly, and there may not be clearly defined outputs. A typical clone detector using the abstract syntax tree of this code would also find it hard to detect the multi-method clone. It would need to compute callgraph information to consider valid multi-method clones, which again, have many subtle differences in code structure. In fact, clone detection tools may not consider these two code listings to be clones, while we argue that they are code relatives and are indeed detected by DyCLINK.

Software clustering and *Code search* are two domains that rely on similarity detection between programs and could ben-

[1] https://commons.apache.org/proper/commons-math/
[2] http://math.nist.gov/javanumerics/jama/

```
1 public SingularValueDecomposition(final RealMatrix
      matrix) {
2   ....
3   // Generate U.
4   ....
5   for (int k = nct - 1; k >= 0; k--) {
6     if (singularValues[k] != 0) {
7       for (int j = k + 1; j < n; j++) {
8         double t = 0;
9         for (int i = k; i < m; i++) {
10          t += U[i][k] * U[i][j];
11        }
12        t = -t / U[k][k];
13        for (int i = k; i < m; i++) {
14          U[i][j] += t * U[i][k];
15        }
16      }
17    ...
18  }
19  // Generate V.
20  for (int k = n - 1; k >= 0; k--) {
21    if (k < nrt &&
22      e[k] != 0) {
23      for (int j = k + 1; j < n; j++) {
24        double t = 0;
25        for (int i = k + 1; i < n; i++) {
26          t += V[i][k] * V[i][j];
27        }
28        t = -t / V[k + 1][k];
29        for (int i = k + 1; i < n; i++) {
30          V[i][j] += t * V[i][k];
31        }
32      }
33    }
34    ...
35  }
36 }
```

(a) Commons maths's `SingularValueDecomposition.<init>`

```
1 public Matrix solve (Matrix B) {
2   return (m == n ? (new LUDecomposition(this)).solve
      (B) : (new QRDecomposition(this)).solve(B));
3 }
4 public QRDecomposition (Matrix A) {
5   ...
6   for (int k = 0; k < n; k++) {
7     ...
8     if (nrm != 0.0) {
9       ...
10      for (int j = k+1; j < n; j++) {
11        double s = 0.0;
12        for (int i = k; i < m; i++) {
13          s += QR[i][k]*QR[i][j];
14        }
15        s = -s/QR[k][k];
16        for (int i = k; i < m; i++) {
17          QR[i][j] += s*QR[i][k];
18        }
19      }
20    }
21    Rdiag[k] = -nrm;
22  }
23 }
24 public Matrix solve (Matrix B) {
25  ...
26  for (int k = 0; k < n; k++) {
27    for (int j = 0; j < nx; j++) {
28      double s = 0.0;
29      for (int i = k; i < m; i++) {
30        s += QR[i][k]*X[i][j];
31      }
32      s = -s/QR[k][k];
33      for (int i = k; i < m; i++) {
34        X[i][j] += s*QR[i][k];
35      }
36    }
37  }
38  ...
39 }
```

(b) Jama's `Matrix.solve`

Figure 1: A partial comparison of matrix decomposition code from two different libraries. Despite differences in each method, both are code relatives.

efit from code relatives. Software clustering locates and aggregates programs having similar code or behavior. The clusters support developers understanding code semantics [32, 38], prototyping rapidly [7], and locating bugs [13]. Code search helps developers determine if their codebases contain programs befitting their requirements [41]. A code search system takes program specifications as the input and returns a list of programs ranked by their relevance to the specification.

Software clustering and code search can be based on static or dynamic analysis. Static analysis relies on features, such as usage of APIs, to *approximate* the behavior of a program. Dynamic analysis identifies traits of executions, such as input/output values and sequences of method calls, to represent the *real* behavior. A system that captures more details and represents program behavior more effectively (e.g., instead of simions) can more precisely detect similar programs in support of both software clustering and code search.

Based on the use cases above, instead of identifying static code clones or dynamic simions, we designed DyCLINK, a system to detect dynamic *Code Relatives*, which represent similar runtime behavior between programs. We have evaluated DyCLINK, finding it to have high precision (94%) when applied to software clustering, results discussed in §5.

3. RELATED WORK

Code similarity detection tools can be distinguished by the similarity metrics that they use, exact or fuzzy matching, and the intermediate representation used for comparison. Common intermediate representations tend to be token-based [4,25,35], AST-based [6,21], or graph-based [6,21,27,28,30,37]. SourcererCC [46] compares code fragments using a static bag-of-tokens approach that is incredibly fast, but does not target specifically similarly behaving code with different structures.

Among static approaches, DyCLINK is most similar to those that used program dependence graphs (PDGs) to detect clones. Komondoor and Horwitz [27] generate PDGs for C programs, and then apply program slicing techniques to detect isomorphic subgraphs. The approach designed by Krinke [30] starts to detect isomorphic subgraphs with maximum size k after generating PDGs. The granularity of Krinke's PDGs is finer than the traditional one: each vertex roughly maps to a node in an AST. The approach proposed by Gabel et al. [17] is a combination of AST and graph. It generates the PDG of a method, maps that PDG back to an AST, and then uses Deckard [21] to detect clones. GPLAG [37] determines when to invoke the subgraph matching algorithm between two PDGs using two statistical filters. Deckard [21] detects similar but perhaps structurally-different code fragments by comparing ASTs.

Compared with these graph-based approaches that identify *static* code clones, DyCLINK detects the similar *dynamic* behavior of programs (code relatives). This allows DyCLINK

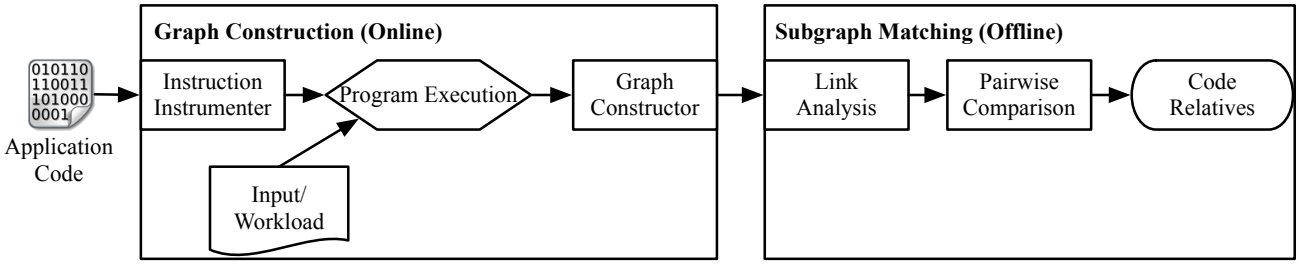


Figure 2: The high-level architecture of DyCLINK including instruction instrumentation, graph construction, link analysis and final pairwise subgraph comparison.

to detect code relatives that are dependent upon dynamic behavior, for example splitting across multiple methods.

Other previous works in dynamic code similarity detection focus on observing when code fragments produce the same outputs for the same inputs. Jiang and Su [22] drive programs with randomly generated input and then observe their output values, identifying clones as two methods that provide the same output for the same input. Li et al. detect functional similarity between code fragments using dynamic symbolic execution to generate inputs [34]. Similarly, the MeCC system [26] detects code similarity by observing two methods that result in the same abstract memory state. CCCD, or concolic clone detection [31], takes a similar approach, comparing the concolic outputs of methods to detect function-level input/output similarity. Elva and Leavens propose detecting functionally equivalent code by detecting methods with exactly the same outputs, inputs and side effects [15]. Juergens et al. propose *simions*, two methods that are found to yield similar (not necessarily identical) outputs for the same input, but provide no automated technique for detecting such simions [11, 23]. We implement a simion detector for Java, HitoshiIO [48], which attempts to overcome the problem of different data structures through a fuzzy equivalence matching. HitoshiIO compares the input/output of functions while observing their executions in-vivo.

Code relatives differ from all of these dynamic code similarity detection systems in that similarity is compared between the computations performed, *not* between the resulting outputs. This important distinction allows for similarly behaving code to be detected even when different data structures and output formats are used. Moreover, it allows for arbitrary code fragments to be detected as code relatives: techniques that compare output equivalence tend to work best at a per-function granularity, because that format provides a clear definition of inputs and outputs.

In addition to work on fine-granularity clones, much work has been done in the general field of detecting similarly behaving software. Marcus and Maletic propose the notion of *high level concept clones*, detecting code that addressed the same general problem, but may have significant structural differences, by using information retrieval techniques on code [39]. Similarly, Bauer et al. mine the use of identifiers to detect similar code [5]. In addition to code, several approaches analyze software artifacts such as class diagrams and design documents. This type of analysis helps developers understand similarities/differences between software at system level [9, 29].

Software birthmarking uses some representative components of a program's execution to create an obfuscation-resilient fingerprint to identify theft and reuse [47, 50]. Code relatives are comparable to birthmarks in that both capture information about how a result is calculated. However, code relatives are computed using more information than lightweight birthmarks focusing on the use of APIs [3, 36, 41, 51].

4. DETECTING CODE RELATIVES WITH LINK ANALYSIS

The high-level procedure of DyCLINK is shown in Figure 2. To begin, the program(s) to be analyzed are instrumented to allow DyCLINK to trace their respective executions. Next, the program(s) are executed given some sample inputs or workloads representative of their typical use cases, and DyCLINK creates graphs to represent executions of each program, where each instruction is represented by a vertex, and each data and control dependency is represented by an edge. Then, DyCLINK analyzes these graphs (offline) to detect code relatives. DyCLINK traces program execution, so its results will be dependent upon the inputs given to the program: some methods may not be executed at all, while others may only be executed along some specific paths. One upside to this approach is that it exposes common behavior, allowing code that handles boundary input cases and hence may not be typically executed to be ignored for the purposes of code relative detection. However, it still requires that the inputs to the program are representative of actual and typical workloads. We will discuss this design decision further in §4.4.

DyCLINK consists of two major components: online graph construction and offline (sub)graph matching. The graph constructor instruments and observes the execution of the code being evaluated to generate these dynamic dependency graphs (§4.1), while the subgraph matcher analyzes the collected graphs to detect code relatives (§4.3). We calculate the similarity of the two dynamic dependency graphs by first link-analyzing their important instructions (centroids), linearizing them into vectors, and then calculating the Jaro-Winkler distance between them. This process will be described in detail in the following sections.

We have selected Java [24] as our target language, so the instructions recorded by DyCLINK are Java bytecodes. DyCLINK makes extensive use of the ASM bytecode instrumentation library [2], requiring no modifications to the JVM to find code relatives even without source code present. To implement the graph matcher for other target languages, we could similarly use runtime binary instrumentation to capture execution graphs, an approach examined by Demme et al. [12]. The subgraph matching mechanism, which occurs offline after program execution, is language agnostic.

4.1 Constructing Graphs

To construct dependency graphs, DyCLINK follows the JVM's stack machine to derive the dependencies between instructions, recording data and control dependencies. Each execution of each method results in the generation of a new dynamic instruction dependency graph G_{dig}, where each vertex represents an instruction and each edge represents an observed dependency. These graphs contain all instructions executed both by that method, *and* by the methods that method calls. Each edge in the graph is labeled with a weight, representing the frequency of its occurrence. We consider three types of dependencies for our graphs:

- $dep_{\texttt{inst}}$: A data dependency between an instruction and one of its operands.
- $dep_{\texttt{write}}$: A read-after-write dependency on a variable.
- $dep_{\texttt{control}}$: A weighted edge indicating that some instructions are controlled by another (e.g., through a jump), corresponding to the frequency that control will follow that edge based on the observed executions.

While it is possible to set a different weight for each type of dependency, we currently weight each equally.

When one method calls another, DyCLINK stores a pointer from the calling method to its callee, allowing for code relatives to be detected that span method boundaries. This way, when a target method is examined for code relatives, DyCLINK actually considers both the trace of that method and the traces of *all methods that it calls*.

DyCLINK uses two strategies to reduce the total number of graphs recorded. First, DyCLINK stores these graphs in a flattened form — when a method calls another many times (e.g., in a loop), DyCLINK identifies that redundancy by using the number of vertices and edges as a hash value, and simply updates execution counts for each edge in the graph. Second, DyCLINK imposes a configurable quota on the number of times (q_{call}) that a given method will be captured at a given call site, which will be discussed in §5.

4.2 Example

To showcase how DyCLINK constructs a dependency graph, consider the `mult()` method in Figure 3. Figure 3a shows the Java source for this method that multiplies two numbers, while Figures 3c and 3b show the Java compiler's translation of this source code into bytecode. Consider tracing an execution of this code, using $\{a = 8, b = 1\}$ as input arguments. Figure 3d shows the graph that may be constructed from such an execution. The label of each numbered vertex is the index of a bytecode in Figure 3c, bytecodes in the `add` method (Figure 3b) are labeled as A2, A3 and A4; each edge is labeled with a counter indicating the number of times it occurred during the run. Every time that `mult()` is executed during profiling, a new G_{dig} will be generated.

To see how the edges are constructed, consider the `iload 2` instruction on line 7 (`iload x` loads a local variable `x` onto the JVM's stack). When this instruction is executed, the controlling instruction is `if_icmplt 7` at line 14, so the dependency $dep_{\texttt{control}}(14, 7)$ is constructed. Any additional dependencies are captured transitively in the graph. Because `iload 2` is reading the 2_{nd} local variable, DyCLINK detects the last instruction executed that wrote it, which is `istore 2` at line 3, creating the dependency $dep_{\texttt{write}}(3, 7)$. `invokestatic` on line 9 has two $dep_{\texttt{inst}}$ from `iload 2` and `iload 0`, because these instructions are used to invoke the `add` method. When `add` is called, its graph is stored separately, with pointers from the `mult` graph into it (vertices A2, A3 and A4). By including this callee graph (`add`) in its caller graph (`mult`), we can detect code relatives that span multiple methods.

```
1 static int mult(int a, int b) {
2   int ret = 0;
3   for(int i = 0; i < b; i++) {
4     ret = add(ret, a);
5   }
6   return ret;
7 }
8
9 static int add(int a, int b) {
10  return a + b;
11 }
```

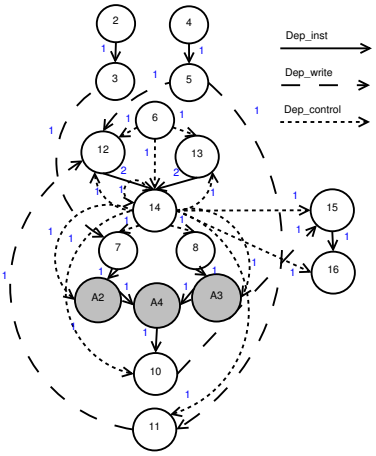
(a) The `mult()` method.

```
1 static add(II)I
2   iload 0
3   iload 1
4   iadd
5   ireturn
```

(b) The `add()` instructions.

```
1 static mult(II)I
2   iconst_0
3   istore 2
4   iconst_0
5   istore 3
6   goto 12
7   iload 2
8   iload 0
9   invokestatic add
10  istore 2
11  iinc 3 1
12  iload 3
13  iload 1
14  if_icmplt 7
15  iload 2
16  ireturn
```

(c) The `mult()` instructions.



(d) The `mult` graph.

Figure 3: The simple `mult()` method in Java (a), translated into Java bytecode (b), and a dynamic instruction dependency graph (c) generated by running `mult(8,1)`.

Once the programs are executed with sample inputs, G_{dig}s are then constructed to represent each method execution. We can proceed to the next phase, subgraph matching.

4.3 LinkSub: Link-analysis-based Subgraph Isomorphism Algorithm

To detect code relatives, DyCLINK first enumerates every pair of G_{dig}s that were constructed: given n G_{dig}s, there are at most $n * (n - 1)$ pairs to compare. Note that because each execution of each method will generate a new G_{dig}, each method will have multiple graphs that represent its executions, meaning that there are more G_{dig}s than methods. Each recorded execution of each method is potentially compared to each of the executions of each other method.

The executions are represented as graphs, so we model code relative detection as a subgraph isomorphism problem. There are two types of subgraph isomorphism (or subgraph matching): exact and inexact [44]. For exact subgraph matching, G_1 needs to have a subgraph that is entirely the same as a subgraph of G_2. Exact subgraph matching would only find cases where all instructions and their dependencies are exact copies between two code fragments; this would be too restrictive to detect code relatives. Because DyCLINK detects *similar* but *not necessarily identical* subgraphs, we are focused on techniques for inexact subgraph matching.

The key to efficiently performing this matching is to filter out pairs of graphs that can never match, reducing the

number of comparisons needed to a much smaller set. For example, for each graph, we calculate its centroid, create a simpler representation of each subgraph (simply a sequence of instructions), and then identify candidate graphs to compare it to, filtered to only those that contain that same instruction. Next, we perform a constant-time comparison between each potentially matching subgraph, calculating the euclidean distance between their instruction distributions, to eliminate unlikely matches. For the remaining subgraphs, we apply a link analysis to each subgraph to create a vectorized representation of its instructions, ordered by PageRank. From these ordered vectors, we apply an edit-distance based model to calculate similarity. Hence, we reduce the running time in two ways: we consider only potential subgraph matches that seem *likely* based on some filters, and then we calculate the actual similarity of those subgraphs.

The overall algorithm is shown at a high level in Algorithm 1. The summary of each subroutine of LinkSub is as follows:

- **profileGraph:** Computes statistical information of a G_{dig}, such as ranking of each instruction and instruction distribution to identify its centroid.
- **sequence:** Sort instructions of G_{dig} by the feature defined by the developer to facilitate locating instruction segments. We use the execution order of each instruction to sort a G_{dig}.
- **locateCandidates:** Given the centroid of a G^{te}_{dig}, locate each instance of that centroid instruction in each potential target graph G^{ta}_{dig}.
- **euclidDist:** Compute the euclidean distance between the instruction distributions of two G_{dig}s.
- **LinkAnalysis:** Apply PageRank to a graph, returning a rank-ordered vector of instructions.
- **calcSimilarity:** Calculate the similarity of two PageRank ordered instruction vector using edit distance.

LinkSub models a dynamic instruction dependency graph of a method as a network, and uses link analysis [8], specifically PageRank [33], to rank each vertex in the network. The first phase of the algorithm (`profileGraph`) ranks each vertex in the graph being examined, calculating the highest ranked vertex (*centroid*) of the graph. This step also calculates instruction distribution for subgraph matching. The next phase lists all instructions of the target graph, G^{ta}_{dig}, by execution order in the `sequence` step to facilitate locating candidate subgraphs. In the next step, `locateCandidates`, we select all subgraphs in the target graph that match the centroid of G^{te}_{dig}. If a subgraph in G^{ta}_{dig} contains the centroid instruction of G^{te}_{dig} then it is potentially a code relative, but if it does not contain the centroid instruction, then it can't be. This is effectively the first filter that reduces the largest set of potential subgraphs to compare.

For each of the potential candidate subgraphs, we next apply a simple filter (`euclidDist`) similar to [37], which computes the Euclidean distance between the distributions of instructions in the graphs from G^{te}_{dig} and a candidate subgraph from the G^{ta}_{dig}. If the distance is higher than the threshold, θ_{dist}, defined by the user, then this pair of subgraph matching is rejected. We empirically came to a threshold of 0.1 (the lower the better) to include only those subgraphs that were mostly similar.

If a candidate subgraph from the G^{ta}_{dig} passes the euclidean distance filter, DyCLINK applies its link analysis to this candidate. DyCLINK calculates a PageRank dynamic vector, DV, for the candidate subgraph (`LinkAnalysis`), where the result is a sorted vector of all of the instructions (vertices from the subgraph), ordered by PageRank.

Data: The target graph G^{ta}_{dig} and the test graph G^{te}_{dig}
Result: A list of subgraphs in G^{ta}_{dig}, $CodeRelatives$, which are similar to G^{te}_{dig}

```
//Compute Statistical Information;
profile_te = profileGraph(G^te_dig);
//Change Representation;
seq_ta = sequence(G^ta_dig);
//Filter to find possible matches;
assigned_ta = locateCandidates(seq_ta, profile_te);
CodeRelatives = ∅ ;
for sub in assigned_ta do
    //Perform multi-phase comparison;
    SD = euclidDist(SV(sub), profile_te.SV);
    if SD > θ_dist then
        continue ;
    end
    DV^sub_target = LinkAnalysis(sub);
    dynSim = calcSimilarity(DV^sub_target, profile_te.DV);
    if dynSim > θ_dyn then
        CodeRelatives ∪ sub ;
    end
end
return CodeRelatives;
```

Algorithm 1: LinkSub.

Finally, in `calcSimilarity`, we use the Jaro-Winkler Distance [10] to measure the similarity of two DVs, which represents the similarity between two G_{dig}s. Jaro-Winkler has better tolerance of element swapping in the instruction vector than conventional edit distance and is configurable to boost similarity if the first few elements in the vectors are the same. These two features are beneficial for DyCLINK, because the length of $DV(G_{dig})$ is usually long, and thus may involve frequent instruction swapping. For representing the behavior of methods, we use the PageRank-sorted instructions from $DV(G_{dig})$. If the similarity between the PageRank vectors from the subgraph of the G^{ta}_{dig} and the G^{te}_{dig} is higher than the dynamic threshold (θ_{dyn}), DyCLINK identifies this subgraph as being inexactly isomorphic to G^{te}_{dig}. We empirically evaluated several values of this threshold, settling on 0.82 as a default in §5. We refer to the subgraph similar to the G^{te}_{dig} as a *Code Relative* in the G^{ta}_{dig}.

The runtime execution cost of our algorithm will vary greatly with the number of subgraphs that remain after the two filtering stages. While each filtering stage itself is relatively cheap (the PageRank computation requires only $O(V+E)$ for a graph with V nodes and E edges), in the worst case, where we would need to calculate the Jaro-Winkler similarity of every possible pair of (sub)graphs, the overall running time would be dominated by these computations for (sub)graphs. In practice, however, we have found that these two filtering phases tend to dramatically reduce the overall number of comparisons needed, making the running time of *LinkSub* quite reasonable, requiring only 43 minutes on a commodity server to detect candidate code relatives in a codebase with over $7,000$ lines of code. Profiling this code base resulted in $1,244$ G_{dig}s, requiring a total worst-case 422+ millions of subgraph comparisons. A complete

evaluation and discussion of the scalability of our algorithm and system are in §5.1.

4.4 Limitations

There are several key limitations inherent to our approach that may result in incorrect detection of code relatives. The main limitation stems from the fact that DyCLINK captures dynamic traces: the observed inputs must exercise sufficiently diverse input cases that are representative of true application behavior. A second and related limitation comes from our design decision to declare that two code fragments are relatives if there is at least a single input pair that demonstrates the two fragments to be similar. An ideal approach would require profiling the application over large workloads representative of typical usage. If we could guarantee that the inputs observed were truly sufficiently diverse to represent typical application behavior, then it may be reasonable to consider the relative portion of inputs that result in a match compared to those that do not. However, with no guarantee that the inputs that DyCLINK observes are truly representative of the same input distribution observed in practice in a given environment, we have decided for now to instead ignore counter examples to two fragments being relatives, declaring them relatives if at least one pair of inputs provide similar behavior.

Consider the following example of a situation where these choices may result in undesirable behavior. The first method will sort an array if the passed array is non-null, returning -1 if the parameter is null. The second method will read a file if the passed file is non-null, returning -1 if the parameter is null. If DyCLINK observes executions of each method with a null parameter, then these two methods will be deemed code relatives, because there is at least one input pair that causes them to exhibit similar behavior. A future version of DyCLINK could instead consider all of the inputs received, and the coverage of each of those inputs towards being representative of overall behavior.

DyCLINK can also fail to detect code that is similar in terms of its input and output functionality if it has different instruction-level behavior. For example, a method can multiply two integers, $\{a, b\}$, in a convoluted way as Figure 3 depicts, or it can simply return $a * b$. By our definitions, these are not code relatives, and wouldn't be detected as such by DyCLINK.

Due to nondeterminism in a running program, DyCLINK may record different execution graphs, causing results to vary slightly between multiple profiling runs. In multi-threaded applications, DyCLINK currently only considers code fragments that execute within the same thread as code relatives - there is no merging of graphs across threads.

5. EVALUATION

We evaluate DyCLINK in terms of both runtime performance and precision. We answer two research questions:

- **RQ1:** Given the potentially immense number of subgraph comparisons, is DyCLINK's performance feasible to scale to large applications?
- **RQ2:** Are the code relatives detected by DyCLINK more precise for classifyling programs than are the similar code fragments found by previous techniques?

Unfortunately, we are limited in our choice of experimental subjects and comparison approaches by what is publicly available. For example, while there are publicly available benchmarks of code clones [49] with a ground truth manually provided, we found many of them did not include sufficient dependencies and build scripts to be compiled and executed dynamically. To focus our evaluation on projects that were build-able and distributed with inputs/test cases, we selected projects from the Google Code Jam repository [18]. Google Code Jam is an annual online coding competition hosted by Google. Participants submit their projects' source code online, and Google determines whether they correctly solve a given problem. Because each submission for the same problem attempts to perform the same task, we assume that each project within the same year will likely share code relatives, while projects between different years solving completely different requirements will likely not share code relatives or at least fewer.

Table 1: A summary of the code subjects from the Google Code Jam competition for classifying software.

		# Proj		Graph Size			
Year	Problem	Tot.	Aut.	Meth.	#	V_{avg}	V_{max}	E_{avg}
2011	Irregular Cake	48	30	106/154	367	398	6698	958.1
2012	Perfect Game	48	34	122/182	195	138.2	2001	276.6
2013	Cheaters	29	21	95/147	374	283.4	2456	631.7
2014	Magical Tour	46	33	105/159	308	223.6	3709	480.5

To compare DyCLINK's code relative detection with static code clone detection, we selected the state-of-the-art clone detector available, SourcererCC [46]. While SourcererCC is highly performant, scaling impressively to "big code", we admittedly do not expect to find many near-miss static code clones in independently written Code Jam entries. In contrast, we would expect to find clusters of dynamic functional I/O simions, since the independently written entries intend to complete the same tasks. Previous simion detectors for object-oriented languages do not address project-specific object data types, due to the technical challenges reported by Deissenboeck et al. [11]. Therefore, we developed a simion detector that we have recently built for Java, HitoshiIO [48], specifically designed to overcome these challenges and enable fair comparison of the similarity models.

The information on the evaluation subjects is shown in Table 1. For each competition year, we show the problem name, the number of projects in the repository, the number of automatic projects without human interactions used in this study, the total number of executed methods in those projects and the statistics for the captured G_{dig}s including the number of graphs and the numbers of vertices and edges. For the executed methods, we provide two numbers: *retained*/*all*. To avoid potentially inflating our results by including matches of trivial methods, we filter out simple methods with little work in them (such as toString and initialization methods). *all* represents the number of all executed methods, while *retained* shows the method number after such filtering.

We discuss some parameter settings of DyCLINK for conducting the experiments in this paper. For constructing G_{dig}s in §4.1, we empirically set the quota at a given call site, q_{call} as 5. This allows for reasonable performance both in terms of code relative detection and runtime overhead. For conducting the inexact (sub)graph matching, we set θ_{dist} as 0.1 and θ_{dyn} as 0.82 in Algorithm 1, where both parameters

Table 3: Code Relatives, Simions and Code Clones detected by project-year and by tool for DyCLINK, HitoshiIO and SourcererCC, using each tool's default settings.

Years Compared	DyCLINK	HitoshiIO	SourcererCC
2011-2011	103	21	5
2012-2012	49	59	13
2013-2103	116	181	6
2014-2014	66	43	4
2011-2012	3	19	9
2011-2013	0	9	9
2011-2014	0	19	6
2012-2013	7	6	15
2012-2014	3	25	8
2013-2014	81	24	16
Total	428	406	91

range from 0 to 1. The details of each parameter setting can be found in the GitHub page of DyCLINK [1]. While searching for the best parameter setting for DyCLINK is out of the scope of this paper, we plan to utilize machine learning techniques for optimizing DyCLINK in future.

5.1 RQ1: Scalability

To evaluate the scalability of DyCLINK, we measured its performance when running on these 118 projects. The key to DyCLINK's performance is the relative reduction in subgraph comparisons that result from filtering and link analysis steps. If we can greatly reduce the number of candidate subgraphs to be compared, then DyCLINK will scale, even on large graphs. Table 2 shows the worst case number of pairwise comparisons that would be needed by a naive subgraph matching algorithm, along with the number of comparisons that were actually necessary to detect the code relatives. We also show the analysis time for each of DyCLINK, HitoshiIO, and SourcererCC.

DyCLINK filtered out over 99% of the potential subgraphs to compare, resulting in a total analysis time of just 43 minutes on an Amazon EC2 "c4.8xlarge" instance. While this analysis time is significantly longer than the static approach, and still more than the simion detector, we believe that the analysis runtime is acceptable given the time complexity to solve the inexact (sub)graph matching problem.

Because DyCLINK is a dynamic profiling approach, there is also a time overhead for collecting the traces and generating the graphs. Our execution tracer implementation is unoptimized and records every single instruction. An optimized version might instead be able to infer and record instructions that expose program behaviors. To trace these applications took a total time of just over 2.5 hours compared to a baseline execution time without instrumentation of approximately 1 minute on an iMac with 8 cores and 32 GB memory; however the instrumentation overhead can vary significantly with the complexity of the program — one single subject took 114 minutes to execute, while the remaining 117 required only a total of 43 minutes to execute. We are confident that the tracing overhead can be significantly reduced with some optimizations as demonstrated by other Java tracing systems, such as JavaSlicer [19].

5.2 RQ2: Code Relative Detection

We first evaluate the quality of the code relatives detected by DyCLINK by looking at the number of code relatives detected in projects across and within each year. For this evaluation, we ran each tool with its default similarity threshold (0.82 for DyCLINK, 0.85 for HitoshiIO and 0.7 for SourcererCC), and a minimum code fragment size of 10 lines of code (45 instructions for DyCLINK). Table 3 shows the number of code relatives detected by DyCLINK as well as the number of code clones detected by the other two systems. DyCLINK detected more similar code fragments on average than the other systems did. Those relatives were skewed to be almost entirely among projects within the same year, while the other tools tended to find similar code fragments more evenly distributed among and within the project years (recall that all projects in the same year performed the same task). This result is encouraging, as we expect that there are more code relatives in code that has the same general purpose than in code that is doing different tasks.

Figure 4 shows an exemplary pair of similar code fragments detected by DyCLINK in Code Jam projects. The two caller methods, `calcMaxBet` and `maxBet`, exhibit similar functionality to maximize bets, so both of them are detected by DyCLINK and HitoshiIO. However, even though their subroutines, `canDo` and `cost`, have similar behavior to evaluate costs, HitoshiIO cannot detect them as functionally similar by observing their I/Os. The reason is that their output values will be hard to detect as similar: while `canDo` performs a comparison between the cost and budget and returns a boolean, `cost` solely computes the cost and leaves the comparison for its caller `maxBet`. This example shows the difficulty to detect dynamic code similarities by observing functional I/Os of programs.

We did not conduct a user study as part of this experiment other than random sampling performed by the authors to ensure the relatives reported were valid. To judge the system accuracy, we investigated specifically its precision in a software clustering experiment.

Software Community Clustering. To judge the efficacy of DyCLINK in performing software clustering, we applied a KNN-based classification algorithm to the Google Code jam projects. Again, our ground truth is that projects from the same year solving the same problem ought to be in the same cluster.

We apply the *K-Nearest Neighbors (KNN)* classification algorithm to predict the label (project year) for each method and then validate the prediction result by *Leave-One-Out* methodology: each sample (method) plays as a testing subject exactly once, where all the rest of the samples play as the training data. The high-level algorithm is shown in Algorithm 2: for each method, we search for the K other methods that have the greatest similarity to the current one in the `searchKNN` step. Each nearest neighbor method can vote for the current method by its real label in the `vote` step. The label voted by the greatest number of neighbor methods becomes the predicted label of the current method. In the event of a tie, we side with the neighbors with the highest sum of similarity scores. Then, we track the precision of the approach as the total number of correctly labeled methods divided by the total number of methods.

For observing the efficacy of the systems under single and multiple neighbors, we set $K = 1$ and $K = 5$. We also vary the line of code thresholds used for each code fragment's minimum size between $\{10, 15, 20, 30\}$. Only programs that pass the threshold setting including LOC and similarity were

Table 2: Number of comparisons performed by DyCLINK on the Google Code Jam projects, showing worst case number of comparisons (without any filtering) and actual comparisons performed along with the relative reduction in comparisons achieved by DyCLINK. We also show the total analysis time needed to complete each set of comparisons.

Years Compared	Subgraphs Compared			Analysis Time (sec)		
	Worst Case	Actual	Reduction	DyCLINK	HitoshiIO	SourcerCC
2011-2011	49,999,944	258,478	99.48%	836.38	64.00	4.1
2012-2012	5,006,827	7,719	99.85%	14.88	49.00	4.4
2013-2103	35,186,281	280,355	99.2%	392.73	51.00	3.9
2014-2014	19,017,387	123,196	99.35%	230.39	53.00	4.3
2011-2012	38,371,375	12,221	99.97%	49.77	133.00	4.9
2011-2013	93,519,230	45,822	99.95%	193.55	125.00	5.0
2011-2014	70,260,597	10,396	99.99%	70.98	133.00	4.9
2012-2013	30,745,400	32,621	99.89%	68.15	96.00	5.1
2012-2014	21,730,445	31,151	99.86%	63.96	114.00	5.0
2013-2014	58,399,594	460,750	99.21%	653.44	105.00	4.7
Total	422,237,080	1,262,709	99.7%	2574.23	923.00	46.3

Data: The similarity computation algorithm $SimAlg$, the set of subject methods to be classified $Methods$ and the number of the neighbors K
Result: The precision of $SimAlg$
realLabel($Methods$);
$matrix_{sim}$ = computeSim($SimAlg$, $Methods$);
$succ = 0$;
for m *in* $Methods$ **do**
  $neighbors$ = searchKNN(m, $matrix_{sim}$, K);
  $m.predictedLabel$ = vote($neighbors$);
  if $m.predictedLabel = m.realLabel$ **then**
    $succ = succ + 1$;
  end
end
$precision = succ/Methods.size$;
return $precision$;

Algorithm 2: Procedure of the KNN-based software label classification algorithm.

considered as neighbors of the current program.

The results of this analysis are shown in Table 4: DyCLINK showed the highest precision among all three techniques when examining code fragments that consisted of at least ten lines of code. When excluding the smallest fragments (for example, looking only at those with 20 lines of code or more), the simion detector HitoshiIO performed slightly better. The methods being incorrectly categorized by HitoshiIO were mostly less than 20 lines of code. SourcererCC did not find sufficient clones that were longer than 30 lines of code to allow for clustering at that level, and hence, the precision value is not available. Because we use the project year as the label for each method, it is possible that some syntactically similar code detected by SourcererCC is not identified as a true positive case.

Figure 5 shows the clustering matrix based on DyCLINK's KNN-based classification result with K = 1, LOC = 10. Each element on both axes of the matrix represents a project indexed by the abbreviation of the problem set to which it belongs and the project ID. We sort projects by their project indices. Only projects that have at least one code relative with another project are recorded in the matrix. The color of each cell represents the relevance between the i_{th} project and the j_{th} project (the darker, the higher), where i and j represent the row and column in the matrix. The project relevance is the number of code relatives that two projects share. Each block on the matrix forms a *Software Community*, which fits in the problem sets that these projects aim to solve. These results show that DyCLINK is capable of detecting programs with similar behavior and then cluster them for further usage such as code search.

5.3 Discussion

Through this evaluation, we have shown that DyCLINK is an effective tool for detecting similar code fragments. There are several potential limitations to our experiments, however. Even though we may have manually come to the conclusion that two code fragments are code relatives and assuming that we are internally valid in that conclusion, two developers' definitions of "similarly behaving" code may differ. We believe that we have limited the potential for this bias through our study design: we purposely selected a suite of projects that are known to be likely to contain similarly behaving code, because they were performing the same overall task. Hence, when we conclude that DyCLINK is effective at finding behaviorally similar code, we come to this conclusion both from our internal review and also from the external construction, that by definition, the code ought to behave similarly (at least on some scale).

However, this selectivity comes at a cost: the projects that we selected might be too homogeneous overall, and not sufficiently representative of software in general. We could bolster our claims by performing a broader study on, for instance, large open-source projects from GitHub. We could construct a user study to help establish a ground truth for what "similar code" really is.

Dynamic analysis and static analysis have their own pros and cons in detecting different types of similar code. Thus, we plan to combine the advantages of DyCLINK and SourcererCC to devise a new approach to detect similar code fragments more effectively with better efficiency.

6. CONCLUSIONS

Determining when two code fragments are "similar" is a subjective and complex problem. We have distilled the problem of detecting behaviorally similar code fragments into a

Table 4: Precision results from KNN classification of the Google Code Jam projects using DyCLINK, HitoshiIO and SourcererCC, while varying K and the minimum fragment length considered. A value of N/A means no sufficient clones were categorized into the project year.

Min Fragment Size	K=1			K=5		
	DyCLINK	HitoshiIO	SourcererCC	DyCLINK	HitoshiIO	SourcererCC
10	0.94	0.81	0.35	0.91	0.77	0.34
15	0.94	0.86	0.48	0.92	0.86	0.45
20	0.87	0.95	0.55	0.90	0.95	0.45
30	0.92	0.91	N/A	0.91	0.91	N/A

```
static long calcMaxBet(long budget,
  long[] x,int winningThings) {
  ...
  if (canDo(budget, x, winningThings, mid)) {
    low = mid;
  } else { high = mid; }
  ...
}

static boolean canDo(long budget,
  long[] x,int winningThings,long lowestBet) {
  long payMoney = 0;
  for (int i = 0; i < x.length; i++) {
    if (x[i] < lowestBet) {
      payMoney += -x[i] + lowestBet;
    }
  }
  return payMoney <= budget;
}
```

(a) The call sequence includes the `canDo` method

```
long maxBet(long[] a,int count,long b) {
  ...
  if (cost(a, count, mid) <= b) {
    left = mid;
  } else { right = mid; }
  ...
}

long cost(long[] a,int count,long bet) {
  long result = 0;
  for (int i = 0; i < count; i++) {
    result += (bet - a[i]);
  }
  for (int i = count; i < a.length; i++) {
    if (a[i] <= bet) {
      result += (bet + 1 - a[i]);
    }
  }
  return result;
}
```

(b) The call sequence includes the `cost` method

Figure 4: An exemplary code relative in Google Code Jam.

subgraph isomorphism problem based on dynamic dependency graphs that capture instruction-level behavior. To feasibly analyze the hundreds of millions of potential matching subgraphs, we have devised a novel link-analysis based algorithm, *LinkSub*, which greatly reduces the number of pairwise comparisons needed to detect code relatives, and then efficiently compares them using PageRank. DyCLINK detects behaviorally similar code better than previous approaches, and has reasonable runtime overhead. We have released DyCLINK under an MIT license on GitHub [1]. A tutorial regarding how to use DyCLINK can be found in §7.

One key limitation of our approach is from its dynamic nature: because it relies on program execution traces to detect code relatives, it is only applicable to situations where the subject code can be executed. In addition to being executable at all, there must be valid inputs that are representative of a program's normal behavior to expose its typical use cases and generate representative traces. In our previous work [48], we applied applications' existing test suites for this purpose, but recognize that test suites may not be truly representative of application usage. Alternatively, automated input generation tools [16, 43] could be used to drive the application. We plan to experiment with input generation techniques, allowing us to apply DyCLINK to larger scale systems than studied in this paper. Furthermore, we plan to construct a benchmark suitable for use for dynamic code similarity detection. This benchmark would contain not only workloads and scripts to compile and run each application, but also a human-judged ground-truth of program similarity, analogous to the static code clone benchmark, BigCloneBench [49].

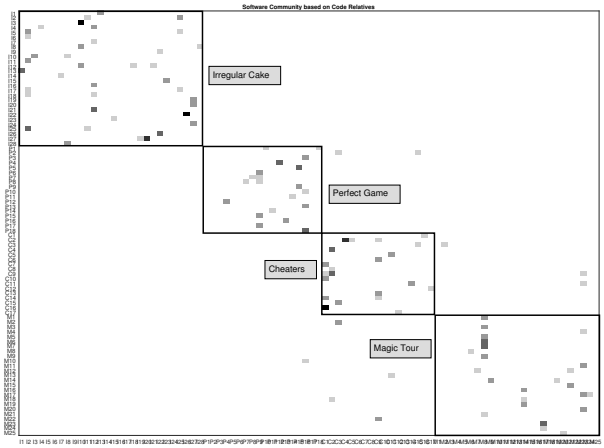


Figure 5: The software community based on code relatives detected by DyCLINK. The darker color in a cell represents a higher number of code relatives shared by two projects.

We also plan to study additional applications of our link-analysis based graph comparison algorithm. For example, we plan to explore the possibility to apply DyCLINK to support software development tasks related to behavior, such as (semi)automatic API generation and code search. Currently, DyCLINK can cluster programs having similar behavior. How to normalize these programs to create a centralized API can be a challenging topic to study.

7. ARTIFACT DESCRIPTION

We provide a tutorial to replay the result of Table 3. A virtual machine (VM) containing DyCLINK and all required software can be accessed from DyCLINK's Github page [1]. Users can first read §7.7 to check the VM's limitation. We conducted our experiments on an iMac with 8 cores and

32 GB memory to construct graphs (§7.4) and Amazon ec2 "c4.8xlarge" instances to match graphs (§7.5).

7.1 Required Software Suites

If the user chooses to use our VM, this step can be skipped. The user needs to install JDK 7 [20] to execute our experiments on DyCLINK. DyCLINK is a Maven project [40]. If the user wants to re-compile DyCLINK, the installation of Maven is required. DyCLINK needs a database system and GUI to store/query the detected code relatives. We use MySQL and MySQL Workbench. For downloading and installing them, the user can check MySQL's website [42]. For setting up the database, the user can find more details in `dycl_home/scripts/db_setup`, where `dycl_home` represents the home directory of DyCLINK.

7.2 Virtual Machine

We set up the credential with "dyclink" as the username and "Qwerty123" as the password for our VM. The home of DyCLINK is `/home/dyclink/dyclink_fse/dyclink`. For starting MySQL, the user can use the command `sudo service mysql start`. The credential for MySQL is "root" as the username and "qwerty" as the password.

7.3 System Configuration

Before using DyCLINK, the user needs to change to the home directory of DyCLINK. The user first uses the command `./scripts/dyclink_setup.sh` to create all required directories for executing DyCLINK. DyCLINK has multiple parameters to specify in the configuration file: `config/mib_config.json`. For reproducing the experimental results, the user can simply use the this configuration file.

7.4 Dynamic Instruction Graph Construction

We put our codebases for the experiments under `codebase/bin`. The user will find 4 directories from "R5P1Y11" to "R5P1Y14". These 4 directories contain all Google Code Jam projects we used in the paper from 2011 to 2014.

Before executing the projects in a single year, the user needs to specify the graph directory for the `graphDir` field in the configuration file. This is to tell DyCLINK where to dump all graphs. For example, the user sets `graphDir` to `graphs/2011` for storing graphs of the projects in 2011. We have created subdirectories for each year under `graphs`.

We prepare a script to automatically execute all projects in a single year: `./scripts/exp_const.sh $yearDir`. For example, the user can execute all projects in 2011 by the command `./scripts/exp_const.sh R5P1Y11`. Most years can be completed between 0.5 to 3 hours on the VM, but 2013 may cost 20+ hours and need more memory.

The `cache` directory records cumulative information for constructing graphs. If users fail any year, they need to first clean the `cache` directory and reset `threadMethodIdxRecord` in the configuration file to be empty, and re-run every year.

7.5 (Sub)graph similarity computation

Because we compute the similarity between each graph within and between years, there will be totally 10 comparisons. For storing the detected code relatives in the database, the user needs to specify the URL and the username in the configuration file.

For computing similarities between graphs in the same year, the user can issue `./scripts/dyclink_sim.sh -iginit -target graphs/$year`, where `$year` is between $\{2011, 2014\}$. For different years, the command is `./scripts/dyclink_sim.sh -iginit -target graphs/$y1 -test graphs/$y2`, where `$y1` and `$y2` are between $\{2011, 2014\}$. DyCLINK will then prompt for user's decision to store the results in the database The user needs to answer "true".

On the VM, we suggest the user to detect code relatives for $2011-2012$, $2011-2014$, $2012-2012$ and $2012-2014$, if we exclude the projects in 2013. The other 6 comparisons may take 20+ hours to complete on the VM.

7.6 Result Analysis

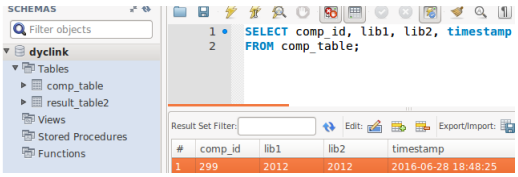


Figure 6: The exemplary UI of MySQL Workbench to check the comparison ID.

For analyzing code relatives for a comparison, the user needs to retrieve the comparison ID from the `dyclink` database. The user first queries all comparisons by the SQL command as Figure 1 shows via MySQL Workbench, and then checks the ID for the comparison. `lib1` and `lib2` show the years (codebases) in a comparison. If the values for `lib1` and `lib2` are different such as $2011-2012$, this comparison contains the code relatives *between* different years. If the values are the same such as $2012-2012$, this comparison is *within* the same year. Figure 6 checks the comparison ID (299) for code relatives within 2012 ($2012-2012$).

For computing the number of code relatives, the user can use the command `./scripts/dyclink_query.sh $compId $insts $sim -f` with 4 parameters. The `$compId` represents the comparison ID. The `$insts` represents the minimum size of code relatives with 45 as the default value. The `$sim` represents the similarity threshold with 0.82 as the default value. The flag `-f` filters out simple utility methods in our codebases. An exemplary command for the $2012-2012$ comparison with `$compId` $= 299$ is `./scripts/dyclink_query.sh 299 45 0.82 -f`.

7.7 Potential Problems

The major potential problem is the performance and memory of VM. Some experiments regarding 2013 may cost too much time and need more memory than the VM has. If the `OutOfMemoryError` occurs, the user can increase the memory for the VM and sets `-Xmx` for JVM in the corresponding commands under the `scripts` directory.

8. ACKNOWLEDGMENTS

We thank the authors of SourcererCC for their advice, suggestions and guidance in running and evaluating their tool. We also thank Apoorv Prakash Patwardhan for analyzing the results of SourcererCC and Sriharsha Gundappa for preparing a virtual machine of DyCLINK. Finally, we appreciate the valuable comments from our reviewers for this paper and the corresponding artifact. This work is supported in part by NSF CCF-1302269 and CCF-1161079.

9. REFERENCES

[1] Dyclink github page. https://github.com/Programming-Systems-Lab/dyclink.

[2] Asm framework. http://asm.ow2.org/index.html.

[3] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining apps for abnormal usage of sensitive data. In *2015 International Conference on Software Engineering (ICSE)*, ICSE '15, pages 426–436, 2015.

[4] B. S. Baker. A program for identifying duplicated code. In *Computer Science and Statistics: Proc. Symp. on the Interface*, pages 49–57, 1992.

[5] V. Bauer, T. Völke, and E. Jürgens. A novel approach to detect unintentional re-implementations. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 491–495, Washington, DC, USA, 2014. IEEE Computer Society.

[6] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–377, 1998.

[7] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 195–205, 2004.

[8] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International Conference on World Wide Web 7*, WWW7, pages 107–117, 1998.

[9] G. Canfora, L. Cerulo, and M. D. Penta. Tracking your changes: A language-independent approach. *IEEE Software*, 26(1):50–57, 2009.

[10] W. W. Cohen, P. Ravikumar, and S. E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proceedings of IJCAI-03 Workshop on Information Integration*, pages 73–78, 2003.

[11] F. Deissenboeck, L. Heinemann, B. Hummel, and S. Wagner. Challenges of the dynamic detection of functionally similar code fragments. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 299–308, March 2012.

[12] J. Demme and S. Sethumadhavan. Approximate graph clustering for program characterization. *ACM Trans. Archit. Code Optim.*, 8(4):21:1–21:21, Jan. 2012.

[13] N. DiGiuseppe and J. A. Jones. Software behavior and failure clustering: An empirical study of fault causality. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 191–200, 2012.

[14] M. Egele, M. Woo, P. Chapman, and D. Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 303–317, 2014.

[15] R. Elva and G. T. Leavens. Semantic clone detection using method ioe-behavior. In *Proceedings of the 6th International Workshop on Software Clones*, IWSC '12, pages 80–81, 2012.

[16] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM.

[17] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 321–330, 2008.

[18] Google code jam. https://code.google.com/codejam.

[19] C. Hammer and G. Snelting. An improved slicer for java. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '04, pages 17–22, New York, NY, USA, 2004. ACM.

[20] Oracle jdk 7. http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html.

[21] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, 2007.

[22] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 81–92, 2009.

[23] E. Juergens, F. Deissenboeck, and B. Hummel. Code similarities beyond copy & paste. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 78–87, Washington, DC, USA, 2010. IEEE Computer Society.

[24] Java virutal machine speicification. http://docs.oracle.com/javase/specs/jvms/se7/html/. Accessed: 2015-02-04.

[25] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, July 2002.

[26] H. Kim, Y. Jung, S. Kim, and K. Yi. Mecc: Memory comparison-based clone detector. ICSE '11.

[27] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 40–56, 2001.

[28] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 253–262, 2006.

[29] S. Kpodjedo, F. Ricca, P. Galinier, G. Antoniol, and Y.-G. Gueheneuc. Madmatch: Many-to-many approximate diagram matching for design comparison. *IEEE Transactions on Software Engineering*, 39(8):1090–1111, 2013.

[30] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 301–309, 2001.

[31] D. E. Krutz and E. Shihab. Cccd: Concolic code clone detection. In *Reverse Engineering (WCRE), 2013 20th*

Working Conference on, pages 489–490, Oct 2013.

[32] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Inf. Softw. Technol.*, 49(3):230–243, Mar. 2007.

[33] P. Lawrence, B. Sergey, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford University, 1998.

[34] S. Li, X. Xiao, B. Bassett, T. Xie, and N. Tillmann. Measuring code behavioral similarity for programming and software engineering education. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 501–510, 2016.

[35] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 176–192, 2004.

[36] M. Linares-Vásquez, C. Mcmillan, D. Poshyvanyk, and M. Grechanik. On using machine learning to automatically classify software applications into domain categories. *Empirical Softw. Engg.*, 19(3):582–618, June 2014.

[37] C. Liu, C. Chen, J. Han, and P. S. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 872–881, 2006.

[38] J. I. Maletic and N. Valluri. Automatic software clustering via latent semantic analysis. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, ASE '99, pages 251–, 1999.

[39] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE '01, pages 107–114, 2001.

[40] Apache maven. `https://maven.apache.org`.

[41] C. McMillan, M. Grechanik, and D. Poshyvanyk. Detecting similar software applications. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 364–374, 2012.

[42] Mysql database. `https://www.mysql.com`.

[43] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[44] K. Riesen, X. Jiang, and H. Bunke. Exact and inexact graph matching: Methodology and applications. In *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*, pages 217–247. Springer, 2010.

[45] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, May 2009.

[46] H. Sajnani, V. Saini, J. Svajlenko, C. Roy, and C. Lopes. SourcererCC: Scaling Code Clone Detection to Big Code. ICSE '16.

[47] D. Schuler, V. Dallmeier, and C. Lindig. A dynamic birthmark for java. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 274–283, New York, NY, USA, 2007. ACM.

[48] F.-H. Su, J. Bell, G. Kaiser, and S. Sethumadhavan. Identifying functionally similar code in complex codebases. In *Proceedings of the 24th IEEE International Conference on Program Comprehension*, ICPC 2016, 2016.

[49] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia. Towards a Big Data Curated Benchmark of Inter-project Code Clones. ICSME '14.

[50] H. Tamada, M. Nakamura, and A. Monden. Design and evaluation of birthmarks for detecting theft of java programs. In *In Proc. IASTED International Conference on Software Engineering*, pages 569–575, 2004.

[51] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE '15, pages 303–313, 2015.