C-TRAIL: A Program Comprehension Approach for Leveraging Learning Models in Automated Code Trail Generation

Roy Oberhauser
Department of Computer Science, Aalen University, Aalen, Germany

Keywords: Program Code Comprehension, Learning Models, Recommender Systems, Obfuscation.

Abstract: With society's increasing utilization of (embedded) software, the amount of program source code is proliferating while the skilled human resources to maintain and evolve this code remain limited. Therefore, software tools are needed that can support and enhance program code comprehension. This paper focuses on program concept location and cognitive learning models, and contributes an automatic code trail generator approach called a Code Trail Recommender Agent Incorporating Learning models (C-TRAIL). Initial empirical results applying the prototype on obfuscated code show promise for improve program comprehension efficiency and effectiveness.

1 INTRODUCTION

The software industry continues to struggle to meet society's seemingly insatiable demand for software production and maintenance. Indicators for the immensity of the problem include code size, the lack and turnover of human resources, and costs. It has been estimated that well over a trillion lines of code (LOC) exist with 33bn added annually (Booch, 2005). E.g., Google has 2bn LOC accessible by 25K developers (Metz, 2015). Active open source projects double in size and number in ~14 months (Deshpande & Riehle, 2008). Conversely, the pool of programmers is not growing correspondingly. E.g., Computer Science degrees in 2011 in USA were equivalent to 1986 in number (~42K) and percentage of 23 year olds (~1%) (Schmidt, 2015). The situation is exacerbated by the typically high employee turnover rates for software companies, e.g., 1.1 years at Google (PayScale, 2016). As to costs, Y2K exacted >$300bn globally (Mitchell, 2009)., while >50% of information systems in the EU needed modification for Euro support (Jones, 2006).

Given limited resources and such a vast amount of code, ~75% of technical software workers are estimated to be doing maintenance (Jones, 2006). Moreover, program comprehension may consume up to 70% of the software engineering effort (Minelli, 2015). Activities involving program comprehension include investigating functionality, internal structures, dependencies, run-time interactions, execution patterns, and program utilization; adding or modifying functionality; assessing the design quality; and domain understanding of the system (Pacione et al., 2004).

One key challenge faced by programmers when presented with an unfamiliar preexisting program codebase is how to become sufficiently familiar with relevant areas in a short time. Questions include: Where should one start? What should one look at next? What is relevant to know and what is optional?

To improve this program comprehension situation, the solution approach Code Trail Recommender Agent Incorporating Learning models (C-TRAIL) contributes a code trail recommender approach builds on our prior work (Oberhauser, 2016) by amalgamating diverse cognitive learning model styles with granular computing, collaborative filtering, and the traveling salesman paradigm. Given only the program code, C-TRAIL provides a web service offering automated code trail guidance to help the user avoid missing relevant areas, avoid dead ends, avoid reorientation waste, and avoid irrelevant areas. Analogous to geographic route planning via navigation software, it readjusts on-the-fly to trail deviations and replans the route.

The paper is organized as follows: Section 2 discusses related work. Section 3 describes the solution concept followed by its realization. Section 5 evaluates the solution, followed by a conclusion.

Oberhauser, R.
C-TRAIL: A Program Comprehension Approach for Leveraging Learning Models in Automated Code Trail Generation.
DOI: 10.5220/0005974901770185
In *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 1: ICSOFT-EA*, pages 177-185
ISBN: 978-989-758-194-6
Copyright © 2016 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

2 RELATED WORK

(Robillard et al, 2014) provides an overview of recommendation systems in software engineering. Mylar (Kersten & Murphy, 2005) utilizes a degree-of-interest model to filter out irrelevant files from the File Explorer and other views in the Eclipse integrated development environment (IDE). NavTracks (Singer et al., 2005) recommends files related to the currently selected files based on previous navigation patterns. For maintenance tasks in unfamiliar projects, Hipikat (Čubranić et al., 2005) recommends software artifacts relevant to a context based on the source code, email discussions, bug reports, change history, and documentation. The FEAT tool uses concern graphs either explicitly created by a programmer or automatically inferred based on navigation pathways utilizing a stochastic model, whereby a programmer confirms or rejects them for the concern graph (Robillard & Murphy, 2003). The Eclipse plugin Suade supports drag-and-drop of related fields and methods into a view to specify a context, and Suade utilizes a dependency graph and heuristics to recommend suggestions for further investigation (Robillard, 2008). Codetrail (Goldman & Miller, 2009) connects source code and hyperlinked web resources via Eclipse and Firefox. (Yin et al., 2010) propose applying coarse-grained call graph slicing, intra-procedural coarse-grained slicing, and a cognitive easiness metric to guide programmers from the easiest to the hardest non-understood methods. (Cornelissen et al., 2009) survey work on program comprehension via dynamic analysis.

In contrast, C-TRAIL automatically generates a code-centric time-limited trail of relevant areas via a web service, ordered based on the selected learning model while not requiring a project history or visualization paradigm. Visualization also has the potential issue of information overload versus relevance, and auto-generated diagrams face ideal element placement issues. Human-generated diagrams may not remain consistent, and may reflect abstractions but still leave a user unfamiliar with the code. Furthermore, the user internal cognitive model may not adhere to a presented visual model, while visual-text paradigm switching may distract or be cognitively burdensome. Support for not navigating class relationships includes the empirical eye-tracking study finding that "software engineers do not seem to follow binary class relationships, such as inheritance and composition" (Guéhéneuc, 2006).

3 SOLUTION APPROACH

Concepts are the fundamental building blocks of knowledge and human learning, and are processable by the human mind, exhibit some perceived regularity, and can be designated by a label (Rajlich & Wilde, 2002). Hence, we designate *concept location* as the understanding about where a concept is implemented in code relative to other concepts, which is the primary focus of this paper within the larger sphere of program comprehension. While the exact identification of concepts and their locations in a program remains an open problem, our solution takes a pragmatic approach utilizing the existing modularization within the program, especially method to class and class to package relationships.

We assume a program comprehension activity is time constrained, and that it is unrealistic to understand a sufficiently large codebase in its entirety (Rajlich & Wilde, 2002), nor is it necessary or always possible (Lakhotia, 1993). Thus, an inherent trade-off is assumed between *sufficient coverage* (ensuring that at least the most essential program areas were presented) and *relevance* (minimizing irrelevant or optional program areas).

Given the diversity of individuals, comprehension activities and intentions (Pacione et al., 2004), programming languages, tooling, and environments, we chose to support comprehension via an automated approach that: 1) recommends a code-centric navigation, 2) supports a spectrum of learning models, 3) utilizes individual profiles and collaborative filtering, and 4) can be readily integrated in various tools and environments.

3.1 Cognitive Learning Models

In the constructivist theory of human learning, humans actively construct their knowledge (Novak, 1998). We thus view program comprehension as individualistic for aspects such as capacity, speed, motivation, and how mental models are constructed. Additionally, programmers possess different application-independent general and application-specific domain knowledge. Information processing habits of an individual are known as cognitive learning styles. C-TRAIL provides individual and automated support for various *learning model* (M:) styles, primarily ordering or adjusting concept location (code area) visitation scope.

M:Bottom-Up: in this learning model, chunking (Letovsky, 1986) is used with the program model being correlated with a situation model (Pennington, 1987). Microstructures are mentally chunked into

larger macrostructures as comprehension increases. C-TRAIL assumes a package hierarchy.

M:Top-Down: this model (Soloway et al., 1988) is typically applicable when familiarity with the code, system, domain, or similar system structures already exists. Beacons and rules of discourse are used to hierarchically decompose goals and plans. To automate support, C-TRAIL assumes a cluster hierarchy and starts trails from the highest hierarchy.

M:Topics/Goal: when programmers are given a specific task, they tend to utilize an as-needed strategy to comprehend only those portions relevant for the task (Koenemann & Robertson, 1991). To support this simply, C-TRAIL supports investigating a limited code subset via topic filtering. Topic filters (positive and negative) can be shared and support a goal (e.g., optimize memory) or apply to a specific topic (e.g., security, database access, user interface).

M:DynamicPath: in this model, ordering is oriented on actual invocation execution traces (Cornelissen et al., 2009).

M:Exploratory: this model supports either discovery or analysis to confirm a hypothesis, with the learner actively deciding and controlling the navigation. It is supported by default, since a user can deviate at any time.

3.2 Solution Principles

C-TRAIL includes these solution principles (P:):

P:POI: code locations, currently at the granularity of functions or methods, are considered *concept locations* identified and viewed as Points-of-Interest (POI), a knowledge concept in a knowledge landscape (the codebase) or a granule (here a cluster of code lines) in granular computing paradigm (Bargiela & Pedrycz, 2012), analogous to geographical locations in navigational systems. A POI is identified by a unique identifier, such as a fully qualified name (FQN) in the Java programming language (concatenating its package name, class name, colon, and its method name).

P:POILocality: conceptually, POIs can be viewed from the perspective of knowledge distance (Qian et al., 2007) or closeness (locality). To reduce the cognitive burden of code context switches, POI visitations are ordered and clustered by locality to reduce unnecessary switches. The *T:POI Distance* technique (Section 3.4) is currently used.

P:POIRanking: a POI's relative importance for comprehension is ranked in accord with a learning model. Statically, the *T:MethodRank* technique (Section 3.4) or a dynamic analysis can be applied.

P:POIFiltering: topic or named goal selection supports a positive/negative POI filtering, currently via FQN pattern matching.

P:POIVisitTime: given no initial data, visitation times can be estimated using static code metrics like LOC and complexity. When the historical visitation times of similar users are available, *T:UserBased-CollaborativeFiltering* (Section 3.4) can be used.

P:Timeboxing: comprehension is usually time-bound, so a subset of priority ordered POIs that can likely be visited in the given timebox is selected, and may be reordered to accommodate POI locality.

P:CodeTrails: the recommendation service agent provides code trails (in a format such as XML) consisting of a navigation and visitation order for the POIs while considering locality. A mapping of the traveling salesman problem and related traveling salesman planning (TSP) algorithms (Lawler et al., 1985) are applied to these granules (the POIs) and the associated knowledge distance between them. While the recommended path may not necessarily be optimal, it provides an efficient path nonetheless through the knowledge landscape (source code). Two modes are supported: *initial* mode generates a trail from scratch, while *adapt* mode dynamically reoptimizes it based on the actually visited POIs and session time left. Visited POIs (including deviations) are detected via events and automatically removed from the adapted trail. Via events, the POI visitation history is tracked and can be replayed later.

P:UserProfile: a user's individual knowledge level (e.g., familiar vs. unfamiliar) and competency level (junior vs. senior) are taken into consideration.

3.3 Solution Architecture

The conceptual architecture (Figure 1) consists of four modules: *Cognitive Learning*, *Knowledge Processing*, a *Database Repository*, and *Integration*.

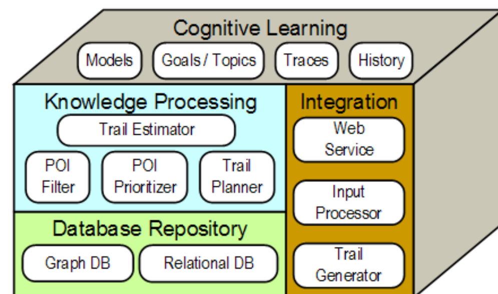

Figure 1: C-TRAIL conceptual architecture.

The *Cognitive Learning* module supports various program code learning *Models*, *Goals*, *Topics*, execution *Traces*, and visitation *History*. The

Knowledge Processing module includes the components *POI Prioritizer* for ranking POIs, a *POI Filter* that filters based on visitations or topics, a *Trail Estimator* for visitation times, and a *Trail Planner*. The *Database Repository* utilizes appropriate database types to retain metadata, knowledge, or data. The *Integration* module includes a REST *Web Service* API (application programming interface) for development tool integration, an *Input Processor* to process inputs, transformations, and events (such as a POI visit) including analysis and tracing inputs, and a *Trail Generator* for generating a planned trail into a desired format.

3.4 Solution Techniques

The solution incorporates these techniques (T:):

T:MethodRank: absent other indicators, it is assumed that frequently utilized domain methods should be comprehended before others. Thus to prioritize POIs, a variation of the PageRank algorithm (Page et al., 1999) we call *MethodRank* is applied. Here, webpages correspond to methods (code locations) and hyperlinks to invocations. Methods with more static references (invocations) in the code set are ranked higher. While runtime invocations (such as loops) are not considered, it can indicate methods with broader relative utilization and thus likely of greater comprehension *relevance*.

T:POIDistance: granules (functions/methods) are assumed to be grouped in classes/files and packages/directories are ordered hierarchically. (Sub)package depth is mapped to a vertical axis, while classes group methods horizontally. Loosely analogous to geographical distance, a distance *POIdist* between any two POIs (3) A and B is determined by the vertical *vdist* (1) and horizontal *hdist* (2) distance.

$$vdist = |\,\text{depth}(A) - \text{depth}(B)\,| \tag{1}$$

$$hdist = \begin{cases} 0 \text{ if class}(A) = \text{class}(B) \\ 1 \text{ otherwise} \end{cases} \tag{2}$$

$$POIdist = vdist + hdist \tag{3}$$

For example, the *POIdist* between methods in the same class is thus 0, between classes in the same package 1, etc. Although a higher cluster may represent a greater abstraction (e.g., only interfaces) and not necessarily be cognitively distant, any clusters between them should still be cognitively "closer". For instance, while the Java programming language has no concept of subpackages (each package is a separate entity), we assume a convention with additional dots implying further depth, and initial matching names implying a common cluster up to the first mismatch.

T:HamiltonianCycle: For *P:CodeTrails* it is assumed that proper modularity and hierarchy are followed, implying a greater *POIdist* is equivalent to a larger mental jump. To reduce the cognitive burden, the shortest trail is sought that provides a POI visitation order such that each POI is visited exactly once (except the start is also the end, i.e. a Hamiltonian cycle). This calculation problem is a special case of the well-known TSP, so a constraint-satisfaction solver can be utilized to calculate this.

T:UserBasedCollaborativeFiltering: predicting POI visitation duration is difficult. Thus, analogous to allotting sufficient visitation time for geographical tourist destinations, user-based collaborative filtering is used to detect profile similarities and then recommend visitation times based on similar users.

3.5 Data Processing

Figure 2 shows the various data processing stages.

Figure 2: C-TRAIL data processing stages.

1) *Input Processing*: source code is imported and analyzed, resulting in a list of all POIs as FQNs. Each POI's cluster depth is determined by counting FQN subpackage depth, used to apply the *T:POIDistance* calculation. For *M:DynamicPath*, dynamic runtime traces are also required as input.

2) *POI Filtering*: Any topic filters are applied, and POIs visited by this user (either in the expected order or out-of-order) are removed from the set.

3) *POI Prioritization*: An ordered list of POIs is created. *T:MethodRank* is utilized when appropriate.

4) *POI Time Planning*: actual per-user POI visitation times are tracked via events and stored. *T:UserBasedCollaborativeFiltering* is used to estimate visitation times. For a cold start, it can be estimated based on factors such as the user's profile, a configured default time per LOC, cyclomatic complexity. Starting at the top, the POI prioritized list is trimmed at the point where the cumulative time exceeds the timeboxed session.

5) *POI Locality Planning*: POIs are reordered using a *T:HamiltonianCycle* planner accounting for locality (nearby POIs visited before distant POIs).

6) *Trail Generation*: the trail in the recommended POI visitation order is then generated.

4 REALIZATION

The C-TRAIL approach is independent of any realization or specific programming language. To determine its viability, a Java prototype was created that generates XML code trails for Java codebases, with a Neo4J graph and a H2 relational database.

4.1 Input Processing

T:MethodRank requires a static analysis of methods (as FQNs) and their target invocation relationships and counts. For Java code, jQAssistant 1.0.0 and the GraphAware Neo4j NodeRank plugin (with a damping factor of 0.85) were used. A Cypher query selects all method FQNs and their invoked method FQNs and the result exported to a CSV file, which is imported to the C-TRAIL Neo4J server. A separate simplified graph is created of FQN(Method)->INVOKES->FQN(TargetMethod) relationships. Then, NodeRanks for every node (i.e., Method) are calculated based on the number of invocations, with the NodeRank stored in each node's property (see Figure 3). Via the Neo4J REST API, the result is retrieved in JSON (see Figure 4), parsed, converted to FQNs, and stored in an H2 MethodRank table. For simplification, the prototype only considers class methods and ignores method overloading.

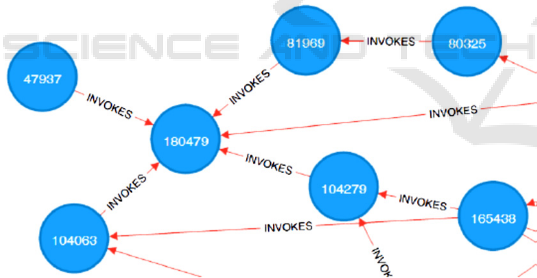

Figure 3: Partial Neo4J graph for T:MethodRank.


```
{
    'id': 41,
    'labels': [
        'STATICANALYSIS',
        'STATIC_de.ba.Class:getString(java.lang.String)
    ],
    'MethodRank': 504220
},
```


Figure 4: Example JSON NodeRank request result.

For *M:DynamicPath*, we wrote a parser for Intrace-Agent trace output (timestamp, ThreadID, method FQN, and entering/leaving file line), following the ThreadID via a hashtable containing stacks of method FQN strings (multi-threading was excluded). Each graph (calling method-INVOKES->called method) is stored in Neo4J. H2 tables store per method FQN:

TraceSessionsBreadth (b): number of traces using this method (maximum of one per trace). Thus, methods used in multiple scenarios (involved in more trace sessions) have a higher value.

TraceHits (h): frequency a method was invoked.

TraceOrderPerSession: a sequential numbering.

4.2 C-TRAIL Service

The C-TRAIL service is accessed via REST (Representational State Transfer) implemented with Restlet. It runs locally or in the cloud and be readily integrated in IDEs or other tooling. To support tool integration, XML was chosen as the trail format (see Figure 5). All learning models were supported.


```
<session guid="XXXXXXXX-5D29-4C1F-85FE-XXXXXXXXXXXX">
  <timebox>2016-01-20T11:00:00</timebox>
  <trailhistory>
    <step visited="2016-01-20T10:00:00">x.Class1:foo</step>
    <step visited="2016-01-20T10:03:22">x.Class2:bar</step>
    ...
  </trailhistory>
  <trail>
    <step visit="2016-01-20T10:06:00">y.Class1:foo</step>
    <step visit="2016-01-20T10:09:00">y.Class1:bar</step>
    ...
  </trail>
</session>
```


Figure 5: Trail output snippet (simplified for space).

For *P:UserProfile,* UUIDs (universally unique identifiers) differentiate users. Based on their profile, in the absence of similar user historical visitation times, configurable multiplication factors (default = .5) are used to adjust visitation times for senior or familiar users (a senior familiar user being four times faster). All user sessions are tracked with GUIDs (globally unique identifiers) and time-boxed (a configurable setting, default is midnight) for *P:Timeboxing*.

P:POIRanking weights various parameters according to the selected learning model. Also, *WeightingMode* provides maximum flexibility (via w_{total}) using configurable parameter weighting inputs (w_n) (4). E.g., this supports deviations from strict trace session order to weight frequently *(f)* or broadly *(b)* executed or visited (*hits*) methods more.

$$w_{total} = w_1 \cdot hits + w_2 \cdot b + w_3 \cdot h + \ \ldots \qquad (4)$$

The prioritized POI list is trimmed to where the accumulated expected visitation times exceed the remaining session time. Actual POI visitation time is tracked via navigation events received from clients and stored in H2 with FQN, UUID, and visitation

time (in seconds). Visited POIs (expected or not) are filtered and removed from the replanned trail.

T:UserBasedCollaborativeFiltering for *POI Time Planning* was realized by integrating Apache Mahout (Schelter & Owen, 2012), mapping the typical triple (user, item, value) to (user, method, time). CustomFileModelMahout was used to convert UUIDs to a compatible Apache Mahout format. *P:POIFiltering* was realized via a user-defined function in H2 that filters using regular expressions.

Applying *T:HamiltonianCycle* on this set, the C-TRAIL *Trail Planner* integrates OptaPlanner and utilizes its constraint solver for TSP using *T:POIDistance*, specifically optimizing the trail with regard to *P:POILocality* and *P:CodeTrails*. For responsiveness, solving was limited to 5 seconds to permit finding a (not necessarily optimal) solution, dependent on POI set size and hardware capability.

4.3 C-TRAIL Client

To demonstrate integratability, an Eclipse IDE plugin was developed (see Figure 6), providing a dropdown learning model choice. Selecting a POI in SERE opens the Eclipse source view to that method. Eclipse navigation events are monitored and sent via REST to the C-TRAIL service to reoptimize and regenerate the client trail based on actual POI visits.

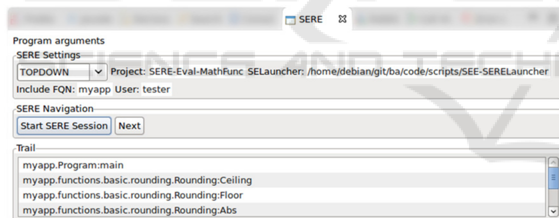

Figure 6: Our Eclipse plugin SERE (a C-TRAIL client).

5 EVALUATION

As the prototype realization showed C-TRAIL's feasibility, the evaluation focused on a practical demonstration of key C-TRAIL conceptual features, performance measurements, and a limited empirical study with learning models and structural analysis. A project codebase consisting of 15 POIs was used (Figure 7a). Package names were abbreviated.

The prototype was run in a VirtualBox VM (Debian 8 x86, one CPU, 1.7GB RAM) on a W10 x64 T9400 CPU@2.5GHz 4GB RAM notebook (viewable as an intentionally non-ideal developer deployment vs. a decent cloud deployment).

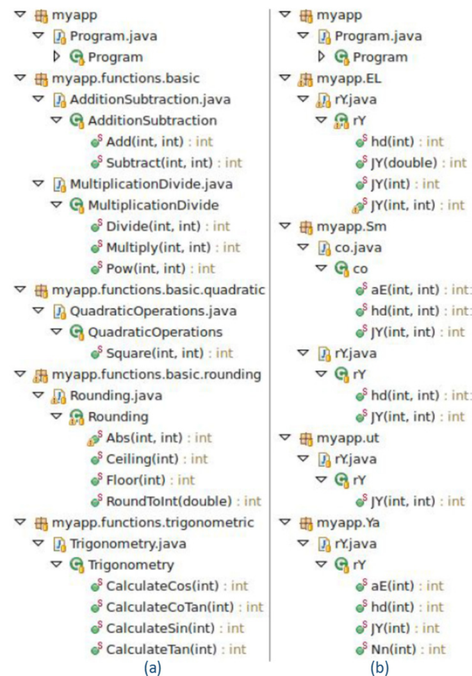

Figure 7: a) Original and b) obfuscated project structure.

5.1 Conceptual Features

To demonstrate key conceptual features including *P:POIRanking*, *P:POILocality*, *P:CodeTrails*, *P:POI*, *T:HamiltonianCycle*, *T:MethodRank*, and *T:POIDistance*, the code trail in Figure 8 was generated based on Figure 7a code. As the session timebox was larger than the cumulative estimated visitation (46 minutes and 4 seconds), no POI was time-filtered. To demonstrate *P:POIVisitTime* and *P:Timeboxing*, the session timebox was then limited to 30 minutes. Lower ranked POIs (having fewer invocations) were removed from the set and the code trail replanned while preserving locality (Figure 9). For *P:UserProfile*, it was verified that changing the profile changed the expected visitation times accordingly (not shown due to space constraints).

```
myapp.Program:main
myapp.f.basic.rounding.Rounding:Ceiling
myapp.f.basic.rounding.Rounding:Floor
myapp.f.basic.rounding.Rounding:Abs
myapp.f.basic.rounding.Rounding:RoundToInt
myapp.f.basic.quadratic.QuadraticOps:Square
myapp.f.trigonometric.Trigonometry:CalculateTan
myapp.f.trigonometric.Trigonometry:CalculateCos
myapp.f.trigonometric.Trigonometry:CalculateSin
myapp.f.trigonometric.Trigonometry:CalculateCoTan
myapp.f.basic.MultiplicationDivide:Divide
myapp.f.basic.MultiplicationDivide:Multiply
myapp.f.basic.MultiplicationDivide:Pow
myapp.f.basic.AdditionSubtraction:Add
myapp.f.basic.AdditionSubtraction:Subtract
```

Figure 8: Code trail without limiting session timebox.

```
myapp.Program:main
myapp.f.trigonometric.Trigonometry:CalculateCos
myapp.f.trigonometric.Trigonometry:CalculateSin
myapp.f.trigonometric.Trigonometry:CalculateCoTan
myapp.f.basic.MultiplicationDivide:Divide
myapp.f.basic.MultiplicationDivide:Multiply
myapp.f.basic.MultiplicationDivide:Pow
myapp.f.basic.AdditionSubtraction:Add
myapp.f.basic.AdditionSubtraction:Subtract
```

Figure 9: Code trail with limited session timebox.

5.2 Performance Measurements

Average total latency (of 10 measurements) for trail generation with 13 POIs and 1504 method visit entries in Apache Mahout was 5.73 seconds. Decomposing this latency, approximately 300 ms was attributed to Apache Mahout, 100 ms to POI prioritization, and less than 100ms for network overhead. OptaPlanner TSP optimization (capped at 5 seconds) was the primary latency factor.

5.3 Empirical Study

"A person understands a program when he or she is able to explain the program, its structure, its behavior, its effects on its operation context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program" (Biggerstaff et al., 1993). The human factor plays a significant role in assessing program comprehension, making it difficult to compare results and benefits. In the absence of readily available program comprehension assessment frameworks, obfuscation was selected as a primary technique in the empirical assessment method.

Obfuscation transforms or destroys the original software structure and semantics and negatively affects the efficiency of attacks while reducing the gap between a novice and skilled attacker (Ceccato et al, 2009). Although obfuscation is usually used to avoid code from being understood by an attacker, we apply it here to explicitly remove the semantic and structural points of reference in order to determine if C-TRAIL actually supports the navigation of unfamiliar code (few semantic or domain anchors).

Using the convenience sampling technique, two programmers having Eclipse, Java, and UML skills were selected. Eclipse and C-TRAIL were used.

5.3.1 Learning Models

M:Top-Down: one programmer was tasked with drawing the project structure of non-obfuscated code without class revisitations (to avoid time-consuming mental sorting techniques or determining POI relations), but could take notes. Without C-TRAIL, it took 13 minutes to produce Figure 10, and with C-TRAIL it took 10 minutes to produce Figure 11 (the circle is due to the trail ending at the starting POI), a 23 % improvement.

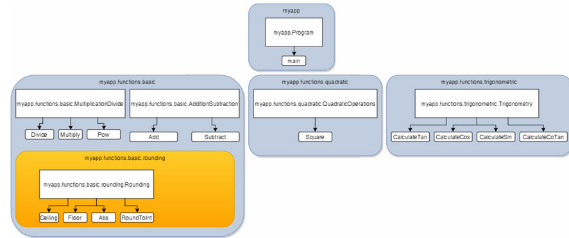

Figure 10: Transposed user diagram without C-TRAIL.

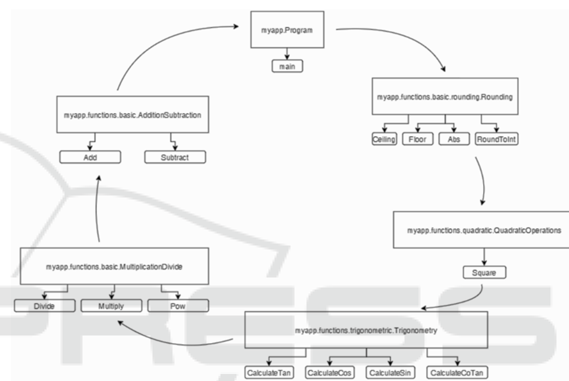

Figure 11: Transposed user diagram using C-TRAIL.

M:Bottom-Up: results similar to *M:Top-Down.*

M:DynamicPath: given only the source code without debugging tools, a programmer inspected the code and reconstructed how the application executes based on the correct ordering of the first ten steps of method execution. It took 3.5 minutes without errors for non-obfuscated code, and 5.8 minutes (66% longer) using obfuscated code. After inputting traces and utilizing a *M:DynamicPath* trail generated by C-TRAIL, the trail distilled the answer.

M:Topics/Goal (utilizing *P:POIFiltering*) and *M:Exploratory* were verified with manual testing.

The learning models support incorporated in C-TRAIL appears promising for improving code navigation and comprehension efficiency.

5.3.2 Structural Analysis

Code identifiers (as in Figure 12) were obfuscated with ProGuard utilizing random dictionaries containing strings of two-character length generated by Random.org. Obfuscated .class files were

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

decompiled to source code files with Java's decompiler (see Figure 7b and Figure 13).

Two programmers were then asked to sketch models first without C-TRAIL and then with (each time with newly obfuscated code and prior notes/diagrams removed). Diagrams with C-TRAIL showed significantly less errors. Table 1 shows the structural analysis time needed for obfuscated code.

```
package myapp.func.trigonometric;
...
public class Trigonometry {
...
  public static int calculateTan (int x) {
    int numeratorSin = calculateSin(x);
    int denominatorCos = calculateCos(x);
    return multiplicationDivide.divide(
      numeratorSin , denominatorCos);
  }
}
```

Figure 12: Snippet of original project source code.

```
package myapp.Ya;
...
public class rY {
  ...
  public static int aE(int paramInt) {
    int i = JY(paramInt);
    int j = hd(paramInt);
    return co.hd(i, j);
  }
}
```

Figure 13: Obfuscated project source code snippet.

Table 1: Structural analysis efficiency for obfuscated code (in minutes).

	User1	User2
Without C-TRAIL	15.5	11.3
With C-TRAIL	8.3	7.5
Improvement	46%	34%

Some observations: Towards an explanation for the efficiency benefit of using C-TRAIL in the obfuscated code setting, the programmers reported that without C-TRAIL support they intuitively compared concept locations mentally (analogous to Bubblesort) to try to somehow determine a concept grouping and relations. We also observed that the diagrams created by users using C-TRAIL code trail guidance exhibited locality order (which C-TRAIL preserves) and had fewer errors, even in the absence of domain or meaningful semantic anchors.

With regard to structural analysis, the limited empirical study indicates that C-TRAIL can potentially improve the effectiveness and efficiency in navigating unfamiliar program code. Future work includes a large-scale empirical study with a diverse pool of subjects utilizing various learning models and project sizes.

6 CONCLUSIONS

The program comprehension situation is exacerbated by a combination of a spiraling amount of program code, ongoing demand for corrective and adaptive maintenance and evolution of legacy or existing codebases, high industry and open source developer turnover rates, and a limited trained human resource pool with the associated high labor costs and limited time. Within the program comprehension sphere, this paper focused on program code concept location familiarity and structural understanding.

This paper contributed a practical solution approach called C-TRAIL that automates the recommendation of code visitation trails given only code or optionally code execution traces. By amalgamating cognitive learning model styles with the traveling salesman, granular computing, and collaborative filtering paradigms, it automates the planning of relevant visitation trails for an available session timebox. Its guidance can help the user not miss essential areas while avoiding dead ends, reorientation waste, and irrelevant areas. As a web service, it can easily be integrated in various tools and IDEs while leveraging available user profile data and collaborative filtering to estimate visitation times. It requires no prior project history inputs and does not depend on a visualization paradigm. The evaluation demonstrated its viability with a prototype of various conceptual features, including integration with an IDE. The limited empirical study showed improved navigation efficiency results when comprehending non-obfuscated and obfuscated code, as well as structural analysis efficiency and effectiveness improvements.

Future work includes a comprehensive empirical study with a diverse population and code repositories, an empirical comparison to other comprehension approaches, support for additional learning models and programming languages, a study of C-TRAIL in an industrial setting, and optimizations to address the TSP solver latency.

ACKNOWLEDGEMENTS

The author thanks Claudius Eisele for his assistance with the realization, evaluation, and diagrams.

REFERENCES

Bargiela, A. and Pedrycz, W., 2012. *Granular computing: an introduction* (Vol. 717). Springer Science & Business Media.

Biggerstaff, T.J., Mitbander, B.G. and Webster, D., 1993. The concept assignment problem in program understanding. In *Proc. 15th Int. Conf. on Software Engineering* (pp. 482-498). IEEE CS Press.

Booch, G., 2005. *The complexity of programming models*. Keynote talk at AOSD 2005, Chicago, IL, March 14-18, 2005.

Ceccato, M., Penta, M.D., Nagra, J., Falcarin, P., Ricca, F., Torchiano, M. and Tonella, P., 2009. The effectiveness of source code obfuscation: an experimental assessment. In *IEEE 17th Int. Conf. on Program Comprehension* (pp. 178-187). IEEE.

Cornelissen, B., Zaidman, A., Van Deursen, A., Moonen, L. and Koschke, R., 2009. A systematic survey of program comprehension through dynamic analysis. *Softw. Eng., IEEE Trans. on*, 35(5), pp.684-702.

Čubranić, D., Murphy, G.C., Singer, J. and Booth, K.S., 2005. Hipikat: A project memory for software development. *Software Engineering, IEEE Transactions on*, 31(6), pp.446-465.

Deshpande, A. and Riehle, D.. 2008. The total growth of open source. In *Proc. 4th Conf. Open Source Systems (OSS 2008)*. Vol. 275, pp. 197–209. Springer Verlag.

Goldman, M. and Miller, R.C., 2009. Codetrail: Connecting source code and web resources. *Journal of Visual Languages & Computing*, 20(4), pp.223-235.

Guéhéneuc, Y.G., 2006. TAUPE: towards understanding program comprehension. In *Proc. 2006 Conf. Center Adv. Studies on Collab. Research* (p. 1). IBM Corp.

Jones, C., 2006. *The economics of software maintenance in the twenty first century*. Retrieved from: http://www.compaid.com/caiinternet/ezine/capersjones-maintenance.pdf. [4 Feb 2016].

Kersten, M. and Murphy, G.C., 2005. Mylar: a degree-of-interest model for IDEs. In *Proc. 4th Int. Conf. Aspect-oriented Softw. Development* (pp. 159-168). ACM.

Koenemann, J. and Robertson, S.P., 1991. Expert problem solving strategies for program comprehension. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 125-130). ACM.

Lakhotia, A., 1993. Understanding someone else's code: analysis of experiences. *Journal of Systems and Software*, 23(3), pp.269-275.

Lawler, E.L., Lenstra, J.K., Kan, A.H.G.R. and Shmoys, D.B., 1985. *The traveling salesman problem: a guided tour of combinatorial optimization*. Wiley, New York.

Letovsky, S., 1987. Cognitive processes in program comprehension. *Journal of Systems and software*, 7(4), pp. 325-339.

Metz, C., 2015. *Google Is 2 Billion Lines of Code—And It's All in One Place*. Retrieved from: http://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/. [4 Feb 2016].

Minelli, R., Mocci, A. and Lanza, M., 2015. I know what you did last summer: an investigation of how developers spend their time. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension* (pp. 25-35). IEEE Press.

Mitchell, R.L., 2009. Y2K: The good, the bad and the crazy. *ComputerWorld* (December 2009).

Novak, J.D., 1998. *Learning, creating, and using knowledge*. Lawrence Erlbaum Assoc., Mahwah, NJ.

Oberhauser, R., 2016. ReSCU: A Trail Recommender Approach to Support Program Code Understanding. In *Proc. 8th Int. Conf. on Information, Process, and Knowledge Manage. (*pp. 112-118*)*. IARIA XPS Press.

Pacione, M.J., Roper, M. and Wood, M., 2004. A novel software visualisation model to support software comprehension. In *Reverse Engineering, 2004. Proc.. 11th Working Conference on* (pp. 70-79). IEEE.

Page, L., Brin, S., Motwani, R. and Winograd, T., 1999. *The PageRank citation ranking: bringing order to the web*. Technical Report. Stanford InfoLab.

PayScale. *Full List of Most and Least Loyal Employees*. Retrieved from: http://www.payscale.com/data-packages/employee-loyalty/full-list. [17 Feb 2016].

Pennington, N., 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, 19(3), pp.295-341.

Qian, Y., Liang, J., Dang, C., Wang, F. and Xu, W., 2007. Knowledge distance in information systems. *J. of Systems Science and Systems Eng.*, 16(4), pp.434-449.

Rajlich, V. and Wilde, N., 2002. The Role of Concepts in Program Comprehension. In *Proc. 10th IEEE Int. Workshop on Program Comprehension*, pp. 271-278.

Robillard, M.P. and Murphy, G.C., 2003. Automatically inferring concern code from program investigation activities. In *Automated Software Engineering, 2003. Proc.. 18th IEEE Int. Conf. on* (pp. 225-234). IEEE.

Robillard, M.P., 2008. Topology analysis of software dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(4), p.18.

Robillard, M.P., Maalej, W., Walker, R.J. and Zimmermann, T. eds., 2014. *Recommendation systems in software engineering*. Berlin: Springer.

Schelter, S. and Owen, S., 2012. Collaborative filtering with apache mahout. *Proc. of ACM RecSys Challenge.*

Schmidt, B., 2015. Retrieved from: http://benschmidt.org/Degrees/. [4 Feb 2016].

Singer, J., Elves, R. and Storey, M.A., 2005. Navtracks: Supporting navigation in software. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on* (pp. 173-175). IEEE.

Soloway, E., Adelson, B. and Ehrlich, K., 1988. Knowledge and processes in the comprehension of computer programs. In *The Nature of Expertise*, A. Lawrence Erlbaum Associates, pp. 129-152.

Yin, M., Li, B. and Tao, C., 2010. Using cognitive easiness metric for program comprehension. In *2nd Int. Conf. on Softw. Eng. and Data Mining* (pp. 134-139). IEEE.