ReSCU: A Trail Recommender Approach to Support Program Code Understanding

Roy Oberhauser
Computer Science Dept.
Aalen University
Aalen, Germany
email: roy.oberhauser@hs-aalen.de

Abstract —**Society is faced with an ever-increasing volume of computer program code that must be developed and maintained, exacerbated by a limited pool of trained human resources. Thus, effective and efficient automated tutor systems or recommenders for program comprehension are imperative. This paper introduces the Recommendation Service for Code Understanding (ReSCU), an approach that utilizes program code as a knowledgebase and automatically recommends a code trail to support effective and efficient human program code comprehension. Initial evaluation results with a prototype and an empirical study with obfuscated program code demonstrates its viability.**

Keywords-recommendation systems; intelligent tutoring systems; knowledge-based systems; program code comprehension; software engineering.

I. INTRODUCTION

The growing utilization of software throughout industry and society entails ever-increasing volumes of (legacy) program code and associated maintenance activity. While the total lines of program code worldwide is unknown, the Year 2000 (Y2K) crisis [1] with global costs of $375-750 billion gave us an indicator of the scale and importance of program comprehension, while a study of 5000 active open source software projects shows code size doubling on average every 14 months [2]. Moreover, the available pool of programmers to develop and maintain code remains limited and is not growing correspondingly. For instance, US bachelor degrees in Computer Science in 2011 were roughly equivalent to that seen in 1986 both in total number (~42,000) and as a percentage of 23 year olds (~1%) [3]. This is exacerbated by high employee turnover rates in the software industry.

Thus, there is resulting pressure on programmers to rapidly come up to speed on existing code or comprehend and maintain legacy code (a type of knowledge) in a cost-effective manner. It thus becomes imperative that programmers be supported with automated tutors and recommenders that efficiently and effectively support program code comprehension. In this space, recommendation systems for software engineering provide information items estimated to be valuable for a software engineering task in a given context [4].

This paper introduces a solution in this space called Recommendation Service for Code Understanding (ReSCU), a knowledge-centric recommendation service and planner for program code comprehension. ReSCU can be viewed as an intelligent tutor system, applying a practical form of granular computing [5] and concepts like knowledge distance. In support of human knowledge comprehension, it automatically recommends knowledge navigation as a Hamiltonian cycle [6] in an unfamiliar knowledge landscape of program code.

The paper is organized as follows: Section II discusses related work. Section III describes the solution concept and then the prototype realization. In Section V, the evaluation is described, which is followed by the conclusion.

II. RELATED WORK

An overview of recommendation systems in software engineering is provided by [4]. In the Eclipse IDE, NavTracks [7] recommends files related to the currently selected files based on their previous navigation patterns. Mylar [8] utilizes a degree-of-interest model in Eclipse to filter our irrelevant files from the File Explorer and other views. The interest value of a selected or edited program element increases, while those of others decrease, whereby the relationship between elements is not considered. In support of developers with maintenance tasks in unfamiliar projects, Hipikat [9] recommends software artifacts relevant to a context based on the source code, email discussions, bug reports, change history, and documentation. The eRose plugin for Eclipse mines past changes in a version control system repository to suggest what is likely also related to this change based on historical similarity [10]. To improve navigation efficiency and enhance comprehension, the FEAT tool uses concern graphs either explicitly created by a programmer [11] or automatically inferred [12] based on navigation pathways utilizing a stochastic model, whereby a programmer confirms or rejects them for the concern graph. With the Eclipse plugin Suade [13], a developer drags-and-drops 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. To support the usage of complex APIs in Eclipse, the Prospector system [14] recommends relevant code snippets by utilizing a search engine in combination with Eclipse Content Assist. Strathcona [15] analyzes structural facts of an incomplete code selection and utilizes heuristic matches to determine the most similar example. The Eclipse plugin FrUiT [16] supports example framework usage via association rule mining of applications that utilize a specific framework.

In contrast, various facets differentiate the ReSCU approach, including independence from any visualization paradigm, generating ordered code trails without necessitating an explicit context or prior history, and that it

 ISBN: 978-1-61208-472-5

requires no human expert intervention or confirmation. Furthermore, the approach is unique in applying a conceptual mapping of geographical points of interest (POI) and the traveling salesman problem/planning (TSP) to source code and the generation of code trail planning. The foregoing tools and approaches enhance program comprehension for certain kinds of developer tasks and intentions and can be viewed as complementary.

III. SOLUTION

The ReSCU solution approach focuses on supporting the learning, understanding, and navigation of unfamiliar program source code by programmers in an automated, systematic way, without requiring additional knowledge, historical information, or human expert assistance.

A. Principles

The solution concept includes these principles (P:):

- *P:POI*: program source code locations are identified and viewed as Points-of-Interest (POI) (or knowledge entities), analogous to geographical locations in navigational systems. Each POI is identified by a unique name, such as a fully qualified name (FQN) in the Java programming language consisting of the concatenation of a package name, class name, colon, and method name. A POI can be viewed as a granule or information entity of interest in a knowledge "landscape".
- *P:POIRanking*: To determine the importance of a POI (or knowledge granule) for human comprehension, they are ranked relative to each other. The algorithm *MethodRank* described below exemplifies such a ranking that fulfills this principle.
- *P:POILocality*: POI locality, which can conceptually viewed as knowledge closeness from the perspective of knowledge distance [17], is taken into consideration. This is intended to address the cognitive burden of context switches to a human when viewing program source code, by ordering POIs such that the number of unnecessary switches in a POI visitation order is reduced. The POI Distance calculation described later is an example for applying this principle.
- *P:Timeboxing*: Human comprehension and learning is assumed to be time-limited in the form of a session. Thus, the visitation time for POIs is estimated, and only the subset of priority ordered POIs that can be feasibly visited in the given timebox is selected. This subset will then be reordered to consider locality.
- *P:CodeTrails*: the recommendation service provides code trails as output with a navigation and visitation order recommendation for the POIs, whereby POI locality is taken into account. A mapping of the TSP and related planning algorithms [18] are applied to these granules (the POIs) and the associated knowledge distance between them. While the path suggest may not necessarily be the most optimal path, it provides an efficient path nonetheless through the knowledge landscape (source code).

In ReSCU, POI visitation planning via the generated code trails focuses on invocation relationships rather than class relationships. Not following class relationships can be viewed as supported by an empirical eye-tracking study finding that "software engineers do not seem to follow binary class relationships, such as inheritance and composition" [19].

B. Features

Besides the aforementioned principles, the solution includes the following additional capabilities:

- *User profiles*: user's knowledge level (e.g., familiar vs. unfamiliar) and competency level (junior vs. senior) are taken into consideration.
- *Trail (Re-)planning*: Two modes are supported: *initial trail* mode that generates a trail from scratch, and *refactor trail* mode that dynamically incorporates user actions and re-optimizes the trail based on the visited POI and the session time left. Visited POIs (including deviations) are detected via events and automatically removed from the next suggested trail.
- *Easily integratable:* A REST-based service interface provides distributed local and remote access to the recommender service from various software development tool and integrated development environments (IDEs).

C. Conceptual Architecture

The conceptual architecture is shown in Figure 1 consists of three primary modules: *Database Repository*, *Knowledge Processing*, and *Integration*.

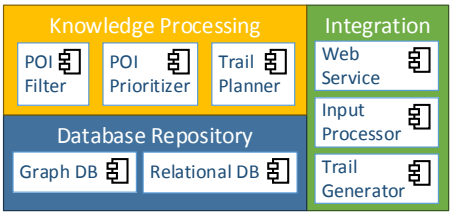

Figure 1. ReSCU solution architecture.

The *Database Repository* module logically groups various databases to retain metadata and knowledge in forms such as a graph database for modeling the source code as a graph of nodes with properties, and a relational/NoSQL database for dealing with non-graph-related knowledge related to source code.

The *Knowledge Processing* module includes components such as a POI Prioritizer for ranking POIs and a Trail Planner for planning the POI visitation time and order.

The *Integration* module includes a Web Service API (application programming interface) for supporting integration with other tools, an input processor to deal with tool events (such as a POI visit) and importing and transforming code information from analysis tools, and a

 ISBN: 978-1-61208-472-5

trail generator for generating or transforming a planned trail into a desired format.

D. MethodRank Calculation

With regard to *P:POIRanking*, it is assumed that in general, given no other knowledge source besides the source code and assuming limited learning time, it is more essential for the user to become familiar with the methods of a project that are used frequently throughout the code, rather than ones that are only sparsely utilized. Thus, a variation of the PageRank [20] algorithm call *MethodRank* is used to prioritize the POIs, whereby instead of webpages we map methods and instead of hyperlinks we map invocations. Thus, those methods that have the most references (invocations) in the code set are ranked the highest. While this does not consider runtime invocations (such as loops), it can be an indicator for a method with broader relative utilization and thus likely of greater interest for comprehension.

E. POI Distance Calculation

To address *P:POILocality*, an underlying assumption is that (sub)packages map vertically to (sub)layers and classes serve as a type of horizontal grouping of methods. Thus, the distance between any two POIs (given in (3)) A and B (analogous to geographical distance) is determined by their *vertical* (1) and *horizontal* (2) distance.

$$VerticalDistance = |\, layer(A) - layer(B) \,| \quad (1)$$

$$HorizontalDistance = \begin{cases} 0 & if\ class(A) = class(B) \\ 1 & otherwise \end{cases} \quad (2)$$

$$POIDistance = VerticalDistance + HorizontalDistance \quad (3)$$

For instance, the POIDistance between methods in the same class is 0, between classes in the same package 1, etc. Depending on the implementation, a higher layer may only represent a greater abstraction (e.g., only interfaces) and not necessarily be that far in cognitive "distance". Nevertheless, any sublayers between them should still be cognitively "closer".

F. Hamiltonian POI Visitation Trail

Assuming the principles of proper modularity and hierarchy are applied in a given project, a greater distance between POIs is equivalent to a larger mental jump. Thus, to reduce mental effort, once the distance for all pairs has been calculated, we desire the overall shortest trail that provides the visitation order for all POIs such that each POI is visited exactly once except that the starting point is also the end point, i.e. a Hamiltonian cycle. The calculation problem is equivalent to the well-known TSP.

G. Knowledge Processing

ReSCU knowledge processing stages are shown in Figure 2 and described below.

Figure 2. ReSCU knowledge processing stages.

1) Input Processing: the source code as text files are imported and analyzed. A list of all the POIs in the project as FQNs is determined. The layer of each POI is determined by counting the subpackage depth of its FQN. If the project actually utilizes a layer structure is irrelevant here. This is then used to apply the aforementioned POI distance calculation.

2) POI Filtering: POIs already visited by this user (either in the expected order or out of order) are filtered from the set for the initial planning or replanning.

3) POI Prioritization: the aforementioned MethodRank calculation is used to create an ordered list of POIs.

4) POI Time Planning: the actual POI visitation time is stored per user. Given no prior actual POI visitation time, a default visitation time can be estimated based on a user's profile utilizing a basis time per line of code in seconds, and factors correlated with the size and complexity of the current POI method, the knowledge level (stranger or familiar), and the competency level (junior or senior). Based on the limited session time available, the set of POIs the POI Time Planner component limits the set to an ordered list by priority that is cut off at the point that the cumlative time exceeds the timeboxed session. This reduces the size of the FQN set for locality planning and traversal.

5) POI Locality Planning: from the resulting set, the POIs are then ordered using a planner for a Hamiltonian cycle and a TSP path that takes locality into account, such that those nearby are visited first before jumping to POIs at a further distance.

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

IV. REALIZATION

To support validation of the solution concept and architecture, a prototype was realized in Java. It currently analyzes and generates code trails for Java program code. For simplification, only class methods are considered and method overloading is ignored (a single FQN is used for methods of the same name in trails).

As a Representational State Transfer (REST) service, the *Web Service* component was realized with Restlet and can be run locally, on the team's server, or the cloud in order to easily integrate with various integrated development environments or software engineering environments. The *Database Repository* used H2 as a relational and Neo4J as a graph database. To support flexible integration, the output trail format is XML.

The actual POI visitation time is tracked via navigation events received via the web service, with the table METHODRATING_TIMEONMETHOD storing MethodID, UserID, and visitation time (in seconds). POIs that were

 ISBN: 978-1-61208-472-5

already visited (expected or not) are then filtered and removed from the replanned trail.

MethodRank requires a data structure with methods (as FQNs) and their target invocation relationships and counts. For this, static code analysis of a project's methods and invoke relationships is performed using jQAssistant 1.0.0 and the GraphAware Neo4j NodeRank plugin [21]. A Cypher query selects all Method FQNs and their invoked Method FQNs and the result is exported to a CSV file. A separate simplified graph is then created by importing the CSV file into the Static Analysis Program with FQN(Method)->INVOKES->FQN(TargetMethod) relationships in the Neo4J server. GraphAware NodeRank then provides NodeRanks (i.e. MethodRanks) for every node (Method) for the number of invocations with the NodeRank stored in each node's property (Figure 3 shows a partial graph in Neo4J). The result is retrieved via the Neo4J REST API in JSON (example shown in Figure 4). The JSON was parsed, converted to FQNs, and placed in the H2 MethodRank table.

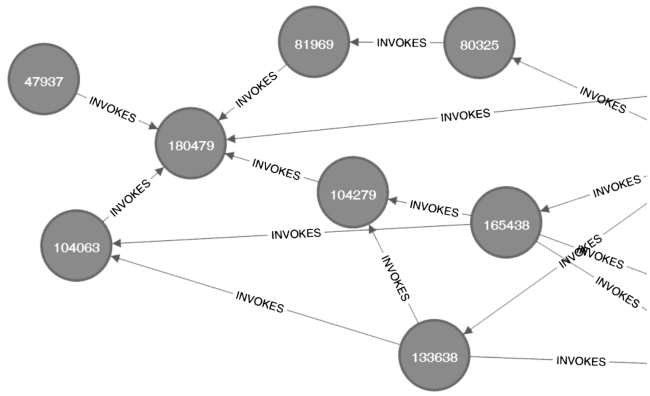

Figure 3. Example partial MethodRank graph in Neo4J.

```
[
  {
    "id": 41,
    "labels": ["STATICANALYSIS",
      "STATIC_de.ba.Class:getString(java.lang.String)"],
    "MethodRank": 504220
  },
  {
    "id": 90,
    "labels": ["STATICANALYSIS",
      "STATIC_de.ba.GlobalSettings:getInstance()"],
    "MethodRank": 335443
  },
  {
    "id": 1801,
    "labels": ["STATICANALYSIS",
      "STATIC_de.ba.package1.Helpers:someHelp()"],
    "MethodRank": 156736
  }
]
```

Figure 4. Example NodeRank request result in JSON.

Users are differentiated by a user ID. The visitation time is adjusted by a factor (default = .5) was used to halve the estimated time if it is a senior engineer, and a factor (.5) also if the user is already familiar with the code. All user sessions are time-boxed (default setting is termination at midnight, but any end time can be set). Once the prioritized POI list is calculated, POIs are selected in priority order to be included in the trail until the accumulated expected visitation times exceed remaining session time. The Hamiltonian path calculation is then applied on this subset.

To order the POI trail according to POI locality, the *Trail Planner* component integrated OptaPlanner, specifically optimizing the trail with regard to the TSP. For sufficient IDE interaction responsiveness during trail generation, the OptaPlanner solving time was explicitly limited to a maximum of 5 seconds to likely provide sufficient time for at least a solution to be found (depending on the project size, session time, and computation hardware) but not necessarily an optimum (absolute shortest path).

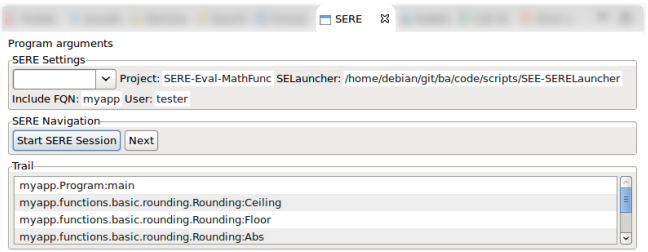

Figure 5. Eclipse client plugin that utilizes the ReSCU service.

To demonstrate REST-based integration of ReSCU with an IDE tool, an Eclipse IDE client (SERE) was developed, shown in Figure 5. The upper part shows the current project, the middle part is used for starting and navigating a session, and the bottom displays the upcoming trail locations (methods). Double-clicking causes the method to be shown in the Eclipse source view.

V. EVALUATION

The focus of our initial evaluation was to a) validate that the solution principles, conceptual architecture, and processing work in harmony when applied to program code as knowledge. Having converted code into a knowledge representation of granules with properties and relationships, determine if it, as a tutor, automatically generates a realistic knowledge-based navigation recommendation for a time-boxed session, and b) empirically validate the effectiveness and efficiency of the automatically generated code trails (i.e., as an automated tutor) in navigating and understanding unfamiliar program code (i.e., unfamiliar presented knowledge). For that, obfuscation was utilized to limit any intuitive mental model creation or semantic ordering so that ReSCU's effectiveness and efficiency for knowledge navigation could be assessed.

The prototype ran in a VirtualBox (Debian 8 x86, single CPU, 1.7GB RAM) VM running on a Windows 10 x64 host with a T9400 CPU@2.5GHz and 4GB RAM. A project consisting of 15 POIs was used as shown in Figure 6a.

 ISBN: 978-1-61208-472-5

A. Code Trail Generation Validation

For the original code (the source for the structure in Figure 6a), a code trail was generated as shown in Figure 7 with a session timebox much larger than the cumulative estimated visitation time for the entire trail (46 minutes and 4 seconds). Thus, no POI was time-filtered.

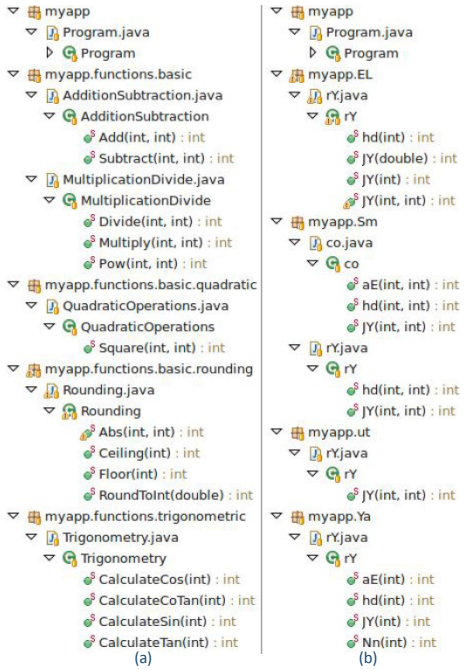

Figure 6. Project structure a) original b) obfuscated.

```
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 7. Code trail generated 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 8. Code trail generated with limited session timebox.

When the session timebox was limited to 30 minutes, lower ranked POIs with fewer invocations were removed from the set and the code trail replanned preserving locality as exhibited in Figure 8.

B. Empirical Structural Code Analysis Study

Obfuscation transforms or destroys the original software structure and semantics and negatively impacts the efficiency of attacks while reducing the gap between a novice and skilled attacker [22]. 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 how well ReSCU supports a programmer navigating unfamiliar code.

Code identifiers (as in Figure 9) were obfuscated with ProGuard utilizing random dictionaries containing strings of two character length generated by Random.org. Obfuscated .class files were decompiled to source code files with Java's decompiler (as in Figure 10).

Using the convenience sampling technique, two users experienced with Java and the Eclipse IDE were asked to sketch a model of the program code using only the classic Eclipse IDE without ReSCU and then, after a new obfuscation, with ReSCU.

```
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 9. Example original project source code snippet.

```
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 10. Example obfuscated project source code snippet.

User1 took 15:30 and User2 11:20 minutes to produce the diagrams transposed in Figure 11a and Figure 11b respectively (italic names were added afterwards to show mappings). A number of structural errors exist in the diagrams.

Repeating it with a fresh obfuscation and with ReSCU, User1 needed 8:20 and User2 7:30 minutes to produce the diagrams transposed in Figure 12a and Figure 12b respectively (italic names were added afterwards to show mappings).

We observed that the diagrams created by users using ReSCU code trail guidance exhibited an order based on locality (which ReSCU preserves) and had fewer errors. This limited empirical study showed improved effectiveness and efficiency in helping navigate unfamiliar program code. Future work will study a larger pool of subjects and projects.

 ISBN: 978-1-61208-472-5

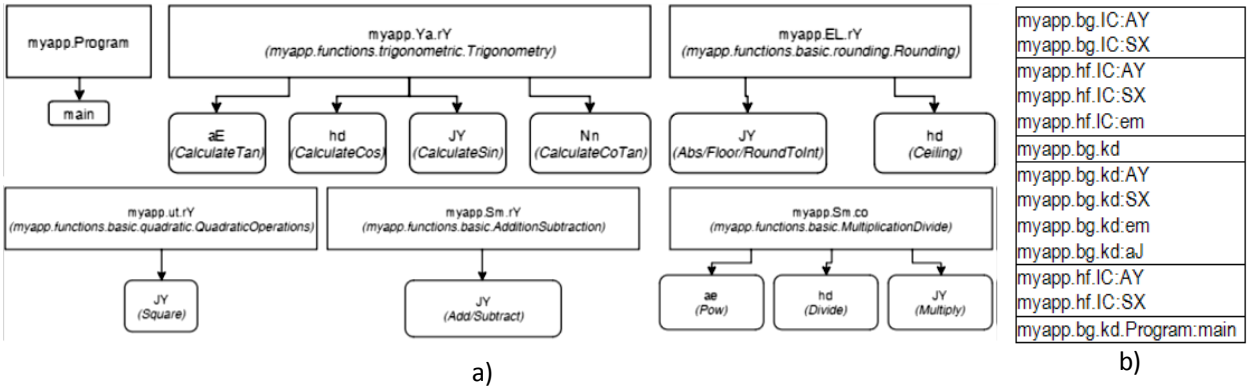

Figure 11. Transposed structure created without ReSCU by a) User1 b) User2.

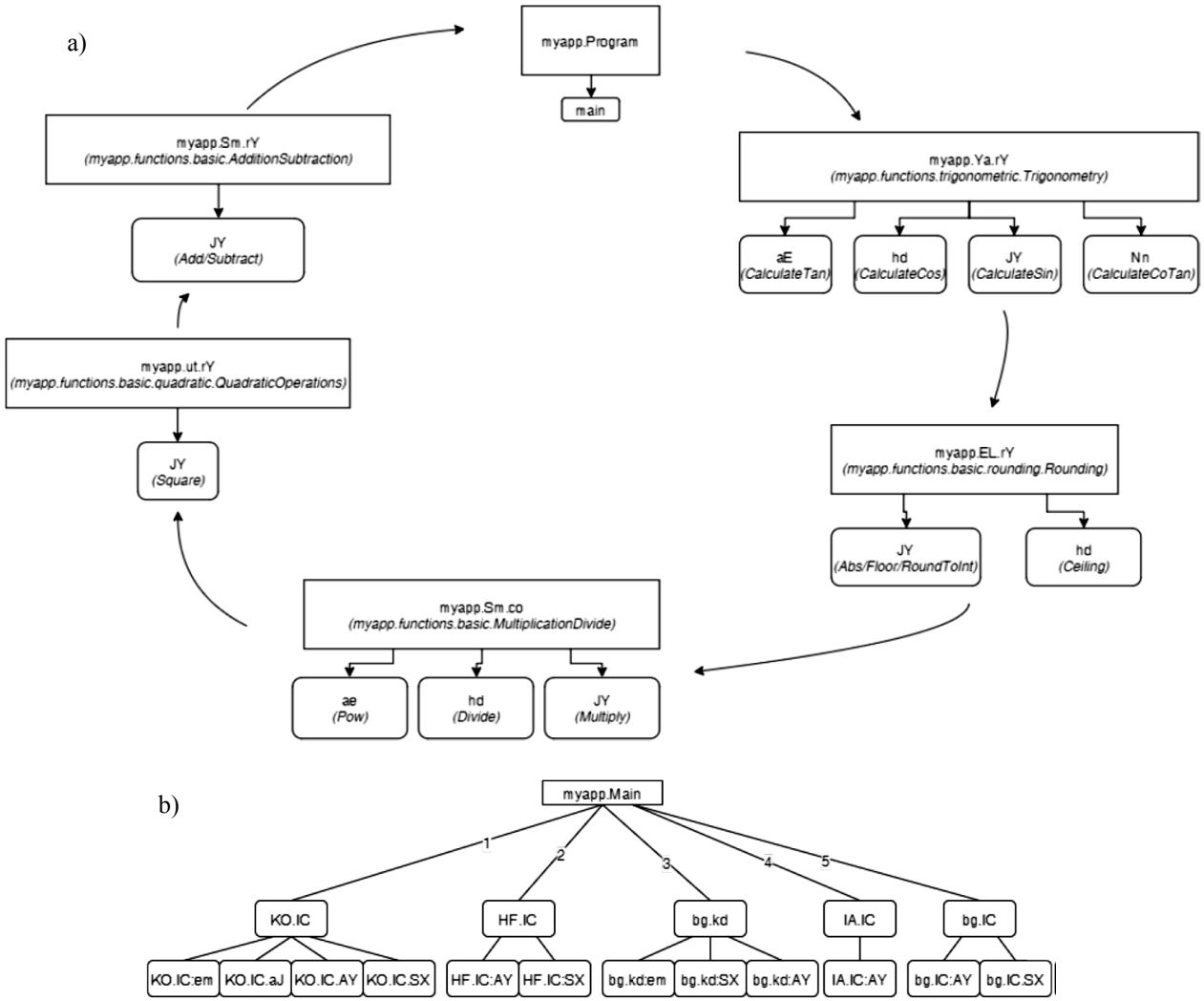

Figure 12. Transposed structure created when using ReSCU by a) User1 b) User2.

VI. CONCLUSION AND FUTURE WORK

As an automated tutor and recommender system in the program code comprehension space, ReSCU applies a conceptual mapping of geographical POIs to code locations, considers the locality or knowledge closeness of such granules, and applies TSP to an unfamiliar knowledge landscape consisting of program code. It incorporates MethodRanking as a variant of PageRanking and granular distance in the form of POI locality. Furthermore, it recommends a knowledge navigation order by generating a code trail as a Hamiltonian cycle. The evaluation based on a prototype and limited empirical study applied to obfuscated code indicated effectiveness and efficiency benefits for the ReSCU solution approach.

Future work includes a comprehensive empirical study, utilization in larger scale code projects, support for additional programming languages, and the integration with visualization paradigms. Application of the elaborated ReSCU solution principles to other domains beyond software engineering could provide beneficial knowledge navigation guidance and recommendations in form of a trail for other unfamiliar knowledge landscapes.

ACKNOWLEDGMENT

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

REFERENCES

[1] L. Kappelman, "Some strategic Y2K blessings," Software, IEEE, 17(2), 2000, pp. 42-46.

[2] Deshpande and D. Riehle. “The total growth of open source”. In: IFIP International Federation for Information Processing. Vol. 275. 2008, pp. 197–209

[3] Schmidt, http://benschmidt.org/Degrees/ 2016.01.26

[4] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, Recommendation Systems in Software Engineering. Springer, 2014.

[5] Bargiela and W. Pedrycz, Granular computing: an introduction. Springer Science & Business Media, vol. 717, 2012.

[6] M. S. Rahman and M. Kaykobad, "On Hamiltonian cycles and Hamiltonian paths," Information Processing Letters, 94(1), 2005, pp. 37-41.

[7] J. Singer, R. Elves, and M.-A. Storey, "NavTracks: Supporting Navigation in Software Maintenance," Proc. Int'l Conf. on Software Maintenance, 2005, pp. 325–334.

[8] M. Kersten and G. Murphy, "Mylar: A degree-of-interest model for IDEs," Proc. 4th international conf. on aspect-oriented software development, ACM, 2005, pp. 159-168.

[9] D. Cubranic G.C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A project memory for software development," Software Eng., IEEE Trans. on, 31(6), 2005, pp. 446-465.

[10] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," Software Eng., IEEE Trans. on, 31(6), 2005, pp. 429-445.

[11] M. P. Robillard and G. Murphy, "FEAT: A tool for locating, describing, and analyzing concerns in source code," Proc. 25th Int'l Conf. on Software Eng., IEEE, 2003, pp. 822–823.

[12] M. P. Robillard and G. Murphy, "Automatically Inferring Concern Code from Program Investigation Activities," Proc. 18th Int'l Conf. Autom. SW Eng., IEEE, 2003, pp. 225-234.

[13] M. P. Robillard, “Topology Analysis of Software Dependencies,” ACM Trans. Software Eng. and Methodology, vol. 17, no. 4, article no. 18, 2008.

[14] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman, "Mining Jungloids: Helping to Navigate the API Jungle," Proceedings of PLDI, Chicago, IL, 2005, pp. 48–61.

[15] R. Holmes, R. J. Walker, and G. C. Murphy, "Approximate structural context matching: An approach to recommend relevant examples," IEEE Transactions on Software Engineering 32(12), 2006, pp. 952–970.

[16] M. Bruch, T. Schaefer, and M. Mezini, "Fruit: IDE support for framework understanding," Proc. 2006 OOPSLA workshop on eclipse technology eXchange, eclipse ’06, ACM, 2006, pp. 55–59.

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

[18] E. L. Lawler, The traveling salesman problem: a guided tour of combinatorial optimization. Wiley, 1985.

[19] Y. G. Guéhéneuc, "TAUPE: towards understanding program comprehension," Proc. 2006 conf. Center for Adv. Studies on Collaborative research (CASCON '06) IBM Corp., 2006.

[20] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: bringing order to the Web," In:World Wide Web Internet And Web Information Systems 54.1999-66, 1998, pp. 1–17.

[21] https://github.com/graphaware/neo4j-noderank [2016.03.18]

[22] M. Ceccato et al. “The effectiveness of source code obfuscation: An experimental assessment,” In: IEEE Int'l Conference on Program Comprehension, 2009, pp. 178–187.

 ISBN: 978-1-61208-472-5