


Efficient Structural Analysis of Source Code for Large Scale Applications in Education

1st Adrian Koegl 

Columbia University

New York, USA

adrian.koegl@columbia.edu

2nd Peter Hubwieser

Technical University of Munich

Munich, Germany

peter.hubwieser@tum.de

3rd Mike Talbot

Technical University of Munich

Munich, Germany

mike.talbot@tum.de

4th Johannes Krugel 

Leibniz University Hannover

Hannover, Germany

krugel@dei.uni-hannover.de

5th Michael Striewe 

University of Duisburg-Essen

Essen, Germany

michael.striewe@s3.uni-due.de

6th Michael Goedicke

University of Duisburg-Essen

Essen, Germany

michael.goedicke@s3.uni-due.de

Abstract—Automated Assessment Systems (AAS) are increasingly used in large computer science lectures to evaluate student solutions to programming assignments. The AAS normally carries out static and dynamic analysis of the program code. In addition, simple forms of learning analytics can often be generated quite easily. However, structural analyses and comparison of solutions for larger sets of student programs are, in many cases, complicated and time-consuming.

In this article, we introduce a methodology with which thousands of programs can be analyzed in less than a second, for example, to search for the use of certain control structures or the application of recursion.

For this purpose, we have developed a software that creates a structural representation for each programming solution in the form of a *TGraph*, which is inserted into a graph database using *Neo4j*. On this database, we can search for structural features by queries in the language *Cypher*.

We have tested this methodology extensively for Java programs, measured its performance, and validated the results. Our software can also be applied to programs in other programming languages, such as *Scratch*. Additionally, we plan to make our software available to the community.

Index Terms—Automated assessment systems, learning analytics, introductory lectures, programming assignments, large-scale education, structural code analysis, source code representation, graph database

I. INTRODUCTION

Triggered by the enormous need for graduates and attractive career prospects, introductory computer science lectures have reached more than a thousand students at many universities. To limit the personnel expenses for programming exercises in such large lectures, more and more *Automated Assessment Systems* (AAS) are used to evaluate the solutions submitted by the students. Furthermore, AASs are of particular importance in cases where no individual human tutoring is available, e.g., in Massive Open Online Courses (MOOCs), to provide immediate individual feedback.

An AAS normally conducts static and dynamic tests to determine the accuracy and evaluate the quality of the programs

written by students. In addition, simpler forms of Learning Analytics are often provided to the instructor, e.g., statistics on the difficulty of tasks or the most common mistakes. However, faculty often want additional information on structural aspects of student solutions. For example, they may want to identify alternative solutions or know the number of students who completed a particular task using recursion. Unfortunately, such analyses quickly become so complex and costly that they can no longer be solved with conventional AASs.

As part of a larger research project on programming competencies [1], we faced precisely this problem. Our goal was, among others, to find alternative solutions as well as common errors and shortcomings in a large number of student solutions to five projects from a typical CS1 lecture (Computer Science 1). Over four fall terms, we have collected 85,762 solutions of programming tasks from 20,315 students, containing 228,888 Java program files in total.

The solutions were uploaded to our AAS [2] by the students, which runs static and dynamic tests for these programs and also provides the students with qualitative feedback on likely errors. For static analysis, the system represents the structure of the programs by *TGraphs*, see section II-B. The dynamic analysis examines the correctness of the programs using test classes.

Since our AAS is not specifically designed for this workflow (as most existing AAS), it cannot efficiently handle queries on the structure of a large number of solutions. Two of the reasons for this inefficiency are that the AAS is not optimized for graph traversal and has to regenerate the representation of all programs for each query.

In this paper, we present a methodology that enables us to perform such structural queries efficiently even on a very large number of programs. The basic idea is to store all these *TGraphs* in a graph database, on which we can perform very efficient queries on the structure of these graphs. To create this database, we have developed software that generates the *TGraphs* of a set of programs for a growing number of programming languages and enters them into a graph database

of a configurable format. However, the prerequisite is that a scheme for the generation of the TGraphs is available for the respective language, as for *Java* or *Scratch*. For the graph database, we have chosen the format of *Neo4j*, making use of the query language *Cypher*.

We have tested this methodology extensively and measured its performance, which we will explain below utilizing illustrative examples. The results of the queries were validated by manual review of random samples. If other instructors or researchers want to apply our methodology, we can provide our software.

II. BACKGROUND AND RELATED WORK

A. Assessment of Student Programs

Due to the high numbers of participants in introductory courses, many universities use automated systems (also called e-assessment systems) for formative or summative assessment. Those systems are also of particular importance in online courses such as MOOCs [3]–[5], where no individual human tutoring is available.

For the subject of programming exercises, a wide range of AASs exists that detect errors by executing test cases and analyzing source code [6], [7]. In general, AASs perform similarly well compared to human graders regarding scoring and the generation of feedback [8]. However, a lot of research is related to direct error feedback with ambivalent results: On one hand, fine-grained feedback and grading schemes proved to be effective [9], but on the other hand, detailed syntax error messages appear to be ineffectual [10]. There is also research on additional aspects like assessing code quality and style [11], [12] or estimating the difficulty of exercises by analyzing large amounts of solution data [13].

An extensive set of Java programs has been studied as part of the Blackbox project, which collects source code and various information about students’ interactions with the BlueJ environment [14], [15]. Programs in visual programming languages are also being studied on a large scale. For example, Aivaloglou and Hermans [16] analyzed a large set of projects from the Scratch repository to examine project complexity and detect code smells.

B. Structural Representation of Program Code

An AAS needs an appropriate representation of the code structure to analyze programs. The parser represents all required structural information on the program code internally by *abstract syntax trees* (AST), see, e.g., [17]. However, queries on program structures, for example, if a particular method is ever called in the program, are difficult and costly on ASTs [18]. To support such queries, structural information can be represented explicitly by adding edges to the AST [19]. The resulting structure is called *abstract syntax graph* (ASG), which in our context is generated by the graph transformation engine described in [20].

To represent ASGs formally, the flexible *TGraph* format is frequently used [21]. A TGraph is defined as an “ordered directed graph with typed and attributed vertices

and edges” [22]. A TGraph consists of nodes and edges, each of which has a label and properties. In our use case, the nodes represent the program’s syntax elements, and the edges describe their relationship. For example, node `v20` in Figure 1 represents the declaration of a method. A public modifier is defined by the node `v21`, connected with `v20` through the `MethodDeclarationModifiers` edge. The same is true for the return type and parameter with according edge label names. The subgraph after the `e24:MethodDeclarationBody` edge represents the method body.

Structural queries on TGraphs can be written in the *Graph Repository Query Language* (GReQL), which was developed as a declarative query language for graph structures [23].

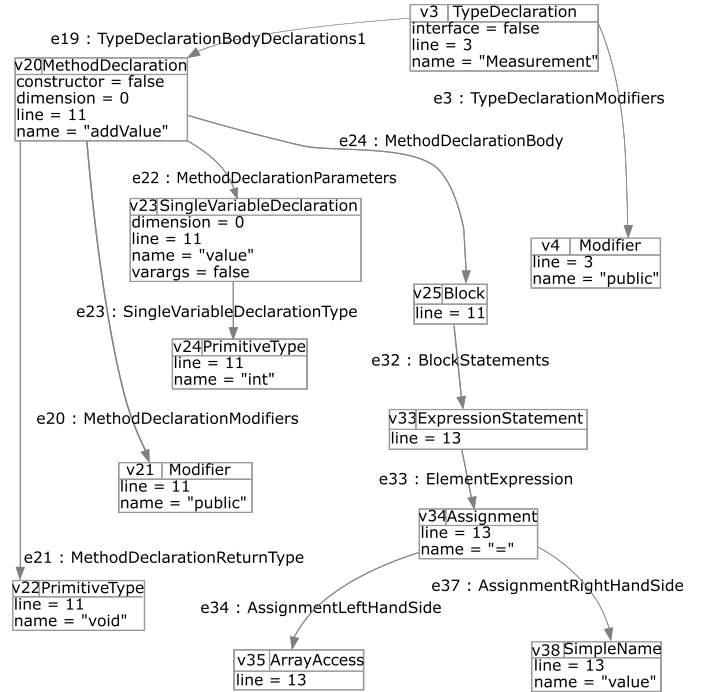


Fig. 1. Exemplary Snippet of a TGraph

C. Graph Databases

Contrary to conventional (relational) database systems, graph database (GDB) systems use graphs or graph-like structures as the basic data structure [24]. The nodes and edges of the graphs are represented as objects and relations between these objects. In so-called *native graph databases*, the edges are implemented by the *index-free adjacency* property [25], meaning that nodes are connected by direct referencing and not via an edge object. This property enables efficient traversal of graphs, while the runtime remains constant to the total size of the database.

The native GDB *Neo4j* is based on the *property graph model*. According to this model, each node and edge has a set of labels and properties, where each edge is directed [26]. However, *Neo4j* allows only one label per edge. In the context of *Neo4j*, labels categorize a set of nodes or edges, which

should reveal their role in the data structure [27, p.20]. While labels are single strings, a property can be viewed as a key-value pair. Here, the key represents a string, with the value containing any primitive data type or array [27, p.26].

Through the *Neo4j-Browser*, it is possible to visualize data and perform queries in the declarative query language *Cypher*¹, designed specifically for property graph models [28]. The central concept in Cypher queries is pattern matching. Patterns in Cypher are expressed in a visual form as “ASCII art”, such as `(a) - [r] -> (b)` [28]. The syntax structure also has similarities with SQL but has been improved to allow intuitive graph queries. For example, recursive structures can be queried in a simplified way [28].

III. METHODOLOGY

As part of a larger research project on programming competencies, we needed to perform detailed structural analyses on extensive sets of student programs, represented by Java files. Since we found that these would be far too time-consuming with the available AAS, we developed a particular novel representation of all these programs in a GDB and software that automatically realized this representation based on the TGraph format. In the end, we were able to perform all necessary evaluations very efficiently by running queries on the resulting GDB.

A. Efficiency Challenges

While trying to analyze and compare the structure of several thousand student programs, for example, to investigate the use of recursion, we found that our AAS could not perform such queries in tolerable time frames. Since it dynamically generates a TGraph for each program analyzed at each query, our first hypothesis was that this recurrent transformation of the source code was causing the main overhead. Indeed, the runtime of the programs’ transformation generates an overhead that increases linearly with the number n of program files analyzed.

However, further performance analysis showed that the time cost of TGraph traversal caused the main braking effect. To investigate this overhead, we measured the runtime of a simple query that checks for the presence of a global integer variable in all program files. The associated query looks for an edge in the TGraph that connects the global variable node to the class declaration node in each program. The runtime of this query increased faster than linear with n , see Figure 2. Our AAS performs simple data queries of graph elements quite efficiently, but takes much more time to traverse the TGraphs.

B. Modelling the Graph Database

As explained above, it was impossible to perform the intended research on our AAS due to its poor graph traversal performance. Using a suitable GDB system for this purpose provided a possible approach to significantly improving efficiency. Based on a market analysis of available systems

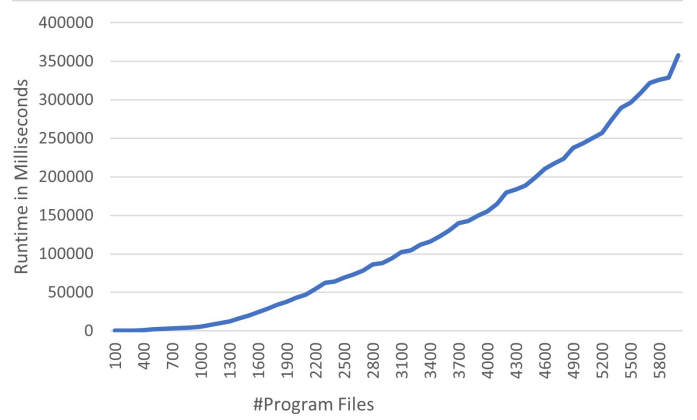


Fig. 2. Runtime of an exemplary query on our AAS

capable of representing the TGraph structure, we decided to use Neo4j.

We planned to store the TGraphs of all collected student programs in the GDB in order to perform structural queries on arbitrary subsets of them. All the programs’ metadata also had to be represented in the GDB to achieve this. Modeling the metadata appropriately and automating this process are the main challenges to optimize efficient graph traversal within such an application. In our context, students had solved each programming task of the projects by writing one or more Java class definitions and uploading them to the AAS. They were allowed to submit as many versions as they wanted for each task, with each submission containing one or more program files. As a result, we had to map the following metadata for each class file: Term ID, Project ID, Student ID (hashed), and Submission ID.

Our tests revealed that queries can be executed most efficiently when the metadata is represented as a tree of nodes. One of the reasons for this is that such a metadata tree connects all inserted TGraphs into one total graph. The resulting metadata graph of this context, visualized in Figure 3, displays for every node its label and `name` property value, as well as the label for every edge. In general, the user gives each metadata layer a label describing it, e.g., `Term`. Each node is also assigned at least one property, such as the key `name` and the respective filename as its value. The filename, which is used to create those properties, may also be divided into several properties. Furthermore, the user of our software defines the label of the relations between each metadata layer, such as `IN_PROJECT`. The TGraphs of a submission are then associated with the respective leaf of this tree, in this context a `Submission` node.

Due to this structure, before being able to perform our intended operations on a certain TGraph, its nodes and edges need to be first identified in the total graph. For this purpose, Neo4j needs to traverse every possible path beginning with the submission leaf. We can help optimize this process by adding a label to every node of a TGraph, such as `TGraph`. This improves the identification of a TGraph by about 30%,

¹<https://neo4j.com/docs/cypher-manual/current/>

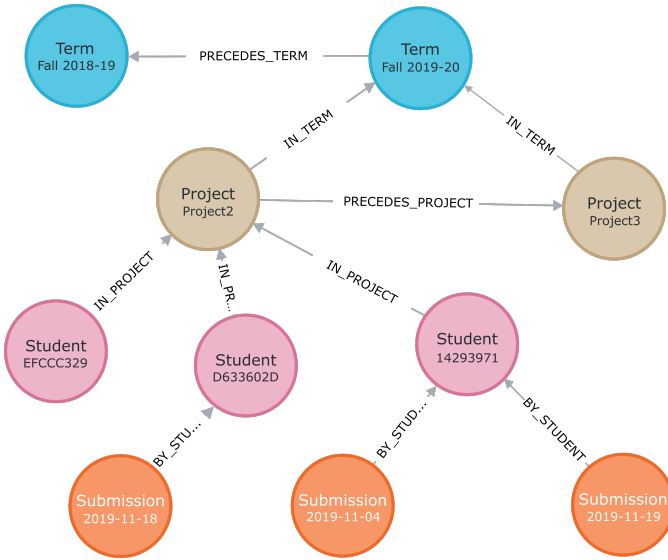


Fig. 3. Exemplary graph of metadata

according to our measurements. Figure 4 shows an exemplary TGraph visualized by Neo4j, corresponding to its original representation in Figure 1. This TGraph is connected to the Submission node with name property value 2019-11-04 of the metadata tree in Figure 3.

In our context, the 228,888 parsed program files result in a total graph with 61,534,620 nodes in the graph database. While the program files originally covered a volume of 1.08 GB in the file system, the corresponding Neo4j database requires 36.27 GB.

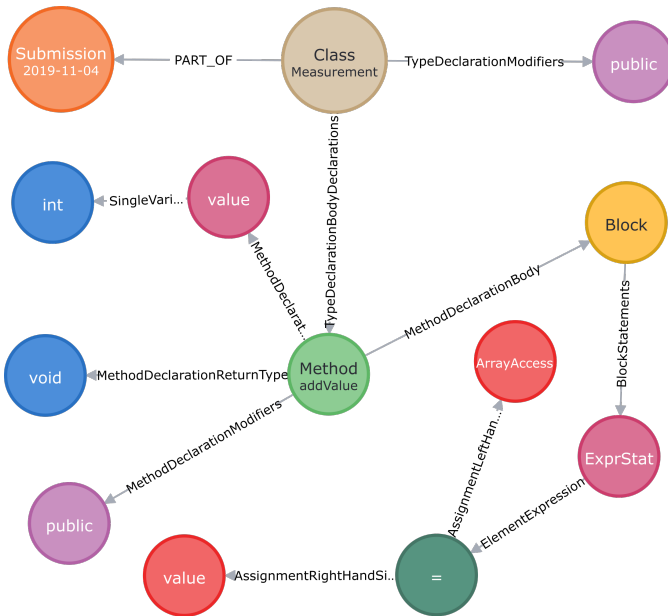


Fig. 4. Visualised TGraph in Neo4j-Browser

C. Cypher Queries

To match patterns in a set of TGraphs, the Cypher query must first identify all the required TGraphs. Suppose we want to analyze all submissions by the student with hash 1D024507 in Project 3 of the fall term 2020. Any query that explicitly analyzes these structures first specifies the semester, project, and student and then traverses to the submissions. From there on, the query recursively matches all nodes with the TGraph label attached to each submission node, as shown in Listing 1.

```
MATCH (:Semester {name:"Fall term 2020"})
  <-- (:Project {name: "Project 3"})
  <-- (:Student {name:"1D024507"})
  <-- (:Submission) <-- (:TypeDeclaration)
  <-- [*]->(x:TGraph)
```

Listing 1: Matching TGraphs with certain metadata

D. Automated Transformation of Programs

We developed a special *Automated Program Representation Software* (APRS), which automizes the process of transforming program files to the TGraph format and inserting them into a graph database. APRS was designed as flexibly as possible so that it can be used in a different context, not depending on any AAS. It can be employed without needing an additional tool or extensive installation and setup, and it solely relies on program files in a directory structure.

A configuration file allows users to define the layers and metadata format in their specific project. Most importantly, the following settings can be defined:

- Programming language of program files
- Directory structure describing the metadata
- Edges between layers of nodes
- Mapping of labels and properties
- Graph database type and connection

Currently available parsers that convert the programs into TGraphs are UML, Haskell, Scratch, and Java. Also, the performance of APRS was optimized by threading the bottleneck of writing to the database. Details of APRS and the validation of the resulting GBD are presented in the Bachelor Thesis [29].

E. Performance Analysis

Ultimately, the performance of queries on our GDB determines the efficiency of our methodology. To evaluate our results, we searched for all nodes of a particular label and the nodes of a certain TGraph. Then we combined these two queries to a structural analysis on TGraphs, followed by a data inquiry on all nodes.

It should be noted that the following runtimes refer to when the result is processed and available. The additional runtime for visualization depends on the number of returned and displayed elements – this aspect is not considered here.

For finding all nodes with a specific label, we retrieved all submission nodes with the query in Listing 2. Neo4j matched the 85,762 submission nodes in 3 ms.


```
MATCH (s:Submission) RETURN s
```

Listing 2: Querying all submission nodes

The most crucial key to efficiency is the rapid identification of a particular TGraph since our research aims to analyze large numbers of them. Matching a single TGraph was completed in 1 ms following the style in Listing 1.

Putting these queries together, the next type of query identifies structures through the labels and properties of nodes on TGraphs. As this is our research’s primary purpose, we provide two worst-case examples by matching patterns on **all** TGraphs.

```
MATCH (st:Student)<--(su:Submission)<--
(:TypeDeclaration)-->(:FieldDeclaration)
-->(:PrimitiveType {name:"int"})
RETURN st,su
```

Listing 3: Querying submissions with global integer variable

The first one corresponds to our AAS’s exemplary query, described in subsection III-A. We issued the equivalent query of returning all submissions containing a global integer variable, which is, in contrast, feasible on all TGraphs. This query, shown in Listing 3, took 701 ms to process the representations of all the 228,888 program files. In comparison, the corresponding query, executed by our AAS, took about 275 sec for only 2.5 % of the program files, see Figure 2. These results show that this typical query was executed about 15,700 times faster on the GDB, assuming that the AAS’s runtime increases linearly.

The second structural query searches in all TGraphs for a method that contains a call to an eponymous method. Finding all such methods requires a graph traversal in every TGraph to check for a `MethodInvocation` node. The corresponding query in Listing 4, which performs a pattern matching on all TGraphs, was completed in 1551 ms.

```
MATCH (s:Submission)<--(t:TypeDeclaration)
↪ -->(m:MethodDeclaration)
WITH m.name as method, s, m, t
MATCH (m)-[*]->(:TGraph:MethodInvocation)
↪ {name:method})
RETURN s,t
```

Listing 4: Pattern matching over all TGraphs

The most time-consuming of the tested query types should retrieve and then process a property of all nodes. As a costly example, we output the unique `name` property values of all nodes processed in 43,831 ms. To perform this operation, a query first collects the `name` property value of all nodes and subsequently evaluates the distinct occurrences. This query is particularly time-consuming because retrieving properties is much more expensive than searching for labels.

IV. APPLICATIONS

According to our measurements, our GDB performs structural queries on a large number of TGraphs very efficiently.

To elucidate two exemplary didactic use cases, the following examples show how to identify iterative vs. recursive solutions in student submissions and how to detect code redundancy.

The task studied in both applications included programming a `PhoneBook` class, which involves managing a phone book through a tree data structure. In the given `insertPerson` method, the students were asked to apply their newly acquired knowledge of recursion to insert a person object into the tree. In total, we had 6,116 submissions at our disposal.

A. Identifying Iterative Solutions

A possible problem in computer science education could be to find out how many of these solutions use an iterative approach. We are interested to know how many students solved the problem correctly but without implementing a recursive method. The constructed query checks for all correct methods if it contains a call to a method with the same name. It then returns all students and submissions not containing this structure. The corresponding query can be found in Listing 5.

```
MATCH (st:Student)<--(su:Submission)<--
↪ (t:TypeDeclaration{name:"PhoneBook"})-->
↪ (m:MethodDeclaration{name:"insertPerson",
↪ points:"1"})
WHERE NOT EXISTS {
    (m)-[*]->(:MethodInvocation
    ↪ {name:"insertPerson"})
}
RETURN st, su
```

Listing 5: Querying correct but iterative solutions

Neo4j returns 44 submissions created by six students who have solved the task iteratively but passed all the static and dynamic checks. This allows us to closely examine these solutions and, among others, improve our static code checks such that those submissions will not be graded as correct anymore. Without the possibility of large-scale structural analysis, it would have been unfeasible to find those edge cases.

B. Querying Redundancy

An exemplary application in research is the extraction of redundant code structures. Assuming that differently implemented methods each serve their purpose, one conceivable parameter for distinguishing the quality of code is redundancy.

The task `insertPerson` is reasonable for checking the applied concept of recursion for contained redundancy. To find different classes of redundancy, we first compared correct students’ solutions. When we found redundant recursive runs, we generalized the structure and constructed a query to return submissions containing the same redundancy type. Then, we inspected all the returned submissions and iteratively optimized the query to accurately return only the submissions containing this structure. Thereby, we were able to identify a few recurring structures representing correct but improvable solutions.

An undeniable redundancy class that we have identified is the further crawling of the phone book tree, although the element has already been inserted. A recurring structure

causing this superfluous tree traversal is displayed in Listing 6. To avoid unnecessary tree crawling after the element has been inserted, the if clause could, e.g., contain a return statement.

```
if (currentNode.getRightSuccessor() == null) {
    currentNode.setRightSuccessor(newPerson);
}
insertPerson(newPerson,
    ↪ currentNode.getRightSuccessor());
```

Listing 6: Excerpt of redundancy generating code

We have constructed a query that returns all submissions of students containing this structure. The query in Listing 7 first matches all correctly implemented submissions of the `insertPerson` method. The requirement for this task is that the actual recursive method is private, so this is taken into account in line 1 of the query. Subsequently, in line 2, it uses each of these `MethodDeclaration` nodes as a starting point to search for a call to an eponymous method. The query then checks whether this call is placed within the block of the if statement that conditions if a method invoked from the object `currentNode` equals null, see line 4. This also implies that the call is not placed in an else clause. Then, in line 5 to 7, it excludes all matches which contain a return statement in the previously matched if clause.

```
MATCH (t:TypeDeclaration
    ↪ {name:"PhoneBook"})-->
    ↪ (m:MethodDeclaration {name:"insertPerson",
    ↪ points:"1"})
    ↪ -[:MethodDeclarationModifiers]->
    ↪ (:Modifier {name:"private"})
MATCH (m)-[*]->(x:TGraph:Block)-->
    ↪ (:ExpressionStatement:TGraph)-->
    ↪ (:MethodInvocation {name:"insertPerson"})
WITH s,a,x
MATCH (x)-->(if:IfStatement:TGraph)-->
    ↪ (ie:InfixExpression:TGraph {name:"=="})-->
    ↪ (:NullLiteral:TGraph)
WHERE NOT EXISTS {
    (if)-->(:Block:TGraph)
    -->(:ReturnStatement)
}
MATCH (ie)-[*1..2]->(:SimpleName
    ↪ {name:"currentNode"})
RETURN s,a
```

Listing 7: Querying redundant tree crawling

This query is applicable in this context without further specifications, as the object `currentNode` is a method parameter that may not be altered by the student, as the task would otherwise be assessed as incorrect by the AAS.

The query returns 78 submissions implementing this particular type of redundancy. A cutout of the visualized return in Neo4j can be seen in Figure 5.

V. FUTURE WORK

A. Future applications

Now that the performance of the queries on our GDB has proven to be satisfactory, nothing stands in the way of a

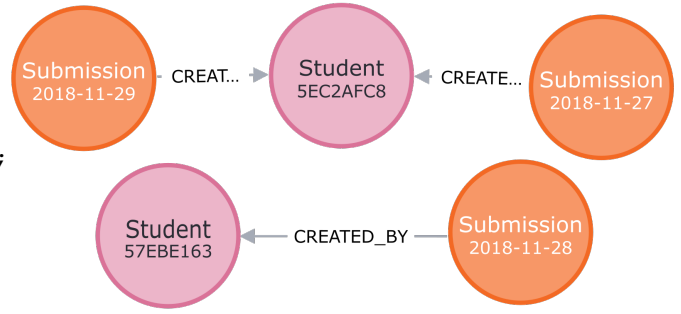


Fig. 5. Cutout of visualised query return of Listing 7

comprehensive application in the context of our and others' research projects. In particular, we will explore the deviations of the student solutions from the expert solution as well as structural differences and correspondences among them in the context of the project described in this paper.

Furthermore, we plan to use ARPS for analyzing the students' submissions of MOOCs. We developed a MOOC for the introduction of object-oriented programming called "LOOP" ("Learning object-oriented programming") [5]. In particular, we target all submissions of this course to investigate the students' approaches, identify common errors, and develop feedback strategies.

In addition, we intend to apply ARPS to structurally investigate the programs on Kings College's *Black Box* repository², which collects some millions of Java programs created with the BlueJ-IDE. The main difficulty here is that the original specification of these programs is unknown. The evaluation of a GDB could supply crucial information on that.

Another extensive collection of programs we are interested in is the *Scratch Repository*³. For this purpose, we have already developed a scheme for the transformation of Scratch projects into TGraphs and conducted performance analyses to evaluate large sets of such programs, see [30].

B. Extension of ARPS

Currently, ARPS is limited to the available parsers for TGraphs. Therefore, we now present two approaches that will be implemented in the future to remove this constraint.

First, we plan to develop a parser generator that transforms the source code of any language into TGraphs from a given formal grammar definition. This general-purpose parser will allow us to extend ARPS using TGraphs to virtually any programming language in the future.

Second, we will relax the dependency on TGraphs so that any other graph format, including an appropriate parser, can be used. This allows, for example, simply using the representation form of an AST. Accordingly, we will make the program available after publication to enable research on an arbitrary number of source code.

²<https://bluej.org/blackbox/>

³<https://scratch.mit.edu/explore/projects>

REFERENCES

- [1] J. Krugel, P. Hubwieser, M. Goedicke, M. Striwe, M. Talbot, C. Olbricht, M. Schypula, and S. Zettler, "Automated measurement of competencies and generation of feedback in object-oriented programming courses," in *2020 IEEE Global Engineering Education Conference (EDUCON)*, 2020, pp. 329–338.
- [2] M. Striwe, "An architecture for modular grading and feedback generation for complex exercises," *Science of Computer Programming*, vol. 129, pp. 35–47, 2016.
- [3] A. Bey, P. Jermann, and P. Dillenbourg, "A comparison between two automatic assessment approaches for programming: An empirical study on moocs," *Journal of Educational Technology & Society*, vol. 21, no. 2, pp. 259–272, 2018. [Online]. Available: <http://www.jstor.org/stable/26388406>
- [4] S. Krusche and A. Seitz, "Artemis: An automatic assessment management system for interactive learning," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 284–289. [Online]. Available: <https://doi.org/10.1145/3159450.3159602>
- [5] J. Krugel and P. Hubwieser, *Strictly Objects First: A Multipurpose Course on Computational Thinking*. Cham: Springer International Publishing, 2018, pp. 73–98. [Online]. Available: https://doi.org/10.1007/978-3-319-93566-9_5
- [6] D. M. Souza, K. R. Felizardo, and E. F. Barbosa, "A systematic literature review of assessment tools for programming assignments," in *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*. Dallas, TX, USA: IEEE, Apr. 2016, pp. 147–156.
- [7] H. Keuning, J. Jeuring, and B. Heeren, "A systematic literature review of automated feedback generation for programming exercises," *ACM Trans. Comput. Educ.*, vol. 19, no. 1, sep 2018. [Online]. Available: <https://doi.org/10.1145/3231711>
- [8] M. Gaudencio, A. Dantas, and D. D. Guerrero, "Can computers compare student code solutions as well as teachers?" in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 21–26. [Online]. Available: <https://doi.org/10.1145/2538862.2538973>
- [9] N. Falkner, R. Vivian, D. Piper, and K. Falkner, "Increasing the effectiveness of automated assessment by increasing marking granularity and feedback units," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 9–14. [Online]. Available: <https://doi.org/10.1145/2538862.2538896>
- [10] P. Denny, A. Luxton-Reilly, and D. Carpenter, "Enhancing Syntax Error Messages AGppears Ineffectual," in *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '14. New York, NY, USA: ACM, 2014, pp. 273–278.
- [11] K. Ala-Mutka, T. Uimonen, and H.-M. Jävinen, "Supporting students in C++ Programming Courses with Automatic Program Style Assessment," *Journal of Information Technology Education*, vol. 3, pp. 245–262, 2004.
- [12] E. Araujo, D. Serey, and J. Figueiredo, "Qualitative aspects of students' programs: Can we make them measurable?" in *2016 IEEE Frontiers in Education Conference (FIE)*, Oct. 2016, pp. 1–8.
- [13] O. Seppälä, P. Ihanntola, E. Isohanni, J. Sorva, and A. Vihavainen, "Do we know how difficult the rainfall problem is?" in *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, ser. Koli Calling '15. New York, NY, USA: Association for Computing Machinery, Nov. 2015, pp. 87–95.
- [14] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting, "Blackbox: a large scale repository of novice programmers' activity," in *Proceedings of the 45th ACM technical symposium on Computer science education*, *SIGCSE '14*. New York, New York, USA: ACM Press, 2014, pp. 223–228.
- [15] N. C. C. Brown, A. Altmir, S. Sentance, and M. Kölling, "Blackbox, five years on: An evaluation of a large-scale programming data collection project," in *Proceedings of the 2018 ACM Conference on International Computing Education Research*, *ICER '18*. New York, NY, USA: ACM, 08082018, pp. 196–204.
- [16] E. Aivaloglou and F. Hermans, "How kids code and how we know: An exploratory study on the scratch repository," in *Proceedings of the 2016 ACM Conference on International Computing Education Research*, *ICER '16*. New York N.Y.: ACM, 2016, pp. 53–61.
- [17] D. E. Knuth, "Semantics of context-free languages," *Mathematical Systems Theory*, vol. 2, no. 2, pp. 127–145, 1968.
- [18] Michael Striwe, Moritz Balz, and Michael Goedicke, "Enabling graph transformations on program code," in *Proceedings of the 4th International Workshop on Graph Based Tools*, Enschede, The Netherlands, 2010.
- [19] M. Striwe and M. Goedicke, "A review of static analysis approaches for programming exercises," in *Computer Assisted Assessment. Research into E-Assessment*, M. Kalz and E. Ras, Eds. Cham: Springer International Publishing, 2014, pp. 100–113.
- [20] Carsten Kollmann and Michael Goedicke, "A specification language for static analysis of student exercises," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. USA: IEEE Computer Society, 2008, pp. 355–358.
- [21] D. Bildhauer and J. Ebert, "Querying software abstraction graphs," in *Working Session on Query Technologies and Applications for Program Comprehension (QTAPC 2008)*, Amsterdam, Netherlands, 2008.
- [22] J. Ebert and A. Franzke, "A declarative approach to graph based modeling," in *Graph-Theoretic Concepts in Computer Science*, vol. 903. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 38–50.
- [23] M. Kamp, "Greql - eine anfragesprache für das gupro-repository," in *GUPRO - Generische Umgebung zum Programmverstehen*, Koblenz, 1998, pp. 173–202.
- [24] R. Angles and C. Gutierrez, "Survey of graph database models," *ACM Comput. Surv.*, vol. 40, no. 1, Feb. 2008. [Online]. Available: <https://doi.org/10.1145/1322432.1322433>
- [25] J. Pokorný, "Graph databases: Their power and limitations," in *Computer Information Systems and Industrial Management*, K. Saeed and W. Homenda, Eds. Cham: Springer International Publishing, 2015, pp. 58–69.
- [26] Renzo Angles, "The property graph database model," in *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management*, Cali, Colombia, 2018.
- [27] I. Robinson, J. Webber, and E. Eifrem, *Graph databases*, 2nd ed. Sebastopol, Cal.: O'Reilly Media, Inc., 2015.
- [28] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An evolving query language for property graphs," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1433–1445. [Online]. Available: <https://doi-org.eaccess.ub.tum.de/10.1145/3183713.3190657>
- [29] A. Kögl, "Design and setup of a graph database for t-graphs and structural analysis of selected examples," Bachelor's thesis, Technical University of Munich, Garching, 2020.
- [30] M. Talbot, K. Geldreich, J. Sommer, and P. Hubwieser, "Re-use of programming patterns or problem solving? representation of scratch programs by tgraphs to support static code analysis," in *Proceedings of the 15th Workshop on Primary and Secondary Computing Education*, ser. WiPSCE '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: <https://doi.org/10.1145/3421590.3421604>