Design and evaluation of a semantic indicator for automatically supporting programming learning

Julien Broisin
Université Toulouse 3 Paul Sabatier, IRIT
118 route de Narbonne
31062 Toulouse, France
julien.broisin@irit.fr

Clément Hérouard
Ecole Normale Supérieure
Campus de Ker lann, Avenue Robert Schuman
35170 Bruz, France
clement.herouard@ens-rennes.fr

ABSTRACT

How to support students in programming learning has been a great research challenge in the last years. To address this challenge, prior works have mainly focused on proposing solutions based on syntactic analysis to provide students with personalized feedback about their grammatical programming errors and misconceptions. However, syntactic analysis falls short on informing learners how they solve the programming problem, even if one key learning outcome of programming relates to the development of an individual's ability to solve a problem. In this article, we introduce an indicator to analyze beginners' code based on semantic proximity. This indicator adapts an edit distance algorithm (i.e., the Levenshtein distance) to express the proximity of the students' code with the expected solution provided by the teacher, in order to express the learners' capacity to solve the given problem. To process our indicator, we applied machine learning techniques to a dataset from an introductory programming course with a sample of 166 students. The first results are encouraging. On the one hand, the semantic indicator can be used to automatically classify source codes as semantically correct or incorrect in 58% of the cases. On the other hand, the indicator is correlated with teachers' summative evaluations of students' codes. Even if further investigations must be conducted to improve the indicator's accuracy, the results of this study make it possible to use our approach as the foundations for future research in semantic-based intelligent and awareness programming systems.

Keywords

Programming learning, Semantic analysis, Educational data mining, Edit distance, Levenshtein algorithm

1. INTRODUCTION

Computer literacy is currently booming. In Europe, particularly in Germany and the United Kingdom, profound educational transformations have been initiated since 2016 to promote digital learning in schools and prepare learners for the acquisition of 21st century skills, which include programming learning. In France, for example, an educational reform of High School curricula that will be operational next year offers a Digital and Computer Science option that includes more than 350 hours of programming learning. This interest in integration of programming learning skills early in the curriculum requires not only prepared teachers, but also technological solutions to support them and their students in their daily practices.

With this purpose, the Technology-Enhanced Learning research community has been interested in designing systems dedicated to support learning of programming, as evidenced by different efforts intended to analyze learners' behaviour [14, 12]. One of the most common approaches in prior works consists in designing systems that analyze learners' programming codes from a syntactical perspective. These systems make a syntactic evaluation of students' codes to detect grammatical errors and provide appropriate and constructive support to learners to avoid misconception errors [15]. However, delivering feedback about syntactical errors falls short on providing meaningful information about how students approach the programming problem. Yet, one of the key learning outcomes of programming relates to the development of an individual's ability to solve a problem [11]. One approach to achieve this goal is to design systems able to analyze source codes from a semantic perspective, i.e., able to show how the problem has been solved. Although there have been some initiatives approaching code evaluation from a semantic perspective [5], works on this line are still scarce and very few solutions have been proposed. Thus, more solutions based on semantic analysis are needed to better understand the potential of this approach in supporting learning of programming.

To advance on semantic analysis-based solutions, this article introduces the design of an indicator revealing the semantic proximity of two distinct source codes in order to express the correctness of a learner's code regarding a given problem, and tackles the following research questions:
Research question 1: How to design a semantic indicator revealing learner's ability to solve a problem?
Research question 2: Is the semantic quality of a learner's production correlated with his/her academic performance?

To answer the first question, we adopt an approach based on the comparison of abstract syntax trees. We adapt the edit distance established by Levenshtein, an algorithm that

Julien Broisin and Clément Hérouard "Design and evaluation of a semantic indicator for automatically supporting programming learning" In: *Proceedings of The 12th International Conference on Educational Data Mining (EDM 2019),* Collin F. Lynch, Agathe Merceron, Michel Desmarais, & Roger Nkambou (eds.) 2019, pp. 270 - 275

has proven effective in comparing and correcting strings of characters. Then, we propose a machine learning method to determine some of the factors required to process the semantic proximity using source codes produced by 166 students. For the second research question, and with the objective of evaluating the precision of the indicator in relation to human perception, we conduct a set of statistical analysis with source codes gathered from an authentic learning context. In particular, we analyze the correlation between the value of our indicator automatically processed, and the teachers' scores manually assigned to students' source codes.

In the following section we review what current approaches dedicated to automatic source code analysis are currently available in the literature, and highlight a lack of semantic-driven proposals. Then, in Section 3, we present how the semantic indicator was designed; as stated above, the indicator stands on the Levenshtein distance adapted to source codes comparison, and depends on a set of parameters leading to its calculation. Section 4 then introduces the machine learning method we used to assign values to these different parameters, whereas Section 5 describes the dataset used to actually determine the values of the parameters. Section 6 gives the results we obtained, and evaluates the quality of the indicator at two different levels: its capacity to act as a semantic classifier, and its correlation with human perception. Finally, we discuss the results of this study as well as the main conclusions of the work.

2. APPROACHES FOR AUTOMATIC SOURCE CODE ANALYSIS

In the context of programming learning, automatic source code analysis is used to achieve different objectives such as improving feedback provided to learners [7, 9], predicting their performance [1, 16], or monitoring their activities [3].

These automatic analyses are performed by compiling the code [6], by searching for typical errors within a source code [9], or by running unit tests provided by teachers [13]. These various works guide learners in the production of syntactically correct programs, but they do not allow the source code to be evaluated at a semantic level: syntactic evaluation is not sufficient to reflect the relevance of a learner's production regarding a problem given by the teacher.

In the meta-review proposed by Ihantola et al. [8] who studied no less than 118 research articles in the field of educational data mining for programming, the word *semantics* is missing from the paper. Also, a search for the terms "semantic analysis programming" in Google Scholar and Web of Science returns a large number of results, but no scientific articles really deal with semantic analysis of code.

The works we have identified that are close to a semantic analysis are those proposed by Bey et al. [5]. In order to propose an automated assessment of learners' code in a Massive Open Online Course (MOOC), the authors propose to compare the control flow graph matching with the code produced by the learner, with a set of graphs stored in a database and manually assessed by experts [2]. If the learner's graph is recognized among the graphs of the database, then the matching score is returned to the learner; if it is not recognized, then the learner's production is manually assessed by a human expert to enrich the database of graphs.

The lack of work addressing automatic analysis of source code from a semantic perspective can be explained by the fact that the semantics of a program can not be calculated. Our research try to go beyond this limitation and propose the design of an indicator that reveals the relevance of a source code regarding a given problem.

3. SPECIFICATION OF THE SEMANTIC INDICATOR

Our objective is to design an indicator able to evaluate the distance between two source codes from a semantic perspective and fulfilling the following requirements: (i) teachers should only produce a solution to the problems delivered to students, in contrast, for example, to methods based on unit tests which are time consuming; (ii) the distance should decrease as the student approaches the solution, so that an incomplete script can be evaluated even if it does not fully address the problem posed. Therefore, if the solution of a problem is provided by the teacher, then the distance between this source code and the learners' productions should give insights about their ability to solve the problem.

The field of natural language processing has for a long time studied problems related, for example, to spelling correction where semantics is naturally taken into account [4]. To design our semantic indicator, we studied algorithms capable of processing the edit distance between two sequences of characters, and built our proposal upon the Levenshtein algorithm which has been shown very useful for calculating the distance between two strings of characters.

3.1 Levenshtein distance

The edit distance between two strings of characters, or Levenshtein distance [10], is defined by a cost calculated from the minimum number of operations (i.e., insertion and deletion of a character, and substitution of one character by another) required to move from one string to another. In the case of character strings comparison, the costs associated with each of these operations are all set to 1, but the cost of the substitution of a character by itself which is 0. To apply this algorithm to source code, two challenges must be tackled: (i) the design of a formal representation of the source code in order to make it comparable at a high abstraction level, and (ii) the assignment of a specific cost to each elementary operation according to the importance of the modifications to be made to move from one source code to another. Indeed, substituting two instructions implementing the same functionality should not have the same cost than replacing an instruction with another one characterized by a very different functionality.

3.2 Formal representation of source code

Our proposal for building a formal representation of source code relies on two steps: transformation of the source code as an Abstract Syntax Tree (AST), and transformation of the AST into a string of elementary instructions.

3.2.1 *Production of the abstract syntax tree*

We have adopted abstract syntax trees to formally represent a source code as they do not represent nodes and branches that do not affect the semantics of a program. We rely on a parser written in JavaScript that reads a script and returns the matching abstract syntax tree formatted as a JSON object; an example illustrating the conversion of a Bash source code to an AST is given in Figure 1.

With the formal representation proposed above, the measurement of our indicator corresponds to the edit distance between two abstract syntax trees. However, in order to adapt Levenshtein distance to our context, we need to convert the AST generated from the source code into a string of characters; from a terminology point of view, we name *tokens* the characters resulting from the AST transformation process so as not to confuse them with those contained in the source code. Thus, we carry out an in-depth exploration of the AST to create the string of tokens.

3.2.2 *Production of the string of tokens*

The Bash language, the subject of our study, proposes 17 control structures, each corresponding to a different token. A token can be of the type `Command`, of the type `Assignation`, or of the type `While`, `Do` or `Done` to indicate the beginning, body or end of a loop respectively. Note that the tokens `Command` keep the name of their command and arguments.The transformation of the AST represented in Figure 1 into a string of tokens is illustrated in Figure 2.

The Levenshtein distance between two strings of tokens is therefore characterized by a number of costs, or parameters, matching with the different elementary operations of insertion, deletion and substitution of each token. The 17 tokens of the Bash language lead to the definition of 308 costs: 17+17 costs to create and delete a token, 16x17 costs to substitute one token by another, 1 cost to substitute a token `Command` by another characterized by a different command name, and finally 1 cost to substitute a token `Command` by another characterized by identical names and different arguments. This number of parameters is too high to implement an effective machine learning method, as it decreases the density of tests and makes more difficult the search for "good" values of parameters.

3.3 Reduction of the number of parameters

A first operation to reduce the number of parameters to be trained consists in ignoring the tokens `Until`, `Subshell` and `Pipeline` because the matching control structures do not appear in the dataset used for this study (see Section 5). We also propose to symmetrisize the problem by assigning equal costs to the insertion and deletion operations of a token; similarly, the substitution of the token A by the token B has a cost equal to the substitution of B by A. This symmetrization makes the distance between the scripts S_1 and S_2 equal to the distance between S_2 and S_1, and reduces the number of parameters to 107.

Finally, we assume a stronger hypothesis by considering that only certain tokens can replace others. Thus, only the following groups of tokens can substitute each other: {`Command`, `Assignation`}, {`If`, `Case`}, {`Then`, `Else`, `CaseItem`}, and {`For`, `While`}. Indeed, it is possible that a script contains a `Case` where another script uses a `If`, but it is unlikely that a substitution of a `For` by a `If` appears frequently, these two instructions having very different objectives.

These simplifications considerably reduce the number of costs, since 23 parameters must now be calculated. Let us note θ the vector containing these 23 costs. Our indicator, noted $d(S, C, \theta)$, is then defined by the edit distance, under the 23 parameters θ, between the string of tokens representing the script S and the one matching with the script C. The next section introduces the machine learning criterion leading to the optimal values of θ.

4. MACHINE LEARNING METHOD

To train the correct values of the 23 costs of the θ parameter, we define the machine learning criterion $Score(\theta)$ to be minimized such that:

$$Score(\theta) = \frac{\sum_{(S,C)\in Correct} \sqrt{d(S,C,\theta)}}{\sum_{(S,C)\in Incorrect} \sqrt{d(S,C,\theta)}} \qquad (1)$$

where $Correct$ (resp. $Incorrect$) is the set of pairs composed of the correct (resp. incorrect) scripts of the learning dataset (see next section) and the associated corrections.

The $Score$ function is low when the correct scripts are associated to short distances, and the incorrect scripts to long distances. We add the square roots of the distances to reduce the influence of high values.

We can notice a property of the function d:

$$\forall \lambda \in \mathbb{R}_+^*, d(S, C, \lambda.\theta) = \lambda.d(S, C, \theta) \qquad (2)$$

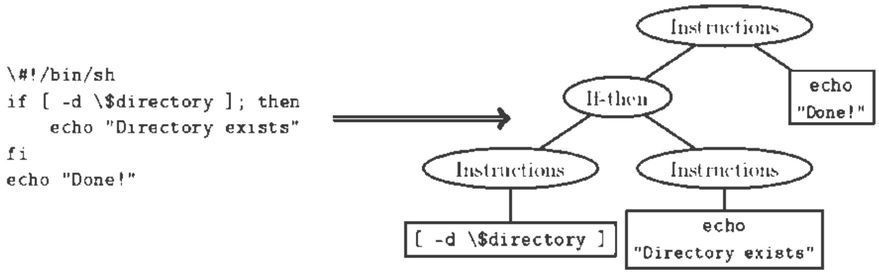
Then it is obvious that $Score(\lambda\theta) = Score(\theta)$. This means that the $Score$ function is constant on all rays from 0, and that exploring the different costs of the θ parameter, noted θ_i such as $0.01 < \theta_i \leq 1$, is sufficient to find the values of θ_i minimizing $Score$. The dataset built from an authentic learning situation and that was used to determine the θ_i is described in the next section.

5. DATASET FOR CALCULATING COST

The dataset was obtained in an IT department of Higher Education Institute of Technology (HEIT) during an introductory course on Bash programming attended by 166 first year students. Learners discover common commands such as `echo`, `ls`, `read` or `cat`, variable management, as well as some control structures of the Bash language (e.g., `for`, `while`, `if`, `case`). Students then put these concepts into practice during hands-on sessions where they produce Bash scripts to try to solve a series of problems.

The dataset includes the students' scripts produced during five practical sessions of one hour and a half. Each time a student saves the modifications of her Bash script, a copy of the script together with its timestamp and the identifier of the student is saved in a directory to enrich the dataset. Thus, our sample includes 19232 scripts. However, a number of nomenclature errors or file numbering made 18% of these scripts unusable: the dataset is composed of 15794 scripts.

This sample includes, for the same exercise, several scripts produced by the same student. However, machine learning methods are effective when they are based on independent data. We thus only consider, per exercise and per student, the last script produced by the learner because we assume



```
\#!/bin/sh
if [ -d \$directory ]; then
    echo "Directory exists"
fi
echo "Done!"
```

Figure 1: Converting a source code into an abstract syntax tree.

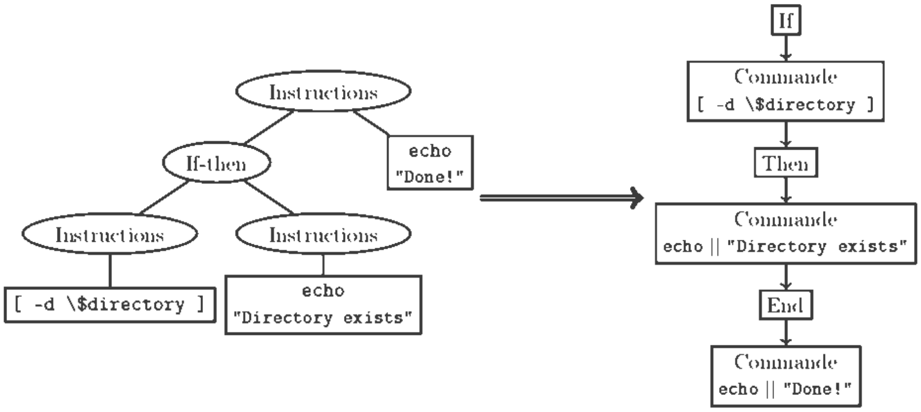


Figure 2: Converting an abstract syntax tree into a string of tokens.

that it is the most complete. This choice allows to obtain an independent dataset, but reduces the sample to 2540 scripts.

Finally, we want to be able to manually classify each script of the dataset into two distinct categories (i.e., *Correct* and *Incorrect*) expressing their semantic accuracy, in order to evaluate the accuracy of our indicator as a semantic classifier (see next section). However, some exercises suggested by the teachers of this course do not allow for a semantic analysis of students' productions. For example, some problems are related to the understanding of the execution of a script provided in the statement, while others have instructions that are too open to assess whether a script represents a solution to the problem. After this classification process, the final sample includes 733 scripts spread over 11 different exercises; 397 scripts are semantically correct, the others being semantically incorrect.

This dataset was randomly divided into a learning dataset whose objective is to identify the values of the θ parameter that minimize the machine learning criterion, and a test dataset that aims at evaluating the relevance of the results obtained during the machine learning phase. These two samples contain the same number of scripts for each exercise, and the proportion of semantically correct and incorrect scripts is preserved. In addition, let us note that a solution was provided by a teacher for each of the exercises.

6. RESULTS AND EVALUATION

6.1 Values of costs

The learning dataset was used to initialize the gradient descent minimization algorithm. The vector minimizing the $Score$ function, noted $\overline{\theta}$, admits values of 0.01 (i.e., the minimum value we have set) almost everywhere except for three parameters: (1) the creation or deletion of a token `Command`, whose cost is 1; (2) the creation or deletion of a token `If`, whose cost is 0.112; (3) the creation or deletion of a token `Function`, whose cost is 0.022.

At this stage, we have the parameter $\overline{\theta}$ which contains the values of the 23 costs that were obtained by our learning method. The objective of the tests conducted in the following section with the parameter $\overline{\theta}$ on the test dataset is twofold: to study the ability of our indicator to distinguish a semantically correct script from an incorrect script ; to study the correlation between our indicator and manual scores assigned to scripts by teachers.

6.2 Study of prediction

A first approach to assess the relevance of our indicator under the parameter $\overline{\theta}$ is to evaluate its ability to automatically classify a script as correct or incorrect from a semantics point of view. In a first step, we calculate the distance $d(S, C, \overline{\theta})$ between each script S of the test dataset and the correction C of the matching problem, in order to obtain a value of our indicator for each correct and incorrect script.

The next step consists in evaluating the prediction model resulting from the different values of the indicator. To do this, we use the ROC metric, which is widely used to observe the performance of a binary classifier. When the classifier calculates a metric m that is compared to a σ threshold to predict the class, the idea of the ROC curve is to vary σ from 1 to 0 and, for each σ value, plot the false positive rate on

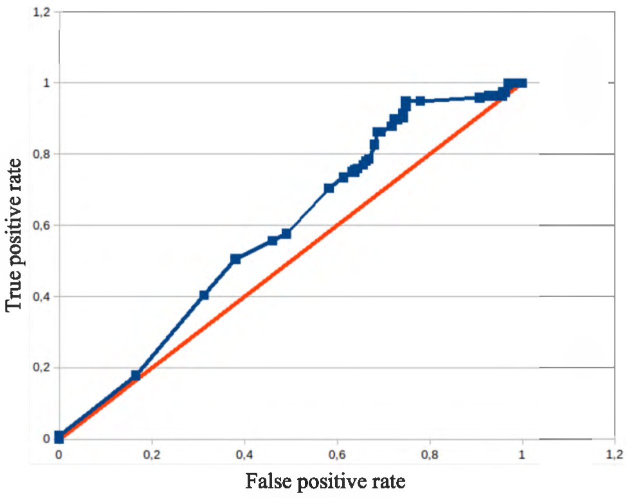


Figure 3: ROC curve of our classifier (in blue) and random bisector (in red).

the abscissa and the true positive rate on the ordinate. The area under the curve (AUC) then indicates the probability that the classifier will place a positive in front of a negative. A classifier that is never wrong has an AUC equal to 1, while a random classifier has an AUC equal to 0.5.

In our study, if $d(S, C, \overline{\theta}) < \sigma$, then the script is classified in the category *Correct* ; otherwise it is classified in the category *Incorrect*. The ROC curve in Figure 3 illustrates the ratio of true positives (i.e., fraction of correct scripts that are actually detected as correct by the classifier) according to the ratio of false positives (i.e., fraction of incorrect scripts that are detected as correct by the classifier). The curve is moderately above the bisector, the area under this curve is equal to 0.585. Our classifier is therefore slightly better than a random classifier, but its performance is not very high; we make assumptions to explain these results in Section 7.

6.3 Study of the correlation with human notation

One of the objectives of our indicator is to reflect the learner's progress towards the solution to the problem, which means that its value is assumed to decrease as a script moves towards the solution. We therefore study the correlation between the value of our indicator under the parameter $\overline{\theta}$ for a given script and the matching correction, and the score assigned to this script by a human tutor.

This study is performed using a second dataset obtained after the final exam of the course described in Section 5. Of the 166 students enrolled in this course, 163 participated in the final exam where learners had to produce a script addressing a given problem. A teacher then assigned a score out of 10 to each of the 163 scripts. After eliminating grammatically incorrect scripts, this second dataset comprises 105 scripts.

Figure 4 shows, for each script, the value of the indicator according to the score assigned by the teacher. This graph shows a negative correlation: the (non-linear) Kendall correlation has a value equal to -0.319 (p value less than $2.7 \cdot 10^{-6}$), the (linear) Pearson correlation having a value of -0.446 (p value equal to $1.9 \cdot 10^{-6}$). We therefore have a correlation between the score given by a human tutor and the value of our indicator. This correlation is not extremely high, but it may be sufficient to automatically estimate a student's progress towards solving a problem.

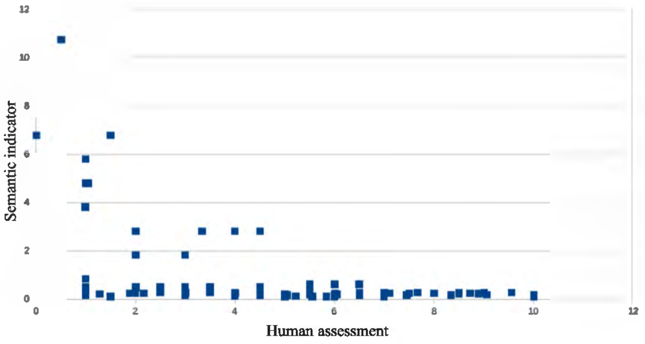


Figure 4: Value of our indicator according to score assigned by a human tutor.

7. DISCUSSION

Our indicator is able to distinguish a semantically correct script from a semantically incorrect script, but its performance is slightly better than that of a random classifier only. A hypothesis to explain these results stems from the heterogeneous nature of the learning dataset, especially regarding the size of the scripts it contains. Indeed, the distance calculated by our indicator tends to increase with each token of a script. So the longer a script is, the more likely it is that the indicator will return a high value (even if the script approaches the solution), while a short and incorrect script is evaluated with a low value due to the low number of tokens.

The correlation study carried out on the second set of data gives indications about a certain capability of generalization of our approach. Indeed, unlike the test dataset extracted from the first dataset, the set of scripts obtained after the terminal exam corresponds to an exercise that is missing from the learning dataset. Therefore, our method seems to apply to exercises that were not used during the learning phase. However, this correlation is not extremely high, and scripts that are very poorly evaluated by human tutors are assigned a value of our indicator that is almost zero. This can also be explained by the hypothesis formulated above, since almost empty and therefore incorrect scripts are poorly rated by the human teacher, while our indicator returns a low value. Tests should be carried out to investigate how to adjust the algorithm to process the indicator, in particular according to the number of tokens comprised in the script representing the solution to the problem.

Developments have been initiated to return this indicator to learners during the programming activity. A first visualization reflects, as a graphical gauge, the value of the indicator each time a learner executes a script. A second visualization shows the learner's progress in solving the problem from the successive values of the indicator for the same script; it gives the variation of the indicator in the form of a gradient of colors ranging from green (for a short distance) to red (for a long distance). The student is thus provided with a real time and easy to interpret view of the evolution of the accuracy of his script from a semantic point of view. These tools to support learners will be experimented in the next academic year with a new group of first-year students from HEIT. This experimentation will also make it possible to repeat the work presented in this study using a new dataset, and thus to improve the quality of our indicator.

8. CONCLUSION

In this paper, we conducted a study to design an indicator whose objective is to act as a foundation for intelligent guidance systems and awareness tools intended for both learners and teachers during a practical activity dedicated to learning programming. While a large variety of research is focused on the syntactic quality of the code produced by learners to support these learning activities, our originality lies in studying the semantic quality of the code, i.e., in evaluating the degree to which a program solves a given problem.

Thus, our main contribution relates on the design of an indicator that reflects a learner's ability to solve a problem. To answer the first research question asked in the introduction, we adapted the Levenshtein distance: from two strings of *tokens* representing the abstract syntax tree of the corresponding programs, the indicator returns an estimated value of the edit distance between the two scripts. To train and test this indicator, we used a dataset composed of scripts produced by students in authentic learning situations. The tests revealed that when it comes to differentiating between semantically correct and incorrect scripts, our indicator has slightly better performance than a random classifier. On the other hand, we observed an inverse correlation between the value of the indicator and the score assigned by a human tutor: the higher the human score of a program formulating a solution to a problem, the smaller the distance between that program and the solution of the problem.

These encouraging results suggest the opportunity to develop new models and tools dedicated to the semantic analysis of programming learning. However, many improvements need to be explored to improve the quality of our indicator. In addition to big amount of data required to refine the parameters of our indicator, the generalization of our approach to different programming languages must be checked, as well as the consideration of more complex programs.

9. REFERENCES

[1] A. Ahadi, S. Lal, J. Leinonen, A. Hellas, and R. Lister. Performance and consistency in learning to program. In *Proceedings of The Nineteenth Australasian Computing Education Conference*, pages 11–16, Geelong, 2017. ACM.

[2] R. Aiouni, A. Bey, and T. Bensebaa. An automated assessment tool of flowchart programs in introductory programming course using graph matching. *Journal of e-Learning and Knowledge Society*, 12(2):141–150, 2016.

[3] A. Alammary, A. Carbone, and J. Sheard. Implementation of a smart lab for teachers of novice programmers. In *Proceedings of The 14th Australasian Computing Education Conference*, pages 121–130, Melbourne, 2012. ACS.

[4] K. Beijering, C. Gooskens, and W. Heeringa. Predicting intelligibility and perceived linguistic distance by means of the levenshtein algorithm. *Linguistics in the Netherlands*, 25(1):13–24, 2008.

[5] 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*, 21(2):259–272, 2018.

[6] E. Carter and G. D. Blank. A tutoring system for debugging: status report. *Journal of Computing Sciences in Colleges*, 28(3):46–52, 2013.

[7] P. Denny, A. Luxton-Reilly, and D. Carpenter. Enhancing syntax error messages appears ineffectual. In *Proceedings of The 2014 Conference on Innovation & Technology in Computer Science Education*, pages 273–278, Uppsala, 2014. ACM.

[8] P. Ihantola, A. Vihavainen, A. Ahadi, M. Butler, J. Börstler, S. H. Edwards, et al. Educational data mining and learning analytics in programming: Literature review and case studies. In *Proceedings of The 2015 ITiCSE on Working Group Reports*, pages 41–63, Vilnius, 2015. ACM.

[9] M. Karam, M. Awa, A. Carbone, and J. Dargham. Assisting students with typical programming errors during a coding session. In *Proceedings of The Seventh International Conference on Information Technology*, pages 42–47, Las Vegas, 2010. IEEE.

[10] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710, 1966.

[11] M. Saeli, J. Perrenet, W. M. Jochems, and B. Zwaneveld. Teaching programming in secondary school: A pedagogical content knowledge perspective. *Informatics in Education*, 10(1):73–88, 2011.

[12] K. Sharma, P. Jermann, and P. Dillenbourg. Identifying styles and paths toward success in moocs. In *Proceedings of The 8th International Conference on Educational Data Mining*, pages 408–411, Madrid, 2015. IEDMS.

[13] K. Sharma, K. Mangaroska, H. Trætteberg, S. Lee-Cultura, and M. Giannakos. Evidence for programming strategies in university coding exercises. In *Proceedings of The 13th European Conference on Technology Enhanced Learning*, pages 326–339, Leeds, 2018. Springer.

[14] J. Spacco, P. Denny, B. Richards, D. Babcock, D. Hovemeyer, J. Moscola, and R. Duvall. Analyzing student work patterns using programming exercise data. In *Proceedings of The 46th ACM Technical Symposium on Computer Science Education*, pages 18–23, Kansas City, 2015. ACM.

[15] A. Taherkhani and L. Malmi. Beacon-and schema-based method for recognizing algorithms from students' source code. *Journal of Educational Data Mining*, 5(2):69–101, 2013.

[16] C. Watson, F. W. Li, and J. L. Godwin. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *Proceedings of The 13th International Conference on Advanced Learning Technologies*, pages 319–323, Beijing, 2013. IEEE.