A Proposal for Source Code Assessment Through Static Analysis

1st Ricardo Lemos de Souza
PPG Modelagem Computacional
Universidade Federal do Rio Grande
Rio Grande - RS, Brazil
rcrdsou@hotmail.com

2nd Fabiana Zaffalon Ferreira
PPG Educação em Ciências
Universidade Federal do Rio Grande
Rio Grande - RS, Brazil
fabinhazaffalon@gmail.com

3rd Silvia da Silva Botelho
Centro de Ciências Computacionais- C3
Universidade Federal do Rio Grande)
Rio Grande - RS, Brazil
silviacb.botelho@gmail.com

***Abstract*—This Research to Practice Work in Progress paper presents a proposal for source code assessment through static analysis. The presence of computation is constantly growing in the contemporaneous world, and in that way, the demand for professionals capable to develop and maintain software is also in constant growth. The present work aims to develop a model where teachers can identify potentially weakness on student's set of skills for programming, and therefore be able to work on solutions for their educational development. Preliminary results show that it is possible to identify prominent skills used to solve a given problem, but also that it is possible to compensate the lack of those skills with others.**

***Index Terms*—Static analysis, source code assessment, computer programming education**

I. Introduction

The presence of computation is constantly growing in the contemporaneous world, be it on computers, smartphones, vehicles or even home appliances capable of connecting to the internet, the number of devices that have embedded software is every day bigger [1] [2]. In that way, the demand for professionals capable to develop and maintain software is also in constant growth.

The learning of computer programming encompasses the development of a set of skills such as logic, mathematics and others not directly related to computer science as well [3] [4]. To attend the growing demand for computer programmers, there is also the propagation of new programming teaching environments, where online courses seeks to develop ways to better teach and asses their students [5] [6].

Web-based courses offers the opportunity to automatically asses the source codes generated by students, and many platforms have developed their own assessment system [6]. Most teaching environments make use of dynamic analyses of source codes, in a way that given a standard set of inputs, the outputs generated by the assessed source code are compared to an expected set of answers [7].

Another approach is static analysis, which does not require the execution of the assessed source code, it utilizes applied statistics on the elements of the code, such as identifiers and operators, and is capable to asses codes that fail to be compiled, and unable to be executed [5]. As an educational resource for assessment, static analysis provides the opportunity to collect data about how students develop their skills, not only by their right answers, but also by their wrong ones [7].

This work makes use of natural language processing techniques, adapted to source code static analysis. The aim is to identify groups of elements in the code, which we believe will represent a set of skills related to the solution of a given problem. Initially, we develop a parser to identify and represent a source code as a numeric vector, extracting not only syntactic elements, e.g. operators and reserved words, but also semantic expressions, e.g. functions and recursive callings.

The parser was applied on a data set of source codes collected on an Online Judge, vectorizing 3144 codes related to six problems. Then, the codes have been organized as a document per tokens matrix for every problem, and later a document per token frequency matrix. Finally, it was made use of statistic methods such as TF-IDF and cosine similarity to search for elements that allow us to identify skill's groupings for every problem.

The present work aims to develop a model where teachers can identify potentially weakness on student's set of skills for programming, and therefore be able to work on solutions for their educational development. In the following sections will be presented some related works, the methodology adopted on this work and preliminary results and observations.

II. Background

Introductory computer programming course are acknowledged by high failure and dropout rates [3] [8]. Novice programmers are face with not only the challenge of learning a computer-based language, but also on how to apply previous knowledge, e.g. mathematics, logic and text interpretation, in a way to elaborate an algorithmic solution to problems [4].

Virtual learning environments (VLE) are widely used for computer programming courses and provide an opportunity for educators to track the development of student's skills by analyzing their source code submissions, and many of this VLE make use of automatic assessment (AA) methodology [7]. Most AA systems make use of either dynamic or static assessment, which are well discussed in the works of Ulla [7] and Ala-Mutka [5].In this work, static analysis of source

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

codes was chosen as the methodology for assessment, given the ability to examine even codes that fail to be interpreted by compilers.

Information retrieval methodologies have been successfully used in the evaluation of source codes. In that way, the use of natural language processing techniques was made, as in the works of Azcona [9], Ulla [10] and Ganguly [11], transforming the formal computer program language in which the codes where submitted to a numeric representation, more adequate to apply statistical techniques.

As the main goal was to identify possible skills through the analysis of source codes, as an initial reference in this stage of the present work, we made use of the concepts/capabilities elaborated by Barr [12]. Barr and Stephenson have organized nine concepts of computational thinking, originally proposed by Wing [13], in a table of examples according to five areas of the K-12 curriculum, one of which is computer science. This work [12] was chosen as reference because of the easily identifiable elements in a source code, as follows:

- Data collection: find a data source for a problem.
- Data Analysis: write a problem to do basic statistical calculations.
- Data representation: use data structures.
- Problem decomposition: define objects, methods and functions.
- Abstraction: use procedures to encapsulate a set of often repeated commands that perform a function; use conditionals, loops, recursion, etc.
- Algorithm & procedures: study classic algorithms, implement algorithms for specific area.
- Parallelization: threading, pipelining, dividing up data or task such a way to be processed in parallel.
- Simulation: algorithm animation, parameter sweeping.

The ninth concept of computational thinking was Automation, which contained no examples, as elaborating an algorithm is a way of automatizing a process or solution that absence is intuitive regarding computer science, thus will be excluded from the present study. In the next session will be described the current methodology being used on the analysis of source codes.

III. Methodology

For the present work we made use of a data set compose of source codes submitted in a competition in an online judge system. It was chosen the "GNU C" language for the experiments, and in that way all codes that did not correspond to that language where excluded. In addition, only problems that totaled at least 300 submissions where selected. As such, we have processed 3144 source codes regarding six problems.

The information available about each submission was: submission Id, which was subtracted on pre-processing; problem Id, as the identification of the problem; language, as the formal computer programming language the submission was written; source, as the actual submission text; result, indicated the correctness result of the submission.

In order to work with the data, it was transformed to a numerical representation. That process was called the tokenization, similar to those described in Azcona [9], Ulla [10] and Ganguly [11]. To extract the information from the source codes a parser have being elaborated with the concepts of Barr [12] in mind. A simple parser schema is represented in Fig.1.

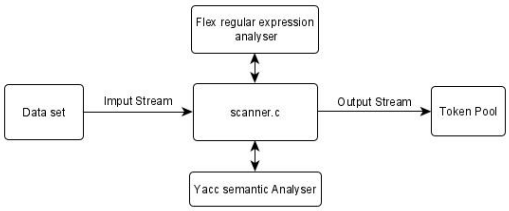

Fig. 1. Basic parser schemma

The use of a customized parser has contributed to the processing speed of the data set. It also made possible to extract semantic aspects present in the source code directly, e.g. function definition and recursive calls, as unique tokens to be added on the total token pool. Given particular aspects of the "C" language, and the data set used, no token was set for the "Parallelization" concept, and it was omitted at the present time. Table I presents concepts and related tokens extracted.

Skill concept	Tokens
Data Collection	Scanf, DB calls, File Calls
Data Analysis	Arithmetic operator
Data representation	Identifiers , Arrays, Pointers, Structs
Problem Decomposition	Function Definitions, Library calls, Source imports, Macro definitions
Abstraction	Function calls, Logic operators, Loops, recursive calls, Selections
Algorithms & procedures	Binary result of submission
Simulation	Comments, printf, plots

TABLE I
SKILL CONCEPT AND TOKENS EXTRACTED

The extracted data was then organized in a document per token matrix for each selected problem, i.e. every problem was represented as a matrix where each submission was allocated in a row with every token extracted inserted in a column, in order of extraction from the original code. Next, another matrix corresponding to each exercise was constructed, corresponding to a document per frequency matrix, in which tokens were grouped according to the concepts previously set, regarding the competences of computational thinking [12], that in the present work will be used as a skill set reference.

The frequency matrices were balanced applying Term frequency per inverse document frequency (TF-IDF) with the equations 1, 2 3, where f represents the frequency, t term or token, d document or submission and N the total number of submissions. The balanced information was then stored in another matrix, a TF-IDF balanced matrix.

$$tf_{t,d} = \frac{f_{t,d}}{max\left(f_{t,d} : t \in d\right)} \tag{1}$$

$$idf_{t,D} = log \frac{N}{\{d \in D : t \in d\}} \quad (2)$$

$$tfidf_{t,d,D} = tf_{t,d} \times idf_{t,D} \quad (3)$$

In order to evaluate the differences of two source codes cosine similarity on a vector space model has been used. Equation 4 was used to determine the similarity.

$$\cos\theta = \frac{d_1 d_2}{||d_1||\,||d_2||} \quad (4)$$

Although the comparison of the similarity of two submissions is not the main focus of the present work, like in many plagiarism detection IR models [10] [11], it was made in search of elements that could point the differences in skills between the submission authors. In the next section will be presented the experiments made, some results and considerations.

IV. Experiments

The original data set was processed as described on the previous sections. For the initial processing as in the following experiments, some standard functions of the MATLAB R2016a software were used like std(), sum(), mean(), etc., while others required for this work were implemented by the authors. For the following experiments the skill "Algorithms & procedures" was also subtracted from the analysis, given that is a binary representation of right and wrong, being used only to establish the "Accepted" and "Error" groups respectively. This section aims to answer:

- RQ1: Can the TF-IDF be used for identifying a set of programming skills on a source code?
- RQ2: Can programming skill levels be estimated using TF-IDF over a source code?

As in all correct submissions and most incorrect all skills were present, IDF was very low, even scoring zero in some cases. This implied that the TF had a higher weight on determining TF-IDF, or even preventing the effective use of the method. In order to adjust the use IDF it was only considered the frequency of the documents in which the token count where higher than the sum of the mean $m_{t,D}$ with its standard deviation $std_{t,D}$, equation 2 was balanced as 5:

$$idf_{t,D} = log \frac{N}{\{d \in D : t > (m_{t,D} + std_{t,D}) \in d\}} \quad (5)$$

The first observation to be consider is that TF-IDF has demonstrated to be an efficient normalizer for comparing common skill throughout all problems. As shown in the Fig.3 the average calculated skills for Problems 1 and 2 can be equated regardless of the solution size, this was observed for all 6 problems, which differ in complexity and token count.

Another observation is that, at this work stage, TF-IDF was not able to emerge discriminatory skills that qualify as a required set to successfully answer a problem. Fig.2 illustrates the mean count of tokens per skill on problem 1 and 2 for accepted submission, i.e. answers judged correct, end erroneous ones, while Fig. 3 shows the mean TF-IDF for

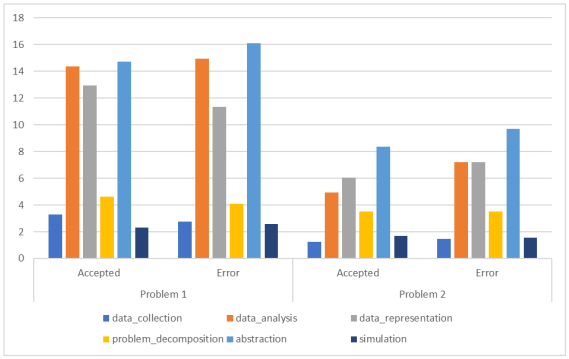

Fig. 2. Mean token per Skill

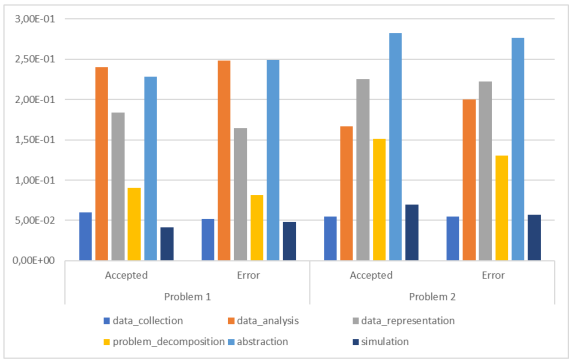

Fig. 3. Mean TF-IDF score per skill

the same problems. By comparing the submissions within a problem set, it could be possible to imply that accepted source codes and erroneous ones shares average scores on each skill.

The nature of problem solving on an algorithmic way makes a variety of different solutions possible, that is easily observable on the data set. The calculated standard deviation for each skill on every problem has not demonstrated to be constant, reaching as much as 33% of the mean on some cases. The same problem can have very high token counts in a submission, while in another a very low. To determine how different are the skills used to solve the same problem only the accepted submissions were observed, in that way it is possible to affirm that the compared submission successfully fulfills the same purpose.

The number of tokens present in a source code has been used elsewhere [14] [15] [16] [17] to determine software quality, and the number of elements present in a software, among other characteristics, may be used as an indication of efficiency, in a way that if two software can execute the same task, the one with the lowest complexity is often the most efficient [17]. In the same way when considering a skill used to elaborate a solution the direct sum of an element may not represent elevated proficiency, i.e. having high occurrences of an element in a source code can infer a low related skill. Given the characteristic of the present work, were the information is extracted based on the frequency of tokens, and used to estimated skills, it was assumed that a skill either has an

inverse proportionality θI to the lowest token frequency (6), or an direct proportionality θD to the highest token frequency (7), as follows:

- Data collection: inverse.
- Data analysis: inverse.
- Data representation: inverse.
- Problem decomposition: direct.
- Abstraction: inverse.
- Simulation: direct.

$$\theta I_{t,d} = \frac{\min f_{t,d}}{f_t} \tag{6}$$

$$\theta D_{t,d} = \frac{f_t}{\max f_{t,d}} \tag{7}$$

For the following experiment the submission with the highest sum of θ scores for the problem 1 submissions was chosen (sub 69), as a comparative the lowest sum of scores submission (sub 85) was selected, and also the most different to the highest submission given the cosine similarity (sub 60). While the TFIDF normalizes the skill scores of different submissions like 69 and 85, as illustrated in Fig 4, and does point the importance of each skill for the assessed submission, it does not allow at this time to compare quality over different codes. However, accounting that both submissions are considered correct, we can infer that submission 69 may be more efficient, given the considerably smaller number of tokens.

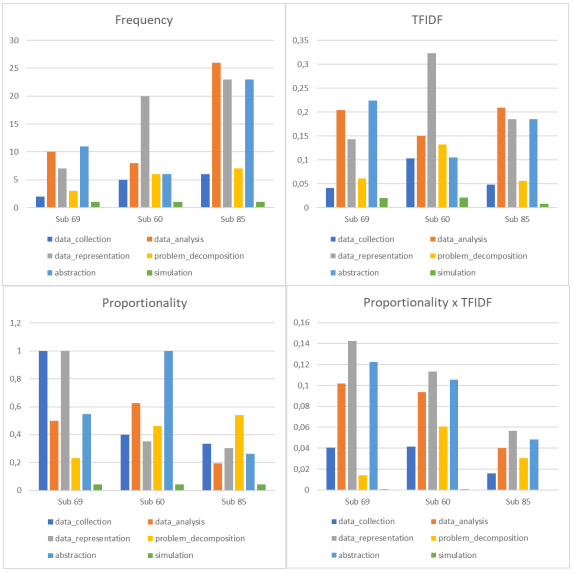

Fig. 4. Submissions comparison

Another indication that submission 69 makes use of higher skills levels is that it scores 3.31789 in the sum of all θ, while sub 85 only 1.670987. As θ score was the criteria to choose both sources 69 and 85, manual assessment of the codes was made to evaluate that indication, and the differences were remarkable, submission 69 was clean and simple, while 85 made excessive use of identifiers, loops and attributions. It may be worth to point that sub 85 made use of programming concepts considered complex, e.g. pointers and dynamic memory allocation, sub 69 in the other hand made use of only very simple structures, thus it is difficult to determine what is the most seasoned programmer of the two submissions.

The cosine similarity of submission 69 weighted with TF-IDF indicated sub 60 as the most different in the problem set, while that may be correct as both codes are not similar even in manual assessment, the proximity of code 69 to code 85 TF-IDF denotes inaccuracy. The revised samples this far have showed a high occurrence of false positives regarding similarity proximity, but high accuracy for differences i.e., at the present work state cosine similarity weighted with TF-IDF is inaccurate to determine if source codes are similar, and accurate to determine if they are different.

As a final experiment, given that the TF-IDF aims to emerge the importance of a skill for a submission, and the θ score aims to set the quality of skills in a submission compared to others in a problem set, an investigation if the product of this two data may be used to assess skill levels over source codes. The final demonstrative in Fig. 4 shows a comparison of skill levels of the same submissions used on the previous experiment. At the present work state that product has showed promising results, though inconclusive given the low number of revised samples.

V. Final Considerations

Assessing of computer programming source codes has been widely discussed over many aspects and applications, among them the educational purpose. While dynamic assessment may provide a practical way to evaluate correctness, in education it is often more rewarding to look beyond, the meaning of assessing is to provide information that can be used to foment growth and development. In this work in progress paper was discussed some experiments in the pursuit of elements to support the teaching and learning of computer programming.

The skills involved in programming are not always clear and constant as is the curricula of novice programming courses, as some experiments have showed, knowledge of complex tools does not always means a polished code, the simplest solution is often the best. Natural language processing techniques were used with success for assessing formal programming languages. The TF-IDF has proven to be valuable to address the importance of a given skill for a submission, though was unable to determine the dependency of a solution for a skill, results this far have showed that programmers can use different skills to address the same problem, and in different levels.

Future work will contemplate the refining of statistical techniques, setting correlations over the generated data and manual assessment to validate results. We also look forward to test other references for skill sets, like Bloom's taxonomy. The use of machine learning techniques will be contemplated to incorporate the work with TF-IDF and proportional scores, and furthermore be used for emerging skill sets.

ACKNOWLEDGEMENT

Universidade Federal do Rio Grande (FURG) and Instituto Federal de Educação Ciência e Tecnologia Sul-rio-grandense (IFSUL).

REFERENCES

[1] X. Su, J. Qiu, T. Wang, and L. Zhao, "Optimization and improvements of a moodle-based online learning system for c programming," in *2016 IEEE Frontiers in Education Conference (FIE)*. IEEE, 2016, pp. 1–8.

[2] P. H. de Souza Bermejo, A. O. Tonelli, R. D. Galliers, T. Oliveira, and A. L. Zambalde, "Conceptualizing organizational innovation: The case of the brazilian software industry," *Information & Management*, vol. 53, no. 4, pp. 493–503, 2016.

[3] A. Robins, "Learning edge momentum: A new account of outcomes in cs1," *Computer Science Education*, vol. 20, no. 1, pp. 37–71, 2010.

[4] C. Cabo, "Student progress in learning computer programming: Insights from association analysis," in *2019 IEEE Frontiers in Education Conference (FIE)*. IEEE, 2019, pp. 1–8.

[5] K. M. Ala-Mutka, "A survey of automated assessment approaches for programming assignments," *Computer science education*, vol. 15, no. 2, pp. 83–102, 2005.

[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)*, 2016, pp. 147–156.

[7] Z. Ullah, A. Lajis, M. Jamjoom, A. Altalhi, A. Al-Ghamdi, and F. Saleem, "The effect of automatic assessment on novice programming: Strengths and limitations of existing systems," *Computer Applications in Engineering Education*, vol. 26, no. 6, pp. 2328–2341, 2018.

[8] J. Raigoza, "A study of students' progress through introductory computer science programming courses," in *2017 IEEE Frontiers in Education Conference (FIE)*. IEEE, 2017, pp. 1–7.

[9] D. Azcona, P. Arora, I.-H. Hsiao, and A. Smeaton, "user2code2vec: Embeddings for profiling students based on distributional representations of source code," in *Proceedings of the 9th International Conference on Learning Analytics & Knowledge*, 2019, pp. 86–95.

[10] F. Ullah, J. Wang, M. Farhan, S. Jabbar, Z. Wu, and S. Khalid, "Plagiarism detection in students' programming assignments based on semantics: multimedia e-learning based smart assessment methodology," *Multimedia Tools and Applications*, pp. 1–18, 2018.

[11] D. Ganguly, G. J. Jones, A. Ramírez-De-La-Cruz, G. Ramírez-De-La-Rosa, and E. Villatoro-Tello, "Retrieving and classifying instances of source code plagiarism," *Information Retrieval Journal*, vol. 21, no. 1, pp. 1–23, 2018.

[12] V. Barr and C. Stephenson, "Bringing computational thinking to k-12: what is involved and what is the role of the computer science education community?" *Acm Inroads*, vol. 2, no. 1, pp. 48–54, 2011.

[13] J. M. Wing, "Computational thinking," *Communications of the ACM*, vol. 49, no. 3, pp. 33–35, 2006.

[14] T. J. McCabe, "A complexity measure." *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[15] M. H. Halstead, "Elements of software science (operating and programming systems series)," *Elsevier Science Inc.*, 1977.

[16] E. L. Moreira, Mireille Pinheiro; Favero, "Um ambiente para ensino de programação com feedback automático de exercícios." *Workshop sobre Educação em Computação (WEI 2009)*, 2009.

[17] L. Capitán and B. Vogel-Heuser, "Metrics for software quality in automated production systems as an indicator for technical debt," in *2017 13th IEEE Conference on Automation Science and Engineering (CASE)*. IEEE, 2017, pp. 709–716.