A Tool for N-way Analysis of Programming Exercises

C. Andujar[1], M. Comino[1], M. Fairen[1], A. Vinacua[1]

[1] ViRVIG, Computer Science Department, Universitat Politecnica de Catalunya, Jordi Girona 1-3, Barcelona, Spain

Abstract

Programming exercises are a corner stone in Computer Science courses. If used properly, these exercises provide valuable feedback both to students and instructors. Unfortunately, the assessment of student submissions through code inspection requires a considerable amount of time. In this work we present an interactive tool to support the analysis of code submissions before, during, and after grading. The key idea is to compute a dissimilarity matrix for code submissions, using a metric that incorporates syntactic, semantic and functional aspects of the code. This matrix is used to embed the submissions in 2D space, so that similar submissions are mapped to nearby locations. The tool allows users to visually identify clusters, inspect individual submissions, and perform detailed pair-wise and abridged n-way comparisons. Finally, our approach facilitates comparative scoring by presenting submissions in a nearly-optimal order, i.e. similar submissions appear close in the sequence. Our initial evaluation indicates that the tool (currently supporting C++/GLSL code) provides clear benefits both to students (more fair scores, less bias, more consistent feedback) and instructors (less effort, better feedback on student performance).

1. Introduction and Related Work

Grading programming exercises takes a considerable amount of time. Automatic judges [AM05, PGR12, FJA16, KCF*19] have been proposed to simplify grading. These are valuable tools but since the output is just a pass/fail verdict, they provide limited feedback to students and disregard non-functional aspects of the code.

Our tool aims to support the following evaluator tasks:

- Define grading criteria considering not only functional correctness but also code quality (readability, efficiency, robustness).
- Analyze the submissions to apply the defined criteria.
- Detect similar or identical code submissions (plagiarism).
- Collect and report evidences supporting plagiarism suspicions.
- Identify frequent errors and major flaws in student's skills.

By supporting these tasks, we wish to achieve the following goals:

- Minimize evaluator efforts.
- Get more fair scores, e.g. by minimizing score variance between similar solutions.
- More accurate plagiarism detection: minimize false positives, spot most true positives.

We realized that many of the tasks above can greatly benefit from a global view of the student submissions (or a representative sample of these) and as such the tool has been designed to facilitate code comparisons. Most code comparison tools (e.g. diff-like) have been designed to facilitate merging different versions of the same file. As such, these tools are often limited to pair-wise comparisons (three and four-way comparisons at most) and show changes at character-level. We do use pair-wise comparisons, but our scenario has substantially different needs. Instead of code merging, the main goal is the analysis of code submissions. Here, we are interested in syntactical aspects (code readability, potential evidences of plagiarism) but also semantic and functional aspects that go beyond character-level differences. Furthermore, we should be able to perform n-way comparisons. Moreover, student submissions might exhibit large variations (e.g. in terms of identifiers, structure and sentence order).

Concerning the assessment of functional correctness, automatic judges [KLC01, PGR12, FJA16] provide a pass/fail verdict based on test sets. Some of these judges support Computer Graphics assignments by comparing the output image of student submissions with the instructor reference solution [ACFV18]. Some authors tackle the problem of semantic understanding of computer programs [HICS80], but state-of-the-art methods for checking functional equivalence [Gör16, Jam17] are limited to elementary algorithms.

A few works [ARVV19] address the automatic syntactic and semantic analysis of student submissions. Besides checking the output against test sets, a set of instructor-defined and automatically-generated rubrics are computed for all submissions to facilitate marking and provide richer student feedback.

Although plagiarism detection was not the main focus of our tool, incidentally we observed that it facilitates copy detection and we could spot copies in past course submissions that were not detected at that time. Most plagiarism detection tools are designed for essays [NLM15, WW16, BRG20] and perform poorly on source code. Some tools do specialize in source code [PMP*02, ASR06, LCHY06, CJ11, RK19, Mos], but often result in false positives that must be inspected carefully.

DOI: 10.2312/eurp.20201122

2. Our approach

Pair-wise dissimilarities Given two source files F_1, F_2, we compute their dissimilarity $d(F_1,F_2)$ as $\alpha d_o(F_1,F_2)+\beta d_c(F_1,F_2)+\gamma d_s(F_1,F_2)$, where α, β and γ are user-defined weights and d_o, d_c and d_s represent respectively operational, character-level, and semantic dissimilarities. The operational dissimilarity is computed by running test sets on all submissions, and counting the cases for which F_1 and F_2 yield a different pass/fail result. The character-level dissimilarity mimics diff-like tools. We first find the longest contiguous matching subsequence s in F_1, F_2, and repeat this recursively to the pieces to the left and to the right of s. Once all matches have been found, d_c is computed as $1-2M/T$, where M is the number of characters in the matching blocks and T is the total number of characters in F_1 and F_2. Before computing d_c, we use a parser to identify tokens, reformat the code and remove comments. The semantic dissimilarity is based on the analysis ideas presented in [ARVV19]. Our tool generates automatically a large number of features (e.g. calls to *asin*() function, or coordinate space in which variables are used). Then, a Pearson's χ^2 test is used to determine the potential impact of each feature on the pass/fail proportion. Features with low p-values are likely to impact output correctness and thus are kept as relevant. We then compute d_s as the number of non-matching features in F_1 and F_2. We use a high-level API [AVV20] for detecting such features, so that the tool can show issues compromising code efficiency (e.g. nested loops), quality (e.g. inadequate coordinate space) or robustness (e.g. float equality comparisons).

Dissimilarity matrix Let N be the number of submissions. During preprocess, we fill an $N \times N$ dissimilarity matrix for each of the three dissimilarities above, where each entry stores the corresponding pair-wise dissimilarity.

Embedding in 2D We represent each submission as a circular node embedded in 2D (Figure 1). Since we wish distances in the embedding to preserve as much as possible inter-node dissimilarities, we use non-metric multi-dimensional scaling [Kru64]. We prevent identical submissions from getting perfectly overlapping nodes, we add a small constant to all computed dissimilarities.

GUI parts The main window of our prototype has two parts: a *2D view* showing the embedded submissions, and a *table* with submission data (student ID, name...) . We used fake IDs and names to preserve privacy. Submissions can be selected via mouse in the 2D view or in the table. The 2D view outlines all submissions (Figure 1). Clusters can be detected easily; depending on the chosen dissimilarity weights, clusters might represent copies or just submissions adopting a similar approach. When the user moves the pointer over a node in the 2D view, a hover box shows the associated (reformatted) source code. If students were provided with some starting code, unmodified lines are shown in gray.

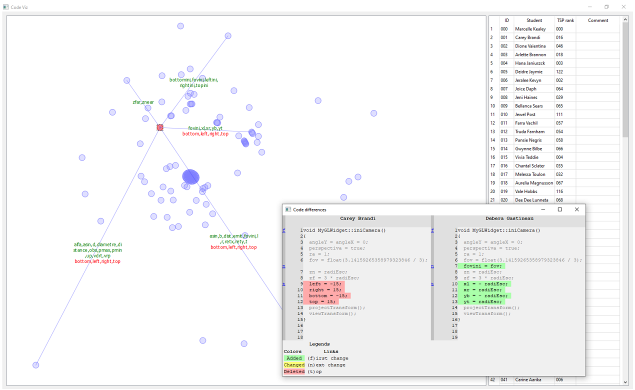
Pair-wise inspection Selecting two submissions opens a new window displaying code differences in a diff-like style. Users can choose to highlight only character-level differences (using the block matches resulting from finding longest contiguous matching subsequences) or to include also differences in terms of semantic features and operational pass/fail results.

One-to-many inspection Selecting a single node shows a summary of differences between the selected node and a random set of surrounding nodes. These differences are shown as labels over arcs connecting the node and its neighbors. We allow users to choose the font size and what differences to include: relevant tokens (e.g. C++ identifiers and keywords), relevant features, and/or pass/fail results. The random subset is chosen so that the labels over different arcs do not overlap (Figure 1).

3. Results, Discussion and Future Work

In an informal evaluation with 8 exercises and 140 student submissions, the tool was found to greatly facilitate tasks before, during and after grading. Before grading, instructors were able to spot clusters immediately, and the tool helped checking whether these clusters corresponded to uncompleted exercises (with no or little changes wrt the initial code), similar approaches, or just copies. Outliers often revealed completely wrong or overly complex solutions. This global picture provides insights to define grading criteria and for weighting different code aspects. During grading, the tool computed an approximate solution to the Traveling Salesperson Problem (TSP) using the chosen dissimilarity matrix as distance matrix. This way submissions could be graded in the TSP rank order, with similar submissions being graded together. In combination with the pair-wise comparison tool, this allowed some submissions to be graded within seconds. Besides assessment performance, instructors reported more consistent scores for similar submissions. After grading, the tool facilitated enormously the task of collecting and reporting evidences for plagiarism suspicions. See accompanying video. We plan to formally evaluate and quantify these advantages through a user study. Limitations: the dissimilarity matrices have quadratic cost. We tested up to 200 submissions and got real-time performance after a few seconds of preprocessing. For massive groups, the approach should operate hierarchically, or on a representative subset. Our current prototype focuses on exercises requiring small pieces of code (up to about 100 lines of code). As future work we plan to add software metrics [SVR*16].

Acknowledgements This work has been funded by the Spanish Ministry of Economy and Competitiveness and FEDER Grant TIN2017-88515-C2-1-R.



	ID	Student	TSP rank	Comment
1	000	Marcelle Kealey	000	
2	001	Carey Brandi	016	
3	002	Dione Vaientina	046	
4	003	Arlette Brannon	018	
5	004	Hana Janiuszck	003	
6	005	Deidre Jaymie	122	
7	006	Jeralee Kevyn	002	
8	007	Joice Daph	064	
9	008	Jeni Haines	029	
10	009	Bellanca Sears	065	
11	010	Jewel Post	111	
12	011	Farra Vachil	057	
13	012	Truda Farnham	054	
14	013	Pansie Negris	058	
15	014	Gwynne Bilbe	066	
16	015	Vivia Teddie	004	
17	016	Chantal Sclater	035	
18	017	Melessa Toulon	032	
19	018	Aurelia Magnusson	067	
20	019	Vale Hobbs	116	
21	020	Dee Dee Lunneta	068	
42	041	Carine Aarika	006	

Carey Brandi

```
void MyGLWidget::iniCamera()
{
  angleY = angleX = 0;
  perspectiva = true;
  ra = 1;
  fov = float(3.14159265358979323846 / 3);

  zn = radiEsc;
  zf = 3 * radiEsc;
  left = -15;
  right = 15;
  bottom = -15;
  top = 15;
  projectTransform();
  viewTransform();
}
```

Debera Gastineau

```
void MyGLWidget::iniCamera()
{
  angleY = angleX = 0;
  perspectiva = true;
  ra = 1;
  fov = float(3.14159265358979323846 / 3);
  fovini = fov;
  zn = radiEsc;
  zf = 3 * radiEsc;
  xl = - radiEsc;
  xr = radiEsc;
  yb = - radiEsc;
  yt = radiEsc;
  projectTransform();
  viewTransform();
}
```

Legends

Colors	Links
Added	(f)irst change
Changed	(n)ext change
Deleted	(t)op

Figure 1: *Snapshot of our tool. See accompanying video.*

References

[ACFV18] ANDUJAR C., CHICA A., FAIRÉN M., VINACUA Á.: GL-socket: A CG plugin-based framework for teaching and assessment. In *EG 2018: education papers* (2018), European Association for Computer Graphics (Eurographics), pp. 25–32. 1

[AM05] ALA-MUTKA K. M.: A survey of automated assessment approaches for programming assignments. *Computer science education 15*, 2 (2005), 83–102. 1

[ARVV19] ANDUJAR C., RALUCA VIJULIE C., VINACUA A.: A Parser-based Tool to Assist Instructors in Grading Computer Graphics Assignments. In *Eurographics 2019 - Education Papers* (2019), Tarini M., Galin E., (Eds.), The Eurographics Association. 1, 2

[ASR06] AHTIAINEN A., SURAKKA S., RAHIKAINEN M.: Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. In *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006* (2006), pp. 141–142. 1

[AVV20] ANDUJAR C., VIJULIE C. R., VINACUA A.: Syntactic and semantic analysis for extended feedback on computer graphics assignments. *IEEE Computer Graphics and Applications 40*, 3 (2020), 105–111. 2

[BRG20] BELLI S., RAVENTÓS C. L., GUARDA T.: Plagiarism detection in the classroom: Honesty and trust through the urkund and turnitin software. In *International Conference on Information Technology & Systems* (2020), Springer, pp. 660–668. 1

[CJ11] COSMA G., JOY M.: An approach to source-code plagiarism detection and investigation using latent semantic analysis. *IEEE transactions on computers 61*, 3 (2011), 379–394. 1

[FJA16] FRANCISCO R., JÚNIOR C. P., AMBRÓSIO A. P.: Juiz online no ensino de programação introdutória-uma revisao sistemática da literatura. In *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)* (2016), vol. 27, p. 11. 1

[Gör16] GÖRG T.: Interprocedural PDG-based code clone detection. *Softwaretechnik-Trends 36*, 2 (2016). 1

[HICS80] HUNT III H. B., CONSTABLE R. L., SAHNI S.: On the computational complexity of program scheme equivalence. *SIAM Journal on Computing 9*, 2 (1980), 396–416. 1

[Jam17] JAMIL H. M.: Automated personalized assessment of computational thinking MOOC assignments. In *2017 IEEE 17th International Conference on Advanced Learning Technologies (ICALT)* (July 2017), pp. 261–263. 1

[KCF*19] KUZI S., COPE W., FERGUSON D., GEIGLE C., ZHAI C.: Automatic assessment of complex assignments using topic models. In *Proceedings of the Sixth (2019) ACM Conference on Learning@ Scale* (2019), pp. 1–10. 1

[KLC01] KURNIA A., LIM A., CHEANG B.: Online judge. *Computers & Education 36*, 4 (2001), 299–315. 1

[Kru64] KRUSKAL J. B.: Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika 29*, 1 (1964), 1–27. 2

[LCHY06] LIU C., CHEN C., HAN J., YU P. S.: Gplag: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining* (2006), pp. 872–881. 1

[Mos] Moss - a system for detecting software similarity. `http://theory.stanford.edu/~aiken/moss/`. Accessed: 2020-03-03. 1

[NLM15] NAIK R. R., LANDGE M. B., MAHENDER C. N.: A review on plagiarism detection tools. *International Journal of Computer Applications 125*, 11 (2015). 1

[PGR12] PETIT J., GIMÉNEZ O., ROURA S.: Jutge.org: An educational programming judge. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2012), SIGCSE '12, ACM, pp. 445–450. 1

[PMP*02] PRECHELT L., MALPOHL G., PHILIPPSEN M., ET AL.: Finding plagiarisms among a set of programs with jplag. *J. UCS 8*, 11 (2002), 1016. 1

[RK19] RAGKHITWETSAGUL C., KRINKE J.: Siamese: scalable and incremental code clone search via multiple code representations. *Empirical Software Engineering 24*, 4 (2019), 2236–2284. 1

[SVR*16] SILVA R. R. O. D., VERNIER E. F., RAUBER P. E., COMBA J. L. D., MINGHIM R., TELEA A. C.: Metric Evolution Maps: Multidimensional Attribute-driven Exploration of Software Repositories. In *Vision, Modeling & Visualization* (2016), Hullin M., Stamminger M., Weinkauf T., (Eds.), The Eurographics Association. 2

[WW16] WEBER-WULFF D.: Plagiarism detection software: promises, pitfalls, and practices. *Handbook of academic integrity* (2016), 625–638. 1