Combining Similarity Metrics with Abstract Syntax Trees to Gain Insights into How Students Program

Manuel Freire-Morán[1,*]

[1]Complutense University of Madrid (UCM), Profesor José García Santesmases 9, 28040 Madrid, Spain

Abstract

Computer Science students often have to program exercises to practice and gain insights, to be then submitted for grading and feedback by instructors. The submission and grading process may be partly automated, for instance by requiring electronic submission and running automated tests on answers; but is mostly a laborious and manual process. Data collected by such systems can be of significant use for learning analytics, helping teachers to better understand how their students have attempted to solve exercises.

Using abstract syntax trees and robust similarity detection, we have built a prototype that can label differences between answers sent to an online judge system. The system works best when differences are small, such as when authors fix their code and later submit updated versions. Other promising uses exist, such as automatically labeling commits to version-control systems, or improving plagiarism-detection systems.

Keywords

Software similarity, Abstract syntax trees, Learning analytics

1. Introduction

When learning Computer Science, many subjects include practical exercises that involve programming. Often, students submit these exercises using generic VLEs (Virtual Learning Environments), such as Moodle; other times, programming-specific submission environments are used, which may include automated tests to be run on submissions.

In the authors' institution, several courses make use of the Domjudge[1] system to automatically judge submissions by students. Typically, problem statements include only a sample of the full number of test-cases that will be used, so submissions to a problem are often correct for samples, but incorrect for the full set. It is frequent for students to send multiple answers, each time making changes to their source-code in an attempt to fix it account for the hidden test-cases. Figure 1 illustrates the teacher dashboard for a domjudge installation; note that some students have sent the same exercise over 10 times. Teachers will often only look at the last submission for grading purposes.

Learning Analytics Summer Institute Spain (LASI Spain) 2023, June 29–30, 2023, Madrid, Spain

*Corresponding author.

manuel.freire@fdi.ucm.es (M. Freire-Morán)

0000-0003-4596-3823 (M. Freire-Morán)

[1]https://www.domjudge.org/

RANK	TEAM	SCORE	I2023.02.21	I2023.02.28	I2023.03.07	I2023.03.14	I2023.03.21	I2023.03.28	I2023.04.11	I2023.04.18
1	I17	9	3	1	2	2	5	1	1	1
2	I08	9	2	2	3	1	1	1	1	2
3	I16	8	19	1	29	2	10	1	2	1
4	I05	7	3	1	2	2	2	5	1	
5	I06	7	5	1	6	2	6		3	9
6	I13	6	7	3	3	1	5			1
7	I01	6	11	1	3	4	1	1	1	1
8	I09	5	4	1	5	2	8	1		4
9	I07	5	6	2	3	2		1	1	2
10	I20	5	5	1			4	1	1	
11	I12	5	1	4	3	1	11		5	3

Figure 1: Score table from Domjudge, showing submissions by students (rows) to problems (columns). Numbers in each cell correspond to the number of submissions. Green cells indicate that the student solved the problem (dark green=1st student to solve it), while red indicates that no submissions managed to pass all test-cases.

Our initial goal was to develop a simple system that could provide insights into how students learn to program, by analyzing how their answers changed over time. A very simple approach would be to use something similar to Unix's diff [1] tool on each pair of succesive versions. Indeed, Domjudge can already show diffs between versions of a single file in a given submission. However, requesting these reports from Domjudge requires significant user intervention, as each version of each file of each submission must be individually queried; and *diff* results generally lack context, as the comparison is performed on plain text and not programs, and can therefore be hard to understand. We wanted a tool that would produce readable output more similar to what a human would comment after looking at those differences with full context. Did the student modify a condition within a particular function? Did the student choose an entirely different approach to solve the problem?. This was prompted by previous work in similarity detection in the context of code plagiarism prevention, so we thought it possible to extend existing tools to tackle automated difference labelling. Note that there have been some attempts to incorporate semantical information in diff-like programs[2], but they are certainly not in mainstream use.

The next section describes our approach to making submission history for a single user easier to understand for teachers, by combining existing similarity-detection code with syntax trees for the specific languages used in these exercises. We then describe results of running a prototype of the tool on actual submissions from a Data Structures course. Finally, conclussions and future work outline how the tool could be adapted to facilitate other learning analytics tasks.

[2]For example, https://semanticdiff.com (commercial) appears to provide semantic visual diffs for several languages

2. Proposal

To compare program semantics instead of their textual contents, knowledge of the programming language is required. Programs can be viewed at two very distinct level: at a lower, sintactical level, tokenizing a source-file allows it to be viewed as a sequence of tokens rather than a sequence of characters. At a higher level, programs can be analyzed as trees of related tokens, grouped into declarations, loops, loop conditions, methods, and so on. These trees are often termed abstract syntax trees (ASTs), and the process of converting source-code into trees that retain their semantics is termed parsing. ASTs can be used not only for compiling or interpreting source-code, but also to analyze programs and transform program constructs.

Source-code parsing is a well-understood topic in computer science. Given a suitable grammar indicating how to parse source-code into ASTs, there are multiple well-known parser-generation programs that can generate the corresponding parsers for those grammars. One such program is Antlr 4 [2], by Terence Parr, which is well-documented, available as open-source, and has an active community which has contributed open-source grammars for many programming languages[3].

Software similarity is often used in academic contexts to locate, and thus deter, plagiarism. Students that know that there is a high chance that plagiarism will get caught will think twice before presenting work of others as theirs. There is substantial literature on the subject if plagiarism, with reviews such as Karnalim's [3] identifying a wide variety of techniques, including string-matching, token-counting, metric-based, or even structural analysis of parse-trees and call-graphs. There is also relevant literature in the field of code-clone analysis, used to improve refactoring in production code-bases. In a recent review on code-clone detection, Ain et Al. [4] identify essentially the same techniques used for plagiarism detection as useful for code-clone analysis.

A simple and robust technique to detect code similarity, not present in Karnalim's review, relies on analyzing the entropy between sources: similar source-code will, when compressed by a high-quality compressor, compress better than totally unrelated code. For a more in-depth analysis of NCD (Normalized Compression Distance), see [5]. NCD itself can be applied to any sequence, and not just source-code. For example, it has been used in both image comparisons and to generate phylogenetic trees for genomics research. When applied to source-code, a significant reduction in noise can be achieved by tokenizing the source-code first, so that whitespace, non-semantically relevant indentation, identifier names, or comments no longer contribute to distance.

We have built the prototype described in this work on top of AC2[4], an existing open-source plagiarism detector with robust similarity detection [6]. AC2 uses NCD to determine the similarity between two submissions; files from each submission are concatenated and tokenized before comparison. Util this work, AC2 used Antlr 4 grammars only for tokenization.

In this work, we describe an extension to AC2 which uses parse trees to annotate differences between subsequent versions of a given submission. After building the ASTs of a submission, it is possible to compare semantically-significant parts of submissions instead of simple concate-

[3] As of june 2023, there are over 250 grammars at https://github.com/antlr/grammars-v4.
[4] AC2 is available at https://github.com/manuel-freire/ac2

nations of files. The smallest segments of program ASTs where changes are located can be used to generate semantically-significant labels for different versions of a single submission. For example, if a single conditional has changed, then the label describing that version increment could be

```
Condition changed in Tree.cpp::find_smallest:
  - if (i <= size) {
  + if (i < size) {
```

2.1. Prototype

The prototype first downloads submissions from a Domjudge server, which results in a set of folders, one per problem (columns in Fig. 1). Inside each problem-folder there is another folder for each student that has attempted to solve it; and inside each student folder, there is an additional per-submission folder. The prototype then uses a parse-tree aware version of AC2 to compare, within each student-folder, each submission to the next one. Output describing these differences is generated to a text-file and placed in the student-folder.

To compare two submissions, both are first parsed into ASTs, which must then be aligned to locate exactly what each change corresponds to. The problem of matching trees has been examined, for example, in [7]. We use a very simple approach, starting with a breadth-first search on both trees that marks each tree node as either identical or at least similar to nodes on the other tree. To easily detect equal nodes, a textual representation of the tokens at each tree level is hashed, at a cost of $O(n \cdot d)$, where n is the total number of tokens in a program and d is the depth of its parse tree. There are significant advantages in reducing d, by selecting a subset of possible parse-rules as significant for alignment, and skipping all others. When an identical node is found via hash-comparison, there is no need to recurse further into the tree, as all subtrees can then be expected to also result in exact matches. For subtrees with no identical counterpart, all candidate subtrees are analyzed for similarity using NCD, with only the most-similar opposing subtree retained for alignment.

The result of aligning two trees A and B is therefore, for each subtree of A, a corresponding subtree of B that is either identical or considered as the most similar available. The last step of the process simply generates a textual descripiton of the differences between non-identical alignments. Additionally, the annotation process must also describe elements in B that have not been matched to A as being newly added; and a similar consideration should be made for elements in A that have entirely dissapeared from B.

The following pseudocode illustrates these steps:

```
let astA = parse(submissionA);
let astB = parse(submissionB);
let hashesA = recursiveHash(astA);
let hashesB = recursiveHash(astB);
let alignmentAB = align(astA, astB, hashesA, hashesB);
annotate(alignmentAB);
```

3. Case study

We have tested the prototype annotation tool on a set of submissions by 17 distinct students for 10 problems (partly illustrated in Fig. 1), with a total of 445 submissions from those students attempting to solve the problems, from which a total of 109 were correct and the remainder returned either compiler-error (73), run-time error (83), time-limit exceeded (27), wrong-answer (146), or no-output (7). The students were enrolled in a course on Data Structures in a university in Madrid, Spain.

The prototype annotated the 449 submissions (with 1919 C++ files in total) in under 15 seconds; performance can certainly be improved, for example by caching parse-trees for files that were part of the problem statements, such as implementations of common abstract data types. For each set of sequential submissions by a student, the tool generates a file that describes what changed from each submission to the next. A sample file is included below (student I04, problem 473):

```
[1] :: s77161_280223_1550_I04_compiler-error -> s77162_280223_1550_I04_compiler-error
  New @BinTree.h::: (10171-char patch)

[2] :: s77162_280223_1550_I04_compiler-error -> s77163_280223_1551_I04_compiler-error
  New @Exceptions.h::: (1581-char patch)

[3] :: s77163_280223_1551_I04_compiler-error -> s77166_280223_1552_I04_correct
  New @Queue.h::: (4857-char patch)
  New @List.h::: (11732-char patch)
```

In the above file, the student forgot to include several dependencies, leading to compiler errors. In the 4th version, the final 2 dependencies were included, leading to a veredict of *correct* by the online judge. Large changes (currently limited to 1024 characters), such as the contents of new files, are hidden by default. In the next example (student I08, problem 478), each change describes the method of the class that encompasses the relevant code, providing important context for a grader familiar with the template used by students:

```
[1] :: s78256_070323_1552_I08_wrong-answer -> s78258_070323_1554_I08_wrong-answer
  Modified @template.cpp::Bank::transfer(int line, string source, string target, int quantity):
@@ -25,5 +25,5 @@
         // Iterate map in alphabetical order and output money in non-empty, non-* accounts
         for (auto it = accounts.cbegin(); it != accounts.end(); it++) {
-            if ((it.key() != "*") && (it.value() > 0)) {
+            if (!(it.key() == "*") && (it.value() > 0)) {
                 cout << it.key() << " " << it.value() << endl;
             }

[2] :: s78258_070323_1554_I08_wrong-answer -> s78259_070323_1555_I08_correct
  Modified @template.cpp::Bank::transfer(int line, string source, string target, int quantity):
@@ -13,5 +13,5 @@
         else {
             if (accounts[source] < quantity) {
                 cout << line << " " << source << " is "
-                     << accounts[source] - quantity
+                     << accounts[source] - quantity
                      << " short of " << quantity << endl;
                 cout << line << " " << source << " is "
```

```
+                   << quantity - accounts[source]
                    << " short of " << quantity << endl;
            }
            else {
```

While the prototype generates readable descriptions of changes in submissions, it is currently experimental code, and will require significant work before being made available for other teachers in the author's institution that use Domjudge for exercises. Each year, over 300 students from the author's institution enroll in subjects that make heavy use Domjudge. Their submissions mostly remain on the server hosting the judge, and we plan to enroll their teachers to test future versions of our tool.

From the point of view of user experience, having to read a large numbers of text files is still work, even if each file manages to quickly describe how several submissions have evolved over time. Additionally, using a fixed cutoff-point to hide details, while often useful, can also hide important details – presenting the information interactively on demand, say within a web application, would result in a better user experience than having to look up additional information by pointing an IDE to the actual source files.

4. Conclussions and future work

The prototype used for the above case-study does provide readable overviews of what has changed in submissions to a given problem over time. This makes it possible for teachers to gain quick overviews of large sets of (similar) submissions from a single author attempting to solve a single problem, opening a large number of collected submissions to potential analysis to better understand how students learn programming. However, the extent to which these overviews are effective at their goal of making it easier for teachers to understand how their students learn can only be measured by further experiments.

Beyond the application presented here, we consider that there is significant potential for tools that can automatically annotate differences in code submissions. The following additional use-cases come to mind:

1. Automated labelling of changes could be used to auto-generate descriptions of changes for version-control systems. Knowing which functions and data-structures were changed by a commit can be much more useful than only knowing the files that were modified.
2. Plagiarism detection can greatly benefit from automatic alignment. For example, when looking at submissions from two students, it is useful to show which code from student *A* best corresponds to a given fragment of code by student *B*. This could also be used to describe the steps that *B* could have taken to cover their tracks.
3. If code snapshots are collected periodically while students are solving a coding exercise, it may be possible to generate interesting insights into how students learn to code. In [8], Budiman and Karnalim describe a VS Code plugin that periodically collects code from students and sends it to a server to be checked for plagiarism. We believe that a snapshot approach can also be used to better understand how students learn to code in many other contexts, such as block-based computational-thinking games.

There also remain multiple hurdles to this vision of semantic labelling of code differences. The general problem of finding the most human-readable description of the differences between two programs is not solvable by traditional programs. Any approach will have to rely on reasonable heuristics for common cases, which however can be extremely effective if and when their preconditions are met. For example, in this paper we rely on an expectation for small amounts of changes between subsequent versions of submissions. In many cases, this is indeed the case, and short and readable descriptions can be succesfully generated. However developing good heuristics can take time, and heuristics are, by nature, fragile.

Future work will include an improved version of the tool, and larger experiments testing both its effectiveness and usability.

Acknowledgments

Co-funded by the Ministry of Education (PID2020-119620RB-I00) and by the Telefónica-Complutense Chair on Digital Education and Serious Games.

References

[1] J. W. Hunt, M. D. McIlroy, An Algorithm for Differential File Comparison, Technical Report CSTR 41, Bell Laboratories, Murray Hill, NJ, 1976.

[2] T. Parr, The definitive ANTLR 4 reference, The Pragmatic Bookshelf, 2013.

[3] O. Karnalim, Simon, W. Chivers, Similarity detection techniques for academic source code plagiarism and collusion: A review, in: 2019 IEEE International Conference on Engineering, Technology and Education (TALE), IEEE, 2019, pp. 1–8. doi:10.1109/tale48000.2019.9225953.

[4] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, B. Maqbool, A systematic review on code clone detection, IEEE Access 7 (2019) 86121–86144. doi:10.1109/ACCESS.2019.2918202.

[5] M. Cebrián, M. Alfonseca, A. Ortega, The normalized compression distance is resistant to noise, IEEE Transactions on Information Theory 53 (2007) 1895–1900. doi:10.1109/tit.2007.894669.

[6] M. Freire, Visualizing program similarity in the ac plagiarism detection system, in: Proceedings of the working conference on Advanced visual interfaces, 2008, pp. 404–407. doi:10.1145/1385569.1385644.

[7] R. Kumar, J. O. Talton, S. Ahmad, T. Roughgarden, S. R. Klemmer, Flexible tree matching, in: Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Three, IJCAI'11, AAAI Press, 2011, p. 2674–2679. doi:10.5555/2283696.2283841.

[8] A. E. Budiman, O. Karnalim, Automated hints generation for investigating source code plagiarism and identifying the culprits on in-class individual programming assessment, Computers 8 (2019). URL: https://www.mdpi.com/2073-431X/8/1/11. doi:10.3390/computers8010011.