How do various plagiarism detection tools differ in their detection results when applied to java/spring coding exercises

Dennis Goßler Dennis Wäckerle

November 24, 2022

Abstract

Contents

1 Introduction

In order to teach students how to program some modules like "Softwaretechnik 2" at the Cologne University of Applied Sciences offer mandatory coding exercises to the students. However, there are attempts to cheat during those exercises and to copy from other students. These plagiarism attempts have to be found manually which can be a very involved task when there are over 100 submissions. In order to simplify this process a plagiarism detection tool shall be used.

There are multiple different tools available for detecting plagiarism. This paper attempts to shine a light on the differences of the results of those tools when applied to Java coding exercises which use the Spring framework. There were four tools identified which could be used to detect plagiarism attempts. These tools and their algorithms will be presented, including an overview of the different methods of plagiarism detection.

A software experiment to evaluate these 4 tools will also be presented the results of which will be available in a part 2 of this paper.

2 How does source code plagiarism detection work

Plagiarism detection has become indispensable in the scientific field. Computer programs and algorithms can as well be copied and modified. Therefore, the question arises how to detect and recognize plagiarism in programming code. Of course, it is possible to detect patterns in different projects by hand. This process becomes unfortunately very labor-intensive the more projects must be compared to each other. The following chapter lists therefore a few methods how such an algorithm could be designed.

2.1 Automatic source code plagiarism detection

There are several ways to design an algorithm that can detect source-code similarities. The following list classifies these different approaches:

- "Strings - look for exact textual matches of segments, for instance five-word runs. Fast, but can be confused by renaming identifiers".
- "Tokens - as with strings, but using a lexer to convert the program into tokens first. This discards whitespace, comments, and identifier names, making the system more robust to simple text replacements. Most academic plagiarism detection systems work at this level, using different algorithms to measure the similarity between token sequences".
- "Parse Trees - build and compare parse trees. This allows higher-level similarities to be detected. For instance, tree comparison can normalize conditional statements, and detect equivalent constructs as similar to each other".
- "Program Dependency Graphs (PDGs) - a PDG captures the actual flow of control in a program, and allows much higher-level equivalences to be located, at a greater expense in complexity and calculation time".
- "Metrics - metrics capture 'scores' of code segments according to certain criteria; for instance, "the number of loops and conditionals", or "the number of different variables used". Metrics are simple to calculate and can be compared quickly, but can also lead to false positives: two fragments with the same scores on a set of metrics may do entirely different things ".
- "Hybrid approaches - for instance, parse trees + suffix trees can combine the detection capability of parse trees with the speed afforded by suffix trees, a type of string-matching data structure".

Source: (Ali et al., 2011)[p. 5]

In the following chapter 4 [Evaluation of the different Tools] the selected algorithms will be analyzed in a bit more detail and the respective special features will be discussed.

2.2 Evaluation of the results from the automated method

Plagiarism detection tools are not perfect, so a human should go other the results, and it should be checked if the clams are valid. The author of the Moss algorithm (Aiken, 2021) notes that

"In particular, it is a misuse of Moss to rely solely on the similarity scores. These scores are useful for judging the relative amount of matching between different pairs of programs and for more easily seeing which pairs of programs stick out with unusual amounts of matching. But the scores are certainly not a proof of plagiarism. Someone must still look at the code."

3 Use Case and Software Experiment

There will be 4 tools analyzed in order to ascertain whether they would be a good fit for a specific use case. The tools will be analyzed by perfoming a software experiment in which the tools are expected to display certain criteria.

3.1 Use Case

The "Softwaretechnik 2"(ST2) module at the Cologne University of Applied Sciences requires the students to complete multiple Java coding exercises which use the Spring framework. The exercises use an automated pipeline called Divekit for the creation of the individual repository which contains personalized task for the individual students and the evaluation of these tasks. The personalized tasks differ from each other by naming scheme only and the evaluation pipeline requires that these names are correct. This is an attempt to curb plagiarism, however despite this attempt plagiarism attempts still occur and currently need to be identified manually which proofs difficult with other 100 submissions. Due to this reason it should be evaluated whether an automated plagiarism detection tool can be used to improve this process.

3.2 Software Experiment

In order to find the most suiting tool for the outlined use case a software experiment has been formulated which will be formed. During this experiment the tools will have to analyze already known plagiarisms. After that they have to analyze the real submissions for the 5 different milestones of the ST2 modules. While there were some plagiarism attempts discovered for milestone 0 and 4 there are no known attempts for the other milestones. So, in order to be able to properly evaluate these tools plagiarisms are to be created and inserted into the evaluation process of the tools.

These plagiarisms are to be created systematically with certain attacks in order to further test the robustness of the tool. These attacks are as follows

1. Rename variables and methods
2. Change control structure, e.g. for-loops to while-loops
3. Insert duplicated code
4. Extract inline methods
5. Move methods inline
6. Move methods to new classes

3.3 Criteria for Evaluation

In order to decide on a tool we must set some criteria by which we are going to select the tool. These are:

- The tool must be able to analyze Java 11 programs which use Spring

- The tool must be able to detect the already found plagiarisms
- The tool must be able to evaluate over 100 submissions in a reasonable time
- The tool must be able to be integrated into the Divekit pipeline or workflow

4 Evaluation of the different Tools

This chapter tests and explains four different plagiarism detection tools. These tools are Moss, JPlag, Plaggie and AC2. These tools often name each other as good alternatives and are one of the most common plagiarism detection programs. In addition, this chapter discusses the different ways in which each algorithm detects plagiarism and how the data from the programs is analyzed and presented.

	C	C++	C#	Java	Kotlin	Python	PHP	VB.net	Javascript	Is expandable?
MOSS	Yes	Yes	Yes	Yes	No	Yes	No	Yes	Yes	No
JPlag	No	Yes	Yes	Yes	Yes	Yes	No	No	No	Yes
Plaggie	No	No	No	Yes	No	No	No	No	No	Yes[1]
AC2	Yes	Yes	No	Yes	No	Yes	Yes	No	No	Yes

Table 1 [The native supported programming languages for each plagiarism detection algorithm]

Table 1 shows which programming language is supported by the corresponding plagiarism detection tool. The last column shows whether the user can add more programming languages to the plagiarism detection algorithm.

4.1 MOSS

"Moss (for a Measure Of Software Similarity) is an automatic system for determining the similarity of programs. To date, the main application of Moss has been in detecting plagiarism in programming classes. Since its development in 1994, Moss has been very effective in this role." (Aiken, 2021)

Moss clams to be one of the best cheating detection algorithms.[2] Moreover, moss supports the largest native programming languages variety[3], but no other languages can be added. Moss supports the following languages: C, C++, Java, C#, Python, Visual Basic, Javascript, FORTRAN, ML, Haskell, Lisp, Scheme, Pascal, Modula2, Ada, Perl, TCL, Matlab, VHDL, Verilog, Spice, MIPS assembly, a8086 assembly, a8086 assembly and HCL2.

A unique feature of Moss is that it "is being provided as an Internet service. The service has been designed to be very easy to use[...]" (Aiken, 2021). In order to use this internet service, a Perl script must be executed which passes the selected programming files to the server. "In response to a query the Moss server produces HTML pages listing pairs of programs with similar code. Moss also highlights individual passages in programs that appear the same, making it easy to quickly compare the files. Finally, Moss can automatically eliminate matches to code that one expects to be shared (e.g., libraries or instructor-supplied code), thereby eliminating false positives that arise from legitimate sharing of code." (Aiken, 2021)

A disadvantage of Moss is that since Nov 13, 2022 there is a transmission limit of 100 submissions per day per user. This could cause problems if one wants to integrate Moss into an automated process.

[1] Plaggie is open source, therefore a custom tokenizer for different languages can be implemented.

[2] "The algorithm behind moss is a significant improvement over other cheating detection algorithms (at least, over those known to us)." (Aiken, 2021)

[3] Of the four plagiarism detection programs listed in this paper

4.1.1 Moss comparison algorithm

To evaluate the uploaded programming files "Moss currently uses robust winnowing, which is more efficient and scalable [...] than previous algorithms we have tried. There are a few issues involved in making such a system work well in practice. For this application, positional information (document and line number) is stored with each selected fingerprint. The first step builds an index mapping fingerprints to locations for all documents, much like the inverted index built by search engines mapping words to positions in documents. In the second step, each document is fingerprinted a second time and the selected fingerprints are looked up in the index; this gives the list of all matching fingerprints for each document.

Now the list of matching fingerprints for a document d may contain fingerprints from many different documents d1,d2,.... In the next step, the list of matching fingerprints for each document d is sorted by document and the matches for each pair of documents (d,d1),(d,d2),... is formed. Matches between documents are rank-ordered by size (number of fingerprints) and the largest matches are reported to the user." (Schleimer et al., 2003)[page 9 section 5.2]

4.1.2 Preliminary results

In order to determine the performance of the Moss program, two test are conducted (See Table 2 and table 3). For each test the related documents are uploaded to the moss server. After a while the server sends back a link to an HTML website, that is showing a table of the uploaded projects, the count of the matching lines and the similarity in percent.

File 1	File 2	Lines Matched
ST2M0-Geber1(81%)	ST2M0-Nehmer1(73%)	311
ST2M0-Geber2(62%)	ST2M0-Nehmer2(66%)	276
ST2M0-Geber1(14%)	ST2M0-Geber2(12%)	81
ST2M0-Geber2(11%)	ST2M0-Nehmer1(12%)	76
ST2M0-Nehmer1(9%)	ST2M0-Nehmer2(9%)	68
ST2M0-Geber1(9%)	ST2M0-Nehmer2(8%)	62

Table 2 [Table of Milestone [0] distance between projects]

The Table 2 shows that ST2M0-Geber1 and ST2M0-Nehmer1 are very similar. These two projects have 311 matching lines of code and have a max similarity of 81%. ST2M0-Geber2 and ST2M0-Nehmer2 are also they similar. They have a max similarity of 66%.

In the second test (Table 3), it is clearly visible that ST2M4-Geber1, ST2M4-Nehmer1 and ST2M4-Nehmer2 are in a triangular correlation. This is due to the fact that the three projects each have a high similarity percentage to each other.

4.2 JPlag

JPlag is a plagiarism detection tool, which was developed by in 2000 at the University of Karlsruhe to help with the detection of plagiarized coding exercises. At that the tool was only available as a WWW service and could analyze C, C++. Scheme and Java programs (Perchelt et al., 2000, p. 4). Currently, the tool is available as an open source application which can be run locally and can be used with a CLI or a Java API. JPlag also currently supports 12

File 1	File 2	Lines Matched
ST2M4-Geber1(54%)	ST2M4-Nehmer2(57%)	938
ST2M4-Geber1(52%)	ST2M4-Nehmer1(52%)	908
ST2M4-Nehmer1(51%)	ST2M4-Nehmer2(54%)	884
ST2M4-NoPlag2(13%)	ST2M4-Nehmer2(17%)	372
ST2M4-NoPlag2(12%)	ST2M4-Nehmer1(15%)	363
ST2M4-Geber1(14%)	ST2M4-NoPlag2(11%)	333
ST2M4-NoPlag2(8%)	ST2M4-NoPlag1(6%)	254
ST2M4-NoPlag1(6%)	ST2M4-Nehmer1(9%)	253
ST2M4-Geber1(9%)	ST2M4-NoPlag1(6%)	238
ST2M4-NoPlag1(5%)	ST2M4-Nehmer2(9%)	239

Table 3 [Table of Milestone [4] distance between projects]

programming languages including the original four languages and more modern languages such as Kotlin ("JPlag - Detecting Software Plagiarism", 2022, Supported Languages) while also allowing to add new languages (Sağlam, 2022).

4.2.1 JPlag's comparison algorithm

JPlag's algorithm is split into two parts the tokenizing and the comparison of the token strings. During the tokenization process all programs are parsed and converted into token strings. In the second phase all token strings are compared in pairs to determine their similarity. This comparison uses a specially optimized version of the Greedy String Tiling algorithm. This algorithm tries to cover one token string with substrings of the other string. The percentage of the covered token strings is the similarity between the two programs (Perchelt et al., 2000, p. 10).

Tokenization The tokenization process is the only language dependent process of the JPlag algorithm (Perchelt et al., 2000, p. 10). The extracted tokens represent syntactic elements of the language like statements or control structures (Sağlam, 2022, How are submissions represented? — Notion of Token). During the parsing process each file is parsed with the result being a set of abstract syntax trees(AST) for each submission. Each AST is traversed depth first with nodes representing grammatical units of the language. During the traversal when entering and exiting a node a token can be created that match the type of the node and will then be added to the current list of tokens. There are block type nodes which can represent classes, if expressions or other elements of the language which have a corresponding beginning and end. When creating tokens from those nodes each have a corresponding "BEGIN" and "END" token. The token list should always have a pair of matching "BEGIN" and "END" tokens (Sağlam, 2022, How does the transformation work?).

Comparing token strings JPlag's comparison algorithm is essentially just the greedy string tiling algorithm for the comparison of two strings, however it is differently optimized to improve its runtime(Perchelt et al., 2000, p. 5). The goal of the algorithm is to find a set of substrings which are not only the same but also satisfy these three rules:

1. "Any token of A may only be matched with exactly one token from B."

2. "Substrings are to be found independent of their position in the string."
3. "Long substring matches are preferred over short ones[...]"

(Perchelt et al., 2000, p. 11)

These rule have some consequences. The first rule doesn't allow the matching of code that have been duplicated while the second rule makes the reordering of the source code not viable. Furthermore, the third rule is introduced since short matches are more likely to be spurious (Perchelt et al., 2000, p. 11).

When these rule 3 is applied sequentially then a greedy algorithm consisting of two phases will be created.

In phase 1 the longest contiguous match in the two strings is searched. This is done by 3 nested loops. The outer loop iterates over all extracted tokens in string A, the second loop compares token T from string A with every token in string B. If two tokens are identical then the innermost loop extends this match as far as possible (Perchelt et al., 2000, p. 11). These matches are called tiles which are permanent and unique (Wise, 1993, p. 3).

In the second phase all matches of the maximal length are marked, when this happens all matched tokens are also marked and can't be used for any further matches. This satisfies the first rule. These two phases will be repeated until not further matches are found (Perchelt et al., 2000, p. 11). Also, of importance is the MinimumMatchLength, which defines the length at which maximal matches and any tile below the maximum matches are ignored. The smallest possible value for MinimumMatchLength is 1. However, in general this value should be higher than 1 since at that length it is unlikely that a match will be significant (Wise, 1993, p. 3)

JPlag only considers two strings to match 100% when the shorter string is completely matched by the longer string. The logic being that the longer string represents a program which has been entirely copied and then extended. The similarity between two programs can be calculated with the following formula.

$$sim(A,B) = \frac{2 + coverage(tiles)}{|A| + |B|}$$

$$coverage(tiles) = \sum_{match(a,b,length) \in tiles} length$$

(Perchelt et al., 2000, p. 13).

GST has a worst-case complexity of $O(n^3)$(Wise, 1993, p. 5) and a best-case complexity of $O(n^2)$(Perchelt et al., 2000, p. 14). The worst-case complexity can't be improved the average-case complexity is improved to $O(n)$ by implementing an idea from the Karp-Rabin algorithm[4] (2000, p. 14). The modified algorithm computes the hash values for all substrings of length MML in string A and B. Then each value from string A is compared with each value from string B. String A is always chosen to be shorter than B because string A will always be

[4]The Karp-Rabin algorithm is an algorithm which is used to find an string pattern in a text(Karp & Rabin, 1987).

traversed completely while string B will be accessed through a hash table, which will further decrease the complexity. If two hash values are the same then a potential possible match has been found beginning at that token where the values match. The possible match is then verified by comparing the substrings token by token. The modified algorithm doesn't start the comparison from the beginning but rather from the back after the maxmatch tokens since all matches before that are not interesting for the algorithm. The algorithms also attempts to simultaneously expand the match further (Perchelt et al., 2000, p. 14).

4.2.2 Preliminary results

JPlag displays its result in a list with all possible combinations of all submissions while also showing the amount of pairs which are in a certain similarity range. The different pairs can be further investigated allowing the user to see the specific matched code.

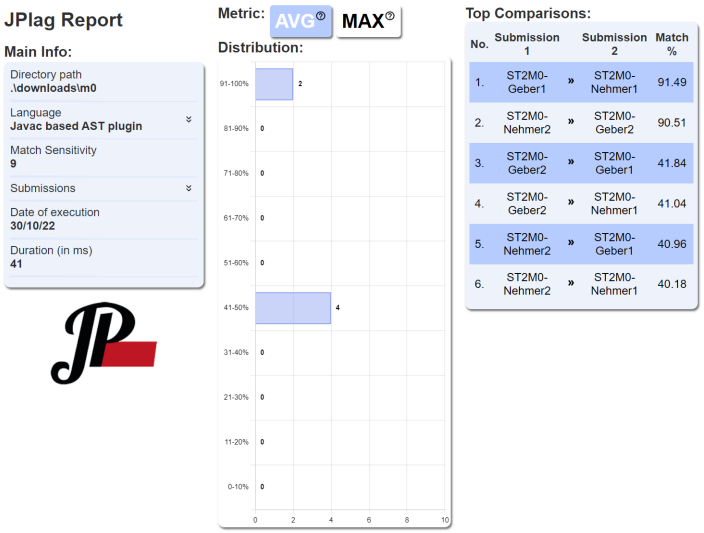

Figure 1 The JPlag overview showing the similarities of m0 code

When using JPlag to check the similarity of known plagiarized code it had similarity value for the plagiarized m0 code of 91.49& and 90.51% while the none plagiarized code had similarity of around 40%. The results for m4 were somewhat less clear since they had a similarity value of 84.77% and 76.2%. Which should still lead to an investigation of the submitted code.

Submission 1	Submission 2	Similarity	Similarity with base code
ST2M0-Geber1	ST2M0-Nehmer1	91.49%	88.11%
ST2M0-Nehmer2	ST2M0-Geber2	90.51%	86.11%
ST2M0-Geber2	ST2M0-Geber1	41.84%	16.34%
ST2M0-Geber2	ST2M0-Nehmer1	41.04%	16.21%
ST2M0-Nehmer2	ST2M0-Geber1	40.96%	14.61%
ST2M0-Nehmer2	ST2M0-Nehmer1	40.18%	14.56%

Table 4 [Table of Milestone [0] showing the similarity between submissions]

It should also be noted that no files were excluded and that there was no base code passed to JPlag, which explains the high similarity between unrelated programs as all submissions for a milestone use the same base code. When excluding common base code then the results for m0 become significantly more clear with the known plagiarism having a similarity value between

86 and 89 percent while the none plagiarized submissions have a value between 14 and 17 percent.

Looking at m4 with the base code excluded still shows a less clear image. With the known plagiarized have a similarity value which is lower by 9 to 7 percentage points and with the none plagiarized submissions having a similarity value which lower by 18 to 16 percentage points. All of which are considerably smaller percentage point drops then those observed in m0. It is not clear why exactly the similarity values of m4 are closer to each other in m4 and why the difference between them doesn't grow that much when excluding the base code in comparison to m0.

Submission 1	Submission 2	Similarity	Similarity with base code
ST2M4-Nehmer2	ST2M4-Geber	84.47%	77.97%
ST2M4-Nehmer2	ST2M4-Nehmer1	77.48%	68.30%
ST2M4-Geber	ST2M4-Nehmer1	76.20%	67.42%
ST2M4-Nehmer2	ST2M4-NoPlag2	49.00%	32.70%
ST2M4-Geber	ST2M4-NoPlag2	47.20%	30.97%
ST2M4-Nehmer1	ST2M4-NoPlag2	46.16%	29.90%
ST2M4-Nehmer1	ST2M4-NoPlag1	41.82%	25.33%
ST2M4-Nehmer2	ST2M4-NoPlag1	40.55%	22.78%
ST2M4-Geber	ST2M4-NoPlag1	39.99%	22.69%
ST2M4-NoPlag2	ST2M4-NoPlag1	36.40%	20.15%

Table 5 [Table of Milestone [4] showing the similarity between submissions]

4.3 Plaggie

Plaggie was developed by Aleksi Ahtiainen, Sami Surakka and MIkko Rahikainen from the Helsinki University of Technology in 2006 in order to detect cheating in Java exercises. It was written in Java 5 and can analyze Java 5 applications (Ahtiainen, 2006, 3. Current list of features). With its most important contribution at the time being the only open source plagiarism tool that could be run locally (Ahtianien et al., 2006, p. 141).

4.3.1 The Algorithm

Plaggie uses CUP as a parser which is a LALR parser[5] that implements standard LALR(1)[6] parser generation ("CUP", n.d.). The algorithm to compare the tokens extracted by CUP is GST which has been extended to exclude existing code but doesn't feature any other optimizations (Ahtiainen, 2006, 4. Algorithm used).

Plaggie display the similarity of individual files and entire submissions. With the similarity of two files being the percentage of the matched tokens in comparison to the total number of tokens in the file. The similarity of the submission A is the average of the best match of each file with the submission similarity of submission B being the average of the corresponding similarity values in submission B (2006, 4. Algorithm used).

4.3.2 Preliminary results

We didn't manage to run Plaggie as it hasn't been updated since 2006 and we ran a newer version of Java which caused compilation error. However, based on the fact that Plaggie only supports Java 1.5 and that JPlag has since been made open source Plaggie would not be our tool of choice since there are more modern tools which are being actively maintained and have support for more modern programming languages.

There are some know successful and unsuccessful attacks identified by Plaggie's developers. With the unsuccessful attacks being:

- "Changing the comments"
- "Changing the indentation"
- "Method and variable name changes"
- "Renaming the classes"

(Ahtiainen, 2006, Known successful attacks)

and the unsuccessful attacks being:

- "Moving inline code to separate methods and vice versa"
- "Inclusion of redundant program code"

[5] LALR stand for Look Ahead, Left to Right, Rightmost derivation. Which is a bottom-up parser that analyses the text from left to right and always rewrites the rightmost nonterminal (Mogensen, 2017).

[6] The (1) refers to one-token lookahead. Which means that only the next input symbol is considered (Mogensen, 2017).

- "Changing the order of if-else blocks and case-blocks"

(Ahtiainen, 2006, Known unsuccessful attacks)

4.4 AC2

"AC is a source code plagiarism detection tool. It aids instructors and graders to detect plagiarism within a group of assignments written in languages such as C, C++, Java, PHP, XML, Python, ECMAScript, Pascal or VHDL (plaintext works too, but is less precise). AC incorporates multiple similarity detection algorithms found in the scientific literature, and allows their results to be visualized graphically." (Freire, 2022a) The first version of "AC was born in the Escuela Politécnica Superior of the Universidad Autónoma de Madrid to deter and detect source-code plagiarism in programming assignments". (Freire, 2022a)

2016 AC version 2 was released. In the version 2.0 the developers switched to Maven and Antr4[7]. The developers of AC also moved their git repository to GitHub an internet hosting service for software development and version control.

AC2 lists some advantages over the other plagiarism detection tools on their GitHub page. In this regard, it is

- "local, and does not require sending data to remote servers (not the case of Moss)local, and does not require sending data to remote servers (not the case of Moss)"
- "robust, using Normalized Compression Distance as its main measure-of-similarity, but with the possibility of integrating other, additional measures to gain better pictures of what is going on (JPlag, Moss and Plaggie have hard-coded analyses, mostly based on sub-string matching after tokenization)."
- "heavy on information visualization. AC will not provide "percentage of copy"; instead, it will create graphical representations of the degree of similarity between student submissions within a group, so instructors can build their own explanations regarding what really happened (see here and here for papers on AC's visualizations)."

(Freire, 2022a)

4.4.1 AC2 comparison algorithm

In order for the AC algorithm to check files for equality, it allows adding folders or zip-files into it. Two filters can be applied to these files. The first filter selects the submissions that will be examined, and the second filter is determining which files are compared to each other.

For example student s1 has in his subfolder 'a.c' and student s2 has in his folder 'aa.c'. So the first filter should determine that each submission starts with the letter 's'. The second filter would look for files that ended with the string '.c'.

[7] "In computer-based language recognition, ANTLR (pronounced antler), or ANother Tool for Language Recognition, is a parser generator that uses LL(*) for parsing. ANTLR is the successor to the Purdue Compiler Construction Tool Set (PCCTS), first developed in 1989, and is under active development."(Wikipedia contributors, 2022)

When all submission and files are correctly grouped together the analysis part can start. "The default analysis (Zip NCD), uses normalized compression distance (NCD) to detect redundancy (= similarity) between submissions. If AC2 knows the programming language of its source files, they will be tokenized prior to comparison. This will allow AC2 to ignore comments, whitespace and exact identifiers, so that simply renaming variables or altering formatting will not affect similarity calculations." (Freire, 2022b)

NCD (normalized compression distance) NCD (normalized compression distance) is a method to determine the similarity between two objects. These two objects can be of any type e.g. text, email, code, ... To determine how equal two objects are, NCD checks how easy it is to transform the two objects into each other.

Tokenization If AC2 can recognize the programming language it will use a lexer and a parser to tokenize the text file. "AC2 uses Antlr4[7] grammars to generate lexers (= tokenizers) and parsers for languages." (Freire, 2022b) Therefore it is relatively easy to add support for a new programming language.

4.4.2 Preliminary results

In order to present the advantages of AC2, several projects were compared using the AC2 algorithm. As explained in section 4.4.1 AC2 comparison algorithm, the filters were used to select and group the submissions. After passing the project files, the algorithm analyzes them and creates an output window. The output that AC2 generates is mainly visual based. So Ac "will create graphical representations of the degree of similarity between student submissions within a group". (Freire, 2022a). See figure 2 and figure 3.

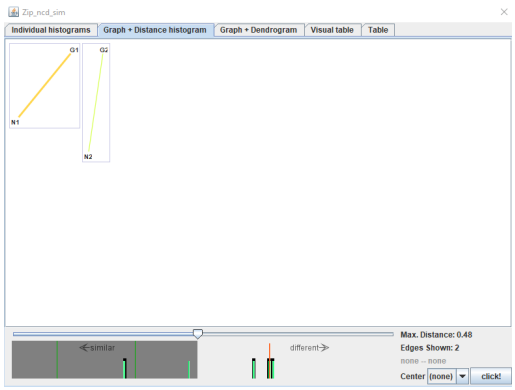

Figure 2 The graphical output of AC2 from milestone [0]

Figure 2 shows milestone 0 and how the different projects have copied from each other. This suggests that project N1 has maybe copied its code form G2 and N2 from G2.

Figure 3 illustrates that G1, N1 and N2 have probably copied from each other, since a correlation exists between the 3 projects. This again illustrates the advantage of AC2 over the other plagiarism detection programs, since it is easier to detect multiple correlations between projects.

Table 6 and table 7 show how far apart the individual projects are. This distinguishes AC2

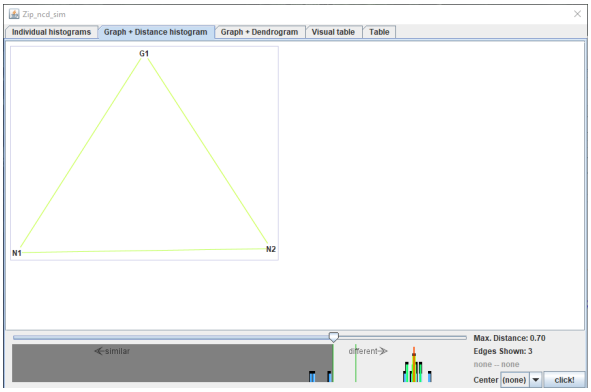

Figure 3 The graphical output of AC2 from milestone [4]

One	The other	Distance
ST2M0-Geber1	ST2M0-Nehmer1	0.2926
ST2M0-Geber2	ST2M0-Nehmer2	0.4609
ST2M0-Geber1	ST2M0-Nehmer2	0.6302
ST2M0-Nehmer2	ST2M0-Nehmer1	0.6705
ST2M0-Geber2	ST2M0-Geber1	0.6718
ST2M0-Geber2	ST2M0-Nehmer1	0.6811

Table 6 [Table of Milestone [0] distance between projects]

One	The other	Distance
ST2M4-Geber1	ST2M4-Nehmer2	0.6538
ST2M4-Geber1	ST2M4-Nehmer1	0.6603
ST2M4-Nehmer2	ST2M4-Nehmer1	0.6957
ST2M4-NoPlag2	ST2M4-Nehmer1	0.6809
ST2M4-Geber1	ST2M4-NoPlag1	0.8647
ST2M4-NoPlag1	ST2M4-Nehmer1	0.8657
ST2M4-NoPlag2	ST2M4-NoPlag1	0.8772
ST2M4-NoPlag1	ST2M4-Nehmer2	0.8792
ST2M4-NoPlag2	ST2M4-Nehmer2	0.8809
ST2M4-NoPlag2	ST2M4-Geber1	0.8823

Table 7 [Table of Milestone [4] distance between projects]

from other plagiarism detection programs, as they often reflect the similarity in percent to the user.

5 Conclusion

When looking at the four tools two of them could be viable for use in the Divekit workflow. Those being JPlag and AC2. Jplag had overall the clearest results when applied to the known plagiarisms, while its optimized algorithms also promises a quick evaluation of the submissions.

AC2 would be the runner-up with its intriguing visualization of the analyzed programs. However, AC2's results were less clear than those of JPlag and it being an entirely graphical application would require more manual work. However, this extra effort shouldn't pose too much of a problem.

MOSS also had very clear results with the plagiarism attempts being clearly having a higher similarity between each other than the non plagiarized programs, but MOSS is only available as a WWW service which is hosted by the Stanford University in the US. This poses a problem as the Cologne University of Applied Sciences is bases in Germany and has to comply with German and EU privacy laws.

Plaggie would probably not be a viable tool since it hasn't been updated since 2006, and it only supports Java 1.5. Furthermore, our attempts to make it run were unsuccessful. Probably because the used Java version was too new for Plaggie. Otherwise, Plaggie doesn't offer anything special as its major contribution was being the only open source tool, which it isn't anymore at this point in time.

Therefore, the software experiment will focus JPlag and AC2 and evaluate these two tools in order to ascertain which tool is the most suitable for use in ST2.

References

Ahtiainen, A. (2006). *Readme for plaggie*.

Ahtianien, A., Surakka, S., & Rahikainen, M. (2006). *Plaggie: Gnu-licensed source code plagiarism detection engine for java exercises*.

Aiken, A. (2021). *A system for detecting software similarity*. Retrieved November 12, 2022, from https://theory.stanford.edu/~aiken/moss/

Ali, A. M. E. T., Abdulla, H. M. D., & Snášel, V. (2011). Overview and comparison of plagiarism detection tools [Online; accessed 22-November-2022]. https://www.researchgate.net/profile/Nirmala-Svsg/post/Is_there_a_difference_in_results_between_plagiarism_checker_program_Turnitin_and_Plag_Scan/attachment/5c1f2d31cfe4a764550af6cb/AS%3A706889746747392%401545547057426/download/poster22.pdf

Cup. (n.d.). Technische Universität München. Retrieved November 7, 2022, from http://www2.cs.tum.edu/projects/cup/

Freire, M. (2022a). *Ac*. https://github.com/manuel-freire/ac2

Freire, M. (2022b). *Ac*. https://github.com/manuel-freire/ac2/wiki/Quickstart

Jplag - detecting software plagiarism. (2022). JPlag. Retrieved November 12, 2022, from https://github.com/jplag/JPlag

Karp, R. M., & Rabin, M. O. (1987). *Efficient randomized pattern-matching algorithms*.

Mogensen, T. Æ. (2017). *Introduction to compiler design* (2nd ed.). Spring.

Perchelt, L., Malphol, G., & Phlippsen, M. (2000, March 28). *Jplag: Finding plagiarisms among a set of programs*.

Sağlam, T. (2022, September 16). *4. adding new languages*. JPlag. Retrieved October 30, 2022, from https://github.com/jplag/JPlag/wiki/4.-Adding-New-Languages

Schleimer, S., Wilkerson, D. S., & Aiken, A. (2003). *Winnowing: Local algorithms for document fingerprinting*. Retrieved November 12, 2022, from http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf

Wikipedia contributors. (2022). Antlr — Wikipedia, the free encyclopedia [[Online; accessed 13-November-2022]]. https://en.wikipedia.org/w/index.php?title=ANTLR&oldid=1115716413

Wise, M. J. (1993). *String similarity via greedy string tiling and running karp-rabin matching*. University of Sydney, Department of Computer Science.