Abstract—Detecting instances of plagiarism in student homework, with program code in particular, is a subject of active research for over 30 years. One of the early proposed methods was extraction and comparison of source-code metrics. Even though this approach has low algorithmic complexity, it is rarely used in recent papers with some authors claiming that better results are obtained using other methods. In this paper, plagiarism detection is treated as an information retrieval problem, specifically query-by-example (QBE). A feature vector is constructed from source metrics and compared using common similarity measures. Further, evolutionary computation methods are used to optimize the similarity measure. It is shown that, by several metrics used, detection results are on par with state-of-the-art methods with significantly lower execution time.

Keywords - plagiarism detection, real-coded genetic algorithm, feature extraction

I. INTRODUCTION

Detecting instances of academic dishonesty in student papers, and in particular in programming homework, has been a subject of active research for over 30 years. In the early years, a common approach was to extract a certain set of software metrics and compare programs based on this. Programs that have similar values for these metrics were suspected to be plagiarised.

Starting in the late 1980s, advances in algorithms for string comparison made it possible to compare programs directly. First, a preprocessing stage is performed where non-relevant parts of code are removed, such as comments (although some authors argue for preserving the comments [1]). Code is tokenized, and some typical substitutions are performed, based on experience with cheating attempts.

After such preprocessing, all non-overlapping common substrings are found using algorithms such as Rabin-Karp. Ratio of the sum of length of those common substrings to total program length represents a degree of similarity. Various literature reviews agree that methods based on substring matching give better real-life results, although they don’t scale well with increasing program length and number of programs compared.

This paper starts from observation that methods based on code metrics presented in literature use fairly primitive distance metrics, such as simple sum of differences. Authors observe that different features have varying contribution to the probability of plagiarism, even after normalization, so a weighted distance measure is chosen. After evaluating Manhattan, Euclidean and Cosine distance, Euclidean was chosen for the best results.

Various optimization methods were evaluated for calculating weights assigned to each metric. This optimization problem is characterized with high dimensionality (32 weights are optimized) and a relatively slow fitness evaluation (several seconds on average). For these reasons, Real Coded Genetic Algorithm (RCGA) was chosen.

This paper also presents a unique set of features determined from authors’ empirical knowledge from manually reviewing suspected instances of plagiarism. Several of these metrics correspond well to those known from literature, however authors believe that metrics from software engineering are not necessarily applicable and a special set of metrics for plagiarism detection needs to be developed.

Work described in this paper is implemented as a tool named fvpd and is made available on authors’ GitHub page.1 In addition, anonymized dataset is available with homeworks of all students who have signed a consent form agreeing that their homeworks be used for plagiarism detection research.

The remainder of this paper is organized as follows. Previous related work is reviewed in Chapter 2. Chapter 3 gives an overview of features extracted from code. Chapter 4 gives a discussion of a customized version of genetic algorithm used in this paper. Chapter 5 gives experimental results and is divided into subsections on datasets used, evaluation methodology, comparison with state-of-the-art tools and possible weaknesses of presented plagiarism detection method. Finally, Chapter 6 gives a conclusion and proposals for future work.

II. RELATED WORK

Literature reviews on plagiarism detection [2], [3], [1], [4] usually classify detection methods and algorithms into three groups:

- methods based on extracting a set of numerical features (called fingerprints or metrics) from source code and comparing them using a distance function (e.g. [5]) or machine learning (e.g. [6]),
- methods based on finding sets of shared substrings in preprocessed source code (e.g. [7]),
- methods based on extracting a syntax tree from code and finding subtree or substring matches (e.g. [8]).

Of special interest is MOSS tool [9] that combines first and second approach, since fingerprints are extracted using hashed engrams of code.

Most reviewers agree that these groups of tools are sorted in decreasing speed but increasing precision. Earliest works in 1970s and 1980s focused on methods based on software metrics, however in more recent years such methods were abandoned. Tools that are used in production settings and give the best results mostly are based on the substring matching

1https://github.com/EnilPajic/fvpd.
approach, which clearly puts an upper limit on the total number of documents that can be processed in reasonable time.

In recent years authors turn to machine learning techniques to overcome these problems [6] with limited success. A number of recent papers focus on AST parsing and parallel processing as methods for reducing plagiarism detection running time on large datasets.

A related research topic is clone detection. Code clone is defined as a fragment of code that is repeated in the same file or across several files [10], [11]. Detecting such clones is a useful task in the process of code quality improvement. While clone detection uses similar methods and algorithms to plagiarism detection, there are differences that require use of specialized tools. In plagiarism detection more preprocessing is typically done, especially transformations that are specific to plagiarism and arise from teachers’ experience. Plagiarism detection tools compare whole files and don’t look for similarities within the same file. Also plagiarism detection tools output pairs of similar homeworks with a similarity score (usually a percentage) for human review. Clone detector output would have to be preprocessed to give a suitable form.

III. FEATURES

Every instance of source code can be approximately represented as a set of numbers that we call features. Features are grouped in an ordered list called a feature vector, and one feature vector represents object (source code in our case). These features should uniquely represent source code in a way that two similar codes have similar feature vectors (hence the idea for a feature-based detection tool).

The challenging task is to select which features to extract for best performance and unique representation of code. After selecting features, the next step is preprocessing. Each feature is a real number, but features can vary in their magnitude (e.g. number of lines of code vs number of functions), therefore biasing the overall result towards one or few features. To avoid this, normalization is performed, scaling feature values to the range [0, 1]. Min-max normalization was used [12] with the formula

$$X_{\text{norm}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}} (Max - Min) + Min$$

where $Max$ and $Min$ are scaling boundaries (0.9 and 0.1 respectively, in our case).

Feature vectors were compared using a modified version of Weighted Euclidean distance. Every feature $v_i$ was multiplied by weight $w_i$ allowing better control over feature importance (e.g. number of functions is more important for similarity than number of comments). These weights were calculated by Genetic algorithm (GA) with real-number coding, optimized to minimize errors, as explained in the next Chapter.

The modification of Weighted Euclidean distance consists of allowing negative weights. The reasoning behind this is that similar values for some features might actually indicate that code is not plagiarised, since those values would typically be manipulated by novice plagiarists [5].

Selection of static features is inspired by Halstead’s software metrics [13] and other previous works [5], but mostly on authors’ experience with plagiarism detection. The list of all static features used is provided in Table I.

A. Discussion of certain features

Some features appear to be similar, but in order to get precise and accurate results they must be interpreted differently. Examples of such features are $LOC$ – lines of code and $SLOC$ – source lines of code. Optimization results for $LOC$’s weight coefficients can be 10 times smaller than $SLOC$’s because the latter does not include lines consisting of whitespace alone, or multi-line commands.

Other important features for plagiarism detection include $ARRAYS$ – number of defined arrays and $ARR\ ACS$ – number of array accesses.

Features $LOOPS$ and $BRANCHES$ represent Cyclomatic complexity metric (introduced by McCabe in 1976 [14]) which is a commonly used software metric to represent program complexity. Two programs that have similar complexity are likely to be similar in general. Likewise, several features count different types of operators, so their sum can be compared to Halsteads operator count.

The $RARE\ OP$ feature, which represents number of uncommon operators such as access to member via pointer $\rightarrow^*$ (C++ only), binary and operator ($\&$, $\&=$), xor operator ($\oplus$, $\oplus=$) etc., is very important in plagiarism detection. If an expert programmer uses a rare operator (that was not explicitly required by the task), a novice programmer will leave it in their plagiarized code.

IV. GENETIC ALGORITHM

As already stated, not every feature is equally important. Each feature is assigned a weight denoting its contribution to overall distance, indicating that programs are not plagiarized.

Optimization is performed using real-coded genetic algorithm (RCGA) [15]. IEEE754 standard is used to represent floating point numbers. Standard RCGA as described in [16] is implemented except for the mutation operator and the crossover operator. The standard mutation operator [17] suggests to use a Gaussian distribution of possible values centered on current value. This type of mutation exists in authors’ tool, but also other types of mutation are used to allow for escaping local optimums. Sometimes a larger change in weights is needed to see measurable improvement, so multiplication and division by 2 are also possible. Finally, sometimes performance can be improved by setting a single weight to zero.

A random value of $N$ which determines mutation type is made for each gene. Gaussian mutation is 4 times more probable than other mutation types.

Two types of crossover operator were evaluated: a random selection of genes from both parents [18] and using mean value of parent genes for each gene. Surprisingly, the best results were obtained when half of children use the first crossover
Approximately 600 students enrolled into these modules in 2016, while in 2017, this number dropped to about 500. Homework is not mandatory for a passing grade and so many of them choose not to do homework at all, and homework participation gradually drops during the semester. Unlike some other similar modules, homework in this module consists of a large number (15-20) of relatively simple assignments. For some of these assignments, there are not many different ways to solve the problem correctly, and so existing plagiarism detection tools tend to report a very large number of false positive results.

The method to fight plagiarism used to date was to require up to 20% of students to deliver an oral defense of their homework. The choice of students for this oral defense is based on code similarity, but also on other criteria such as suspicious behaviour (copy-pasting homework, suddenly solving homework at the last minute, etc.) and also past success on the module. All students who fail to defend their homework are marked as plagiarized in a “ground truth” file in addition to those who have final code similar to someone else. Also, when two or more homeworks are found to be similar, it is expected that one of those students will succeed in their oral defense (the original author who shared their homework), so this original likewise remains in the “ground truth” file.

Further, instances of very short (unfinished) homeworks that were nonetheless submitted to the grading system were removed entirely from the dataset. Even though proper classification of such homeworks is certainly a useful feature, there is a large difference in how various tools handle such cases: some allow to specify a threshold for homework length, other tools use some undisclosed heuristic, and some tools will simply mark all such homeworks as plagiarized and let a human supervisor unmark them.

This basic ground truth represents a difficult task for plagiarism detection.

The structure of the ground truth file is such that some homeworks are grouped in similar pairs (triplets, quadruplets, etc.), while others are listed alone. Therefore, when evaluating a plagiarism detection tool, we do not verify if such tool correctly identified pairs of similar documents but simply count false positive and false negative results including both sides of their detected pairs.

Anonymized datasets are available from authors per request.

V. EXPERIMENTAL RESULTS

A. Description of Dataset

Dataset construction for plagiarism detection is a challenging problem. Several papers use the output from existing known tools as a “jury” for measuring new proposed algorithms [3], [19]. Another common approach is constructing artificial datasets that reflect various realistic situations [4], [1].

The new dataset presented here is constructed from homeworks submitted by students during two one-semester college modules. The first module, named A, is a module on introductory programming in C. Module named B is delivered after A and consists of intermediate programming in C++.
(in percents). So, to obtain a binary classification one must use a threshold. The issue of what threshold to use is rarely discussed in the literature.

While experimenting with tools based on substring matching found in literature [7], [20], an attempt was made to determine an optimal threshold for each collection with a view that there should be some relationship between similarity threshold and average document length or complexity. Our finding was that optimal threshold shows a very large variance and that no such relationship could be established.

The source of these problems with accuracy and error is that those measures are suited for unsupervised classification where the ultimate goal is to obtain 100% accuracy. While plagiarism detection tools have advanced in recent years, it is still not possible to use them without human supervision [21], [3] (see also MOSS project website).

Consider a collection that consists of 100 documents of which 5 are plagiarized. A null-algorithm that reports 0% similarity for all document pairs would have $FP = 0$ and $FN = 5$, which gives a fairly high accuracy of 95%. Another tool that correctly detects all 5 plagiarized documents but also reports 5 falsely positive results would have the same accuracy, even though such tool is clearly far more useful to a practitioner.

Typically, plagiarism detection tools are used such that a human supervisor scrolls through the sorted list of documents and reviews each result, similar in a way to browsing through search results. During such use, false positive results are easily noticed and discarded, while real cases of plagiarism that are given low similarity (false negatives) are a far greater problem since a human supervisor will eventually give up after several pages of results. This suggests that metrics known from information retrieval would be better suited for comparison, specifically metrics that assign a greater penalty for false negative classification, such as recall.

$Precision$ is defined as the number of correctly retrieved documents (true positives) among the top $n$ results:

$$P_{n} = \frac{TP}{n}$$

while $recall$ is the ratio of correctly retrieved documents (among the top $n$) to the total number of relevant documents in the dataset:

$$R_{n} = \frac{TP}{P}$$

The remaining issue is choice of $n$. Document collections in our dataset vary greatly both in the total number of documents and in the number of known instances of plagiarism. To account for this, we choose $n$ to be the number of documents that are known to be plagiarized ($n = P$), such that a tool that correctly ranks all plagiarized documents before those non-plagiarized would have a recall of 1, while a null-algorithm described earlier would have recall of 0. Thus, our chosen metric represents both precision and recall.

A shortcoming of precision-recall as a comparison metric is that it cannot be used on collections in which there are no known instances of plagiarism.

### C. Comparison with State of the Art

Experiments were performed using homeworks gathered in modules A and B during academic years 2015, 2016 and 2017. Final recall for all modules is given for our tool (fvpd) and contrasted with three tools found in literature that are predominantly used today: JPlag [7], Sim [20] and MOSS [9]. Results can be seen in Table II (our tool is labeled “fvpd”).

To avoid overfitting, the dataset described above was divided into training set and test set. Three combinations were explored: coefficients trained just for C programming language (module A2017 is used for training set), just the C++ programming language (module B2015), and for both programming languages (modules A2016 and B2015 are combined into training set).

Firstly it can be observed that, for our datasets, Sim tool gave far superior recall to JPlag and MOSS, especially with early datasets A2015 and B2015, the result which was not found in literature. JPlag and MOSS in general have similar performance, except with B2016 dataset where MOSS was superior to JPlag and on par with Sim.

Our tool presented in this paper gave very good results that were comparable to JPlag and MOSS on all datasets for A module (C programming language). However, results were mediocre for B2015 dataset and very poor for B2016 dataset, regardless of which training set was used. Using C++ homeworks for training set slightly improved results on the C++ test set but not by much. This point needs to be further researched.

For completeness, another comparison metric is classification error at optimal threshold, given in table III. First, for each homework in dataset, an optimal threshold for binary classification of documents is determined such that error is minimal. Then, $CErr$ is calculated at that threshold. This table also supports the conclusions given above.

### D. Time and Memory Performance

Tools based on substring matching have, in general, an exponential time complexity. A naïve implementation would have an $O(N^2 n^3)$ complexity [3], where $N$ is the total number of documents and $n$ is their length in characters. However, clever programming and various tricks allow state-of-the-art tools to perform very well in most realistic usage cases. The worst-case complexity of $O(N^2 n^{1.12})$ has been claimed [3], [22].

Method proposed in this paper consists of two steps: feature extraction and comparison. Using notation given above, feature extraction has time complexity of $O(N n)$ and a negligible memory usage. Comparison step is a well known “k nearest neighbors” problem that has high performance approximate solutions using methods such as kd-trees [23] and Locality Sensitive Hashing (LSH) [24].

To verify this claim, two folders were created for the purpose of speed test: folder speed1 contains 4000 C programs of 4kB average size, while folder speed2 contains 1350 C++ programs of 14,5kB average size. On an Intel Core i7 CPU, our fvpd extractor took on average 2.25s on speed1 but only...
1.55s on speed2. Meanwhile, Sim tool [20] took on average 1.14s on speed 1 but 4.96s on speed 2. From this we can see an important effect of average document length as well as programming language used on overall extraction speed.

Sim is a highly optimized plagiarism detection program based on substring matching, written using C in 1988, and is by far the fastest plagiarism detection tool available today. Conversely, our extractor was written as a proof of concept and surely can be optimized further. JPlag tool was much slower, while testing the speed of MOSS tool in these test cases was difficult since this tool is only available as a web service, therefore network latency dominates the overall detection time.

E. Plagiarism Detection Weaknesses

There are several possible avenues of attack on the method presented in this paper. Plagiarists can insert large quantities of code which is never executed or doesn’t affect the correctness of output, skewing the metrics presented. To avoid this, a preprocessing stage could be added which detects and eliminates such superfluous code. Further, C++11 and later standards introduced a number of new programming constructs that should be separately counted.

VI. Conclusion

A method for plagiarism detection based on extracting features, or software metrics, from source code is presented in this paper. It is shown that, using an appropriate distance function, results that are broadly on par with state-of-the-art are possible. This is important since feature extraction and retrieval can be delivered much faster on a very large dataset compared to traditional methods based on substring matching and hashing.

These results can also have impact on code clone detection techniques, which is a venue for future research.

Much further work is needed, especially in the area of selecting features. A thorough statistical analysis needs to be performed to find if some of the features are redundant, and new features should be introduced, especially those that are targeted at object-oriented code.

Different distance metrics can be explored such as generalized Mahalanobis distance.

The method described in this paper has produced a set of weights that shows a surprising level of programming language specificity, even with languages that are otherwise very similar, such as C and C++. One possible explanation is that plagiarists use different cheating strategies in modules A and B. In production usage it may or may not be practical to use different pretrained weights for each language used. Further study in this regard should be performed. Authors also plan to address the weaknesses described in previous chapter.

REFERENCES


