© 2006 *Institute of Mathematics and Informatics, Vilnius*

Desktop Tools for Offline Plagiarism Detection in Computer Programs

Maxim MOZGOVOY
Department of Computer Science, University of Joensuu
Länsikatu 15, 80100 Joensuu, Finland
e-mail: maxim.mozgovoy@cs.joensuu.fi

Received: December 2005

Abstract. Plagiarism in universities has always been a difficult problem to overcome. Various tools have been developed over the past few years to help teachers detect plagiarism in students' work. By being able to categorize the multitude of plagiarism detection tools, it is possible to estimate their capabilities, advantages and disadvantages. In this article I consider modern plagiarism software solutions, paying attention mostly to desktop systems intended for plagiarism detection in program code. I also estimate the speed and reliability of different plagiarism detection systems that are currently available.

Key words: plagiarism detection, similarities detection, file comparison.

1. Introduction

According to investigations conducted at Duke University (Bliwise, 2001), the prevalance of plagiarism is widespread. For example, about 40% of all students confessed to copying sentences without citing the original source, 11% reported almost verbatim copying of material, 9% "utilized" another student's computer program. Plagiarism is not only found in student work; there are several reports (Brumfiel, 2002; Collberg and Kobourov, 2003) about instances of plagiarism and so-called *self-plagiarism* (see below) in articles submitted by researchers for scientific conferences and journals.

The rapid development of computing and Internet technologies has made plagiarism much easier to carry out. In the past, people had to spend considerable time to find a relevant document, then copy its fragments by hand. With advances in computing technology it takes considerably less time to search for relevant documents and literally seconds to cut and paste sections of an original document into one's own. Moreover, there are numerous "paper-mills" (e.g., `http://www.exampleessays.com`, `http://www.directessays.com`, `www.dissertationsandassignments.com`), where students can buy recycled or custom made papers.

Much can be done to prevent plagiarism before it occurs. Plagiarism prevention techniques include smart design of assignments, supervised tests, work process tracing and so on (Wiedemeier, 2002; Zobel and Hamilton, 2002). Some teachers even use special plagiarism-preventing software tools, such as Anti-Plagiarism Editor (APE), which tracks all potential cheats, such as the cutting and pasting of large text-blocks (Vamplew and Dermoudy, 2005). The teacher can decrease the level of cheating just by notifying students that their work will be checked for plagiarism with a software tool (Braumoeller and Gaines, 2001). Sometimes the plagiarism can be of an unintentional nature. The students are just not sufficiently educated to use the sources properly. Fortunately, there are good style guidelines (see, e.g., Trivedi and Williams, 2002), which can be utilized by the teachers.

Although priority should be given to plagiarism prevention, detecting plagiarism after it occurs is still a very important task. The last few years have brought about many solutions for automatic plagiarism detection in essays and in program code. Some of them are currently being widely used.

The world of plagiarism detection software is not uniform: different approaches exist, each of which aims at detecting different kinds of plagiarism. In this review we will focus on “offline” (or “hermetic”) systems designed for program code similarity analysis. Instead of making an ordinary survey of software tools, we provide a taxonomy for the most popular approaches and examine their strengths and weaknesses. This work is intended to provide a short overview of the detection tools currently available and to point out several important topics for future discussion.

2. Definitions of Plagiarism

One issue that is important for every plagiarism detection tool is how plagiarism is defined. Unfortunately, there are no formal, operational definitions of plagiarism. Usually people cite the definition given by Webster’s Encyclopedic Dictionary: “the unauthorized use of the language and thoughts of another author and the representation of them as one’s own” (Verco and Wise, 1997). Some definitions are more precise, but still informal. An example is Manber’s definition of plagiarism, which is implied in the following quote: “Our goal is to identify files that came from the same source or contain parts that came from the same source” (Manber, 1994).

It seems quite clear that the concept of plagiarism itself is so blurred that one cannot expect someone to invent any fully adequate definition that is suitable for direct implementation as a computer program. Much of the advice given to people who need to detect plagiarism is based on common sense and/or deal with fuzzy, informal concepts (Clough, 2000). Concerning cheating techniques in general, it is necessary to take into account issues like unauthorized collaboration between students, asking (or even paying) for help from a skillful outsider, or self-plagiarism (i.e., recycling one’s own published text without quoting). Furthermore, it is sensible to consider results obtained due to influence or “creative understanding” of other people’s works. Software tools cannot handle all such situations because of the limits of current technology. Because of the difficulties of creating an all-encompassing definition of plagiarism, when we refer to *plagiarism* in this rest of this article we generally refer to *plagiarism that is detectable via software solutions*, unless otherwise stated.

Basically, every tool implements its own definition of plagiarism, which usually becomes clear from the progam's documentation. The reliability of the software detector greatly depends on the relationship of its definition of plagiarism to the amorphous definition of *plagiarism* that is used by human detectors. For example, the study by the authors of the software package JPlag shows that string matching-based file-file comparison routines can reveal similarity, which is considered to be a good indication of plagiarism by human detectors (Prechelt *et al.*, 2002). Manber (1994) proposes an explicit criterion for file similarity: "we say that two files are similar if they contain a significant number of common substrings that are not too short".

3. "Online" Detection Systems

Online detection systems can check an article for fragments of text that can also be found somewhere on the Net. No doubt, the Internet is the number one source for possible borrowings in the case of ordinary students' essays[1]; therefore, the importance of online detection systems should not be underestimated. Although a thorough analysis of online detection systems is beyond the scope of this article, there are several issues related to online detection that are worth mentioning.

Online detection systems are close relatives of ordinary Internet search engines. Online detection systems concentrate on the speed and width of detection, at the cost of the quality of detection. For example, the developers of the *Turnitin* system (`www.turnitin.com`) claim that their database consists of over 4.5 billion pages, which is updated daily with 40 million pages. They also claim they maintain "a huge database of books and journals, and a database of the millions of papers already submitted".

It is not surprising that only a few different "online" plagiarism detection services exist because many of such systems require enormous computational resources. Some of them, like Turnitin, maintain their own databases, while others (e.g., EVE2 available at `www.canexus.com`) utilize the power of existing search engines.

Although they are very important tools, online plagiarism detection services currently cannot use the advanced, but time-consuming, document comparison routines used in some offline systems. Also, their authors have to deal mostly with technical issues that are not related to plagiarism detection directly (like organizing large-scale document banks). Furthermore, "online" systems are usually commercial, so their documentation primarily contains advertisements and independent reviews are shallow in algorithmic details.

4. "Offline" Detection Systems

Offline detection systems adopt a "hermetic" model of the textual world. All "borrowings" are assumed to be made from the documents inside a given collection. For example, in the case of offline detection systems, the entire textual area in which the offline

[1] It is not so, e.g., in case of plagiarized program code.

detection system searches for sources of plagiarism might be a collection of documents provided by a teacher. (In contrast, the textual area of online systems is the entirety of text on the Internet.)

In some cases, the “hermetic” model turns out to be inappropriate. For example, if students are asked to write an essay about the economic situation of a certain country in a certain historical period, it is unlikely to find cases of “hermetic” plagiarism since everybody has their own topic; there is, basically, nothing to copy and paste. On the other hand, laboratory work assignments in computer science tend to provoke “knowledge-sharing” between students because each of the students’ assignments probably share a high degree of similarity[2]. Also it is hard to find a piece of code on the Net that will do exactly what is called for. Knowledge-sharing is very common: it is the most frequent plagiarism technique according to (Sheard *et al.*, 2002). Several plagiarism detection systems are specially designed to analyze computer programs rather than natural language.

It is worthwhile to re-emphasize that the term “online” in this article refers to systems that search the Net to reveal plagiarism; “offline” systems search collections of documents only. The form of user interaction with the system is not important for determining if a system is an offline system or an online sytem since an “offline” system can be implemented as a Web service (JPlag) or an “online” system can be an installable desktop application (EVE2).

We argue that the majority of offline plagiarism detection tools fall into one of three categories: fingerprint-based, string matching-based, and tree-matching based systems. In the rest of this article we give an overview of those categories. We end with a discussion of issues related to speed and visualization, authorship identification, and the principles of evaluation.

4.1. *The Fingerprint-Based Approach and LSA*

The fingerprint-based approach was first used in attribute counting systems, which were largely used for plagiarism detection in the past (Grier, 1981; Faidhi and Robinson, 1987).

The basic idea in the fingerprint-based approach is to create a kind of *fingerprint* for every document in the collection. Each fingerprint may contain several numerical *attributes* that somehow reflect the structure of the document. For example, the system can store the average number of words per line, the number of lines, the number of passages, the number of unique words, and so on. If two fingerprints are close to each other (according to a *distance function*), the documents themselves can also be considered as being similar.

Over the last several years, a couple of different metrics have been tested. It is generally believed nowadays that fingerprint-based approaches are quite weak since even slight textual modifications can considerably affect the fingerprint of a document; newer systems, based on content comparison, almost forced out attribute counting systems (Verco and Wise, 1997).

[2] Usually teachers just slightly modify one “skeleton” assignment to obtain different task variants.

A good example of a fingerprint-based system is *Accuse* (Grier, 1981). The fingerprints in *Accuse* include seven parameters: *the number of unique operators, the number of unique operands, the total number of operators, the total number of operands, the number of code lines, the amount of variables declared* (*and used*), and *the total number of control statements*. The correlation scheme computes an "increment" for each attribute pair:

```
$A_1$ = attribute$_i$ count in the first file
$A_2$ = attribute$_i$ count in the second file
increment$_i$ = importance of attribute$_i$ - ($A_1$ - $A_2$)
```

Then these increments, summed up, yield the final similarity ratio.

Several more-advanced approaches for creating fingerprints, showing reasonable levels of reliability, are currently in use. The fingerprints in modern systems are usually made up of the values obtained by applying a mathematical function (a sort of hash function) to specially selected substrings in the collection of files (Hoad and Zobel, 2003). Fingerprints based on file content are used, for example, in the well-known MOSS system[3] (Schleimer *et al.*, 2003).

Since file comparison in fingerprint-based systems is performed by means of comparison of small fingerprints, the speed of detection is usually high. If fingerprint size is constant, the complexity of the work[4] is $\mathrm{O}(N)$, where N is the number of documents in the collection (Hoad and Zobel, 2003). (Additional $\mathrm{O}(nN)$ – where n is the average file length – time is required to create fingerprints). To obtain similarity ratios for all possible file pairs, the system should make $\mathrm{O}(N^2)$ comparisons in total.

It is possible for fingerprint-based systems to utilize a universal text retrieval method, such as latent semantic analysis (LSA). LSA allows for the creation of a special fingerprint in the form of a numeric vector for every document, based on frequencies of words found in the text. Unfortunately, LSA destroys the structure of the documents since it treats every document as a bag of non-connected words. For specialized tasks, like plagiarism detection in program code, LSA has been reported to have a 50% or even higher similarity between independent programs since the programs usually share the same vocabulary (Nakov, 2000). However, comprehensive evaluation of LSA methods applied to the plagiarism detection problem is still missing.

4.2. *The Content Comparison Techniques*

If fingerprint comparison is not enough, the detector can compare the contents of documents. While the core idea is simple, none of the techniques of content comparison is completely reliable. Since different systems use different algorithms; currently it is not possible to determine which technique is the best – each has its own advantages and disadvantages. Furthermore, the task of plagiarism detection in program code turns

[3] I consider MOSS to be primarily fingerprint-based system, though it also utilizes string matching techniques.

[4] I.e., the complexity of the comparison of the query document against the collection.

out to be a problem that requires specialized solutions. There are several tools specially designed for software plagiarism detection.

Programming languages are formal and much simpler than any natural language. This makes the problem of detection easier since the techniques of possible plagiarism are limited in programming languages. It is possible to classify those techniques and explicitly program procedures that are insensitive to all of them. For natural languages this approach is obviously much harder to perform.

There have been attempts to list the possible techniques that a plagiarizer can do to hide plagiarism (Joy and Luck, 1999; Jones, 2001). Some of these techniques are:

1. Changing comments (rewording, adding, changing comment syntax and omitting).
2. Changing white space and layout.
3. Renaming identifiers.
4. Reordering code blocks.
5. Reordering statements within code blocks.
6. Changing the order of operands/operators in expressions.
7. Changing data types.
8. Adding redundant statements or variables.
9. Replacing control structures with equivalent structures (while-loop by do-while loop; nested if statements by a switch-case block and so on).
10. Replacing the functional call by the body of the function.

It is presumed that a plagiarizer does not have to understand the algorithmic meaning of the program. In fact, any of the techniques listed above can be performed automatically by a compiler-level tool that can recognize the semantic meaning of individual statements without recognizing the functionality of the whole program.

However, not all such changes are equally advanced (Joy and Luck, 1999). Some of them, like changing formatting or identifier names (*lexical* changes), do not even require the knowledge of the programming language used. Other ones (e.g., *structural* changes) can be done only if the plagiarizer is familiar with language semantics.

We can use the list of plagiarizer's techniques to also define *plagiarism*: *a plagiarized program is a program that can be obtained from the original one by means of one or more of the actions listed above*.

Next we discuss the algorithmic solutions that have been implemented in various systems.

4.3. *Tokenization*

Tokenization (Joy and Luck, 1999; Prechelt *et al.*, 2002; Mozgovoy *et al.*, 2005) is a very popular technique used by most source code plagiarism detection systems. Its main purpose is to render useless all kinds of renaming tricks. Tokenization algorithms basically substitute various elements of program code with single tokens. For example, any identifier can be replaced by the token <IDT>, and every numerical value by the token <VALUE>. Now, if a program contains a line

```
a = b + 45;
```

it will be replaced by the string

<IDT> = <IDT> + <VALUE>;

So trying to rename the variables will not help since every line of the form "identifier $=$ identifier $+$ value;" is translated into the same tokenized sequence.

There are methods of tokenization (or similar procedures) that perform more advanced substitutions. For example, Baker's parameterized match algorithm[5] (Baker, 1995) will treat two given code fragments as identical if one of them is obtained from the other one by a series of regular substitutions of identifiers.

Tokenization techniques can also utilize semantic information about the control structures of the programming language being used. For instance, any loops can be substituted by the <BEGIN_LOOP>...<END_LOOP> structure.

There are two main drawbacks related to tokenization:

1. Any tokenizer is language-dependent. So a separate routine is needed for every programming language[6]. Fortunately, there are free parsers available for all popular languages nowadays.
2. The detector, dealing with tokenized files, becomes more "paranoiac". Tokenization increases the degree of similarity between any two given programs.

Tokenization is usually the first action performed by detectors. After the tokenization of input files, a system-specific comparison routine is invoked.

4.4. *String Matching-Based Algorithms*

The scheme of the usual file content comparison system is shown in the following pseudocode:

```
FOR EACH collection file F
  FOR EACH collection file G, F $\neq$ G
    Calculate similarity between F and G
```

The core function, which calculates similarity, may vary highly from one system to another. The most widespread technique is to utilize a string matching procedure, treating input files (tokenized files in case of software plagiarism) as strings. This gives a true content-comparison system; though the semantic issues of a file (such as the meaning of loops and functional calls) still remain unanalyzed.

Early systems like YAP (Wise, 1992) used simple mechanisms, like the UNIX sdiff tool, that perform line-by-line comparison of two files under Levenshtein distance. During the the last few years, significantly more advanced string matching methods have been implemented in plagiarism detectors.

4.4.1. *The Running-Karp-Rabin Greedy-String-Tiling (RKS-GST) Algorithm*

The RKS-GST algorithm was used, e.g., in Michael Wise's YAP3 tool (Wise, 1996). The basic aim of the RKS-GST algorithm is to find a "best tiling" for two given input files, i.e.,

[5] Which is not a tokenizer, but its purpose is similar in our context.

[6] JPlag, for example, explicitly allows to select the correct parser to apply to files of the given collection.

the joint coverage of non-overlapping strings that includes as many tokens from both files as possible. The existence of a polynomial algorithm that provides an exact solution is still an open problem (Wise, 1996) so it is necessary to make several heuristic assumptions to develop a practically applicable procedure. The fact that longer tiles are more valuable than shorter ones leads to greedy heuristics, which are actually implemented in the RKS-GST algorithm.

The RKS-GST algorithm can be described (very superficially) as follows. The routine begins by analyzing the matches of length initial-search-length and greater. These matches are obtained by calling the Karp-Rabin procedure (Karp and Rabin, 1987). Then the matches are analyzed (beginning from the longest one). If the current match does not overlap with the existing tiling, it is added to the coverage as a new tile. After all matches are processed, a new search occurs with the smaller match length. When the match length reaches the minimum-match-length threshold value, the algorithm finishes its work.

The RKS-GST algorithm was shown to have $\mathrm{O}(n^3)$ complexity in the worst case (where n is the sum of the lengths of the input strings), while the expected running time (obtained empirically) is almost linear – just $\mathrm{O}(n^{1.12})$ (Wise, 1994). So the overall complexity of the RKS-GST-based system, which produces similarity ratios for all file pairs, should be $\mathrm{O}(N^2 n^{1.12})$.

This method was used later in a well-known system – JPlag (Prechelt *et al.*, 2002). The tiling approach is now considered to be quite advanced and reliable; most widely-used systems implement algorithms that can be treated as tiling variations. However, the actual performance (in terms of speed and reliability) highly depends on the assumptions that are used.

A version of the greedy string tiling has also been implemented in Sherlock project (Joy and Luck, 1999).

4.4.2. *Parameterized Matching Algorithms*

The parameterized matching algorithms approach is mainly associated with Brenda Baker's DUP tool (Baker, 1995). As was already mentioned, parameterized matching allows the system to find identical sections of code as well as sections with systematic substitutions of identifiers. In practice, the matching is done by replacing identifiers with their offsets: the first occurrence is substituted by zero, while the next occurrences are replaced by the number of tokens since their last use (Clough, 2000). This technique can be considered as an ordinary text matching routine combined with an advanced (less "paranoiac") tokenizer.

DUP's running time is estimated to be linear in input length[7] (Baker, 1995), but quadratic in the worst case.

4.5. *Parse Trees Comparison Routines*

Going one step further, it is reasonable to analyze parse trees[8] of the programs instead of their listings. Probably, this idea first was utilized in Sim utility (Gitchell and Tran, 1999).

[7] For all-against-all detection it is $\mathrm{O}(N^2 + Nn)$ in our terms.

[8] Parse tree (built by the parser) represents the syntactic structure of the program.

Sim still uses an ordinary string matching routine to compare programs, but instead of analyzing code blocks, it compares corresponding parse trees, which are converted to strings. So Sim is a hybrid approach that lies somewhere between ordinary string matching and tree comparison. The complexity of the algorithm used is O(s^2), where s is the maximum size of the parse trees (Gitchell and Tran, 1999). Since the size of the parse tree of the file is proportional to the file length, the overall complexity of process for obtaining all necessary similarities is estimated as $\mathrm{O}(N^2n^2)$.

The pure tree comparison procedure was implemented in the Brass project (Belkhouche *et al.*, 2004). Since tree comparison is more complex and therefore slower than string matching, Brass uses a kind of string comparison routine to filter only "suspicious" documents. Then a special "micro comparison" algorithm is applied to provide more reliable results.

Though this approach seems to be the most advanced, little research in this area has been made so far. For example, it is still unknown if it is worthwhile to perform such a complex analysis of input files – i.e., it is unknown whether it is necessary to compare parse trees to reveal instances of plagiarism or if usual string matching algorithms are reliable enough. Furthermore, any fast enough tree comparison routine requires some optimizations like greedy heuristics. For now, it is not clear how these techniques affect reliability.

The complexity of the Brass algorithm has not yet been analyzed, but it is reasonable to suppose it is not faster than YAP3/JPlag systems.

4.6. *Speed and Visualization Issues*

Developing a user-friendly interface for plagiarism detection is a separate issue. Currently several systems, such as MOSS, JPlag and Sherlock, provide impressive interface solutions, which can serve as decent templates for other projects.

There are no universal recommendations, but it is clear that every system should be able to:

1) show a list of all similar file pairs with the corresponding degrees of similarity; and
2) give a detailed report about any selected pair: plagiarized blocks should be highlighted, and it also should be clear which blocks were considered as similar.

PRAISE (Lancaster and Culwin, 2004) and Sherlock detectors provide quick visualization of results in the form of a graph where each vertex represents a single document, and each edge shows the degree of similarity between two documents. (If the value of similarity is lower than a certain threshold, no edge is created). This mechanism is very useful, especially for small collections; therefore I can advise authors to implement something similar in their detectors. Note that a good visualization module will not only help to find a plagiarism case, but also to prove it quickly in a conflict situation (i.e., to show the evidence of plagiarism in any particular case).

Speed issues have also been in the scope of interest during recent years. By following the general scheme of pairwise file comparison, it is necessary to perform $\mathrm{O}(N^2)$ file-file comparisons for a collection, consisting of N files (which is usually considered to be a considerable amount of work). That is why the problem of inventing fast comparison routines always has been crucial[9]. The usual approach is to develop a fast comparison procedure, which can be used as a filter for "interesting" pairs. Such filter procedures can be applied to the files themselves (Belkhouche *et al.*, 2004) or to their fingerprints (which are much faster, but generally less reliable) (Manber, 1994).

Our recent project (Mozgovoy *et al.*, 2005) tries to bring about a significant increase in detection speed by means of algorithmic solutions. We combine all collection files into a single structure (which is a small modification of the well-known suffix array (Manber and Myers, 1990)), and then compare separate documents against this collection at once.

Finally we obtain a total complexity of $\mathrm{O}(nN\gamma + N^2)$, including the time to build the suffix array index structure, where N is the total number of files, n is the average file length and γ is a finely-tunable constant, $\gamma = \Omega(\log nN)$. Actually, γ represents the minimal length of matches, which our algorithm tries to find. A too small γ value will give many false matches, while a too large value can lead to skipping important substrings. Normally we select some "typical" value for the length of the string that a plagiarizer can copy & paste (e.g., 10–20 tokens).

Any plagiarism detection routine based on pairwise file comparisons will have a complexity of $\mathrm{O}(f(n)N^2)$ at least, where $f(n)$ is the complexity of comparing two files of length n.

5. Authorship Identification

The problem of authorship identification and stylometry is widely-known. Authorship identification methods include many different techniques: Shallow parsing, Markov models, Qsum algorithm, entropy and content analysis, etc. (Cook, 2003).

It seems clear that these methods can be used to reveal possible instances of plagiarism indirectly. It can be an indication of plagiarism if no essay chunks can be found either on the Net or inside other students' submissions, but authorship analysis shows that two different parts of this essay belong to two different authors. Although the authorship identification problem has already been researched for years, only a few studies have been made in plagiarism detection. Recent investigations report only limited success with this approach (Hersee, 2000; Bonsall, 2004). It turns out that authorship analysis methods can produce reliable results for large text blocks only; possible deviations between different paragraphs of the same text are too high in many cases. This makes known authorship identification methods unreliable for plagiarism detection, but this direction needs more research.

[9] This may be not the case for "offline" plagiarism detection on small data sets, but important for finding similarities in the source code of a large software project.

6. Principles of Evaluation

It seems obvious that any plagiarism detection system should first of all be able to detect plagiarism; all other issues can raise interest only if the detector itself is reliable. Unfortunately, most projects still lack proper evaluation. Such testing can be difficult for "online" systems that have to maintain huge data collections; also, as we said before, the issues of speed and coverage can have greater importance than quality. But even papers on "offline" detection systems usually have very simple evaluation procedures, which show the positive sides of new algorithms (Belkhouche *et al.*, 2004; Gitchell and Tran, 1999; Joy and Luck, 1999).

Geoffrey Whale (Whale, 1990) tried to adapt well-known metrics – *recall* and *precision*, which are used in information retrieval -, for the evaluation of plagiarism detection systems. Whale's approach was used to evaluate several known systems (Verco and Wise, 1997). Though these metrics are valuable for describing the reliability of the system, they are difficult to measure. Basically, it should be known beforehand which files from the collection contain instances of plagiarism; however, only human experts can provide reliable sample results. Nonetheless, manual evaluation suffers from other problems:

1. Even human markers in many cases have different opinions about particular submissions;
2. it is not feasible to manually check real-world collections that contain hundreds of submissions.

Recent work by Hoad and Zobel (Hoad and Zobel, 2003) shows that the highest false match (HFM, the highest percentage given to an incorrect result) and separation (the difference between the lowest correct result and the HFM) can serve as better metrics for plagiarism detection systems than recall and precision.

The authors mention the difference between text retrieval and plagiarism detection. In text retrieval the measure of similarity between a user query and any document in the collection is a "score" without any upper bounds. Theoretically, there should be no "ideal" queries, which give a maximal possible score for an arbitrary document. In plagiarism detection the situation is different: The exact copy of the original document is the ideal match, so the upper limit of the similarity function for every given file is known beforehand.

A good system should try to minimize the HFM and maximize separation, though these values are not independent, so only the ratio HFM/separation is really important. A high separation value can compensate for high HFM and vice versa – low separation is satisfactory if the HFM is not high.

In the recent paper (Mozgovoy *et al.*, 2005) my co-authors and I tried to compare different plagiarism detection systems using the "conformism test". The conformism test determines how many submissions, considered as plagiarized by some certain system, are found in a common "plagiarized files subset", defined by several other systems ("the jury") by processing the same test collection.

Though different systems often differ in which file pairs originated from the same source, they usually agree about the presence or absence of plagiarism in a certain file.

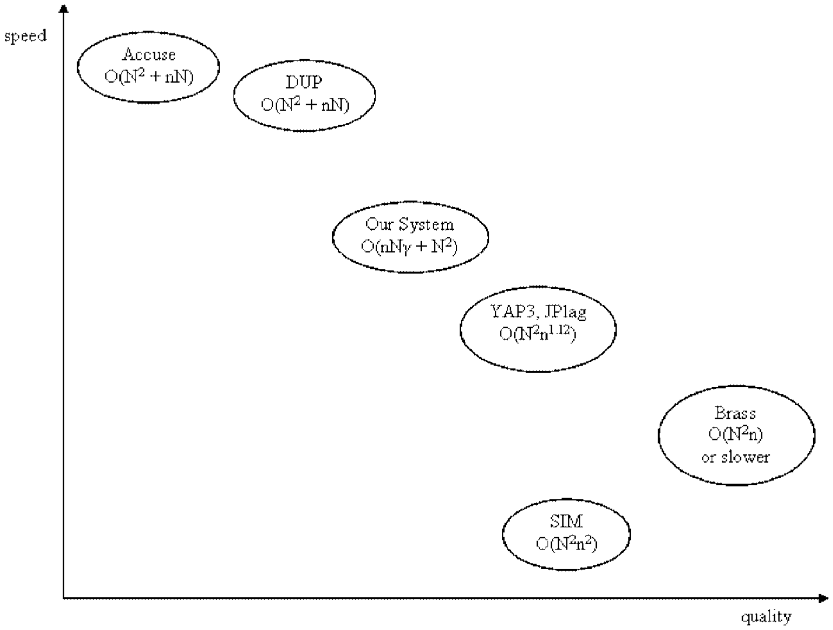

Fig. 1. Comparison of different plagiarism detection systems.

Our views on the current plagiarism detection systems are summarized in Fig. 1.

The asymptotic complexities of different systems are mostly taken from the corresponding articles[10]. Quality measures are based primarily on our own thoughts about underlying approaches; we have neither strict proofs, nor reliable sources to substantiate them.

7. Conclusions

Since the problem of plagiarism is always relevant, the software solutions that help teachers to detect plagiarism cases are being continually developed. Simple attribute counting tools evolved into complex systems that use advanced string- and tree-matching mechanisms in combination with impressive visualization modules. The structured, restricted nature of programming languages makes plagiarism detection in software projects harder for the people, but simpler for the computers. Therefore, a serious progress was achieved in this direction. Probably, some systems already reached the limitations of the corresponding category. For example, it is hard to believe that someone can develop a string matching-based file-file comparing system that would be significantly better than JPlag. On the other hand, the progress can be made in a new (like tree matching) or little-studied (authorship attribution) direction.

[10] Brass complexity is not given in (Belkhouche *et al.*, 2004), so we can only estimate a reasonable order.

It is also clear that plagiarism detection software cannot substitute a well-planned teaching process, aimed (in particular) at plagiarism prevention. The appropriate design of the assignments and the proper organization of the educational process can significantly reduce cheating.

8. Still Open Questions

Despite the existence of numerous systems intended for plagiarism detection, there are still many open questions and topics for future research:

1. How advanced should the technology used for plagiarism detection be? It is clear now, that the metrics-based approach is insufficient, but the best alternative between string matching and tree matching is still unknown.
2. How should the proper user interface be designed? Which features should be implemented in every plagiarism detector?
3. How can enormous complexity growth be avoided? Is it possible to create a fast *and* reliable system? At least, is it possible to use a hybrid approach: a fast filter plus a reliable file-file comparator?
4. Can authorship identification techniques be adapted for the task of plagiarism detection?
5. Is it possible to invent a simple and reliable procedure for the evaluation of new systems? How can we measure the degree of reliability? Which deviations from human experts' opinions are crucial and which are not?

Acknowledgements

I am grateful to Kimmo Fredriksson for guiding me in this research and to Justus Randolph for reviewing the paper.

References

Baker, B.S. (1995). On finding duplication and near-duplication in large software systems. In *Proc. of Second IEEE Working Conf. on Reverse Eng.*, pp. 86–95.

Braumoeller, B., and B. Gaines (2001). Actions do speak louder than words: deterring plagiarism with the use of plagiarism-detection software. *PS*: *Political Science and Politics*, **34**(4), 835–839.

Bliwise, R. (2001). A matter of honor. *Duke Magazine*, May-June, 2–7.

Belkhouche, B., A. Nix and J. Hassell (2004). Plagiarism detection in software designs. In *Proc. of the 42nd Annual Southeast Regional Conference*, pp. 207–211.

Bonsall, B. (2004). *The Automatic Detection of Plagiarism*. University of Sheffield. http://www.dcs.shef.ac.uk/intranet/teaching/projects/archive/ug2004/pdf/u7bb.pdf

Brumfiel, G. (2002). Physicist found guilty of misconduct. *Nature*, Sept., 419–421.

Collberg, C., and S. Kobourov (2003). *Self-Plagiarism in Computer Science*. Technical Report TR03-03, University of Arizona.

Clough, P. (2000). *Plagiarism in Natural and Programming Languages*: *an Overview of Current Tools and Technologies*. Internal Report CS-00-05, University of Sheffield.

Faidhi, J.A.W., and S.K. Robinson (1987). An empirical approach for detecting program similarity within a university programming environment. *Computers & Education*, **11**(1), 11–19.

Grier, S. (1981). A tool that detects plagiarism in pascal programs. *ACM SIGCSE Bulletin*, **13**(1), 15–20.

Gitchell, D., and N. Tran (1999). Sim: a utility for detecting similarity in computer programs. In *Proc. of the 30th SIGCSE Technical Symposium on Computer Science Education*, New Orleans, Louisiana, pp. 266–270.

Hersee, M. (2000). *Automatic Detection of Plagiarism*: *An Approach Using the Qsum Method*. University of Sheffield.
`http://www.dcs.shef.ac.uk/intranet/teaching/projects/archive/ug2001/pdf/u8msh.pdf`

Hoad, T.C., and J. Zobel (2003). Methods for identifying versioned and plagiarised documents. *Journal of the American Society for Information Science and Technology*, **54**(3), 203–215.

Joy, M., and M. Luck (1999). Plagiarism in programming assignments. *IEEE Transactions on Education*, **42**(2), 129–133.

Jones, E.L. (2001). Metrics based plagiarism monitoring. *The Journal of Computing in Small Colleges*, **16**(4), 253–261.

Karp, R.M., and R.M. Rabin (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, **31**(2), 249–260.

Lancaster, T., and F. Culwin (2004). Using freely available tools to produce a partially automated plagiarism detection process. In *Proc. of the 21st ASCILITE Conference*, Perth, Australia, pp. 520–529.

Manber, U. (1994). Finding similar files in a large file system. In *Proc. of USENIX*, San Francisco, California, pp. 1–10.

Cook, M. (2003). *Experimenting to Produce a Software Tool for Authorship Attribution*. University of Sheffield.
`http://www.dcs.shef.ac.uk/intranet/teaching/projects/archive/ug2003/pdf/u0mc2.pdf`

Mozgovoy, M., K. Fredriksson, D. White, M. Joy and E. Sutinen (2005). Fast plagiarism detection system. In *SPIRE'05*, November 2–4, Buenos Aires, Argentina, pp. 267–270.

Manber, U., and G. Myers (1990). Suffix arrays: a new method for on-line string searches. In *SODA '90*: *Proc. of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 319–327.

Nakov, P. (2000). Latent semantic analysis of textual data. In *Proc. of the Conference on Computer Systems and Technologies*, Sofia, Bulgaria, pp. 5031–5035.

Prechelt, L., G. Malpohl and M. Philippsen (2002). Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, **8**(11), 1016–1038.

Sheard, J., M. Dick, S. Markham, I. Macdonald and M. Walsh (2002). Cheating and plagiarism: perceptions and practices of first year IT students. In *Proc. of ITiCSE'02*, June 24–26, Aarhus, Denmark, pp. 183–187.

Schleimer, S., D.S. Wilkerson and A. Aiken (2003). Winnowing: local algorithms for document fingerprinting. In *SIGMOD*, San Diego, pp. 76–85.

Trivedi, L., and S. Williams (2002). *Using Sources*. Hamilton College.
`http://www.hamilton.edu/academics/resource/wc/usingsources.html`

Verco, K.L., and M.J. Wise (1997). Plagiarism à la mode: a comparison of automated systems for detecting suspected plagiarism. *The Computer Journal*, **39**(9), 741–750.

The New Webster's Encyclopedic Dictionary of the English Language. Random House Value Publishing, Inc.

Whale, G. (1990). Identification of program similarity in large populations. *The Computer Journal*, **33**(2), 140–146.

Wiedemeier, P.D. (2002). Preventing plagiarism in computer literacy courses. *The Journal of Computing in Small Colleges*, **17**(4), 154–163.

Wise, M.J. (1992). Detection of similarities in student programs: YAP'ing may be preferable to plague'ing. *ACM SIGSCE Bulletin*, **24**(1), 268–271.

Wise, M.J. (1994). *Running Rabin-Karp Matching and Greedy String Tiling*. Basser Department of Computer Science Technical Report, Sydney University.

Wise, M.J. (1996). YAP3: improved detection of similarities in computer program and other texts. In *Proc. of SIGCSE '96 Technical Symposium*, Philadelphia, USA, pp. 130–134.

Zobel, J., and M. Hamilton (2002). Managing student plagiarism in large academic departments. *Australian Universities Review*, **45**(2), 23–30.

M. Mozgovoy is a PhD student at the University of Joensuu, Finland. Formerly working as a researcher and a teacher (theory of computing, C++ language, data mining, and basics of plagiarism detection), he is now concentrated on postgraduate studies. His scientific interests include also natural language processing and various aspects of artificial intelligence.

Plagiatui kompiuterinėse programose aptikti skirtos darbalaukio priemonės

Maxim MOZGOVOY

Universitetai su plagijavimo problema susiduria visais laikais, šią problemą spręsti visuomet būna sunku. Per pastaruosius kelerius metus buvo sukurta įvairių priemonių, padedančių dėstytojams aptikti plagiatą tarp studentų darbų. Derama daugybės plagiatui aptikti skirtų priemonių klasifikacija gali tinkamai pasitarnauti nustatant atskirų priemonių galimybes, privalumus bei trūkumus. Straipsnyje aptariamas modernios programinės įrangos, skirtos kovoti su plagijavimu, spektras, daugiausia telkiant dėmesį į darbalaukio sistemas, sukurtas kompiuterių programos plagiatui aptikti. Taip pat aptariamas šiuo metu prieinamų plagiatui aptikti skirtų sistemų darbo greitis ir patikimumas.