UNIVERSITY OF JOENSUU
COMPUTER SCIENCE AND STATISTICS
DISSERTATIONS 18

MAXIM MOZGOVOY

ENHANCING COMPUTER-AIDED PLAGIARISM DETECTION

ACADEMIC DISSERTATION

To be presented, with the permission of the Faculty of Science of the University of Joensuu, for public criticism in Louhela Auditorium of the Science Park, Länsikatu 15, Joensuu, on November 14, 2007, at 13 o'clock.

UNIVERSITY OF JOENSUU
2007

Supervisors: Professor Erkki Sutinen
Department of Computer Science and Statistics
University of Joensuu
Joensuu, Finland

Doctor Kimmo Fredriksson
Department of Computer Science
University of Kuopio
Kuopio, Finland

Reviewers: Professor Lauri Malmi
Laboratory of Software Techniques
Helsinki University of Technology
Helsinki, Finland

Professor J. Michael Spector
Learning Systems Institute
Florida State University
Tallahassee, Florida, USA

Opponent: Doctor Kinshuk
School of Computing and Information Systems
Athabasca University
Athabasca, Canada

ISBN 978-952-219-049-9 (paperback)
ISSN 1796-8100 (paperback)
ISBN 978-952-219-050-5 (PDF)
ISSN 1796-8119 (PDF)

Computing Reviews (1998) Classification: E.1, H.3.3, H.3.4, J.1, K.3.1

Yliopistopaino
Joensuu 2007

Enhancing Computer-Aided Plagiarism Detection

Maxim Mozgovoy
Department of Computer Science and Statistics
University of Joensuu
P.O.Box 111, FIN-80101 Joensuu FINLAND
mmozgo@cs.joensuu.fi

University of Joensuu, Computer Science and Statistics, Dissertations 18

Joensuu, 2007, 131 pages

Abstract

Plagiarism is a widespread phenomenon that frequently attracts attention of scholars. Many works are dedicated to plagiarism, including pedagogical, juridical, ethical, and technological aspects and issues.

Computer science does not deal with forming a right attitude to plagiarism nor with developing plagiarism prevention instruments such as honor codes. Computers can be used to detect instances of plagiarism in students' works, helping the teacher to create a fair atmosphere at a classroom. Furthermore, studies indicate that the students are more plagiarism-aware when they know about the usage of plagiarism-checking tools.

This thesis is dedicated to the problem of computer-aided plagiarism detection, i.e. to the development and the use of software instruments that help to reveal plagiarism. The creation of such tools raises specific algorithmic problems that deserve attention. The results covered in this work, include:

- Building the taxonomy of existing plagiarism detection methods according to their speed and reliability characteristics.
- Studying and improving string matching algorithms used in plagiarism detection. Introducing "tokenizers" for the natural language texts, applying natural language parsers for plagiarism detection in order to enhance the quality of the detectors.

- Optimizing the speed performance of string matching based plagiarism detection algorithms by applying a combined fast and reliable scoring scheme. Developing an efficient parameterized matching procedure.
- Developing a fast string matching based plagiarism detection algorithm.

Keywords: plagiarism detection, text similarity detection, string matching, natural language processing, tokenization, parameterized matching.

Acknowledgements

I am very grateful to my supervisors Prof. Erkki Sutinen and Dr. Kimmo Fredriksson for their constant support and inspiration. Numerous creative ideas by Prof. Sutinen and the critical mind of Dr. Fredriksson made this work much better than it could be according to my initial plans.

I wish to thank also Prof. Lauri Malmi and Prof. J. Michael Spector, the reviewers of this thesis, for their valuable comments and suggestions.

The Department of Computer Science and Statistics organized excellent summer schools, which were very important for my PhD studies. The main contributions to these events belong to IMPDET community, so I wish to thank IMPDET's Director of Research Dr. Jarkko Suhonen. The Department also provided necessary financial support that made the participation in the conferences and the summer schools possible for me. As a former IMPIT student, I am extremely grateful to all people involved into this program, especially to our coordinator Wilhelmiina Hämäläinen.

Sincere thanks to all my co-authors. No research is possible without collaboration, and some joint work was really pleasant for me. I hope this fruitful cooperation will be continued in the future.

Maxim Mozgovoy, October 2007

List of Original Publications

[MFWJS05] M. Mozgovoy, K. Fredriksson, D. White, M. Joy, and E. Sutinen. Fast Plagiarism Detection System. *Lecture Notes in Computer Science*, vol. 3772, 2005, p. 267-270.

[Mozgovoy06] M. Mozgovoy. Desktop Tools for Offline Plagiarism Detection in Computer Programs. *Informatics in Education*, vol. 5(1), 2006, p. 97-112.

[FM06] K. Fredriksson, M. Mozgovoy. Efficient Parameterized String Matching. *Information Processing Letters*, vol. 100(3), 2006, p. 91-96.

[MTK06] M. Mozgovoy, V. Tusov, V. Klyuev. The Use of Machine Semantic Analysis in Plagiarism Detection. *Proc. of the 9th International Conference on Humans and Computers*, Japan, 2006, p. 72-77.

[MKK07] M. Mozgovoy, S. Karakovskiy, V. Klyuev. Fast and Reliable Plagiarism Detection System. *Proc. of FIE'07 Conference*, 2007.

[MKS07] M. Mozgovoy, T. Kakkonen, E. Sutinen. Using Natural Language Parsers in Plagiarism Detection. *Proc. of SLaTE'07 Workshop*, 2007.

Contents

1. Introduction

Webster's Encyclopedic Unabridged Dictionary of the English Language defines plagiarism as "the unauthorized use of the language and thoughts of another author and the representation of them as one's own" [Webster01]. Plagiarism in the universities and colleges, usually expressed in copying another student's homework or computer program, is a permanent subject of attention of any responsible administration, being directly related to the overall quality of education. That's why computer-aided plagiarism detection is mainly used in educational institutes.

It would be incorrect to reduce all plagiarism-related issues to the process of catching and punishing a person who showed academic dishonesty. Plagiarism studies deal also with plagiarism prevention — a corpus of actions aimed at reducing both the possibilities to plagiarize, and the will to plagiarize (see Fig. 1.1).

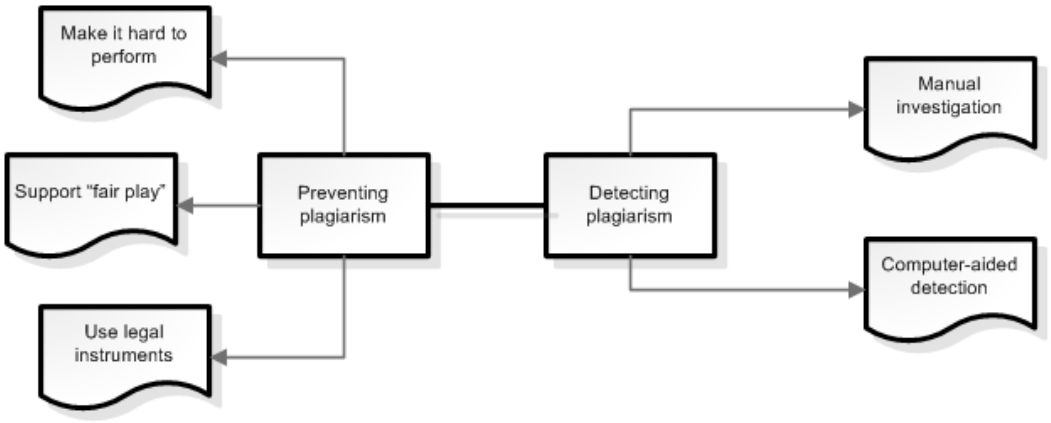


Fig. 1.1. Preventing and detecting plagiarism

1.1. Preventing Plagiarism: Technical, Cultural, and Legal Issues

There are three main kinds of plagiarism prevention measures:

- To make plagiarism technically hard to perform by preparing individual exercises, concentrating on the classroom work, and by using software tools.
- To support "fair play" among students by forming their positive attitude to academic honesty principles, and to explicitly formulate "fair" and "unfair" techniques.
- To publish legal documents such as honor codes and university regulations, stating strict punishments for plagiarism.

1.1.1. Making Plagiarizing Hard to Perform

The most straightforward suggestion for a teacher is to create a unique assignment for each student. In this case no solution can be reutilized. Obviously, while it may work for small-sized groups, such a method is impractical for larger universities.

Hwang and Gibson [HG82] propose to provide a separate classroom quiz for every assignment. The resulting grade in this scheme is the sum of the assignment grade and the quiz grade. The students who have done the tasks on their own are expected to get higher grades for the quiz; also a large difference between these two grades can be an indication of plagiarism.

Wiedemeier [Wiedemeier02] suggests giving more freedom to the students in specifying their tasks, at the same time keeping the same general requirements. For example, in case of computer literacy courses the students can be asked to prepare MS Word letters intended for different events (like birthday invitation or meeting announcement), using fonts, styles and formatting tools specified by the teacher.

Some technical instruments can also help to prevent plagiarism. Vamplew and Dermoudy [VD05] even propose to enforce students to use a special Anti-Plagiarism Editor (APE) that keeps track of all potentially unfair actions, such as massive copy & paste operations.

1.1.2. Promoting Fair Play Principles

It may seem unobvious, but many cases of plagiarism have an unintentional nature, or are caused by low academic culture and lack of attitude to honesty principles.

Teachers note that in many cases plagiarism occurs because students do not know how to cite properly, or because they want to avoid redundant quotations [Brainard05]. Keeping this fact in mind, university administration can provide courses on basic scientific writing style. There are several guidelines dedicated to proper writing and citing [TW04, Turabian96].

The educational and cultural background of the students also can considerably affect their attitude to plagiarism. The study [AKVMVU04] discusses the attempt to compare opinions on plagiarism-related issues of Finnish and Russian students, studying in Lappeenranta University of Technology in Finland. The survey showed that cheating is much more widespread among Russians, mostly due to the peculiarities of educational system in Russia that pays little attention to academic honesty. Sometimes students even do not recognize that they are doing something wrong. Similar cases are reported in [IHBW03]. Thus, instruction may be needed to form the right attitude to plagiarism among newcomers.

It should be also emphasized that possible academic dishonesties are not limited to plagiarism. These actions include practicing unauthorized collaboration, using cribs, substituting other students on the exams, falsifying results [Bliwise01]. Therefore, it is important to develop high ethical standards in university environment from other points of view, too.

1.1.3. Developing Honor Codes and Dishonesty Policies

While some measures aimed at forming the right attitude to plagiarism among students can be effective, studies show that the strongest factor affecting the level of academic honesty in the university is a clear dishonesty policy, maintained by the administration. Correspondingly, the students are much more likely to cheat if they feel that no legal consequences will follow in cases of discovered dishonesty incidents [HCMS02]. Moreover, consistent administrative sanctions form negative attitudes to dishonesty among students and personnel, so the perceived value of fair play

increases [CCWS99]. Harris [Harris94] suggests to constantly remind students of the existence of policies in order to prevent plagiarism.

Having consistent policies, it also makes sense to inform the students that they have a high probability of being caught on cheating in order to reduce plagiarism. Braumoeller and Gaines [BG01] claim that the number of plagiarism incidents has reduced after notifying the students that their work is going to be checked with a software tool.

It should be noted also that many university honor codes expect students to watch for potential dishonesties, which means reporting the cases of unacceptable actions to administration [Meizlish05]. However, reporting plagiarism is often considered as rating, especially among students who cheat [CJ05], so the effectiveness and ethical justification of this measure can be doubted.

1.2. Detecting Plagiarism

Even in institutions practicing good dishonesty preventing methods, plagiarism still can occur. Therefore, it is important to be able to detect unfair actions when they happen. Moreover, as it was already mentioned, the use of plagiarism detection methods is a good deterrent measure itself.

Many plagiarism types can be now discovered only by thorough human investigation. Other types are detectable with the help of computer instruments.

1.2.1. Investigating Plagiarism Cases Manually

Let us return to the definition of plagiarism as to "use of the language and thoughts of another author". To make plagiarism in a certain work evident, it is necessary to find the origin of separate unquoted sentences and thoughts. Sometimes it can be hard even for a skillful teacher, and completely impossible for the computer.

A student can copy an old and obscure work that is not easily accessible. A text can be obtained by translating a work composed in a foreign language. Translation takes time, but it is still incomparable to the time needed to perform the assignment in a proper way. A plagiarizer can just hire somebody to complete the assignment.

In such cases a teacher has only one option: to use common sense. There are various guides (e.g. [JK03]) that list several clues helping to identify possible plagiarism. For example:

- look for strange formatting elements such as unexpected line breaks or text blocks of unusual font;
- examine citations: they can be old — a potential indication of copying an old work with even older citations — or inconsistent;
- analyze writing style: it should be consistent with previous works by this student;
- examine content: it can be created by gluing paragraphs of different style and topic; there can be inconsistencies in the text.

It is harder to discover whether the paper was authored by a paid external person. One of such cases followed a true detective investigation by the university, and ended with a court action and newspaper headlines [Zobel04].

1.2.2. Using Computer Tools to Detect Plagiarism

The cases of plagiarism, which are hard to reveal, are not very frequent. Usually, students copy & paste an accessible source, possibly trying to hide plagiarism by rearranging the text. The use of computer tools has been proved to be a fast and reliable method of detecting such instances of plagiarism.

A teacher should select the corresponding software solution depending on the type of course taught. Normally, a plagiarism detection system is specially designed for program code or natural language processing. The tool can also be aimed either at a global search for Internet-accessible documents that are similar to the submitted ones, or at a thorough analysis of a local collection of documents for possible internal borrowings. These types of detectors are called open systems and hermetic systems correspondingly.

A teacher, suspecting the possibility of copying the works from the Internet, can use one of the open systems to discover the original document. Such detectors are special

variations of Internet searching engines. They maintain huge databases of books and journals, and index Internet pages. The user has to upload a collection of documents, and they will be checked for potential occurrences of plagiarism.

Since dealing with such large amounts of data needs significant computational resources, only a few large-scale open systems exist. The relevant examples are Turnitin [Turnitin07] and MyDropBox [Mydropbox07]. On the other hand, a system can utilize a third-party searching engine for detection, and consolidate results. This is the method by which EVE2 [Eve07] works. Open systems are usually commercial services, and the payments are set on the monthly subscription basis.

In practical computer science courses, copying from the Internet is not as common as utilizing a program of another student. This happens because it is hard to find a computer program that does exactly what was asked, while most students in the group usually get similar assignments. In such cases it makes sense to compare individual files in the submission against each other.

Since comparing individual files of small-sized collections does not require high-performance computational devices, there are much more hermetic systems available, and most of them were developed in the universities. Probably, the most popular and well-known hermetic systems are MOSS [SWA03], JPlag [PMP02], and YAP3 [Wise96]. Recently, an open-source project Plaggie [ASR06], based on the similar approach to JPlag, was released.

From the algorithmic point of view, there is a serious difference between open and hermetic systems. Open systems are similar to Internet searching engines. They have to deal mostly with data storage problems, parallel and distributed computing, indexing and performance issues, which are not directly related to file-file comparison techniques. Arguably, the authors of open systems should have not so much freedom in selecting a comparison algorithm. Such algorithms should be necessarily fast; otherwise, the service will be impractical.

Being relatives of file-file comparison routines, such as UNIX `sdiff` command, hermetic systems serve as a playground for researchers to test various similarity detection techniques. There are plenty of projects, and quite many of them utilize novel or ad hoc algorithms. Speed and space requirements are not so strict for local collection analysis, so advanced techniques can be applied. For example, most present projects implement special methods to fight the attempts to hide plagiarism. A plagiarizer can try to substitute words with their synonyms (in case of natural language text collections) or rename variables (in case of computer programs). Modern plagiarism detection systems can overcome these tricks by using special thesauri and tokenizers (see Section 5.4.1).

The existence of a variety of online course management systems, such as Moodle [Moodle07] and WebCT [Webct07] raises the question of automatic plagiarism detection for all students' submissions being uploaded. Surprisingly, in most cases such a feature is not offered. Currently, this functionality is provided by the BOSS course management system [Boss07] and it seems natural to expect similar capabilities in other course management or collaborative work environments in the future.

It should be also noted that a plagiarism detection tool can only provide the evidence of similarity between pairs of documents. It cannot prove the existence of plagiarism. Therefore, it is important for the human evaluator not to base the final judgment solely on the similarity scores, generated by a software tool.

2. Research Contribution

As shown in the Introduction, the problem of plagiarism raises numerous research questions in different scientific fields, so any single work has to be limited to a certain subtopic. This thesis is primarily focused on hermetic systems for computer-aided plagiarism detection.

2.1. Research Questions

Even a smaller subfield of hermetic software tools for plagiarism detection has much to offer for researchers. There are issues related to speed, reliability, usability, specialization for the documents of a specific nature, etc. Our studies are limited to the following questions:

1. **Which kinds of plagiarism detection systems exist today?** There are numerous hermetic plagiarism detection systems. However, not all of them implement completely novel methods and algorithms. Most systems can fall into specific categories of speed, method type, and expected reliability. These categories have to be revealed.
2. **Is "speed / reliability" trade-off inevitable in plagiarism detection?** Most systems are based on comparison schemes that can be either "fast" or "reliable". Reliable content comparison takes time, while a fast fingerprint-based approach can be less time consuming. The problem is to discover, is it possible to design a system that combines the positive sides of both methods, i.e. which is fast and reliable at the same time.
3. **How can the quality of computer-aided plagiarism detection be improved?** Advanced plagiarism detection systems use auxiliary algorithms to fight against possible attempts to hide plagiarism. The question is to examine these methods to find how they can be developed further and modified to be applicable in adjacent areas.

2.2. Contribution of Individual Papers

The paper [Mozgovoy06] analyzes existing plagiarism detection systems, focusing on hermetic file collection analysis. Different approaches to plagiarism detection are examined. As a result, the work proposes a classification of hermetic systems according to algorithms used, execution speed, and expected reliability. The problem of definitions of plagiarism is also analyzed in this paper.

The paper [MFWJS05] introduces a novel plagiarism detection system, mostly designed at the University of Joensuu. We tried to combine the reliability of content comparison methods with speed, achieved by using indexed data structures. As a result, we have developed a system that is both accurate and fast. These claims are supported by experimental data. The algorithm was developed by Kimmo Fredriksson and me. Later I improved it, and implemented the whole system. Mike Joy and Daniel White kindly supplied us with the tokenization module taken from their detection system Sherlock, and helped to embed it. Erkki Sutinen was supervising the project, and also proposed some useful ideas.

Though the system introduced in [MFWJS05] can accurately compare submitted files, it has inherent limitations in finding actually copied substrings of text. In other words, the system is better to use for scoring documents, but not for close examination of pairs of similar files. The paper [MKK07] shows how such a fast filter can be combined with slower but accurate file-file comparison module. The obtained system grades all the files using a fast module, and then applies accurate content comparison algorithm only for pairs of similar files. Therefore, the resulting speed is still fast, while the quality of detection is noticeably higher. The combined system was primarily designed by me. Sergey Karakovskiy was working on experimental part: all tables and graphs are his contributions. Vitaly Klyuev helped with writing; he also supported the work with valuable statistical calculations.

Any plagiarism detection system that deals with source code files has to be able to fight against simple techniques of plagiarism hiding, such as variable renaming, adding or removing comments, and altering text formatting. It can be done, for instance, by

utilizing a well-known method called *parameterized matching* (p-matching). In paper [FM06] we design a parameterized matching algorithm based on a widely used shift-or string matching method. The resulting solution has lower computational complexity than standard p-matching algorithm. This theoretical part was mostly performed by Kimmo Fredriksson. I was working on implementation and experiments.

While plagiarism in source code can be hidden by means of variable renaming, plagiarism in natural language is concealed with the help of synonyms. In paper [MTK06] we develop a possible technique to fight against such kinds of actions. The system, described in [MTK06], uses a special type of thesaurus which converts synonymous words to identical sequences of tokens. The basic idea of using such a module in plagiarism detection was mine. Vitaly Tusov, who works in the field of natural language processing, provided a good parser/thesaurus module. Vitaly Klyuev helped in evaluating the system and in writing the paper.

Synonyms are not the only allies of a plagiarizer. Plagiarism can be effectively hidden by swapping words and phrases, where possible. Many plagiarism detection systems are tuned to detect matches longer than a certain threshold only. So by swapping words, a plagiarizer can split a continuous "borrowing" into a list of isolated chunks that are not marked by the detection program. This technique can be overcome by using a natural language parser that converts sentences into special structures with predefined order of words. Such parser application is considered in [MKS07]. In this research, I worked on both theoretical and experimental parts, Tuomo Kakkonen adapted Stanford Parser for the document preprocessing algorithm, and Erkki Sutinen was supervising the process.

Table 2.1 summarizes the contribution of the individual papers to the thesis. The relations between the papers are shown in Fig. 2.1.

Table 2.1. Contribution of the individual papers.

Paper	Research Questions	Chapters
[MFWJS05]	2	6
[Mozgovoy06]	1	3, 4
[FM06]	3	6
[MTK06]	3	5
[MKK07]	2	6
[MKS07]	3	5

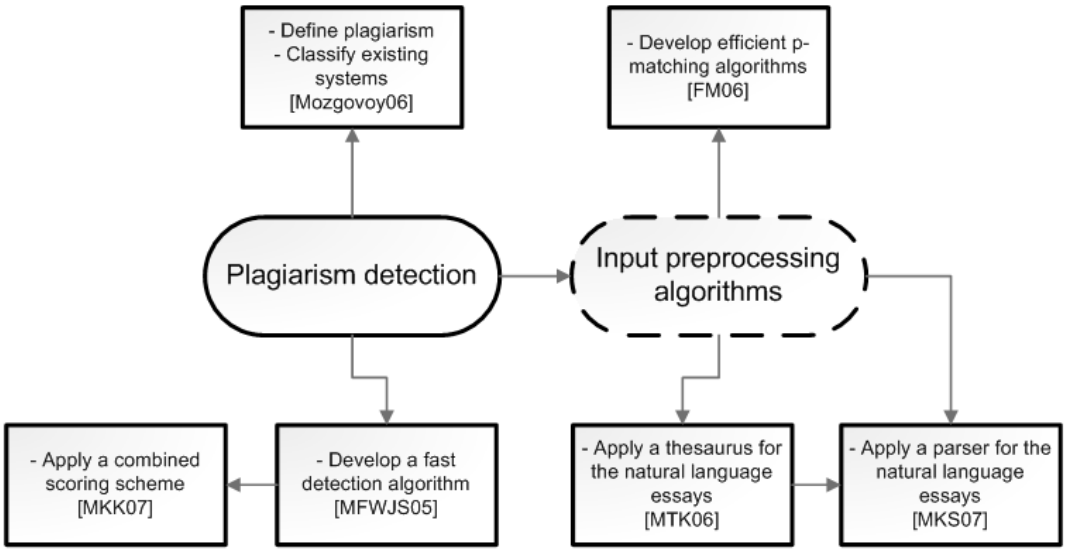


Fig. 2.1. Relations between the individual papers

3. Definitions of Plagiarism

The most common definitions of plagiarism come from dictionaries — one example is cited in the introductory section of this work. However, all of them are not sufficiently formal to be directly implemented in a computer system. One may think that omitting the exact definition does not make problems, since *computer-detectable* plagiarism is something almost self-evident; at least, documentation of many systems often just claim that plagiarism is detected without providing precise specification of the system's capabilities. On the highest level of abstraction it is almost true, but there are additional details on the lower levels.

Obviously, a computer can detect plagiarism if and only if it treats a number of documents as similar. So in this case the term "plagiarism" is used as a synonym to the term "similarity", while similarity is calculated by means of a certain file-file comparison function. In the real world, there is, at least, one difference: some documents are original compositions, while others contain unauthorized borrowings. Usually we do not expect that a computer knows who is an author, and who is a plagiarizer, so generally we accept this kind of behavior. Moreover, a computer usually cannot distinguish true plagiarism from proper citing, and we also understand that. Keeping these facts in mind, it can be even noted that the established term "plagiarism detection system" is imprecise. It emphasizes the primary application of a tool, but may lead to misunderstanding, since a real capability of a system is similarity check.

So it is clear that even the first simplification that is done literally by all plagiarism detection systems — reducing plagiarism to similarity — is usually not stated explicitly, though there is an obvious difference. Furthermore, by using any specific methods to calculate document similarity, the system introduces more simplifications and assumptions that can take us far away from the initial concept.

Nevertheless, any specific system implements a certain method, which calculates a measure of plagiarism. Several authors try to identify these actions quite informally: "our goal is to identify files that came from the same source or contain parts that came

from the same source" [Manber94]. Later in the same work Manber proposes an explicit criterion for plagiarism: "we say that two files are similar if they contain a significant number of common substrings that are not too short". In fact, Manber expressed what most content-comparison systems actually do. So the difference between common-sense plagiarism, and "plagiarism, detectable via software solutions" becomes evident. For example, documents that include proper quotations contain common substrings with the source they cite, but they are not plagiarized. On the other hand, Prechelt et al. [PMP02] note that high file-file similarity, revealed by performing straightforward content comparison, is usually considered as a good indication of plagiarism by human experts.

Misunderstanding of the basic work principles of plagiarism detection systems leads to strong criticism expressed by the disappointed users. For example, Turnitin [Turnitin07] is criticized for not distinguishing proper quotations from plagiarized text, for treating self-quotations (when a student has published online some parts of the work before) as plagiarism, for inconsistence (when some known quotes are marked, while other ones are not), and so on [Gerald06]. One of the reasons for such attacks is the lack of a proper definition of "Turnitin plagiarism". The users have to consider this system as a black-box that has its own opinion of what is plagiarism. Turnitin is also criticized for a variety of ethical reasons, such as promoting presumption of guilt and possible violations of students' intellectual property rights [Carbone01, Glod06], but these issues are outside the scope of this work.

By accepting the close relationship between plagiarism detection and content comparison, the authors of plagiarism detection systems can make use of existing string matching algorithms, originally intended for other file-file comparison tasks. For example, parameterized matching [Baker97] was designed to find duplicating code blocks in large software systems (indicating more likely the need for refactoring, not the presence of plagiarism). Analogously, UNIX `diff` tool that outputs the differences between the two files is commonly used to create ***patches*** for transforming the first file into the second one.

It is unlikely that a conventional plagiarism detection system will be able to distinguish true plagiarism from file-file similarity in the foreseeable future, so generally we have to accept plagiarism-as-similarity concept, when dealing with software tools. A human evaluator just should be aware of this issue, as stated above.

4. Hermetic Plagiarism Detection Systems

The existence of a variety of plagiarism detection systems raises a question of possible classification. Such a taxonomy can help teachers to select the most appropriate software solutions for their needs. It can be usable for researchers, too, by revealing core algorithms of different tools. Researchers, who work with string matching, may consider plagiarism detection as one of possible applications of their achievements. The classification of plagiarism detection systems is discussed in [Mozgovoy06]. Here we will consider only the most important classes, shown in Fig. 4.1.

Naturally, there is no single criterion to perform classification. In the work of our research group, we tried to improve existing solutions in terms of reliability and performance. Therefore, we were primarily interested in evaluating current approaches according to these parameters. Also, we were mostly aimed at processing students' textual submissions, so the problem of plagiarism in non-plain-textual data remains out of scope of this work. Being interested in advanced text preprocessing algorithms, we focused mainly on string matching-based systems.

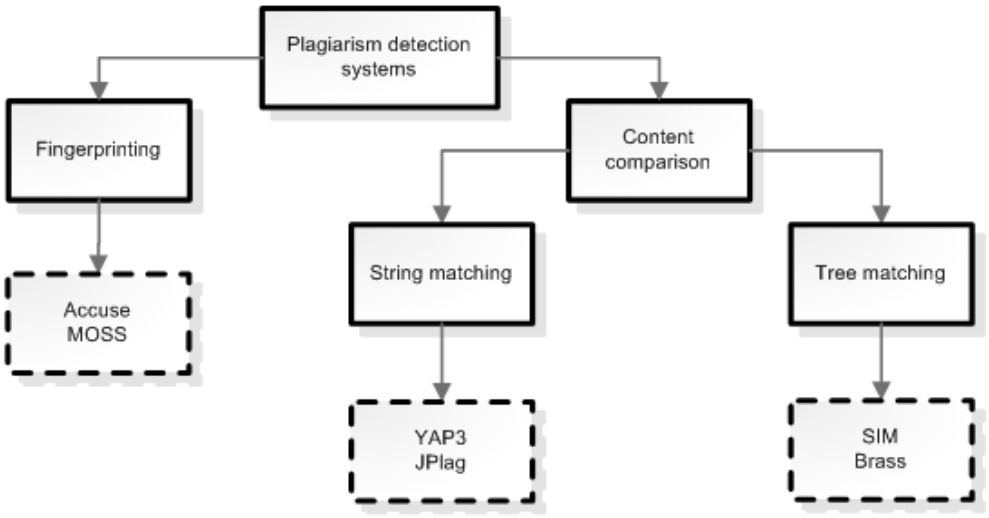


Fig. 4.1. Simplified classification of plagiarism detection systems

It should be mentioned that most hermetic systems are either universal, i.e. can process text documents of any nature, or are specially fine-tuned to detect plagiarism in source code files. Natural language processing technologies are usually left for open systems. This fact can be explained, keeping in mind that the students are more likely to copy computer programs from each other, while natural language essays are easier to find in the Internet. A student can ask another person to solve a programming task, but it is difficult to find a ready-made solution for a particular assignment in the Web. Meanwhile, natural language compositions on a variety of common topics are widely available online.

4.1. Fingerprint-Based Systems

The core idea of fingerprinting is to create *fingerprints* for all documents in the collection. Formally, fingerprint is a short sequence of bytes that characterizes a longer file. For example, fingerprints can be obtained by applying any hash function to a file. In plagiarism detection systems fingerprints are usually more advanced than simple hash codes: each fingerprint contains several numerical attributes that reflect the structure of the document. Typical attributes include average number of words per line, the number of unique words, and the number of passages. If two fingerprints are close to each other (according a given criterion, usually provided as a *distance function* in mathematical sense), the corresponding documents are treated as similar.

Formerly, fingerprints were used in *attribute counting systems*, which represent the early era in computer-aided plagiarism detection (see Fig. 4.2). The very first project of this kind [Ottenstein77] used Halstead's metrics [Halstead77] to score file-file similarity. In the following years, plenty of metrics have been tested, and more advanced systems were built [Grier81, FR87]. Nowadays it is generally believed that attribute counting is inferior to content comparison, since even small modifications can greatly affect fingerprints. As a result, later systems usually do not follow this technique [VW97].

There are several recent projects that combine fingerprinting with elements of string matching. Such an approach is used e.g. in the notable MOSS program [SWA03]. Other approaches to fingerprinting include the use of general-purpose text retrieval techniques such as LSA [Nakov00] and vector space model [SE06].

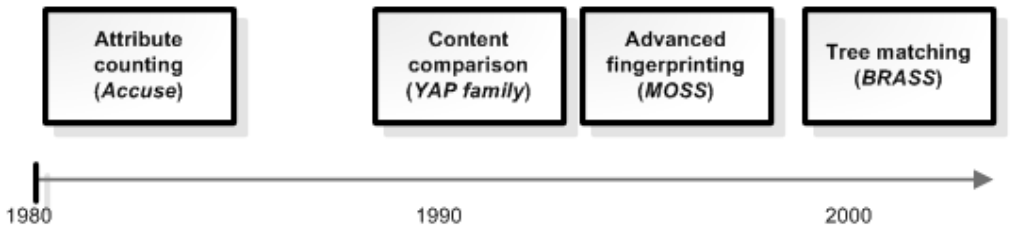


Fig. 4.2. The timeline of hermetic plagiarism detection algorithms

4.2. Content Comparison Techniques

Content comparison techniques deserve a more in-depth study, being the cornerstones of an overwhelming majority of present plagiarism detection systems. There are different algorithms aimed at file-file comparison, varying in terms of speed, memory requirements and expected reliability. Generally, most schemes follow already cited Manber's definition of similarity: "we say that two files are similar if they contain a significant number of common substrings that are not too short".

4.2.1. String Matching Based Content Comparison

Usually, content comparison systems work according to the following algorithm:

```
FOR EACH collection file F
    FOR EACH collection file G, F ≠ G
        Calculate similarity between F and G
```

The core function that calculates similarity varies from one system to another. String matching based methods compare files by treating them as strings. This approach, though, usually does not take into account the hierarchical structure of the computer program, considering it as raw data. The situation for natural language documents is the same.

The similarity score calculation is tool-dependent. For example, FPDS [MFWJS05] determines similarity between files `F` and `G` using the following formula:

```
sim(F, G) = MatchedTokens(F, G) / TotalTokens(G)
```

Here `MatchedTokens(F, G)` denotes the number of tokens of the file `F`, which are matched in the file `G`.

Early string matching based plagiarism detection systems like YAP [Wise92] used simple mechanisms, such as line-by-line comparison of two files under edit distance [Levenshtein66]. Later, more advanced general string matching methods, such as Smith-Waterman algorithm, were applied [Irving04].

One of the most popular file comparison methods is Running-Karp-Rabin Greedy-String-Tiling (RKR-GST) algorithm, implemented in a variety of systems, including YAP3 [Wise96], JPlag [PMP02], and Plaggie [ASR06].

The basic aim of this algorithm is to find a maximal tiling for a pair of given files, i.e. the joint coverage of non-overlapping strings that includes as many tokens from both files as possible (see Fig. 4.3). Moreover, the use of short tiles of length smaller than a specified threshold is forbidden. The problem of constructing such coverage appears to be NP-complete [Wise94], so practical implementations always rely on heuristic assumptions. RKR-GST algorithm utilizes greedy heuristics, assuming that longer matches are more valuable than shorter ones.

The RKR-GST algorithm can be outlined 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 [KR87]. 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.

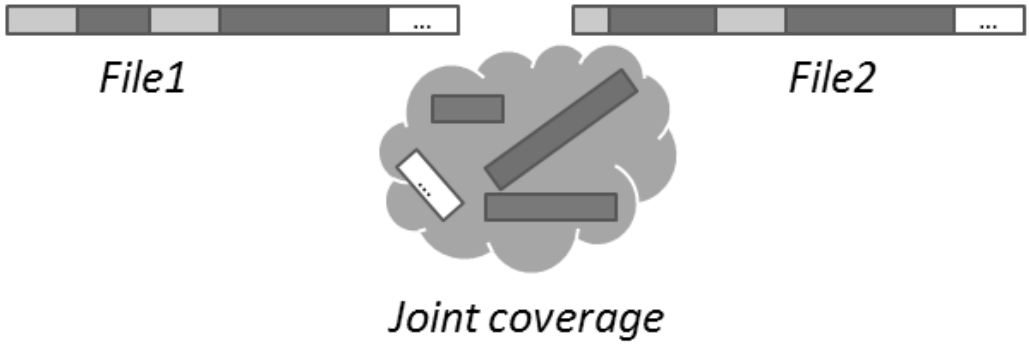


Fig. 4.3. Joint coverage of two files

Our own string matching based system FPDS [MFWJS05] tries to get rid of computationally inefficient file-file comparison by using a special index structure, based on a suffix array, and by applying several ad hoc heuristics. This project is described in Section 6.3.

4.2.2. Parse Trees Comparison

Clearly the actual file content better describes its structure than any kind of fingerprints. However, raw content data is not structured, while most kinds of documents do have an internal organization. Natural language texts are divided into sections, subsections, paragraphs and sentences. Source code files contain classes, functions, logic blocks and control structures.

While it is still not completely discovered how one can utilize such kind of information effectively, there are systems that implement comparison techniques, directly or indirectly based on the file structure.

The idea of using parse trees of computer programs was firstly implemented in the Sim utility [GT99]. Sim relies on ordinary string matching methods, but instead of comparing source files, it calculates file-file similarity between textual representations of corresponding parse trees. Hence, a parser is used as a front-end for the string matching algorithm. This approach was later applied in our own projects, too [MTK06, MKS07].

The pure tree comparison procedure was implemented in the Brass project [BNH04]. 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 carried out so far. For example, it is still unknown how such a complex analysis of input files influences the final results — i.e., it is undiscovered whether parse trees comparison is superior to usual string matching. Furthermore, any practical (in terms of execution speed) tree comparison routine requires optimizations like greedy heuristics. For now, it is not clear how these techniques affect reliability.

5. Input Preprocessing Algorithms

As stated before, most content comparison techniques follow Manber's criterion of plagiarism. In practice, though, there are situations when it turns out to be inapplicable, i.e. when dissimilar input files (according to Manber's definition) include clear instances of plagiarism. For example, equivalent computer programs can greatly differ in comments and in identifier naming conventions. However, such distorted similarity can be revealed by utilizing specialized file preprocessing algorithms.

5.1. Hiding Plagiarism

Obviously, a naïve type of copy & paste plagiarism can be easily detected with most methods of content comparison. However, there are tricks that can be used to hide plagiarism, or, speaking more precisely, to make it harder to note.

For natural language texts such actions can include rewording and paraphrasing. The use of different words does not eliminate the existence of plagiarism, since the use of "thoughts of another author" [Webster01] should be treated as an illegal action as well. At the cost of hard effort, a plagiarizer can successfully make the case extremely difficult to reveal. One notable example from the author's past experience is a small group of international students, studying in English, who applied the following method: take an article, translate to their native language, and pass it next to the fellow student to translate it back into English. Obviously, the resulting text can greatly differ from the original one, especially if the source text was composed by a native English speaker. Though being labor-consuming, this type of plagiarism hiding seems to be effective enough: a teacher can reveal it only by means of indirect hints.

For computer programs, the situation is rather different. On one hand, computer programs are more structured; therefore there is not so much possibility to paraphrase them. Moreover, such paraphrasing requires clear understanding of the underlying algorithm, which is not usually expected from a plagiarizer. On the other hand, there are

very simple methods like variable renaming, which are easy to perform without any understanding of a computer program.

Possible actions of plagiarism hiding in computer programs are listed in works [JL99] and [Jones01]:

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.

This list entails a broader definition of plagiarism, suitable for computer programs: 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 [Mozgovoy06].

In any case, it is clear that direct implementation of Manber's criterion is useless when the plagiarizer performs some "post-processing" actions after copy & paste.

5.2. Overcoming Hiding Techniques by Preprocessing

Most current plagiarism detection solutions somehow transform input files before actual comparison to fight against the aforementioned tricks. Usually this process includes tokenization and/or parsing. In this section we will talk in more detail about different kinds of preprocessing algorithms, applicable in a variety of plagiarism

detection projects, regardless of their comparison scheme. The type of preprocessing and the algorithms used do depend on the nature of the input documents (see Fig. 5.1).

It should be mentioned that not all the plagiarism hiding techniques are equally easy to reveal. In the above list, the simplest tricks to fight are 1 and 2: the preprocessing function only has to eliminate all the comments and redundant white spaces to make the instances of plagiarism clear. The techniques 3, 7, and 9 can be overcome by means of tokenization, described in the next section. The remaining tricks are resolved with an advanced string matching algorithm, such as RKR-GST that can detect matches even if they are reordered.

It should be noted that input preprocessing algorithms and heuristic string matching procedures can lead to *false matching* and/or *mismatching* blocks of text. False matching occurs when a preprocessor converts two different text strings into equal character sequences. Sometimes such conversion makes sense, but in other cases it can be incorrect. Mismatching problem arises when two equal strings are not matched during comparison. It can happen if the file-file comparison routine utilizes a certain heuristics like greedy matching, so the equal blocks of text can remain unprocessed.

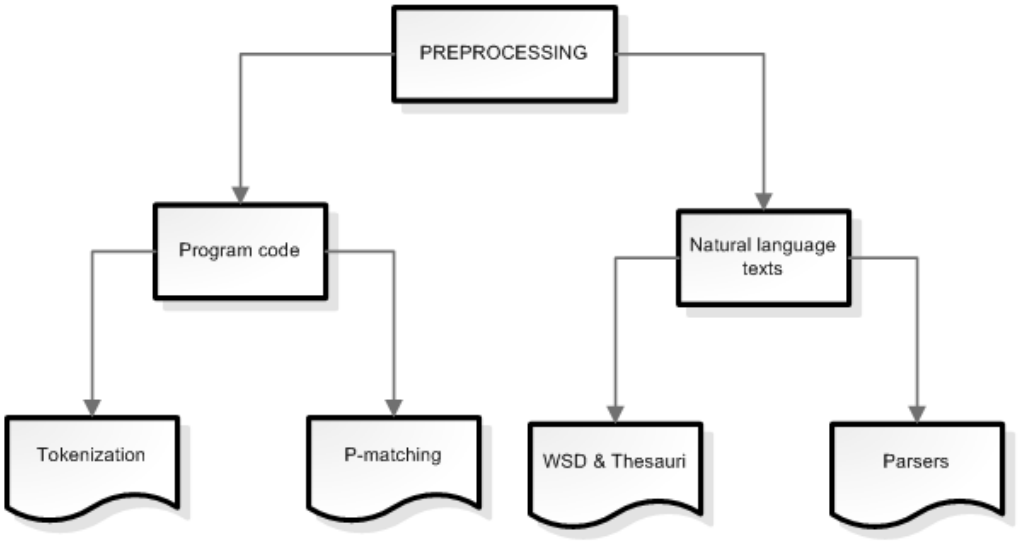


Fig. 5.1. Instruments for text preprocessing

5.3. Preprocessing Source Code

Let us first consider the problem of plagiarism detection in computer programs. It is important to mention that most of plagiarism-hiding techniques deal with lexical changes of the document, not involving any semantic information. Keeping in mind this fact, we can develop a tool that preprocesses input documents, eliminating lexical differences.

5.3.1. Tokenization

Probably, the simplest tool that can be applied in such a case is a *tokenizer*. A tokenizer is a program that takes a computer program as an input, and outputs the corresponding sequence of tokens, (i.e. of minimal meaningful units of a language), representing variables, function calls, statements, and control structures. Usually, tokenizer also removes all comments.

For example, a Java fragment

```
package student;
import wrabble.*;
public class Board implements IBoard
{
    MyTileWord tileword;
    private Tile tiles[][];
```

can be preprocessed into the following sequence, where line breaks and white spaces are kept for better readability:

```
<PACKAGE><IDENTIFIER>
<IMPORT><IDENTIFIER>.*
<MODIFIER><CLASS><IDENTIFIER><MODIFIER><IDENTIFIER>
{
    <IDENTIFIER><IDENTIFIER>
    <MODIFIER><IDENTIFIER><IDENTIFIER>[][]
    <MODIFIER><IDENTIFIER><IDENTIFIER>[][]
```

Technically, a tokenizer is a simplified programming language translator that transforms a program into the corresponding tokenized representation. Such a tool can be quite easily obtained by utilizing numerous freely available parser generators with ready-made grammars. For instance, our system [MFWJS05] relies on JavaCC parser generator [Enseling00].

As seen from the example, the preprocessor substitutes every variable identifier with token <IDENTIFIER>, and every numeric value with token <VALUE>. Therefore, if a program contains a line

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

it will be replaced by a sequence

```
<IDENTIFIER> = <IDENTIFIER> + <VALUE>
```

So renaming variables is useless now, since every line of the form "identifier = identifier + value;" is translated into the same tokenized string. Our tokenizer also substitutes all kinds of loop structures by <LOOP> token. Hence, changing loop statement type will not help to hide plagiarism as well.

Usually, a tokenizer preprocesses any single file in O(*n*) time, where *n* is a length of a file. As will be shown, plagiarism detection has a higher complexity, so tokenization should not be a major factor for the overall complexity of the algorithm.

5.3.2. Parameterized Matching

The obvious drawback of tokenization roots in the superficial analysis of input files. Simply speaking, by reducing all variables to <IDENTIFIER>, all values to <VALUE>, and all loops to <LOOP>, we lose significant information about possible differences of the files. Therefore, the system is more likely to detect similarities, introduced by the tokenizer.

To preserve differences in variable names, a *parameterized match* (p-match) algorithm can be utilized. Since this method is a topic of our special interest (see [FM06]), it will be described in more detail later in Section 6.2. Here we will consider only its purpose.

A parameterized match algorithm 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. It is assumed that the algorithm knows which elements are identifiers, and

which are not. For example, parameterized match routine considers the following code fragments as equal:

```
int y = 10;                    int c = 10;
int x = 5;                     int n = 5;
x = myfun(x, y);               n = callfun(n, c);
x += 5;                        n += 5;
```

Without going into details for now, it should be noted that this algorithm can be used together with tokenization to get benefits of both methods.

5.4. Preprocessing Natural Language Texts

Since natural language documents do not have a strict formal structure like computer programs, it is much harder to implement any reasonable preprocessing algorithm. Actually, many plagiarism detection systems analyze input documents "as is", without any preprocessing.

Fortunately, there are no such straightforward methods for plagiarism hiding like variable renaming for natural language texts. Any attempt to conceal plagiarism requires some mental work. Therefore, it is unlikely that all obvious "borrowings" are so carefully disguised that the system will be unable to find, at least, some of them. There are exceptions like careful double translation of the source documents, but the abundance of this technique is doubtful.

Nevertheless, plagiarism hiding occurs in natural language texts, too. So any sensible methods of preprocessing such documents can be beneficial.

5.4.1. Using Word Sense Disambiguation Modules and Thesauri

One of the common techniques of hiding plagiarism is to substitute words with their synonyms. It can sometimes confuse even humans, while computers are completely unable to fight with such tricks if natural language processing algorithms are not applied.

Such word substitution can be considered as a rough equivalent of variable renaming. And if we can write a "tokenizer" that replaces all synonymous words with the same

token, the trick will not help a plagiarizer. Fortunately, there are electronic thesauri available (such as WordNet [Fellbaum98]) that can be used for this purpose. Since words can have different meanings in different contexts (like “minor key” and “door key”), and, therefore, different sets of synonyms, the actual meaning of the word in a given context should be firstly discovered by means of a word sense disambiguation (WSD) method [EK03].

The problem of applying this idea is a topic of our work [MTK06]. For natural language processing, we have used ***semantic analyzer for the Russian language*** [Tusov04], but similar solutions, suitable for other natural languages, can be utilized as well.

A semantic analyzer [Tusov04] is a complex natural language processing system. We use its module that classifies the words of the input text according to the built-in tree of concept classes, currently containing about 1600 elements. A small extraction of the tree is shown in Fig. 5.2.

For example, a class <PHYSICAL-OBJECT> has a subclass <ALIVE>, having, in its turn, a subclass <ANIMAL> that includes classes <ANIMAL-WILD> and <ANIMAL-DOMESTIC>. The current version of the tree reflects the general knowledge about the human environment. It can be reasonable to perform some tuning of the tree for the specific tasks.

The use of such a tree of concept classes can be considered as applying a thesaurus, paired with a WSD module. By processing input documents with a semantic analyzer, we get lists of word classes instead of original words. Next, these files can be graded by using any conventional plagiarism detection system.

The detection quality of this approach was evaluated by analyzing a collection of 350 documents taken from the `NEWSru.com` news server. Most downloaded files are short news messages, falling into one of the following categories: *In Russia*, *In the World*, *Economics*, *Religion*, *Criminal*, *Sport*, and *Culture*. The size of the articles varies from

450 bytes to 19 KB with the median size of about 2 KB. The typical article consists of 8-12 small paragraphs that are made of strict narrative sentences and quotations.

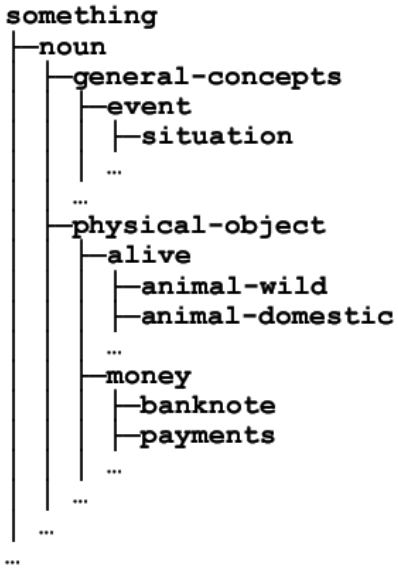


Fig. 5.2. A fragment of concept classes tree

The rationale for selecting newsreels is a high probability of having two or more documents related to the same event or quoting each other. Therefore, the chances of finding, at least, several pairs of similar files are high, too. Direct plagiarism is not expected in this case.

It should be noted that evaluation is a hard issue to deal with in plagiarism detection. For large data sets, thorough human evaluation is almost impossible. We can make use of other detection systems, assuming that the commonly marked subset of files reliably enough indicates truly similar documents. This jury-based approach was used in [MFWJS05], and will be described in more detail in Section 6.3.2. Unfortunately, we had no access to other systems that use natural language processing, so the only option was to use a smaller collection and manual evaluation. The following results are provided in [MTK06].

To evaluate the preprocessor, we ran FPDS [MFWJS05] on original and preprocessed collections of documents. Without preprocessing, the system found 20

relevant pairs of similar documents with at least 4% degree of similarity. Typical examples include:

- A pair of documents about the solar eclipse on 20th of March, 2006. The first tells about the countries where this phenomenon was observed; the second is dedicated purely to the observation of the eclipse in Russia.
- A pair of documents on the weather conditions in Europe. The first is about floods in the EU (Spring 2006); the second contains some weather predictions for the EU, including subsequent floods.
- A pair of documents on rumors about Russian military assistance to the Iraqi government in March of 2003. The first outlines the position of Moscow; the second states the reaction of Washington.

After tokenization the similarity degrees of the same file pairs increased, in most cases, by a factor of 1.5, and remained the same in few cases. Meanwhile, four additional false pairs were detected, but with very low similarity ratios that did not exceed 4-5%.

The consequences of tokenization are especially noticeable in several typical situations. They include:

- Changes to grammar cases in Russian. The phrases *in one of resorts* and *of one of resorts* are not matched due to the changes of the endings of the words. After tokenization they become almost identical.
- The use of distinct words of the same classes in the same contexts in different documents. The phrases *the residence in Greece* and *the residence in Athens* do not match, but do match after the tokenization (*Greece* and *Athens* are translated to the same class <PLACE>).

The latter case is related to mismatches as well. For example, the phrases *Vladimir Putin claimed* and *George Bush claimed* are treated as the same sequence <NAME><NAME><SPEAK> after the tokenization. It may be argued, though, that the

system of classes, currently implemented in semantic analyzer, was not specially designed for plagiarism/similarity detection procedures. A more advanced hierarchy may include a careful taxonomy that minimizes such collisions.

It should be noted that in the simplest case our preprocessing can be considered as a variation of stemming technique that is widely used in information retrieval. This can be very helpful when dealing with languages with developed morphology (like Russian).

5.4.2. Using Parsers

As noted, replacing words with their synonyms can be treated as a rough equivalent of variable renaming technique to hide plagiarism. We can try to overcome it using WSD modules and thesauri. Unfortunately, the plagiarizer can also add structural changes to the document, for example, by modifying the word order where possible. For example, the phrase "light bright sun" can be reworded as "bright light sun" without losing its meaning.

Formally, since most modern plagiarism detection systems are aimed at finding joint coverage of a pair of analyzed files, such a rewording should not hide plagiarism. The system can find all three words ("bright", "light", and "sun") independently. In practice, current systems usually limit the shortest substring to detect. Plaggie [ASR06] and FPDS [MFWJS05] include such "shortest string length to match" fine-tunable constants explicitly. The reason for this decision is a large number of false short matches. If we match every single word of the text, the resulting grade will be a measure of closeness of the authors' vocabularies, not of plagiarism. This holds both for natural language and for program code: any computer program consists of the same statements, and most programmers use the same variable names in similar situations (for example, *i* and *j* for loop counters). The actual value of "shortest string length to match" constant is defined manually. The problem of its automatic selection is a reasonable research topic; however, it has not been investigated properly so far.

Sadly, the use of "shortest string length to match" constant leads to "split match" problem [MKS07]. For instance, if the system is programmed to ignore all matches shorter than two words, it will be unable to detect plagiarism in "light bright sun" case.

Overcoming this issue is a topic of our work [MKS07]. The key idea is an observation that a natural language parser can transform the input sentences into sequences, reflecting the internal structure of the phrases, while not preserving precise order of words. For instance, a parser can recognize homogeneous parts of the sentence, and sort them in alphabetical order. Consider a phrase: "I ate the pizza, the pasta, and the donuts". The plagiarizer can reword it as "I ate the pasta, the donuts, and the pizza", but the detector is insensible to such changes.

For our experiments, we used Stanford Parser, based on an unlexicalized model [KM03]. Like most parsers, it produces an output in the form of dependency trees (see Fig. 5.3). We have designed and implemented a post-processing tool that transforms the dependency trees into a format where the word order has no effect. The format represents the words in the sentence sorted according to their *grammatical relations* (GR) that designate the type of the dependency between the words. Stanford Parser supports most types of GRs, recognized by linguists (see Table 5.1). The complete list of grammatical relations is provided in [MMCM06]. The words inside each GR group are sorted in alphabetical order. The example of the original Stanford Parser output and of the transformed format (for the phrase "I ate the pizza, the pasta and the donuts") is shown in Fig. 5.4.

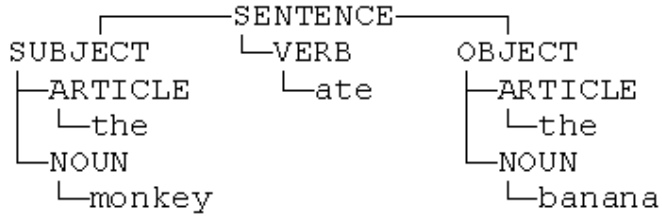


Fig. 5.3. Dependency tree of the phrase *the monkey ate the banana*

Table 5.1. Examples of grammatical relations

Abbreviation	Full name
DET	Determiner
DOBJ	Direct object
NN	Noun compound modifier
NUM	Numeric modifier
NSUBJ	Nominal subject

It should be mentioned that the use of such a parser-based preprocessor for plagiarism detections system introduces a noticeable drawback. Since the initial structure of the sentences is not preserved, the plagiarism detection system that analyses preprocessed files only is unable to highlight matches in the original documents. There are two obvious ways to overcome this problem: either the system should be programmed to highlight the whole plagiarized sentences instead of word chains, or the parser should generate some metadata about the parsed files, helping to restore the links between words in original and parsed files. The latter gives more flexibility, but requires serious modifications of the parser, so we do not consider this scenario.

```
nsubj(ate-2, I-1)
det(pizza-4, the-3)
dobj(ate-2, pizza-4)
det(pasta-7, the-6)
conj(pizza-4, pasta-7)
cc(pizza-4, and-8)
det(donuts-10, the-9)
conj(pizza-4, donuts-10)

[ate, cc[and], conj[donuts, pasta,
pizza], det[the, the, the],
dobj[pizza], nsubj[I]]
```

Fig. 5.4. Stanford Parser (top) and post-processor (bottom) outputs

The evaluation of the idea was performed according to the same scheme as used in [MTK06]. We graded the files collection using FPDS [MFWJS05], then ran the same process for the collection of preprocessed files. The results were analyzed manually.

A collection of 128 messages was obtained from the website of BBC NEWS (`news.bbc.co.uk`). Each message was assigned one of the following categories: *Business*, *Europe*, *Science/Nature*, or *Technology*. The median size of each message (after removing all formatting) is about 2 KB. We have also prepared several files with intentional plagiarism, performed using copy & paste with subsequent change of word and phrase order. It should be mentioned that news is hard to plagiarize with such a method, since the reels are laconic, and do not contain enough adjectives or phrases to swap. However, in free-form essays we used there are more possibilities for such swap-powered plagiarism.

As a result, FPDS found 11 pairs of messages containing vast quotations from each other, and 3 pairs of messages informing about the same event while scoring initial (not preprocessed) collection. The similarity ratios of later pairs were 5%-33%. There were no other similar pairs, according to manual evaluation. The similarity scores of plagiarized free-form essays lay in the range 10%-30%. The inclusion of parser increased similarity ratios for overlapping news messages by 7%-13%. This observation indicates that the combined system has an overall tendency to assign higher similarity grades to the same file pairs. Consequently, two more file pairs of similarity 5% and 7% appeared in the resulting log, but they were not considered as similar by human graders. In practice, it is possible to get rid of incorrectly matched pairs by raising a similarity threshold for final file pair list. For the plagiarized free-form essays the similarity ratios have increased significantly — to 50%-80%. The results were also noticeably affected by the value of "shortest string length to match" constant. The smaller the constant, the less effective is the use of the parser. Large constant values cause higher probability to not identify swap-powered plagiarism, detectable by means of parser.

The use of the parser can also have two positive side effects. First, plagiarism detection over preprocessed files can be performed with larger "shortest string length to

match" value, which entails faster work of the algorithm (see [MFWJS05]). Second, the difference between similarity scores of original and preprocessed collections can itself serve as a good indication of word swaps, and therefore, of plagiarism. However, this supposition is not yet researched.

Another good idea for future research is to utilize tree matching approach [BNH04] for natural language texts. Since parsers do provide tree-like representation of sentences of the documents, it can be sensible to compare these trees directly.

6. Speed and Reliability of Plagiarism Detectors

The existence of variety of plagiarism detection technologies necessarily entails a question about their performance — both in terms of time and space requirements, and in terms of detection quality. Observations show that usually speed is achieved at the cost of reducing detection quality and vice versa. The central part of this chapter is dedicated to the problem of designing plagiarism detection schemes that can be fast and reliable at the same time.

The quality of detection, i.e. the reliability of the system, can be estimated either by human evaluation, or by comparing the output with the results, provided by other plagiarism detection systems. We use both approaches to evaluate our solutions.

6.1. Performance Attributes of Plagiarism Detectors

Common sense suggests a simple principle: the more time is invested to file comparison, the more reliable results can be obtained; in some cases, it is possible to save some time by using more memory space. These guesses are generally supported by work [Mozgovoy06] that studies speed and reliability of various plagiarism detection projects. Its main conclusions can be expressed in several theses (see also Fig. 6.1):

- Classic fingerprinting methods, such as attribute counting, are the fastest, but unreliable [VW97]. If the fingerprint size is constant, the system requires $O(n\mathrm{N})$ time to create fingerprints, and $O(\mathrm{N}^2)$ time to grade the collection. All-against-all comparison is assumed. Here n is the average file size, and N is a number of documents in the collection.
- RKR-GST-based content comparison methods have worst case complexity of $O(n^3)$ for one file-file comparison procedure. However, experimentally discovered average complexity is just $O(n^{1.12})$ [Wise94]. So, the resulting complexity of all-against-all comparison can be estimated as $O(\mathrm{N}^2 n^{1.12})$. First implemented in YAP3 project [Wise96], this approach is still used in recent systems like Plaggie [ASR06], being considered as highly reliable.

- Most preprocessors, such as tokenizers, run in linear time — O(nN) to preprocess the whole collection of N files, and do not make a dominating contribution into asymptotic complexity of any content comparison-based system.
- Advanced tree matching-based projects can require even quadratic (in respect to file length) time for one file-file comparison, so for complete collection scoring the time needed can be estimated as O(N^2n^2) [GT99].

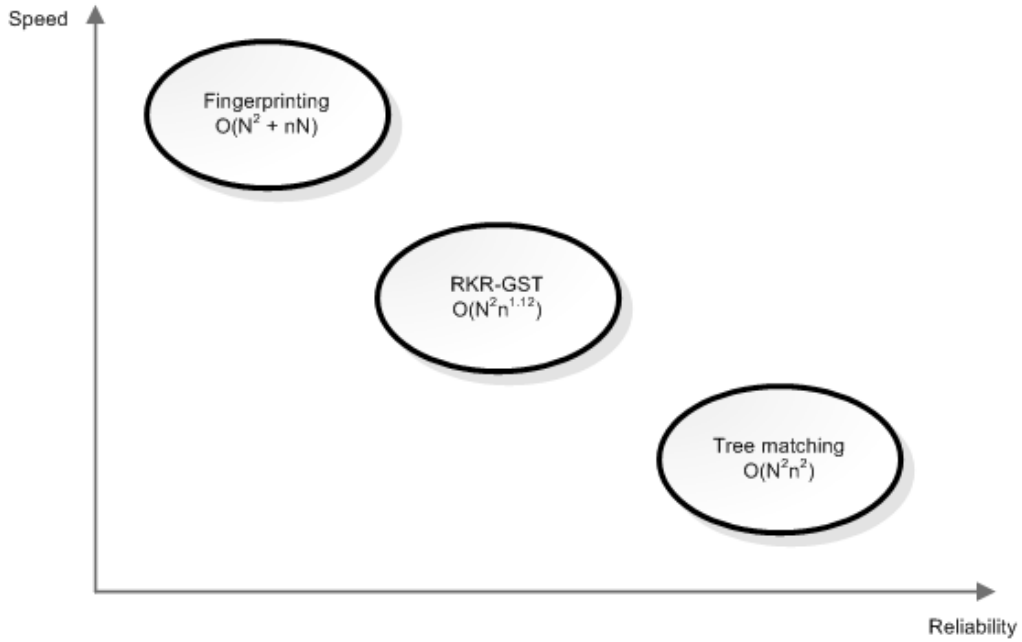


Fig. 6.1. Speed and reliability of different plagiarism detection schemes

Our work [MFWJS05] shows that the core ideas of content comparison-based systems can be paired with specially organized data structures and reasonably applied heuristics in order to get better speed performance. Time is an important factor when large collections are processed. Since the same techniques can be applied to any sort of file-file similarity calculation, not limited to plagiarism detection, the possibility to get a set of documents of considerable size becomes obvious. The importance of scalability is underlined, e.g. in [BTZ07].

6.2. Fast Parameterized Matching

Let us first consider the problem of improving performance of standard implementations of parameterized match algorithm. Only main results will be given here, see [FM06] for more details.

Parameterized matching (see also section 5.3.2) is a special type of string comparison routine that treats two strings as equal if one can be obtained from the other one by a series of regular substitutions of identifiers. It is assumed that a set of admissible identifiers is given, so the program can distinguish identifier tokens from non-identifier tokens. The core idea of parameterized matching can be implemented as a separate preprocessor that erases all information about actual variable names in the input text, while preserving their relational structure. This makes possible to use p-matching-based preprocessor as an advanced tokenizer for plagiarism detection and similarity checking systems, aimed at software source code analysis.

6.2.1. Preliminaries

Formally speaking, parameterized matching deals with strings over two non-intersecting alphabets: the alphabet of *fixed* symbols (Σ, $|\Sigma| = \sigma$), and the alphabet of *parameter* symbols (Λ, $|\Lambda| = \lambda$). These alphabets correspond to non-identifier and to identifier tokens of the programming language. The strings over the alphabet $\Sigma \cup \Lambda$ are called *parameterized strings*. The problem is to find a match of *pattern* string $P[0 \ldots m-1]$ in a *text* string $T[0 \ldots n-1]$. A match is found at position j if and only if for all $i \in \{0, \ldots, m-1\}$ holds that $M_j(P[i]) = T[j + i]$, where:

- $M_j(x)$: $\Sigma \cup \Lambda \rightarrow \Sigma \cup \Pi$ is one-to-one mapping, and Π is an arbitrary set;
- $M_j(x) = x$, for all $x \in \Sigma$.

Such a matching can be reduced to ordinary string comparison by utilizing *prev*() encoding [Baker97]. This algorithm takes a parameterized string, and substitutes all parameter symbols s in it with non-negative integers, equal to the number of symbols since the last occurrence of s in the input string. The first occurrence of the parameter

symbol is represented by zero. So, *prev*() encoding is one of possible practical implementations of $M_j(x)$ mapping. Consider, for example, a pattern P = aaZYZabXYZaX over alphabets Σ = {a, b} and Λ = {X, Y, Z}. Then *prev*(P) = aa002ab055a4. Note that *prev*() function remains the same for any position *j*.

The only problem here is how to maintain *prev*(T[$j \ldots j + m - 1$]) and all dependent values efficiently as *j* increases. The solution is constructed by utilizing the following proposition, proved in [Baker97]:

Let S' = prev(S). Then for S'' = prev(S[j...j + m – 1]), for any j, $\forall i$*: S[i]* $\in \Lambda$ *it holds that S''[i] = S'[j + i] iff S'[i] < i. Otherwise S''[i] = 0.*

In addition to the classical definition of p-matching problem, a task of multiple-pattern search can be considered as well. The rationale is simple: when searching several patterns, a special multi-patterned version of matching algorithm can be significantly faster than a series of consequent calls of the single-patterned routine version.

To the present day, only a few string matching algorithms were generalized to support parameterized matching [AFM94, Baker95, IS96, ST06]. Our work shows how to generalize well-known Shift-Or [BYG92] and Backward DAWG Matching [CCGJLPR94] methods for parameterized strings. Both these algorithms are designed to search a given pattern string P inside a given text string T. For simplicity, we assume that the input alphabets are of constant size.

6.2.2. Parameterized Bit-Parallel Matching

The standard Shift-Or is implemented as a finite automaton [BYG92], which is constructed according to the following scheme (see Fig. 6.2). There are $m + 1$ states in total, marked as $0, \ldots, m$. The state 0 is the initial state, while the state m is the final state. The transitions of the automaton are:

- from i to $i + 1$, for character P[i], $i = 0, \ldots, m - 1$;
- from 0 to 0, for every c $\in \Sigma$.

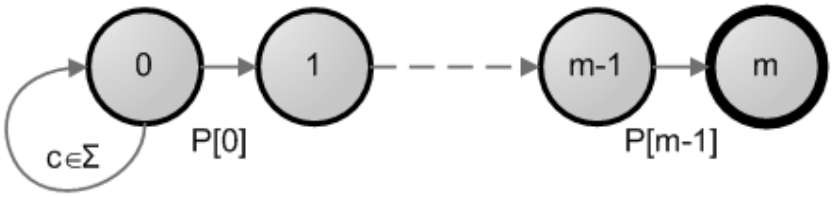


Fig. 6.2. Shift-Or algorithm automaton

The transitions of the automaton are implemented by means of the table B, having one bit-mask entry for each $c \in \Sigma$. The transitions correspond to the single bits of B, and the states are stored in the bit vector D:

- B[c] has i^{th} bit unset $\Leftrightarrow$ P[i] = c, for $i = 0, \ldots, m-1$
- state i is active $\Leftrightarrow$ D has i^{th} bit unset

Initially, all bits of D are set. For each input symbol c, the vector is updated as follows:

```
D = (D << 1) | B[c]
```

To generalize Shift-Or [BYG92] algorithm, we have to make sure that:

- pattern P is encoded using *prev*() function;
- *prev*(T[$j \ldots j + m - 1$]) is maintained in O(1) time per text position;
- bit-mask array B (see [BYG92]) must be built so that all parameterized pattern prefixes can be searched in parallel.

These issues are solved using some nontrivial preprocessing (see [FM06]), finally resulting in Algorithm 6.1 (w is machine word size in bits).

Algorithm 6.1. Parameterized_Shift-Or(T, n, P, m)

```
P' = Encode(P, m)
for i = 0 to σ + m − 1 do B[i] = (1 << m) − 1
for i = 0 to λ − 1 do prv[σ + i] = −∞
for i = 0 to m − 1 do B[P'[i]] = B[P'[i]] & ~(1 << i)
for i = 1 to m − 1 do B[σ + i] = B[σ + i] & (B[σ] | (~0 << i))
D = ~0
mm = 1 << (m − 1)
for i = 0 to n − 1 do
```

```
c = T[i]
if c ∈ Λ then
    c = i − prv[T[i]] + σ
    if c > σ + m − 1 then c = σ
    prv[T[i]] = i
D = (D << 1) | B[c]
if (D & mm) ≠ mm then report match
```

The algorithm runs in O(n·ceiling(m/w)) worst case time. For longer patterns ($m > w$) it is possible to search first w symbols of pattern using our algorithm, then check the whole pattern with the help of a conventional routine only if w-symbol prefix is found. This technique results in O(n) average time.

Shift-Or algorithm can be improved to run in O($n \cdot \log_\sigma(m)/m$) average time, which is optimal [FG05]. The same technique can be applied for our version of parameterized Shift-Or [FM06]. However, for multiple patterns our method is inferior to Aho-Corasick based approach [AC75, IS96].

6.2.3. Parameterized Backward Trie Matching

Another finite automata-based Backward DAWG Matching [CCGJLPR94] algorithm also can be generalized for parameterized strings. We are scanning the text window T[$i \ldots i + m - 1$] backwards. The text window is (backwards) *prev*-encoded as we go, and the read substring of T is matched against the trie of *prev*-encoded suffixes of a reverse pattern P^r. The algorithm remembers the longest matched suffix to continue search if the window size is not enough. If the length of that suffix was l, the next window to be searched is T[$i + m - l \ldots i + m - 1 + m - l$]. The process is repeated until the whole text is scanned (see Algorithm 6.2).

Algorithm 6.2. Parameterized_BDM(T, n, P, m)

```
root = Create_Trie_From_Prev_Encoded_Suffixes_Of(P^r)
for i = 0 to λ − 1 do prv[σ + i] = −∞
i = 0
while i < n − m do
    j = m; shift = m; u = root
    while u ≠ null do
        c = T[i + j − 1]
        if c ∈ Λ then
            c = m − j − prv[T[i + j − 1]] + σ
            if c > σ + m − 1 then c = σ
```

```
            prv[T[i + j - 1]] = m - j
        j = j - 1
        u = child(u, c)
        if u ≠ null and is_suffix(u) then
            if j > 0 then shift = j else report match
    for k = i + j to i + m - 1 do
        if T[k] ∈ Λ then prv[T[k]] = -∞
    i = i + shift
```

Under various reasonable assumptions [FM06] the average running time of the algorithm is estimated as $O(n \cdot \log_\sigma(m)/m)$ too. In case of r patterns it becomes $O(n \cdot \log_\sigma(r \cdot m)/m)$.

6.2.4. Experimental Results

For experiments, we created a long tokenized string of concatenated Java source files, taken from various open source projects, including jPOS [jPOS07], smppapi [smppapi07], and TM4J [TM4J]. This string was used as a text to be searched. Then it was encoded using JavaCC-based parser into a sequence of 1259799 tokens, including 51 reserved Java words and 10213 unique identifiers. A set of 100 patterns for each length of 1…32 symbols was randomly extracted from the input text.

Fig. 6.3 shows the average number of tokens searched per second for each algorithm. Besides the above described algorithms, we have evaluated modified versions of Shift-Or and Backward Trie Matching methods. Shift-Or procedure was optimized as proposed in [FG05]. The underlying data structure of Backward Trie Matching algorithm was substituted with a suffix array. In Fig. 6.3, PSO denotes the basic parameterized Shift-Or algorithm, PFSO is the optimized Shift-Or version, PBTM is the parameterized Backward Trie Matching algorithm, and PBAM is the suffix array-based version of PBTM.

We also experimented with the multi-pattern version of PBAM (searching 100 patterns simultaneously). The plot clearly indicates that the amortized (average) speed per pattern is better than for any of the single pattern matching algorithms.

As a result, we conclude that the proposed methods of Shift-Or and BDM generalization are easy to implement and work well in practice [FM06].

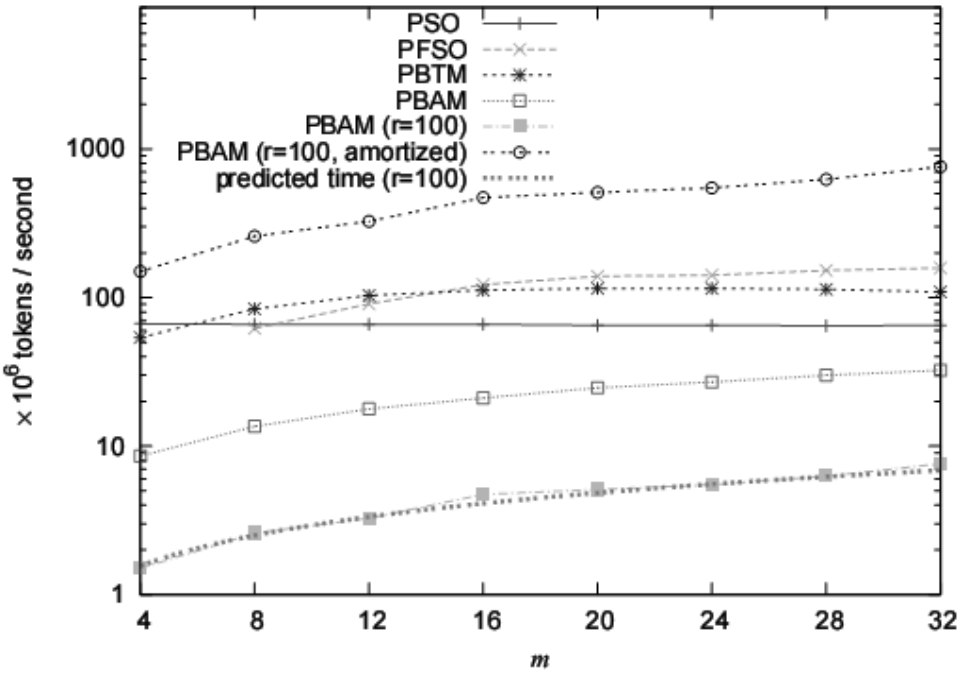


Fig. 6.3. The search speed of different p-matching algorithms

6.3. Reducing Asymptotic Complexity

As shown in Section 6.1, a high-quality content comparison-based method requires $O(N^2 f(n))$ time to grade the whole input collection. Here N is the total number of files in the collection, n is an average file size, and $f(n)$ is time required to grade one file pair. Typically, the asymptotic complexity of $f(n)$ is at least linear.

In this section we will consider Fast Plagiarism Detection System (FPDS), proposed in [MFWJS05]. It is a content comparison program that utilizes additional data structures and special heuristics in order to achieve better asymptotic complexity. Our system works with tokenized Java files, but any similarly preprocessed data can be analyzed as well. Sublinear complexity makes our system scalable, unlike most existing solutions. Another scalable approach is described in [BTZ07].

6.3.1. FPDS Algorithms

FPDS is based on a special suffix array-like data structure. A suffix array is a lexicographically sorted array of all suffixes of a given string [MM90]. Our index structure is a bit more complex: it stores all input collection files (as strings of tokens) in the suffix array, while each array entry also contains the file identifier of the corresponding suffix. The use of such index structure requires additional O(nN) space, where n is an average file size, and N is a number of files in the collection. The construction of the array takes O(nN·log(nN)) time using a specialized algorithm [MM90]. It can be improved to O(nN), see [KSB03]. Our index structure allows it to rapidly find a file (or files) containing any given substring. This is achieved with a binary search, and requires O(m + log(nN)) time on average, where m is the length of the substring.

A query file Q is compared against an existing collection using Algorithm 6.3. It tries to find the substrings of Q in the suffix array (see Fig. 6.4). Matching substrings are recorded, and each match contributes to the similarity score.

Algorithm 6.3. Scoring Single File

```
p = 1 // the first token of Q
WHILE p <= |Q| - γ + 1
    IF Q[p...p + γ – q] is found in the suffix array
        UpdateRepository
        p = p + γ
    ELSE
        p = p + 1
    END IF
END WHILE
FOR EACH file Fi in the collection
    Similarity(Q, Fi) = MatchedTokens(Fi) / q
```

Matching every token of the input file leads to a plenty of false matches, so we use a parameter γ, standing for "shortest string length to match" in tokens. The matches are stored in the repository, see Algorithm 6.4. The similarity between the file being tested and any file in the collection is just a number of tokens matched in the collection file divided by the total number of tokens in the test file, so it is a value between 0 and 1.

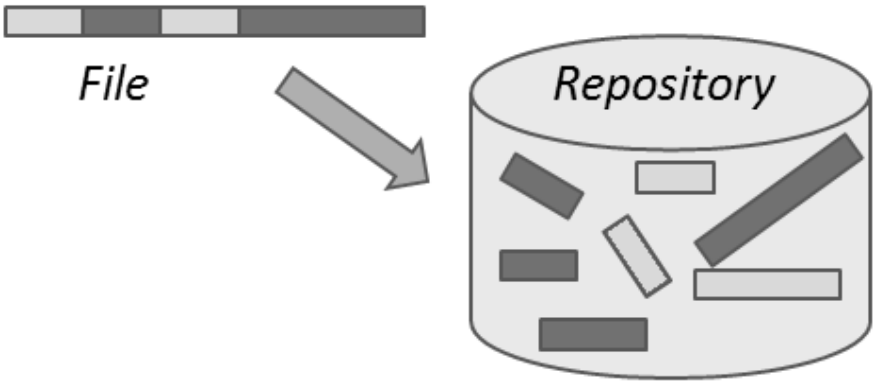


Fig. 6.4. Matching a file against repository

The procedure of repository update resolves two types of collisions when two or more matches are found, but only one of them should be stored in order to preserve consistent scoring scheme. The first collision type appears when more than one match is found in the same collection file, since every match should be taken into account only once. The second collision type is just a reverse problem: the situation when two different chunks of the query file correspond to the same chunk in a specific collection file. We use "longest match wins" heuristics to resolve collisions.

Algorithm 6.4. Updating the Repository

```
let S be the set of matches of Q[p...p + γ - 1]
IF some of the strings in S are found in the same file
    leave only the longest one
FOR EACH string M of the remaining list S
    IF M does not intersect with any repository element
        insert M into the repository
    ELSE IF M is longer than any conflicting repository elements
        remove all conflicting repository elements
        insert M into the repository
    END IF
END FOR
```

6.3.2. FPDS Performance

Line 3 of Algorithm 6.3 takes $O(\gamma + \log nN)$ average time, where as before, n is an average file size, γ is a fine-tunable constant, and N is the size of the collection. If the substring was found, we call Algorithm 6.4. This can happen at most $O(n/\gamma)$ times, so line 5 of Algorithm 6.3 takes at most $O(n/\gamma)\cdot$Complexity(Algorithm 6.4). If we make the

simplifying assumption that two randomly picked tokens match each other with fixed probability p, then on average we obtain np^γ matches for substrings of length γ. The total complexity of Algorithm 6.4 is on average $O((n/\gamma \cdot nNp^\gamma)^2)$. To keep the total average complexity of Algorithm 6.3 to $O(n(\gamma + \log nN)$, it is enough that $\gamma = \Omega(\log nN)$. Since the score of each file can be computed in O(N) time, the total average complexity of Algorithm 6.4 can be made $O(n(\gamma + \log nN) + N)$. Using suffix trees instead of suffix arrays, the $O(\gamma + \log nN)$ factors can be reduced to O(1), so the resulting complexity is $O(n + N)$.

Furthermore, the scoring has to be performed N times to obtain scores for all collection files, so the time becomes $O(nN + N^2)$ with suffix tree, and $O(nN \cdot \log(nN) + N^2)$ with suffix array.

These estimations were experimentally tested using a collection of Java files of a medium size of 14KB, gathered from actual students' laboratory works. The time required to grade one file is shown in Fig 6.5; the all-against-all scoring time is represented in Fig. 6.6. Larger γ values make the algorithm run faster with a constant factor.

It is quite easy to estimate asymptotic complexity of the system and to test it experimentally, but the more significant feature of any detector is its reliability, which is much harder to evaluate.

It is not feasible to perform thorough human evaluation of the output due to the large size of input collection. However, we can examine the reports that are produced by different plagiarism detection systems on the same dataset. While these reports can vary highly, the simple sets of files that contain plagiarisms usually correlate quite well, according to the output of each particular system [MFWJS05].

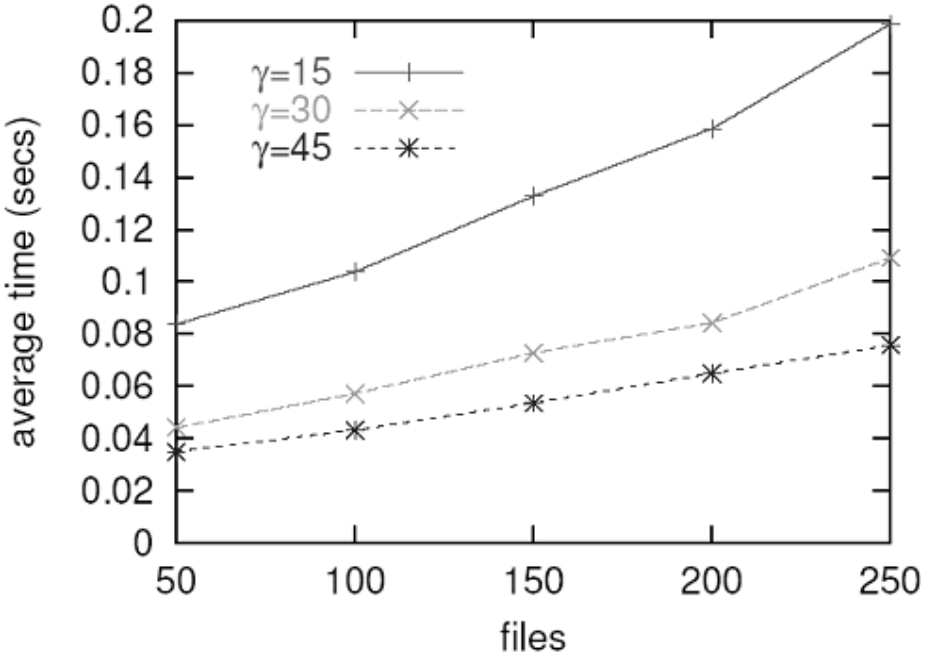


Fig. 6.5. Time required to grade a single file

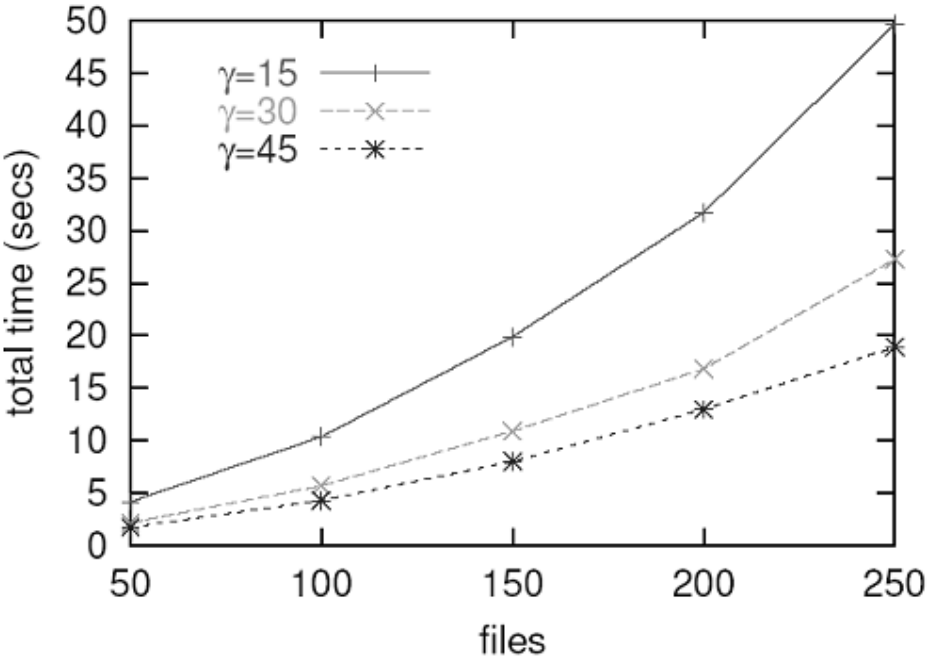


Fig. 6.6. Time required to grade a collection

To evaluate FPDS, we have used a jury approach. The idea is to get lists of suspicious files in the collection from various systems, and then to organize a simple voting. If a particular document is considered as suspicious, i.e. having enough instances of plagiarism by the majority of systems, we expect our program to mark it as suspicious as well. The systems can be fine-tuned to show more or less files, so we tried to obtain equal-size outputs.

For FPDS evaluation we used MOSS [SWA03], JPlag [PMP02], and Sherlock [JL99] systems as the jury. The reports of different systems are shown in Fig. 6.7. The diagram represents the score of every file of an example 50-files collection, proposed by each system, including FPDS.

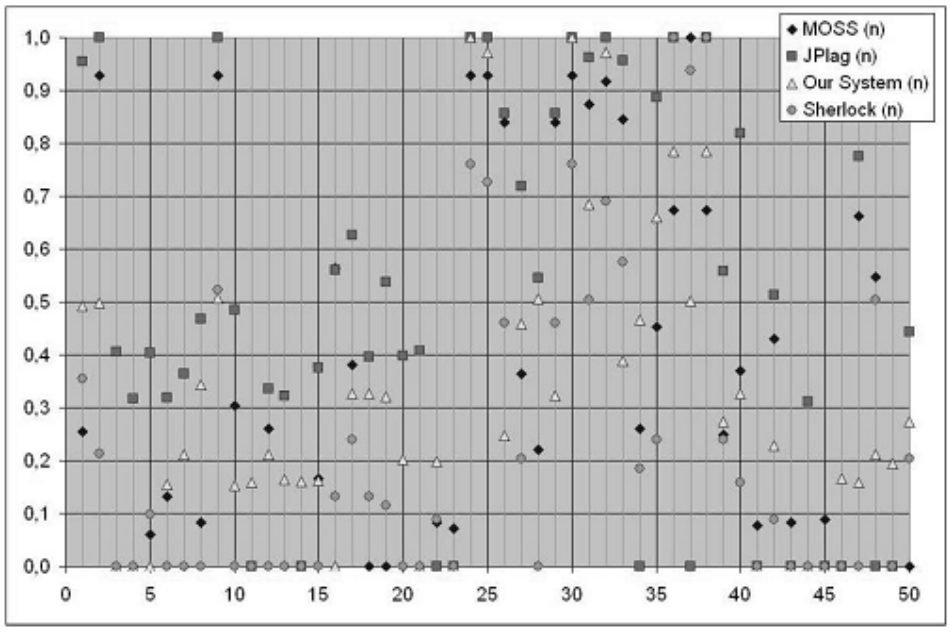


Fig. 6.7. Reports of the jury

At a glance, the votes seem to vary greatly from one system to another. However, the situation becomes clearer after conformist test. To perform it, we have compared the opinion of each particular system with the joint voting of three remaining programs (the jury). The file is considered to be marked by the jury if at least two of three jury systems considered it as suspicious. The results of the test for the collection of 155 Java files are summarized in Fig. 6.8.

The table clearly shows that our system (FPDS) is even more conformist than other systems, agreeing with the jury in 115 cases of 155.

	MOSS	JPlag	Our System	Sherlock
Agreed	111	109	115	99
Disagreed	44	46	40	56

Fig. 6.8. The results of conformist test

6.4. Combining Speed and Accuracy

While FPDS proved to be a fast and reliable plagiarism detector, it still suffers from the problems of another kind. It is not enough for a modern plagiarism detection system to print a list of similar file pairs. The system should provide a convenient user interface for analyzing matched pairs. Following Sherlock, FPDS implements a handy similarities graph visualization window (see Fig. 6.9).

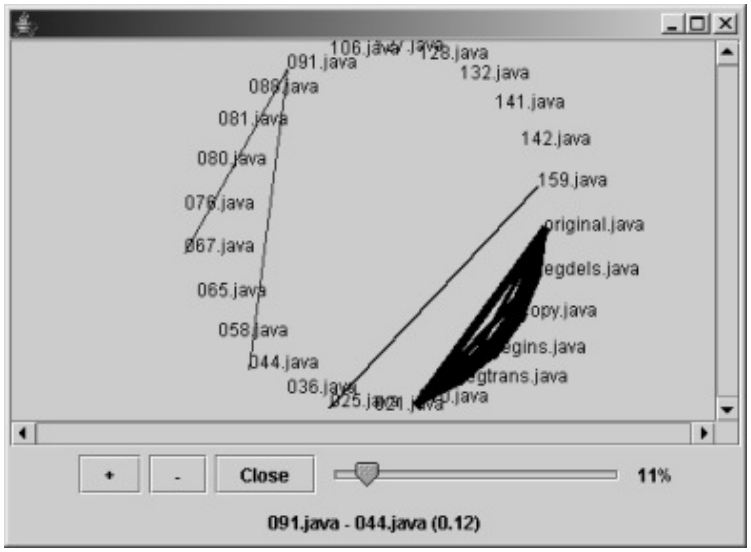


Fig. 6.9. Similarities graph window

This tool allows the user to see matched file pairs at any given similarity threshold. By clicking on some graph edge, the user opens a window with two text boxes filled with the contents of the corresponding files. For the further convenience of the manual evaluation, a good system should highlight found matches.

Here comes the problem of FPDS. Its algorithm is inherently not designed to build continuous coverage of file pairs (in contrast to, e.g., RKR-GST). The equal textual chunks, detected with FPDS, can be disorderedly scattered through the files. While these matches still serve as reliable indication of plagiarism, they are not suitable for visualization. Furthermore, the heuristics used in FPDS can lead to occasional ignoring of the matches, which makes visualization even worse. It also can be argued that direct file-file comparison can give more precise similarity score instead of a rough estimation made by FPDS.

One of possible solutions is to use a combined approach. In the first step FPDS serves as a filter that quickly determines a set of suspicious files. In the second step a precise file-file comparison routine is invoked.

This idea is studied in our work [MKK07]. As a file-file comparison routine we have used RKR-GST-based Plaggie system [ASR06]. On large collections, Plaggie works considerably slower (see Fig. 6.10).

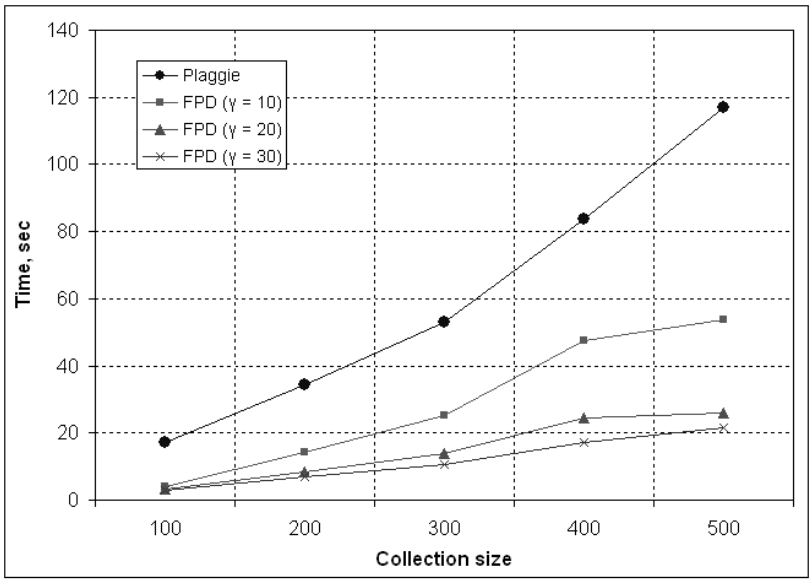


Fig. 6.10. Running time of FPDS and Plaggie

It should be noted that such combined method makes sense only if:

- the resulting system has to be significantly faster than Plaggie — otherwise we can just use Plaggie;
- FPDS and Plaggie should agree in most cases; in other words, generally FPDS should not exclude files, which are considered as plagiarized by Plaggie, from the input set.

Collection 1 (100 files)					
	Files marked				
Sim	Plaggie	FPDS20	FPDS30	Plaggie ∩ FPDS20	Plaggie ∩ FPDS30
ST1	23	32	19	18 (78.3%)	16 (69.6%)
ST2	14	21	11	13 (92.9%)	10 (71.4%)
Collection 2 (200 files)					
	Files marked				
Sim	Plaggie	FPDS20	FPDS30	Plaggie ∩ FPDS20	Plaggie ∩ FPDS30
ST1	72	87	66	62 (86.1%)	56 (77.8%)
ST2	51	62	46	46 (90.2%)	41 (80.4%)
Collection 3 (300 files)					
	Files marked				
Sim	Plaggie	FPDS20	FPDS30	Plaggie ∩ FPDS20	Plaggie ∩ FPDS30
ST1	92	119	89	78 (84.8%)	69 (75.0%)
ST2	63	82	60	56 (88.9%)	50 (79.4%)
Collection 4 (400 files)					
	Files marked				
Sim	Plaggie	FPDS20	FPDS30	Plaggie ∩ FPDS20	Plaggie ∩ FPDS30
ST1	118	149	115	104 (88.1%)	93 (78.8%)
ST2	80	104	78	71 (88.8%)	64 (80.0%)
Collection 5 (500 files)					
	Files marked				
Sim	Plaggie	FPDS20	FPDS30	Plaggie ∩ FPDS20	Plaggie ∩ FPDS30
ST1	138	184	141	118 (85.5%)	108 (78.3%)
ST2	92	124	91	80 (87.0%)	71 (77.2%)

Fig. 6.11. Agreement in Plaggie and FPDS results

We have tested these assumptions using five sample collections of 100, 200, 300, 400, and 500 distinct Java files taken from Apache Tomcat project source code. We do not expect plagiarism in Tomcat, but some code duplication in a large software project is likely to occur. The median file size is 6.15KB; the size of the largest (500-file)

collection is 4.58MB. The table in Fig. 6.11 illustrates an agreement between Plaggie and FPDS, for $\gamma = 20$, and $\gamma = 30$.

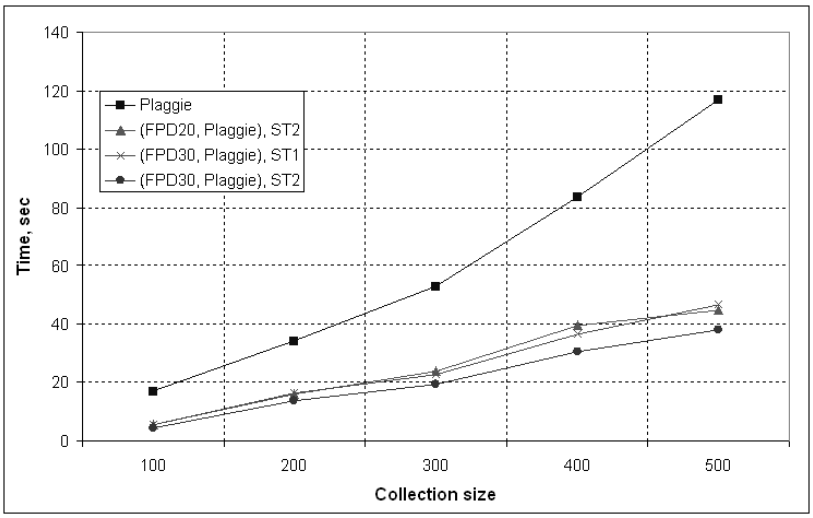


Fig. 6.12. Running time of Plaggie and Combined system

This conclusion is supported by statistical analysis. In order to prove an association between the two variables, we have applied the nonparametric test of Spearman's rank correlation [Lehmann98] to the data presented in Fig. 6.11. According to the procedure, the null hypothesis is: there is no correlation between two variables; the rank correlation coefficient for the entire population is equal to zero. The alternative hypothesis claims that there is a correlation between two variables and the aforementioned rank correlation coefficient is not equal to zero. The obtained results show that there is a significant correlation between the data we obtained using Plaggie and FPDS ($n = 10$, $\alpha = 0.01$, $rs = 0.997$, critical value $z = 0.794$). Here rs is the sample statistic; n is the number of pairs of sample data (see columns 2 and 3 of the input table); α is the significance level, and z is a critical interval. Because the sample statistic exceeds the critical value, we made a conclusion about significant correlation. The same strong correlation is found between the data generated by Plaggie and common fractions of files marked by both systems (columns 2 and 5 of the input table). From this outcome, we can expect the same behavior of two systems when they analyze Java source code, and we conclude that FPDS is quite an accurate filter for Plaggie.

Since the probability distribution of plagiarized files in the collection is hard to estimate, the speed of the combined system was tested experimentally (see Fig. 6.12). The graphs clearly show that the combined system runs significantly faster.

As the main result, we can expect that the proposed system will filter files very rapidly and quite accurately with the help of FPDS-based algorithm. The number of files sent for a detailed investigation to the Plaggie part is small enough. Then the RKR-GST-based Plaggie can provide a detailed similarity report, which is more reliable and much better suitable for further human evaluation.

7. Conclusions

The algorithms of plagiarism detection are constantly developed. We have seen how similarity checking evolved from simple attribute counting systems to the complex procedures of tree matching over tokenized input files. Several novel techniques for plagiarism detection are demonstrated in this work.

This thesis shows how modern plagiarism detection systems can fight with rewording, paraphrasing, variable renaming and other similar methods of plagiarism hiding. It explains how a plagiarism detection system can be fast and reliable at the same time, how it can score files and present the final results to the user. The work also reveals the frontiers of current plagiarism detection techniques, suggesting possible directions for future research.

The proposed research questions are answered as follows:

1. The classification scheme, developed in [Mozgovoy06], shows the state of the art in hermetic plagiarism detection algorithms. It is clear, which types of algorithms are used today, what are their advantages, drawbacks, time requirements, and quality. The scheme also helps the developers to position their systems better.
2. The paper [MFWJS05] shows how to build a hermetic plagiarism detection system that can have speed characteristics of fingerprint-based algorithms and reliability of content comparison. This is done by using suffix tree-based data structures with heuristic string matching. The work [MKS07] proposes a tandem of the algorithm, explained in [MFWJS05], with a conventional content comparison system. This approach allows getting more precise similarity scores and is better suitable for the human analysis of the results.
3. It is hard to overestimate the importance of text preprocessing in plagiarism detection. Simple textual transformations can reveal carefully camouflaged plagiarism. The papers [MTK06] and [MKK07] demonstrate how text processing can be used for plagiarism detection in natural language essays. The paper [FM06]

proposes several algorithms of efficient parameterized matching — a preprocessing technique, helpful for plagiarism detection in software source code.

8. Perspectives

The last 25 years of plagiarism detection systems development have achieved impressive results. Started from simple attribute counting systems that are more like proofs-of-concept than tools that can be recommended for everyday use, software plagiarism detectors evolved into complex content comparison analyzers and successful commercial web services.

However, there are still unsolved problems and possible applications that will be subjects of future research:

- **Broader use**. Taking into account the growing popularity of online submission systems, we can expect the broader use of automatic plagiarism detection. An essay, submitted to a course instructor or a paper, uploaded to a conference website, can be automatically checked for instances of plagiarism, just like e-mail messages are commonly checked for viruses by the mail server.
- **Social impact**. Being just file comparison services, Internet searching engines developed into a social phenomenon. The results they provide can have a noticeable impact on business [Lenssen06] or interfere with local laws [McLaughlin06]. Likewise, computer-calculated plagiarism detection scores should be used with care. Currently, the number of social collisions is relatively small in this field [Glod06, MM07] but we can expect their growth.
- **Better detection quality**. Current systems are still unable distinguish plagiarism from proper quotations. They are also unable to detect borrowings, translated from the other languages. Finally, they cannot decide which of the similar documents was the original source. Rewording and paraphrasing are efficient plagiarism hiding measures, since natural language parsers are still rarely used.

9. References

[AC75] A. Aho, M. Corasick. Efficient String Matching: an Aid to Bibliographic Search. *Communications of the ACM*, vol. 18(6), 1975, p. 333-340.

[AFM94] A. Amir, M. Farach, S. Muthukrishnan. Alphabet Dependence in Parameterized Matching. *Information Processing Letters*, vol. 49(3), 1994, p. 111-115.

[AKVMVU04] S. Alaoutinen, N. Kontro-Vesivalo, D. Medvedev, J. Voracek, A. Uteshev. Academic Honesty in Cross-Border Education — Opinions of Involved Students. *Proc. of FIE'04 Conference*, vol. 3, 2004, p. S3E/20-S3E/25.

[ASR06] A. Ahtiainen, S. Surakka, M. Rahikainen. Plaggie: GNU-Licensed Source Code Plagiarism Detection Engine for Java Exercises. *Proc. of the 6th Baltic Sea Conference on Computing Education Research*, 2006, p. 141-142.

[Baker95] B. Baker. Parameterized Pattern Matching by Boyer-Moore-type Algorithms. *Proc. of ACM-SODA*, 1995, p. 541-550.

[Baker97] B. Baker. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance. *SIAM Journal on Computing*, vol. 26(5), 1997, p. 1343-1362.

[BG01] B. Braumoeller, B. Gaines. Actions Do Speak Louder than Words: Deterring Plagiarism with the Use of Plagiarism-Detection Software. *PS: Political Science and Politics*, vol. 34(4), 2001, p. 835-839.

[Bliwise01] R. Bliwise. A Matter of Honor. *Duke Magazine*, vol. 87(4), 2001, p. 2-7.

[BNH04] B. Belkhouche, A. Nix, J. Hassell. Plagiarism Detection in Software Designs. *Proc. of the 42nd Annual Southeast Regional Conference*, 2004, p. 207-211.

[Boss07] BOSS website: www.dcs.warwick.ac.uk/boss (accessed: 21-Aug-2007).

[Brainard05] S. Brainard. Avoiding Plagiarism Guide. Milne Library, SUNY Geneseo, USA, 2005.

[BTZ07] S. Burrows, S. M. M. Tahaghoghi, J. Zobel. Efficient Plagiarism Detection for Large Code Repositories. *Software — Practice & Experience*, vol. 37(2), 2007, p. 151-175.

[BYG92] R. Baeza-Yates, G. Gonnet. A New Approach to Text Searching. *Communications of the ACM*, vol. 35(10), 1992, p. 74-82.

[Carbone01] N. Carbone. Turnitin.com, a Pedagogic Placebo for Plagiarism. *Bedford/St. Martin's TechNotes*, 2001.

[CCGJLPR94] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, W. Rytter. Speeding Up Two String Matching Algorithms. *Algorithmica*, vol. 12(4), 1994, p. 247-267.

[CCWS99] J. Cochran, M. Chamlin, P. Wood, and C. Sellers. Shame, Embarrassment and Formal Sanction Threats: Extending the Deterrence/Rational Choice Model to Academic Dishonesty. *Sociological Inquiry*, vol. 69(1), 1999, p. 91-105.

[CJ05] J. Colwell, F. Jenks. Student Ethics in Online Courses. *Proc. of FIE'05 Conference*, 2005, p. T2D/17-T2D/19.

[EK03] Ph. Edmonds, A. Kilgarriff (Eds.) Journal of Natural Language Engineering (Special Issue Based On Senseval-2), vol. 9(1), 2003.

[Enseling00] O. Enseling. Build Your Own Languages with JavaCC.

JavaWorld, Dec. 2000.

[Eve07] EVE2 website: www.canexus.com (accessed: 21-Aug-2007).

[Fellbaum98] C. Fellbaum. WordNet: An Electronic Lexical Database. The MIT Press, 1998, 423 p.

[FG05] K. Fredriksson, Sz. Grabowski. Practical and Optimal String Matching. *Lecture Notes in Computer Science*, vol. 3772, 2005, p. 374-385.

[FM06] K. Fredriksson, M. Mozgovoy. Efficient Parameterized String Matching. *Information Processing Letters*, vol. 100(3), 2006, p. 91-96.

[FR87] J. Faidhi, S. Robinson. An Empirical Approach for Detecting Program Similarity within a University Programming Environment. *Computers & Education*, vol. 11(1), 1987, p. 11-19.

[Gerald06] S. Gerald. Confessions of a User. *Composition Southeast Weblog*, 2006.

[Glod06] M. Glod. Students Rebel Against Database Designed to Thwart Plagiarists. *Washington Post*, September 22, 2006.

[Grier81] S. Grier. A Tool that Detects Plagiarism in Pascal Programs. *ACM SIGCSE Bulletin*, vol. 13(1), 1981, p. 15-20.

[GT99] D. Gitchell, N. Tran. Sim: a Utility for Detecting Similarity in Computer Programs. *Proc. of the 30th SIGCSE Technical Symposium on Computer Science Education*, 1999, p. 266-270.

[Halstead77] M. Halstead. Elements of Software Science. Elsevier, 1977, 142 p.

[Harris94] J. Harris. Plagiarism in Computer Science Courses. *Ethics in the Computer Age*, 1994, p. 122-134.

[HCMS02] T. Harding, D. Carpenter, S. Montgomery, N. Steneck. A

Comparison of the Role of Academic Dishonesty Policies of Several colleges on the Cheating Behavior of Engineering and Pre-Engineering Students. *Proc. of FIE'02 Conference*, vol. 3, 2002, p. S1H/15-S1H/20.

[HG82] C. Hwang, D. Gibson. Using an Effective Grading Method for Preventing Plagiarism on Programming Assignments. *Proc. of the 13th SIGCSE Technical Symposium*, 1982, p. 50-59.

[IHBW03] L. Introna, N. Hayes, L. Blair, and E. Wood. Cultural Attitudes Towards Plagiarism: Developing a Better Understanding of the Needs of Students from Diverse Cultural Backgrounds Relating to Issues of Plagiarism. Lancaster University, 2003.

[Irving04] R. Irving. Plagiarism and Collusion Detection Using the Smith-Waterman Algorithm. *TR-2004-164*, University of Glasgow, 2004.

[IS96] R. Indury, A. Schäffer. Multiple Matching of Parameterized Patterns. *Theoretical Computer Science*, vol. 154(2), 1996, p. 203-224.

[JK03] R. Jackson, K. Kern. Deterring and Detecting Plagiarism. *Instruction Commons Guides*, Iowa State University, 2003.

[JL99] M. Joy, M. Luck. Plagiarism in Programming Assignments. *IEEE Transactions on Education*, vol. 42(2), 1999, p. 129-133.

[Jones01] E. Jones. Metrics Based Plagiarism Monitoring. *The Journal of Computing in Small Colleges*, vol. 16(4), 2001, p. 253-261.

[jPOS07] jPOS website: www.jpos.org (accessed: 11-Sep-2007).

[KM03] D. Klein, C. Manning. Accurate Unlexicalized Parsing. *Proc. of the 41st Meeting of the Association for Computational Linguistics*, 2003, p. 423-430.

[KR87] R. Karp, R. Rabin. Efficient Randomized Pattern-Matching Algorithms, *IBM Journal of Research and Development*, vol. 31(2), 1987, p. 249-260.

[KSB03] J. Kärkkäinen, P. Sanders, S. Burkhardt. Linear Work Suffix Array Construction. *Journal of the ACM*, vol. 53(6), 2006, p. 918-936.

[Lehmann98] E. Lehmann. Nonparametrics: Statistical Methods Based on Ranks, Revised. Pearson Education, 1998, 480 p.

[Lenssen06] Ph. Lenssen. German BMW Banned From Google. *Google Blogoscoped*, blogoscoped.com/archive/2006-02-04-n60.html (accessed: 12-Sep-2007).

[Levenshtein66] V. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, vol. 10, 1966, p. 707.

[Manber94] U. Manber. Finding Similar Files in a Large File System. *Proc. of the USENIX Technical Conference*, 1994, p. 1-10.

[Meizlish05] D. Meizlish. Promoting Academic Integrity in the Classroom. *CRLT Occasional Papers*, University of Michigan, #20, 2005.

[McLaughlin06] A. McLaughlin. Google in China. *The Official Google Blog*, googleblog.blogspot.com/2006/01/google-in-china.html (accessed: 12-Sep-2007).

[MFWJS05] M. Mozgovoy, K. Fredriksson, D. White, M. Joy, and E. Sutinen. Fast Plagiarism Detection System. *Lecture Notes in Computer Science*, vol. 3772, 2005, p. 267-270.

[MKK07] M. Mozgovoy, S. Karakovskiy, V. Klyuev. Fast and Reliable Plagiarism Detection System. *Proc. of FIE'07 Conference*, 2007.

[MKS07] M. Mozgovoy, T. Kakkonen, E. Sutinen. Using Natural Language Parsers in Plagiarism Detection. *Proc. of SLaTE'07 Workshop*,

2007.

[MM07] A. Morrow, J. McLean. The War on Plagiarism. *Eyeopener*, Ryerson University, 17-Oct-2007.

[MM90] U. Manber, G. Myers. Suffix Arrays: a New Method for On-Line String Searches. *Proc. of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, 1990, p. 319-327.

[MMCM06] M. de Marneffe, B. MacCartney, C. Manning. Generating Typed Dependency Parses from Phrase Structure Parses. *Proc. of 5th International Conference on Language Resources and Evaluation*, 2006.

[Moodle07] Moodle website: moodle.org (accessed: 21-Aug-2007).

[Mozgovoy06] M. Mozgovoy. Desktop Tools for Offline Plagiarism Detection in Computer Programs. *Informatics in Education*, vol. 5(1), 2006, p. 97-112.

[MTK06] M. Mozgovoy, V. Tusov, V. Klyuev. The Use of Machine Semantic Analysis in Plagiarism Detection. *Proc. of the 9th International Conference on Humans and Computers*, Japan, 2006, p. 72-77.

[Mydropbox07] MyDropBox website: www.mydropbox.com (accessed: 21-Aug-2007).

[Nakov00] P. Nakov. Latent Semantic Analysis of Textual Data. *Proc. of the Conference on Computer Systems and Technologies*, Bulgaria, 2000, p. 5031-5035.

[Ottenstein77] K. Ottenstein. An Algorithmic Approach to the Detection and Prevention of Plagiarism. *SIGCSE Bulletin*, vol. 8(4), 1977, p. 30-41.

[PMP02] L. Prechelt, G. Malpohl, M. Philippsen. Finding Plagiarisms among a Set of Programs with JPlag. *Journal of Universal Computer Science*, vol. 8(11), 2002, p. 1016-1038.

[SE06] B. Stein, S. M. zu Eissen. Near Similarity Search and Plagiarism Analysis. *Selected Papers from the 29th Annual Conference of the German Classification Society*, 2006, p. 430-437.

[ST06] L. Salmela, J. Tarhio. Sublinear Algorithms for Parameterized Matching. *Lecture Notes in Computer Science*, vol. 4009, 2006, p. 354-364.

[SWA03] S. Schleimer, D. Wilkerson, A. Aiken. Winnowing: Local Algorithms for Document Fingerprinting. *Proc. of the ACM SIGMOD'03 International Conference on Management of Data*, 2003, p. 76-85.

[smppapi07] smppapi website: www.smppapi.sourceforge.net (accessed: 11-Sep-2007).

[TM4J07] TM4J website: www.tm4j.org (accessed: 11-Sep-2007).

[Turabian96] K. Turabian. A Manual for Writers of Term Papers, Theses, and Dissertations (6th Ed). The University of Chicago Press, 1996, 318 p.

[Tusov04] V. Tusov. Computer Semantics of the Russian Language (in Russian). St. Petersburg University Press, 2004, 400 p.

[Turnitin07] Turnitin website: www.turnitin.com (accessed: 21-Aug-2007).

[TW04] L. Trivedi, S. Williams. Using Sources. Hamilton College Writing Center, 2004.

[VD05] P. Vamplew, J. Dermoudy. An Anti-Plagiarism Editor for Software Development Courses. *Proc. of the 7th Australasian*

Conference on Computing Education, vol. 42, 2005, p. 83-90.

[VW97] K. Verco, M. Wise. Plagiarism à la Mode: a Comparison of Automated Systems for Detecting Suspected Plagiarism. *The Computer Journal*, vol. 39(9), 1997, p. 741-750.

[Webct07] WebCT website: www.webct.com (accessed: 21-Aug-2007).

[Webster01] Webster's Encyclopedic Unabridged Dictionary. Thunder Bay Press, USA, 2001, 2230 p.

[Wiedemeier02] P. Wiedemeier. Preventing Plagiarism in Computer Literacy Courses. *Journal of Computing in Small Colleges*, vol. 17(4), p. 154-163.

[Wise92] M. Wise. Detection of Similarities in Student Programs: YAP'ing may be Preferable to Plague'ing. *ACM SIGCSE Bulletin*, vol. 24(1), 1992, p. 268-271.

[Wise94] M. Wise. Running Rabin-Karp Matching and Greedy String Tiling. *Basser Department of Computer Science Technical Report*, Sydney University, 1994.

[Wise96] M. Wise. YAP3: Improved Detection of Similarities in Computer Programs and Other Texts. *Proc. of SIGCSE'96 Technical Symposium*, 1996, p. 130-134.

[Zobel04] J. Zobel. "Uni Cheats Racket": A case study in plagiarism investigation. *Proc. of the Australasian Computer Education Conference*, 2004, p. 357-365.

Publication [MFWJS05]

M. Mozgovoy, K. Fredriksson, D. White, M. Joy, and E. Sutinen. Fast Plagiarism Detection System. *Lecture Notes in Computer Science*, vol. 3772, 2005, p. 267-270

Fast Plagiarism Detection System

Maxim Mozgovoy[1], Kimmo Fredriksson[1]*, Daniel White[2], Mike Joy[2], and Erkki Sutinen[1]

[1] Department of Computer Science, University of Joensuu, PO Box 111, FIN–80101 Joensuu, Finland
{Maxim.Mozgovoy, Kimmo.Fredriksson, Erkki.Sutinen}@cs.joensuu.fi
[2] Department of Computer Science, University of Warwick, Coventry CV4 7AL, U.K.
{D.R.White, M.S.Joy}@warwick.ac.uk

Introduction. The large class sizes typical for an undergraduate programming course mean that it is nearly impossible for a human marker to accurately detect plagiarism, particularly if some attempt has been made to hide the copying. While it would be desirable to be able to detect all possible code transformations we believe that there is a minimum level of acceptable performance for the application of detecting student plagiarism. It would be useful if the detector operated at a level that meant for a piece of work to *fool* the algorithm would require that the student spent a large amount of time on the assignment and had a good enough understanding to do the work without plagiarising.

Previous Work. Modern plagiarism detectors, such as Sherlock [3], JPlag [5] and MOSS [6] use a tokenization technique to improve detection. These detectors work by pre-processing code to remove white-space and comments before converting the file into a tokenized string. The main advantage of such an approach is that it negates all lexical changes and a good token set can also reduce the efficacy of many structural changes. For example, a typical tokenization scheme might involve replacing all identifiers with the `<IDT>` token, all numbers by `<VALUE>` and any loops by generic `<BEGIN_LOOP>...<END_LOOP>` tokens. Our algorithm also makes use of tokenised versions of the input files and we use suffix arrays [4] as our index data structure to enable efficient comparisons.

While all the above-mentioned systems use different algorithms to each other, the core idea is the same: a many-to-many comparison of all files submitted for an assignment should produce a list sorted by some similarity score that can then be used to determine which pairs are most likely to contain plagiarism. A naïve implementation of this comparison, such as that used by Sherlock or JPlag, results in $O(f(n)N^2)$ complexity where N is the size (number of files) of the collection, and $f(n)$ is the time to make the comparison between one pair of files of length n. Without loss of detection quality, our method achieves $O(N(n+N))$ average time by using indexing techniques based on suffix arrays. If the index structure becomes too large, it can be moved from primary memory to secondary data storage without significant loss of efficiency [2].

The approach we describe can be also used to find similar code fragments in a large software system. In this case the importance of fast algorithm is especially

* Supported by the Academy of Finland, grant 202281.

Algorithm 1 Compare a File Against an Existing Collection

```
1   p = 1 // the first token of Q
2   WHILE p ≤ q − γ + 1
3     find Q[p...p + γ − 1] from the suffix array
4     IF Q[p...p + γ − 1] was found
5       UpdateRepository
6       p = p + γ
7     ELSE
8       p = p + 1
9   FOR EVERY file F_i in the collection
10    Similarity(Q, F_i) = MatchedTokens(F_i)/q
```

high due to large file collection size. The Dup tool [1] uses parametrized suffix trees to solve this task, but the algorithms are relatively complex compared to our approach.

Algorithms and Complexity. Our proposed system is based on an index structure built over the entire file collection. Before the index is built, all the files in the collection are tokenized. This is a simple parsing problem, and can be solved in linear time. For each of the N files in the collection, The output of the tokenizer for a file F_i is a string of n_i tokens. The total number of tokens is denoted by $n = \sum n_i$.

We use suffix array as an index structure. A suffix array is a lexicographically sorted array of all suffixes of a given string [4]. The suffix array for the whole document collection is of size $O(n)$. We consider the total memory requirements to be acceptable for modern hardware. A suffix array allows us to rapidly find a file (or files), containing any given substring. This is achieved with a binary search, and requires $O(m + \log_2 n)$ time on average, where m is the length of the substring (it is also possible to make this the worst case complexity, see [4]). The array can be constructed in time $O(n \log n)$, assuming atomic comparison of two tokens.

Algorithm 1 is intended for finding all files within the collection's index that are similar to a given query file. It tries to find the substrings of the tokenised query file, $Q[1..q]$, in the suffix array, where q is the number of tokens. Matching substrings are recorded and each match contributes to the similarity score. The algorithm takes contiguous non-overlapping token substrings of length γ from the query file and searches all the matching substrings from the index. These matches are recorded into a 'repository'. This phase also includes a sanity check as overlapping matches are not allowed.

The similarity between the file Q being tested and any file F_i in the collection is just a number of tokens matched in the collection file divided by the total number of tokens in the test file (so it is a value between 0 and 1), i.e.

$$Similarity(Q, F_i) = MatchedTokens(F_i)/q,$$

In Algorithm 2, we encounter two types of collisions. The first one appears when more than one match is found in the same file. If several matches that are found correspond to the same indexed file, these matches are extended to

Algorithm 2 Update the Repository

```
1   Let S be the set of matches of Q[p...p + γ − 1]
2   IF some of the strings in S are found in the same file /* collision of type 1 */
3     leave only the longest one
4   FOR every string M from the remaining list S
5     IF M doesn't intersect with any repository element
6       insert M to the repository
7     ELSE IF M is longer than any conflicting rep. element /* collision of type 2 */
8       remove all conflicting repository elements
9       insert M to the repository
```

Γ tokens, $\Gamma \geq \gamma$, such that only one of the original matches survives for each indexed file. Therefore, for each file in the index, the algorithm finds all matching substrings that are longer than other matching substrings and whose lengths are at least γ tokens. The second one is the reverse of the first problem: we should not allow the situation when two different places in the input file correspond to the same place in some collection file. To resolve the difficulty we use 'longest wins' heuristics. We sum the lengths of all the previous matches that intersect with the current one, and if the current match is longer, we use it to replace the intersecting previous matches.

The complexity of Algorithm 1 is highly dependent on the value of the γ parameter. Line 3 of Algorithm 1 takes $O(\gamma + \log n)$ average time, where n is the total number of tokens in the collection (assuming atomic token comparisons). If we make the simplifying assumption that two randomly picked tokens match each other (independently) with fixed probability p, then on average we obtain np^γ matches for substrings of length γ. If Q was found, we call Algorithm 2. Its total complexity is, on average, at most $O((q/\gamma \cdot np^\gamma)^2)$. To keep the total average complexity of Algorithm 1 to at most $O(q(\gamma + \log n))$, it is enough that $\gamma = \Omega(\log_{1/p} n)$. This results in $O(q \log n)$ total average time. Since we require that $\gamma = \Omega(\log n)$, and may adjust γ to tune the quality of the detection results, we state the time bound as $O(q\gamma)$. Finally, the scores for each file can be computed in $O(N)$ time. To summarize, the total average complexity of Algorithm 1 can be made $O(q(\gamma + \log n) + N) = O(q\gamma + N)$. The $O(\gamma + \log n)$ factors can be easily reduced to $O(1)$ (worst case) using suffix trees [7] with suffix links, instead of suffix arrays. This would result in $O(q + N)$ total time.

Note that we have excluded the tokenization of Q and that we have considered the number of tokens rather than the number of characters. However, the tokenization is a simple linear time process, and the number of tokens depends linearly on the file length.

To compare every file against each other, we can just run Algorithm 1 for every file in our collection. After that, every file pair gets two scores: one when file a is compared to file b and one when the reverse comparison happens, as the comparison is not symmetric. We can use the average of these scores as a final score for this pair.

Summing up the cost of this procedure for all the N files in the collection, we obtain a total complexity of $O(n\gamma + N^2)$, including the time to build the suffix array index structure. With suffix trees this can be made $O(n + N^2)$

Evaluation of the System. It is not feasible in the nearest future to compare our system's results with a human expert's opinion on real-world datasets as a human would not have the time to conduct a thorough comparison of every possible file pair. However, we can examine the reports that are produced by different plagiarism detection software when used on the same dataset. The systems used for the analysis include MOSS [6], JPlag [5] and Sherlock [3]. Every system printed a report about the same real collection, consisting of 220 undergraduate student's Java programs.

The simple approach (to consider only detection or rejection) allows us to organize a 'voting' experiment. Let S_i be the number of 'jury' systems (MOSS, JPlag and Sherlock), which marked file i as suspicious. If $S_i \geq 2$, we should expect our system to mark this file as well. If $S_i < 2$, the file should, in general, remain unmarked.

For the test set consisting of 155 files marked by at least one program, our system agreed with the 'jury' in 115 cases (and, correspondingly, disagreed in 40 cases). This result is more conformist than the results obtained when the same experiment was run on the other 3 tested systems. Each system was tested while the other three acted as jury.

Conclusions. We have developed a new fast algorithm for plagiarism detection. Our method is based on indexing the code database with a suffix array, which allows rapid retrieval of blocks of code that are similar to the query file. This idea makes rapid pairwise file comparison possible. Evaluation shows that this algorithm's quality is not worse than the quality of existing widely used methods, while its speed performance is much higher. For the all-against-all problem our method achieves $O(\gamma n)$ (with suffix arrays) or $O(n)$ (with suffix trees) average time for the comparison phase. Traditional methods, such as JPlag, need at least $O((n/N)^2 N^2) = O(n^2)$ average time for the same task. In addition, computing the similarity matrix takes $O(N^2)$ additional time, and this cannot be improved, as it is also the size of the output.

References

1. B. S. Baker. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance. *SIAM Journal on Computing*, 26(5):1343–1362, 1997.
2. D. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. *Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, 1996.
3. M. S. Joy and M. Luck. Plagiarism in programming assignments. *IEEE Transactions on Education*, 42(2):129–133, 1999.
4. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of SODA '90*, 319–327. SIAM, 1990.
5. L. Prechelt, G. Malpohl, and M. Phlippsen. JPlag: Finding plagiarisms among a set of programs. Technical report, Fakultat for Informatik, Universitat Karlsruhe, 2000. `http://page.mi.fu-berlin.de/~prechelt/Biblio/jplagTR.pdf`.
6. S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of SIGMOD '03*, 76–85. ACM Press, 2003.
7. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

Publication [Mozgovoy06]

M. Mozgovoy. Desktop Tools for Offline Plagiarism Detection in Computer Programs. *Informatics in Education*, vol. 5(1), 2006, p. 97-112

Informatics in Education, 2006, Vol. 5, No. 1, 97–112

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 ≠ 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 $O(N^2n^2)$.

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 $O(N^2)$ 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.

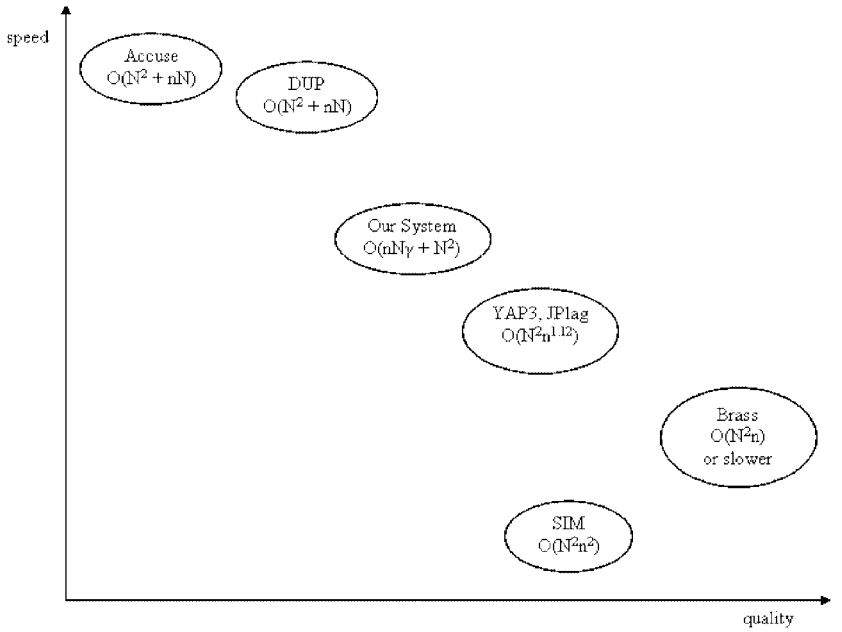


Fig. 1. Comparison of different plagiarism detection systems.

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.

Publication [FM06]

K. Fredriksson, M. Mozgovoy. Efficient Parameterized String Matching. *Information Processing Letters*, vol. 100(3), 2006, p. 91-96

Available online at www.sciencedirect.com
ScienceDirect
Information Processing Letters 100 (2006) 91–96
Information Processing Letters
www.elsevier.com/locate/ipl

Efficient parameterized string matching

Kimmo Fredriksson *,[1], Maxim Mozgovoy

Department of Computer Science, University of Joensuu, P.O. Box 111, 80101 Joensuu, Finland

Received 13 February 2006; received in revised form 1 June 2006; accepted 22 June 2006

Available online 2 August 2006

Communicated by S.E. Hambrusch

Abstract

In parameterized string matching the pattern P matches a substring t of the text T if there exist a bijective mapping from the symbols of P to the symbols of t. We give simple and practical algorithms for finding all such pattern occurrences in sublinear time on average. The algorithms work for a single and multiple patterns.

Keywords: Algorithms; Parameterized string matching; Bit-parallelism; Suffix automaton

1. Introduction

In traditional string matching problem one is interested in finding the occurrences of a pattern P from a text T, where P and T are strings over some alphabet Σ. Many variations of this basic problem setting exist, such as searching multiple patterns simultaneously, and/or allowing some limited number of errors in the matches, and indexed searching, where T can be preprocessed to allow efficient queries of P. See, e.g., [13,16,11] for an overview and references. Yet another variation is *parameterized matching* [6]. In this variant we have two disjoint alphabets, Σ for *fixed* symbols, and Λ for *parameter* symbols. In this setting we search *parameterized* occurrences of P, where the symbols from Σ must match exactly, while the symbols in Λ can be also renamed. This problem has important applications, e.g., in software maintenance and plagiarism detection [6], where the symbols of the strings can be, e.g., reserved words and identifier or parameter names of some (possibly tokenized) programming language source code. Hence one might be interested in finding code snippets that are the same up to some systematical variable renaming.

A myriad of algorithms have been developed for the classical problem, but only a few exist for parameterized matching. In [5] exact on-line matching algorithm for a single pattern was developed. This algorithm runs in $O(n \log \min(m, |\Lambda|))$ worst case time. However, the average case time was not analyzed. Another algorithm was given in [2], that achieves the same time bound both in average and worst cases. In the same paper it was shown that this is optimal, and that in particular the log factor cannot be avoided for general alphabets. However, for fixed alphabets we can avoid it, as shown in the present paper. In [14] it was shown that multiple patterns can be searched in $O(n \log(|\Sigma| + |\Lambda|) + occ)$ time, where occ is the number of occurrences of all the patterns. Other algorithms exist for the off-line problem

* Corresponding author.
E-mail address: kfredrik@cs.joensuu.fi (K. Fredriksson).
[1] Supported by the Academy of Finland, grant 202281.

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2006.06.009

[6,9]. In this paper we develop algorithms that under mild assumptions run in optimal time on average, are simple to implement and perform well in practice. Our algorithms are based on generalizing the well-known Shift-Or [4] and Backward DAWG (Directed Acyclic Word Graph) Matching algorithms [7,10]. Our algorithms generalize for the multipattern matching as well.

2. Preliminaries

We use the following notation. The *pattern* is $P[0\ldots m-1]$ and the *text* is $T[0\ldots n-1]$. The symbols of P and T are taken from two disjoint finite alphabets Σ of size σ and Λ of size λ. The pattern P matches the text substring $T[j\ldots j+m-1]$, iff for all $i \in \{0\ldots m-1\}$ it holds that $M_j(P[i]) = T[j+i]$, where $M_j(\cdot)$ is one-to-one mapping on $\Sigma \cup \Lambda$. Moreover, the mapping must be identity on Σ, but on Λ can be different for each text position j. For example, assume that $\Sigma = \{A, B\}$, $\Lambda = \{X, Y, Z\}$ and $P =$ AAZYZABXYZAX. Then P matches the text substring AAZYZABXYZAX with identity mapping, and AAXYXABZYXAZ with parameter mapping X $\mapsto$ Z, Y $\mapsto$ Y, and Z $\mapsto$ X. This mapping is simple with *prev* encoding [6]. For a string S, *prev*(S) maps all parameter symbols s in S to a non-negative integer p, where p is the number of symbols since the last occurrence of symbol s in S. The first occurrence of the parameter is encoded as 0. If s belongs to Σ, it is mapped to itself (s). For our example pattern, *prev*$(P) =$ AA002AB055A4. This is the same as the encoding for the two example substrings, i.e., *prev*(AAZYZABXYZAX) = *prev*(AAXYXABZYXAZ). Hence the problem is reduced to exact string matching, where we match *prev*(P) against *prev*$(T[j\ldots j+m-1])$ for all $j = 0\ldots n-m$. The string *prev*(S) can be easily computed in linear time for constant size alphabets. The only remaining problem then is how to maintain *prev*$(T[j\ldots j+m-1])$ (and any algorithmic parameters that depend on it) efficiently as j increases. The key is the following lemma [6].

Lemma 1. *Let $S' =$ prev(S). Then for $S'' =$ prev$(S[j\ldots j+m-1])$ for all i such that $S[i] \in \Lambda$ it holds that $S''[i] = S'[i]$ if $S'[i] < m$. Otherwise $S''[i] = 0$.*

We are now ready to present our algorithms. For simplicity we assume that Σ and Λ are finite constant size alphabets. For large alphabets all our time bounds hold if we multiply them by $O(\log(m))$.

3. Parameterized bit-parallel matching

In this section we present bit-parallel approach for parameterized matching, based in Shift-Or algorithm [4]. For the bit-parallel operations we adopt the following notation. A machine word has w bits, numbered from the least significant bit to the most significant bit. We use C-like notation for the bit-wise operations of words; & is bit-wise `and`, | is `or`, ^ is `xor`, $\sim$ negates all bits, $\ll$ is shift to left, and $\gg$ shift to right, both with zero padding. For brevity, we make the assumption that $m \leqslant w$, unless explicitly stated otherwise.

The standard Shift-Or automaton is constructed as follows. The automaton has states $0, 1, \ldots, m$. The state 0 is the initial state, state m is the final (accepting) state, and for $i = 0, \ldots, m-1$ there is a transition from the state i to the state $i+1$ for character $P[i]$. In addition, there is a transition for every $c \in \Sigma$ from the initial state to the initial state, which makes the automaton nondeterministic. The preprocessing algorithm builds a table B, having one bit-mask entry for each $c \in \Sigma$. For $0 \leqslant i \leqslant m-1$, the mask $B[c]$ has ith bit set to 0, iff $P[i] = c$. These correspond to the transitions of the implicit automaton. That is, if the bit i in $B[c]$ is 0, then there is a transition from the state i to the state $i+1$ with character c. The bit-vector D encodes the states of the automaton. The ith bit of the state vector is set to 0, iff the state i is active, i.e., the pattern prefix $P[0\ldots i]$ matches the current text position. Initially each bit is set to 1. For each text symbol c the vector is updated by $D \leftarrow (D \ll 1) \mid B[c]$. This simulates all the possible transitions of the nondeterministic automaton in a single step. If after the update the mth bit of d is zero, then there is an occurrence of P. If $m \leqslant w$, then the algorithm runs in time $O(n)$.

In order to generalize Shift-Or for parameterized matching, we must take care of three things:

(i) P must be encoded with *prev*;
(ii) *prev*$(T[j\ldots j+m-1])$ must be maintained in $O(1)$ time per text position;
(iii) the table B must be built so that all parameterized pattern prefixes can be searched in parallel.

The items (i) and (ii) are trivial, while (iii) is a bit more tricky. To compute *prev*(P) we just maintain an array *prv*$[c]$ that for each symbol $c \in \Lambda$ stores the position of its last occurrence. Then *prev*(P) can be computed in $O(m)$ time by a linear scan over P. To simplify indexing in the array B, we assume that $\Sigma = \{0\ldots\sigma-1\}$, and map the *prev* encoded parameter offsets into the

```
1   P' ← Encode(P, m)
2   for i ← 0 to σ + m − 1 do B[i] ← ∼0 ≫ (w − m)
3   for i ← 0 to λ − 1 do prv[σ + i] ← −∞
4   for i ← 0 to m − 1 do B[P'[i]] ← B[P'[i]]&∼(1 ≪ i)
5   for i ← 1 to m − 1 do B[σ + i] ← B[σ + i]&(B[σ] | (∼0 ≪ i))
6   D ← ∼0; mm ← 1 ≪ (m − 1)
7   for i ← 0 to n − 1 do
8       c ← T[i]
9       if c ∈ Λ then
10          c ← i − prv[T[i]] + σ
11          if c > σ + m − 1 then c ← σ
12          prv[T[i]] ← i
13      D ← (D ≪ 1) | B[c]
14      if (D&mm) ≠ mm then report match
```

Algorithm 1. P-Shift-Or(T, n, P, m).

range $\{\sigma \ldots \sigma + m - 1\}$. The text is encoded in the same way, but the encoding is embedded into the search code. The only difference is that we apply Lemma 1 to reset offsets that are greater than $m - 1$ (i.e., offsets that are for parameters that are outside of the current text window) to zero. Otherwise the search algorithm is exactly the same as for normal Shift-Or.

The tricky part is the preprocessing phase. We denote the *prev* encoded pattern as P'. At first P' is preprocessed just as P in the normal Shift-Or algorithm. This includes the parameter offsets, which are handled as any other symbol. However, this is not enough. We illustrate the problem by an example. Let $P = \text{XAXAX}$ and $T = \text{ZZAZAZAZ}$. In encoded forms these are $P' = 0\text{A}2\text{A}2$ and $T' = 01\text{A}2\text{A}2\text{A}2$. Clearly P has two (overlapping) parameterized matches in T. However, P' does not match in T' at all. The problem is that as the algorithm searches all the m prefixes of the pattern in parallel, then some non-zero encoded offset p (of some text symbol) should be interpreted as zero in some cases. These prefixes have lengths from 1 to m. To successfully apply Lemma 1 we should be able to apply it in parallel to all m substrings. In other words, any non-zero parameter offset p must be treated as zero for all pattern prefixes whose length h is less than p, since by Lemma 1 the parameter with offset p is dropped out of the window of length h. This problem can be solved as follows. The bit-vector $B[\sigma + i]$ is the match vector for offset i. If the j bit of this vector is zero, it means by definition that $P'[j] = i$. If any of the i least significant bits of $B[\sigma]$ are zero, we clear the corresponding bits of $B[\sigma + i]$ as well. More precisely, we set

$$B[\sigma + i] \leftarrow B[\sigma + i] \& \big(B[\sigma] \mid (\sim 0 \ll i)\big).$$

This means that the offset i is treated as offset i for prefixes whose length is greater than i, and as zero for the shorter prefixes, satisfying the condition of Lemma 1.

Algorithm 1 gives the complete code. The algorithm clearly runs in $O(n\lceil m/w \rceil)$ worst case time. For long patterns one can search just a length w prefix of the pattern, and verify with the whole pattern whenever the prefix matches, giving $O(n)$ average time. However, note that a long variable name (string) is just one symbol (token) in typical applications, hence w bits is usually plenty. Finally, note that for unbounded alphabets we cannot use arrays for *prv* and B. We can use balanced trees instead, but then the time bounds must be multiplied by $O(\log(m))$.

Standard Shift-Or can be improved to run in optimal $O(n \log_\sigma(m)/m)$ average time [12]. The algorithm takes a parameter q, and from the original pattern generates a set $\mathcal{P}$ of q new patterns $\mathcal{P} = \{P^0, \ldots, P^{q-1}\}$, each of length $m' = \lfloor m/q \rfloor$, where $P^j[i] = P[j + iq]$ for $i = 0 \ldots \lfloor m/q \rceil - 1$. In other words, the algorithm generates q different alignments of the original pattern P, each alignment containing only every qth character. The total length of the patterns in $\mathcal{P}$ is $q\lfloor m/q \rfloor \leqslant m$. For example, if $P = \text{ABCDEF}$ and $q = 3$, then $P^0 = \text{AD}$, $P^1 = \text{BE}$ and $P^2 = \text{CF}$. Assume now that P occurs at $T[i..i + m - 1]$. From the definition of P^j it directly follows that $P^j[h] = T[i + j + hq]$, where $j = i \bmod q$ and $h = 0 \ldots m' - 1$. This means that we can use the set $\mathcal{P}$ as a filter for the pattern P, and that the filter needs only to scan every qth character of T. All the patterns must be searched simultaneously. Whenever an occurrence of P^j is found in the text, we must verify if P also occurs, with the corresponding alignment.

This method clearly works for parameterized matching as well. We generate the set of patterns $\mathcal{P}$, and also *prev*-encode them. In the search phase the text is also encoded on-line, encoding only every qth symbol, but assuming that they are consecutive. In other words, every parameter offset is effectively divided by q to agree with the encoding of the patterns. Finally, the verifica-

tion phase checks if $prev(P) = prev(T[v \ldots v+m-1])$, where v is the starting position of a potential match.

The search of the pattern set can be done using the parameterized Shift-Or algorithm. This is possible by concatenating and packing the set of patterns into a single machine word [12,4]. Another alternative is to use the parameterized version [14] of Aho–Corasick algorithm [1]. Both lead to the same average case running time, but the latter does not require that $m \leqslant w$, as it is not based on bit-parallelism. We denote the Shift-Or based algorithm as PFSO. The filtering time is $O(n/q)$. The filter searches the exact matches of q patterns, each of length $\lfloor m/q \rfloor$. We are not able to analyze the exact effect of the parameter alphabet to the probability that two randomly picked symbols match. However, if we assume that a constant fraction ε of the pattern positions are randomly selected to have a randomly selected symbol from Σ, then the probability that P^j occurs in a given text position is $O((1/\sigma)^{\lfloor \varepsilon m/q \rfloor})$. A brute force verification cost is in the worst case $O(m)$ (but only $O(1)$ on average). To keep the total time at most $O(n/q)$ on average, we select q so that $n/q = mn/\sigma^{\varepsilon m/q}$, i.e., $q = O(m/\log_\sigma(m))$. The total average time is therefore $O(n \log_\sigma(m)/m)$. This is optimal [17] within a constant factor.

Finally, note that this method works for searching r patterns simultaneously. The only difference is that we search q pieces of all the r patterns simultaneously, and verify the corresponding pattern whenever any of the rq pieces match. Redoing the analysis we obtain that the $O(\log(m))$ factor is replaced with $O(\log(rm))$. In this case we prefer using the Aho–Corasick based algorithm [14], since the number of patterns it can handle does not depend on w.

4. Parameterized backward trie matching

We now present an algorithm based on Backward DAWG Matching (BDM) [7,10]. BDM is optimal on average, i.e., it runs in $O(n \log_\sigma(m)/m)$ average time. We call our parameterized version of BDM as Parameterized Backward Trie Matching, PBTM, for short. In the preprocessing phase PBTM builds a trie for the encoded suffixes of the reversed pattern. A trie is a rooted tree, where each edge is labeled by a symbol. The edges of the path from the root node to some leaf node then spell out the string of symbols stored into that leaf. The pattern in reverse is denoted by P^r. The set of its suffixes is $\{P^r[i \ldots m-1] \mid 0 \leqslant i < m\}$ (note that this corresponds to the prefixes of the original pattern). Each suffix is then encoded with *prev*, and the encoded strings are inserted into a trie. For example, if $P = \text{AZBZXBXY}$, then the set of stored strings is $\{00b20b2a, 0b20b2a, b00b2a, 00b2a, 0b2a, b0a, 0a, a\}$, The trie allows efficient searching of any pattern substring that occurs in P^r. A brute force algorithm for this takes $O(m^2)$ time, but can be improved to $O(m)$ by using efficient suffix tree construction algorithms for parameterized strings [9]. An alternative to the trie is suffix array [15], i.e., the trie can be replaced with sorted array of *prev* encoded suffixes of the reverse pattern. For the above example string, $P = \text{AZBZXBXY}$, we create an array $A = \{00b20b2a, 00b2a, 0a, 0b20b2a, 0b2a, a, b00b2a, b0a\}$. Following an edge in the trie can then be simulated by a binary search in the array. We call the resulting algorithm PBAM. The benefit is that the array based method is easy to implement space efficiently since only one pointer is needed for each suffix.

We now show how this can be used for efficient search. Assume that we are scanning the text window $T[i \ldots i+m-1]$ *backwards*. The invariant is that all occurrences that start before the position i are already reported. The text window is *prev*-encoded (backwards as well) as we go, and the read substring of this window is matched against the trie. This is continued as long as the substring can be extended without a mismatch, or we reach the beginning of the window. If the whole window can be matched against the trie, then the pattern occurs in that window. Whether the pattern matches or not, some of the occurrences may still overlap with the current window. However, in this case one of the suffixes stored into the trie must match, since the reverse suffixes are also the prefixes of the original pattern. The algorithm remembers the longest such suffix, that is not the whole pattern, found from the window. The window is then shifted so that its starting position will become aligned with the last symbol of that suffix. This is the position of the next possible pattern occurrence. If the length of that longest suffix was ℓ, the next window to be searched is $T[i+m-\ell \ldots i+m-1+m-\ell]$. The shifting technique is exactly the same independent of whether or not the pattern occurs in the current window This process is repeated until the whole text is scanned.

Some care must be taken to be able to do the encoding of the text window in $O(1)$ time per read symbol. To achieve constant time per symbol we must use an auxiliary array *prv* (as before) to store the position of the last occurrence for each symbol. We cannot afford to initialize the whole array for each window, so before shifting the window we rescan the symbols just read in the current window, and reinitialize the array only for

```
1   root ← EncSTrie(P^r)
2   for i ← 0 to λ − 1 do prv[σ + i] ← −∞
3   i ← 0
4   while i < n − m do
5       j ← m; shift ← m; u ← root
6       while u ≠ null do
7           c ← T[i + j − 1]
8           if c ∈ Λ then
9               c ← m − j − prv[T[i + j − 1]] + σ
10              if c > σ + m − 1 then c ← σ
11              prv[T[i + j − 1]] ← m − j
12          j ← j − 1
13          u ← child(u, c)
14          if u ≠ null AND issuffix(u) then
15              if j > 0 then shift ← j else report match
16      for k ← i + j to i + m − 1 do if T[k] ∈ Λ then prv[T[k]] ← −∞
17      i ← i + shift
```

Algorithm 2. PBTM(T, n, P, m).

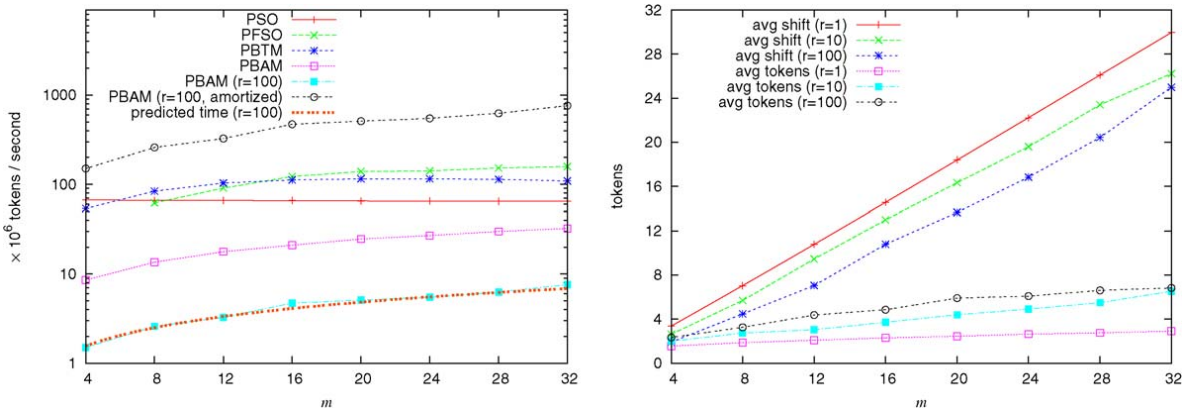


Fig. 1. Left: the search speed in 10^6 tokens/second. Right: the average shift and average number of tokens inspected in each window of length m.

those symbols. This ensures O(1) total time for each symbol read. Algorithm 2 gives the code.

The average case running time of this algorithm depends on how many symbols x are examined in each window. Again, if we make the simplifying assumption that a constant fraction of the pattern positions are randomly selected to have a randomly selected symbol from Σ, then the original analysis of BDM holds for PBTM as well, and the average case running time is $O(n \log_\sigma(m)/m)$. For general alphabets and for the PBAM version the time must be multiplied by $O(\log(m))$. Finally, this algorithm can be easily modified to search r patterns simultaneously. Basically, if all the patterns are of the same length, this generalization requires just storing all the suffixes of all the patterns into the same trie. This results in $O(n \log_\sigma(rm)/m)$ average time. With modest additional complexity patterns of different lengths can be handled as well in the same way as with regular BDM [11].

5. Comparison

For a single pattern our only competitor [5] is based on (Turbo) Boyer–Moore [8,10] algorithm. However, BM-type algorithms are known to be clearly worse than the more simple bit-parallel and suffix-automaton based approaches [16], an this becomes more and more clear as the pattern length increases. Moreover, BM-type algorithms have poor performance when generalized for multiple string matching [16]. As for the multiple matching, our only competitor [14] is the algorithm based on Aho–Corasick automaton, but as detailed in Section 3, we can use exactly their algorithm (even the same implementation) as a fast filter to obtain (near) optimal average case time. Their worst case time can be also preserved. Hence, their algorithm cannot beat ours. We note that all our algorithms can be improved to take only $O(n)$ (or $O(n \log(rm))$ for unbounded alphabets) worst case time. PFSO can be combined with PSO (as

in [12]) and PBTM with the algorithm in [14]. See also [3,10] for similar techniques.

Our goals in this paper are two-folded. First, to develop algorithms that have optimal average case running time for both single and multiple patterns. All the previous results only prove optimal worst case time. Second, to be practical, i.e., to develop algorithms that are simple to implement and have good average case time in practice. We now show that our algorithms behave like predicated, with realistic real world data.

5.1. Experimental results

We have implemented the algorithms in C++, and compiled them with Borland C++ Builder 6. We performed the experiments on the AMD Sempron 2600+ (1.88 GHz) machine with 768 MB RAM, running Windows XP. A tokenized string of concatenated Java source files (taken from various open source projects, such as jPOS, smppapi, and TM4J) was used as a text to be searched. The tokenization procedure (based on JavaCC[2] parser) converted an input file into a sequence of two-byte codes, representing single characters, reserved Java words and distinct identifiers. The initial string had a size of 5.48 MB, and after encoding it consisted of 1259799 tokens, including 51 reserved Java words and 10213 unique identifiers. A set of 100 patterns for each length reported was randomly extracted from the input text. We report the average number of tokens searched per second for each algorithm.

Fig. 1 summarizes the results. PSO denotes the basic parameterized shift-or algorithm, PFSO the fast parameterized shift-or, PBTM the parameterized backward trie matching algorithm, and PBAM the suffix array version of PBTM. For short patterns plain PSO and PBTM give the best results. PSO is the fastest for $m < 8$, and PBTM takes over until $m = 16$, and PFSO dominates for longer patterns in case of optimal q selection. For $m \in \{8, 12, 16, 20, 24, 28, 32\}$ we used $q = \{2, 3, 4, 4, 4, 5, 6\}$, respectively. For long patterns PBTM suffers from the large alphabet size. In our implementation we used arrays to implement the trie nodes and for long patterns the trie requires a lot of initialization time and memory, not fitting into the CPU cache. PBAM does not have this flaw, but the binary search step needed for each accessed text symbol makes it comparatively slow. We also experimented with the multipattern version of PBAM, searching $r = 100$ patterns simultaneously. The plot shows that while the raw speed is reduced, the amortized speed per pattern is clearly better than for any of the single pattern matching algorithms. The time also coincides nicely with the theoretical curve $O(n \log_\sigma(rm) \log_2(rm)/m)$, supporting our analysis. This is also clear given the right plot, showing the average number of tokens inspected in each text window, and the average shift for $r = 1, 10, 100$. These behave like in random texts supporting our assumptions in the analysis.

We have shown how two well-known algorithms, namely Shift-Or and BDM, can be generalized for parameterized matching. The algorithms are easy to implement, and work well in practice.

[2] http://javacc.dev.java.net/.

References

[1] A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search, Comm. ACM 18 (6) (1975) 333–340.
[2] A. Amir, M. Farach, S. Muthukrishnan, Alphabet dependence in parameterized matching, Inform. Process. Lett. 49 (3) (1994) 111–115.
[3] R.A. Baeza-Yates, String searching algorithms revisited, in: Proceedings of WADS'89, in: Lecture Notes in Computer Science, vol. 382, Springer, Berlin, 1989, pp. 75–96.
[4] R.A. Baeza-Yates, G.H. Gonnet, A new approach to text searching, Comm. ACM 35 (10) (1992) 74–82.
[5] B.S. Baker, Parameterized pattern matching by Boyer–Moore-type algorithms, in: Proceedings of the 6th ACM–SIAM Annual Symposium on Discrete Algorithms, San Francisco, CA, 1995, pp. 541–550.
[6] B.S. Baker, Parameterized duplication in strings: algorithms and an application to software maintenance, SIAM J. Comput. 26 (5) (1997) 1343–1362.
[7] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M.T. Chen, J. Seiferas, The smallest automaton recognizing the subwords of a text, Theoret. Comput. Sci. 40 (1) (1985) 31–55.
[8] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Comm. ACM 20 (10) (1977) 762–772.
[9] R. Cole, R. Hariharan, Faster suffix tree construction with missing suffix links, in: Proceedings of ACM–STOC'00, Portland, Oregon, 2000, pp. 407–415.
[10] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, W. Rytter, Speeding up two string matching algorithms, Algorithmica 12 (4) (1994) 247–267.
[11] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, Oxford, 1994.
[12] K. Fredriksson, Sz. Grabowski, Practical and optimal string matching, in: Proceedings of SPIRE'2005, in: Lecture Notes in Computer Science, vol. 3772, Springer-Verlag, Berlin, 2005, pp. 374–385.
[13] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, Cambridge, 1997.
[14] R.M. Idury, A.A. Schäffer, Multiple matching of parameterized patterns, Theoret. Comput. Sci. 154 (2) (1996) 203–224.
[15] U. Manber, G. Myers, Suffix arrays: a new method for on-line string searches, SIAM J. Comput. 22 (5) (1993) 935–948.
[16] G. Navarro, M. Raffinot, Flexible Pattern Matching in Strings, Cambridge University Press, Cambridge, 2002.
[17] A.C. Yao, The complexity of pattern matching for a random string, SIAM J. Comput. 8 (3) (1979) 368–387.

Publication [MTK06]

M. Mozgovoy, V. Tusov, V. Klyuev. The Use of Machine Semantic Analysis in Plagiarism Detection. *Proc. of the 9th International Conference on Humans and Computers*, Japan, 2006, p. 72-77

The Use of Machine Semantic Analysis in Plagiarism Detection

Maxim Mozgovoy*
University of Joensuu, Finland
mmozgo@cs.joensuu.fi

Vitaly Tusov
St. Petersburg State University, Russia
tusovvitalij@mail.ru

Vitaly Klyuev
University of Aizu, Japan
vkluev@u-aizu.ac.jp

Abstract

Plagiarism detection systems are known for years in the university community. However, most of the existing detectors for the natural language texts use rather simple comparison methods that make the instances of plagiarism easy to hide. The software, designed for plagiarism detection in computer programs, utilizes far more advanced techniques. We propose a method, which adds functionalities similar to tokenization and tree matching, to the natural language texts-oriented detectors. This method requires noticeable work to be applied in practice, but also makes use of the existing software for parsing and word sense disambiguation.

Keywords

machine semantic analysis, plagiarism detection, string matching, plagiarism, computational linguistics.

1. Introduction

Plagiarism in universities remains in the scope of interest of researchers for years. Scientific investigations cover various aspects of plagiarism: its origins, pedagogical and ethical issues, plagiarism prevention and detection, people's attitude, legal affairs and honor codes, etc. From the point of view of practical computer science, one of the most interesting directions is plagiarism detection.

Numerous systems were developed in recent years to detect plagiarism in natural language texts as well as in computer programs. Here we propose a possible technique that can help to improve existing natural language-oriented plagiarism detection software. This technique can be roughly treated as an analogue of a well-known *tokenization* procedure in program code-oriented plagiarism detection systems. Furthermore, we discuss a way of utilizing language parsers to deal with the rephrasing of the sentences.

Our current studies are based on *semantic analyzer* for the Russian language [1], but other similar solutions can be used as well. The detection system, which is described here, is just a part of our research on natural language processing. We are also trying to use semantic analysis for information retrieval tasks and for machine translation.

2. Related Works

Most existing plagiarism detectors are specially designed to process either program source code or natural language texts. In the first case the system usually treats a submitted collection of documents as hermetic and performs a pairwise comparison between single submissions only. Such projects utilize advanced techniques to detect partial matches (RKS-GST [2], matching in the repository [3]), and regular changes of the code structure (tokenization [4], p-matching [5]). The systems, designed to find similarities in the natural language texts, mainly search the Internet for the possible matches. Generally, they do not use sophisticated comparison methods, aiming mostly at processing speed and wide coverage (e.g. the developers of *Turnitin* [6] system claim they maintain "a huge database of books and journals, and a database of the millions of papers already submitted").

"Hermetic" systems for plagiarism detection in the natural language texts exist as well, though they are little-known. We can mention, e.g. CopyCatch Gold [7], YAP3 [8], and WCopyfind [9]. As a rule, the detection software can find only partial exact matches: rephrasing and rewording can conceal the evidence of plagiarism. CopyCatch Gold reduces the effect of rewording by taking into account only

* The corresponding author.

hapax legomena words (those that appear only once in the text) during the comparison, but this technique is not very reliable.

3. Tokenization

Tokenization [4] is a well-known method that makes useless all kinds of renaming tricks in plagiarism in computer programs (such as variable renaming and changing the type of loop structure). Tokenization algorithms substitute the 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;** this line 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 to the same tokenized sequence (the aforementioned example is taken from [10]).

Tokenization can be treated as substitution of single elements of some class by the name of the class itself. E.g. 5, 11.5 and -32 are elements of the class <VALUE>. In the natural language texts we can use the same approach. For instance, the words *device* and *gadget* are interchangeable in many contexts. If we substitute these words by the name of their class <MECHANISM>, such rewording will be useless for the plagiarizer.

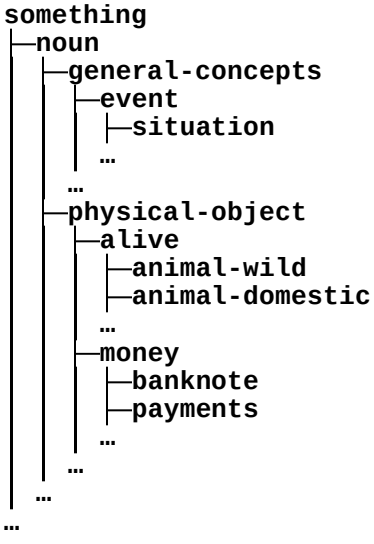
4. Word Classes

The above described technique can be quite easily implemented by having a dictionary that matches every word of a natural language with the corresponding class. Our current version of the system (for Russian) includes more than 1600 classes that form a hierarchy. The small extraction from this tree is shown in Fig. 1.

For example, a class <PHYSICAL-OBJECT> has a subclass <ALIVE>, having, in its turn, a subclass <ANIMAL> that includes classes <ANIMAL-WILD> and <ANIMAL-DOMESTIC>. Undoubtedly, the problem of classification is very nontrivial, and no "best classification" can be invented. We created only one possible hierarchy that (according to our studies) satisfactorily reflects general knowledge about human environment. A variation of a specially created hierarchy might be helpful in order to process documents, dedicated to some narrow fields. For example, it is usually reasonable to consider *gadget* and *device* as direct successors of the class <MECHANISM>, but for the technical texts a more detailed classification of mechanisms will produce better results.

The use of subclasses can help to tune the tokenizer. For example, we might want to find more plagiarisms by widening the generalizations, e.g. it is possible to substitute the word *fox* with the more general class <ANIMAL> instead of <ANIMAL-WILD>.

Figure 1. A Fragment of Concept Classes Tree



The obvious difficulty concerns polysemantic words and homonyms. For instance, the system should select the correct class for the word *table* from the two alternatives — <FURNITURE> and <DRAWING/TABLE>. We may suggest to use any tool for word sense disambiguation (WSD), referenced in [11]. Our software relies on the results, provided by the semantic analyzer that performs WSD as well.

5. Fast Plagiarism Detection Algorithm

To obtain working software, we took a system [3], and substituted the tokenization module with the natural language version. The corresponding author is a member of the team that created the aforementioned system.

The system is intended for hermetic, many-to-many comparison of all files of the submitted collection of documents that contain Java listings. Most hermetic detection programs perform naïve pairwise file-to-file comparison, which results in $O(f(n)N^2)$ complexity, where N is the number of files in the collection and $f(n)$ is the time to make the comparison between one pair of files of length n. Our software tries to decrease the algorithmic complexity while preserving almost the same quality of detection.

The system firstly creates a suffix array from the tokenized collection of files. A suffix array is a lexicographically sorted array of all suffixes of a given string. It allows us to quickly find a file (or files), containing any given substring. A binary search is utilized to achieve this.

To find all collection files that are similar to a given query file, the system executes Alg. 1. It tries to find the substrings of the tokenized query file, Q[1..q], in the suffix array, where q is the number of tokens. Matching substrings are recorded and each match contributes to the similarity score. The algorithm takes contiguous non-overlapping token substrings of length γ from the query file and searches all the matching substrings from the index. These matches are recorded into a 'repository'. This phase also includes a sanity check as overlapping matches are not allowed.

Algorithm 1. Search a File in a Collection

```
p = 1 // the first token of Q
WHILE p ≤ q – γ + 1
  find Q[p...p + γ – 1] from the suffix array
  IF Q[p...p + γ – 1] was found
    UpdateRepository
    p = p + γ
  ELSE
    p = p + 1
FOR EVERY file Fi in the collection
  Similarity(Q, Fi) = MatchedTokens(Fi)/q
```

In Alg. 2, the system encounters two types of collisions. The first one appears when more than one match is found in the same file. If several matches that are found correspond to the same indexed file, these matches are extended to Γ tokens, $\Gamma \geq \gamma$, such that only one of the original matches survives for each indexed file. Therefore, for each file in the index, the algorithm finds all matching substrings that are longer than other matching substrings and whose lengths are at least γ tokens.

Algorithm 2. Update the Repository

```
Let S be the set of matches of Q[p...p+γ–1]
IF some elems of S are found in the same file
  leave only the longest one
FOR every string M from the remaining list S
  IF M doesn't intersect with repository elems
    insert M to the repository
  ELSE IF M is longer than conflicting elems
    remove all conflicting repository elements
    insert M to the repository
```

The second type of collision is the reverse of the first problem: we should forbid the situation when two different places in the input file correspond to the same place in some collection file. To resolve collisions we use 'longest wins' heuristics. We sum the lengths of all the previous matches that intersect with the current one, and if the current match is longer, we use it to replace the intersecting previous matches.

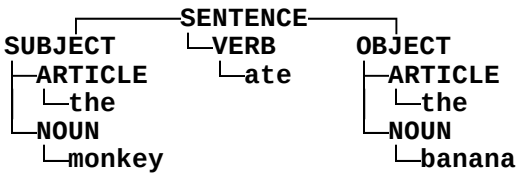
The complexity of Algorithm 1 is highly dependent on the value of the γ parameter. Line 3 of Algorithm 1 takes $O(\gamma + \log n)$ average time, where is n the total number of tokens in the collection (assuming atomic token comparisons). If we make the simplifying assumption that two randomly picked tokens match each other (independently) with fixed probability p, then on average we obtain np^γ matches for substrings of length γ. If Q was found, we call Algorithm 2. Its total complexity is, on average, at most $O((q/\gamma \cdot np^\gamma)^2)$. To keep the total average complexity of Algorithm 1 to at most $O(q(\gamma + \log n))$, it is enough that $\gamma = \Omega(\log_{1/p} n)$. This results in $O(q \log n)$ total average time. Since we require that $\gamma = \Omega(\log n)$, and may adjust γ to tune the quality of the detection results, we state the time bound as $O(q\gamma)$. Finally, the scores for each file can be computed in $O(N)$ time. To summarize, the total average complexity of Algorithm 1 can be made $O(q(\gamma + \log n) + N) = O(q\gamma + N)$. The $O(\gamma + \log n)$ factors can be easily reduced to $O(1)$ (worst case) using suffix trees with suffix links, instead of suffix arrays. This would result in $O(q + N)$ total time.

This analysis does not include tokenization, but it is a linear process (both for Java files and for the natural language texts), and the number of tokens depends linearly on the file length.

6. Tree Matching

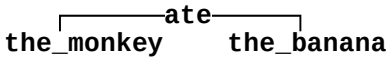
Any sentence of the given text can be automatically represented in the form of the tree, which reflects the structure of the sentence. The principles of organization of such parse trees still serve as the subject of wide discussions. Most automatic English parsers use Chomsky-styled Penn Treebank grammars [12], based on the traditional linguistic approach to the syntax analysis. For example, the phrase *the monkey ate the banana* will be parsed by such software as shown in Fig. 2.

Figure 2. Parsing the Sentence



Our semantic analyzer also builds a parse tree for any given sentence, but it is not based on Chomsky grammars (they are not well-suitable for the Russian language, because the order of the parts of a sentence in Russian is not fixed). The semantic analyzer treats the sentence as a control structure, having a functional nature. More specifically, it considers the sentence as a superposition of *words-functions* that depend on *words-arguments*.

The parse tree for the same phrase *the monkey ate the banana* in this model will look like this:



Here the word *ate* is considered as a computable function of two arguments: the_monkey and the_banana.

Having ready-made parse trees (of any kind), we can invoke a tree matching procedure. This technique is described in [13] for the case of plagiarism detection in program code.

Initially the algorithm builds a flowchart-styled parse tree for each file to be analyzed. Then for each pair of files, the algorithm performs a rough "abstract comparison", when only types of the parse tree elements (like ASSIGNMENT, LOOP, BRANCHING) are taken into account. This is done recursively for the each level of tree nodes.

If the similarity percentage becomes lower than some threshold at some step, the trees are immediately treated as not similar.

If the abstract comparison indicates enough similarity, a special low-level "micro comparison" procedure is invoked. At this point each node represents an individual statement. Thus, each tree node turns into a separate subtree that has to be compared with the corresponding subtree taken from another file.

Note that the "abstract comparison" is a step when tokenized sequences are compared, so for our purposes (plagiarism detection over tokenized texts) we can skip the next "micro comparison" procedure. This technique seems to be the most advanced way of comparing structured documents, but our results in this direction are still very preliminary for any kind of evaluation.

On the other hand, it is already clear that the tree matching can help to reveal rewording. If we treat the children of every tree node as an unordered collection of nodes, e.g. the phrases *the monkey ate the banana* and *the banana was eaten by the monkey* will be very close after the tokenization.

7. Evaluation

The evaluation part is a very problematic issue for any kind of plagiarism detection system. It is especially hard for the software that searches the Internet for the possible occurrences of plagiarism, but even papers on "hermetic" systems usually just show the positive sides of the proposed approach [4, 13]. However, we can examine the reports that are produced by different plagiarism detection software when used on the same dataset.

The original system [3] was evaluated by using such "jury" method. The programs utilized for the analysis include MOSS [14], JPlag [2] and Sherlock [4]. Every system printed a report about the same real collection, consisting of 220 undergraduate students' Java programs (varying in size from 2 KB to 50 KB; the median length is 15 KB). Although the 'opinions' of all the tested systems are different for many of the files, most files are either detected or

rejected by the majority of systems. This simple approach (to consider only detection or rejection) allows us to organize a 'voting' experiment. Let S_i be the number of 'jury' systems (MOSS, JPlag and Sherlock), which marked file *i* as suspicious. If $S_i \geq 2$, we should expect our system to mark this file as well. If $S_i < 2$, the file should, in general, remain unmarked. For the test set consisting of 155 files marked by at least one program, our system agreed with the 'jury' in 115 cases (and, correspondingly, disagreed in 40 cases). This result is more conformist than the results obtained when the same experiment was run on the other 3 tested systems. Each system was tested while the other three acted as jury.

For the evaluation of the new system we used a collection of 350 documents taken from the NEWSru.com news server. Each document had an informative title and was assigned to one of the following categories: *In Russia*, *In the World*, *Economics*, *Religion*, *Criminal*, *Sport*, and *Culture*. The size of the articles varies from 450 bytes to 19 KB with the median size of about 2 KB. The typical article consists of 8-12 small paragraphs that are made of strict narrative sentences and quotations.

This selection was based on the assumption that the newsreels often publish different documents on the same topic (though we do not expect direct plagiarism in this case), so the possibility to find similar files is quite high. Since we do not know about any other plagiarism detection systems that use natural language processing techniques, the results were analyzed manually.

The system found 20 relevant pairs of similar documents (with at least 4% degree of similarity). The typical examples include:

- A pair of documents about the solar eclipse on 20th of March. The first tells about the countries where this phenomenon is observed; the second is dedicated purely to the observation of the eclipse in Russia.
- A pair of documents on the weather conditions in Europe. The first is about floods in the EU; the second contains some weather predictions for the EU (including subsequent floods).
- A pair of documents on rumors about Russian military assistance to the Iraqi government in March of 2003. The first outlines the position of Moscow; the second states the reaction of Washington.

After tokenization the similarity degrees of the same file pairs increased (in most cases) by a factor of 1.5 or (in few cases) remained the same. Meanwhile, four additional false pairs were detected (but with very low similarity ratios that did not exceed 4-5%).

There are typical situations encountered in the experiment that noticeably affected the detection process after tokenization. They include:

- Changes to grammar cases in Russian. The phrases *in one of resorts* and *of one of resorts* are not matched at due to the changes of the endings of the words. After tokenization they become almost identical.
- The use of distinct words of the same classes in the same contexts in different documents. The phrases *the residence in Greece* and *the residence in Athens* do not match, but do match after the tokenization (*Greece* and *Athens* are translated to the same class <PLACE>).

The latter case is related to many mismatches as well. For example, the phrases *Vladimir Putin claimed* and *George Bush claimed* are treated as the same sequence <NAME><NAME><SPEAK> after the tokenization. It may be argued, though, that the system of classes we used was not specially designed for plagiarism/similarity detection procedures. A more advanced hierarchy may include a careful taxonomy that minimizes such collisions.

It should be noted that in the simplest case the tokenization can be considered as a variation of stemming technique that is widely used in information retrieval. Although, for some languages (including Russian) stemming procedures are not simple, since they have to deal with many non-trivial grammatical issues.

The use of tokenization results in the immediate increase of the number of matches. Tokenization of level 0 (stemming), level 1 (with the terminal classes in the hierarchy) and level 2 (with the direct ancestors of the terminal classes) makes sense, but the use of tokenization of higher levels results in many false matches, since the classes become too general. For example, tokenization of level 3 substitutes the word *cat* with the class name <ALIVE>. The same class corresponds to all alive objects, such as *worm*, *chairman* or *wife*.

The system also did not detect several file pairs that could be treated as similar under certain conditions. For example, some documents can have only a few common substrings, but most human readers consider them as similar. The manual analysis shows that the number of such pairs in our collection is less than five, and the corresponding files are not originated from the same source, i.e. they do not contain instances of plagiarism.

8. Conclusion

Plagiarism detection for text in natural languages is a challenge. Most natural language processing tools, such as parsers and taggers remain unused by the authors of plagiarism detection systems. Also these tools are language dependent and designed for English.

Our approach gives a possible solution to make a language independent system to determine plagiarism in collections of the texts. The key idea behind it is the use of hierarchies of concepts and the functional style of representing the sentences. To implement the system, we adopted algorithm [3], applied the tokenization technique and the tree matching procedure. We utilized the concept hierarchy for the Russian language. Our tests showed the promising results which include intelligent tokenization and high speed processing of the text data (O(q + N) is required to test a query file of size q against a collection of N files). The tree matching procedure is still very experimental, but we believe that it can significantly improve the quality of plagiarism detection. Our solution is scalable (see [3]), so it is suitable for large essays banks.

References

[1] V.A. Tusov, *Computer Semantics of the Russian Language (in Russian)*, S.-Petersburg University Press, S.-Petersburg, 2004.

[2] L. Prechelt, G. Malpohl, and M. Philippsen, *JPlag: Finding Plagiarisms among a Set of Programs*, Technical report, Fakultät für Informatik, Universität Karlsruhe, Germany, 2000.

[3] M. Mozgovoy, K. Fredriksson, D. White, M. Joy, and E. Sutinen, "Fast Plagiarism Detection System", *Lecture Notes in Computer Science*, vol. 3772, 2005, pp. 267-270.

[4] M.S. Joy, M. Luck, "Plagiarism in Programming Assignments", *IEEE Transactions on Education*, vol. 42(2), 1999, pp. 129-133.

[5] B.S. Baker, "Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance", *SIAM Journal on Computing*, vol. 26(5), 1997, pp. 1343-1362.

[6] Turnitin: www.turnitin.com

[7] CopyCatch Gold: www.copycatchgold.com

[8] M.J. Wise, "YAP3: Improved Detection of Similarities in Computer Program and Other Texts", *Proceedings of SIGCSE '96*, 1996, pp. 130-134.

[9] WCopyfind: plagiarism.phys.virginia.edu

[10] M. Mozgovoy, "Desktop Tools for Offline Plagiarism Detection in Computer Programs", *Informatics in Education*, vol. 5(1), 2006, pp. 97-112.

[11] Ph. Edmonds, A. Kilgarriff (Eds.), *Journal of Natural Language Engineering (Special Issue Based On Senseval-2)*, vol. 9(1), 2003.

[12] M.P. Marcus, B. Santorini, M.A. Marcinkiewicz, "Bulding a large annotated corpus of English: the Penn Treebank", *Computational Linguistics*, vol. 19, 1993, pp. 313-330.

[13] B. Belkhouche, A. Nix, J. Hassell, "Plagiarism Detection in Software Designs", *Proceedings of the 42nd Annual Southeast Regional Conference*, 2004, pp. 207-211.

[14] S. Schleimer, D. S. Wilkerson, A. Aiken, "Winnowing: Local Algorithms for Document Fingerprinting", *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 2003, pp. 76-85.

Publication [MKK07]

M. Mozgovoy, S. Karakovskiy, V. Klyuev. Fast and Reliable Plagiarism Detection System. *Proc. of FIE'07 Conference, 2007*.

Fast and Reliable Plagiarism Detection System

Maxim Mozgovoy[1], Sergey Karakovskiy[2], and Vitaly Klyuev[3]

Abstract – **Plagiarism and similarity detection software is well-known in universities for years. Despite the variety of methods and approaches used in plagiarism detection, the typical trade-off between the speed and the reliability of the algorithm still remains. We introduce a new two-step approach to plagiarism detection that combines high algorithmic performance and the quality of pairwise file comparison. Our system uses fast detection method to select suspicious files only, and then invokes precise (and slower) algorithms to get reliable results. We show that the proposed method does not noticeably reduce the quality of the pairwise comparison mechanism while providing better speed characteristics.**

Index Terms – Plagiarism detection, similarity detection, string matching.

INTRODUCTION

A wide range of plagiarism and similarity detection systems was developed in recent years. Basically their task is to find similarities in files, which can indicate either plagiarism (in case of students' works) or code duplication (in case of a software project analysis). The quality of the system is primarily determined by the method of similarity calculation. The same method usually has a high influence on the speed of the detection. Typically, the faster is detection routine, the less precise results it provides.

In our work we study the possibility of using the combined approach: the faster (and less precise) algorithm performs the initial selection of suspicious files, and then the more reliable (and slower) routine calculates similarity ratios for file pairs. It should be noted that we concentrate on a so-called "offline" (or "hermetic") plagiarism detection, which deals with analysis of local file collections. "Online" plagiarism detection (search for similar documents in the Internet) is related more to information retrieval, so coverage can turn out to be much more important than precision.

SIMILARITY DETECTION TECHNIQUES

A good example of speed-reliability trade-off is found in the formerly popular attribute counting approach. Attribute counting systems (such as [1] and [2]) create special "fingerprints" for collection files, including metrics, such as average line length, file size, average number of commas per line, etc. The files with close fingerprints are treated as similar. Clearly, small fingerprint records can be compared rapidly, but this approach is now considered unreliable, and rarely used nowadays [3].

Modern plagiarism detection systems usually implement certain content-comparison techniques. The most popular approaches include heuristic string tiling (finding the joint coverage for a pair of files) [4, 5], and parse trees comparison [6, 7]. Usually these algorithms work for file pairs, so the comparison routine should be called for each possible file pair found in the input collection. It means that $O(f(n)N^2)$ time is required to perform the detection. Here N is the number of files in the collection, and $f(n)$ is the time needed to compare two files of length n.

Fast Plagiarism Detection System (FPDS) [8] tries to improve the algorithmic performance of plagiarism detection by utilizing a special indexed data structure (suffix array) to store input collection files. A special heuristic search routine is used to compare any given file against the whole collection at once. The complexity of the complete detection procedure is $O(nN\gamma + N^2)$, where N is the total number of files, n is the average file length, and γ is a special finely-tunable constant ($\gamma = \Omega(\log nN)$ should be used for the best performance). In terms of quality, FPDS shows the results, which are close to the ones, provided by other content-comparison systems.

TOKENIZATION

Tokenization [9] is a commonly-used technique that fights against renaming variables and changing loop types in computer programs. Simple tokenization algorithms substitute the elements of program code with single tokens. For example, all identifiers can be substituted with <IDT>, and all values with <VALUE> tokens. So, a line **a = b + 45;** will be replaced by **<IDT>=<IDT>+<VALUE>;**. Therefore, renaming variables will not help the plagiarizer.

A more advanced example of tokenization is p-match algorithm [10] that keeps track of usage of the variable names. If another file has a variable appearing in the same context, it is treated as identical.

All modern plagiarism detection systems, aimed at program code analysis, implement some tokenization-like procedure.

PLAGGIE AND FPDS

Plaggie is a recent open source content comparison-based plagiarism detection system [11]. It is based on a simplified algorithm, implemented in better-known JPlag project [5]. In its turn, JPlag develops further the idea of the Running-Karp-Rabin Greedy-String-Tiling (RKR-GST) algorithm, used in

[1] Maxim Mozgovoy, University of Joensuu, mmozgo@cs.joensuu.fi
[2] Sergey Karakovskiy, St. Petersburg State University, sergey.karakovskiy@gmail.com
[3] Vitaly Klyuev, University of Aizu, vkluev@u-aizu.ac.jp

1-4244-1084-3/07/$25.00 ©2007 IEEE

YAP3 tool [4]. Currently it is one of the most advanced and reliable content comparison methods [12, 13].

Though the speed of detection may vary due to project-specific heuristics, the (empirically obtained) expected running time for the pure RKR-GST routine is $O(N^2 n^{1.12})$, where N is the number of files in the input collection, and n is the average file size [14].

FPDS is a rapid content comparison-based plagiarism detector, aimed at better performance at the cost of a slight quality loss [8]. Our study showed that in most cases the results, provided by FPDS is reliable enough to be used in practice. On the other hand, the underlying algorithm itself (heuristic search in the indexed data structure) brings some features that are not always acceptable. Firstly, the search routine can skip certain matches, easily detectible through RKR-GST. Secondly, the algorithm is not optimized for finding continuous matches, so the similar chunks can be uniformly spread inside the files being analyzed. This makes harder to examine the similarities visually. In contrast, Plaggie (as well as JPlag) provide detailed HTML report of the results, including similarity ratios and schemes of detected overlappings for each suspicious file pair.

The algorithm used in FPDS has one special input constant γ, standing for "typical length of a match in tokens". Large γ values lead to more mismatches, but make algorithm work faster. Smaller γ provide more precise results, but if γ becomes too small, the system will report a lot of false matches (normally we assume that one or two-token match should not be treated as plagiarism). For similarity detection in computer programs, we consider $\gamma \in [10, 30]$ to be appropriate.

The relative speed of FPDS and Plaggie can be understood from the fig. 1, demonstrating the time needed to process a sample collection of 100, 200, 300, 400, and 500 files. We have used 500 distinct files taken from Apache Tomcat project source code as our collection, then selected random smaller subsets. The complete collection size is 4.58 MB; the median file size is 6.15 KB.

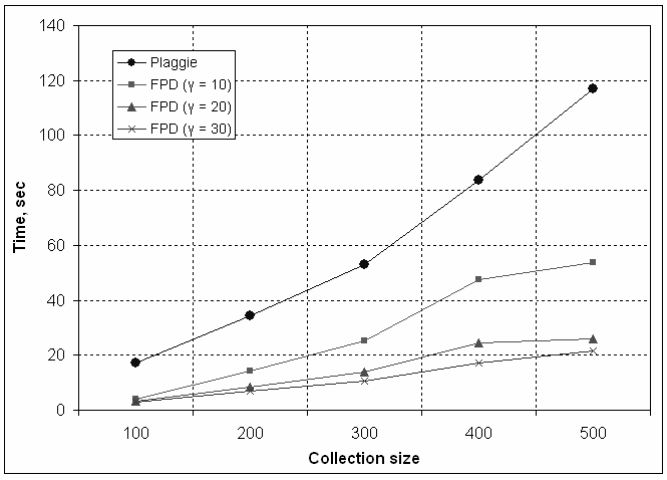


FIGURE 1
TIME REQUIRED FOR DETECTION: PLAGGIE VS. FPDS.

COMBINED APPROACH

High detection speed of FPDS with better reliability and result reporting capabilities of Plaggie suggest the idea of a combination of these two systems. Firstly, FPDS analyzes the input collection and outputs a set of suspicious files using a given similarity threshold. Then Plaggie is invoked for this set to produce a detailed and reliable report.

Technically, FPDS outputs a list of file-file similarities for every possible file pair:

```
file_i file_j sim(file_i, file_j)
```

The subset of suspicious documents we generate for further analysis by Plaggie includes every file `F`, for which

```
max(sim(F, G)) >= threshold
 G
```

It should be noted that `sim()` function is not symmetrical. If, for example, file `F` is small and file `G` is large, it can happen that `sim(F, G)` = 100% (if `G` contains `F`), while `sim(G, F)` is always less than 100%.

A combination of plagiarism detection systems we propose has to satisfy the following assumptions:

- The combined system should be noticeably faster than Plaggie.
- FPDS and Plaggie should "agree" in most cases; in other words, FPDS generally should not exclude files, treated as plagiarized by Plaggie, from the input set.

The intuitive reason for the second assumption is a close relationship between the algorithms used in both systems. Our experiments show that both these assumptions are correct. Let us consider the same five collections (100...500 files), consisting of Java source files. Table I demonstrates the agreement between Plaggie and FPDS (with $\gamma = 20$, $\gamma = 30$).

The systems mark files according to similarity thresholds ST1 and ST2. For Table I, ST1 = (0.4 for FPDS, 0.6 for Plaggie); ST2 = (0.6 for FPDS, 0.8 for Plaggie). It should be noted that since FPDS can skip matches, generally it provides lower similarity ratios that Plaggie. Therefore, we have to set higher similarity threshold for Plaggie in order to get closer reports. The percentage in parentheses shows the fraction of Plaggie-marked files, marked also by FPDS. Since these figures are enough high (81.9% on average), we can make a conclusion about reasonable agreement between the two systems. Therefore, the use of FPDS as a filter does not noticeably reduce the quality of plagiarism detection.

The speed of combined system is considerably higher than Plaggie's. The comparison of detection time graphs is shown on the fig. 2.

There are many studies investigating the nature of plagiarism but we can say nothing about the probability distribution of suspicious files. Study [16] reported that 38% of 53 students in one class plagiarized another student's assignment or allowed other students to plagiarize their assignment. After including the teacher's policy concerning academic dishonesty into the course syllabus, the number of students involved in plagiarism decreased dramatically: Only

9% of 87 students were incriminated in such incidences. No software was used to discover the aforementioned cases. Another study [17] reported the results of a survey conducted among the teachers of computing schools in Great Britain. Approximately 50% of the aforementioned schools provided the data. Responses to the question about the number of students in the school caught plagiarizing in the last academic year were ranging from a very small fraction (0.0014%) to approximately 13%.

TABLE I
AGREEMENT IN PLAGGIE AND FPDS RESULTS

Collection 1 (100 files)					
	Files marked				
Sim	Plaggie	FPDS20	FPDS30	Plaggie ∩ FPDS20	Plaggie ∩ FPDS30
ST1	23	32	19	18 (78.3%)	16 (69.6%)
ST2	14	21	11	13 (92.9%)	10 (71.4%)
Collection 2 (200 files)					
	Files marked				
Sim	Plaggie	FPDS20	FPDS30	Plaggie ∩ FPDS20	Plaggie ∩ FPDS30
ST1	72	87	66	62 (86.1%)	56 (77.8%)
ST2	51	62	46	46 (90.2%)	41 (80.4%)
Collection 3 (300 files)					
	Files marked				
Sim	Plaggie	FPDS20	FPDS30	Plaggie ∩ FPDS20	Plaggie ∩ FPDS30
ST1	92	119	89	78 (84.8%)	69 (75.0%)
ST2	63	82	60	56 (88.9%)	50 (79.4%)
Collection 4 (400 files)					
	Files marked				
Sim	Plaggie	FPDS20	FPDS30	Plaggie ∩ FPDS20	Plaggie ∩ FPDS30
ST1	118	149	115	104 (88.1%)	93 (78.8%)
ST2	80	104	78	71 (88.8%)	64 (80.0%)
Collection 5 (500 files)					
	Files marked				
Sim	Plaggie	FPDS20	FPDS30	Plaggie ∩ FPDS20	Plaggie ∩ FPDS30
ST1	138	184	141	118 (85.5%)	108 (78.3%)
ST2	92	124	91	80 (87.0%)	71 (77.2%)

We have to note, finding suspicious files using any system is not a proof of plagiarism. They have to be investigated carefully by human inspection. A technique of this inspection is not trivial. For example, on large scale tests of Plaggie for the introductory course on programming, the number of suspicious files was large and teachers had to track the students exercise by exercise using the following criteria: Similarity reported by the system was 100%, the same student pairs were under suspect, and suspicious-looking programs consisted of more than 100 lines [11]. After this complicated filtering, a number of students were asked for explanation. 80% of detected cases were confirmed.

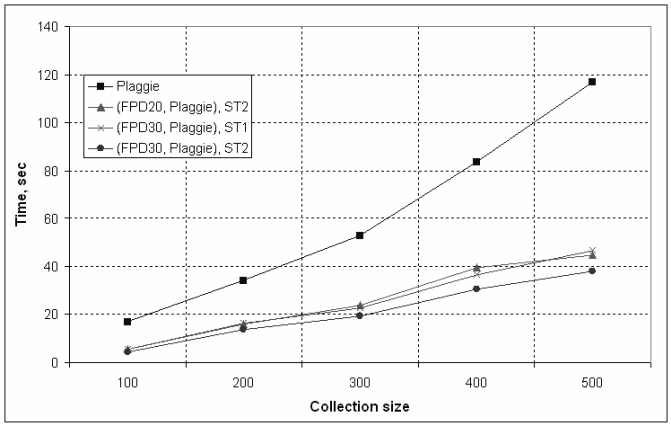


FIGURE 2
TIME REQUIRED FOR DETECTION: PLAGGIE VS. COMBINED SYSTEM.

EVALUATION

Evaluation of the systems is not a easy task. Authors and users published empirical evaluations [5, 8, 11]. Study [11] reported that Plaggie and JPlag produced practically the same results on a pool consisting of 65 student Java programs. Authors of FPDS [8] compared their system with JPlag, Sherlock and MOSS. These systems were acting as the "jury". They found: FPDS recognized 74% of suspicious files detected by the "jury". The test set consisted of 220 student Java programs. As it was noticed earlier, this result is very close to the comparison between FPDS and Plaggie: On average 81.9% of files detected by Plaggie as suspicious were detected by FPDS as well. We applied the nonparametric test of Spearman's rank correlation [15] to the data presented in Table 1. It is used to test for an association between two variables. According to the procedure, the null hypothesis is: There is no correlation between two variables; the rank correlation coefficient for the entire population is equal to zero. The alternative hypothesis claims that there is a correlation between two variables and the aforementioned rank correlation coefficient is not equal to zero. According to the results obtained, we can conclude that there is significant correlation between the data we obtained using Plaggie and FPDS ($n = 10$, $\alpha = 0.01$, $rs = 0.997$, critical value $z = 0.794$). Here rs is the sample statistic; n is the number of pairs of sample data (see columns 2 and 3, Table 1); α is the significance level, and z is a critical interval. Because the sample statistic exceeds the critical value, we made a conclusion about significant correlation. The same strong correlation is found between the data generated by Plaggie and common fractions of files marked by both systems (columns 2 and 5, Table 1). From this outcome, we can expect the same behavior of two systems when they analyze Java source code, and FPDS is a quite accurate filter for Plaggie.

CONCLUSIONS

We have developed a new fast and reliable plagiarism detection system by combining older Plaggie and FPDS projects. We have showed that the use of FPDS as a filter does

not noticeably reduce the reliability of Plaggie, but provides much better algorithmic performance than naïve file-file comparison techniques. The results were tested on a large enough set of Java source files.

As a main result, we can expect that the proposed system will filer files very fast and quite accurate. It will discard significantly the files that cannot mach criteria of plagiarism. The FPDS part is responsible for this operation. The number of files sent for a detailed investigation to the Plaggie part is small enough.

ACKNOWLEDGMENT

The authors wish to thank Kimmo Fredriksson (University of Joensuu) for reviewing the paper and providing valuable comments.

REFERENCES

[1] Grier, S., "A tool that detects plagiarism in Pascal programs", *ACM SIGCSE Bulletin*, vol. 13(1), 1981, pp. 15-20.

[2] Faidhi, J.A.W., Robinson, S.K., "An empirical approach for detecting program similarity within a university programming environment", *Computers & Education*, vol. 11(1), pp. 11-19.

[3] Verco, K.L., Wise, M.J., "Plagiarism à la mode: a comparison of automated systems for detecting suspected plagiarism", *The Computer Journal*, vol. 39(9), pp. 741-750.

[4] Wise, M.J., "YAP3: improved detection of similarities in computer program and other texts", *Proc. of SIGCSE'96 Technical Symposium*, 1996, pp. 130-134.

[5] Prechelt, L., Malpohl G., Philippsen, M., "Finding plagiarisms among a set of programs with JPlag", *Journal of Universal Computer Science*, vol. 8(11), pp. 1016-1038.

[6] Gitchell, D., Tran, N., "Sim: a utility for detecting similarity in computer programs", *Proc. of the 30th SIGCSE Technical Symposium on Computer Science Education*, 1999, pp. 266-270.

[7] Belkhouche, B., Nix, A., Hassell, J., "Plagiarism detection in software designs", *Proc. of the 42nd Annual Southeast Regional Conference*, 2004, pp. 207-211.

[8] Mozgovoy, M., Fredriksson, K., White, D., Joy, M., Sutinen, E., "Fast plagiarism detection system", *Lecture Notes in Computer Science*, vol. 3772, 2005, pp. 267-270.

[9] Joy, M., Luck, M., "Plagiarism in programming assignments", *IEEE Transactions on Education*, vol. 42(2), 1999, pp. 129-133.

[10] Baker, B.S., "On finding duplication and near-duplication in large software systems", *Proc. of the 2nd IEEE Working Conference on Reverse Engineering*, 1995, pp. 86-95.

[11] Ahtiainen, A., Surakka, S., Rahikainen, M., "Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises", *Proc. of the 6th Baltic Sea Conference on Computing Education Research*, Uppsala, Sweden.

[12] Lancaster, T., Culwin, F., "Using freely available tools to produce a partially automated plagiarism detection process", *Proc. of the 21st ASCILITE Conference*, 2004, pp. 520-529.

[13] Mozgovoy, M., "Desktop tools for offline plagiarism detection in computer programs", *Informatics in Education*, vol. 5(1), 2006, pp. 97-112.

[14] Wise, M.J., "Running Rabin-Karp matching and greedy string tiling", *Basser Department of Computer Science Technical Report*, 1994.

[15] Wessa, P., "Free statistics software", *Office for Research Development and Education*, v. 1.1.17, URL http://www.wessa.net

[16] Wiedermeier, P.D., "Preventing plagiarism in computer literacy courses", *Journal of Computing Sciences in Colleges*, vol. 17(4), 2002, pp. 154-163.

[17] Culwin, F., MacLeod, A., Lancaster, T., "Source code plagiarism in UK HE computing schools, issues, attitudes and tools", *South Bank Universiiy Technical Report*, 2001.

Publication [MKS07]

M. Mozgovoy, T. Kakkonen, E. Sutinen. Using Natural Language Parsers in Plagiarism Detection. *Proc. of SLaTE'07 Workshop*, 2007

Using Natural Language Parsers in Plagiarism Detection

Maxim Mozgovoy
mmozgo@cs.joensuu.fi

Tuomo Kakkonen
tkakkone@cs.joensuu.fi

Erkki Sutinen
sutinen@cs.joensuu.fi

University of Joensuu
Finland

Abstract

The problem of plagiarism detection system design is a subject of numerous works of the last decades. Various advanced file-file comparison techniques were developed. However, most existing systems, aimed at natural language texts, do not perform any significant preprocessing of the input documents. So in many cases it is possible to hide the presence of plagiarism by utilizing some simple techniques. In this work we show how a natural language parser can be used to fight against basic plagiarism hiding methods.

Index terms: plagiarism detection, natural language parsing, string matching, natural language processing.

1. Introduction

Plagiarism in universities is an important problem, remaining as a topic for scientific works for years. The studies of plagiarism include the understanding of phenomenon itself, developing methods of plagiarism prevention and techniques of plagiarism detection. The later problem turns out to be a technical task in many cases, since plagiarism detection can be effectively done with the help of computer tools.

A plagiarizer, though, can make some efforts to hide plagiarism. For example, in program code files it is possible to rename variables and to change control structures, modifying the initial lexical structure of the program. Several techniques, including tokenization [1] and parameterized matching [2] were developed to fight with such changes. However, similar methods are harder to apply for natural language texts, so usually it turns out to be easier to hide plagiarism in this case.

In this work we show that NLP tools can be used as a rough equivalent of tokenization for natural language texts, overcoming simple plagiarism hiding techniques. Our previous work [3] was dedicated to the problem of rewording sentences. The current work shows how to fight with "split match" problem.

2. "Split Match" Problem

Arguably, the most popular detection scheme in modern plagiarism detection systems is file-file content comparison by means of general string matching algorithms. Generally, the systems try to find the best joint coverage of the two files, and treat the size of this coverage as their similarity ratio. For example, running Karp-Rabin greedy string tiling (RKR-GST) [4] is used in YAP3 [5], JPlag [6], and Plaggie [7] systems. Similar approach is utilized in Sherlock [1] and FPDS [8].

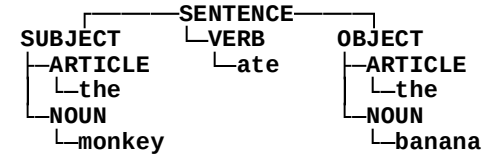
The task of finding optimal joint coverage appears to be NP-complete [5], so heuristic assumptions are used in practical algorithms. For example, nearly all systems use a variation of greedy matching. Another widely used heuristics suggests limiting the minimal length of substrings to be matched. The need of this constraint is caused by the peculiarities of plagiarism detection problem. By including short substrings into the joint coverage, the program provides a lot of false matches that do not indicate plagiarism. Typical copy & paste plagiarism results in duplicating sequences of words, while the presence of the same single words can be an indicator of vocabulary similarity only, not of plagiarism.

Systems like Plaggie [7] and FPDS [8] include such "shortest string length to match" fine-tunable parameters explicitly. Our experiments show that the reasonable size of this parameter is about 10-20 (tokens) in case of program code, and about 4-6 (words) in case of natural language texts.

Unfortunately, the use of this heuristics can cause plagiarism mismatching if a plagiarizer swaps words (this is easy to do in natural language texts, especially in languages such as German, Finnish and Russian that allow freer word order than English). For example, suppose that the original document contains a phrase "light bright sun". A plagiarizer rewords it to "bright light sun". If a system is tuned to match substrings of length 2 (words) and longer only, the plagiarized phrase will not be matched. Keeping in mind the fact that the usual value of "shortest string length to match" constant can be 5-6 words, the intentional word swapping can noticeably affect the detection results.

3. Text Parsing as a Solution

Like computer programs, natural language sentences have syntactic and semantic structure. There are software tools available that can be used to build parse trees for individual sentences. Most automatic English parsers use Chomsky-styled Penn Treebank grammars [9], based on the traditional linguistic approach to the syntax analysis, producing phrase structure-styled analyses. For example, the phrase *the monkey ate the banana* will be represented as



Natural language parsers can recognize noun phrases, homogeneous parts of the sentence, etc. It is clear that word swapping can occur, in particular, in sentences with conjunctions,

such as “and”, “or”, “but”, etc. For example, the phrase “I ate the pizza, the pasta and the donuts” can be reworded as “I ate the pasta, the donuts and the pizza”. Instead of comparing sentences as word strings, we can first analyze them by a parser that recognizes the syntactic structure. These syntactically tagged structures normalize differences between sentences with the same proposition expressed with different word order, thus revealing potential plagiarism.

In contrast to most other parsers based on *probabilistic context-free grammars* (PCFGs), *Stanford Parser* is based on an unlexicalized model [10]. We used version 1.5.1 (30 May 2006) of the system in our experiments. This parser uses a *Cocke-Younger-Kasami* (CYK) [11, 12] search algorithm and can output both dependency and phrase structure analyses [13]. Klein and Manning [10] reported labeled precision and recall figures of 86.9 and 85.7 respectively for this parser. The authors claim that the parser is able to analyze all the sentences in section 23 of the Penn Treebank [9] in a machine with 1GB of memory.

We ran the experiments on the English PCFG grammar and used the dependency output consisting of 48 dependency types. An post-processor tool was implemented in Java that transforms the outputs of Stanford Parser from dependency trees into a format in which the word order has no effect. The format represents the words in the sentence sorted according to their *grammatical relations* (GR) that designate the type of the dependency between the words. The words inside each GR group are sorted in alphabetical order. Figure 1 gives an example of the original Stanford Parser output and transformed format.

```
nsubj(ate-2, I-1)
det(pizza-4, the-3)
dobj(ate-2, pizza-4)
det(pasta-7, the-6)
conj(pizza-4, pasta-7)
cc(pizza-4, and-8)
det(donuts-10, the-9)
conj(pizza-4, donuts-10)

[ate, cc[and], conj[donuts, pasta,
pizza], det[the, the, the],
dobj[pizza], nsubj[I]]
```

Figure 1. Stanford Parser (on the top) and out post-processor (on the bottom) outputs for the sentence: “I ate the pizza, the pasta and the donuts.”

4. Technical Issues

The files generated by natural language parser, are ordinary text documents, and can be used as an input for most general-purpose plagiarism detection system. We have used system [8], earlier developed at our university.

The detection is performed in two phases. First, the parser processes input collection file by file, and generates a collection of parsed files. Second, plagiarism detection system checks the parsed files for similarity. Such flexible scheme allows us to experiment with different parsers, tokenizers, and preprocessors, but in current case has one noticeable drawback. The problem is that the parser destroys the initial word order in every sentence of the input text. Therefore, the plagiarism detection system cannot precisely highlight similar blocks of text in original file pairs. There are two obvious ways to overcome this problem: either the system should be programmed to highlight the whole plagiarized sentences instead of word chains, or the parser should generate some metadata about the parsed files, helping to restore the links between words in original and parsed files. The later is preferable, but requires serious modifications of the parser.

5. Evaluation

Reliable evaluation of a plagiarism detection system is a hard task to perform. Many works use quite informal justifications of the approach used [1, 14]. One of the possible scientific methods includes the use of different plagiarism detection systems as “jury” to evaluate the examined system. Though the separate “opinions” of other systems cannot work as reliable indicators of quality of the system being evaluated, the collaborative “voting” determines the subset of plagiarized files more reliably. This method was used to evaluate the system [8].

Unfortunately, this approach is hard to follow with natural language texts, since the only system we know that utilizes natural language parsing is our recent project [3], that is based on the same system [8], paired with a parser of the Russian language. Therefore, the results were analyzed manually.

Our positive experience with [3] encouraged to use short news messages as an input collection. Such a selection is based on the fact that quite often different agencies provide information about the same event. Furthermore, agencies often cite one another, increasing the number of possible duplications (sometimes reworded). We do not expect plagiarism in this case.

For the evaluation of the system, a collection of 128 messages was obtained from the website of BBC NEWS (http://news.bbc.co.uk). Each message falls into one of the following categories: *Business*, *Europe*, *Science/Nature*, and *Technology*. The median size of each message (after removing all formatting) is about 2 KB.

We have also prepared several files with intentional plagiarism, performed using copy & paste with subsequent change of word and phrase order. It should be mentioned that news is hard to plagiarize with such a method, since the reels are laconic, and do not contain enough adjectives or phrases to swap. However, in free-form essays we used there are more possibilities for such “swap-powered” plagiarism.

The system [8] without parser found 11 pairs of messages containing vast quotations from each other, and 3 pairs of messages informing about the same event. The similarity ratios of later pairs are 5%-33%. The similarity ratios of plagiarized free-form essays were estimated as 10%-30%. The inclusion of parser increases similarity ratios for overlapping news messages by 7%-13%. This observation indicates that the new system has an overall tendency to assign higher similarity grades to the same file pairs. As a result, two more file pairs of similarity 5% and 7% appeared in the resulting log, but they were not considered as similar by human graders. In practice, it is possible to get rid of incorrectly matched pairs by raising a similarity threshold for final file pair list. For the plagiarized free-form essays the similarity ratios have increased significantly — to 50%-80%. The results are also noticeably affected by the value of “shortest string length to match” constant. The smaller is constant, the less effect has the use of the parser. Large constant values cause higher

probability to mismatch "swap-powered" plagiarism, detectable by means of parser.

6. Discussion

Our experiments show that the use of natural language parser to find swapped words and phrases can be effective for intentional plagiarism, but usually impractical for casually similar documents. If two files use the same source or cite each other, the probability of finding intentional swaps is low. On the contrary, plagiarizers have strict motivation to hide copy & paste plagiarism, so swaps are much more likely to occur.

Since the existence of swaps is a good indication of plagiarism, one of the possibilities for future research can be comparison of file pair similarity without and with parsing. The high difference in numbers can indicate the presence of intentional word swaps, and, therefore, of plagiarism.

The use of smaller "shortest string length to match" constant can effectively fight against swaps as well, but it also significantly increases the possibility of false matches. Furthermore, larger values of this constant make detection algorithm work faster [8].

References

[1] M.S. Joy, M. Luck, "Plagiarism in Programming Assignments", *IEEE Transactions on Education*, vol. 42(2), 1999, pp. 129-133.

[2] B.S. Baker, "Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance", *SIAM Journal on Computing*, vol. 26(5), 1997, pp. 1343-1362.

[3] M. Mozgovoy, V. Tusov, V. Klyuev, "The Use of Machine Semantic Analysis in Plagiarism Detection", *Proceedings of the 9th International Conference on Humans and Computers*, Japan, 2006, p. 72-77.

[4] M.J. Wise, "Running Karp-Rabin Matching and Greedy String Tiling", *Technical Report #463*, Basser Department of Computer Science, University of Sydney, 1993.

[5] M.J. Wise, "YAP3: Improved Detection of Similarities in Computer Program and Other Texts", *Proceedings of SIGCSE '96*, 1996, pp. 130-134.

[6] L. Prechelt, G. Malpohl, and M. Philippsen, "JPlag: Finding Plagiarisms among a Set of Programs", *Technical report*, Fakultät für Informatik, Universität Karlsruhe, Germany, 2000.

[7] A. Ahtiainen, S. Surakka, M. Rahikainen, "Plaggie: GNU-Licensed Source Code Plagiarism Detection Engine for Java Exercises", *Proceedings of the 6th Baltic Sea Conference on Computing Education Research*, 2006, pp. 141-142.

[8] M. Mozgovoy, K. Fredriksson, D. White, M. Joy, and E. Sutinen, "Fast Plagiarism Detection System", *Lecture Notes in Computer Science*, vol. 3772, 2005, pp. 267-270.

[9] M.P. Marcus, B. Santorini, M.A. Marcinkiewicz, "Bulding a Large Annotated Corpus of English: the Penn Treebank", *Computational Linguistics*, vol. 19, 1993, pp. 313-330.

[10] D. Klein, C. Manning, "Accurate Unlexicalized Parsing", *Proceedings of the 41st Meeting of the Association for Computational Linguistics*, 2003, pp. 423-430.

[11] T. Kasami, "An Efficient Recognition and Syntax-analysis Algorithm for Context-free Languages", *Scientific Report AFCRL-65-758*, Air Force Cambridge Research Lab, Bedford, Massachusetts, USA, 1965.

[12] D. Younger, "Recognition and Parsing of Context-free Languages in Time n^3", *Information and Control*, vol. 10(2), 1967, pp. 189–208.

[13] M-C. de Marneffe, B. MacCartney, C. Manning, "Generating Typed Dependency Parses from Phrase Structure Parses", *Proceedings of the 5th International Conference on Language Resources and Evaluation*, 2006.

[14] B. Belkhouche, A. Nix, J. Hassell, "Plagiarism Detection in Software Designs", *Proceedings of the 42nd Annual Southeast Regional Conference*, 2004, pp. 207-211.

Errata

[Mozgovoy06]	Missing reference: (Vamplew and Dermoudy, 2005) P. Vamplew, J. Dermoudy. An Anti-Plagiarism Editor for Software Development Courses. *Proc. of the 7th Australasian Conference on Computing Education*, vol. 42, 2005, p. 83-90.
[Mozgovoy06]	p. 99, paragraph 1: "progam's" → "program's"
[Mozgovoy06]	p.100, paragraph 3: "sytem" → "system".
[Mozgovoy06]	Sec. 4.4, last paragraph: "the the" → "the".
[Mozgovoy06]	Sec. 4.2. Note that the plagiarism hiding techniques can be used for code optimization. However, if a certain program is an optimized version of another student's program, it is considered as plagiarism.
[FM06]	Sec. 3, paragraph 2: "*m*th bit of d" → "*m*th bit of D"

Dissertations at the Department of Computer Science and Statistics

Rask, Raimo. Automating Estimation of Software Size during the Requirements Specification Phase—Application of Albrecth's Function Point Analysis Within Structured Methods. Joensuun yliopiston luonnontieteellisiä julkaisuja, 28: University of Joensuu. Publications in Sciences, 28. 128 pp. Joensuu, 1992.

Ahonen, Jarmo. Modeling Physical Domains for Knowledge Based Systems. Joensuun yliopiston luonnontieteellisiä julkaisuja, 33: University of Joensuu. Publications in Sciences, 33. 127 pp. Joensuu, 1995.

Kopponen, Marja. CAI in CS. University of Joensuu, Computer Science, Dissertations 1. 97 pp. Joensuu, 1997.

Forsell, Martti. Implementation of Instruction-Level and Thread-Level Parallelism in Computers. University of Joensuu, Computer Science, Dissertations 2. 121 pp. Joensuu, 1997.

Juvaste, Simo. Modeling Parallel Shared Memory Computations. University of Joensuu, Computer Science, Dissertations 3. 190 pp. Joensuu, 1998.

Ageenko, Eugene. Context-based Compression of Binary Images. University of Joensuu, Computer Science, Dissertations 4. 111 pp. Joensuu, 2000.

Tukiainen, Markku. Developing a New Model of Spreadsheet Calculations: A Goals and Plans Approach. University of Joensuu, Computer Science, Dissertations 5. 151 pp. Joensuu, 2001.

Eriksson-Bique, Stephen. An Algebraic Theory of Multidimensional Arrays. University of Joensuu, Computer Science, Dissertations 6. 278 pp. Joensuu, 2002.

Kolesnikov, Alexander. Efficient Algorithms for Vectorization and Polygonal Approximation. University of Joensuu, Computer Science, Dissertations 7. 204 pp. Joensuu, 2003.

Kopylov, Pavel. Processing and Compression of Raster Map Images. University of Joensuu, Computer Science, Dissertations 8. 132 pp. Joensuu, 2004.

Virmajoki, Olli. Pairwise Nearest Neighbor Method Revisited. University of Joensuu, Computer Science, Dissertations 9. 164 pp. Joensuu, 2004.

Suhonen, Jarkko. A Formative Development Method for Digital Learning Environments in Sparse Learning Communities, University of Joensuu, Computer Science, Dissertations 10. 154 pp. Joensuu, 2005.

Xu, Mantao. K-means Based Clustering and Context Quantization, University of Joensuu, Computer Science, Dissertations 11. 162 pp. Joensuu, 2005.

Kinnunen, Tomi. Optimizing Spectral Feature Based Text-Independent Speaker Recognition. University of Joensuu, Computer Science, Dissertations 12. 156 pp. Joensuu, 2005.

Kärkkäinen, Ismo. Methods for Fast and Reliable Clustering. University of Joensuu, Computer Science, Dissertations 13. 108 pp. Joensuu, 2006.

Tedre, Matti. The Development of Computer Science: A Sociocultural Perspective. University of Joensuu, Computer Science, Dissertations 14. 502 pp. Joensuu, 2006.

Akimov, Alexander. Compression of Digital Maps. University of Joensuu, Computer Science, Dissertations 15. 116 pp. Joensuu, 2006.

Vesisenaho, Mikko. Developing University-level Introductory ICT Education in Tanzania: A Context Approach. University of Joensuu, Computer Science, Dissertations 16. 199 pp. Joensuu 2007.

Huang, Haibin. Lossless Audio Coding for MPEG-4. University of Joensuu, Computer Science, Dissertations 17. 86 pp. Joensuu 2007.

Mozgovoy, Maxim. Enhancing Computer-Aided Plagiarism Detection. University of Joensuu, Computer Science, Dissertations 18. 131 pp. Joensuu, 2007.

Julkaisija	Joensuun yliopisto Tietojenkäsittely- ja tilastotieteen laitos
Publisher	University of Joensuu Department of Computer Science and Statistics
Vaihdot	Joensuun yliopiston kirjasto / Vaihdot PL 107, 80101 Joensuu Puh. 013-251 2677, fax 013-251 2691 e-mail: vaihdot@joensuu.fi
Exchanges	Joensuu University Library / Exchanges P.O. Box 107, FI-80101 Joensuu, FINLAND Tel. +358-13-251 2677, fax +358-13-251 2691 e-mail: vaihdot@joensuu.fi
Myynti	Joensuun yliopiston kirjasto / Julkaisujen myynti PL 107, 80101 Joensuu Puh. 013-251 4509, fax 013-251 2691 e-mail: joepub@joensuu.fi
Sales	Joensuu University Library / Sales of Publications P.O. Box 107, FI-80101 Joensuu, FINLAND Tel. +358-13-251 4509, fax +358-13-251 2691 e-mail: joepub@joensuu.fi