Scalable Plagiarism Detection

Asger Juul Brunshøj

Kongens Lyngby 2016

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

We design, implement and evaluate an automatic plagiarism detection system for peergrade.io—a new service that facilitates peer assessment of assignments among students. We present a survey of the state of the art in the field of plagiarism detection. Our design is focused on scalability and speed. We propose the use of a space efficient but probabilistic data structure for the index. In addition, we propose a new, efficient and linear method for direct comparison of two documents. Furthermore, we present an efficient way of computing rolling hashes of overlapping n-grams. We evaluate our system on the PAN-PC-10 corpus and show it to be highly competitive when compared to contestants of the PAN 2010 competition on plagiarism detection. Most notably, we achieve a significantly higher recall on this corpus than any ofter system we are aware of.

Resume (Danish)

Vi designer, implementerer og evaluerer et system til automatisk plagiatdetektion til peergrade.io—en ny service der tillader studerende at bedømme hinandens afleveringer. Vi præsenterer et studie af eksisterende litteratur omhandlende plagiatdetektion. Vores design fokuserer på skalerbarhed og effektivitet. Vi foreslår et indeks der anvender en pladseffektiv, men probabilistisk datastruktur. Vi foreslår også en ny, effektiv og lineær metode til at sammenligne to dokumenter direkte mod hinanden. Derudover præsenterer vi en effektiv måde at beregne rullende hash-værdier for overlappende n-grammer. Vi evaluerer vores system på PAN-PC-10 korpusset og viser at det er yderst konkurrencedygtigt sammenlignet med deltagerne i PANs konkurrence i plagiatdetektion 2010. Vi opnår markant højere *recall* værdi på dette korpus end noget andet system vi er bekendt med.

Preface

This master's thesis is original, unpublished work by the author, Asger Juul Brunshøj from January 2016 to June 2016 at DTU Compute, Department of Applied Mathematics and Computer Science. Associate professors Philip Bille, Inge Li Gørtz and Ph.d student David Kofoed Wind supervises the thesis. The project is carried out in collaboration with peergrade.io. The thesis has an assigned workload of 30 ECTS credits.

Thank you to friends, family and advisors who have all shown continued interest in this project from beginning to end.

Asger Juul Brunshøj

Kongens Lyngby, June 15, 2016.

Terminology and Symbols

d_{src}	Potential source of plagiarism in d_{susp}.
d_{susp}	The 'suspicious' document being searched for plagiarized passages.
$d_{\text{susp}}^{\text{inv}}$	Inverted file representation of d_{susp}.
$\|d\|$	Number of n-grams in document d.
Fingerprint	Hash of an n-gram.
I_{cand}	Maximum number of candidate source documents returned from source retrieval.
I_{len}	Number of hash buckets in the index.
I_{min}	Minimum number of n-gram matches for a document to be considered a candidate source document.
I_{refs}	Maximum number of document references in the index per n-gram.
n-gram	n consecutive tokens.
P_{min}	Minimum length (in matched tokens) of a passage for that passage to be included in the report.
sid	Short ID (an integer) representing a document in the index.
Token	A unit of text, typically a single word.
δ	Maximum gap size in tokens for passages to be considered adjacent.

Contents

Abstract i

Resume (Danish) iii

Preface v

Terminology and Symbols vii

1 Introduction 1
- 1.1 Definition of plagiarism . . . 1
- 1.2 Plagiarism detection for peergrade.io . . . 2
- 1.3 Aim of the project . . . 3
- 1.4 Classifying forms of plagiarism . . . 3
  - 1.4.1 Focusing on plagiarism with little or no obfuscation . . . 4
- 1.5 Focusing on scalability . . . 4
  - 1.5.1 Scaling with peergrade.io . . . 5
  - 1.5.2 Indexing of external sources . . . 5
- 1.6 Scope of the project . . . 6
- 1.7 Space requirements of representing information as text . . . 6

2 Related Work: The State of the Art 9
- 2.1 System architecture . . . 10
- 2.2 Tokenization . . . 11
  - 2.2.1 n-grams . . . 13
- 2.3 The inverted file . . . 14
  - 2.3.1 Index compression . . . 14
- 2.4 Source retrieval . . . 14
  - 2.4.1 Query by bag-of-words . . . 15
  - 2.4.2 Query by phrases . . . 16
- 2.5 Text alignment . . . 17
- 2.6 Corpora and evaluation frameworks . . . 18
  - 2.6.1 Performance measures . . . 18
  - 2.6.2 The PAN 2010 corpus . . . 19
- 2.7 Summary . . . 20

3 Proposed Plagiarism Detection System **21**
3.1 Source retrieval . . . 21
3.1.1 Tokenization . . . 23
3.1.2 Contents of the index data structure . . . 24
3.1.3 Indexing a document . . . 25
3.1.4 Querying the index . . . 26
3.2 Text alignment . . . 28
3.2.1 Inverted file for d_{susp} . . . 28
3.2.2 Steps of the algorithm . . . 29
3.2.3 Combining passages . . . 31
3.2.4 Examples . . . 31

4 Implementation **35**
4.1 Not yet implemented . . . 37
4.2 Optimizations . . . 37
4.2.1 Rolling hash function for fingerprinting . . . 38
4.2.2 Splitting a string on whitespace . . . 42
4.2.3 Tokenization . . . 44
4.3 Evaluation . . . 45
4.3.1 Effectiveness . . . 46
4.3.2 Comparing to the 2010 PAN competition contestants . . . 47
4.3.3 Efficiency . . . 48
4.4 Deployment . . . 49

5 Discussion **51**
5.1 Limitations . . . 51
5.2 Unclean text extraction . . . 52
5.3 False detections . . . 52
5.4 Further work . . . 52

6 Conclusions **55**

A Text pre-processing example **57**

Bibliography **59**

Chapter 1

Introduction

1.1 Definition of plagiarism

A simple definition of plagiarism is to pass off the work of someone else as one's own without crediting the source. The plagiarized work can be information, intellectual property or ideas. More precisely:

> Plagiarism is the unauthorized use or close imitation of the ideas and language/expression of someone else.[Han01]

The motivation for fighting plagiarism differs for different groups of people. Plagiarism is sometimes fought for monetary reasons. Then, it is fought fiercely and the weapons are copyright, patents and trademarks.

The academic world discourages plagiarism primarily for moral reasons and encourages citation. Plagiarism is a prevalent problem in academia and in educational institutions. A student generally does not commit an act of plagiarism for monetary gain, and the argument has been made that to an extent, students copying the work of their seniors can be seen as part of a learning process. The moral wrong-doing is captured in this quote:

> The wrong in plagiarism lies in misrepresenting that a text originated from the person claiming to be its author when that person knows very well that it was derived from another source, knows that the reader is unlikely to know this, and hopes to benefit from the reader's ignorance[Sam94]

Whether the reason for discouraging plagiarism is monetary or moral, the underlying rationale is that "it is economically fair and morally right to protect original creative investment, both intellectual and economic". This quote is also from Hannabuss (2001)[Han01], which covers the issues of plagiarism and the motivation behind plagiarism in different fields more thoroughly.

1.2 Plagiarism detection for peergrade.io

This project is done in collaboration with peergrade.io[1], a new service that facilitates peer grading of assignments among students. At the time of writing peergrade.io is used at five major universities of Denmark in a subset of their offered courses. Whether an instructor grades the assignments in addition to the peer grading by students depends on the course. When students are peer grading each other's work, cases of plagiarism that would have stood out in the eyes of an instructor may not be detected by fellow students. And if they are detected, they may not be reported. This provides further incentive for an automatic plagiarism detection system to be incorporated into peergrade.io.

In recent years, most existing systems in Denmark used by students to hand in electronic assignments at the university and high school level offer some level of automatic plagiarism detection. At the high school level, Lectio[2] is widely used to submit assignments and offers its own in-house plagiarism detection system, but this system is not designed to index any resources but the assignments that are submitted to Lectio directly[HF12]. At the university level, CampusNet (used by DTU, AU and Aarhus BSS) uses URKUND[3] (a Swedish company) for automatic plagiarism detection. Going beyond the borders of Denmark, there are many commercial systems. Perhaps the most widespread plagiarism detection system in use today is Turnitin[4]. Turnitin claims to be used by more than 10,000 institutions in 135 countries[Tur12]. However, pricing as well as the entailing privacy issues of uploading original work by students to a third-party

[1] https://www.peergrade.io/
[2] https://www.lectio.dk/
[3] http://www.urkund.com/
[4] http://turnitin.com/

service, has motivated peergrade.io to seek the development of a new system tailored to its needs.

1.3 Aim of the project

The goal of this project is to design and implement a stand-alone plagiarism detection service that facilitates the needs of peergrade.io. The focus is on designing a fast and scalable system. The method is to conduct a survey of existing algorithms for plagiarism detection, and to invent new ideas for plagiarism detection based on knowledge gathered from the survey. Finally, the system is implemented and its performance is evaluated.

1.4 Classifying forms of plagiarism

In order to detect plagiarism, it is necessary to understand what common forms of plagiarism may be encountered. Plagiarism can be an exact copy of the source, or it can be similar without being an exact copy, in which case we say the plagiarism is *obfuscated*. The most important of the common forms of (detected) plagiarism is deemed to be:

- No obfuscation, i.e. direct copy and pasting of a document—either in its entirety or parts of it. This is the easiest form to detect.
- Obfuscation by replacing individual words with synonyms.
- Obfuscation by rotation, such as changing "Messi is the top scorer today" to "Today, the top scorer is Messi".
- Paraphrasing (reformulating an idea using one's own words).
- Merging, in which the plagiarized text is merged with original text or other plagiarized text.
- Summarizing the ideas of a larger body of text to a smaller number of paragraphs or sentences.
- Translation from one language into another.

There are some subtle issues in determining when text re-use should or should not be considered plagiarism. For example, to what extent can it be considered plagiarism when an author re-uses text from her own body of previous publishings? Such subtleties of defining what constitutes plagiarism are indeed interesting; see Samuelson, Pamela 1994[Sam94] for a discussion of this particular subtlety of *self-plagiarism*.

Even if plagiarism can be perfectly defined, another branch of subtleties arise when attempting to judge whether something that looks like plagiarism actually is plagiarism, or is merely similar by chance. Separating the two is not the goal of an automatic plagiarism detection system. For now, all we can ask of a plagiarism detection system is to identify likely cases of plagiarism and present the evidence to a human. The human must act as judge in each individual case.

1.4.1 Focusing on plagiarism with little or no obfuscation

In a survey by Turnitin[Tur12], 879 secondary and higher education instructors were asked to identify the forms of plagiarism that occurred most frequently, and the forms that disconcerted them the most. The survey uses a slightly different list than the one presented above and notably, they have not considered translation or summarizing at all. However, their results show quite decisively that "no obfuscation at all" or "close to no obfuscation" is the most frequently detected forms and also the most disconcerting forms of plagiarism. However, it must be said that the most frequently detected forms of plagiarism may be biased toward forms that are easier to detect.

Casual conversations with instructors at DTU hint that the cases of detected plagiarism that lead to actual consequences in terms of disciplining the student, are those that leave little doubt that the act of plagiarism is intentional. That is, those cases with close to no obfuscation or no obfuscation at all. Furthermore, a plagiarism detection system should be conservative in its detections, since false positives result in a waste of the user's time. The design of a new plagiarism detection system is therefore motivated to primarily focus on forms of plagiarism with little obfuscation.

1.5 Focusing on scalability

The focus is on scalability in the design of a new plagiarism detection system for peergrade.io. The motivation for focusing on scalability is twofold: First,

the system should be fast in order to facilitate real-time reports of plagiarism for a given document. Second, but equally important, it should scale to allow indexing of a great number of external resources.

1.5.1 Scaling with peergrade.io

The system needs to be fast enough to offer detection and reporting of detected plagiarism for assignments as they are handed in in real-time. This provides the best user experience in an on-line system, but note that not all plagiarism detection systems are designed for speed; some applications do not require the system to run in real-time. Peergrade.io is expected to grow, and a plagiarism report on the same assignment may be requested several times after the initial hand-in.

Caching the plagiarism reports is generally not considered favorable. Consider the case of two students in the same course handing in the same assignment. It could be that both students have plagiarized from the same external source, or one from the other. The first assignment may appear original to the plagiarism detection system if its content is not indexed. The second assignment will be flagged as a possible case of plagiarism. In this case, the first assignment should also be flagged as potentially plagiarized. This requires that it is inspected for plagiarism not only at the moment it is handed in, but at some time after the second assignment has been submitted.

If notifications of detected plagiarism are to be automatically sent to the instructor, then a good option may be to schedule the plagiarism detection at the end of the hand-in period for an assignment.

1.5.2 Indexing of external sources

The Internet has become an excellent and easily accessible source of information in recent years and as a consequence, the plagiarized source will in many cases be information found on the Internet. This motivates the indexing of selected resources from the world wide web such as Wikipedia[5]. In addition to traditional physical resources such as text books, this means that there is a massive amount of free information that would be beneficial to index, although only indexing a small part of it is feasible.

[5] https://www.wikipedia.org/

The amount of resources that can be indexed directly affects the effectiveness of a plagiarism detection system, so the system needs to scale to accommodate a large index.

1.6 Scope of the project

The plagiarism detection system is only able to recognize cases of plagiarism when the source text is already indexed by the system. There is a discipline known as *intrinsic plagiarism detection* that attempts to detect plagiarism without comparing to any external sources, but that is out of scope for this project.

Based on the surveys mentioned above, the system is designed to work well on forms of plagiarism that have little or no obfuscation, with a focus on natural text; plagiarism detection for other content such as source code is a different problem that requires algorithmic designs targeted for that specific content in order to be as effective as possible[C+03]. In particular, no effort is made to detect plagiarism obfuscated by translation. For research into cross-lingual plagiarism detection, see Potthast et al. 2011[PBCSR11].

The performance and scalability of the proposed system is evaluated on large, publicly available corpora. The performance is not evaluated directly on the assignments already submitted to peergrade.io. At the time of writing, peergrade.io is working on integrating the system into their service.

A survey of the field is presented in Chapter 2. Chapter 3 presents the proposed design. The implementation, optimization and evaluation of the system is the content of Chapter 4. Chapter 5 discusses further work and potential problems to aware of when the system is used in a live setting.

1.7 Space requirements of representing information as text

Before analyzing specific approaches to plagiarism detection, it is useful to have a preliminary understanding of how much space is required to store information as text. Five examples are given in the following, spanning five orders of magnitude: bytes, kilobytes, megabytes, gigabytes and terabytes.

In ASCII encoded text, a single character in the English language can be rep-

resented in one byte. The emerging standard of encoding Unicode characters is UTF8, a variable length encoding in which a single character in the English language can still be represented in one byte, whereas international characters require two, three or sometimes four bytes, depending on how common that character is.

The source file for this thesis consumes 123 KiB.

At the time of writing, around 6300 assignments have been submitted to peergrade.io. The assignments total an estimated 15 MiB of uncompressed text—a relatively small amount.

The English-language Wikipedia comprises 49 GiB of uncompressed text[6]. This is where the necessity of designing for scalability shows itself, since a plagiarism detection system will be faster if the index fits in memory.

Going further, 1 billion web pages were crawled in 2009 known as the ClueWeb09 Dataset[7], amounting to about 25 TiB of uncompressed text.

It must be noted that natural language can be compressed quite effectively to take up less space. For example, the ClueWeb09 Dataset compresses to about 5 TiB—one fifth of its uncompressed size. Specific compression schemes for natural language are covered in Brisaboa et al. 2007[BFNP07]. These compression schemes are designed to allow important operations on the compressed text. They can allow efficient random access and interestingly they can allow searching in the compressed text many times faster than in the original text. Compression makes it more feasible to store large amounts of text. However, utilizing compressed text for the purpose of plagiarism detection directly (if at all viable) is out of scope for this project.

[6] https://en.wikipedia.org/wiki/Wikipedia:Database_download#English-language_Wikipedia

[7] http://lemurproject.org/clueweb09/

Chapter 2

Related Work: The State of the Art

This chapter presents a broad survey of the state of the art in plagiarism detection systems. Automatic plagiarism detection is an active field. It has ties and parallels to such fields as bioinformatics (the problem of gene sequence alignment in DNA), information retrieval (the problem of designing search engines) and code de-duplication (refactoring duplicate code into methods, functions and modules).

The PAN conference is one example of a body that fosters innovations in the field of automatic plagiarism detection. PAN is a series of scientific events and shared tasks on digital text forensics[1]. The conference has offered yearly competitions for automatic plagiarism detection systems. Plagiarism detection was introduced as a subtask at the conference in 2009. Variations of the task has been offered in every year since.

[1] http://pan.webis.de/

2.1 System architecture

The majority of modern plagiarism detection systems are split into two components, *source retrieval* and *text alignment*. This terminology is in common use today, however it is not the only terminology found in the literature. Source retrieval is sometimes called *candidate retrieval*. Text alignment is sometimes called *boundary detection* or simply *post processing*.

Before going further, a bit of terminology is introduced: A *suspicious document*, d_{susp}, is a document under investigation for plagiarism. The terminology is not meant to indicate any strong suspicion that the document contains re-used text. It is simply a document that is undergoing investigation by the plagiarism detection system. Any document that the suspicious document is directly compared to is called a *source document*, d_{src}.

Source retrieval identifies a set of *candidate source documents*—the documents most likely to contain text that has been re-used in the suspicious document. Text alignment compares each candidate source document to the suspicious document more carefully.

Source retrieval Given a suspicious document d_{susp}, and a large collection of source documents, the problem of *source retrieval* is to efficiently find a set of candidate source documents. In order for the system to scale, the time complexity of this task should not depend on the size of the document collection. Source retrieval inherits its name from the closely related field of information retrieval.

Text alignment Given a suspicious document d_{susp} and a source document d_{src}, the problem of *text alignment* is to identify contiguous passages of text between the two. The name text alignment comes from the task of gene sequence alignment in bioinformatics, where the goal is to identify regions of similarity in DNA. Classic techniques for sequence alignment include dynamic programming, which has been applied to the text alignment task as well.

The PAN competition was initially organized as one task on "plagiarism detection", not two separate tasks on "source retrieval" and "text alignment". Before the task was split, most systems did not consider the source retrieval part of the problem at all[HPS15]. Instead they resorted to such practices as comparing every document in the collection practically word for word to every other document in the collection. Which is only feasible on very small corpora. In

2012 this sparked PAN to split up their "plagiarism detection" task into "source retrieval" and "text alignment".

Alternative approaches are sometimes suggested, including graph-based systems, in which sentences are represented as nodes, and close proximity or similarity between sentences are represented with edges[OSB10].

The PAN task focusing on source retrieval directly has a slightly different definition than the one stated above. It does not encourage the construction of a local index. Rather, a joint API is provided by PAN to query external search engines Indri[SMTC05] and ChatNoir[PHS+12]. Both index the ClueWeb09. Source retrieval as defined by PAN, is the task of identifying candidate source documents with as few queries as possible[HPS15]. The rationale behind basing the task on the efficient formulation of queries to external search engines, is that the web today is too large to mine and index by any small operation. Furthermore, the major search companies charge a significant amount of money per query to their APIs. Thus, a cost-effective strategy in this context is to retrieve candidate source documents with as few and as effective queries as possible. This has spurred interest in techniques such as *adaptive querying*, in which queries are formulated based on the results of previous queries.

Since peergrade.io is not only looking to detect plagiarism from the web, there is a need for a local index to detect plagiarism from the body of assignments that has been submitted to peergrade.io. Querying a local index is cheap in comparison, which means that techniques that utilize many more queries can be used. Since a local index must be built either way, the focus here is on designing an efficient system with a local index. The design focuses on scalability, such that some parts of the web or other external resources can be included in it. Querying external search engines as part of the source retrieval task is out of scope for this project, but is not necessarily a bad idea.

2.2 Tokenization

The goal of *Tokenization* is to represent the extracted text-content from a document as a sequence of units or *tokens*. A token can be a character, or a word. When a plagiarist re-uses text and merges it into her own work, modifications are typically made to punctuation, sentence boundaries or the words themselves, in order to fit the style. As text is split into tokens, there are a number of modifications to the text that can help to detect obfuscated plagiarism, listed below. The purpose is to undo slight modifications made by the plagiarist, such that two passages of text that convey the same idea are detected as duplicates, even

when the verbatim texts are not exactly identical.

- *Casefolding:* One of the most common tricks applied by nearly everyone is to convert the text to lowercase. *Casefolding* goes a step further by converting some particular characters such as ß to ss, etc.

- *Removing punctuation characters:* Most systems remove all *punctuation characters*, i.e. non-alphanumeric characters.

- *Correcting spelling:* Geographical differences in spelling such as 'colour' and 'color' can be mapped to one or the other. With the use of a dictionary, even common misspellings can be corrected.

- *Stemming:* Reducing words to their base form is called word *stemming*. For example, a stemming algorithm would reduce the word 'running' to 'run'. In information retrieval, word stemming can be an effective way to normalize queries. The technique has therefore also been adopted in the field of plagiarism detection. However, [HZ03] concludes through experiments that stemming is a disadvantageous idea for plagiarism detection.

- *Stop-words:* Stop-words are contextual words such as 'the', 'and', 'of', 'is', etc. They can be defined in a number of ways. One way is to define stop-words as a fixed set of words for every language. If the language of a document is not known, then another simple, but less accurate way is to define the stop-words as the N most frequently occurring words in the document.

  Their place in information retrieval and plagiarism detection systems is interesting, and slightly controversial. A common practice in information retrieval, is to remove *stop-words* from the text[Sta11]. This practice is also in prevalent use in plagiarism detection systems, see for example Potthast et al. 2014[PHB+14] or Gustafson et al. 2008[GPN08].

  Ceska and Fox 2009 has experimented with many forms of text pre-processing for the purpose of detecting plagiarism[CF09]. Interestingly, they conclude that the removal of stop-words in plagiarism detection systems *hurts* the effectiveness of the system. Surprisingly, the same conclusion can be found in the field of information retrieval[ZM06]. These are controversial findings, considering that the practice of removing stop-words is in wide use.

  Building on these conclusions, Stamatatos 2011 proposed a plagiarism detection system where anything *but* stop-words are removed[Sta11]. This approach is shown to be both effective and efficient. Furthermore, it is well-suited to detect such forms of obfuscated plagiarism where the plagiarist has replaced words with synonyms. A solution based only on stop-words significantly reduces the total number of words in the text, which can help to reduce the size of the index.

- *Number replacement:* This idea was found in Ceska and Fox 2009[CF09]. The idea is that when text is plagiarized, certain patterns of characters may be more frequently altered than others. For example, if the plagiarized text is an economic analysis of a company, then number figures might be adapted to the new context. Number replacement transforms all numbers into a dummy symbol.

2.2.1 n-grams

An *n-gram* is a fixed-length sequence of tokens. For example, consider the text "Alice was beginning to get very tired of sitting by (...)" from the beginning of Alice's Adventures in Wonderland by Lewis Carroll. When each character represents a token, then a 5-gram (an n-gram with $n = 5$) could be any five-long sequence of consecutive characters in the token sequence, such as 'Alice', 'lice ', 'ice w' and so on. When the tokens are split on word boundaries, a 5-gram could be 'Alice was beginning to get', 'was beginning to get very', and so on. The n-grams overlap to form a *sequence of n-grams*.

Most articles generally choose an n in the range of 3 to 7, while some goes higher. Some results show that short n-grams down to $n = 2$ or $n = 3$ is best[BCR09], while others settle higher; the stop-word based approach by Stamatatos uses 11-grams[Sta11]. The best choice generally depends on what choices are made in the other parts of the design, and the corpus the system is tested on. The choice of n is perhaps a parameter that should be tuned to the application at hand.

Modifications are sometimes made after the n-gram has been formed from consecutive tokens. One idea is to sort the tokens that make up the n-gram. The purpose is to make the system more resistant to obfuscation by rotation.

The proceedings from the PAN competitions highlight some other variations of n-grams that have been employed, such as skip-word n-grams[RR14]. Skip-word n-grams create additional n-grams by skipping words, such that for example the 4-gram "plagiarism detection is fun" produced three additional n-grams: "plagiarism detection is", "plagiarism detection fun", "plagiarism is fun", "detection is fun".

A *fingerprint* is a hash of an n-gram.

An example is given in Appendix A, tying the concepts and terminology of this section together.

2.3 The inverted file

An *inverted file* is a data structure that uses hashing to *invert* a text. Tokens or n-grams from the text are hashed and inserted in the inverted file, using the hash (fingerprint) as an index. This allows us to determine in expected constant time whether a given token or n-gram occurs in the text. Each entry in the hash-table is made to store certain satellite data, which can be a list of references to occurrences of the token in the text, or it can be statistics such as the frequency of the token in the text. What is stored as satellite data depends on the use of the inverted file in the overall system. Zobel et al. 2006 provide an in-depth tutorial on the use of the inverted file in information retrieval[ZM06]. The inverted file is a cornerstone in both the source retrieval and text alignment tasks.

If the inverted file is kept in memory, it is usually implemented with a hash-map. However, if it is stored on disk, it may use a B-tree or B^{ϵ}-tree.

2.3.1 Index compression

When a list of document references are stored in the source retrieval index for a given n-gram for example, then each document is represented by an integer. The list can be compressed with integer coding techniques. Zobel et al. 2006 describe various coding techniques including unary, gamma, delta, Golomb and Rice codes, and in what setting each should be preferred[ZM06]. The conclusion is that the advantages of a compressed index far outweigh any disadvantages associated with decompression.

2.4 Source retrieval

Source retrieval makes use of the inverted file to construct an index. However, whereas an inverted file for a single document stores references to the position of tokens within that document, in source retrieval it is used on a document level. All documents in the document collection are indexed in the same inverted file and the inverted file stores references to documents. That is, now the references are to positions in the document collection, rather than positions within a single document. Recall that the purpose of source retrieval is to just return a set of candidate source documents for closer inspection.

The general idea is to define indicators of similarity between two documents, and to use an index to query for documents similar to the suspicious document. The literature reveals two general approaches to querying the index: *bag-of-words* queries and *phrase* queries. The latter can be thought of as querying by n-grams. The bag-of-words approach compares similarity by token frequency, typically on the document level, whereas phrase queries consider documents to be similar if they contain many of the same phrases or n-grams.

2.4.1 Query by bag-of-words

Constructing an index for bag-of-words queries is the universal way to construct the index in information retrieval, used by all current search engines[ZM06]. The bag-of-words method is based on simple statistics of token frequencies (the number of occurrences of a token). The index is a hash-map which stores an entry for every token t found in the entire document collection. The index essentially becomes a dictionary over the English language, but with references back to the documents that contain the tokens.

Each entry stores the frequency of t in the entire document collection (i.e. the number of documents in the collection that contain t). It also stores the set of all documents containing t and the token frequency of t for each of those documents.

To query the index, start by initializing an array A of *similarity scores*. A is initialized with zeros and its length is the number of documents in the collection, such that every document has an entry in A. Then for every token t in the query, the index is used to retrieve a set of documents containing t as well as the frequencies of t in those documents. The corresponding entries in A are incremented by an amount reflecting the similarity to the suspicious document based on the token frequencies. The measure of similarity is determined from a mixture of simple statistics:

- The frequency of the token in the query.
- The frequency of the token in the document.
- The number of documents containing the token.
- The total number of occurrences of the token in the document collection.
- The number of documents in the collection.
- The number of indexed tokens in the collection.

In information retrieval the query is usually no longer than a few words, but in plagiarism detection, the query consists of every token in the suspicious document.

Once all tokens have been looked up from the index, return the documents corresponding to the entries in A with the greatest similarity scores.

2.4.2 Query by phrases

The bag-of-words approach works wonders for retrieving documents that are similar in content. However, phrase querying plays an important role in plagiarism detection, since it has to detect a higher level of similarity on a more local level, in order to detect passages of copied or paraphrased content.

In order to index n-grams, the naive idea is to simply index every n-gram in the document collection, just like every token is indexed in the bag-of-words approach. This is not viable however, as it causes the index to explode in size since there are many more unique n-grams than tokens; with overlapping n-grams, the index would be several times larger than the document collection that is being indexed.

One approach often used in information retrieval for search engines that feature phrase querying, is to use the bag-of-words approach, and apply a filter based on a boolean intersection algorithm on top of it. Only documents that feature all or close to all tokens in the query are allowed through the filter, and only their similarity scores are incremented.

Another viable approach known as *fingerprinting* is to only index a subset of the n-grams. There are a number of different *selection strategies* for choosing which n-grams to index.

- *Full fingerprinting:* Every n-gram is used for indexing. That is, all hashes of overlapping n-grams are used as fingerprints.
- *Random selection:* The fingerprints are filtered, allowing only those fingerprints satisfying some set condition through the filter. For example, only those fingerprints where the hash value modulus some number k equals zero, are allowed through the filter[Man94]. Then k is chosen based on the desired level of resolution.
- *Structure based selection:* n-grams are not allowed to cross structural boundaries in the text. An example could be sentence boundaries, which has been

shown to work well[GPN08].

- *Winnowing:* The sequence of n-grams is split in chunks or *windows* of some fixed length. All the n-grams within a chunk produces a fingerprint, and the fingerprint with the lowest value is chosen to represent the chunk[SWA03]. The rest of the fingerprints within that chunk are discarded. The length of the chunks can be chosen to achieve the desired resolution.

Full fingerprinting in comparison to one of the other selection strategies represents an inherent trade-off between effectiveness and efficiency. The most effective system uses full fingerprinting, but at the cost of increased running time and index space inefficiency. Fingerprinting and selection strategies are covered in more depth in Hoad and Zobel 2003[HZ03].

2.5 Text alignment

Given a suspicious document d_{susp} and a source document d_{src}, the purpose of text alignment is to identify and output passages of text that are highly similar. Approaches to text alignment are generally inspired by the "seed and extend" paradigm from the gene sequence alignment task in bioinformatics. An index is built over d_{susp} using an inverted file. The inverted file maps n-grams to positions in d_{susp}. Then the n-grams from d_{src} are looked up from the inverted file, in order to create *links* between n-grams in d_{src} and d_{susp}. These links are called *seeds* in some literature, inspired by the "seed and extend" terminology.

Once all the links between the two documents have been gathered, the next task is to *extend* them. The goal of the extension subtask is to combine and stretch the seeds, in order to form *passages* of text. There are three general approaches to this subtask highlighted in the proceedings from the 2014 PAN competition[PHB+14]. The simplest is a *rule-based* approach, in which two seeds are merged to form a passage if they are adjacent in both d_{susp} and d_{src}. The seeds do not have to be absolutely adjacent—adjacency is defined as the distance not exceeding a certain threshold, expressed as a character or token limit. The process can be applied recursively to merge seeds and passages. Beside the rule-based approach, dynamic programming and clustering have been used to form passages from seeds. In comparison to these two other approaches, the rule-based approach is apparently no longer competitive[PHB+14]. However, the rule-based approach has the advantage of simplicity.

The final step is to filter the emerging passages. Passages that are under a certain length threshold are typically discarded. The rest are returned as the

detected (possibly) plagiarized passages between the two documents.

Some plagiarism detection systems aim to detect more obfuscated types of plagiarism, such as translation. These systems typically attempt to detect the type of plagiarism at hand in order to adaptively use specialized approaches[PHB+14].

2.6 Corpora and evaluation frameworks

A *corpus* is an annotated collection of source documents and suspicious documents. Passages of text from the source documents have been introduced into the suspicious documents. A corpus includes annotations of which passages of text in the suspicious documents are plagiarized, and which source documents they came from.

Many publicly available corpora exist. However, the problem of generating high quality evaluation corpora has only recently been thoroughly studied. Plagiarism detection systems proposed in the literature are generally not evaluated under the same conditions, making them difficult to compare against each other. Most corpora are rather small, yet a few large ones do exist. Work has been put into the generation of high-quality corpora by Potthast et al. 2010[PSBCR10], motivated by the lack of not only a standardized corpora within the research field, but also the lack of proper performance metrics. The corpora used in the PAN competitions are based on their research.

2.6.1 Performance measures

Without delving into the mathematical details, the performance metrics introduced in Potthast et al. 2010 are *recall*, *precision*, *granularity* and *plagdet*. A perfect plagiarism detection system achieves a score of unity in all four measures. Recall, precision and plagdet are bounded from zero to unity. The granularity is greater than or equal to unity.

Let p denote the body of plagiarized passages that have been introduced in the corpus.

- *Recall:* The ratio of p that are detected by the plagiarism detection system. A recall less than unity means that some plagiarized passages were not detected.

- *Precision:* The ratio of the detected passages that come from p. A precision less than unity means false positives were reported.

- *Granularity:* The granularity expresses whether plagiarized passages are detected in their entirety, as apposed to being detected as multiple separate passages. A granularity greater than unity means that at least one plagiarized passage was falsely detected as two separate plagiarized passages, or overlapping passages were reported.

- *Plagdet:* A combination of the other three measures, in an attempt to provide a single score on which plagiarism detection systems can be directly compared.

The performance measures are in fact introduced in two variations: *macro-averaged* and *micro-averaged* measures. The main difference between the two is that the macro-averaged measures give equal weight to each plagiarism case regardless of its length. Whereas the micro-averaged measures give more weight to longer passages. Both are proposed since it is not obvious which one should be preferred. The evaluation script bundled with the corpus defaults to the *macro-averaged* measures.

2.6.2 The PAN 2010 corpus

The 2010 version of the PAN corpus is still the best corpus available for comparing performance on the forms of plagiarism we are focusing on here. The 2011 version is heavily obfuscated, with all participants achieving very low recall, even with algorithms that take linear time in the size of the corpus[PEBc+11]. In the following years the source retrieval task has been based around queries to search engine APIs, and the corpora have been adapted for this purpose. The PAN competitions have continued to focus on highly obfuscated plagiarism.

The 2010 version contains several gigabytes of text. The material comes from Project Gutenberg[2]. The corpus contains 4 forms of plagiarism[PBcE+10]:

- *None:* Directly copied plagiarized passages make up 40 % of the corpus.

- *Artificial:* These passages were generated by algorithms doing word-shuffling and replacement by synonyms. 20 % of the corpus is made up of artificial plagiarism with "low obfuscation", and another 20 % is made up of "heavy obfuscation".

[2] https://www.gutenberg.org/

- *Simulated:* These passages were generated by crowd-sourcing. Participants were asked to play the role of a plagiarist. These passages make up 6 % of the corpus.
- *Translated:* Translation from one language to another, making up 14 % of the corpus.

The passages vary in length from 50 to 5000 words. The corpus is split into 30 % intrinsic and 70 % external plagiarism passages.

2.7 Summary

The techniques reviewed in this survey offer a great wealth of possible combinations and choices of parameters. Most existing systems are designed from some combination of these techniques with some specific configuration of parameters. Research into effective and efficient plagiarism detection systems is ongoing, and some new and innovative techniques are still surfacing occasionally, such as the purely stop-word based approach by Stamatatos. In the following chapter we will propose some of our own.

Chapter 3

Proposed Plagiarism Detection System

The foundation of the proposed design is based on the survey in Chapter 2. The proposed design is in part composed of techniques from previous work, but also introduces new ideas into the design.

3.1 Source retrieval

The index is designed to support queries on n-grams, and not on bag-of-words, as it is generally believed to be a superior approach to plagiarism detection. Each entry in the index stores a list of document references to documents containing the corresponding n-gram.

We use full fingerprinting, i.e. every consecutive n tokens from the token sequence of a document form an n-gram. In Chapter 2 it was established how this is a trade-off of effectiveness versus efficiency. Full fingerprinting favors effectiveness. The downside of full fingerprinting is that it consumes a great deal of space, but we shall propose a way to mitigate this downside.

In order to motivate this, recall that the bag-of-words approach does not have

as big of an issue with space, despite indexing every token. In the bag-of-words approach, when the index is constructed using a hash-map implementation of an inverted file, the keys are tokens and the values are lists of document references. The bag-of-words approach can afford to index all tokens in the entire document collection, because there is a limited number of tokens (words in the English language). The rate at which new tokens are encountered is rapidly decreasing as more and more documents are indexed. However, indexing all n-grams is not feasible since the number of new unique n-grams that are seen for each new document added to the index does *not* die off. The number of unique n-grams that can be constructed using tokens from the English language is basically exponential in n, where the base of the exponential function is the size of the language (the number of unique tokens). With full fingerprinting as the selection strategy, indexing overlapping n-grams would produce an index several times larger than the document collection, since all the n-grams would be stored as keys.

To overcome this problem, and in order to preserve the ability to use full fingerprinting as our selection strategy, we propose an index based on a hash-map that does not store the keys, but simply stores the values. On the downside, this allows hash collisions of n-grams to go undetected. A normal hash-map addresses the possibility of hash collisions by storing a linked list of key-value pairs in each hash-bucket, or by using linear probing, or something else. So in other words, rather than guard against hash-collisions in one of these ways, we simply implement the hash map such that each hash bucket stores an array of values, and the keys are forgotten.

To query this structure, hash the key (the n-gram) to get the correct hash bucket, and retrieve the array of values stored there. We are unable to tell which of the values actually correspond to the n-gram at hand, and which ones correspond to other n-grams that happened to resolve to the same hash bucket. When using this structure, we will assume that all the values correspond to the n-gram at hand.

The rationale for allowing hash collisions to go undetected is that source retrieval is a numbers game. When candidate source documents are retrieved for a suspicious document d_{susp}, it is simply the documents that have the most matching n-grams that are returned.

Let d' denote a document that is not the source of plagiarized passages in d_{susp}. Then the n-grams in d' are unlikely to have many hash collisions with n-grams from d_{susp}, provided that there are a large number of hash buckets. Whereas an actual source document d_{src} is expected to have a considerable amount of matching n-grams from the plagiarized passage(s).

The expected number of undetected hash collisions between d' and d_{susp} is inversely proportional to the number of hash buckets, i.e. the size of the index.

Let $|d'|$ and $|d_{\text{susp}}|$ denote the number of hash buckets in the index that correspond to n-grams in d' and d_{susp}, respectively. That is, $|d'|$ is roughly speaking the number of unique n-grams in d'. Let I_{len} be the number of hash buckets in the index. Then the number of hash collisions between n-grams of d' and n-grams of d_{susp} follows a hypergeometric probability distribution. The expected number of hash collisions is $|d'| \cdot |d_{\text{susp}}|/I_{\text{len}}$. The risk of false positives in source retrieval is inversely proportional to I_{len}, so the risk naturally diminishes as the system is scaled up.

However, the machine used to implement and develop the system during the project period has limited available memory. As a result, the index did not have enough hash buckets to fully mitigate the risk of hash collisions. There was an observable bias towards returning long documents. This is undesirable for at least two reasons: First, it means that shorter documents that are actual sources of plagiarism might not be included in the set of returned candidate source documents when they should be. Secondly, it creates more work for the text alignment task, slowing down the system.

In order to eliminate this bias for small index sizes, a source document is ranked not by its number of n-gram matches, but by the difference between the number of n-gram matches and the expected number of hash collisions.

In summary, this is a probabilistic data structure, similar to a Bloom Filter in that it allows false positives but there are no false negatives. The advantage is that we are now only using space to store actual references.

3.1.1 Tokenization

The source retrieval module has to support at least two operations:

- Index a source document given its text content as a string.
- Retrieve a set of candidate source documents given the text content of a suspicious document as a string.

Both operations require the string to first be pre-processed to produce a sequence of tokens.

First the string is split on whitespace. Whitespace is any character defined as whitespace by the Unicode standard, including tabs and newline characters.

The sequence of tokens is refined by applying a set of rules. Punctuation is removed and the tokens are converted to lowercase.

Finally, any token is removed if the token's length is zero, or the token consists purely of whitespace (from splitting of consecutive whitespace). The resulting sequence of tokens contains no whitespace, punctuation and no zero-length tokens, and all characters have been converted to lowercase.

In addition, the idea of sorted n-grams is used. Note however that the n-grams are not actually sorted—the fingerprints of the n-grams are computed in such a way that the ordering of the tokens that make up the n-grams does not matter, so the n-grams themselves do not have to be sorted. Computation of the fingerprints is covered in section 4.2.1.

3.1.2 Contents of the index data structure

To construct a new, empty index, initiate an array I of length I_{len}. For each entry in I, initiate and point to a new array of length I_{refs} to hold the document references.

In addition, a table holds the number of n-grams in each of the indexed documents.

To keep the index small it is crucial to use *short identifiers* for the documents that are indexed. Therefore, every document is assigned a unique integer before it is indexed. The first document is identified by the integer 1, the next document by the integer 2 and so on. It is this integer that is stored inside the index to reference the document. So in addition to the main index I, two auxiliary data structures are maintained: Two maps to convert between the integer identifiers and some longer information to identify a document, be it a file location or a primary key to its entry in a database. One map maps one way, the other maps the other way.

Limit document references to I_{refs} per n-gram A new parameter I_{refs} was introduced above. The document references stored at any given hash bucket h could be implemented with a dynamic array. This would allow the list of references to grow as necessary, in order to store a reference to every document

that contains an n-gram that resolves to h. Then I_{refs} is not needed as a parameter. However, we choose to limit the number of document references at any hash bucket to I_{refs}. The purpose of eliminating the influence from common n-grams is to limit the amount of false positives. An additional benefit is that eliminating common n-grams preserves space.

As an example, an analysis of the collective set of documents at arXiv showed that some n-grams are very common. The 7-gram "this work was supported in part by" occurred in 4.55% of all documents[SGWG06]. Matching this specific 7-gram would be a false positive in the majority of cases; it is not a strong indicator of plagiarism.

3.1.3 Indexing a document

Given a document d with the text content s as a string, index the document by the following steps:

1. Assign a short identifier *sid* to d.
2. Pre-process s as described above to produce a sequence of tokens and a fingerprint for every n-gram in the token sequence.
3. For fingerprint f, access the hash bucket h at position f mod I_{len}, to get a list of document references r.
4. Check that a reference to d does not already exist in r.
5. If r is full, so that no reference to d can be stored without breaching the limit I_{refs}, mark h as too common by setting the first value in r to some special value.
6. Otherwise, insert *sid* in r.
7. Repeat from step 3 until all fingerprints have been processed.

Analysis

Pre-processing s takes linear time in the length of s. Inserting a reference for each n-gram takes linear time in the number of n-grams. The number of n-grams $|d|$ is linear in the length of s, so the running time is linear $O(|d|)$.

3.1.4 Querying the index

When retrieving candidate source documents, we will introduce another parameter I_{min}: Any documents with less than I_{min} matching n-grams will not be considered a possible candidate source document.

Additionally the output is limited to I_{cand} candidate source documents. This parameter presents a direct trade-off between effectiveness and efficiency. On one hand, setting I_{cand} too low risks actual source documents not being included in the returned candidate source documents. On the other hand, setting I_{cand} too high creates unnecessary work for text alignment.

Given a suspicious document d_{susp} with the text content s as a string, query for the candidate source documents by the following steps:

1. Get the short identifier *sid* for d_{susp}.
2. Initiate an empty hash-map H to count document references.
3. Pre-process s to produce a sequence of tokens and fingerprints for every n-gram in the token sequence.
4. For fingerprint f, access the hash bucket h at position f mod I_{len}, to get a list of document references r.
5. Check whether h is marked as too common, and if so, skip to the next fingerprint.
6. Otherwise, for each document reference in r, initiate or increment the counter in H.
7. Repeat from step 4 until all fingerprints have been processed.
8. Filter out any references from H that has a count below some threshold I_{min}.
9. Identify the I_{cand} most frequently occurring document references. These are the candidate source documents.
10. Convert the *sid*s for the candidate source documents to the original document identifiers and return them.

Analysis

$|d_{\text{susp}}|$ n-grams have to be looked up in the index. At each hash-bucket, up to I_{refs} document references are found. Each of these references have a counter in H that must be incremented. The running time to populate H is therefore $O(|d_{\text{susp}}| \cdot I_{\text{refs}})$.

The counts in H are corrected for each document by subtracting the number of matches expected to be due to hash collisions. Thereafter, identifying the I_{cand} top scoring document references from H can be done in a number of ways:

- *Sorting:* The naive approach is to sort H and then pick the first I_{cand} elements from the sorted list. This takes $O(|H| \log |H|)$ time.
- *Heapsort:* If the sorting is performed with a heapsort, then extract only the first I_{cand} elements from the heap and stop. This can be thought of as a 'half' sort. In any case, inserting into the heap takes $O(|H| \log |H|)$ time.
- *Min-heap of size I_{cand}:* A min-heap of size I_{cand} storing the largest elements seen at a given moment. This allows every element in H to be compared only to the heap's root element, and if the element is found to be larger than the root, it takes its place in the heap. Using a heap in this way takes only $O(|H| \log I_{\text{cand}})$ time.
- *Stack-allocated array:* Replacing the heap with an array means replacement can be done in constant time since there is no heap invariant to maintain. However finding a new smallest element in the array takes $O(I_{\text{cand}})$. Therefore, this approach takes $O(|H| I_{\text{cand}})$ in total. But it is likely to be faster in practice since the array is small and can easily be stack-allocated, so there is no I/O cost apart from iterating through H.
- *Selection:* A selection algorithm like quickselect would find the I_{cand}'th largest element in expected $O(|H|)$ time. Then another pass through H can pick up the I_{cand} elements that are larger.

The stack-allocated array is considered the best approach in practice, since I_{cand} is small and it wins on I/O cost.

The upper bound on the number of entries in H is $O(|d_{\text{susp}}| \cdot I_{\text{refs}})$. The total running time to query the index is therefore $O(|d_{\text{susp}}| \cdot I_{\text{refs}} \cdot I_{\text{cand}})$. The important point is that it does not depend on the size of the index or the number of documents that are indexed.

3.2 Text alignment

The literature survey of Chapter 2 indicates that text alignment is generally performed by first generating seeds/links between the two documents. Then the seeds are combined by clustering algorithms or heuristic rules to form passages of text believed to be plagiarized.

We propose a method that blurs the lines between these two steps. It is based on heuristic rules, not on clustering or dynamic programming. The text alignment is performed in a single scan of the source document d_{src}, extending seeds into passages on-the-fly. This is proposed for speed purposes, but at the same time, it is a conveniently simple method.

The output from text alignment is a set of similar passages between d_{src} and d_{susp}. A passage consists of: the length of the passage (in number of tokens) and passage boundaries in d_{src} and d_{susp}. The passage boundaries are expressed as byte offsets (referring to the UTF8 strings) to the beginning and end of the passages.

The candidate source documents are matched against the suspicious document d_{susp} independently of each other.

3.2.1 Inverted file for d_{susp}

We utilize the idea of the inverted file to index n-grams, just as in the proposed solution to source retrieval. Every n-gram is indexed by its fingerprint (the full fingerprinting selection strategy). Let $d_{\text{susp}}^{\text{inv}}$ denote the inverted file of d_{susp}. For a given fingerprint, $d_{\text{susp}}^{\text{inv}}$ stores a list of references into the token sequence of d_{susp}, pointing to the n-grams with the given fingerprint.

The inverted file is implemented as a standard hash-map using linear probing, using the fingerprint as the key and the position of the n-gram in the token sequence of d_{susp} as the value. (Since the fingerprint is already a perfectly good 64 bit hash, the hash-map need not compute its own hash. It simply uses a modulo operation on the fingerprint in order to map it to a hash-bucket.)

The inverted file could have used the n-gram as its key rather than its fingerprint. But that is less efficient. As we shall see later in section 4.2.1, the fingerprints are very cheap to compute, and can be computed in such a way that the fingerprint emulates the benefits of sorted n-grams, without actually having to sort the n-grams.

The fingerprints are simply 64 bit integers, but if the actual n-grams were used as keys, every n-gram would have to be constructed as a new string. That would require the inverted file to consume at least n times as much space as d_{susp} itself, in order to store all the overlapping n-grams. While space efficiency is not of major concern in text alignment, using the fingerprints simply has the benefit of being faster with no real downside. When the inverted file is queried and a matching fingerprint is found, the actual n-grams are also compared exactly, in order to rule out the risk of a hash collision on the 64 bit fingerprints.

3.2.2 Steps of the algorithm

The steps are summarized in the following. The first steps are:

1. Pre-process d_{susp} to produce a sequence of n-grams.
2. Invert d_{susp}.

Then, for every candidate source document d_{src} do:

1. Pre-process d_{src} to produce a sequence of tokens.
2. Identify passages by scanning d_{src} while comparing to $d_{\text{susp}}^{\text{inv}}$.
3. Filter passages so that only passages that have more than $P_{\min}$ matching tokens are retained.

$P_{\min}$ is a parameter that must be set. In more detail, identify passages (step 2 above) by:

1. Iterate through n-grams in d_{src} until an n-gram matches in $d_{\text{susp}}^{\text{inv}}$. Assume for the moment that the matching n-gram occurs only once in d_{susp}.
2. Begin a new passage p_i with initial boundaries encompassing the matched n-gram in d_{src} and in d_{susp}.
3. Continue to extend the passage boundaries of p_i for as long as the tokens in d_{src} and d_{susp} are matching. End the passage when there is a mismatch between the next tokens.

4. Compare the boundaries of p_i to the boundaries of the preceding passage p_{i-1}, and combine the two passages into one if the gap between the boundaries within some threshold δ in both d_{src} and d_{susp}. When combining two passages, further attempt to combine the resulting passage with its preceding passage p_{i-2}, and so on.
5. Repeat until the end of d_{src} is reached.

The boundary gap δ is measured in tokens.

To summarize, a passage is born from a seed (a matching n-gram between d_{susp} and d_{src}). It is then extended as far as possible. Finally, it is combined with the preceding passage if they are close to being adjacent.

A strategy is needed for when an n-gram occurs more than once in d_{susp}. $d_{\text{susp}}^{\text{inv}}$ can hold references to all occurrences of the n-gram, but a strategy is needed for how to begin a new passage based on the match. Options include:

- Choose any one match of the n-gram at random.
- Construct and extend passages for all matches of the n-gram, then pick the longest and discard the others.
- Construct and extend passages for one match at a time, stopping if a passage exceeds some threshold length l. If none of the matches result in a passage exceeding length l, then pick the longest and discard the others.
- Pick the match nearest to the preceding passage, in hopes of combining with the preceding passage.

A combination of the third and fourth option was used in an early prototype of the implementation, however the first option is used in the final version. The third or fourth option is still believed to yield better results. The first option was ultimately used because the implementation was ported to another programming language, and time restraints lead to the first option being implemented. Its only analytical advantage is that it yields a strictly linear running time, whereas the other options yield only expected linear running time.

In summary, constructing and using $d_{\text{susp}}^{\text{inv}}$ has expected linear running time in the number of n-grams in the two documents. Extending and combining passages is strictly linear.

In total, this method for text alignment has expected linear $O(|d_{\text{susp}}|)$ running time for constructing $d_{\text{susp}}^{\text{inv}}$, which only has to be done once, and expected linear $O(|d_{\text{src}}|)$ running time for finding passages for each candidate source document.

Parallelization

Note that once d_{susp} has been inverted, the candidate source documents can be processed in parallel. The threads only need read access to $d_{\text{susp}}^{\text{inv}}$ and the token sequence of d_{susp}.

It is a scheduling problem to identify the optimal order in which to process the candidate source documents in order to minimize the total *wall time* or *makespan*. A 4/3-approximation algorithm is to process the candidate source documents in order of decreasing length (called the *longest processing time* rule).

3.2.3 Combining passages

Passages are combined in order to report a plagiarized passage to the user as a whole, rather than as several similar substrings. This is important in order to provide a good user experience. We say we want low *granularity*.

Consider forms of obfuscation where words are missing from the plagiarized passage or extra words have been introduced, or words have been replaced by synonyms. It is evident that identifying such a passage as just one passage can be achieved by simply attempting to combine new passages with the preceding passage.

It is less obvious to what degree plagiarized passages that are obfuscated by rotation can be correctly detected as just one passage. Examples are analyzed in the following to show that it should indeed be fairly effective to just attempt to combine new passages with the preceding passage.

3.2.4 Examples

Imagine a plagiarized passage made from three sentences, where the sentences have been rotated, i.e. they appear in a different order in d_{susp} compared to d_{src}. Consider the 6 ways of obfuscating a plagiarized passage of three sentences by rotating the sentences as shown in Figure 3.1. All of them are reported as one detected passage covering all three sentences, as intended.

(A) The plagiarized passage is an exact copy of the original passage; there is no rotation. What happens is that the n-gram at the beginning of the first sentence is matched in $d_{\text{susp}}^{\text{inv}}$ and a new passage p_1 is created based on that seed.

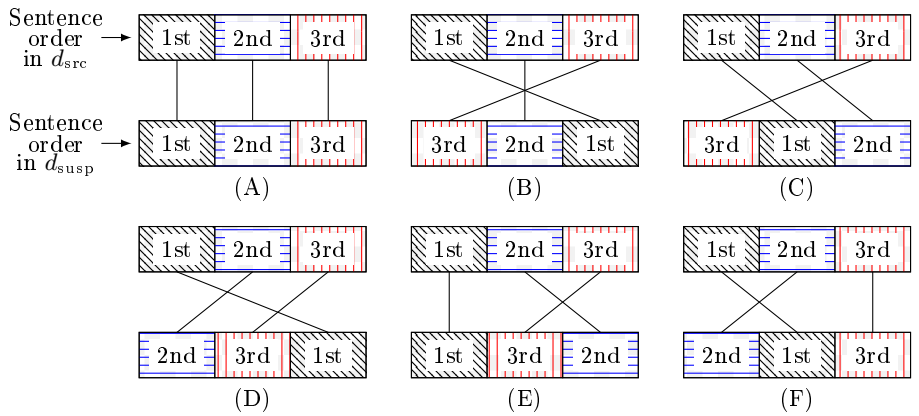

Figure 3.1: Rotations of three sentences that make up a plagiarized passage.

The extension of the seed will extend to cover all three sentences, so all three sentences will be reported as part of one plagiarized passage p_1.

(B) The first detected passage p_1 contains the first sentence. The second detected passage p_2 contains the second sentence. p_1 and p_2 are adjacent, so they are combined into a new passage p_{12}. The third sentence is detected as p_3. p_3 is now also adjacent to the combined passage of the first two sentences p_{12}, in both d_{susp} and d_{src}. Therefore they are combined into one passage p_{123}, covering all three sentences.

(C) The first two sentences are detected as p_1. The third sentence is detected as the second passage p_2, and is then combined with p_1 into p_{12}, covering all three sentences.

(D) p_1 contains the first sentence. p_2 extends to contain the second and third sentence. Then, the two passages are adjacent, so they are combined into one.

(E) Each sentence is initially detected as its own passage, p_1, p_2 and p_3. p_3 is adjacent to p_2, so they are combined into a new passage p_{23}. The newly created passage p_{23} is combined with the preceding passage p_1 since they are adjacent.

(F) The first two sentences are detected as passages p_1 and p_2, and are then combined into p_{12}. Then the third sentence is detected as passage p_3, and is combined with p_{12} to form a single passage p_{123}.

All possible rotations of three sentences are correctly reported as one passage.

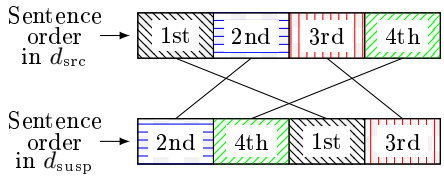

Figure 3.2: Rotation of four sentences that make up a plagiarized passage.

However, there are more complex involving more sentences that are not reported as one passage. Consider the example with four sentences shown in Figure 3.2. Assuming that the length of each sentence is greater than δ, the four passages will not be combined. This example is detected as four separate passages, leading to high granularity. An approach to text alignment based on clustering might be expected to report this example as one passage.

In conclusion, a linear scan such as the one proposed here offers a reasonably competitive alternative to clustering approaches in terms of effectiveness.

Chapter 4

Implementation

The system is primarily written in Python 3. Early prototypes of the proposed system were written entirely in Python. The core algorithms and data structures covered in the preceding chapter were then re-written in Rust to improve performance. Concerned parties will find the code available at Github[1].

Python has severe limitations in terms of performance and concurrency. It has issues with performance because it is dynamically typed, interpreted and because everything is an object. It has issues with concurrency because of its infamous global interpreter lock (GIL).

It was not possible to evaluate the system on the PAN corpus when it was written entirely in Python, because the index refused to fit in RAM. Investigations quickly showed why:

```
>>> import sys
>>> sys.getsizeof(1)
28
>>> sys.getsizeof("")
49
```

[1] https://github.com/peergradeio/plagiarism-backend

An integer object takes up 28 bytes of memory and a string has 49 bytes of overhead.

A multitude of tools are available to alleviate the performance issues of Python. Among these are `PyPy`, `Cython` and `Numpy` but there are many others.

To give examples, `PyPy` is an alternative and possibly more efficient implementation of Python, as opposed to the standard CPython implementation. But it suffers from poor package compatibility and lacks behind when updates and new features are introduced into the language.

`Cython` is a compiler that allows the use of C-extensions such as integer types directly in Python code. However, `Cython` is not compatible with all packages and not even all Python syntax.

Even with a wealth of tools like these at our disposal, writing performant Python code invariably feels like fighting the language. At the end of the day, Python is simply designed to sacrifice performance for simplicity and flexibility.

One can often get by with the many compiled libraries that are available in the Python ecosystem that Python can hook into. But in this project we want full control over the data structures, hash functions and so on that we use, and this is not easily achieved in Python.

It was decided that a better option was to take the plunge and learn a more low level language. Among such options as C, C++, Swift and Go, Rust stood out as a new, modern language that bridges the gap between the high performance of C and C++, and modern language design with concurrency in mind, such as Go and Swift. In short, Rust stands out because it guarantees memory safety even across threads. It does so at compile time and avoids the use of a garbage collector.

The project was put on hold for about two weeks in order to learn the Rust programming language. This has been extremely rewarding, and given the experienced gathered, if the system were to be re-implemented from scratch, the decision would again be to write in Rust.

Section 4.1 describes some shortcomings of the implementation compared to the design presented in Chapter 3. Section 4.2 covers optimizations made to speed up the pre-processing of text into n-grams. Section 4.3 presents evaluation results of the system on the PAN corpus. Section 4.4 discusses the deployment of the system as a service accessible to peergrade.io.

4.1 Not yet implemented

Due to time constraints, some corners are cut in the Rust implementation.

- It was covered in Section 3.2.2 how a random match is chosen when an n-gram occurs in multiple places in the d_{susp}.
- Text alignment has not been parallelized. Plagiarism reports can be generated concurrently, but for now, each report is generated using only one CPU thread.
- In text alignment, when a passage is initiated based on an n-gram match in $d_{\text{susp}}^{\text{inv}}$, that match is not validated. This means that a hash collision on the 64 bit fingerprint could go undetected. This would happen very rarely, and most likely the passage would not be extended, and it would be eliminated because it does not meet the threshold length P_{min}.

These shortcomings are minor, and are not estimated to seriously hinder the effectiveness or efficiency of the system.

4.2 Optimizations

This section delves into some of the optimization done to make some underlying base routines run efficiently. This section is supposed to highlight the iterative and experimental nature of the implementation process. All of the approaches presented in this section use linear running time in the size of the input. But, in terms of shaving a constant factor off the running time, wild improvements are achieved in comparison to the (sometimes naive) first implementations.

When a document is submitted to the system, its text content is extracted to a string s and stored. This is only done once. However, the source retrieval and text alignment tasks require s to be split on whitespace and require tokenization rules to be applied to mold s into a sequence of tokens. Additionally, fingerprints of the n-grams have to be computed.

While text pre-processing of s could be done once and the results stored, that would take up space. Thus, we will attempt to make it as fast as possible in this section. Even with these optimizations, text pre-processing turns out to account for a significant proportion of the computation cost of generating plagiarism reports. Optimization of text pre-processing is worth doing.

This section highlights some of the approaches and iterative improvements that were implemented for three tasks: Splitting s on whitespace, applying the tokenization rules on the split string, and computing fingerprints of all overlapping n-grams.

The test data used to measure performance of the following techniques is based on Alice's Adventures in Wonderland by Lewis Carroll[2]. However, the text is too short to reliably measure performance of some of the techniques, so the text was appended to itself 1000 times, producing a 167.5 MiB text file, encoded in UTF8. The benchmarks are performed on an Intel i5-3570 CPU clocked at 3.40GHz.

4.2.1 Rolling hash function for fingerprinting

The problem statement is the following: Given a sequence of tokens, produce fingerprints (hashes) of all (overlapping) n-grams.

The rest of this section presents iterative improvements in solving this problem. All of these implementations are in Rust, the implementations that were written in Python are not included here. The produced fingerprints are 64 bit unsigned integers in all cases. The results are summarized in Table 4.1.

A queue of tokens and joining tokens before hashing

The first implementation to solve this problem is the following somewhat naive strategy. Iterate through the token sequence, and maintain a queue of the preceding n tokens. In each iteration, the n-gram is formed by reading the previous n tokens from the queue, and joining/concatenating them into a new string. The new token is pushed to the queue and an old token is removed, thus maintaining a window of n tokens. The newly formed n-gram is passed to the hash function to produce a fingerprint. This takes 13.44 seconds to produce fingerprints of all n-grams in the 167.5 MiB text file, using $n = 10$.

In search of an improvement, inspiration is found in the Rabin Karp string searching algorithm. The Rabin Karp algorithm is known for its *rolling hash function*, allowing a hash value to be cheaply computed for a moving window of k characters. The rolling hash in the Rabin Karp algorithm is typically implemented with a polynomial function, though other alternatives are also suggested in the original articles. It is not straightforward to use in our case because our

[2] http://www.gutenberg.org/ebooks/11

Table 4.1: Iterative optimization improvements in computing fingerprints.

Approach	Time [s]	Throughput [MiB/s]
Queue of tokens, join before hash	13.4371	12.4
Queue of hashes, XOR	0.9349	179.2
Queue of hashes, XOR, bitshift	0.9287	180.4
Queue of hashes, XOR, FNV	0.3498	478.9
Queue of hashes, XOR, djb2	0.3243	516.5
Array of hashes, XOR, djb2, modulus	0.0319	5250.8
Array of hashes, XOR, djb2, no modulus	0.0320	5234.4
Lower bound by iteration	0.0320	5234.4

unit is a token, not a fixed-size character. Inspired by the performance of the Rabin Karp rolling hash function, we will propose a way to construct a rolling hash function to produce rolling fingerprints for n-grams, i.e. windows of n tokens. The backbone will still be any traditional hash function that works on strings.

A queue of hash values and bitwise `XOR`

The first improvement is to only hash the present token in each iteration. The queue is then used to store the preceding n hashes, instead of the preceding n tokens. The fingerprint is now defined as a bitwise `XOR` between the hashes of the individual tokens that make up the n-gram. As long as the individual hashes are produced with a good hash function, the bitwise `XOR` between them should also yield a value with suitable randomness. It is no longer necessary to read all elements from the queue in each iteration to form the fingerprint. Instead, the hashing can be done in a rolling fashion by simply updating the fingerprint in each iteration by performing an `XOR` with the token entering the window and an `XOR` with the token leaving the window. This works since applying an `XOR` twice cancels its influence. The running time is brought down to 0.945 seconds with this implementation—an order of magnitude faster.

Preserving token order with bitshifting

Note that we have altered the properties of the resulting fingerprint by defining the fingerprint as a bitwise `XOR` between the hashes of the individual tokens. The fingerprint has the following property that it did not have before: *Any ordering of the tokens that make up the n-gram produces the same fingerprint.* This is

an advantage, because it emulates the effect of sorting the n-grams. Sorting the n-grams achieves exactly this property. However, for the purpose of developing a rolling hash function, we would like to have this property as a feature that can be turned off if necessary.

The solution to turning it off, is to bitshift the hash one place to the left in each iteration. Thus, define fingerprints as the `XOR` between the first token bitshifted $n-1$ places, the second token bitshifted $n-2$ places, etc. until the n'th token. The n'th token in the n-gram does not need bitshifting. The token that is leaving the moving window is bitshifted n places before the `XOR` that nullifies its influence in the rolling hash.

The ordering of the tokens preserves its influence with this one-place bitshifting in each iteration. The bitshifting is cheap, and produces no noticeable difference in the performance of the rolling hash function.

Replacing the hash function

Rust uses SipHash[AB12] as its default hashing algorithm. SipHash is a cryptographic hash function (although the Rust implementation has not yet been cleared for cryptographic purposes). This suggest that a speed-up is imminent if we are willing to forgo the properties of a cryptographic hash function (we are not doing any kind of authentication or storing passwords). Two other hash functions were tried as replacements of SipHash: The FNV hash function (Fowler-Noll-Vo)[3] and the djb2 hash function[4].

The FNV hash function is simple. It consists of a multiplication and an `XOR` per 8 bytes of information. The djb2 hash function is almost as simple as FNV and is even faster. It can be said to use multiplication like FNV and addition, but it implements its multiplication by utilizing a trick with bitshifting, such that the only operations are bitshifting and addition—which are both very cheap. djb2 claims to produce excellent randomness for strings.

There is a slight spread between the results using FNV and djb2. However, replacing SipHash with either of these two reduces the running time down to about a third (from 0.935 to 0.324 seconds). The system is implemented with the djb2 hash function.

[3] http://www.isthe.com/chongo/tech/comp/fnv/
[4] http://www.cse.yorku.ca/~oz/hash.html

Using a stack-allocated array instead of a queue

The preceding approaches use a queue allocated on the heap. But since we know that we are only ever storing n elements in it, and those elements are 64 bit integers, it could easily fit on the stack instead. The queue is replaced by a stack-allocated array A of length n. The array mimics the functionality of the queue. It will hold the previous n token hashes, and the index in the array that is updated with hashes of new tokens changes in a circular manner. That is, in iteration i, the index in the array that is updated is index $i \bmod n$.

Using a stack-allocated array greatly speeds up the program, pushing the running time down to 0.0319 seconds—another order of magnitude.

Avoiding the modulus operator

At this point, producing fingerprints is so fast that we do not risk it being a bottleneck in the system. However, out of academic curiosity, we attempt to push it as far as possible.

The djb2 hash function uses only addition and bitshifting operations, and the computation of the rolling fingerprint uses only bitshifting and `XOR` operations. All are cheap operations. Perhaps the most expensive operation is the modulus operator used to find indices in the array A? We do not actually need the modulus operator. We might as well just maintain a variable that points to the current index in A. At the end of each iteration, increment it, and if it reaches n, reset it to zero. This produces no measurable improvement in performance however.

Lower bound by iteration

Our attempts at improvement have reached stagnation at this point, so it is interesting to see whether there is any further improvements to gain at all. An empty function is implemented, that does not produce fingerprints at all. All it does, is iterate through the tokens (as the other implementations need to do as well). Simply iterating through the tokens takes 0.0320 seconds, which is no improvement. So it appears that the function is entirely I/O bound at this point—at least on the machine used for these tests.

In conclusion, the computation of fingerprints for overlapping n-grams using this proposed rolling hash function is very fast. The time it takes to move tokens

from RAM into cache overshadows the time it takes to actually compute the fingerprints.

4.2.2 Splitting a string on whitespace

At first, splitting strings on whitespace was attempted with regular expressions. It soon became apparent that regular expressions is an inefficient way of solving this task. The regular expressions were abandoned, and the problem was solved by a UTF8 table lookup for each character, in order to separate whitespace characters from non-whitespace characters.

This section provides an overview of the experimental attempts to use regular expressions and table lookups to split a string on whitespace. This section is a bit of a digression from the core plagiarism detection problem. It is not crucial and can be skipped. However, many small subproblems had to be tackled during the implementation phase of the project. This section is included as further illustration of the work-flow and methodology.

The results are summarized in Table 4.2.

Two approaches to writing regular expressions were attempted in Python: Capture groups were used with the regular expression `(\S+)` to return groups of non-whitespace characters. A direct splitting approach was also attempted with `\s+`, instructing the regular expression to split the string at whitespace. Python has at least two regular expression modules, `regex` and `re`. The performance of both are measured. Note that the Python regular expressions are compiled to C code before the measurement is done.

In Rust, the `regex` crate was used (a package in Rust is called a *crate*). In addition to the `captures_iter` (capture group) and `split` methods, the Rust package also allows simply iterating over matches to `\S+` with the `find_iter` method, rather than utilizing the more general concept of capture groups. Importantly, in the Rust regular expression package, `find_iter` and `split` return pointers into the original string (called a *slice*), rather than returning new strings. Additionally, the iterating approach can return the starting position of a match in the original string. When implementing for text alignment, it became apparent that it is not sufficient to simply split on whitespace and return a sequence of tokens. The starting position of each token in the original string must be returned as well. This limits the utility of the regular expression approaches except for the last one based around the `find_iter` method.

The performance measurements show that the regular expressions are somewhat

Table 4.2: Iterative optimization improvements in splitting a string on whitespace.

Approach	Time [s]	Throughput [MiB/s]
Python3 `re` capture groups (`\S+`)	12.617	13.28
Python3 `regex` capture groups (`\S+`)	15.558	10.77
Python3 `re` split `\s+`	9.953	16.83
Python3 `regex` split `\s+`	7.552	22.18
Rust `regex` capture groups (`\S+`)	18.568	9.02
Rust `regex` split `\s+`	2.424	69.10
Rust `regex` find_iter `\S+`	2.570	65.18
Rust table-based `split_whitespace` method	0.950	176.32
Rust table-based `match_indices` method	0.548	305.66
Rust table-based `split` method	0.612	273.69

unpredictable without looking into the underlying algorithms.

Three approaches not involving regular expressions were tried in Rust. There is a built-in method on strings in Rust called `split_whitespace` which separates the string by a UTF8 based table look-up. That is faster than the regular expressions by an order of magnitude. However, it only returns the sequence of tokens, not the positions in the original string that are necessary for text alignment. Motivated by the evident speed improvements gained by abandoning regular expressions, a more general method called `match_indices` is tried out. This method does return the necessary information, but the splitting behavior of `split_whitespace` has to be manually emulated by specifying a boolean filter on whitespace characters. Surprisingly, `match_indices` turns out to outperform the built-in `split_whitespace` method.

There is a third method, `split`. This method returns the same information as `split_whitespace`, except the user specifies the boolean filter to split on. Curiously, when the same boolean filter is used, `split` is also faster than `split_whitespace`, but not as fast as `match_indices`. This begs the question, what extra work is the built-in `split_whitespace` doing since it is slower? The source code of `split_whitespace` was used as a guide when writing our own boolean filter to emulate the behavior of `split_whitespace`. The secret as to why it is slower lies somewhere in the implementation details[5]. While the answer is probably interesting, time restraints forced focus to be diverged to other parts of the project.

[5]Permalink to `split_whitespace` method implementation in Rust: https://dxr.mozilla.org/rust/rev/db2939409db26ab4904372c82492cd3488e4c44e/src/libcollections/str.rs#802

In summary, an approach based on UTF8 table look-ups is way faster than a regular expression, as might have been predicted. The `match_indices` method with the a boolean filter for catching whitespace characters is the fastest method.

4.2.3 Tokenization

The tokenization involves three steps: Converting to lowercase, removing punctuation characters and sorting n-grams.

We are not optimizing the sorting of n-grams. A shortcut using fingerprints covered in Section 4.2.1 means that n-grams are only actually sorted when matches in the inverted file are validated in the text alignment task. The results for the other two problems are shown in Table 4.3.

Removal of punctuation characters

Filtering out punctuation characters was achieved with regular expressions and with an approach based on UTF8 table look-up. The table look-up is only slightly faster than the regular expression in this case.

Converting to lowercase

Python has a function for converting to lowercase as well as a function for case folding. Rust has a method for converting to lowercase, but no built-in method for case folding. Interestingly, the built-in method in Rust is significantly slower than its Python equivalent.

Converting to lower case is a significant bottleneck of the text pre-processing steps with this function. The Python function is not easily used in-place of the Rust one, and clearly if Python can be fast then so can Rust.

This issue was brought to the attention of the Rust developers at the official Mozilla IRC channel. It turns out that the Rust function in the standard library works by some form of binary search on a unicode mapping table.

A new function was put together in collaboration, using a secondary table to fast-track the look-ups if the character is an ASCII character (this is fairly easy

Table 4.3: Performance of different implementations of tokenization

Approach	Time [s]	Throughput [MiB/s]
Lowercase conversion		
Python3 `lower()`	1.059	158.17
Python3 `casefold()`	1.596	104.95
Rust `to_lowercase()`	5.780	28.98
Rust `to_lowercase_custom()`	0.340	492.78
Punctuation filtering		
Rust regex punctuation filter	1.118	149.82
Rust table punctuation filter	0.976	171.62

to implement, since the first byte of a UTF8 character is reserved for ASCII characters).

On the test data used here (Alice's Adventures in Wonderland), the new function is an order of magnitude faster than the built-in one, because while it is encoded in UTF8, it is composed almost entirely of ASCII characters. The same is expected to be true for documents submitted to peergrade.io.

Since the majority of characters in normal text are already lowercase, there may be room for even further improvement if the conversion can be done in-place.

4.3 Evaluation

The system is evaluated using the parameters shown in Table 4.4. The optimal set of parameters are simply guessed at; no training was done in order to tune these parameters.

$n = 5$ was used for the n-grams, inspired by the winning submission in the 2010 PAN competition[KB10].

The system benefits from I_{len} being as large as possible. It received a value of $I_{\text{len}} = 100000007$ since the machine used for testing has limited available memory.

With this choice of I_{len} and I_{refs}, the index is 35% full after indexing the 11148 source documents contained in the test corpus (about 1.8 GiB). The index is expected to consume $100000007 \times 8 \times 16$ bits, if a 16 bit integer is used to

Table 4.4: Parameters used in evaluation tests. Refer to Terminology and Symbols in the beginning of the report for an overview of the parameters with descriptions.

Parameter	I_{len}	I_{refs}	n	I_{cand}	I_{min}	δ	P_{min}
Value	100000007	8	5	8	5	30	10

Table 4.5: Results from evaluation on 2010 corpora.

Corpus	Plagdet	Precision	Recall	Granularity
PAN-PC-10 Train custom	0.915	1.0	1.0	1.132
PAN-PC-10 Test	0.699	1.0	0.970	1.653

store the *sid*. This is about 1.5 GiB. Yet it takes up about 3.5 GiB of RAM in practice. This means that there is a lot of overhead involved, see further work Section 5.4. If the overhead can be reduced, then indexing 1.8 GiB takes up 35% of 1.5 GiB. Thus, a lower bound of the space consumption of the index lies at about 30% of the size of the source documents. This is very reasonable considering that we use full fingerprinting, and implementing integer coding will bring it down further.

After indexing the source documents of the test corpus, 1.41% of the hash buckets are filled up and marked as too common. 7.12% of the hash buckets remain empty.

4.3.1 Effectiveness

The system is evaluated on two corpora from 2010: the test corpus and a custom version of the training corpus, in which we have removed intrinsic plagiarism and passages obfuscated by translation. See the results in Table 4.5.

As expected, our system performs significantly better on the customized training corpus than it does on the test corpus.

Table 4.6: Performance of top 5 out of 18 contestants in the PAN 2010 competition on the PAN-PC-10 Test corpus.

PAN 2010 participants	Plagdet	Precision	Recall	Granularity
J. Kasprzak and M. Brandejs	0.7971	0.94	0.69	1.00
D. Zou et al.	0.7090	0.91	0.63	1.07
M. Muhr et al.	0.6948	0.84	0.71	1.15
C. Grozea and M. Popescu	0.6209	0.91	0.48	1.02
G. Oberreuter et al.	0.6066	0.85	0.48	1.01

4.3.2 Comparing to the 2010 PAN competition contestants

The performances of the top 5 participants in the 2010 competition are shown in Table 4.6. This data is available online[6] and are discussed in more detail in the proceedings from the 2010 PAN conference[PBcE+10].

These evaluation results use the PAN-PC-10 Test corpus, which means that it includes intrinsic plagiarism and obfuscation by translation. We make no attempt to detect these two forms of plagiarism, so we do not expect to be at the top of the list.

Yet the benefits of using full fingerprinting become apparent when we compare our results with the contestants of the 2010 competition. It is remarkable that we achieve a recall of 0.970 whereas the best recall out of the 2010 contestants is 0.71. Presumably, this high score in recall is due in part to our use of full fingerprinting in source retrieval.

On the other hand, our system suffers in granularity, indicating that reported passages are overlapping or single passages are reported as several separate passages. Our precision score is perfect, so in order to achieve a higher plagdet score, we might need to be more liberal in which passages we consider adjacent. In doing so, we will be trading lower granularity for a lower precision. In practice this means increasing the value of δ. If this does not work, then perhaps our linear method of combining passages need to be re-evaluated.

In conclusion, we get third place, among 19 contestants including ourselves. Obviously, improvements have been made to these systems since the 2010 competition took place, and it would be interesting to find common ground to compare against more recent systems.

[6] http://pan.webis.de/clef10/pan10-web/plagiarism-detection.html

Table 4.7: Proportion of time spent doing individual subtasks during the generation of a plagiarism report for a typical suspicious document.

Subtask	Percentage [%]
Source retrieval	
Text pre-processing of d_{susp}	3.0
Index look-ups for n-grams	8.3
Find top *sid*s from H	0.3
Source retrieval total	**11.6**
Text alignment	
Text pre-processing of d_{susp}	2.4
Construct $d_{\text{susp}}^{\text{inv}}$	2.3
8 times text pre-processing of d_{src}	68.2
8 times find passages	15.5
Text alignment total	**88.4**

4.3.3 Efficiency

It takes about 200 seconds to index the 1.8 GiB source documents from the test corpus on an Intel i5-3570 CPU. This is a rate of 9.20 MiB per second.

Plagiarism reports for 15925 suspicious documents (3.4 GiB) are generated in 7255 seconds (2 hours). This is a rate of 0.48 MiB of suspicious documents per second.

Peergrade.io's collection of submitted assignments are estimated at about 6000 documents totaling 15 MiB of text. Thus, in about 30 seconds on a single CPU thread, plagiarism reports can be generated for every assignment ever submitted to peergrade.io.

Table 4.7 shows where the time is spent while generating a plagiarism report for a typical suspicious document. Note that these are rough estimates, based on the generation of a single report. The ideal would be a full profile over an entire corpus. One takeaway is that in-spite of the optimization presented in Section 4.2, text pre-processing accounts for over 70 percent of the time used for generation of this plagiarism report.

4.4 Deployment

The system is deployed as a stand-alone web-service. It presents itself with a web-accessible API for submitting documents (which are then indexed) and for requesting reports of detected plagiarism for a given document. A basic infrastructure has been put together including a database.

The system stands completely independent from peergrade.io. Peergrade.io can be seen as merely one user potentially among many users of the plagiarism detection system. Primitive authentication functionality has been implemented.

Designing, implementing and testing this surrounding infrastructure in addition to the core plagiarism detection system has been a large part of the work put into this project. However, it shall receive only a very cursory overview in this section, as the algorithmic design of the core plagiarism detection modules is the main contribution of this thesis.

The database holds information about the submitted documents, and the extracted text content from the documents. It also holds information about users of the plagiarism detection system. peergrade.io is the first such user, but the system has been designed with a foundation that can handle additional users.

Components

`MongoDB` is used for the database. The web API is written in Python with the `Flask` framework[7]. The core system written in Rust also implements a local HTTP web server. The Python code communicates with the local Rust server through HTTP requests. The Iron web framework for Rust[8] is used.

Text extraction

The system currently supports text extraction from two file formats: plain text files and PDF. Text extraction from PDF uses `pdftotext`[9]. The infrastructure is in place to add more as necessary.

[7] http://flask.pocoo.org/
[8] http://ironframework.io/
[9] http://linux.die.net/man/1/pdftotext

File format detection

Rather than determine the file format from the HTTP header when a new document is submitted, or rely on the filename extension, the MIME type is determined using `libmagic`[10]. This was found to be more reliable.

Hosting

The plagiarism detection system is hosted on an Amazon EC2 instance, using `Apache` as the web server.

Populating the index

It is out of scope for this project to data mine the web for resources to populate the index with. That task is left to peergrade.io.

Other experiences

More experiences were gathered from the deployment process than can be mentioned here. The development involved research into topics and use of tools which are too numerous to discuss here, yet most of them were unfamiliar concepts going in to the project. They include `Git`, SSH, `tmux` and the Unix ownership system; unit testing and monkey-patching; non-blocking concurrency, and to what extend concurrency is possible in Python, in face of Python's GIL.

[10] https://github.com/threatstack/libmagic

Chapter 5

Discussion

This chapter discusses potential problems that may arise in practice, and ideas for further work.

5.1 Limitations

There is at least one easy way that a student who knows the inner workings of the plagiarism detection system can circumvent it. Text cannot be extracted from scanned documents (short of using optical character recognition). Even new and electronic documents can be saved "as an image" to PDF for example. This creates a vectorized or rasterized version of the document similar to a scanned document.

Plagiarized images and figures are not detected.

The system is designed to work on natural language text. It may or may not perform well on code or mathematical formula. However, as long as these are reasonably extracted from the document as text in some form, then they will be treated just like normal text and split into tokens.

5.2 Unclean text extraction

Text extraction from such formats as PDF can be unclean. Headers, footers and page numbers are hard to get rid of. Figure captions may appear in the middle of a paragraph.

These unclean text extractions may hinder the plagiarism detection somewhat. Another adverse effect is that they are ultimately propagated to the plagiarism report that the user sees.

Even when a passage is copied verbatim, some light obfuscation may sneak its way into the passage such as hyphens at line breaks. If the text is first split on whitespace (including newline characters) and punctuation is stripped afterwards, then the word that was split by the line break will end up as two separate tokens. Possible solutions to this problem are to add another level of pre-processing before the string is split, to detect such combinations as a hyphen followed by a newline character. Another approach that could remedy the problem is the use of skip-word n-grams.

5.3 False detections

A major source of false detections may come from quotes and re-used text that is actually correctly cited. Although high-school teachers in some cases think that this can be a feature, as it helps them check whether their students are correctly using citations[HF12].

On a similar note, false detections may come from the list of references, since a reference will be typeset the same way in many documents citing the same source.

Famous quotations are another source of false detections.

5.4 Further work

Parameter tuning The parameters of the system should be tuned. There is a training corpus as well as a test corpus from PAN 2010. The system can be trained on the training corpus with a variety of parameter choices. The best

performing set of parameters can then be used on the test corpus to achieve an honest score.

Reduce index overhead The index consumes about 2.3 times more memory than it is expected to. Presumably this comes from overhead that can be avoided. A `Vec` is the Rust implementation of dynamic list data structure. The index currently uses one `Vec` of length I_{len} and each entry is another `Vec` of length I_{refs}. Presumably these `Vecs` have overhead causing the index to be inflated. This is avoided by simply flattening the structure, using one primitive 1D array and careful indexing.

Reduce granularity The bottleneck in the plagdet score of our system is a high granularity. If this can not be improved with parameter tuning, then the way in which adjacent passages are combined should be re-evaluated.

Promising ideas from the literature survey There are a couple promising ideas presented in Chapter 2, which were not used in the design of our system. Most notably, an interesting idea that may fit well into our system and our goal of scalability, is to decrease the size of the index by not applying a full fingerprinting strategy for source retrieval. Winnowing, or not indexing n-grams that cross sentence boundaries, are the two most attractive alternatives.

Index compression Applying an integer coding scheme to the source retrieval index can be expected to reduce its size significantly, without any significant cost to query time.

Text alignment in parallel The text alignment can and should be done in parallel. There is nothing stopping us from doing so. It simply has not been prioritized because of time constraints.

Proper authentication At the moment, only a simple authentication system is implemented to prevent unauthorized access to the plagiarism detection service API. It consists of a password or *apikey* sent with the request in plain text. Proper authentication should be set up.

Selection of resources to populate the index with To aid in this task, a study by Turnitin[1] examines the internet resources that are sources of plagiarism by higher education students in their assignments. The study is based on more than 28 million student papers submitted to Turnitin between July 2011 and June 2012. They conclude that Wikipedia is the source three times as often as the second contender.

Explode PAN corpus The PAN corpora are put together and annotated in such a way that specific forms of obfuscation can be filtered out. This is what allowed us to create a derivative of the PAN 2010 corpus where passages meant for intrinsic plagiarism detection or obfuscation by translation have been filtered out. In guiding the direction of further development, it might be helpful to explode the corpus into each of its constituent forms of plagiarism, in order to asses how the plagiarism detection system performs on each specific kind of obfuscation. Derivative corpora could be made for each of "no obfuscation", "simmulated plagiarism", "low artificial obfuscation" and "heavy artificial obfuscation".

Storing the index on disk In order to truly scale up the size of the local index, it can be stored on disk instead of in RAM. Large search engines store their index on disk, distributed over many machines. Tests would have to be conducted in order to assess how big of an impact this has on performance. Note that a search engine can more easily maintain fast queries because the query length is often no more than a few words, so only a few look-ups have to be made in the index. In contrast, with full fingerprinting the plagiarism detection system has to look up every n-gram in the entire document.

Compression of extracted text content The extracted text content of documents is currently stored in the database. This can be compressed, provided that a compression scheme is chosen that decompresses fast, so as not to slow down the system. This might not be relevant at the moment as RAM is severely more limited than hard drive space. But if the index is scaled up and stored on disk, then many more documents also have to be stored in the database.

[1] http://go.turnitin.com/paper/highered/sources-student-writing

Chapter 6

Conclusions

For source retrieval, we propose a probabilistic index data structure based on a modified key-value store that does not store the key. The probabilistic index allows the utilization of a full fingerprinting selection strategy, while limiting space usage to about 30% the size of the indexed material, without using integer coding. Our system achieves excellent recall, outperforming all other systems evaluated on the PAN-PC-10 corpus.

For text alignment, we propose a method based on a linear scan of the candidate source documents, combining the two steps of the seed and extend paradigm. We extend seeds into passages and combine passages in a linear fashion during the scan. This method is fast and achieves perfect precision, but achieves poor granularity compared to other systems. It is unclear whether this is due to a poor choice of parameters, or drawbacks in the linear design of combining passages.

For text pre-processing we propose a rolling hash function based on `XOR` and bitshifting. It achieves fast hashing of overlapping n-grams while eliminating the need to explicitly sort n-grams. The rolling hash function is fast and flexible.

In total, the system achieves a plagdet score of 0.699 on the PAN-PC-10 test corpus, securing a position among the best systems evaluated on this particular corpus, proving its competitiveness.

The Rust implementation is fast, concurrent and capable of plagiarism report generation in real-time. Text pre-processing remains the performance bottleneck of the system, in spite of extensive optimizations in that direction.

Appendix A

Text pre-processing example

```
STRING:
"Automatic PLAGIARISM DETECTION, is an interesting problem!"

SPLIT ON WHITESPACE:
"Automatic", "PLAGIARISM", "DETECTION,", "is", "an", "interesting", "problem!"

TOKEN SEQUENCE:
"automatic", "plagiarism", "detection",  "is", "an", "interesting", "problem"

SORTED 4-GRAMS:
"automatic detection is plagiarism"
             "an detection is plagiarism"
                          "an detection interesting is"
                                       "an interesting is problem"

FINGERPRINTS:
d4c378b058512cf...
             afa86de637cbc28...
                          dc77be94100a94b...
                                       5c56166b47af573...
```

Bibliography

[AB12] Jean-Philippe Aumasson and Daniel J Bernstein. Siphash: a fast short-input prf. In *Progress in Cryptology-INDOCRYPT 2012*, pages 489–508. Springer, 2012.

[BCR09] Alberto Barrón-Cedeño and Paolo Rosso. On automatic plagiarism detection based on n-grams comparison. In *Advances in Information Retrieval*, pages 696–700. Springer Berlin Heidelberg, 2009.

[BFNP07] Nieves R. Brisaboa, Antonio Fariña, Gonzalo Navarro, and José R. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10(1):1–33, 2007.

[C+03] Paul Clough et al. Old and new challenges in automatic plagiarism detection. In *National Plagiarism Advisory Service, 2003; http://ir. shef. ac. uk/cloughie/index. html.* Citeseer, 2003.

[CF09] Zdenek Ceska and Chris Fox. The influence of text pre-processing on plagiarism detection. In *Proceedings of the International Conference RANLP-2009*, pages 55–59, Borovets, Bulgaria, September 2009. Association for Computational Linguistics.

[GPN08] Nathaniel Gustafson, Maria Soledad Pera, and Yiu-Kai Ng. Nowhere to hide: Finding plagiarized documents based on sentence similarity. In *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology-Volume 01*, pages 690–696. IEEE Computer Society, 2008.

[Han01] Stuart Hannabuss. Contested texts: issues of plagiarism. *Library Management*, 22(6/7):311–318, 2001.

[HF12] Kristoffer Henry Hansen and Morten Frølich. Efficient plagiarism detection. Master's thesis, DTU, 2012.

[HPS15] Matthias Hagen, Martin Potthast, and Benno Stein. Source retrieval for plagiarism detection from large web corpora: recent approaches. *Working Notes Papers of the CLEF*, pages 1613–0073, 2015.

[HZ03] Timothy C Hoad and Justin Zobel. Methods for identifying versioned and plagiarized documents. *Journal of the American society for information science and technology*, 54(3):203–215, 2003.

[KB10] Jan Kasprzak and Michal Brandejs. Improving the reliability of the plagiarism detection system. 2010.

[Man94] Udi Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 2–2, Berkeley, CA, USA, 1994. USENIX Association.

[OSB10] Ahmed Hamza Osman, Naomie Salim, and Mohammed Salem Binwahlan. Plagiarism detection using graph-based representation. *arXiv preprint arXiv:1004.4449*, 2010.

[PBcE+10] Martin Potthast, Alberto Barrón-cedeño, Andreas Eiselt, Benno Stein, and Paolo Rosso. Overview of the 2nd international competition on plagiarism detection. In *In Proceedings of the SEPLN'10 Workshop on Uncovering Plagiarism, Authorship and Social Software Misuse*, 2010.

[PBCSR11] Martin Potthast, Alberto Barrón-Cedeño, Benno Stein, and Paolo Rosso. Cross-language plagiarism detection. *Language Resources and Evaluation*, 45(1):45–62, 2011.

[PEBc+11] Martin Potthast, Andreas Eiselt, Alberto Barrón-cedeño, Benno Stein, and Paolo Rosso. Overview of the 3rd international competition on plagiarism detection. In *In Working Notes Papers of the CLEF 2011 Evaluation*, 2011.

[PHB+14] Martin Potthast, Matthias Hagen, Anna Beyer, Matthias Busse, Martin Tippmann, Paolo Rosso, and Benno Stein. Overview of the 6th international competition on plagiarism detection. In *CLEF*, 2014.

[PHS+12] Martin Potthast, Matthias Hagen, Benno Stein, Jan Graßegger, Maximilian Michel, Martin Tippmann, and Clement Welsch. Chatnoir: A search engine for the clueweb09 corpus. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '12, pages 1004–1004, New York, NY, USA, 2012. ACM.

[PSBCR10] Martin Potthast, Benno Stein, Alberto Barrón-Cedeño, and Paolo Rosso. An evaluation framework for plagiarism detection. In *Proceedings of the 23rd international conference on computational linguistics: Posters*, pages 997–1005. Association for Computational Linguistics, 2010.

[RR14] Diego Antonio Rodriguez-Torrejon and José Manuel Martín Ramos. Coremo 2.3 plagiarism detector text alignment module - notebook for PAN at CLEF 2014. In *Working Notes for CLEF 2014 Conference, Sheffield, UK, September 15-18, 2014.*, pages 997–1003, 2014.

[Sam94] Pamela Samuelson. Self-plagiarism or fair use. *Communications of the ACM*, 37(8):21–25, 1994.

[SGWG06] Daria Sorokina, Johannes Gehrke, Simeon Warner, and Paul Ginsparg. Plagiarism detection in arxiv. In *Data Mining, 2006. ICDM'06. Sixth International Conference on*, pages 1070–1075. IEEE, 2006.

[SMTC05] Trevor Strohman, Donald Metzler, Howard Turtle, and W. Bruce Croft. Indri: a language-model based search engine for complex queries. Technical report, in Proceedings of the International Conference on Intelligent Analysis, 2005.

[Sta11] Efstathios Stamatatos. Plagiarism detection using stopword n-grams. *J. Am. Soc. Inf. Sci. Technol.*, 62(12):2512–2527, December 2011.

[SWA03] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 76–85, New York, NY, USA, 2003. ACM.

[Tur12] Turnitin. White paper: Defining plagiarism: The plagiarism spectrum. Technical report, http://go.turnitin.com/paper/plagiarism-spectrum, 2012. Visited May 2016.

[ZM06] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), July 2006.