Imperial College London

DEPARTMENT OF COMPUTING

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

Detecting Source Code Plagiarism From Online Software Repositories

Author:
Dominic Rusch

Supervisor:
Dr. Thomas Lancaster

Second Marker:
Dr. Arthur Gervais

June 2022

Submitted in partial fulfillment of the requirements for the MEng Computing course of Imperial College London

Abstract

The majority of existing research and tools within the field of source code plagiarism detection focus on detecting collusion; similar code created by peers working together on individual assignments. Very little research has been conducted on plagiarism of source code from the internet, such as software repositories. Existing research consequently works with relatively small scales, and a gap in plagiarism detection software therefore exists, which can be exploited by those wishing to do so.

This thesis presents a source code plagiarism detection system which can effectively detect modified source code from software repositories. A novel technique within the field, candidate retrieval, is used to reduce millions of source code files to a small set of similar candidates within tens of seconds. This system has been demonstrated to address many challenges associated with the large scale of online source code plagiarism detection, including scalability and performant data structures, as well as mitigating noise present at large scales via novel source code similarity algorithms.

Consequently, detection of multiple plagiarised source code sections is possible for a single provided file, including detection of code translated between programming languages, and heavily modified code segments. The presented system operates at multiple orders of magnitude larger than prior research, in addition to supporting arbitrary languages of different paradigms as a result of an extensible design.

Keywords: source code plagiarism, plagiarism detection, candidate retrieval, large scale

Acknowledgments

I would like to thank my supervisor, Dr. Thomas Lancaster, for his advice and guidance throughout my thesis. His regular insightful feedback allowed me to consider matters I would have otherwise been unable to, resulting in thought-provoking discourse and development.

I am also thankful to Dr. Arthur Gervais, both for agreeing to be my second marker, and for providing valuable critique during development of my thesis.

Lastly, I thank my friends and family, many of whom have supported me during this project through discussion and motivation.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Challenges and Contributions . . . 1

2 Background **3**
2.1 Source Code Plagiarism Techniques . . . 3
2.1.1 Plagiarism Obfuscation . . . 3
2.1.2 Pragmatic Plagiarism Obfuscation . . . 4
2.2 Views of Source Code Plagiarism . . . 4
2.2.1 Academic Perspectives . . . 5
2.2.2 Student Perspectives . . . 5
2.3 Source Code Plagiarism Detection . . . 6
2.3.1 Plagiarism Detection for Large Code Repositories . . . 6
2.3.2 Plagiarism Detection across languages . . . 7
2.3.3 JPlag . . . 8
2.3.4 Codequiry . . . 8
2.3.5 Natural Language Processing . . . 8
2.3.6 SIM . . . 9
2.3.7 Holmes . . . 9
2.3.8 Sherlock . . . 10
2.4 Sources of Plagiarism . . . 10
2.5 Traditional Plagiarism Detection . . . 10
2.5.1 Extrinsic Detection . . . 11
2.5.2 Intrinsic Detection . . . 11

3 Overview **12**
3.1 System Components . . . 12
3.2 Technologies . . . 13
3.2.1 Programming Language . . . 13
3.2.2 Environment . . . 13
3.2.3 Database . . . 14

4 Database Population **15**
4.1 Platform Choice . . . 15
4.2 Choice of Languages . . . 15
4.3 Crawling . . . 16
4.3.1 Repository Iteration . . . 16
4.3.2 File Iteration . . . 17

4.3.3 Source Code Entries . . . 18
4.3.4 Code Upkeep . . . 18
4.4 Analysis of crawled repositories . . . 19

5 Tokenizers 20
5.1 Requirements . . . 20
5.2 Token Choice . . . 21
5.2.1 Utilisation . . . 22
5.3 Implementation . . . 22
5.3.1 Character Mapping . . . 22
5.3.2 Java Tokenization . . . 23
5.3.3 C/C++ Tokenization . . . 24
5.3.4 Haskell Tokenization . . . 24
5.4 Obfuscation Prevention . . . 26
5.4.1 Untokenized code . . . 26

6 Candidate Retrieval 27
6.1 *N*-grams . . . 27
6.1.1 Suitability . . . 27
6.1.2 Drawbacks . . . 28
6.1.3 Effect of *n* . . . 29
6.1.4 Choice of *n* . . . 29
6.1.5 Distributions . . . 30
6.1.6 Properties . . . 31
6.2 Index . . . 32
6.2.1 Access Times . . . 32
6.2.2 Program Similarities . . . 32
6.3 Index Generation . . . 33
6.3.1 Source Code Iteration . . . 34
6.3.2 Bulk Writes . . . 34
6.3.3 Partitions . . . 34
6.3.4 Generation Speed . . . 35
6.3.5 Batch size . . . 36
6.3.6 Main Memory Efficiency . . . 36
6.4 Index Querying . . . 37
6.4.1 Index Checker . . . 37
6.4.2 Similarity Assignment . . . 38
6.4.3 Candidate Set . . . 39
6.4.4 Time Complexity . . . 40
6.4.5 Parallelization . . . 40
6.4.6 Testing . . . 41
6.5 Validation . . . 42
6.5.1 Obfuscated Plagiarism . . . 43
6.5.2 Multiple Sources . . . 44
6.5.3 Noise . . . 46
6.5.4 Potential Noise Mitigation . . . 47
6.6 Drop-off . . . 47
6.6.1 Similarity Function Choice . . . 49
6.6.2 Power Law Drop-off . . . 49

6.6.3 Linear Drop-off 50
6.6.4 Effect 51

7 Detailed Analysis **53**
7.1 Candidate Comparison 53
7.1.1 Implementation 54
7.1.2 Candidate Set Size 54
7.1.3 Candidate Parsing 54
7.2 Results 55
7.2.1 Similarities 55
7.2.2 Issues 56
7.3 Extensibility 56

8 Evaluation **57**
8.1 Plagiarism Obfuscation Levels 57
8.2 Multiple Sources 58
8.3 Plagiarism from Large Files 59
8.4 Small Plagiarised Sections 60
8.5 Inter-lingual Detection 61
8.6 Comparison 62
8.7 Summary 62

9 Conclusion **63**
9.1 Parameters 64
9.2 Future Work 64
9.2.1 *N*-grams 64
9.2.2 Subsection Queries 64
9.2.3 Index Updates 65
9.2.4 Intermediate Representations 66
9.2.5 Pragmatic Extensions 66
9.2.6 Horizontal Scaling 66

Bibliography **66**

A Appendix **72**

List of Figures

2.1 Code2Vec prediction of the name for a function computing whether an input number is prime. Removal of braces completely alters the prediction. 9
2.2 Example output showing the matches between submitted programs in a corpus, using Sherlock. 10

4.1 Example of an entry within the database's source code table. 18

5.1 A simple Java class which can increment values. 26
5.2 A plagiarised version of figure 5.1. 26

6.1 Percentiles of the number of distinct Java programs containing the same *n*-gram . . 30
6.2 A Java function which prints the values in an array. 30
6.3 Character stream of figure 6.2. 31
6.4 Generated 3-grams of figure 6.3. 31
6.5 Abridged 3-gram frequencies of figure 6.4, and therefore program 6.2. 31
6.6 An example of an index generated using 3-grams. 32
6.7 Similarity equation for an indexed program p, given a provided source program s for a chosen *n*. $|s|$ denotes the number of tokens in s. g denotes an *n*-gram within both p and g. 'sim' denotes the similarity function, used to compare *n*-gram frequencies within p and s, labelled $f_{p,g}$ and $f_{p,s}$ respectively. 38
6.8 The initial similarity function used to compare *n*-gram frequencies, equivalent to their minimum. 38
6.9 Similarity assignment algorithm for index querying. 39
6.10 Figure showing the score for an individual source *n*-gram which occurs 10 times, for different similarity functions. 48
6.11 The final similarity function used to compare *n*-gram frequencies during candidate retrieval, utilising linear drop-off. 51

A.1 Overview of the components within the plagiarism detection system. Arrows indicate one component being used by another. 73
A.2 Distribution of the number of crawled files within each crawled repository. 73
A.3 Distribution of the number of tokens within each crawled file. 75
A.4 Percentiles of the number of distinct Haskell programs containing the same *n*-gram. 77
A.5 Percentiles of the number of tokens within indexed Java files. 77
A.6 Percentiles of the number of *n*-grams per program in which the *n*-gram occurs in, for the Java 5-gram index. 78

List of Tables

4.1 Analysis of file types within crawled repositories 19

5.1 Analysis of the success rate of tokenization per language. 25

6.1 Time taken to tokenize and index a batch of programs for the Java 5-gram Index, given the existing number of programs within the Index and optimisations used. . . 36
6.2 Table comparing the time taken to retrieve candidates for different length files within the Java 5-gram index when using synchronous and parallel queries. 41
6.3 Candidate set similarity thresholds when querying unmodified plagiarised source code. 42
6.4 Varying program similarity and ranks compared to the actual Java true candidate, for different levels of plagiarism obfuscation. 43
6.5 Varying program similarity and ranks compared to the actual Haskell candidate, for different levels of plagiarism obfuscation. 44
6.6 Table showing the rank of multiple true candidates for sources containing multiple unmodified plagiarised code sections, within the Java 6-gram index. 45
6.7 Table showing the rank of multiple true candidates for sources containing multiple plagiarised code sections with medium plagiarism obfuscation performed, within the Java 6-gram index. 46
6.8 Table showing the rank of multiple true candidates for sources containing multiple unmodified code sections, within the Java 6-gram index using $\frac{3}{5}$ power law drop-off. 49
6.9 Table showing the rank of multiple true candidates for sources containing multiple unmodified code sections, within the Java 6-gram index using linear drop-off. . . . 50
6.10 Multiple true candidate's ranks for sources containing multiple plagiarised sections with medium plagiarism obfuscation performed, in the Java 6-gram index using linear drop-off. 50
6.11 Table showing the true ranks for large programs from which subsections were plagiarised from, for different similarity functions. 51
6.12 Varying program similarity and ranks compared to the actual Java true candidate, for different levels of plagiarism obfuscation when using linear drop-off. 52

7.1 Table showing the similarities assigned by JPlag for very large true candidates, compared to the average candidate similarity. 56

8.1 Varying program similarities compared to the actual Haskell candidate, for different levels of plagiarism obfuscation, using linear drop-off. The true rank was 1 for each file. 58

8.2 Table showing the rank of multiple true large candidates for sources containing multiple unmodified code subsections, within the Java 6-gram index using linear drop-off. 59
8.3 Table showing the true ranks and similarities for plagiarised subsections of large programs, within the Haskell 6-gram index, using linear drop-off. 60
8.4 Table showing the true rank of small candidates, when incorporated within a larger program, for the Java 6-gram index. 61

A.1 Analysis of the number of crawled repositories 72
A.2 Analysis of the number of files within crawled repositories 74
A.3 Analysis of the number of characters within each crawled file 74
A.4 Analysis of the number of lines within each crawled file 74
A.5 Analysis of the number of tokens within each crawled file 74
A.6 Table showing the number of tokens and distinct n-grams per index. 75
A.7 Table showing the similarities of all programs compared to an arbitrary program, from sample of 10,000 Java programs, for varying values of n and averaged over 10 programs. 76
A.8 Table showing the number of distinct Java programs which contained the same n-gram. 76
A.9 Table showing the number of distinct Haskell programs which contained the same n-gram. 76
A.10 Table showing the rank of multiple true candidates for sources containing multiple unmodified code sections, within the Java 4-gram index using no drop-off. 76
A.11 Table showing the rank of multiple true candidates for sources containing multiple unmodified code sections, within the Java 6-gram index using square root drop-off. 78
A.12 Table showing the rank of multiple true candidates for sources containing multiple plagiarised code sections with medium plagiarism obfuscation performed, within the Java 6-gram index using square root drop-off. 79
A.13 Table showing the rank of multiple true candidates for sources containing multiple plagiarised code sections with medium plagiarism obfuscation performed, within the Java 6-gram index using $\frac{3}{5}$ power law drop-off. 79
A.14 Table showing the similarity assigned to multiple true candidates by JPlag, after having been retrieved in the candidate set. 79
A.15 Table showing the similarity assigned to multiple true candidates by JPlag, after having been retrieved in the candidate set. Candidates were subsections of the original files 80
A.16 Table showing the rank of multiple true candidates for sources containing multiple plagiarised code sections with light plagiarism obfuscation performed, within the Java 6-gram index using linear drop-off. 80
A.17 Table showing the rank of multiple true candidates for sources containing multiple unmodified code sections, within the C 5-gram index using linear drop-off. 80
A.18 Table showing the rank of multiple true candidates for sources containing multiple unmodified code sections, within the header 5-gram index using linear drop-off. . . 81
A.19 Table showing the rank of multiple true candidates for sources containing multiple unmodified code sections, within the C++ 5-gram index using linear drop-off. . . . 81
A.20 Table showing the rank of multiple true candidates for sources containing multiple unmodified code sections, within the Haskell 6-gram index using linear drop-off. . 81
A.21 Table showing the true ranks and similarities for plagiarised subsections of large programs, within the Haskell 5-gram index, using linear drop-off. 82

A.22 Table showing the rank and similarity of true candidates for inter-lingual plagiarism detection, within the Java and C++ 5-gram indexes. 82

Chapter 1

Introduction

1.1 Motivation

Plagiarism is the act of using other's work, and presenting it as your own without reference to the original work [1]. Plagiarism of source code can occur from peers, textbooks, and online sites or software repositories [2], and represents a serious issue for the integrity of academia. Source code plagiarism is often motivated by desire for higher grades, and can undermine honest student's confidence in tuition [3]. However, the ease of access to source code through the internet has made plagiarism more convenient, and consequently demands adaptation from assessment procedures [4][5].

Source code plagiarism can take many different forms, for example copying and minimally adapting source code, obtaining code from someone else, or collaborating with others to produce similar source code (*collusion*), all without adequate reference to original authors [6]. Within the field of source code plagiarism detection, collusion has primarily been studied, and many tools exist to detect collusion within a corpus of source code [7].

However, collusion may constitute an increasingly smaller proportion of source code plagiarism, as a consequence of the vast amount of accessible online code. Plagiarism primarily occurring via collusion could be considered outdated, yet this is what is predominantly detected for within academic institutions [8], likely as few tools exist to detect plagiarism from online sources [9]. Thus, tools capable of detecting source code plagiarised from online corpora are required.

1.2 Challenges and Contributions

Many challenges exist for this project. To detect plagiarism from online software repositories, the source code from such repositories will need to be readily accessible for efficient comparison. Therefore, source code will be retrieved and stored from such repositories, within a database. As there is a substantial number of online repositories, this project will demonstrate performance at scale by utilising millions of source code files during plagiarism checks. Consequently, implementing efficient and scalable methods will be crucial, both when generating the required infrastructure to detect plagiarism at this scale, as well as during detection itself.

In order to detect plagiarism from stored files, a two-stage process will be utilised. The first stage will filter stored source code files to a small set, from which the second stage can perform more selective albeit resource-intensive analysis on each retrieved file, thus detecting final instances of suspected plagiarism. These stages are referred to as candidate retrieval and detailed analysis. Candidate retrieval is novel within the field of source code plagiarism detection, whilst the majority of existing research focuses on small corpora, and therefore can be employed within detailed analysis.

For candidate retrieval to retain the most likely instances of plagiarism, a metric to assign similarity scores to stored source code files is required, when compared to provided programs. However, plagiarised source code can be greatly altered, for example by adding additional functionality, deliberately modifying code to avoid detection, or even by translating plagiarised code to another programming language. Furthermore, plagiarism could have occurred from multiple sources, and it is possible that only small sections of a provided program are plagiarised. Such factors make effective plagiarism detection a difficult task, which is compounded by the large scale of this project, as noise within the system is introduced from the vast number of stored source code files.

The need to implement similarity metrics in an efficient and scalable manner is therefore paramount, whilst being resistant to plagiarised code which has been significantly modified or disguised. This thesis therefore builds on existing research to detect source code plagiarism, and introduces novel techniques which mitigate noise at scale and aid detection of altered source code. Moreover, efficient methods to generate required infrastructure are demonstrated, allowing the presented plagiarism detection system to arbitrarily scale.

Within prior research, the largest known study has utilised ~60,000 files [10]. This thesis demonstrates effective detection of plagiarism from ~3.4 million files of multiple programming languages and paradigms, within tens of seconds. The presented system is unaffected by superficial techniques used to obscure plagiarism, whilst being resistant to further modified plagiarism. Moreover, several distinct sources of plagiarism are able to be identified, in addition to plagiarised code translated between languages.

Vast potential for future work is also discussed, presenting several novel techniques which could increase the efficacy of proposed detection methods. The key contributions of this thesis are:

- The process of candidate retrieval for source code. This is demonstrated to be both performant and scalable, and able to effectively filter large source code stores. Within this thesis, the most similar candidates can be retrieved from millions of source code files within tens of seconds. This technique is novel within the field of source code plagiarism detection and is described in chapter 6, with the implementation covered in section 6.4.

- Mitigation of noise introduced at large scales. With vast numbers of source code files, some can be deemed more similar than actual plagiarised programs if significant modifications have been made to a provided file containing plagiarism. Novel similarity metrics are therefore presented which effectively reduce noise, thus allowing original plagiarised sources to be retrieved with a far higher recall. This is presented in section 6.6.

- Efficient and scalable infrastructure generation. Candidate retrieval utilises bespoke data structures, which can be time-consuming to generate. Multiple techniques are therefore demonstrated to ensure scalability. These are novel within the field, as prior detection engines only check for plagiarism within a provided corpus, and so do not require persistent storage. This is discussed in section 6.3.

The presented system is also extensible and can be seamlessly modified to handle additional programming languages. Whilst this detection system uses source code from online software repositories, it is generic to the input source, and could be populated with files from any other medium. Likewise, custom detailed analysis of identified source code from candidate retrieval can easily be implemented, allowing this system's adaptation to any desired use case.

Chapter 2

Background

This chapter will examine how and why source code plagiarism occurs, in addition to existing detection engines. Sources of plagiarised source code and approaches to detecting text-based plagiarism will also be reviewed.

2.1 Source Code Plagiarism Techniques

To understand the challenges faced during source code plagiarism detection, it is helpful to see how those aiming to plagiarise code would typically do so, for example students in an academic environment.

Collusion is one instance of source code plagiarism. Students can work together to produce similar code submissions, often with minor adaptations, for example converting a 'for' loop to an equivalent 'while' loop, changing the order of some statements, or renaming variables. However, the submitted programs will still likely be similar overall. This type of plagiarism has had numerous publications researching its impact and detection [6][9], and various tools exist to detect collusion within a submitted corpus, such as JPlag [11], Sherlock [12], and MOSS [13].

Students can also plagiarise by copying code from online sources, again with minimal or moderate adaptations. A difference to collusion is that shorter sections may be plagiarised, such as single functions or sections of a larger body. Collusion may typically produce similar code for more of the submission, due to students working together for large sections of assignments.

Inadequate acknowledgement is also a form of plagiarism used by some students. This is typically enacted through students excluding citation of sources, possibly including their own previous work. Inadequate acknowledgement also includes providing false references, for example citing a source for a piece of work, despite the given source not including the referenced material [6]. Therefore, detection of this type of plagiarism would be possible by identifying a true source and noting mismatches.

Source code plagiarism via language conversion is another technique employed by some students. This can include minimal adaptations, for example if certain patterns differ slightly between languages. This technique may be used if a student wishes to obfuscate evidence of plagiarism to a greater extent, or if no implementations of the same language are readily available to plagiarise code from.

2.1.1 Plagiarism Obfuscation

Plagiarism obfuscation is the act of modifying plagiarised code in an attempt to conceal the fact that it was plagiarised. Typically, the semantics of code would not be changed, in order to

retain original functionality. Various structural changes can be made to obfuscate plagiarism, for example the conversion of loops as previously mentioned.

Structural changes to obfuscate plagiarism can also include re-ordering of statements whilst preserving semantics, inverting the order of certain operators such as turning '`i++`' into '`++i`', and re-ordering arithmetic or boolean expressions. For instance, a boolean condition '`a && !b`' could be changed to '`!b && a`' in most cases, whilst preserving semantics.

The translation of source code between programming languages (*inter-lingual plagiarism*) could also be considered a form of plagiarism obfuscation. Whilst certain languages may be similar, inter-lingual plagiarism often includes many notable structural changes, whether as a consequence of an automated tool's artefacts, or different design patterns or implementations performed manually.

Non-structural changes to obfuscate plagiarism are also common [14]. These include renaming variables and reformatting source code, and are referred to as *superficial* techniques, due to logical code remaining unchanged.

2.1.2 Pragmatic Plagiarism Obfuscation

Whilst there are many possible plagiarism obfuscation techniques, in practice, one attempting plagiarism obfuscation would likely not employ all of the aforementioned techniques. Plagiarism can occur due to some students wishing to spend "as little time and energy as possible" on assigned work [15]. Plagiarism can also occur due to time constraints, whereby students feel as though they need to plagiarise in order to finish assignments in time, as a consequence of their own mismanaged time [16]. Consequently, students plagiarising source code would likely not want to spend much time obfuscating plagiarism, or indeed have the time to do so at all.

Superficial changes to plagiarised source code may therefore be more likely, for example modifying the text in comments, changing the formatting of code sections, or renaming variables and other identifiers. This is as these superficial techniques do not change the semantics of a program in any way, and so no time would be needed to ensure that the plagiarised code still performs as expected. Such superficial modifications which do not structurally change code are able to be completely mitigated through tokenization of source code. Yet, such changes are likely still expected, as they can drastically affect the visual appearance of source code.

For those who do structurally modify code in an effort to obfuscate plagiarism, such changes are also likely to be restricted in scope, for the same reasons. For example, outlining or inlining variables is a simple obfuscation technique which structurally affects source code, however is trivial to check that semantics remain identical:

`c = PI * diameter;`	*Original statement*
`c = 3.14 * diameter;`	*Modified statement via an inlined variable*

Likewise, re-arranging arithmetic or other nested expressions can be done with ease, for instance modifying '`x = y * 2`' to '`x = 2 * y`', or removing braces from single line statements. Such simplistic structural changes could therefore be frequently expected in plagiarised code, however complex structural changes may be less likely. Whilst this would be beneficial for detecting source code plagiarism, heavily obfuscated plagiarism can of course still occur, and so efforts to mitigate obfuscation are needed in effective detection tools.

2.2 Views of Source Code Plagiarism

Compared to traditional plagiarism, source code plagiarism is a newer field, and consequently does not have as widely spread views on what constitutes plagiarism. From a survey of 770

computing students in 2011, the authors concluded that existing "training is not fully effective in raising awareness of what constitutes plagiarism" for students [17]. Opinions on what constitutes source code plagiarism may also differ between computing academics and students, as well as within both individual groups, such as self-plagiarism [17].

2.2.1 Academic Perspectives

In 2006, *Joy & Cosma* conducted a survey of 59 UK academics. Views on source code plagiarism were explored, looking into whether various examples constituted either source code plagiarism, another academic offence, or no offence [18]. The authors noted that literature at the time indicated "no commonly agreed description of what constitutes source-code plagiarism", which is reflected in the survey responses.

Although most questions had a majority consensus, multiple academics for each question had differing opinions, showing the lack of overall consensus amongst computing academics. Multiple questions also had split opinions with no clear majority, showing that certain cases involving potential plagiarism had no single common view. For example, when asked which of the aforementioned categories "providing false references (i.e. references exist but do not match the source-code that was copied)" fell into, 33 marked it as plagiarism, and 25 marked it as a different academic offense. Similarly, "providing pretend references (i.e. references that were made-up by the student and that do not exist)" received 32 responses of plagiarism, and 26 of differing academic offenses. These almost split views show the lack of consensus with regards to plagiarism via inadequate acknowledgment.

For an instance of collusion, where "two students work together for a programming assignment that requires students to work individually and the students submit very similar source-codes", 37 academics viewed this as plagiarism, whilst 18 thought it was a different academic offense. This demonstrates that a significant portion of academics had different opinions of what counts as collusion, even for a presently accepted instance of plagiarism [6].

However, the survey discussed [18] is not up to date with the rapidly advancing and growing field of computer science, and also had a relatively small sample size. Despite these drawbacks, the differing views and lack of overall consensus still demonstrate some of the challenges faced with addressing source code plagiarism in academic environments.

2.2.2 Student Perspectives

In 2010, *Joy & Cosma* explored the question "Do students understand what source code plagiarism means?", via a survey of 770 computing students [17]. Whilst most questions were generally answered correctly, certain topics presented clear issues for the surveyed students. When given a "clear instance" of self-plagiarism, just 6.8% of the students either correctly identified that plagiarism had taken place, or were unsure if it counted as plagiarism. As the authors note, this suggests that most students view self-plagiarism as acceptable in general, or do not realise it is a form of plagiarism at all. Future education in academia should likely note this, and may give cause for source code plagiarism detectors to include previous submissions of students, to account for this kind of plagiarism.

An example of plagiarism where "a student writing a C++ program who finds equivalent code in a Java textbook, converts this to C++, and submits the program without reference to the textbook" was given. As noted previously, this is an example of source code plagiarism via language conversion. However, just 48.6% of students correctly identified this as plagiarism, again suggesting a lack of understanding for this type of plagiarism. As for the previous example, extra emphasis should likely be placed on this within academia, or at least of the importance of always

referencing any code sources when feasible.

Most other examples of source code plagiarism were correctly identified by students, with more than ~90% successful identification. One can conclude that most students do understand what source code plagiarism means, however there do exist areas which are still unclear, both for students and academics. Compared to the prior mentioned survey of computing academics, the student perspectives survey had ~13x more respondents, and is more recent, addressing both primary concerns. However, it is still more than a decade old as of writing, yet still serves the purpose of demonstrating the existence of 'grey areas' within plagiarism, which are not fully understand or agreed upon.

2.3 Source Code Plagiarism Detection

Different techniques exist to detect source code plagiarism, broadly being categorized as either *attribute-counting-based*, or *structure-based* [19].

Attribute-counting-based techniques compare the frequencies of attributes between programs, such as the number of unique operators and operands. Early source code plagiarism detectors measured such attributes, and calculated pairs of program's similarity as the difference of their values [10]. However, this basic technique considers the overall program, and therefore would only detect instances of plagiarism where the entire file was copied, possibly with minimal adaptation. Such techniques have existed since 1976, and over the following decades evolved to measure additional attributes and language statements [19].

Some work also focused on measuring attributes within the error and execution logs of a program, based on the premise that attributes within code by themselves are not sufficient to measure similarity [20]. However, this requires additional data which may not always be possible to retrieve, and was not evaluated against any other existing tools.

Clustering-based techniques also exist within attribute-counting, and attempt to group similar source codes together, instead of presenting them as pairs [21]. This is as collusion can involve more than just 2 students, so presenting such instances in a pairwise fashion may be misleading.

Structure-based techniques inspect the structure of programs to determine plagiarism. This is typically enacted via string matching, whereby the statements within a program are tokenized, and then the resulting tokens are compared to those of another program. The Running Karp-Rabin Greedy String Tiling (RKRGST) algorithm is commonly used; many plagiarism detectors use optimised or other forms of it [19]. Some tools, such as the approach discussed in 2.3.1, use string alignment algorithms instead of RKRGST.

Techniques analysing the source code creation process also exist, utilising logs made during a programming assessment as evidence of collusion. However, these require that the assessment takes place in-person [4].

2.3.1 Plagiarism Detection for Large Code Repositories

In 2007, *Burrows et al.* introduced a system for efficient plagiarism detection within large code repositories [10]. They were motivated due to scalability concerns and limits present within existing plagiarism detection tools, and note that typical tools "search for matches in an exhaustive pairwise fashion, which is unscalable".

Their system tokenized the constructs of input programs to produce *n*-grams. For example, the *3*-grams of `ABCDE` are `ABC`, `BCD`, and `CDE`. In this case, '`int`' appearing within a program could map to `A`, whilst the '`for`' expression could map to `B`, and so on.

An inverted index of the *n*-grams can then be constructed, mapping each *n*-gram to pairs of *(program number, occurrence)*. For example, a *3*-gram of `ABC` appearing within programs *1* and *2*

of a given corpus, with *4* and *5* occurrences respectively would exist within the inverted index as:

$$\texttt{ABC} \rightarrow 1,4 \quad 2,5$$

Once the inverted index has been constructed, each program's *n*-grams can be queried within the index, obtaining the total number of identical *n*-gram occurrences with other programs. As the program itself is in the index, it will have the maximum number of occurrences, which can be used to relatively score other programs similarity. For example, if a given program had 100 total occurrences of its *n*-grams, then another program with 50 shared *n*-gram occurrences would have a similarity score of 50%.

The authors chose an *n* value of 4 for higher efficiency, but noted that this resulted in a lower selectivity. To mitigate this, local string alignment is used for programs with more than 30% similarity, to identify matching program sections and generate a more accurate similarity score.

The authors evaluated the running time of their system. It took ˜24 minutes to query 296 programs against an index containing 61,540 programs (excluding the alignment time), and the further alignment took ˜12 minutes. However, the results were for a 733 MHz Pentium III processor with 512 megabytes of RAM. With more modern hardware, these times would likely be significantly lower. Existing versions of other source code plagiarism tools at the time like JPlag performed better for some collections, suggesting possible improvement for future work. Potentially replacing the latter alignment step with a pre-existing tool such as JPlag or Sherlock on the retrieved source candidates would be a reasonable trade-off for time against performance.

2.3.2 Plagiarism Detection across languages

In 2006, *Arwin & Tahaghoghi* introduced XPlag, a tool to detect source code plagiarism via language conversion [22]. This was the first published study to detect inter-lingual plagiarism.

XPlag uses GCC to compile C or Java code to Register Transfer Language (RTL). Optimisation is enabled, which simplifies RTL instructions and inlines functions. The latter also helps to overcome the plagiarism technique of inlining (or extracting) functions. The resulting code is then filtered for 'significant' keywords (as decided by the authors), such as `reg`, `set` or `if_then_else`.

The authors noted that looping statements were compiled differently for C and Java. Therefore, generic `loop_begin` and `loop_end` keywords were included, and the specific looping keywords were filtered. Other keywords had similar sequences across the languages, so could remain unchanged. The filtered RTL uses indentation, representing nested expressions within the originally generated RTL instructions.

The filtered RTL was then tokenized, including the respective indentation of the keyword. Consider the following code example:

```
instr
    set
        reg
        int
```

This could be mapped to `0a1b2c2d`, where `instr`, `set`, `reg` and `int` map to `a`, `b`, `c` and `d` respectively, and the prepended numbers represent the indendation level. The filtered RTL comprises of multiple code blocks such as in the example code. The tokenized code blocks are then used to produce *3*-grams. The *3*-grams are then indexed, mapping to programs which also contain the given *n*-gram. When a new C or Java program is queried within the index, the resulting similar programs will be returned, and the overall similarity can be calculated via the relative percentage of similar *3*-grams.

The authors evaluated XPlag with 3 collections, containing C, Java, and mixed source codes. The average similarity of programs for various n values was measured, and as expected, programs within the test corpus were most similar for smaller n values, due to smaller groups of code blocks occurring more frequently. The precision and recall on the collections for different n values was also measured, to identify the eventual decided-upon n value of 3. XPlag performed similarly to JPlag for the C and Java collections. Xplag also successfully identified inter-lingual plagiarism within the mixed collection, with an optimal trade-off for precision and recall around 25-40% similarity of the RTL.

The techniques used within XPlag may be applicable for this project. However, due to the large number of source code to be crawled, compiling each to an intermediate representation may add a large overhead. Yet, this may not present an issue, as pre-processing of the crawled code can occur long before before query time. An alternative approach could be to avoid inter-lingual detection, or to attempt to tokenize multiple languages with the same token grammar, avoiding compilation.

Furthermore, the system presented is relatively unscalable for more languages. Challenges faced for just C and Java required bespoke solutions (e.g. different loop representations in RTL), which may not be applicable for other languages. XPlag's approach also restricts languages to those supported by the GCC compiler. As discussed in 2.3.1, the approach of indexing n-grams is more scalable than pairwise comparison of programs.

2.3.3 JPlag

JPlag [11] is an open-source source code plagiarism detector, which currently supports 6 programming languages and plain-text, including Java, Python, C++ and C#. It was originally developed in 1996, and has been under ongoing development with new releases since. JPlag tokenizes input programs, and then uses an optimised version of the RKRGST algorithm to obtain similarity scores for programs, in a pairwise fashion.

In a 2002 paper from JPlag's authors, various forms of plagiarism attacks are analysed [14]. Among those consistently detected are modification of code formatting (e.g. changing line breaks and tabs), modification of comments, and changing variable names. This is to be expected, as JPlag removes whitespace and comments before tokenization. Certain attacks were generally undetected by JPlag, including function inlining, and modification of control structures. Moving subexpressions to new variables also proved to be successful plagiarism techniques, being undetected in the author's evaluation over 16 instances.

2.3.4 Codequiry

Codequiry [23] is an online service providing code plagiarism and similarity checking. It supports 20+ languages, and indexes online source-code hosting sites, such as Github [24] and StackOverflow [25]. It uses a clustering-based technique, presenting identified program clusters to users through an online interface.

The site claims that Codequiry "has proven to be more effective than other source code similarity algorithms such as MOSS", yet no evidence supporting this claim is provided. The tool also does not have any independent evaluation or research associated with it. However, Codequiry's apparent commercial viability suggests that indexing online code is feasible.

2.3.5 Natural Language Processing

Whilst certain existing plagiarism detection techniques could be considered to be natural language processing (NLP), established NLP techniques utilising machine learning could be used,

such as Code2Vec [26], which attempts to learn code representations. Code2Vec could therefore potentially be used to vectorize code snippets, therefore allowing efficient detection of source code plagiarism at a large scale. However, Code2Vec currently only supports C# and Java [27]. Furthermore, Code2Vec relies on elements of code of which plagiarism obfuscation can easily change, such as variable names and braces [28]. As shown in figure 2.1, removal of braces within one-line statements completely alters the prediction of what the provided code performs. The code to compute whether a number is prime shown in figure 2.1 is one of the examples on the Code2Vec website. Therefore, Code2Vec was deemed unsuitable for use in source code plagiarism detection.

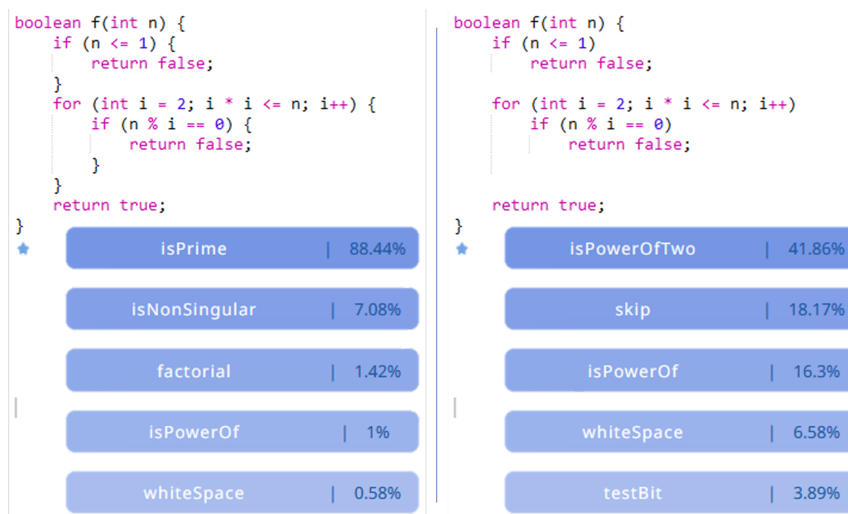

Figure 2.1: Code2Vec prediction of the name for a function computing whether an input number is prime. Removal of braces completely alters the prediction.

2.3.6 SIM

SIM detects plagiarism within a corpus of C programs, passing each through a lexical analyser to generate a stream of integer tokens [29]. Pairwise comparison is used, with the second program being split into sections representing modules of the program. These are then separately aligned with the tokens of the first program, allowing detection of plagiarised programs where the order of modules are permuted.

2.3.7 Holmes

Holmes is a plagiarism detection engine for Haskell programs [30]. Like most detection tools, Holmes ignores comments, whitespace, and also disregards the order of function definitions. Removal of template code is also supported. Holmes uses heuristics to compare the token streams of programs, utilising both sorted and unsorted streams. The call graph of functions is also utilised to assign similarities.

As noted by the authors, plagiarism can often occur from prior year's academic assignments, and therefore Holmes was evaluated with a corpus containing 2,122 submissions ranging over 12 years, with successful identification of 63 "clear cut cases of plagiarism". Multiple refactors to obfuscate plagiarism were evaluated, with resistance demonstrated, resulting in a capable plagiarism detection tool for Haskell.

2.3.8 Sherlock

Sherlock [12] allows users to provide a template file, which all other programs in the given corpus are assumed to be based upon. Common source code fragments are then removed, and the programs are tokenized before being compared pairwise [19]. Users have the ability to choose whether whitespace or comments are ignored. A report of similarities is generated, as well as 'match graph', showing programs similarity above a given threshold, as shown in figure 2.2.

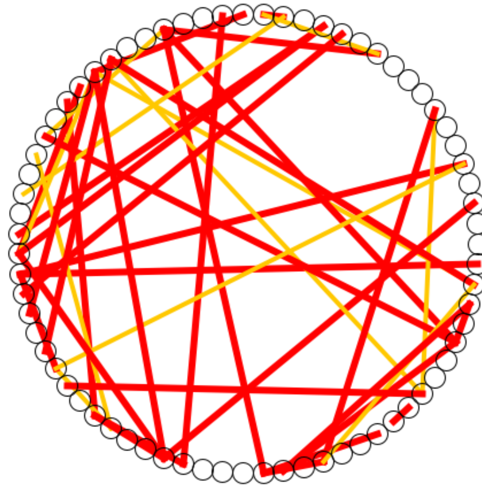

Figure 2.2: Example output showing the matches between submitted programs in a corpus, using Sherlock.

2.4 Sources of Plagiarism

Within academia, students have many possible sources for code plagiarism. As discussed previously, other students are a common source, whether in the form of collusion, or even stealing code. Various source code banks exist online, such as GitHub [24], GitLab [31], and Google Code [32]. Students can also plagiarise from textbooks, which may be harder to detect if the assignment author is not familiar with them, or they are not available and indexed online [17].

Internet sites and blogs such as Stack Overflow [25] and GeeksforGeeks [33] also host source code in the form of questions with answers, or blog posts.

2.5 Traditional Plagiarism Detection

Within the field of traditional text-based plagiarism detection, there are two general approaches: extrinsic and intrinsic detection. Traditional plagiarism detectors typically first locate potential suspicious parts of a text, and then perform a subsequent detailed analysis [4].

Exploring the general approaches for traditional plagiarism is useful to understand the differences between detecting text-based and source code plagiarism. In addition, various evaluation methods and metrics within traditional plagiarism are applicable to source code plagiarism.

2.5.1 Extrinsic Detection

Traditional extrinsic plagiarism detection compares given texts to a large corpus of existing work, from which plagiarism may have occurred. This typically occurs in two stages: candidate retrieval and detailed analysis, as are used for source code plagiarism detection within this thesis.

Candidate retrieval is the process of retrieving possible sources from a large corpus, from which a given text may have plagiarised work from. To future-proof and ensure scalability, many existing traditional plagiarism systems use online search-engine APIs, instead of maintaining a bespoke collection [4]. This approach is likely not as applicable for source code, due to most search-engines focus on text, as opposed to source code queries. For example, a snippet of plagiarised source code with modified variable names and logically equivalent but altered statements would be hard to compare via text-based approaches. However, a source code plagiarism detection system could allow terms related to specific assignments to be provided, enabling targeted crawling of repositories, ahead of querying.

For candidate retrieval, evaluation via the recall metric is more important than precision. This is as, if a candidate is missed in this step, then any subsequent analysis cannot possibly identify it as a source of plagiarism. The same is true for source code plagiarism. For traditional detection systems which use APIs, the number of queries may also be an important metric, as fewer queries would typically cost less, as well as being more efficient [4].

Following candidate retrieval, detailed analysis occurs. This step filters the retrieved candidates to obtain probable sources of plagiarism, to present to the user for them to determine if plagiarism has occurred. This usually occurs by detecting sections of the input text within the retrieved candidates. Initial sections which seem to match are also filtered out if they are too short, or not similar enough, according to a given system's criteria [4]. Exclusion should also occur for a source code plagiarism detection system, as smaller code segments will likely occur in many programs, despite no plagiarism having taken place. Furthermore, even in a source with plagiarised sections, this does not mean that other short 'similar' sections (e.g. lines consisting of a call to a generic `print` function) are also plagiarised, any may only serve to confuse the end-user if included.

If one possible source has multiple probable plagiarised sections, then the sections can be merged into one, via an approach called *extension* [4]. This technique seems applicable in the context of detecting source code from online repositories. As with traditional plagiarism, if a student were to plagiarise from one section of a source, then it is likely they have also viewed the rest of the source. Such sections may not be contiguous though, so thresholds can be used to limit multiple sections being merged.

2.5.2 Intrinsic Detection

Traditional intrinsic detection does not use external sources to identify text-based plagiarism. *Stylometry* is used to analyse an author's writing style via statistical methods, with the goal of identifying stylistically different sections within the same work [34]. This is a potential indicator of plagiarism, and such identified submissions can become input for extrinsic detectors, or flagged for human review [4].

This technique could be applied for source code. As with traditional text-based work, individual styles exist for writing code, so may be able to factor into plagiarism detection. However, the extent of personal style within code is less, due to more restrictive grammar. Furthermore, this would be relatively easy to prevent for students wishing to plagiarise, due to code-formatting tools and linters being commonplace. Nevertheless, for instances of suspected plagiarism, code stylometry could be applied to aid in determining whether source code plagiarism has occurred.

Chapter 3

Overview

To implement a system capable of proficiently detecting source code plagiarised from online repositories, the following criteria will need to be met:

1. Accurate detection of plagiarised source code
2. Efficient detection of plagiarism at a large scale
3. The ability to detect multiple plagiarised sources
4. Resistance to plagiarism obfuscation techniques
5. Scalable and extensible

3.1 System Components

In order for the plagiarism detection system to be effectively used, a user should be able to provide source code to the system, with any possible plagiarised sources and matching sections being output in the case of plagiarism having occurred.

As discussed in section 2.5.1, within the field of traditional text-based plagiarism detection when utilising a large source corpus, two filtering stages can be used to identify plagiarism sources. For this project, this broad approach will be used, such that an initial *candidate retrieval* stage filters the entire source code corpus down to a much smaller size, from which *detailed analysis* can be performed; a more selective albeit resource-intensive stage, with the goal of producing the final output to a user.

As initially presented in this report in 2.3.1, an *index* can be used to map token n-grams to all the programs which contain the same n-gram. As mentioned, a *token* is an individual component of source code, such as a variable name, or syntactic symbol like '`while`' and '`==`'. These tokens can be combined into *n-grams*. An n-gram is a contiguous sub-sequence of n elements from a sequence, generated by moving a sliding window of length n. For example, the 3-grams of `ABCDE` would be `ABC`, `BCD`, and `CDE`. n-grams are typically constructed from source code tokens by first mapping the tokens to an individual character, and so the entire program's sequence of tokens can be transformed into a sequence of characters [10]. Therefore, a single n-gram can represent n consecutive tokens within a source code file. In this manner, a provided source code file can be checked for plagiarism against indexed files by comparing the similarity and distribution of their n-grams. This approach yields resistance to plagiarism obfuscation techniques, as a modified plagiarised program will likely still contain many original sequences of n tokens, enabling identification of the original source.

An index will therefore be constructed and populated with token *n*-grams from each downloaded file, with the goal of allowing efficient, plagiarism-resistant and scalable queries within the candidate retrieval stage. Consequently, several primary components will be required to create a working end-to-end scalable plagiarism detection system:

- *Crawler*: to iterate and download files from online repositories.
- *Database*: to store source code and metadata of all crawled repositories.
- *Tokenizers*: to parse source code and produce a consistent token stream to generate *n*-grams.
- *Index*: to map token *n*-grams to programs containing them, allowing efficient queries.
- *Candidate Retrieval*: to identify possible candidates from the index of which a provided program could have been plagiarised from.
- *Detailed Analysis*: to analyse and refine the retrieved candidates, ultimately providing the user with any identified plagiarised sources.

The basic interaction between these components is shown in figure A.1. The design of each component and justification over alternatives will be discussed in the following sections.

3.2 Technologies

3.2.1 Programming Language

Whilst any mainstream programming language would be viable to use for this project, certain languages are better suited than others. Specifically, this project would ideally utilise existing solutions for known challenges which will arise, such as large-scale data stores, and source code parsing/tokenization. The primary programming language to be used for this project was therefore chosen to be Python [35], as actively maintained libraries exist to perform such tasks, in addition to being able to utilise previous experience with the language.

Furthermore, as a dynamic language without requiring compilation, turn-around between code modifications and execution is minimal. This reduces total time and overhead during program executions with small run-times, such as experimentation, initial development, certain analysis tasks, and parsing of individual source files. For tasks with an artificial overhead, such as rate-limited crawling, no advantage could be gained from a faster-to-execute language such as C or C++, so the relatively slower execution speed of Python is mostly mitigated, overall making it a suitable choice of primary language.

3.2.2 Environment

A consistent and replicable environment is essential to accurately measure criteria for this project, as well as enabling productive development. A Virtual Machine (VM) from the Department of Computing (DoC) at Imperial College was therefore provisioned for the duration of this project. The DoC VM used has 4 single-threaded cores operating at 2GHz, each with a 512KB cache, and with 8GB of RAM, running the Ubuntu 20.04 operating system.

The use of a VM enables productivity via remote development from multiple locations and machines. A new VM instance also provides a known working environment without any extraneous applications installed, which may inadvertently cause issues or unaccounted overhead, as is possible with a personal machine.

3.2.3 Database

This project requires persistent storage for many components, most notably for the database to store source code, but also for the index to map *n*-grams to programs, and for the tokenizers, to store the mapping between tokens and characters.

Many database libraries exist for Python, such as relational databases like MySQL [36] and SQLite [37], as well as 'No-SQL' libraries like MongoDB [38], Redis [39], or Neo4J [40]. As the design and schemas of the plagiarism detection system's database were not known from the outset of this project, a flexible database solution was desired. Relational SQL databases were therefore ruled out, also due to their lack of ease in regard to scalability, when compared to non-relational databases.

Whilst Redis offers high speeds and data structures likely to be used such as hash tables, like relational databases it also requires a well-defined schema. Redis also wouldn't perform optimally due to the limited RAM on the provisioned VM [41]. Furthermore, despite Redis' key-value pair structure being useful for the plagiarism detector's index, this structure would infringe on storing crawled source code, which would ideally exist as independent documents. Neo4J, which is a graph database, also is not preferable for this same reason of a data model misfit.

MongoDB was therefore used as the database for this project. MongoDB is a document-oriented database, offering a flexible schema with far fewer constraints than other systems. It also offers performant scalability, however is not as fast as fixed-schema databases, when operating with the same scale and schema.

Whilst a mixture of databases could be used, such as MongoDB to store source code files and Redis to store the aforementioned index, the structure of such data stores was not known at the start of this project, in addition to the simplicity of one database system being preferential. Hence, MongoDB would solely be used for this project.

As any decided-upon data model may require changing at some point within the project's development cycle, a non-rigid solution such as MongoDB offers is also favourable. MongoDB is completely supported on this project's VM environment [42], and also has an actively maintained Python interface, PyMongo [43].

PyMongo allows parallel requests across database tables [44], potentially enabling optimisations. MongoDB will automatically assign unique IDs to each document if not provided, which will be practical for referring to individual files retrieved from crawling, offering efficient comparisons and storage across other database tables. MongoDB can also act as a key-value store, such as for the index, as it uses a tree-like structure which provides efficient lookups, given an entries ID [45].

Chapter 4

Database Population

In order to detect source code plagiarised from online repositories, the code from these repositories needs to be readily accessible to allow for efficient comparison against provided programs. Therefore, code from online repositories was downloaded and stored (*crawled*), along with metadata such as each file's respective URL. If potential plagiarism is identified from a crawled file, a user can therefore immediately access the original online source, allowing navigation of other potentially related files, and further investigation.

4.1 Platform Choice

For this project, thousands of public repositories hosted on GitHub [24] were crawled. GitHub is one of the most popular code hosting sites, with more than 50 million public repositories [46]. GitHub repositories contain files relevant to specific projects, and can be flagged with an arbitrary primary programming language, whilst containing many files of potentially different languages.

Due to the large number of public repositories, crawling files from repositories exclusively from GitHub would not be a limiting factor for this project in regard to obtaining sufficient source codes for a large scale. Furthermore, GitHub exposes an API which allows iteration over public repositories [47]. A python library, PyGithub [48], provides typed interfaces to easily access this API, abstracting lower-level details.

It would also be possible to obtain files from GitHub by other methods, such as web-scraping, whereby files would be retrieved in an automated process without the API. This could allow for a larger source code corpus to be crawled, due to avoidance of rate limiting as discussed in 4.3.1. Whilst scraping is permitted within the GitHub terms of service [49], added complexity when compared to API usage was not deemed worthwhile, as a sufficiently large scale would still be attainable exclusively via API requests. Likewise, other repository crawling techniques such as repository cloning and file copying were considered, but decided against for the same reasons.

4.2 Choice of Languages

For this project, a crawler was created which would iterate over public GitHub repositories flagged with certain programming languages, and store the contents of specified file types. These files would be stored within a *source code table* created within the MongoDB database, in order to provide logical separation to components such as the index, which would be implemented afterwards whilst new source code was crawled concurrently.

Files of multiple programming languages were decided to be crawled and used for this project, as opposed to a single language. With a working plagiarism detection system, this would of course allow potential plagiarism in provided files of such multiple languages to be detected.

Furthermore, the use of multiple languages for this project would demonstrate the generality and extensibility of the final system, showing that any novel techniques or approaches are applicable not just for a single chosen language (for example by exploiting distinct language characteristics or idioms), but for an arbitrary language of a future researcher or developer's choice.

However, the use of a single language could allow for a more optimised final system, as a result of excluding generalizations which would be needed to support multiple languages. In addition, using files of a single language would allow for a larger 'vertical' scale to be demonstrated, as a consequence of being able to utilise the crawler solely for an individual language, thus obtaining more source code files of the same type within a given time frame.

Java, C, C++ and Haskell were chosen to search GitHub for, such that repositories flagged with any of these languages would have their files inspected. Within each identified repository, files with extensions of the four languages would be downloaded. Header files with the file extension '`.h`' would additionally be crawled, which are used within C and C++ projects to contain declarations shared between files. This was a practical decision, as this would allow entire C or C++ projects to be checked for plagiarism. From each repository, files with extensions of `.java`, `.c`, `.cpp`, `.hs`, and `.h` were therefore stored within the database's source code table[1].

Java, C, and C++ were chosen due to their popularity; they are well known languages which have been used within prior plagiarism detection systems [12][14], enabling relative evaluation at later stages. The ubiquity of these languages also makes their choice practical, as many situations where source code plagiarism detection is desired would use these languages, such as academic assignments. Actively maintained Python libraries also exist which can parse these languages [50][51], reducing unspecific work to the scope of this project. The syntax of Java and C/C++ is also relatively similar, such as certain keywords, logical operators, and inbuilt types. Thus, comparison between the languages may be possible, as will be discussed in section 8.5.

Haskell was chosen as it is a functional language, thus acting as an indicator of performance for different language paradigms, additionally illustrating the extensibility of the developed plagiarism detection system. However, there is no existing maintained Python library to parse or tokenize Haskell files, a challenge which will be discussed in section 5.3.4. Haskell is also a popular functional language within academia [52], therefore its choice is pragmatic. As mentioned within section 2.3.7, no research of large-scale plagiarism detection (or one incorporating online sources) has been conducted with Haskell.

4.3 Crawling

4.3.1 Repository Iteration

To crawl files from repositories hosted on GitHub, an access token is required, used to authenticate and enable API requests. For this project, a personal access token was generated. The primary bottleneck for this project when crawling is GitHub's API rate limiting, which for a personal access token limits requests to 5,000 per hour [53], or ˜83 requests per minute.

As mentioned, requests to GitHub's API would be performed via the PyGithub library, which provides the `search_repositories` [54] function. This interacts with the GitHub Search API [55], which utilises *query parameters* to filter results for a given *search request*. The specified 4 languages were therefore used as query parameters, allowing iteration of the appropriate repositories. However, the GitHub Search API only yields up to 1,000 repositories for a single search request. Therefore, multiple requests would be needed in order to obtain the desired scale.

[1] Additional C++ files with the extension of '`.cc`' were crawled, due to their relative ubiquity. No other different file extensions were additionally crawled. Across all crawled repository types, 87,474 C++ files with an extension of `.cc` were crawled, and are indicated as `.cpp` files within table 4.1.

A challenge this represents is the ordering of repositories which are returned from a search request. Ideally, repositories would be ordered by a distinct property, such as creation date, so that new search requests could offset the start date by the time delta of the last retrieved repository, thus ensuring iteration of unique repositories. However, the only sort options provided by the GitHub Search API are '`stars`', '`forks`', '`help-wanted-issues`', or '`updated`' [56], none of which can guarantee unique repository iteration. Moreover, the API does not provide the functionality to offset or skip search results, so for any provided ranking system, only the top 1,000 or fewer ranked repositories will be returned.

The implemented crawler therefore did not provide a sorting option, and instead only filtered results via the aforementioned language parameters, using the Search API's default "best match" ranking [55]. Upon investigation this was sufficient, as this ranking returned repositories in a seemingly partly random fashion, allowing the 1,000 or fewer results to be sufficiently different for each request. This did mean that repositories returned were not unique however, and so could already have been encountered beforehand. Therefore, the source code table would be checked for each repository by examining if any files from it were already present, and therefore whether the repository had already been encountered and so could be skipped.

As mentioned, GitHubs API rate limit is ~83 requests per minute. However, the rate limit employed when searching for repositories is different, permitting just 30 requests per minute [55]. Whilst this may not be a problem if each repository were unique (as the consequent delay from iterating through each repositories files would likely be sufficient), if many repositories were to be skipped in a row, then GitHub's rate limit could be exceeded. This would result in an exception and interrupt the desired program flow, potentially missing entire repositories. Furthermore, "excessively frequent requests" could potentially risk API suspension if repeated failed requests are made [57].

Accordingly, varying iteration delays were used for iteration of repositories and files. After performing rate limiting and post-request filtering to ensure unique repositories, iteration through the files of such repositories could be performed.

4.3.2 File Iteration

As discussed, Java, C, C++ and Haskell files would be stored for this project, in addition to header files common within C and C++ repositories. The GitHub Repository API [58] provides the ability to iterate over the contents of a repository, returning files and directories interspersed. When retrieving the desired files from a repository, directories would therefore be expanded via an additional API request, yielding the contained files in a recursive manner, whilst respecting the discussed rate limit. Upon inspecting each acquired file, those with the desired file types described in 4.2 would be stored in the source code table.

To store a file within the MongoDB source code table, conversion to an appropriate type containing only the necessary information was required, in order to minimize storage space. This would also ensure that future database tasks such as iteration would be as performant as possible, as extra contents within a requested database document would not have be transferred or filtered, an overhead which would be appreciable at this project's scale.

Many properties of a file are available via the GitHub API, such as a file's encoding, content, size, name, path, and various forms of URLs. One required property to be stored was of course a file's contents, which is encoded when retrieved. Another property chosen to be stored was a file's HTML URL, i.e. the human-readable URL which one may navigate to when browsing a repository, such as '`https://github.com/user/repo/readme.md`'. If a plagiarised file were to be identified, then providing the file's URL allows users to straightforwardly investigate other related files from the original online source. Furthermore, the URL contains the repository's name and

```
{
    '_id': ObjectId('627080fd99f409d97376ef65'),
     'content': 'package net.bytebuddy.test.packaging;\n\n'
                'public class MemberSubstitutionTestHelper {\n\n'
                '    private static String foo;\n'
                '}\n',
     'last_modified': 'Sun, 01 May 2022 18:35:01 GMT',
     'url': 'https://github.com/raphw/byte-buddy/.../TestHelper.java'
}
```

Figure 4.1: Example of an entry within the database's source code table.

author should they be required, preventing the need to additionally store them.

The date of each file's last modification was also chosen to be stored. As files may be modified after they have been crawled, presenting a user with the last known modified date could help to prevent misunderstanding, if discrepancies between source code versions or plagiarised content were to arise. The last modified date would also be helpful if a file from an online repository were to be plagiarised, and then modified or removed such that it is no longer available on the latest repository version, a challenge discussed in 4.3.4.

4.3.3 Source Code Entries

After the desired properties of each file had been obtained, they were stored within the source code table. A document containing the decoded contents, last modified date, and URL was constructed and inserted. This occurred individually for each file as opposed to using bulk operations, as the database insertion time was insignificant compared to the rate limiting delay. The risk of losing multiple files to be inserted due to unforeseen exceptions was also avoided with this approach. An entry within the source code table is shown in 4.1.

As mentioned in section 3.2.3, if a document is inserted to a MongoDB table without a provided unique ID, then one is automatically assigned within the '`_id`' key. Whilst this ensures each entry is unique within the source code table, it also allows use as an identifier within other system components such as the index, in order to reference file contents in a compact format.

The automatically generated `ObjectId` also contains a timestamp of when the document was inserted, within the first 4 bytes [59]. This could be displayed to users if potential plagiarism were to be identified from a file, informing them how recently the file contents had been checked. It also permits sorting or filtering by insertion date. The last 3 bytes of the ID are an "incrementing counter, initialized to a random value" [59], and therefore could be used for database partitioning, if required. Whilst the URL could have been used within the '`_id`' field as it is unique per file, for the reasons discussed, the addition of the MongoDB `ObjectId` was beneficial.

The crawler was run in a different process via a Linux screen [60], allowing concurrent development work whilst new source code files were obtained. Crawler output was logged for potential maintenance and analysis.

4.3.4 Code Upkeep

Continued modification or additions of code within crawled repositories is a challenge not addressed within this project; repositories were not re-crawled or checked for removal of code. In the case of code removal from repositories, keeping the code present in a plagiarism database may be useful, as plagiarism could still have occurred before the removal. Such an additive model of source code versions may therefore be beneficial, and is discussed in section 9.2.3.

4.4 Analysis of crawled repositories

In total, files from 7,208 repositories were crawled, with the number of different repository types crawled shown in table A.1. This resulted in a total of 3,393,994 files, including 1,324,892 Java files, as shown in table 4.1. As no additional query parameters were provided to the Search API other than the languages, the varying number of repositories and files crawled was originally reflective of their relative distribution on GitHub [46], with the number of Haskell files and repositories being significantly smaller, at ˜25,000.

Haskell repositories were therefore exclusively crawled for a period of time, in order to obtain a scale in the order of hundreds of thousands. As shown in table 4.1, ˜100,000 additional Haskell files were crawled, resulting in 130,409 in total, a far larger scale as required to meet the criteria outlined in section 3. Whilst crawling would have ideally obtained more Haskell files, as mentioned, the GitHub repository search API does not provide the ability to order repositories sufficiently. Therefore, after ˜1000 Haskell repositories had been crawled from an exclusive search request, minimal repositories were retrieved even on repeated requests. Crawling of Haskell files was subsequently suspended, as a sufficient order of magnitude had been obtained to demonstrate the developed system's performance on a functional language paradigm, whilst additional API requests provided diminishing benefit.

On average, each crawled repository contained 471 files, however this was heavily skewed by the larger repositories, as demonstrated by the median number of files per repository of 59, within table A.2. The distribution of the number of crawled files within repositories is shown in figure A.2; ˜75% of crawled repositories had between 10 and 1,000 files.

File types within repositories	Repository Type				All repositories
	Java	C	C++	Haskell	
Java (.java)	1,295,313	5,825	23,504	250	1,324,892
C (.c)	14,714	440,150	55,423	481	510,768
C++ (.cpp)	25,823	44,954	468,084	35	538,896
Header (.h)	46,757	380,143	461,465	664	889,029
Haskell (.hs)	174	65	59	130,111	130,409
Total	1,382,781	871,137	1,008,535	131,541	3,393,994

Table 4.1: Analysis of file types within crawled repositories

Across all crawled files, there were a total of 897,386,536 lines of code crawled, as can be seen in A.4. At this scale of nearly 1 billion lines of code, a large scale was deemed obtained, as desired. Within the crawled files, a total of 4,180,230,929 tokens were obtained, averaging ˜4.7 tokens per crawled line. However, the total number of lines includes whitespace and comments, so the actual number of tokens per logical line is higher. Figure A.3 shows the distribution of the number of tokens within each crawled file. As shown, ˜55% of files had between 100 and 1,000 tokens. ˜97% of files had between 10 and 10,000 tokens, with a median of 369 and mean of 1,256, as detailed in table A.5. This demonstrates how the presence of large files heavily skew the mean number of tokens, contributing a large number of tokens; the largest 1% of files contain ˜22% of the total number of tokens. In total, more than 31 billion characters were parsed from each file for this project, as shown in table A.3.

Chapter 5

Tokenizers

As mentioned in section 3.1, to effectively compare source code programs, conversion to an appropriate form is required. For this project, the stored program contents within the source code table will therefore be *tokenized*. Tokenization is the act of converting plain-text source code into a sequence of its individual tokens, referred to as a *token stream*. This token stream can then be used to generate a set of n-grams, which can then be compared against those of other programs.

5.1 Requirements

It is therefore required to have a reliable and consistent tokenizer for each language, ensuring that different source code files in one language containing identical code subsections will have those sections reflected within the resulting token stream. As no actively maintained Python library exists to tokenize Haskell, a Haskell library was used to perform tokenization, as discussed further in section 5.3.4. Within Python, the `pycparser` [50] and `javalang` [51] libraries provide tokenization of C/C++/Header and Java files, respectively. These can be used to parse the stored source code files, producing a token stream from each.

However, the resulting token streams are not fit for use within a plagiarism detection system. This is as the libraries tokenize programs literally, such that information required to produce a consistent program execution remains. For example, consider the following Java snippet:

```
String myStr = "Contents";                                   (5.1)
```

An example of the resultant token stream from this snippet could be:

```
StringType, Identifier("myStr"), Equals, String("Contents"), SemiColon
```

Whilst this makes sense in most tokenization contexts, for the detection of plagiarism, the exact value of tokens such as identifiers or strings is undesired, due to the ease of their modification. For instance, a student plagiarising the given snippet may rename '`myStr`' to '`thisString`' in an attempt to avoid detection, or change the case of the string's contents, such as making it all lowercase. These superficial program changes should not be reflected within the token stream, as their inclusion would hinder detecting plagiarism from an original source.

Therefore, resultant token streams for two programs should be equivalent, if superficial modifications are exclusively made.

As mentioned, generated token streams will be converted into a set of n-grams. Each resultant n-gram and the number of its occurrences in a given program can be compared against another.

However, if there is a sparse distribution of n-grams such that many n-grams occur in very few programs, then comparisons will become less effective in the case of plagiarism obfuscation. This is as the original n-gram distribution of a program before obfuscation can be more likely to drastically change, therefore impeding on the potential to identify such plagiarism. Hence, the possible set of n-grams (and consequently tokens) should be finite, with a reasonable upper bound in practice, in order to facilitate effective program comparisons.

5.2 Token Choice

Superficial changes which can be made in an effort to obfuscate plagiarism include modifications to comments and whitespace, in addition to renaming variables. Rewording, removing, or adding comments are an easy task when plagiarising code, as these changes do not affect program execution in any way. Whitespace modification can occur by inserting additional blank lines or moving braces to new lines, as well as entirely reformatting a file, whether by hand or with an automated tool, as discussed in section 2.1.1. Hence, the disregard of whitespace[1] and comments is also a desirable property, as with many other plagiarism detection systems [61].

Whitespace and comments were therefore ignored within tokenization, and so no additional tokens would be generated when they were encountered. Strings were assigned a single token, such that all strings would be treated equally, regardless of their content. Whilst this could reduce the ability to detect plagiarism from an identically copied program within a large corpus, it makes superficial plagiarism obfuscation techniques futile. Moreover, if strings were not mapped to a single token, then there would be a practically unlimited number of possible resultant tokens. As discussed, a finite set of tokens was required, justifying the use of a single token for strings. For the same reasons, identifiers, which are used for variable, class, and function names, were also assigned a single token. The Java snippet in 5.1 would therefore be tokenized as:

```
StringType, Identifier, Equals, String, SemiColon
```

Numeric constants such as '`100`' or '`0xABC`' were chosen to be assigned to the same token as identifiers. Whilst the need for mapping numeric constants to a single token is the same as for strings, the choice of using the same token as identifiers provides additional benefits. Primarily, it mitigates simplistic forms of plagiarism obfuscation such as constant inlining or outlining, as the token stream will stay the same locally. Consider the following Java snippet:

```
int x = 2 + 3;                                  (5.2)
```

One could attempt to obfuscate plagiarism of this snippet by outlining constant values, for instance by re-defining the value of `3` within a variable called `three`:

```
int x = 2 + three;                              (5.3)
```

In this case, the token streams of both snippets 5.2 and 5.3 would be identical, at least locally. Whilst this modification could still introduce additional tokens from new constant declarations, this effect would be limited. The reuse of existing constants would introduce no additional tokens, and using the same outlined constant in multiple following statements would have a minor effect on the overall token stream. Most importantly, the resultant n-grams generated from the snippets would be contained in both program's total n-gram set. This ensures productive program

[1] Whilst some languages like Python do utilize whitespace, the tokenization process for such languages would be similar. Whitespace could be treated as an individual token, still ignored, or tabs/spaces could be regarded as tokens. As whitespace is extraneous to the languages within this project, its disregard is justified.

comparisons, and hinders this form of plagiarism obfuscation purely from the choice of tokens. Likewise, constant inlining is also mitigated, for example replacing the mathematical constant `Pi` in the expression '`2*Pi*radius`' with its constant value.

Whilst strings could also have been assigned the same token, it was decided to keep their token distinct. This is as numeric constants can occur within compound statements much more frequently, possibly due to the higher number of inbuilt arithmetic operators than string operators within most languages. For example, if the expression '`(4 - 3) / 2`' were to be tokenized without the same token for constants and identifiers, then the resultant n-grams would be vastly transformed if constant inlining or outlining were to occur. Strings are likely to be included in fewer compound expressions, and so were kept as their own token. Furthermore, if strings were to be assigned to the same token as identifiers and numeric constants, then this may entail too much generality, becoming unproductive for plagiarism detection. This is an area that could be expanded upon in future work.

5.2.1 Utilisation

The choice of which tokens to actually utilise is also not necessarily obvious. For example, braces within a program can often be easily modified, such as removing the braces of a single line 'if' or 'for' statement. Thus, ignoring braces during tokenization could be beneficial in preventing such modifications affecting plagiarism detection. Likewise, semi-colons can be extraneously added which can disrupt token streams. A statement ending with multiple semi-colons is valid, as empty statements are considered to be in-between each semi-colon. Yet, such additional tokens would also disrupt token streams, potentially hindering plagiarism detection if consistently performed. Exclusion of braces and semi-colons is a consideration for future work, but was not performed within this project.

5.3 Implementation

As discussed, the use of existing libraries enables a token stream to be obtained from given source code. However, processing the resultant individual tokens would be impractical, as the generation of n-grams requires an input sequence of fixed-length elements. In addition, when comparing the n-gram distribution of two programs, the actual source code tokens are not required, merely equal and distinct values for comparison.

Therefore, generated token streams would be transformed into a character stream, permitting performant and space-efficient representations of programs. Each language's tokenizer would consequently consist of three primary stages:

1. Generate a token stream for an input program, ignoring extraneous content.

2. Convert the tokens to appropriate forms, e.g. mapping strings to one token.

3. Transform the resultant token stream to a deterministic character stream.

5.3.1 Character Mapping

In order to convert token streams to character streams, a unique and deterministic mapping was required. It was therefore decided to enumerate the possible characters within Python, using the `chr` (char) function. For example, `chr(97) = 'a'`, and `chr(98) = 'b'`. As these characters

would be exclusively used internally and not output to users, they were not required to be human-readable[2]. Hence, the chosen enumeration used an incremental *char counter* to determine the next character to be assigned to a token, which was initialised to 2. This is as `chr(0)` and `chr(1)` were reserved for alphanumeric tokens (identifiers/numeric constants) and strings, respectively. This was common for each language used within this project.

The token-to-character maps (*token maps*) were chosen to be populated as new tokens were encountered. This is as there are hundreds of possible tokens within each language, and hard-coding corresponding values would have been impractical. Certain tokens were undocumented within the respective parsing libraries such as '>>>=' within C++, or novel within program use, such as '<>' within Haskell, as discussed further in section 5.3.4.

However, populating token maps during tokenization presented particular challenges. For a new tokenizer instance, tokens could be encountered in a different order than previous instantiations, but would still be required to map to the same character in order to preserve determinism. As a consequence, persistent storage within the MongoDB database was necessary, and so the token maps were stored within a new database table.

The translation of token streams to character streams therefore utilised a persistent map for each language. Upon encountering a token not present within the existing map, tokenizers would obtain a new character to map it to, by incrementing the aforementioned char counter and storing the resultant character. This process updated the token map and char counter within the database each time a new token was encountered, as the overhead was small, whilst ensuring persistence and consistency. Furthermore, this process only needed to be performed a relatively small finite number of times, in practice only regularly for the first few tokenized programs.

Each new instance of a tokenizer therefore fetched the language-specific token map from the MongoDB database into memory, ensuring deterministic and performant tokenization. If the token maps were to be queried within the database for each token in a program, then high access overhead times would be incurred over many programs. A complication faced was that MongoDB does not permit the '`.`' or '`$`' symbol to appear within a map's key, or the '`$`' symbol within a map's value. Therefore, substitutions were made within tokens to replace these symbols upon obtaining or storing the token maps. Similarly, when a new character was obtained via the char counter, the '`$`' symbol was skipped.

5.3.2 Java Tokenization

As mentioned in section 5.1, the `javalang` library [51] was used to obtain an initial token stream for Java programs. This library ignores whitespace and comments by default, so no additional parsing was required in this regard. The tokens were then parsed, such that those representing strings, identifiers, or numeric constants were mapped to the predetermined respective tokens. Any other tokens were then mapped to a unique character, as described in section 5.3.1.

Not every crawled program was able to be tokenized, however. While not requiring compilable code in order to be tokenized, syntactically correct tokens are needed. For example, the provided program sequence '`int x +`' can be tokenized, whilst the sequence '`x = "string`' cannot, due to the presence of an unterminated string.

Therefore, crawled files containing syntactically invalid tokens would fail tokenization. In addition, the presence of bugs or unsupported language features within the tokenization library may cause it to be unable to parse a given program. Certain crawled files may also contain sections intended to be parsed and removed by a bespoke preprocessor, therefore failing to be tokenized.

[2]In initial development, printable characters were exclusively used to aid in testing, but after token-to-character mappings were confirmed to correctly function, this was no longer required.

These issues are common across the tokenizers for all languages, but in total affect a negligible number of programs; across all crawled Java files, only ~0.1% failed to be tokenized. Therefore, no actions were taken to remedy untokenized files. Yet, this could present an issue for provided input programs to the complete plagiarism detection system, as they too require tokenization. Such programs may be unlikely to contain esoteric features or syntactically invalid tokens, but if they do then would be able to receive additional attention by users to address any issues, due to their relatively small total number.

5.3.3 C/C++ Tokenization

The `pycparser` library [50] was used to tokenize C, C++, and their respective header files. Whilst this library does not explicitly support C++ files, library updates to support certain C++ features are apparent, and in practice ~99% of C++ files were successfully parsed.

However, `pycparser` expected preprocessed files, and does not ignore comments, with no option to do so. Therefore, the stored source code files would need to be preprocessed. This was chosen to be performed without the C preprocessor, `cpp` [62], despite its recommendation by `pycparser` . This is as `cpp` is intended for use within a project. As only a single file at a time would be provided to the preprocessor when tokenizing programs from the source code table, `cpp` would therefore fail if any headers or external files are referenced, as is common in most files. Whilst it is technically possible to retrieve the required files for each program from the source code table, it would be extremely impractical to do so unless collating and tokenizing the files as they are crawled. As tokenization for this project was performed on pre-crawled files, it was chosen to implement the most necessary preprocessing techniques.

Therefore, before code was tokenized with `pycparser`, all comments were removed, in addition to end-of-line backslashes which are used in C/C++ programs to divide statements over multiple lines. A noticeable portion of files also failed due to `pycparser` being unable to handle byte-order marks [63] at the start of files, so these were also filtered.

After tokenizing the preprocessed file, the tokens could be mapped to characters in the same manner as the Java tokenizer, thereby producing a deterministic character stream. The C/C++ token map was chosen to be instantiated with the same corresponding characters and tokens as Java. This was possible as Java programs were tokenized first, and would potentially allow for inter-lingual plagiarism to be identified, as will be covered in section 8.5. Whilst a higher percentage of programs may have been able to be tokenized using the C preprocessor, the vast majority of programs were still tokenizable. The percentage of successfully tokenized programs was 98.9%, 98.7% and 96.1%, for C, C++ and Header files respectively, resulting in an insignificant number of untokenized files from the source code table, as shown in table 5.1.

5.3.4 Haskell Tokenization

As mentioned, no Python library currently exists to tokenize Haskell programs, likely due to Haskell's relative lack of popularity when compared to the other languages used within this project. Therefore, a Haskell program was created, utilising the `Language.Haskell.Exts` [64] module to tokenize programs. This would receive Haskell source code as an input argument, and output the generated token stream. A Python tokenizer file specific for Haskell could then be created as with the other languages, which ran the Haskell tokenizer in a subprocess and retrieved the output. This permitted parsing of the resultant token stream in a similar fashion to Java and C/C++, allowing the same language-generic tokenization code to be reused. Minimal time overhead is incurred when utilising a subprocess, facilitating performant tokenization. However, memory overhead can be notable due to the use of *fork* operations [65], as will be discussed

further in section 6.3.6.

As the token stream generated within the Haskell tokenizer was transferred via text, no definite sequencing was encoded. Whilst the token stream was comma-separated, certain tokens included commas within their syntax, causing issues when parsing the resultant stream. Therefore, the token stream was preprocessed within the Haskell program to be separated by a unique identifier. This was then used within the Python program to split the received token stream, yielding each token as a string.

Before conversion to a character stream, an additional processing stage was performed to obtain appropriate tokens. Within the Haskell tokenization library, operator symbols were not necessarily included within the set of recognised keywords, unlike the Java and C libraries. Instead, most were tokenized to a separate token, containing the symbol within a string. Certain operators like '!' or '–' were recognised keywords, whilst others such as '==' and '>' were not. This is as it is possible for users to define their own operators in Haskell, therefore utilising and possibly overloading symbol combinations such as '.=' or '<>' as operators [66].

These operators can still be kept for plagiarism detection comparisons, as a way to selectively compare Haskell programs. However, the alteration or redefinition of such symbols is possible, allowing an operator consisting of new symbols to reference another. One could therefore define a new operator as another, and replace all instances of the latter operator with the new one in an effort to obfuscate plagiarism.

Therefore, operators composed of 3 or more symbols were mapped to the same character, as was done for strings. Operators composed of 2 symbols or fewer were still assigned unique characters, as these included many popular symbols which may be less likely to be overwritten, whether due to convention, or their pre-existing use within other program sections. These included operators such as '+', '*' and '$' (Haskell's ubiquitous function application operator [67]). This is a topic which could be researched further in the future.

Whilst identifiers and function names can be modified, certain common library functions may be less likely to be modified, for example the *map* function. Retaining certain identifiers can therefore be appropriate, as with Holmes [30]. Identifiers from the popular Haskell 'Maybe' module were therefore kept [68], and future work could expand on determining other identifiers to retain.

After the appropriate tokens were extracted from the Haskell program's provided token stream, the aforementioned mapping of strings and identifiers could occur, before all tokens were mapped to characters, and the character stream output. In total, 90.2% of Haskell programs were able to be tokenized.

File Type	Tokenization		
	Failed Programs	Total Programs	Success
Java	1,506	1,324,892	99.89%
C	5,425	510,768	98.94%
C++	7,099	538,896	98.68%
Header	34,273	889,029	96.14%
Haskell	12,788	130,409	90.19%
All files	61,091	3,393,994	98.20%

Table 5.1: Analysis of the success rate of tokenization per language.

5.4 Obfuscation Prevention

As mentioned, comments and whitespace are ignored within tokenization, whilst identifiers, numeric constants, and strings are mapped to specific characters. This enables superficial forms of plagiarism obfuscation to be neutralised simply via tokenization. For example, consider the following simplistic Java code snippet:

```
public class Incrementer
{
    /* Increments the provided integer value */
    public int increment(int value)
    {
        return value + 1;
    }
}
```

Figure 5.1: A simple Java class which can increment values.

If one were to plagiarise 5.1, they could remove the comment and reformat the file, in addition to renaming each identifier and swapping the order of the addition statement, as shown in 5.2:

```
public class MyClass {
    public int addOne(int number) {
        return 1 + number;
    }
}
```

Figure 5.2: A plagiarised version of figure 5.1.

However, as a consequence of the tokenization process, the resultant token and character streams of 5.1 and 5.2 are completely identical. This demonstrates how superficial plagiarism obfuscation techniques can be prevented from working.

5.4.1 Untokenized code

Whilst easily modifiable source code constituents such as comments are undesired when comparing two programs and therefore untokenized, they can still aid in plagiarism detection. If a plagiarism detection system were to identify a potential original plagiarised source, then identical or very similar whitespace, comments or identifiers would serve as practical forensic evidence. Therefore, such constituents are still beneficial after detailed analysis has occurred, as users can be presented with similar code sections, allowing them to visually compare such elements between their provided input file and an identified potential source.

Chapter 6

Candidate Retrieval

With a populated source code table and the ability to tokenize each program, the required infrastructure to check an input program (*source*) for plagiarism from stored files was complete.

At the scale of millions of source code files containing almost one billion of lines of code, linear iteration through each file to compare and check for plagiarism is both infeasible and unscalable. Therefore, an index was used, such that only stored programs which shared a common section of the input source would be compared, drastically reducing the number of programs needing to be compared against a source.

6.1 *N*-grams

As discussed, an index is effectively a key-value store, consisting of *n*-gram keys, whereby each corresponding value comprises of all programs which contain the *n*-gram, and the number of that *n*-gram's occurrences within the program. For example, if an index was generated using 4-grams, and the 4-gram 'ABCD' occurred in 3 programs, then the 'ABCD' entry in the index could be:

$$\texttt{`ABCD':\ \{`X':3,\ `Y':1,\ `Z':5\}} \tag{6.1}$$

The entry shown in 6.1 indicates that the 4-gram 'ABCD' occurs within program 'X' three times, program 'Y' once, and program 'Z' five times.

These *n*-gram frequencies can therefore be used to compare arbitrary programs, based on their *n*-gram frequency distribution. For example, if an input program 'A' contained the 4-gram 'ABCD' five times, then it would be the most similar to program 'Z', at least for this specific *n*-gram. As programs contain many *n*-grams, this process can be repeated for each of the *n*-grams within the input source 'A', thereby producing an overall similarity across all relevant stored programs.

Whilst the use of an index containing *n*-grams is more performant and scalable than iteration through stored files, it is not as selective as a pairwise comparison could be. Therefore, the index is used to obtain *candidates* for a subsequent detailed and more selective stage. The purpose of candidate retrieval is therefore to reduce the corpus of stored code to a reasonable size, with the goal of including any possible plagiarism sources within this subset. From this smaller corpus, detailed analysis can be performed, resulting in the final identified sources of potential plagiarism.

6.1.1 Suitability

The use of *n*-grams offers many benefits for plagiarism detection. As *n* is generally a small value such as 6 or less, an *n*-gram reflects a local section of the program it is generated from, typically at most a line. Therefore, the entire distribution of a program's *n*-grams represents a global overview of the program.

Due to *n*-gram's locality, they are versatile with regards to plagiarism obfuscation. Whilst one section of plagiarised code could be heavily modified, any unmodified sections will still reflect the local *n*-gram distribution of the original source. Such sections can therefore provide sufficient similarity to the original file during comparison, allowing it to be retrieved as a candidate.

For a completely plagiarised program, consistent significant modification of tokens is required to alter each *n*-gram and therefore successfully obfuscate plagiarism. For a program containing ℓ tokens, at least $\lfloor \frac{\ell}{n} \rfloor$ token modifications would need to occur, and no more than n tokens apart from each other. Such modifications would also have to be structural and not superficial, due to the tokenization techniques already employed, as described in section 5.4. In practice, such consistent token modifications are not always possible, as replacing certain tokens with others is not trivial; unlike within text-based plagiarism, synonyms are not a frequent concept within language syntax.

Within source code, a logical atom to perform one certain task can consist of many tokens, such as an if-statement or small function. Hence, even if such atoms are rearranged whilst preserving program semantics, the tokens and therefore *n*-grams within such atoms would be retained. For example, swapping the order of functions would have a negligible effect on the overall *n*-gram distribution, as only $n-1$ *n*-grams would be affected per code section's boundary. Thus, broad rearrangement of a file's contents is ineffective; modification to the tokens within logical atoms is needed to successfully obfuscate plagiarism, which is an inconsequential and potentially time-consuming task.

As a result of these discussed factors, those attempting to obfuscate plagiarism would likely have to invest a significant amount of time into performing such obfuscation. However, as discussed in section 2.1.1, this is likely not possible for those performing plagiarising source code, as plagiarism can often arise as a result of time constraints. This warrants merit to the use of *n*-grams when examining largely plagiarised programs.

6.1.2 Drawbacks

Whilst comparing programs via *n*-grams provides several advantages, they do not offer a perfect solution. Due to their locality, if a smaller plagiarised section were to be contained within a much larger original program, then the *n*-grams from the plagiarised code could be obscured. This is as the larger portion of novel code would contribute many more *n*-grams, disguising the plagiarised segment. However, the plagiarised section would have to be notably short, relative to the entire program, as will be explored further in section 8.4. This could also be mitigated by different comparison methods, as discussed in section 9.2.2.

As mentioned, only programs sharing *n*-grams will be compared to a provided source, when querying the index. However, due to the sizeable number of large programs as can be seen in figure A.6, such programs would likely contain many distinct *n*-grams as a result of their size. Large programs would therefore be compared to most queried programs, introducing a possible source of noise and potentially obscuring an actual source of plagiarism from being retrieved as a candidate. This issue will be addressed in section 6.5.4.

Furthermore, as mentioned, *n*-grams are not as selective as a pairwise comparison of input source and stored files could be. For example, if an extended section of code were to contain frequent different tokens such that the *n*-grams were different, then comparison via *n*-grams would be difficult. However, a different approach such as *local alignment*, an approximate string matching technique, would potentially be able to identify a match as a result of considering mismatches [10]. Albeit, local alignment may not be scalable to the same degree as *n*-gram indexing.

Obfuscation techniques modifying significant structural code would impair comparisons via

n-grams. For example, converting a 'for' statement to a 'while' statement and vice versa would disrupt the local *n*-grams. Such changes compounded with others discussed may result in an actual plagiarised file not being retrieved as a candidate. Despite this, as mentioned in section 6.1.1, such consistent modification would likely be time-consuming in practice.

As *n*-grams are generated from a language's set of possible tokens, the detection of inter-lingual plagiarism may be challenging, for example detecting a C program translated to Java. However, for certain languages there could be methods to account for this, as will be further explored in section 8.5.

A last drawback of using *n*-grams is deciding upon a value of *n*. Whilst not necessarily an issue, the choice of *n* is an important hyper-parameter for the entire plagiarism detection system, and can completely change the system's efficacy. Yet, this choice is mitigated by producing a system capable of handling arbitrary *n* values. Different *n* values can therefore also be used and evaluated, allowing for an extensible system to handle different use cases.

6.1.3 Effect of *n*

As an *n*-gram is generated from a contiguous section of *n* tokens from a program, the varying values of *n* can greatly affect the ability to compare one program to another. In general, the higher the value of *n*, the more unique each *n*-gram becomes, as a larger code section is taken into account. For example, if the value of *n* was equal to the number of tokens within a program, then only one *n*-gram could be generated, and so would only match exactly that provided program.

In general, the number of possible *n*-grams is t^n, where t is the number of distinct tokens. For example, if a language comprises of 100 possible tokens, then there would be 100^3 possible 3-grams. Therefore, an increase *n* by 1 increases the number of possible *n*-grams by a factor of *t*. However, in practice, the number of *n*-grams generated for a given *n* is remarkably less than t^n. This is as many token permutations are unlikely to actually occur, such as mixtures of braces and semi-colons which are syntactically valid but unrealistic, for example '`{{;};}`'. Furthermore, many token permutations are indeed syntactically invalid, and so the *n*-grams corresponding to such permutations would not be generated.

Table A.6 shows the disparity between the theoretical and actual number of *n*-grams. For Java programs utilising 103 tokens, there could be $103^5 \approx 11.6$ billion 5-grams, but only ˜200,000 have been observed over ˜1.3 million programs, just ˜0.02% of the theoretical limit. Whilst the number of distinct *n*-grams would be expected to grow as the scale increases and more programs are obtained, it would still be far lower than the theoretical limit. This is crucial for scalability and performance, as number of total *n*-grams affects how many entries are required

6.1.4 Choice of *n*

Whilst higher *n* values offer higher distinction as a consequence of fewer occurrences, they are also less resistant to plagiarism obfuscation. This is as for each modified token, up to *n* *n*-grams could be altered. In *Burrows et al*'s 2007 paper utilising *n*-grams for plagiarism detection, a value of 4 was chosen [10]. The value of 4 was therefore used as a starting point for experimentation.

For initially determining which *n* values to consider, a sample of 10,000 Java programs was used to construct similarities for 10 randomly chosen programs within the sample. Queries were performed using *n* values between 2 and 9, observing the consequent distribution of similarities, in addition to the number of resultant *n*-grams. As candidate retrieval is concerned with retrieving the most similar programs, the 99th percentile of program similarity was primarily observed.

As shown in table A.7, 2 and 3 grams had very high similarities, even at the 99th percentile. This suggests that many programs were indistinguishable from each other using such *n*-gram

values, which may become worse at larger scales due to additional noise. Likewise, the higher n values of 8 and 9 had notably low similarities at the 99th percentile. Whilst this benefits identification of unmodified plagiarised code, it demonstrates how alteration of n-grams within a program will significantly lower similarities quickly, and therefore would likely not offer sufficient distinction from other programs.

Furthermore, table A.7 shows the number of n-grams generated from the 10,000 programs, which follow an exponential distribution. As the higher n values have exponentially more n-grams, the time taken to index and iterate through such n-grams would also increase exponentially. Therefore, whilst the plagiarism detection system would be developed to handle arbitrary n, values between 4 and 6 were chosen to primarily investigate and utilise, due to their relative selectivity whilst being able to handle token disruption, and the number of resultant n-grams.

As mentioned, plagiarism obfuscation resulting in token modifications would disrupt 6-grams to a higher degree than 4-grams. However, the greater selectivity of 6-grams could benefit plagiarism detection for longer unmodified sections. Of course, 5-grams could offer a suitable middle-ground when considering such trade-offs. Therefore, evaluation would be performed for each and a suitable default choice could be chosen, whilst facilitating the combination of n-grams during comparison, if desired.

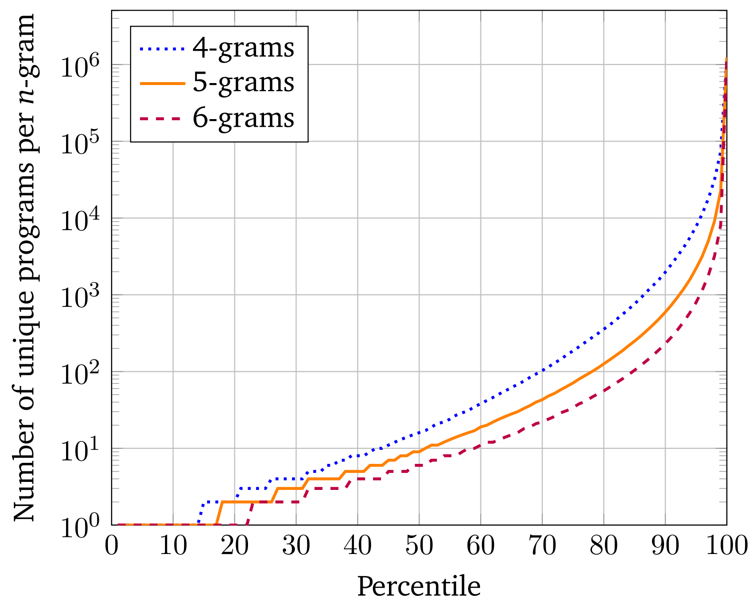

Figure 6.1: Percentiles of the number of distinct Java programs containing the same n-gram

6.1.5 Distributions

The n-gram distribution of each stored program therefore needs to be computed, in addition to any sources provided for plagiarism detection. As discussed, n-grams are generated by moving a sliding window of length n over a sequence of fixed-length elements, such as a character stream. For example, the n-gram distribution of the Java function shown in figure 6.2 will be generated.

```
public void printArray(int[] array) {
    for (int i = 0; i < array.length; i++) {
        System.out.println(array[i]);
    }
}
```

Figure 6.2: A Java function which prints the values in an array.

This function consists of 41 tokens, of which 16 are unique. When tokenized with the Java Tokenizer described in section 5.3.2, the character stream shown in figure 6.3 is produced[1]:

```
PVN(I[]N){F(IN=N;N<N.N;N+){N.N.N(N[N]);}}
```

Figure 6.3: Character stream of figure 6.2.

When generating n-grams for a given sequence of length ℓ, there are $\ell - n + 1$ resultant n-grams. Therefore, if we were to generate 3-grams from the character stream in 6.3, we would obtain 39 3-grams:

```
PVN  VN(  N(I  (I[  I[]  []N  ]N)  N){  ){F  {F(  F(I  (IN  IN=
N=N  =N;  N;N  ;N<  N<N  <N.  N.N  .N;  N;N  ;N+  N+)  +){  ){N
{N.  N.N  .N.  N.N  .N(  N(N  (N[  N[N  [N]  N])  ]);  );}  ;}}
```

Figure 6.4: Generated 3-grams of figure 6.3.

Of these 39 3-grams, 36 are unique, as the 3-gram '`N;N`' occurs twice, whilst '`N.N`' occurs three times. The resultant 3-gram frequencies are therefore shown in figure 6.5:

```
{
  'PVN': 1,
  'N.N': 3,
  'N;N': 2,
  '[]N': 1,
      ...
}
```

Figure 6.5: Abridged 3-gram frequencies of figure 6.4, and therefore program 6.2.

The 3-grams shown in figure 6.5 each represent a short section of the program within figure 6.2. The entire set of 3-gram frequencies therefore acts as a 'fingerprint' for the program. From an abstract perspective, this fingerprint for an input source is what is compared against those of stored files, in order to retrieve the smaller desired set of candidates.

6.1.6 Properties

For two programs P and Q, the sum of their individual n-gram distributions is effectively equal to the n-gram distribution of their concatenated contents. This is as their n-gram distributions will sum independently, with the exception of the $n-1$ n-grams at the 'border' of concatenation. We therefore have $\text{n-grams}(P) + \text{n-grams}(Q) \approx \text{n-grams}(P + Q)$; summing n-gram distributions is *additive*.

This means that if a code section is plagiarised within a larger source file, the n-gram distribution of the independent plagiarised section will be a subset of the overall source file's n-gram distribution. Consequently, identification of plagiarised subsections is feasible. Likewise, concatenation of multiple plagiarised sections will largely retain the n-gram distributions of each, if the overlap of n-grams is not significant. This is as the n-gram distribution of the entire source can effectively be decomposed into the individual distributions of each file, when comparing n-grams in a pairwise manner.

[1]For clarity, the generated characters were substituted with printable characters as some were unreadable, in addition to the same character as token being used where possible.

6.2 Index

The process described in section 6.1.5 can be repeated to generate *n*-gram frequencies for each stored file. As discussed, an index can be created by mapping each *n*-gram to its corresponding program frequencies. It is imperative this index has a constant access time, so that retrieving the associated program frequencies for an *n*-gram takes the same amount of time, regardless of how many *n*-grams or programs are indexed. This facilitates performant and scalable index queries. If this were not the case, the time complexity of the entire candidate retrieval algorithm would be multiplied by a factor of N, where N is the number of *n*-grams indexed. This would of course be a vastly undesired consequence, greatly hindering the scalability and performance of candidate retrieval.

6.2.1 Access Times

A data structure with a constant access time was therefore required to use for the index. As initially presented in section 3.2.3, this was one of the factors for choosing MongoDB. MongoDB documents within tables possess an ID field, which can be used to access the corresponding entry in logarithmic time, due to MongoDB's tree-based structure [45]. Despite this structure providing a logarithmic access time, in practice a constant access time is obtained. This is because the number of children per node is 8192, hence accesses via a specific key are logarithmic with a base of 8192 [69].

This means that if a table contains 8192 or fewer documents, the entire table will be stored within a single level of the tree. Likewise, for 8192^2 or fewer documents, only two levels are required. 8192^2 is 67,108,864, which is far greater than the unique number of *n*-grams which will need to be stored in an index, as discussed in section 6.1.3. Consequently, only two levels would ever need to be traversed, and so a constant access time is obtained for accessing documents corresponding to keys.

Therefore, constant access time is procured for a MongoDB table with *n*-grams as IDs and program frequencies as corresponding values. Hence, such a data structure is sufficient for the index used within candidate retrieval.

6.2.2 Program Similarities

As outlined, the goal of candidate retrieval is to reduce the total dataset of stored source files to a much smaller subset, whilst retaining all possible files which may have been plagiarised from. Therefore, a concept of ranking is required, in order to decide which programs to keep in the resulting candidate set.

To illustrate how program similarity is measured using an index, consider the following small index consisting of three 3-grams:

```
{
  'ABC': {'X': 1, 'Y': 4},
  'CFH': {'X': 3, 'Y': 3, 'Z': 1},
  'DEF': {'X': 2, 'Z': 4},
  'DGG': {'X': 4},
}
```

Figure 6.6: An example of an index generated using 3-grams.

As discussed, the index shown in 6.6 could have been generated by iterating through the programs `X`, `Y` and `Z`, calculating their respective 3-gram frequencies, and updating the relevant

entry within the index.

To query a provided program `P` in the index, we would first generate its 3-gram frequencies, such as `{'ABC': 3, 'CFH': 2, 'DEF': 1}`. Across each of `P`'s 3-grams, we would retrieve the corresponding programs, and assign a score based on how many 3-grams are shared. For example, the 3-gram '`ABC`' would assign a score of 1 to program `X`, and 3 to `Y`.

Note that `Y` only received a score of 3, as no program can contribute a higher score than the query program itself; the score is capped. This is because a program with a higher frequency of *n*-grams than the queried program does not necessarily mean it is more similar, as in this case a very large program with many *n*-grams would be mistakenly classified as 'similar' to most input sources.

Repeating this process for each 3-gram within `P` yields the similarity scores of `{'X': 4, 'Y': 5, 'Z': 2}`. The program frequencies corresponding to the 3-gram '`DGG`' are never queried, as `P` does not contain this 3-gram.

In order to produce similarities, each program's score is divided by the total number of 3-grams within `P`, obtaining values between 0 and 1: `{'X': 0.67, 'Y': 0.83, 'Z': 0.33}`. This is as if `P` was indexed within the index in 6.6 before querying, then it would have received a score of 6 due to each 3-gram entry being a perfect match, therefore resulting in a similarity of 1.

If the candidate set were to be retrieved by taking the top 2 similar programs from the index, then the resulting candidates would be `{'X', 'Y'}`. This demonstrates how the entire initial set of programs can be reduced in an efficient manner. Resultant processing could then be performed on `X` and `Y` to determine if plagiarism had likely occurred, as will be discussed in section 7.

6.3 Index Generation

An index needs to be populated with *n*-grams and their respective program frequencies. As these *n*-grams depend on the language they are generated from, there will need to be independent indexes for each source code language. In addition, separate indexes will be required for different values of *n*. Therefore, there will be several distinct indexes, each providing querying capability for different source languages and choice of *n*, as discussed in 6.1.4.

An *index generator* was therefore created, which would perform the following steps:

1. Iterate through the specified files within the source code table.

2. Tokenize the given program to a stream of characters.

3. Generate *n*-gram frequencies from the character stream.

4. For each *n*-gram, update the respective entry within the index to include the given program's frequency, creating the entry if not present.

After these steps have been performed, an index will have been generated for a given language and *n*, enabling querying of new programs. Index generation was initially performed for Java files, using 5-grams.

However, this initial approach encountered challenges as a consequence of scale, both in regard to the number of source code files, and the size of resultant indexes. Therefore, various techniques and optimisations were used to ensure that index generation would be scalable and performant.

6.3.1 Source Code Iteration

One challenge initially encountered was the iteration of the MongoDB source code table. Iteration of MongoDB tables is performed via a *cursor*, which provides the ability to access documents. MongoDB returns the documents in groups, in order to minimize data transfers. However, cursors can time out after a period of inactivity, which can arise if processing a certain group takes too long [70]. Therefore, an offset and limit were used, such that a bounded number of programs were indexed with one cursor. After this batch had been indexed, the process could repeat, with the offset incremented by the chosen limit. The batch size used was between 20,000 and 40,000, depending on the speed of each language's tokenizer.

As such challenges interrupted steady indexing, the use of an offset also allowed index generation to resume from a certain point. Throughout the entirety of index generation, *logging* was used, so that debugging output could be permanently stored, allowing for smoother development and subsequent analysis. Hence, any interruption of index generation could be addressed, and later resumed to re-process minimal files, utilising the offset.

Furthermore, as described in section 4.3.3, new source files were being concurrently crawled during development. Therefore, if index generation were to exhaust the currently stored files, then indexing could resume from the last indexed program after the source code table had been further populated.

6.3.2 Bulk Writes

However, as index generation progressed, the rate of file processing drastically decreased. This was as index updates took progressively longer, due to the larger number of n-grams indexed, as well as the amount of program frequencies which each n-gram mapped to. Continuing with the existing approach would have been unscalable and exceedingly time-consuming. Accordingly, *bulk writes* were used.

Within each batch, a local map was instantiated. This map was then modified in the same manner as for the index beforehand, such that each program's n-gram frequencies within the batch corresponded to the particular n-gram within the map. At the end of each batch, this map's n-grams would be iterated, and the corresponding values written to the persistent index within the MongoDB database. This map was effectively a batch-sized index within main memory. Consequently, updates to the local map were significantly faster, in addition to reducing the overhead required to constantly perform database operations. Thus, the resulting updates were more performant, whilst also enabling a large-scale index to be pragmatically generated.

MongoDB provides the ability to execute write operations on a single database table in bulk [71]. Therefore, bulk writes were used, instead of sequentially iterating through the local map and updating each n-gram's program frequencies individually. An array of the required writes was therefore provided to MongoDB, in addition to specifying the bulk writes as unordered. This allows MongoDB to parallelize the execution of each operation, as well as only requiring a single database request per batch. As the number of n-grams within a single batch can be in the hundreds of thousands for larger n values, the bulk operation facilitates scalable index generation. Hence, the number of database writes per batch was reduced from potentially more than 100,000 to 1, in addition to the writes executing in parallel.

6.3.3 Partitions

Despite index generation performing well with bulk operations during the creation of the 5-gram Java index, no more updates were able to be processed after ˜600,000 files. This was due to MongoDB's document limit of 16MB being reached [72]; for each document within a MongoDB

table, it cannot exceed 16MB in size. With regards to indexes, each *n*-gram and its corresponding program frequencies is a document, for example `{"ABCD": {"X": 3, "Y": 1, "Z": 5}}`. Therefore, such entries were containing enough program frequencies to exceed the 16MB limit.

As described in section 4.3.3, if no explicit ID is provided to a document, then MongoDB automatically generates an ID which includes a date and incrementing counter. Whilst the *n*-grams are used as IDs within the index, within the source code table each program is assigned an ID, which is then used to refer to it within the index. The aforementioned index document would therefore actually look like `{"ABCD": {627080fd99f409d97376ef65: 3, ...}}`. The last 3 bytes of the ID are the incrementing counter, which is cyclic, locally to each database table. Consequently, the parity of the program IDs within the source code table are uniformly distributed.

In order to avoid the 16MB document limit imposed by MongoDB, each index would be *partitioned*, such that the overall index would be composed of several smaller tables. It was chosen to perform this partitioning based on the ID of a given program, due to their described uniform distribution. This meant that each program would be assigned to a given index partition based on its ID's counter value, which would result in evenly sized partitions, each capable of being queried independently.

It was chosen to create 8 partitions for each index, primarily due to the usage of a 4-core CPU on the development VM. Furthermore, 8 partitions would provide sufficient space for the generation of each index, when considering the total number of crawled source code files, compared to the number of files at which the 16MB limit was reached for the un-partitioned original index. Moreover, partitioning allows parallelization of queries, as discussed further in section 6.4.5.

Database partitioning also enables a high degree of scalability. By adding more indexes 'horizontally', arbitrarily many index partitions could be created. For a system with sufficient threads, whether on a single machine or multiple machines, the number of files indexed could scale with no bound. By keeping the size of documents within each partition low, faster query speeds per partition are also obtained, which can be fully utilised within a parallel system. Despite partitions being implemented to address MongoDB's document size limit, their use is pragmatic, and would be a beneficial implementation regardless of a document size limit.

As a result of partitioning, the term 'index' is now more abstract, and refers to the collection of partitioned database tables which comprise the index. As partitioning was only enacted after generation of the first Java 5-gram index had begun, this un-partitioned index would be discarded, and a new partitioned index would be generated in its place. Yet, this re-indexing of the initial programs took relatively little time, due to the application of the previously discussed techniques. Furthermore, the smaller relative size of partitions meant that the time taken to update each was significantly lower.

6.3.4 Generation Speed

The effect of these discussed optimisations was significant, and resulted in performant index generation, which most importantly was scalable to an arbitrary number of source code files. As shown in table 6.1, for an index of size ~200,000 files there was a ~65x speedup in the time taken to index a program, when utilising bulk writes and partitioning. Furthermore, the time taken to index programs increased linearly for the partitioned index, a crucial requirement for scalable index generation. In addition, whilst this time growth is linear, the gradient could be considerably decreased by utilising more partitions. This would be possible and expected for an even larger-scale detection system, for instance using tens or hundreds of millions of source code files per index, potentially across multiple machines.

Table 6.1 also demonstrates both the independent effects of bulk writes and partitioning. With an index containing ~200,000 programs already, bulk writes decreased the batch update time by

more than 5x. Likewise, at a scale of ˜560,000 indexed programs, partitioning decreased the time per program by a factor of 6 compared to an un-partitioned index using bulk writes.

Index Optimisations	Indexed Programs	Batch Size	Batch Update Time (minutes)	Time per Program (seconds)
None	0	20,000	104	0.312
	166,000	20,000	560	1.680
Bulk Writes	216,500	20,000	37	0.111
	556,500	20,000	108	0.324
Bulk Writes & Partitioning	0	40,000	8	0.012
	200,000	40,000	17	0.026
	560,000	40,000	36	0.054
	1,000,000	40,000	65	0.098

Table 6.1: Time taken to tokenize and index a batch of programs for the Java 5-gram Index, given the existing number of programs within the Index and optimisations used.

6.3.5 Batch size

As mentioned, when using bulk writes, the batch size was used to determine how many programs to index within main memory before writing to persistent storage. In the case of no index optimisations, the batch size in table 6.1 refers to the number of programs indexed, as each program was indexed individually. Larger batch sizes were desired, as they resulted in fewer database queries, and so less total incurred overhead. The initial batch size of 20,000 was chosen for bulk writes due to the overall update times observed, such that if any unexpected failures were to occur then re-computation would not be overly costly. In addition, processing of such sized batches was comfortably within the cursor timeout period.

With a partitioned index, the updates within a batch were split across 8 partitions, as previously discussed. This did not reduce the number of *n*-gram updates per index notably though, as many *n*-grams occur and overlap across the programs allocated to each partition. However, the number of programs updated within each partition was divided by 8, which primarily afforded the update time savings. Therefore, a final batch size of 40,000 was chosen. Future work could look into modifying the batch size, as an optimal value is likely implementation dependent.

6.3.6 Main Memory Efficiency

As initially presented in section 5.3.4, Haskell tokenization was implemented via invoking a Haskell program, with Python's subprocess library [73]. The subprocess library invokes Linux's fork function, which "creates a new process by duplicating the calling process" [65]. This duplication involves the memory of the calling process, which would be the index generator. The fork function uses copy-on-write memory, so that memory is not duplicated until either process actually modifies such memory. However, the index generator performs memory modifications during each program iteration. Consequently, for each instantiated subprocess to run Haskell tokenization, the index generator's memory usage would be temporarily doubled.

As index generation heavily utilised main memory when generating a batch's index, the resulting memory spike caused the development VM to nearly run out of main memory, possibly as only 8GB of ram was provisioned. As a result, the Linux kernel's out-of-memory (OOM) killer would shut down processes which it deemed to use excessive memory. This included the MongoDB process, therefore preventing further iteration of the source code table, any index updates,

or assignment of new persistent language tokens. It was also not feasible to reduce the memory usage to a sufficient level by decreasing the batch size, in order to avoid process killing. This was as the index generation would be greatly hindered by a prohibitive batch size, negating the benefits of the discussed optimisations.

Therefore, the index generation process was split into two steps. Firstly, programs would be iterated and tokenized, as before. However, the token stream would then be written to persistent storage, along with the program ID[2]. Secondly, the index generator would then iterate through the stored token streams, convert them to character streams, and then update the relevant index with the subsequent *n*-gram frequencies, as previously described.

As this process only utilised subprocess invocations within the first step, the second step would not encounter any memory issues. Thus, the second stage could perform as described, without any significant alterations. Whilst the first step could still encounter memory issues, this was mitigated by first reading the required source code files to main memory, before tokenizing each and storing the result. Therefore, a MongoDB process was only required before any subprocess invocations were performed, allowing the first stage to operate without any memory issues. As with index generation, this first step was performed in batches, to reduce the memory usage as a result of the source code files in main memory. A batch size of 35,000 was used, as a trade-off between database requests and main memory needed. Consequently, the system performed within memory constraints and the OOM killer was not invoked, as the heavy memory usage from index generation was only present within the second stage.

This two-part technique is also applicable for other indexes, as it means that repeated tokenization of the same programs does not need to occur, for example when generating a 4 and 5-gram index for the same language. However, this was not implemented beforehand as tokenization overhead was relatively minimal, and index generation was not a bottleneck.

For future systems employing such an approach, persistent storage of each program's token stream is not needed; batches could be used for the two stages such that each is performed in tandem. This would reduce intermediate memory usage if the token streams are not required after index generation. For this project, each Haskell program's token stream was persistently stored and kept after index generation however, as they were used for analysis.

6.4 Index Querying

Resultant indexes included 4, 5 and 6-gram indexes for Java and Haskell, and 5-gram indexes for C, C++, and Header files. Java and Haskell were chosen to generate and evaluate additional 4-gram and 6-gram indexes due to their scale and unique paradigm within this project, respectively.

6.4.1 Index Checker

As presented in section 6.2.2, when querying an index, the most similar programs are desired to be retrieved, according to their respective *n*-gram frequencies. An *index checker* was therefore created to retrieve candidates for a given source code file, which consisted of the following steps:

1. Tokenize the input source to a stream of characters.
2. Generate *n*-gram frequencies from the character stream.
3. For each *n*-gram, retrieve corresponding program frequencies from the index.
4. For each program frequency, increment that program's similarity in a local map.
5. Return the top T programs based on their calculated similarity.

[2]A CSV file was used for persistent storage, as the only required actions were iteration (for index generation), and appending (during the tokenization process).

As development of index generation and querying was concurrent, initially querying was performed on un-partitioned indexes.

6.4.2 Similarity Assignment

As within index generation, the provided source program to be checked against the index was first tokenized, and the resulting n-gram frequencies obtained. The chosen index was then queried for each *n*-gram, utilising MongoDB's '`in`' query operator to retrieve all *n*-gram program frequencies within the index in one request, therefore minimizing transfers [74]. As the *n*-grams are the IDs for each document, they can be retrieved in constant time, and so the overall time complexity to retrieve N *n*-gram's program frequencies from the index is $\mathcal{O}(N)$. For each of the indexed program's n-gram frequencies, the specific n-gram similarity was then computed, with the overall result across each n-gram summed, as shown in figure 6.7.

$$\text{similarity}_n(p \mid s) = \frac{1}{|s| - n + 1} \sum_{g \,\in\, p \cap s} \text{sim}(f_{p,\,g}, f_{s,\,g})$$

Figure 6.7: Similarity equation for an indexed program p, given a provided source program s for a chosen n. $|s|$ denotes the number of tokens in s. g denotes an *n*-gram within both p and g. 'sim' denotes the similarity function, used to compare *n*-gram frequencies within p and s, labelled $f_{p,\,g}$ and $f_{p,\,s}$ respectively.

As described previously and shown in 6.7, the similarity of each indexed program and the provided source is performed on the intersection of their *n*-gram sets. This facilitates indexed programs with the largest *n*-gram overlap to have the greatest potential for a higher similarity, as more *n*-gram frequencies will be compared. Moreover, only indexed programs which actually contain identical short code sections, *n*-grams, will need to be checked.

As explained in figure 6.7, the *similarity function* determines how the respective frequencies of the source and indexed program's *n*-grams should be compared. For each shared *n*-gram of the source and indexed programs, a *score* is assigned from the similarity function, which is a positive integer. An indexed program's total score is therefore the sum of each score, with an indexed program's *similarity* computed as its total score divided by the source program's total number of *n*-grams, which is equal to the $|s| - n + 1$ as shown. As described in section 6.2.2, the *min* function was used, such that the score for an individual n-gram was equal to the frequency of the indexed program, albeit bounded at the frequency of the source. The initial similarity function is therefore shown in figure 6.8.

$$\text{sim}(f_{p,\,g}, f_{s,\,g}) = \min(f_{p,\,g}, f_{s,\,g})$$

Figure 6.8: The initial similarity function used to compare *n*-gram frequencies, equivalent to their minimum.

The use of the *min* function meant that if an indexed program contained a number of *n*-grams greater than or equal to the source's program, it would obtain the maximum possible score for that n-gram. Therefore, programs which did not contain sufficient identical *n*-grams would be penalised. This approach has also been used within prior research [10]. When each *n*-gram's score across all programs is considered, the resultant top-ranking programs therefore reflect a higher degree of similarity.

Therefore, the resulting queried programs can be ordered by their computed similarities from 6.7, obtaining a *rank* for each. The rank of an indexed program is its position compared to other programs within the index, when ordered by resultant similarities. As ordering could exclusively be performed on program's total scores, similarities are not strictly needed. However, as total

scores are unbounded, they are less clear when compared to a similarity score between 0 and 1, both for evaluation and presenting to users.

6.4.3 Candidate Set

As previously mentioned, the candidates can therefore be retrieved by selecting the top T ranked programs, generating a *candidate set*.

As there are potentially millions of compared programs, retrieving the top T candidates would cause unnecessary overhead if they were to be naively obtained, for example by sorting the entire sequence of candidates and then returning the top T. This would have a time complexity of $\mathcal{O}(M \log(M))$, where M is the number of compared programs within the index, due to the requirement to sort the entire list of program similarities. Therefore, the `nlargest` function within Python's `heapq` library was utilised, which has a time complexity of $\mathcal{O}(M \log(T))$ where M and T are as before, by leveraging an intermediate heap [75]. As T will be a constant value for candidate retrieval, the time complexity of this is actually $O(M)$. Furthermore, in practice T is considerably smaller than M, on the order of thousands instead of millions. Even if T were not constant, the logarithmic factor would be negligible, and at this scale would have ~M comparisons in total [76], demonstrating the negligible constant factor of the linear time complexity.

Whilst the chosen candidate set was decided to be the top T ranked programs, an alternative approach could have been to choose all programs with a similarity higher than a certain threshold, as is used within existing plagiarism detection tools [14]. However, similarity distributions can change significantly across different indexes, both as a result of overall language *n*-gram distributions, and the varying values of *n*. Even if a threshold was chosen for each individual index, the number of programs above the threshold could drastically change due to the *n*-gram distribution of the input source. For example, a short program composed of common *n*-grams would have higher average similarities than a large program with many infrequent *n*-grams. As the detailed analysis stage is more resource-intensive due to a more selective comparison process, if too many programs are provided then it may become prohibitively slow, reducing scalability with regards to an input corpus of sources. Furthermore, the value of T could be tailored to suit both the candidate retrieval and detailed analysis stages, based on comparison times.

The value of T was chosen to be 2,000 for this project, in order to allow for real-time end-to-end plagiarism detection when using detailed analysis. A larger and therefore less selective value could be chosen to increase recall within real usage, but was decided against for this project due to the extra time such candidates would entail within detailed analysis, as discussed further in section 7.1.2. The value of 2,000 therefore represented a trade-off between sufficient selectivity and time constraints.

```
source_n_gram_freqs = n-gram frequencies of input source program;
scores = map of program IDs to scores, initialised to 0;

for (n_gram, source_freq) in source_n_gram_freqs:
    for (program, program_freq) in index[n_gram]:
        scores[program] += sim(source_freq, program_freq);

sims = map from program IDs to similarities;
for (program, program_score) in scores:
    sims[program] = program_score / len(source_n_gram_freqs);
```

Figure 6.9: Similarity assignment algorithm for index querying.

6.4.4 Time Complexity

Whilst the similarity equation given in figure 6.7 requires iteration through each indexed program's *n*-grams shared with the given source, partial computation can be utilised in order to efficiently compute values across the entire index. Figure 6.9 shows the similarity assignment algorithm used, whereby each index program's total score is computed incrementally by iterating over each of the source's *n*-gram first, with the similarities computed after, thereby only requiring one nested loop.

As the similarity function 'sim' operates in constant time, the time complexity of the overall similarity assignment algorithm is $\mathcal{O}(NM)$, where N is the number of *n*-grams in the input source, and M is the number of programs in the queried index. M is actually the number of indexed programs containing common *n*-grams, which is in practice a far smaller number than the total number of indexed programs. However, as M could theoretically be the total number of indexed programs (such as for a source containing every indexed *n*-gram), the stated time complexity is obtained. Hence, the steps outlined in section 6.4.1 have the following time complexities:

1. Tokenization: $\mathcal{O}(K)$ where K is the number of source tokens.
2. *n*-gram generation: $\mathcal{O}(K)$
3. Frequency generation: $\mathcal{O}(N)$ where N is the number of source *n*-grams.
4. Similarity Assignment: $\mathcal{O}(NM)$ where M is the number of queried programs.
5. Candidate Set Retrieval: $\mathcal{O}(M)$

The time complexity of the entire candidate retrieval stage during index querying is therefore $\mathcal{O}(K + N + NM + M)$, which is equivalent to $\mathcal{O}(NM)$. This is as $\mathcal{O}(N)$ and $\mathcal{O}(M)$ are insignificant compared to $\mathcal{O}(NM)$, whilst $\mathcal{O}(K)$ is pragmatically negligible, or one could assume that a source's *n*-gram frequencies are obtained before candidate retrieval occurs.

6.4.5 Parallelization

After partitioned indexes were used, querying needed to check each partition, as the indexed programs were uniformly distributed across each. Each partition could have been sequentially queried identically as to before, with the resulting scores updating the same local similarity map. However, each partition is independent of each other, both semantically in regard to each being a self-contained complete index, and logically in regards to each being a distinct MongoDB table. Therefore, MongoDB's parallelism could be exploited to query each partition independently [44].

The similarity assignment algorithm presented in figure 6.9 was therefore refactored to an individual function, which could run in an isolated thread. This function was then provided the *n*-gram frequencies of the source, as well as the table name for its assigned index partition. The function would then return the similarity map for all programs within its partition, as before.

Python's `Pool` function from the `multiprocessing` library was utilised, allowing independent MongoDB queries to be performed in different threads [77]. The main thread running the index checker could therefore instantiate a 'worker' thread for each partition, and then merge the resulting program similarities before retrieving the candidate set as previously discussed.

Whilst all program similarities were returned from each worker thread's similarity assignment function, in an optimal system only the top T programs would have to be returned from each thread, reducing overall processing and data-transfer time. As only the top T candidates are needed, those from each thread could be merged and then the final T used. For this project, the entirety of each partition's program similarities were returned; processing was deliberately

performed within the main thread after each thread's query had finished, only then merging the obtained similarities. This allowed access to each program's similarity for subsequent analysis, as well as allowing the total number of queried programs across each index partition to be obtained.

Whilst 8 partitions were used for each index, the development machine used only has 4 single-threaded cores, as mentioned in section 3.2.2. Despite a greater number of partitions permitting faster index generation times, index querying does not benefit from a greater number of partitions than threads. Therefore, speed-up has an upper bound of the number of threads, in this case 4x. The initial value of 8 partitions was chosen due to the assumption that the development CPU was hyper-threaded, which was not the case. Despite this, the use of 8 partitions likely has negligible impact compared to 4, as the overhead not related to program iteration or database queries is insignificant. The use of parallelization is greatly beneficial as shown in table 6.2, yielding a ~3x speed-up compared to synchronous querying.

Moreover, the speed of index checking is apparent; for a source file containing 468 tokens, ~120,000 indexed programs were able to be checked on average each second, allowing the resultant candidates to be retrieved in less than 10 seconds from an initial corpus of ~1.3 million programs.

Number of tokens in file	Distinct 5-grams	Total compared programs	Query implementation	Total time (seconds)	Parallel speed-up
468	266	1,148,631	Synchronous	27.71	2.93
			Parallel	9.47	
1,385	631	1,187,648	Synchronous	38.85	2.89
			Parallel	13.42	
2,914	1,092	1,201,586	Synchronous	54.26	3.02
			Parallel	17.98	

Table 6.2: Table comparing the time taken to retrieve candidates for different length files within the Java 5-gram index when using synchronous and parallel queries.

Table 6.2 also demonstrates how the number of compared programs varies as a result of the distinct n-grams within the provided source file. As shown, the total number of compared programs increases due to the inclusion of new n-grams, increasing total processing time as a consequence of iterating through additional program frequencies. However, even though the number of distinct n-grams more than quadruples between the first and last queried programs in table 6.2, the total time taken only increases by a factor of ~1.9. This is despite candidate retrieval's time complexity being linear with regards to the number of distinct n-grams, as fewer program frequencies are likely obtained per additional distinct n-gram. This is due to frequent n-grams appearing in more programs, and therefore representing the program frequencies primarily retrieved. This is reflected by the slowly increasing total number of compared programs, which only increases by ~4.6% whilst the number of distinct n-grams greatly increases as discussed.

6.4.6 Testing

A crucial issue with detecting source code plagiarism from an index is that if a program is not present within the index, then plagiarism of that program cannot possibly be identified. For example, if plagiarism occurred from a file within an un-crawled repository, such plagiarism could not be detected. Whilst there could be ways to address this, they are not implemented within this project, but will be detailed in section 9.2.3.

Therefore, to test and evaluate candidate retrieval, source code present within the index was used. This provided knowledge that the plagiarised program was possible to be identified, and therefore any cases of undetected plagiarism via missed candidates could be investigated[3].

File Extension	Number of tokens in file	n	Actual Similarity	Average Candidate Similarity	Threshold Similarity
.java	472	4	1.0	0.771	0.735
		5	1.0	0.666	0.624
		6	1.0	0.567	0.522
.hs	361	4	1.0	0.544	0.494
		5	1.0	0.397	0.350
		6	1.0	0.278	0.233

Table 6.3: Candidate set similarity thresholds when querying unmodified plagiarised source code.

As indexed programs would be used for testing, their respective IDs could be used to observe their similarities and ranks, within the generated similarity map. For plagiarised programs with no modifications, a similarity of 1 is expected to the equivalent indexed program as the n-gram frequencies would be identical, and is observed in table 6.3. As also expected, modifying the contents of a plagiarised file, whether keeping semantics or not, would alter the n-gram distribution and so a similarity lower than 1 should be observed.

Table 6.3 shows the average similarity of indexed programs within the candidate set, as well as the threshold similarity of the candidate set, equal to the Tth ranked program. Whilst each program had a similarity of 1.0 due to no modification having occurred, plagiarism obfuscation or other alterations could be performed with the *true candidate* still being identified, as long as the provided source still had a similarity greater or equal to the shown threshold similarities.

A true candidate refers to the actual plagiarised program, within an index. Within testing and evaluation, inclusion of the true candidate within the candidate set will be the primary observation, as this indicates successful plagiarism detection within candidate retrieval. In the case of plagiarism having occurred from multiple sources, there would be multiple true candidates.

Within this project, testing and evaluation will primarily occur for Java and Haskell programs. This is due to Java having the largest number of files indexed, therefore demonstrating the largest scale with over 1.3 million files indexed. Haskell also demonstrates the applicability to a different paradigm, and therefore the extensibility and generality of the overall plagiarism detection system, with regards to application to arbitrary programming languages not included within this project.

Furthermore, evaluation will primarily occur for candidate retrieval. This is as it is a novel technique within the field of source code plagiarism detection research, as it focuses on reducing a very large source code corpus to a small corpus with the goal of including any plagiarised programs. Considerably more research has been performed on corpora containing hundreds or thousands of files, compared to tens of thousands or even millions, as discussed in section 1.1.

6.5 Validation

Validation was initially performed on the Java indexes. A chosen Java file was modified using plagiarism obfuscation techniques, whilst keeping the semantics equal. Three different levels of

[3] Un-indexed code can still be used for evaluation, for example in the case of observing similarities for unplagiarised code, or to act as original code within a file containing a small section of plagiarised code.

plagiarism obfuscation were initially used, which progressively built on previous levels. Such levels will be discussed further in section 8.1.

Using different levels of obfuscation serves to both test and evaluate candidate retrieval. When compared to a true candidate, lower similarities would be expected with increasingly modified plagiarised code, demonstrating how the system's comparison shown in figure 6.9 functions correctly. Furthermore, such similarities should not be below the threshold similarity for a given query as previously discussed, such that the true candidate is included within the candidate set, i.e. its rank is less than T.

As described, a further obfuscated program would be expected to have a lower *true similarity*, which is the similarity of the provided obfuscated program to its true candidate. Consequently, the *true rank* would be expected to drop, however at a reasonable rate. Whilst such evaluation can be qualitative, quantitatively examining the true rank to be below the candidate set size T is imperative, as mentioned. Lower levels of plagiarism obfuscation would also be expected to have a notably low true rank, such that inclusion within the candidate set is almost guaranteed, regardless of T. Different n values can also be observed, determining how they perform relative to each other.

6.5.1 Obfuscated Plagiarism

File Extension	Number of tokens in file	Metric	n	Plagiarism Obfuscation		
				Light	Medium	Heavy
.java	472	Similarity	4	0.931	0.837	0.741
			5	0.907	0.776	0.655
			6	0.879	0.717	0.582
		Rank	4	1	159	1,231
			5	1	48	536
			6	1	7	224

Table 6.4: Varying program similarity and ranks compared to the actual Java true candidate, for different levels of plagiarism obfuscation.

As shown in table 6.4, the true similarities and ranks drop as increasing levels of plagiarism obfuscation are performed. Light plagiarism obfuscation lowers true similarities from 1 to ˜0.9, with similarity decreasing more for 6-grams. This was initially hypothesised in section 6.1.3, as larger n-grams are affected more by token alterations. Despite decreasing true similarities compared to unmodified plagiarism as shown in table 6.3, the true candidate is ranked first for light plagiarism obfuscation, showing how for any value of T, the true candidate would still have been correctly retrieved. For medium and heavy plagiarism, the similarities drop further, with 6-grams being affected the most. Despite having a steep decrease in similarity compared to 4 and 5-grams, the use of 6-grams still yields better true ranks. This is as a consequence of the remaining unaltered 6-gram frequencies being closer to the true candidate's 6-gram distribution, compared to other indexed programs.

During candidate retrieval, a candidate's rank within the candidate set does not matter as long as it is below T, as all such candidates will then be analysed further, treated equally. T was chosen to be 2,000 for this project as detailed in section 6.4.3, and therefore the true candidate would be correctly retrieved for each combination of plagiarism obfuscation and choice of n within table 6.4. Therefore, obfuscated plagiarism can be detected within the Java indexes, correctly

retrieving the true candidate from over 1.3 million indexed files. Moreover, T could easily be modified to a higher value such as 10,000 in a more selective system where real-time plagiarism detection is not desired, unlike for evaluation of this project.

File Extension	Number of tokens in file	Metric	n	Plagiarism Level		
				Light	Medium	Heavy
.hs	361	Similarity	4	0.811	0.694	0.620
			5	0.786	0.627	0.543
			6	0.761	0.582	0.478
		Rank	4	1	2	18
			5	1	1	2
			6	1	1	1

Table 6.5: Varying program similarity and ranks compared to the actual Haskell candidate, for different levels of plagiarism obfuscation.

Table 6.5 shows the same experiment, but for the Haskell indexes. As with the Java program, the true similarity drops notably as increasing levels of plagiarism obfuscation are performed. However, the true ranks are considerably higher, and the heavily obfuscated code is ranked first when using the 6-gram index. This is likely due to the relatively small scale however, as only ~130,000 Haskell files were indexed, a factor of 10 smaller than the Java indexes.

Furthermore, the n-grams generated from crawled Haskell files were more unique, as shown in figure A.4, due to the larger token set. Table A.8 shows how a Java 5-gram occurred in 1,292 programs on average, whereas a Haskell 5-gram occurred in just 60 programs on average, as shown in table A.9. Therefore, detection of Haskell programs could be considered 'easier', due to the smaller scale. However, despite the relatively smaller scale, more than 100,000 Haskell programs were still compared to the plagiarised program in table 6.5, demonstrating successful identification at a large scale for a functional paradigm.

6.5.2 Multiple Sources

As initially outlined in section 3, the ability to detect plagiarism from multiple different files was desired. Therefore, sources containing plagiarised code from multiple indexed programs were created to evaluate retrieval of multiple true candidates. For this purpose, indexed files were chosen to be plagiarised based on multiple factors. Firstly, plagiarism sources were desired to contain mostly logical code, as opposed to templates or interface definitions. This would allow for a more accurate reflection of actual plagiarism, enabling pragmatic evaluation. Secondly, self-contained code without many dependencies was desired, as this also may more accurately reflects realistic plagiarism, due to extensive dependency resolving being a relatively time-consuming task when compared to simply copying an independent online source file.

Source code files containing solutions to algorithmic problems, such as those used within programming interviews or academic assignments were therefore plagiarised. Such source code files can be plagiarised in practice, and often implement solutions to specific problems, existing independently without any dependencies. In order to obtain such indexed files, the source code table was iterated to inspect files from repositories containing LeetCode solutions. LeetCode [78] is a site which hosts such algorithmic problems, and many GitHub repositories were crawled for this project which contained solutions to such problems.

The relevant code within such source code files was therefore used to generate plagiarised

sources for evaluation. Plagiarised sources were created by appending code sections from multiple sources together, ranging from a single plagiarised source to several. Initially, testing using up to 9 different sources was chosen, but latter evaluation used 7 plagiarism sources in total, and so 7 files used for evaluation were generated. For example, the first source file of this evaluation set would contain plagiarised code from one indexed program. The second source file would contain the same code, but with plagiarised code from an additional source. This process repeated such that the 7th file would contain code from 7 different sources. This number of sources was deemed appropriate, as realistic plagiarism would likely not utilise notably more files, in addition to initial tests suggesting that detection of more sources would likely be less feasible.

Initial evaluation was conducted on Java programs, and plagiarised sections from chosen source code files ranged in length from 331 to 467 tokens. Such token lengths corresponded to the 54th and 64th percentile of crawled Java programs respectively, as shown in figure A.5, and below the mean number of tokens of 723. The number of lines in such plagiarised sections ranged from 38 to 54, corresponding to the 21st and 36th percentile, including whitespace and comments which were not altered. The discrepancy between the percentiles of the number of lines and tokens is likely due to the chosen files containing entirely logical code as mentioned, which could contain a higher number of tokens per line on average than other code such as template files. Such files were therefore deemed to reflect reasonably realistic plagiarism candidates.

Whilst the order of such programs may matter, a single arbitrary order was chosen, as evaluating other permutations would be too time-consuming whilst yielding little additional information, due to relative candidate ranks likely not changing significantly.

Number of plagiarised files in source	Total source tokens	True rank of unmodified plagiarised files						
		1	2	3	4	5	6	7
1	467	1	-	-	-	-	-	-
2	933	82	79	-	-	-	-	-
3	1,372	1,699	1,346	2,655	-	-	-	-
4	1,839	4,358	3,104	5,906	4,547	-	-	-
5	2,201	6,889	4,999	8,866	7,172	18,450	-	-
6	2,571	12,474	9,893	15,994	14,189	31,974	28,320	-
7	2,902	15,234	12,177	19,025	17,238	36,685	32,455	44,136

Table 6.6: Table showing the rank of multiple true candidates for sources containing multiple unmodified plagiarised code sections, within the Java 6-gram index.

Table 6.6 shows the results of detecting multiple sources of plagiarism. As previously demonstrated, identification of a single source is successful, with a true rank of 1 obtained. Likewise, code plagiarised from 2 different programs is successfully identified, with each true candidate ranking ˜80. However, for 3 or more separate plagiarised files, plagiarism detection starts to fail. As T was chosen to be 2,000, the third source would fail to be identified, and from a source composed of 4 or more programs, none can be identified. Whilst T could be increased, even if a value of 10,000 were chosen, failure to detect the fifth true candidate would still occur, likewise for each candidate in the 6 or 7-program sources.

This evaluation was also performed on the 4 and 5-gram Java indexes, however the true ranks were consistently worse due to the plagiarised code being unmodified as shown in table A.10, hence 6-grams performed considerably better as described in section 6.1.3.

At a large scale, detecting plagiarism from multiple sources is a significantly harder challenge

than detecting plagiarism of a single source. As discussed in section 6.1.6, the summation of n-gram distributions as a result of concatenating files is additive. This means that a concatenation of multiple programs can effectively be decomposed into the individual n-gram distributions, when comparing n-grams pairwise, as is done during index checking. However, with more sources concatenated, the probability of significant overlap between each plagiarised section's n-grams increases, impeding the ability to accurately retrieve each true candidate.

Furthermore, in addition to each program's distribution disrupting each other's, the addition of new tokens and n-grams with each new source causes the file to become larger. Consequently, similarity to the numerous very large indexed programs increases due to their large number of n-grams, and so more *noise* is accumulated. Noise refers to the inclusion of unplagiarised programs within the candidate set, as a combination of the described factors, possibly culminating in the exclusion of true candidates from the candidate set. As shown in 6.6, the noise increases greatly, resulting in failure to retrieve true candidates. For the file composed of 7 plagiarised sources, each retrieved candidate had 3,711 lines on average, demonstrating how larger files were detected to be more similar than the true candidates.

6.5.3 Noise

As discussed, noise from large indexed programs containing many n-grams can cause detection of true candidates to fail. This is primarily as they can contain supersets of queried program's n-grams, therefore obtaining high or maximal scores for a given n-gram during pairwise comparisons. For example, if a provided source program contained the frequently occurring n-gram '`ABCD`' 10 times, then there could be potentially thousands of large programs which also contained 10 or more occurrences of this n-gram, therefore obtaining a maximal score for that specific n-gram. When repeated for all n-grams within the source, large programs will likely obtain a high score, introducing the aforementioned noise which can obscure a true candidate.

Number of plagiarised files in source	Total source tokens	True rank of medium-level obfuscated plagiarised sources						
		1	2	3	4	5	6	7
1	467	7	-	-	-	-	-	-
2	933	1,112	858	-	-	-	-	-
3	1,372	6,424	5,558	3,192	-	-	-	-
4	1,839	13,893	6,233	8,336	12,275	-	-	-
5	2,201	17,455	7,810	11,495	15,020	69,375	-	-
6	2,571	25,994	15,180	20,537	27,105	>100k	45,818	-
7	2,902	29,456	17,612	23,263	30,989	>100k	50,035	69,422

Table 6.7: Table showing the rank of multiple true candidates for sources containing multiple plagiarised code sections with medium plagiarism obfuscation performed, within the Java 6-gram index.

For example, 508 Java 5-grams occur in more than 100,000 indexed programs, and 3,919 occur in more than 10,000 programs. Therefore, when such 5-grams are present within a source, large indexed programs will be more likely to obtain high scores. As more than 229,000 Java files contain more than 1,000 tokens and consequently n-grams, a significant challenge for detection of multiple sources is noise from large files. This effect is compounded when plagiarism obfuscation has been performed on each plagiarised source, as can be seen in table 6.7.

After medium plagiarism obfuscation was performed on each plagiarised code section, detection significantly worsened, and only detection of up to 2 sources was possible, as shown in table

6.7. As with no plagiarism obfuscation like table 6.6, the Java 4 and 5-gram indexes performed worse than 6-grams. Within the 5-gram index, not even 2 medium obfuscated sources could be detected; the true candidate's ranks were 4,494 and 4,178.

6.5.4 Potential Noise Mitigation

The issue noise presented within large-scale plagiarism detection was therefore desired to be addressed. Due to noise primarily being caused by large programs, penalisation of programs based on the number of contained tokens was considered, as such programs would likely contain more common *n*-grams. However, this would affect detection of actual plagiarism from large programs and cause true candidates to be missed, depending on the level of penalisation and plagiarised code similarity.

Inclusion of an *n*-gram's global frequency across the entire index could also be considered would comparing programs. For instance, infrequent *n*-grams could have a higher weighting for an indexed program's total score, as their uniqueness may aid detection of plagiarism from a specific program. However, infrequent global *n*-grams will likely still occur in many large programs; as shown in figure A.5, the largest 1% of Java files contain ˜22% of the total number of tokens. Furthermore, this would involve additional storage space and processing time. Albeit, this approach may still have merit and could be explored further in future work.

During plagiarism obfuscation or other modification of plagiarised code, the frequency of each *n*-gram can change. Obfuscation could reduce the frequency of certain *n*-grams, and potentially increase the frequency of others. However, it is likely to observe an overall decrease in the total frequency of plagiarised *n*-grams. This is as modification of a single token will change n *n*-grams. Consequently, the n new *n*-grams produced via modifying a single token would already have to be present within the plagiarised code's *n*-gram set, in order for the total number of *n*-grams to remain the same. Yet even in this case, the overall distribution would still be changed.

Despite modification to plagiarised code likely decreasing the total frequency of the original *n*-gram set, inclusion of a plagiarised segment will only contribute to *n*-gram frequencies, due to their additive property as discussed in section 6.1.6. Therefore, shared *n*-gram frequencies with a true candidate could also increase. A presumption that a plagiarised source's *n*-gram frequencies should stay the same as the true candidate's frequencies can therefore be made, on average, based on such observations. This can also be seen from the fact that unmodified plagiarised code will have the exact same *n*-gram distribution.

Consequently, indexed programs could be penalised for containing too many shared *n*-grams, as opposed to the overall number of *n*-grams. This could be enacted on a per-*n*-gram basis, such that if an indexed program contained a specific *n*-gram more than the source, it would be penalised. This is reasonable from a pragmatic perspective; if a source with 20 occurrences of the 3-gram '`ABC`' was queried, then an indexed program with 25 such occurrences should be deemed more similar than one with 35 occurrences. However, such potential candidates should not be completely discounted if they contain too many specific *n*-grams, as only a subsection of code from an indexed program could be plagiarised.

To address this, *drop-off* was implemented, such that programs with too many shared *n*-grams would be penalised, obtaining a lower total score and therefore similarity.

6.6 Drop-off

Drop-off is the penalisation of indexed programs which contain excess *n*-grams, enacted on a per-*n*-gram basis. Consequently, if a section of code is plagiarised from an indexed program,

any additional different code within the indexed program would not be penalised, as the n-grams would be different. This ensures that plagiarism from large files can still be detected, as many additional n-grams can still be present without any detriment, unlike previously discussed penalisation based on the length of programs.

Drop-off is implemented via a different similarity function, which compares the frequencies of a shared n-gram between the provided source and an indexed program. Therefore, the equation presented in figure 6.7 is unchanged. The previously used similarity function for initial evaluation and within previous research was the *minimum* function [10], which does not utilise drop-off. Instead, n-gram scores were capped at the frequency of the source n-gram, as discussed in section 6.4.2. This results in a constant period after the source frequency is reached, as can be seen in figure 6.10.

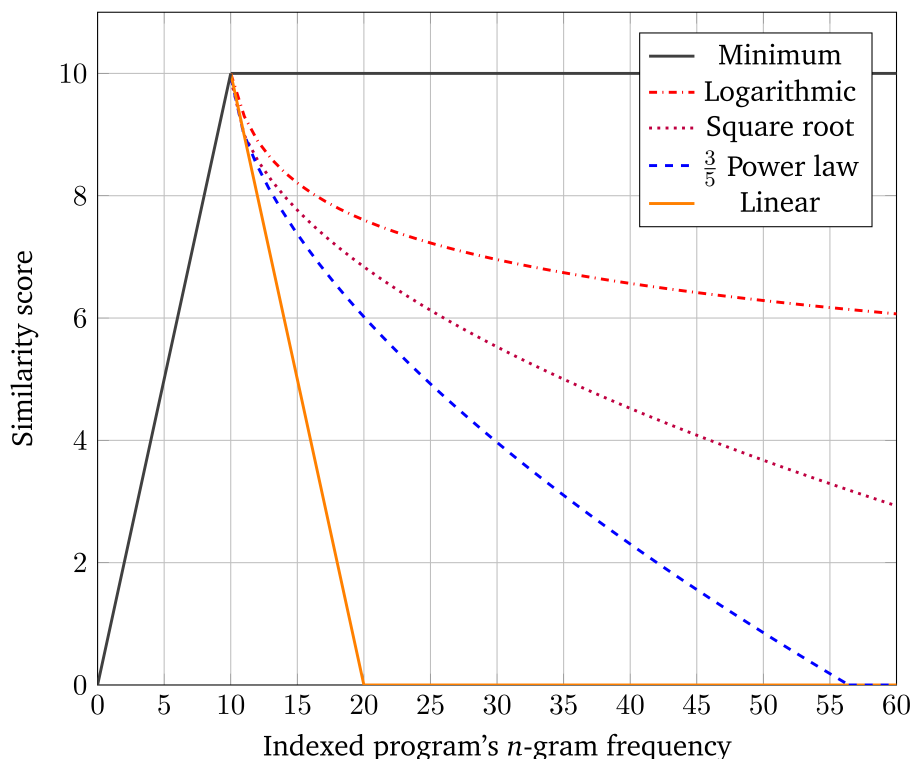

Figure 6.10: Figure showing the score for an individual source n-gram which occurs 10 times, for different similarity functions.

Figure 6.10 shows possible implementations of similarity functions with drop-off, as well as the original *minimum* function. As shown, similarity functions utilising drop-off assign a lower score for a given n-gram if it occurs more frequently in an indexed program than in the source program. All examined similarity functions utilised the same initial linear section, such that if an indexed program's n-gram occurred fewer times than the source, then that n-gram would obtain a score equal to its frequency. This was as no alternative initial section was deemed plausible with regards to optimising plagiarism detection. As shown, similarity functions do not assign negative scores and are bound at 0, as the inclusion of extraneous n-grams should not detract from a program's similarity, for example in the case of only plagiarising a subsection of a file.

6.6.1 Similarity Function Choice

Initially, square root drop-off was chosen to be used. Whilst this would penalise dissimilar programs as previously discussed, detection of plagiarism from larger programs would still be possible, as a result of the longer quadratic tail. Linear drop-off was hypothesised to be too harsh, negatively impacting such plagiarism. As shown in figure 6.10, if an indexed program's n-gram occurred 45 times, it would obtain a score of 4 with square root drop-off, whereas the *minimum* function would assign a score of 10 for a source n-gram occurring 10 times.

Such square root drop-off had a greatly positive effect. For the same evaluation of multiple unmodified plagiarised sources as in table 6.6 using 6-grams, the true ranks were considerably lower on average. As shown in table A.11, when a file composed of 4 plagiarised programs was queried, the respective ranks of the true candidates were 2, 1, 12 and 3, and 4 true candidates in the file composed of 5 programs were successfully retrieved. However, only 1 true candidate out of 7 was retrieved within the largest program composition when using a candidate set size T of 2,000. This demonstrates how noise still hinders candidate retrieval utilising drop-off, both for larger programs and those containing more plagiarised sources.

With the *minimum* similarity function, the medium-level obfuscated program containing 4 plagiarised sources could not identify any, with an average true rank of 10,184. With square root drop-off, the same queried program correctly identified each obfuscated true candidate, with an average true rank of just 97, as shown in table A.12. This demonstrates the significant impact drop-off can have on detecting multiple obfuscated sources.

6.6.2 Power Law Drop-off

Due to the apparent success of utilising square root drop-off, a harsher drop-off was used, with the aim to further mitigate noise. As square root drop-off is equivalent to a power of 0.5, the value of 0.6 or $\frac{3}{5}$ was chosen to be tested, as can be seen in figure 6.10. This fractional power used for the similarity function can be referred to as a power law. Evaluation was repeated for power law drop-off, and again had notably beneficial effects compared to square root drop-off, as shown in table 6.8.

Number of plagiarised files in source	Total source tokens	True rank of unmodified plagiarised sources						
		1	2	3	4	5	6	7
1	467	1	-	-	-	-	-	-
2	933	1	2	-	-	-	-	-
3	1,372	1	2	3	-	-	-	-
4	1,839	2	1	5	3	-	-	-
5	2,201	89	63	235	105	1,599	-	-
6	2,571	622	536	1,156	862	5,260	4,796	-
7	2,902	1,325	1,180	2,150	1,711	7,821	7,133	11,080

Table 6.8: Table showing the rank of multiple true candidates for sources containing multiple unmodified code sections, within the Java 6-gram index using $\frac{3}{5}$ power law drop-off.

When compared to the same queried programs using the *minimum* similarity function without drop-off in table 6.6, the difference is considerable. Table 6.8 shows how 5 different sources can be identified. Moreover, with medium-level plagiarism obfuscation performed as shown in table A.13, the average true ranks for 4 plagiarised sources was 47, compared to 97 for square

root drop-off as discussed. As steeper drop-offs appeared to provide better results for detecting plagiarism from multiple sources, linear drop-off was investigated and contrasted to power law drop-off.

6.6.3 Linear Drop-off

As shown in figure 6.10, linear drop-off is symmetric, assigning a maximal score only in the case that an indexed program's n-gram occurs the same number of times as the source's n-gram. This symmetry reflects the presumption that on average, a plagiarised source's n-gram frequencies should stay the same as the true candidate's frequencies, as previously described in section 6.5.4. The same programs composed of multiple unmodified plagiarised programs were then evaluated.

Number of plagiarised files in source	Total source tokens	True rank of unmodified plagiarised sources						
		1	2	3	4	5	6	7
1	467	1	-	-	-	-	-	-
2	933	1	2	-	-	-	-	-
3	1,372	1	3	2	-	-	-	-
4	1,839	1	3	4	2	-	-	-
5	2,201	1	2	6	3	100	-	-
6	2,571	8	23	56	33	882	994	-
7	2,902	67	103	166	120	1,741	1,884	3,173

Table 6.9: Table showing the rank of multiple true candidates for sources containing multiple unmodified code sections, within the Java 6-gram index using linear drop-off.

Table 6.9 demonstrates the immense benefit of utilising linear drop-off within n-gram comparisons. A file containing plagiarised code from 6 different programs can be searched within an index containing more than 1.3 million files, with all 6 sources retrieved within 15 seconds. Likewise, 6 true candidates were retrieved for the 7-source plagiarised file, yet if a candidate set size of 10,000 were to be used such as in a practical system, then each could be amply retrieved.

Number of plagiarised files in source	Total source tokens	True rank of medium-level obfuscated plagiarised sources						
		1	2	3	4	5	6	7
1	467	1	-	-	-	-	-	-
2	933	1	2	-	-	-	-	-
3	1,372	2	3	1	-	-	-	-
4	1,839	5	3	1	7	-	-	-
5	2,201	31	3	1	41	9,280	-	-
6	2,571	90	36	21	327	23,960	1,808	-
7	2,902	314	118	46	900	31,846	3,147	4,508

Table 6.10: Multiple true candidate's ranks for sources containing multiple plagiarised sections with medium plagiarism obfuscation performed, in the Java 6-gram index using linear drop-off.

The effects of plagiarism obfuscation are also greatly mitigated; table 6.10 exhibits detection for 5 of 6 true candidates, after a medium level of plagiarism obfuscation has been performed. The fifth section of plagiarised content appears to be an outlier, as is consistent with other tests including it such as table 6.7. Plagiarism obfuscation appears to have had a larger effect on

detection, as its true rank has decreased an order of magnitude more, compared to each other file when unmodified as in table 6.9.

For a file containing multiple instances of plagiarism, only one true candidate would have to be retrieved in order for plagiarism to be detected during detailed analysis, and therefore alerted to a user. If a user were to identify plagiarism within a source code file, then the number of plagiarised programs would not necessarily change any resultant outcomes of plagiarism detection.

As mentioned, linear drop-off was initially not used due to hypothesis that plagiarism detection from large programs would be significantly impaired, as large programs are likely to have higher *n*-gram frequencies. Multiple very large programs were therefore plagiarised from, with different similarity functions used as can be seen in table 6.11.

Lines in indexed program	Tokens in indexed program	Plagiarised tokens in source file	True rank from similarity function		
			Minimum	Power law drop-off	Linear drop-off
992	3,212	217	1	1	1
2,237	10,757	280	1	937	456
5,224	20,082	394	1	2	6
5,239	98,634	374	1	27	23

Table 6.11: Table showing the true ranks for large programs from which subsections were plagiarised from, for different similarity functions.

Table 6.11 demonstrates the detection of small plagiarised sections from very large files, with thousands of lines of code. As shown, variance exists within the ranks of true candidates, depending on the uniqueness of the plagiarised sections. The sections plagiarised were the bottom-most functions containing logical code within the large files, in an effort to prevent any bias. As expected, the minimum function will always assign a similarity of 1 to any unmodified code, and therefore will obtain a true rank of 1. However, linear drop-off performed comparably to power law drop-off, rejecting the initial hypothesis. Both drop-off functions assigned low true ranks, demonstrating how detection of plagiarised code even from very large programs is feasible. The average time taken for each candidate retrieval was less than 10 seconds.

6.6.4 Effect

Consequently, candidate retrieval was chosen to utilise a similarity function with linear drop-off. This was implemented as a piecewise function, as shown in figure 6.11. As demonstrated, the use of linear drop-off greatly reduces the noise present at a large scale as a consequence of large files and the number of indexed files. Whilst unmodified plagiarism would obtain higher similarities with the *minimum* function, the trade-off for detecting multiple sources, as well as the ability to effectively mitigate plagiarism obfuscation was considered worthwhile. As drop-off is performed on a per-*n*-gram basis within the similarity function, detection of distinct subsections of plagiarised code is also unaffected.

$$\mathrm{sim}(f_{p,\,g}, f_{s,\,g}) = \begin{cases} f_{p,\,g} & f_{p,\,g} \leq f_{s,\,g} \\ 2f_{s,\,g} - f_{p,\,g} & f_{s,\,g} < f_{p,\,g} < 2f_{s,\,g} \\ 0 & f_{p,\,g} \geq 2f_{s,\,g} \end{cases}$$

Figure 6.11: The final similarity function used to compare *n*-gram frequencies during candidate retrieval, utilising linear drop-off.

Table 6.12 shows the same program queried as in table 6.4, except using linear drop-off instead of the *minimum* function. As shown, even after heavy plagiarism obfuscation has been performed, the true candidate is still ranked first, whereas in table 6.4 the ranks were 1,231, 536 and 224 for 4, 5 and 6-grams respectively. As shown, the similarities are lower with linear drop-off, however true ranks are still lower due to other indexed programs being penalised to a greater extent than the true candidate, as desired. Notably, the decrease in similarity is different for each value of n. Average similarities decreased by 0.22, 0.12 and 0.06 for 4, 5 and 6-grams respectively. This resulted in the similarities for 6-grams being higher than the lower n-values, in some cases.

File Extension	Number of tokens in file	Metric	n	Plagiarism Obfuscation		
				Light	Medium	Heavy
.java	472	Similarity	4	0.725	0.606	0.504
			5	0.797	0.649	0.518
			6	0.821	0.648	0.524
		Rank	4	1	1	1
			5	1	1	1
			6	1	1	1

Table 6.12: Varying program similarity and ranks compared to the actual Java true candidate, for different levels of plagiarism obfuscation when using linear drop-off.

This may be reflective of the type of plagiarism obfuscation performed. For example, modification to complex expressions, which are typically longer in nature, could be less likely when performing plagiarism obfuscation. This is as such obfuscation is more attentive and therefore potentially time-consuming, when compared to plagiarism obfuscation via syntactic yet semantic preserving changes, such as adding braces to one-line statements or swapping adjacent variable definitions. Therefore, longer sections may be more frequently unmodified, whereas token alterations would introduce more 4 and 5-grams than 6-grams. Such changes were therefore likely made to the program evaluated within table 6.12.

Similar evaluation as discussed in this section occurred for other indexes, however primarily Java examples have been included so far for simplicity, as the Java indexes contain the most files. Overall, candidate retrieval demonstrates the ability to reduce the entire source code database to a small set of potential candidates, from which the most likely plagiarised programs could be identified and presented to users. Candidate retrieval preserves relevant programs whilst being resistant to plagiarism obfuscation, the presence of multiple other plagiarised sources, and the inclusion of code from very large files.

Furthermore, this was performed in a performant manner, with candidates being identifiable in real-time, often within tens of seconds. Scalability has also been demonstrated, as a result of careful consideration with regards to overall time complexities and constant factors, in addition to the potential for indefinite parallelization due to partitioning and bulk operations.

Chapter 7

Detailed Analysis

Once candidates have been retrieved, detailed analysis can be performed to identify suspected instances of plagiarism, to present to users. Users can then manually review the identified code sections to determine if plagiarism has occurred. As the candidate set size is far smaller than the number of indexed files, less efficient comparison techniques are able to be used whilst allowing a performant overall system. This allows a higher degree of selectivity than candidate retrieval, therefore reducing the number of potential plagiarised files to a few, or none if no potential plagiarism is detected.

Therefore, existing source code plagiarism detection software was used for detailed analysis. JPlag was chosen for numerous reasons. Firstly, it is a well-known and researched plagiarism detector [19], having been under development since 1996 [11]. As a result, various analyses of JPlag's performance have been conducted, which are therefore applicable for this project's detailed analysis stage [14]. Secondly, JPlag supports detection of Java, and C++, so consequently C and header files too, as discussed in section 2.3.3. Whilst not currently supporting Haskell, JPlag's text plagiarism detection could be utilised, or a heuristic to return top candidates could be employed. However, JPlag is extensible with regards to language support, and so a new parser could be implemented to enable Haskell detection. Of course, JPlag is not a requirement, and so future systems utilising candidate retrieval could implement detailed analysis in a manner to best suit their use case.

Lastly, JPlag's source code is easily accessible via a GitHub repository [11], and is licensed under the GNU General Public License v3.0 (GPL-3). GPL-3 permits modification with certain restrictions, none of which are prohibitive for this project [79]. Therefore, JPlag can be modified to suit the requirements for detailed analysis.

7.1 Candidate Comparison

Traditional use of plagiarism detection software such as JPlag provides a corpus of programs, in the hope of identifying any potential collusion. Therefore, each provided program is compared with each other in a pairwise fashion, as collusion between any given program pairs could have occurred. This results in $\frac{N(N-1)}{2}$ total comparisons, where N is the number of submitted programs; $\mathcal{O}(N^2)$.

As JPlag will be used to identify any plagiarism in a source file from within a candidate set, pairwise comparison of every single program is not required. Instead, only N comparisons are needed to be performed, where N is the size of the provided candidate set, equal to T as previously discussed. The time complexity with regards to program comparisons can therefore be reduced to $\mathcal{O}(N)$.

Consequently, JPlag will be able to compare many more programs in the same amount of time, as a result of each candidate only needing to be compared to the source once. For example, a candidate set containing 10,000 programs would only require the same number of comparisons as a corpus containing 142 programs. This demonstrates how a large number of retrieved candidates is feasible, and comparable to existing use of JPlag for collusion.

7.1.1 Implementation

JPlag was cloned from its GitHub repository to the development VM used for this project, permitting modifications to be easily made and allowing the resultant compiled executable to be directly ran.

When ran from a command line, JPlag receives a directory which contains the programs to be compared for plagiarism. As with the Haskell tokenizer, JPlag would be invoked via Python's subprocess library, thereby allowing a central program to coordinate each step in plagiarism detection. After the candidates have been retrieved for a given source, their contents can be written to a temporary directory to provide to JPlag. As the candidate set simply consists of each indexed program's ID, the source code table can be queried with the IDs to retrieve the required source code contents and URLs. This allows URLs to be provided to users in the case of identified plagiarism, without a secondary database transfer needed, reducing overhead. Whilst the contents of each candidate could potentially be passed to JPlag through inter-process communication or loaded within JPlag directly, this approach required minimal modification to JPlag, whilst in practice having little overhead.

After parsing the contents of each file in the provided directory, JPlag generates the program pairs to be compared. As discussed, this usually results in $\frac{N(N-1)}{2}$ total pairs when detecting collusion. In order to efficiently detect plagiarism from candidates, the generation of comparison pairs was therefore modified to only generate N pairs, enumerating the candidates. The source file was included within the provided directory to JPlag, and used a different filename format for identification. Therefore, the source file was present in each compared pair, and each candidate would only be compared once.

7.1.2 Candidate Set Size

Whilst the candidate set size T is influenced by both candidate retrieval and detailed analysis, the value was chosen primarily based on the performance of detailed analysis. As discussed, a value of 10,000 or higher is feasible within a system desiring maximal recall from candidate retrieval. However, time taken to perform detailed analysis for larger candidate sets was notably higher, hindering evaluation. For example, JPlag took ~3 minutes with 10,000 candidates. A real-time speed was desired, and therefore different values of T were tested. JPlag took ~25 seconds for 2,000 candidates, which was deemed an appropriate trade-off between candidate set size and detailed analysis performance.

7.1.3 Candidate Parsing

However, the time taken to actually compare each program pair within JPlag is minimal; for 10,000 candidates, comparisons took just 3 seconds. The majority of time is instead spent parsing each file. As many large indexed programs can be included in candidate sets, parsing files may take longer than expected, compared to traditional detection of collusion where files may be shorter. The average number of lines within each candidate was 547 for T = 2,000, whereas for T = 10,000, each candidate had 603 lines on average. Whilst not significantly higher, this

demonstrates how larger files can be included within larger candidate sets, disproportionally increasing JPlag's parsing time.

The overhead of parsing each candidate could be mitigated by storing the parsed format which JPlag requires. However, this would likely require significant additional storage, so may not be pragmatic, and was not explored for this project. As a consequence of choosing 2,000 candidates to be retrieved, the entire plagiarism detection process for an input source program would typically finish in less than a minute.

7.2 Results

After JPlag finishes pairwise comparisons, the results are written to a '*results*' directory. The generation of the results directory was unmodified and therefore could be used as for collusion detection, which may be beneficial if a user already has experience with JPlag. As the results directory consists of JSON files, these can be parsed within the primary Python program.

7.2.1 Similarities

JPlag assigns a similarity to each program pair, which in this case will be a similarity of each candidate to the source program. The results directory contains an overview with each candidate's similarity, with those above a certain threshold output to users, along with their URL as discussed. Matching sections of similar programs are highlighted, so that users can immediately review potential plagiarised sections, whilst being able to factor in elements such as variable names, comments and formatting, which are not used during automated plagiarism detection.

The similarity threshold of 25% was chosen to output such matching sections, based on initial tests, however this threshold could be modified by users based on their use cases. Program similarities above 10% are also output, allowing further investigation of any potential plagiarised programs with similarities close to the chosen threshold.

The threshold of 25% is lower than most typical use cases for JPlag. However, average similarities within a candidate set are low, due to most candidates implementing completely different functionality. This differs to identification of collusion, where each program in the provided corpus would typically implement solutions to the same problem, such as for a programming assignment. For example, the average JPlag similarity of 2,000 candidates was 3.28%, for an arbitrary chosen indexed program containing ˜1,500 tokens.

Table A.14 shows the similarities JPlag assigns to files including plagiarised code from multiple sources, generated from the evaluation shown in table 6.9. As can be seen, JPlag correctly detects all candidates with a similarity above 25% for the source containing 5 plagiarised programs, of which the identified matching segments are then output, along with their respective URLs. For the larger composed sources, 5 of 6 and 4 of 7 candidates have similarities above 25%. The average similarities are also shown in table A.14, with the true candidates in the largest file having an average similarity of 25.04%. This demonstrates how JPlag can correctly identify matching sections between a source file and multiple plagiarised programs, even for a source containing thousands of tokens.

When repeated for plagiarised subsections of programs, the average similarity assigned by JPlag for all 7 candidates dropped to 23.89%, for when the source file contained 3,447 tokens whilst the plagiarised programs totaled 8,522 tokens. For 5 candidates, each matching section was still correctly identified from the candidate set, as shown in table A.15. This demonstrates how candidates do not have to be plagiarised in their entirety for JPlag to accurately detect matching plagiarised sections.

Lines in indexed program	Tokens in indexed program	Plagiarised tokens in source file	True JPlag similarity (%)	Average JPlag candidate similarity (%)	Total time taken (s)
992	3,212	217	11.87	1.56	77
2,237	10,757	280	5.51	0.03	72
5,224	20,082	394	3.41	0.54	81
5,239	98,634	374	1.11	1.11	27

Table 7.1: Table showing the similarities assigned by JPlag for very large true candidates, compared to the average candidate similarity.

7.2.2 Issues

The use of JPlag for detailed analysis does not present a perfect solution, however. JPlag is primarily a tool used to check for collusion, as previously discussed. Therefore, similarities are computed assuming each provided program would be similar in length and contents to another one, if plagiarism has occurred. Similarity is measured over an entire file, as JPlag does not account for one program being the source program or not. Consequently, programs with plagiarised sections from very large programs can obtain low similarities, as shown in table 7.1.

This demonstrates how the similarities assigned by JPlag can be unsuitable for detecting plagiarism from very large sources. Therefore, a different metric or comparison method could be utilised, however was not explored within this project due to the focus on candidate retrieval and the additional time overhead that would be required. In some cases of extremely large programs containing ~100,000 tokens, the similarity JPlag assigns is indistinguishable from the average candidate similarity, despite low true ranks of the original plagiarised program within candidate retrieval, as shown in table 6.11.

Within the complete plagiarism detection system, each component can act as a bottleneck. For example, if a plagiarised program from an online repository is either not crawled, or unable to be tokenized and therefore indexed, then such plagiarism cannot be identified. Likewise, if candidate retrieval fails to retrieve the program, or detailed analysis does not assign a sufficient similarity, then the true plagiarised program would not be flagged for review by the user. Hence, the success rate of the entire system is determined by the success of each component, mandating attention to each step. For example, if inter-lingual plagiarism is successfully identified during candidate retrieval, but the detailed analysis is unable to detect such types of plagiarism, then the overall system will not be capable of raising such plagiarism to users.

7.3 Extensibility

As described, many possible implementations of detailed analysis are possible. This project has primarily focused on candidate retrieval, due to it being a novel technique within the field of source code plagiarism detection. Whilst detailed analysis could be improved, many well-researched tools exist to detect plagiarism on the scale of thousands of files, therefore reducing the need to optimise detailed analysis within this project. Yet, JPlag serves to correctly identify plagiarism from candidates in the majority of cases, and produces an understandable format for human review, making it a satisfactory choice.

The compartmental nature of this project's plagiarism detection system allows detailed analysis to be easily interchanged, as a corpus of candidates can be retrieved within seconds, even at the scale of millions of files. Therefore, detailed analysis can be adapted for any specific use case, even allowing different analysis methods to be utilised for different file types if desired.

Chapter 8

Evaluation

As previously mentioned, the majority of evaluation will focus on candidate retrieval. Within the field of source code plagiarism detection, this technique appears to be novel and has not been researched at the scale presented in this project before. Java files will primarily be evaluated, as over 1.3 million Java files are indexed, demonstrating the largest scale within this project. Haskell files will also be evaluated more than C, C++ and Header files, due to it demonstrating performance for functional languages, and the potential for extension to other paradigms.

When plagiarising code from online software repositories, such code could be incorporated into a source program in multiple ways. For example, the plagiarised code could comprise the entire program, or it could be only a relatively small subsection, with the rest of the source being original code. Moreover, multiple programs could be plagiarised as discussed, in addition to only small sections of code from files available online being plagiarised. Online code could also be translated to another language, either as a form of obfuscation, or out of necessity, for instance if code solutions to an assignment in Java can only be found, whilst the assignment requires C++.

These different forms of plagiarism will be evaluated, in an effort to cover the most likely forms of plagiarism from online code repositories. Whilst plagiarism of multiple sources and subsections of large files has already been partially evaluated in chapter 6, additional forms will be reviewed in this chapter further, along with evaluation for each generated index.

8.1 Plagiarism Obfuscation Levels

As demonstrated in section 6.5, evaluation on the effect of plagiarism obfuscation can be performed. Therefore, different levels of obfuscation were performed to various files. The levels used for evaluation consisted of light, medium, and heavy plagiarism obfuscation, in addition to unmodified plagiarism. Each of these obfuscation levels are inclusive of the changes from the previous level, thereby ensuring that each level is progressively more different and therefore harder to detect than the previous. As candidate retrieval is immune to superficial changes such as variable renaming or simplistic expression re-ordering as presented in section 5.4, these were not performed. Thus, only significant structural changes which would actually affect detection were performed, and 'unmodified' code could be assumed to have had superficial changes performed.

Light obfuscation included techniques such as adding or removing braces from one line statements, reordering variable or function definitions, inlining or outlining variable definitions, rearranging statements whilst preserving semantics, inverting certain operators, and turning multiple statements into a single compound expression. Medium obfuscation included more of the aforementioned techniques, as well as replacing 'for' loops with 'while' loops and vice versa, reordering if/else bodies and conditions, modifying loop termination, re-ordering arithmetic and

boolean expressions, and compressing or expanding constant statements.

Heavy plagiarism obfuscation performed more of the aforementioned techniques. However, medium plagiarism obfuscation was deemed to be extensive by itself, whilst being time-consuming to actually perform to the extent that pragmatic applications were questionable. Consequently, the majority of evaluation extended to the level labelled medium. Of course, such terms are subjective and may differ between research, as well as variation existing within each level due to the techniques performed. Table 8.1 shows the effect of the different obfuscation levels for the plagiarised Haskell program originally used in table 6.5.

File Extension	Number of tokens in file	n	Plagiarism Level		
			Light	Medium	Heavy
.hs	361	4	0.732	0.494	0.400
		5	0.734	0.525	0.406
		6	0.736	0.523	0.381

Table 8.1: Varying program similarities compared to the actual Haskell candidate, for different levels of plagiarism obfuscation, using linear drop-off. The true rank was 1 for each file.

Table 8.1 does not show unmodified code, which has a similarity of 1, mathematically following from the equation presented in figure 6.7 when each n-gram frequency is identical. As demonstrated, the similarity compared to the actual candidate is significantly affected by each plagiarism obfuscation level, as well as not necessarily being equal or ordered when considering different n values. Whilst such similarities are relatively low however, they are still far higher than similarities of other indexed programs, hence the true candidate being correctly identified with a rank of 1 in each case. For 6-grams, the next highest similarities were 0.202, 0.170 and 0.165 for the varying levels of obfuscation, demonstrating how further obfuscation could have been performed whilst still identifying the true candidate with a rank of 1. Of course, even a rank of 1 is not needed during candidate retrieval. The similarity threshold for the candidate set was less than 0.1 for each obfuscation level with 6-grams, illustrating the correct penalisation of different programs via the described similarity comparison techniques. Thus, significant additional obfuscation or other modifications could be performed to the already obfuscated code, whilst still correctly retrieving the true candidate and therefore facilitating plagiarism detection.

8.2 Multiple Sources

As initially described in chapter 3, the ability to detect multiple plagiarised sources was desired. This has been shown to be feasible through the evaluation methods outlined in section 6.5.2. As shown in table 6.10, the ability to detect multiple plagiarised sources which have each had medium-level plagiarism obfuscation is also possible. This evaluation also performed for a light level of plagiarism obfuscation on each source, as shown in table A.16. As expected, performance was worse than unmodified plagiarism, and worse than medium-level obfuscation. For the composition of 5 plagiarised Java sources, each was retrieved, in addition to all 7 programs in the largest file having true ranks below 10,000, such that retrieval of each within a performant system would be possible.

The method outlined for detecting multiple sources is also greatly beneficial for observing performance of other plagiarism scenarios, as the evaluated files range from containing 1 to 7 plagiarised sources. Consequently, the latter files contain a large amount of code, and so each individual plagiarised source only represents a small section. Thus, evaluation of small plagiarised subsections is incorporated into this method, as can occur during real instances of plagiarism. This evaluation method was therefore repeated for the C, C++ and header indexes.

As mentioned, only 5-gram indexes were generated for C, C++ and header files. Table A.17 shows evaluation using the C 5-gram index, which provides effective plagiarism detection for 5 separate sources, and 4 of 6 sources if a candidate set size of 10,000 were used. The performance for C++ and header files was notably better, however. For header files as shown in table A.18, all 7 true candidates in the largest program had ranks below 10,000. The C++ performance is shown in table A.19; the ranks for each true candidate in the file containing 5 sources were less than 70, with an average of 27. The performance of the various files across the C, C++ and header indexes demonstrates how performance for retrieving both individual candidates, multiple candidates, and candidates comprising a small relative section of the source is possible, in addition to the extensibility for arbitrary languages. Across each of these indexes, the average time taken to retrieve candidates was 12 seconds, with a minimum of 5 seconds and a maximum of 23 seconds. This shows how for each of these indexes containing between 500,000 and 900,000 files, performance was real-time, exemplifying the scalability of candidate retrieval. Performance within the Haskell 6-gram index was notable. Whilst this was expected due to fewer indexed Haskell files, shorter sections were plagiarised, with an average length of 238 tokens as shown in table A.20. Recall was 1 for each plagiarised file except for the largest, which retrieved 6 of 7 candidates with an average rank of 757 and a maximum of 3,013, demonstrating multi-paradigm performance.

This form of evaluation was also repeated for plagiarised subsections of larger Java files. As shown in table 8.2, the 7 plagiarised programs consisted of 8,522 tokens in total, of which 3,447 were plagiarised. Despite this, each candidate had a rank below 10,000 when querying the program composed of them all. As shown, programs 2 and 7 likely contained similar *n*-grams, as the second program's similarity increased when the 7th plagiarised subsection was incorporated. This could occur during real plagiarism, as plagiarised code may be similar to existing original code. Thus, table 8.2 demonstrates how detecting multiple plagiarised subsections is feasible.

Number of plagiarised files in source	Total source tokens	Total tokens in indexed programs	True rank of unmodified plagiarised sources						
			1	2	3	4	5	6	7
1	473	1,047	1	-	-	-	-	-	-
2	1,039	2,452	1	2	-	-	-	-	-
3	1,541	3,534	2	17	1	-	-	-	-
4	2,121	4,890	4	19	2	1	-	-	-
5	2,627	5,967	12	694	1	4	2,354	-	-
6	3,052	7,818	164	5,416	14	232	3,271	2,826	-
7	3,447	8,522	500	426	85	504	5,499	3,874	5,663

Table 8.2: Table showing the rank of multiple true large candidates for sources containing multiple unmodified code subsections, within the Java 6-gram index using linear drop-off.

8.3 Plagiarism from Large Files

As previously shown in table 6.11, plagiarism of small sections from programs containing thousands of lines is able to be detected, with low true ranks. The same evaluation was performed using linear drop-off for Haskell programs, as shown in table 8.3. Notably, the rank of the true candidate is considerably higher than the similarity of the 2,000th ranked candidate, i.e. the similarity threshold of the candidate set. For the programs with true ranks of 6 and 7, the similarity threshold is lower, suggesting that the plagiarised sections contained relatively unique *n*-gram distributions, additionally reflected by the lower true similarities.

Lines in indexed program	Tokens in indexed program	Plagiarised tokens in source file	True candidate rank	True candidate similarity	Candidate similarity threshold	Time taken (s)
1,092	4,639	174	1	0.734	0.083	0.83
2,647	13,575	381	6	0.202	0.056	0.84
2,827	15,266	438	7	0.178	0.037	0.90
1,533	20,017	356	1	0.647	0.120	0.92

Table 8.3: Table showing the true ranks and similarities for plagiarised subsections of large programs, within the Haskell 6-gram index, using linear drop-off.

The same plagiarised programs were evaluated with Haskell's 5-gram index as shown in table A.21. The true candidate ranks dropped to 1, 153, 138 and 1 respectively, showing how performance is worse but still comfortably within the candidate set. As shown, the time taken to retrieve the candidate set is minimal, taking around one second for each Haskell index, due to the smaller overall size compared to the Java indexes. Java Index queries take ~10x longer than the Haskell indexes, due to being ~10x larger, reflective of the linear time increase with regards to the number of indexed programs, as shown in section 6.4.4.

8.4 Small Plagiarised Sections

When performing plagiarism, only a small section of code may actually be incorporated within the resultant source code, for example plagiarising a single function within a larger original program. Whilst this has already been partially evaluated via the files composed of 6 or 7 plagiarised sections within the multiple source tests, such sections were typically larger than 300 tokens, whilst plagiarism could of course occur for a smaller section. Detection of plagiarised programs ranging in size from ~50 to 150 tokens (~10 to 30 lines) were therefore evaluated, when incorporated within larger original programs.

As mentioned, if a queried program solely consisted of unmodified plagiarised code (whether a subsection or not), then the true candidate would be correctly retrieved, even if the code sequence consisted of just 10 tokens. This as as the n-gram frequencies would be identical, therefore yielding a similarity of 1 and consequently retrieving the candidate. In practice, small sections of plagiarised code may be less likely to be significantly structurally modified, as less code can be modified without affecting semantics. Whilst superficial changes like renaming variables may still be likely, this do not affect detection as previously discussed, so can be assumed to have been performed on the evaluated code. Unmodified code sections were therefore evaluated, whilst appending random code sections in order to emulate original code. The chosen code was retrieved from Rosetta Code [80], which hosts code samples in many languages, and was not crawled within this project. The random code chosen contained 1,000 tokens in 128 lines, and when checked within the Java 6-gram index had an average candidate similarity of 0.174, with a threshold similarity of 0.161. It was therefore deemed to be a suitable representation of original code, and was trimmed to be appended in stages to the selected small plagiarised code sections.

Table 8.4 shows how larger plagiarised sections are easier to detect, as expected. The number of lines in the additional appended code were 26, 60, 104 and 128 for the increasing number of tokens, respectively. As shown, detection of plagiarised subsections with more than 100 tokens is feasible when included in an original program consisting of 750 to 1,000 tokens. For plagiarised sections containing ~10 lines of code, roughly 4-5x as much original code would need to be incorporated to fail in retrieving the true candidate. However, this appears to improve with larger plagiarised sections, with 6-7x as much original code needed to obscure plagiarism of ~20

Number of lines in plagiarised section	Number of tokens in plagiarised section	Number of additional tokens appended to program				
		0	250	500	750	1,000
9	63	1	51	47,004	>100k	>100k
11	68	1	6	37,466	>100k	>100k
10	90	1	1	758	22,066	59,474
18	92	1	1	2,731	54,330	>100k
16	130	1	1	23	6,378	23,028
21	134	1	1	8	4,075	18,786
26	163	1	1	2	853	5,503

Table 8.4: Table showing the true rank of small candidates, when incorporated within a larger program, for the Java 6-gram index.

line sections, likely due to the plagiarised code's *n*-gram distribution being more unique within the queried index. Using a candidate set size of 10,000, the recall of the set of small plagiarised sections drops from $\frac{5}{7}$ to $\frac{1}{7}$, for 500 to 1,000 appended tokens.

Evidently, detection of small plagiarised subsections is a challenge for this project's proposed candidate retrieval. Whilst the use of linear drop-off has significant benefit, detection of plagiarised sections fewer than ˜200 tokens can be hindered when included in large original programs. If just a single short function in a large program is plagiarised, then detection may be unlikely. This is due to the larger original *n*-gram distribution obscuring the plagiarised section's distribution, in addition to the likelihood of overlapping *n*-gram distributions increasing, impeding accurate similarity assignment. Yet, future work could mitigate this, as discussed in section 9.2.2.

8.5 Inter-lingual Detection

As previously mentioned, inter-lingual plagiarism is a form of plagiarism whereby code written in one language is translated to another, whether by means of an automated tool or manually, and presented as one's own work. This is challenging to automatically detect in all cases, however certain languages can be easier to detect inter-lingual plagiarism between than others. For example, Java and C++ code can have significant overlap, as the majority of basic syntax and operators are identical. Therefore, if a Java program were to be translated to C++, then the character stream of each program could be similar, if the same tokens within each program were mapped to the same characters. This would consequently produce similar *n*-grams, allowing comparison to be performed in the same manner as previously detailed. Of course, the same is also true for translating C++ to Java. As discussed in section 5.3.3, the C/C++ token map was instantiated with the existing Java tokens, therefore enabling this described method of inter-lingual comparison.

However, this may not work for all cases of inter-lingual plagiarism, as many constructs differ between languages. For instance, accessing a character within a Java string uses the syntax '`string.charAt(index)`', whereas C++ uses the syntax '`string[index]`', therefore generating different streams for the semantically identical expression. Yet, to detect inter-lingual plagiarism during candidate retrieval, the only requirement is that the true candidate's rank is less than the size of the candidate set. Thus, *n*-gram mismatches can be handled to a certain extent.

Inter-lingual plagiarism was therefore performed and evaluated within this project. If Java code is translated to C++, then the resultant C++ file can be tokenized as before with the C++ tokenizer. After the *n*-gram frequencies have been generated, they can then be compared within the Java index, permitting comparison and detection to the true candidate. Of course, in an

actual scenario the knowledge that the provided C++ file was translated from Java is not held. Consequently, indexes for both Java and C++ can be queried with the resultant *n*-gram frequencies, before merging the resulting similarity sets. Hence, candidates can be retrieved in the same fashion as before. This approach was accordingly implemented, and evaluated in table A.22.

In order to perform inter-lingual plagiarism, automatic tools were used to translate Java to C++ [81] and vice versa [82]. The output of the respective tools was not modified before querying within the index. Some of the translated programs were used within prior evaluation, and the larger evaluated files were chosen as they mostly contained logical code. Translation and successful detection was also performed on randomly selected programs during testing. As shown in table A.22, inter-lingual plagiarism detection between Java and C++ (and therefore C/header files) is possible, with larger plagiarised sections being easier to detect as expected. Both Java translated to C++ and vice versa were able to be identified with a true rank of 1, and translated files ranging from 333 to 794 tokens were successfully retrieved.

However, as mentioned, JPlag would act as a bottleneck due to it being unable to detect inter-lingual plagiarism currently. Yet, tools such as XPlag [22] could be used to detect such instances. The performance of inter-lingual plagiarism detection could likely be improved by compiling languages to a common intermediate language as XPlag does, which will be expanded upon further in section 9.2.4.

Whilst this approach works due to the overlap between Java and C/C++ tokens, translation across significantly different languages would be infeasible, such as between Haskell and Java. However, such inter-lingual plagiarism is likely less commonplace, and would usually have to be performed manually, therefore consuming much more time. Despite research into translating Haskell to Java [83], no tools appear to be able to convert Java to Haskell currently.

8.6 Comparison

The performance of JPlag on small corpora has already been evaluated [14], and so has only been briefly evaluated in this thesis. Within the field of source code plagiarism detection, no other comparable method to candidate retrieval as presented in this project has been evaluated. Whilst *Burrows et al* utilised an index, their proposed system employed a similarity threshold to determine similar programs, and was partially evaluated by querying 296 files within a corpus of 61,540 programs [10]. Detection took 1,441.8 seconds, or 4.87 seconds per file.

Across evaluation shown using the Java 6-gram index containing 1,324,892 files, the average time taken for candidate retrieval was 12.88 seconds when querying individual files. Therefore, per queried program this system filtered ˜100,000 files per second on average, compared to ˜10,000 files per second for the *Burrows* paper. Moreover, the figure for this project includes all overhead for each single queried file, instead of distributed over hundreds. Query times per file could be greatly reduced when querying multiple files, and is discussed in section 9.2.5.

8.7 Summary

In total, 187 source files were used or adapted for evaluation, resulting in 462 independent analyses of the performance of the plagiarism detection system. These analyses covered many potential instances of plagiarism for each index, in addition to evaluating the system under different parameters, such as the size of the candidate set and size of *n*-grams. Candidate retrieval has been shown to be scalable and performant, whilst able to detect obfuscated plagiarism with a high recall, including detection of multiple plagiarised sources. Additional techniques and optimisations could still be utilised in future work to improve performance though, such as for detection of small plagiarised subsections in source files.

Chapter 9

Conclusion

In this thesis, a novel source code plagiarism detection system has been presented, which utilises candidate retrieval to reduce corpora containing millions of source code files from online software repositories to thousands. From this small set of candidates, existing research has been employed to identify plagiarised files, in a more selective albeit resource-intensive process referred to as detailed analysis. Thus, high recall of candidate retrieval ensures plagiarised programs are passed to detailed analysis, from which high precision flags such programs to users, ignoring the rest.

The primary contribution of this thesis is the process of candidate retrieval. Whilst this technique exists within the field of traditional text-based plagiarism detection [4], it does not appear to have been applied to source code plagiarism detection before, possibly as plagiarism obfuscation takes vastly different forms for source code, compared to text. Candidate retrieval as described in this report is performant, being able to filter millions of source code files in tens of seconds. It is also scalable, having linear time complexity with regards to the number of total files from which candidate retrieval is performed. Various techniques have been demonstrated to enable far larger scales, such as the use of partitioned indexes, bulk operations, and parallelization. Novel techniques have been implemented to reduce the effect of noise at scale, such as the use of drop-off within similarity assignment, enabling effective detection of obfuscated plagiarism, multiple sources of plagiarism, and inter-lingual plagiarism.

Through the process of program tokenization, several superficial plagiarism obfuscation techniques are also completely prevented, which are easier to perform and therefore potentially more likely than structural code modifications.

Extensibility of the proposed plagiarism detection system has been demonstrated, with additional languages able to be supported, regardless of their paradigm. Likewise, bespoke detailed analysis can be implemented simply, potentially on a per-language level if desired. Whilst this project has utilised source code from online software repositories, the system is generic to the input source, and could be populated with code from prior assignments within academia, prior submissions to online assessments, or files from any other medium. The criteria presented in section 3 are therefore considered to be met, however there is still vast potential for future work.

Within the field of source code plagiarism detection, this is the largest known research conducted for each of the programming languages used, with the next largest study using a corpus of ˜60,000 files [10]. This increase in scale of multiple orders of magnitude is possible as a result of the two-stage candidate retrieval and detailed analysis process, and can be employed at far larger scales with similar time performance, via partitioning and parallelization.

Of course, the proposed solution cannot detect all instances of plagiarism, and many potential improvements to the overall system are possible, for example to aid detection of small plagiarised subsections within larger original source code, and to further mitigate obfuscation. These will be explored further when discussing future work in section 9.2.

9.1 Parameters

For future work utilising candidate retrieval, the number of candidates to retrieve depends on the performance of detailed analysis, and the desired time frame of detection. A candidate set size of 10,000 would be suitable for a reasonably efficient final stage, but values around 2,000 as used within this project should provide sufficient recall in most cases, and would be appropriate for intensive detailed analyses. Larger values could be chosen for performant detailed analyses, or in situations where the time taken is not as important. For instance, retrieval of 100,000 candidates would take the same time during candidate retrieval as 10,000, hence the time frame and consequently candidate set size primarily depends on detailed analysis.

During evaluation, the n value of 6 for n-gram comparisons was found to perform better in general than 4 or 5-grams, as at larger scales, higher n values provide additional distinction of programs. Whilst they can be affected more by plagiarism obfuscation, in practice consistent disruption during obfuscation is both challenging and time-consuming, resulting in remaining structurally consistent fragments being relatively more identifiable with 6-grams. Detection via 4 and 5-grams was still effective though, often resulting in correct candidate retrieval as with 6-grams. Future research could utilise 5-grams as a starting point, as they are performant whilst being less resource-intensive. However, 6-grams should likely be used to optimise recall during candidate retrieval.

9.2 Future Work

9.2.1 *N*-grams

Higher values of n could be investigated, however they become exponentially more resource demanding. Whilst each index and therefore n value was evaluated independently, indexes could be combined when querying a source code file. This could yield the benefits of each n value, for example the relative resistance to plagiarism obfuscation of lower n values, combined with the distinctiveness of higher n values. Whilst this was briefly tested during this project, 6-grams were found to consistently outperform the lower values. However, merging candidate sets from multiple n values could have merit, especially if higher n values were found to be impaired by obfuscation in certain cases. Candidate sets could be merged in many different ways, for example by taking the union of each to produce a larger set, or by averaging similarities across sets to keep the final candidate set the same size, or by retrieving the top T candidates from the union of each set, ranked by similarity.

9.2.2 Subsection Queries

For this project, when querying a program within an index, the entire program's n-gram distribution is considered during comparison. This is effective for retrieving candidates when the entire source is plagiarised, however becomes less capable when plagiarised sections are small, as their n-gram distributions become obscured as discussed. This could be mitigated by querying subsets of the source's n-gram distribution. For example, if a source program contained 5 functions, then the n-gram distribution of each function could be queried independently. The resulting candidate sets could then be merged, possibly using a technique described in section 9.2.1. If only one function was plagiarised, then this approach would allow more effective detection of the plagiarised candidate, as the plagiarised section's n-gram distribution would not be affected. This could also assist detecting multiple plagiarised candidates, if one function were plagiarised from each.

However, if plagiarised content consisted of multiple functions or the entire program, then subsequent detection would be hindered, as the plagiarised section's *n*-gram distribution would be fragmented. Therefore, the power set of a program's functions could be queried, i.e. each possible combination of including a function or not, excluding the empty set. Whilst this would provide optimal function coverage, this approach results in $2^f - 1$ queries where f is the number of functions, and therefore would not be scalable for very large sources.

Plagiarism may also not be local to a function. It could occur within subsections of functions, and functions may themselves consist of thousands of tokens. Instead, the same approach can be used, but with token-based subsets of a program instead of functions. For instance, if a queried program contained 1,000 tokens, then the first and last 500 tokens could be queried, in addition to the section consisting of the 250-750th tokens, in order to account for plagiarised sections continuing over subsection boundaries. As before, the entire program itself could then also be queried, with the resulting candidate sets merged in an appropriate fashion. Querying program subsections could continue for arbitrarily large source programs, thus improving detection of smaller plagiarised sections within larger programs. As is currently performed, all required *n*-gram program frequencies could be retrieved the from the index at once and stored in memory in order to reduce overhead, despite not all frequencies being used for each source program subset.

9.2.3 Index Updates

As mentioned, if an online source file is not indexed when checking a source program for plagiarism, then plagiarism of the online file is not possible. This could be mitigated by allowing users to provide relevant search terms or repositories to be indexed ahead of plagiarism detection, for example when a programming assessment is initially assigned. Repositories provided, similar to those provided, or matching provided search terms can therefore be crawled and the files indexed, allowing their detection. Furthermore, if plagiarism is identified to have been successfully detected from a repository by a user, then similar repositories could be crawled. The challenge of identifying similar online repositories has been studied before [84].

As originally discussed in section 4.3.4, continued modification or addition of code within crawled repositories is not addressed within this project. However, plagiarism could occur from a file which is later modified, added or removed. An additive model for indexing code could be used; no source code would be removed from an index, as prior versions of online files can still be plagiarised from, or plagiarism may have been performed before the online file was modified. However, adding every new version of an online file would be extremely impractical, as changes may be very minor. Therefore, new versions of an indexed file could only be used if their change is deemed significant, a task left for future work. In the case that a previously indexed file is updated to be a superset, i.e. no existing code is changed, then the new version of the file could replace the prior version, as this would minimally affect detection whilst saving vast storage space and query times over many files. Thus, the same program ID could be used, only requiring updates to the index.

Prior crawled repositories could therefore be checked again, allowing plagiarism from new, modified or removed files to be detected. Deciding when to re-crawl a repository is an additional challenge to be addressed, as it may be more beneficial to crawl new repositories in most cases. The additive model described could therefore be used if a repository was identified to be relevant, such as if plagiarism had already been detected from it, as previously described. Heuristics could also be used to determine repository priorities for crawling. For instance, crawling many smaller repositories may be more beneficial than crawling few large ones.

9.2.4 Intermediate Representations

The source code used within this project has been tokenized directly, without any alterations performed other than minor preprocessing for C/C++ files. However, performing tokenization on an optimised intermediate representation of such source code may be beneficial, whether via LLVM [85] or to RTL, as discussed in section 2.3.2. Using an optimised intermediate representation would remove superfluous code, in addition to inlining functions or variables when deemed beneficial. Different loop structures could also be compiled to the same representation. Consequently, compilation of semantically equal but syntactically different source codes would be more uniform on average, and so detection would be more resistant to plagiarism obfuscation strategies.

As compilation to an intermediate representation would occur before tokenization, no other component described in this project would require change. Compiling multiple languages to the same intermediate representation would also likely improve inter-lingual plagiarism detection, as has been previously exploited [22]. Whilst it may not be possible to compile all languages to the same representation, as each language's indexes are independent, compilation would not have to be performed for each indexed language, and instead could be enacted on a per-language basis.

9.2.5 Pragmatic Extensions

There are a number of extensions which could enable further pragmatic use. Currently, only one source file is queried within indexes at once, but this could be easily extended to support querying of corpora. Additional overhead could be reduced by querying the *n*-grams of all provided source files at the same time, thus reducing the bottleneck of database retrieval. Faster database options such as Redis could also reduce data transfer times. Furthermore, as stored source code is only accessed after candidate retrieval, access is infrequent. Therefore, cold storage devices [86] or other solutions could be used to store source code in a cost-effective manner, as scale increases.

The ability to remove template code from provided files could be added. This could be done by simply subtracting the *n*-gram frequencies of provided template code and overlapping sections from each source, as *n*-gram frequencies are additive, as discussed in section 6.1.6.

9.2.6 Horizontal Scaling

As described, partitions of each index are independent and so are queried in parallel, dividing query times by up to the number of parallel processes. Whilst only one machine with 4 threads was utilised in this project, index partitioning and queries could be distributed across arbitrarily many machines, therefore linearly reducing the time taken to perform candidate retrieval. As demonstrated during evaluation, an index containing approximately 100,000 files can be queried in 1 second, whilst 1,000,000 files takes roughly ten seconds. Despite these times being suitably performant, it could be assumed that 100,000,000 indexed files would take around 1,000 seconds to query, which may be too slow in certain use cases.

Therefore, partitions could be distributed across multiple machines, thereby scaling the system horizontally. A main thread would then distribute a program's *n*-gram frequencies across each machine, which would then compute the candidate set in parallel across each partition, as within this project. As can be common within distributed systems, network transfer might represent a bottleneck. Detailed analysis could therefore also be performed on each machine, such that only likely plagiarised program IDs or source code would have to be transferred, thus keeping candidate retrieval local to each machine. This would also allow a larger overall candidate set to be utilised, as candidates would themselves be distributed. Of course, such a system would likely only be required at a very large scale, and indexes containing even tens of millions of files would be performant via vertical scaling, for example by using a machine with 16 or more threads.

Bibliography

[1] University of Oxford. Definition of Plagiarism. https://www.ox.ac.uk/students/academic/guidance/skills/plagiarism, 2022. Accessed: 2022-01-27.

[2] Ramesh R Naik, Maheshkumar B Landge, and C Namrata Mahender. A review on plagiarism detection tools. *International Journal of Computer Applications*, 125(11), 2015.

[3] Kevin W Bowyer and Lawrence O Hall. Reducing effects of plagiarism in programming classes. *Journal of Information Systems Education*, 12(3):141, 2001.

[4] Tomáš Foltýnek, Norman Meuschke, and Bela Gipp. Academic plagiarism detection: a systematic literature review. *ACM Computing Surveys (CSUR)*, 52(6):1–42, 2019.

[5] Paul Darbyshire and Stephen Burgess. Strategies for dealing with plagiarism and the web in higher education. *Journal of Law and Governance*, 1(4), 2006.

[6] Georgina Cosma and Mike Joy. Towards a definition of source-code plagiarism. *IEEE Transactions on Education*, 51(2):195–200, 2008.

[7] Matija Novak. Review of source-code plagiarism detection in academia. In *2016 39th International convention on information and communication technology, electronics and microelectronics (MIPRO)*, pages 796–801. IEEE, 2016.

[8] Fintan Culwin, Anna MacLeod, and Thomas Lancaster. Source code plagiarism in uk he computing schools. *Issues, Attitudes and Tools, South Bank University Technical Report SBU-CISM-01-02*, 2001.

[9] Thomas Lancaster and Fintan Culwin. A comparison of source code plagiarism detection engines. *Computer Science Education*, 14(2):101–112, 2004.

[10] Steven Burrows, Seyed MM Tahaghoghi, and Justin Zobel. Efficient plagiarism detection for large code repositories. *Software: Practice and Experience*, 37(2):151–175, 2007.

[11] JPlag - Detecting Software Plagiarism. https://github.com/jplag/JPlag, 2022. Accessed: 2022-01-20.

[12] Sherlock - Plagiarism Detection Software. https://warwick.ac.uk/fac/sci/dcs/research/ias/software/sherlock/, 1999. Accessed: 2022-01-20.

[13] MOSS - A System for Detecting Software Similarity. https://theory.stanford.edu/~aiken/moss/, 2021. Accessed: 2022-01-24.

[14] Lutz Prechelt, Guido Malpohl, Michael Philippsen, et al. Finding plagiarisms among a set of programs with jplag. *J. UCS*, 8(11):1016, 2002.

[15] Håvard Skaar and Hugo Hammer. Why students plagiarise from the internet: The views and practices in three norwegian upper secondary classrooms. *International Journal for Educational Integrity*, 9(2), 2013.

[16] Robert J Youmans. Does the adoption of plagiarism-detection software in higher education reduce plagiarism? *Studies in Higher Education*, 36(7):749–761, 2011.

[17] Mike Joy, Georgina Cosma, Jane Yin-Kim Yau, and Jane Sinclair. Source code plagiarism—a student perspective. *IEEE Transactions on Education*, 54(1):125–132, 2010.

[18] Georgina Cosma and Mike Joy. Source-code plagiarism: an academic perspective. *Department of Computer Science, The University of Warwick*, 2006.

[19] Oscar Karnalim, William Chivers, et al. Similarity detection techniques for academic source code plagiarism and collusion: a review. In *2019 IEEE International Conference on Engineering, Technology and Education (TALE)*, pages 1–8. IEEE, 2019.

[20] Edward L Jones. Metrics based plagarism monitoring. *Journal of Computing Sciences in Colleges*, 16(4):253–261, 2001.

[21] Lefteris Moussiades and Athena Vakali. Pdetect: A clustering approach for detecting plagiarism in source code datasets. *The computer journal*, 48(6):651–661, 2005.

[22] Christian Arwin and Seyed MM Tahaghoghi. Plagiarism detection across programming languages. In *Proceedings of the 29th Australasian Computer Science Conference-Volume 48*, pages 277–286. Citeseer, 2006.

[23] Codequiry - Code Plagiarism & Similarity Checker. `https://codequiry.com/`, 2018. Accessed: 2022-01-20.

[24] Github. `https://github.com/`, 2022. Accessed: 2022-01-22.

[25] Stack Overflow. `https://stackoverflow.com/`, 2022. Accessed: 2022-01-22.

[26] code2vec: Learning distributed representations of code. `https://code2vec.org/`, 2022. Accessed: 2022-04-06.

[27] Code2vec - Github page. `https://github.com/tech-srl/code2vec`, 2022. Accessed: 2022-04-06.

[28] Anupriya Prasad. *Code Clone Detection Using Code2Vec*. University of California, Irvine, 2020.

[29] David Gitchell and Nicholas Tran. Sim: a utility for detecting similarity in computer programs. *ACM Sigcse Bulletin*, 31(1):266–270, 1999.

[30] Jurriaan Hage, Brian Vermeer, and Gerben Verburg. Plagiarism detection for haskell with holmes. In *CSERC*, pages 19–30, 2013.

[31] GitLab. `https://about.gitlab.com/`, 2022. Accessed: 2022-01-22.

[32] Google Code. `https://code.google.com/archive/d/code.google.com`, 2022. Accessed: 2022-01-22.

[33] GeeksforGeeks. `https://www.geeksforgeeks.org/`, 2022. Accessed: 2022-01-22.

[34] Bela Gipp. Citation-based plagiarism detection. In *Citation-based plagiarism detection*, pages 57–88. Springer, 2014.

[35] Python. https://www.python.org/, 2022. Accessed: 2022-05-22.

[36] MySQL - relational database. https://www.mysql.com/, 2022. Accessed: 2022-05-22.

[37] SQLite - SQL database engine. https://www.sqlite.org/, 2022. Accessed: 2022-05-22.

[38] MongoDB - document-oriented database program. https://www.mongodb.com/, 2022. Accessed: 2022-05-22.

[39] Redis - in-memory data structure store. https://redis.io/, 2022. Accessed: 2022-05-22.

[40] Neo4j - graph-based transactional database. https://neo4j.com, 2022. Accessed: 2022-05-22.

[41] Redis - Hardware Requirements. https://docs.redis.com/latest/rs/administering/designing-production/hardware-requirements/, 2022. Accessed: 2022-05-22.

[42] MongoDB Production Notes - Kernel and File Systems. https://www.mongodb.com/docs/manual/administration/production-notes/#kernel-and-file-systems, 2022. Accessed: 2022-05-22.

[43] PyMongo - Python library for working with MongoDB. https://pymongo.readthedocs.io/en/stable/, 2022. Accessed: 2022-05-22.

[44] MongoDB FAQ - Lock granularity. https://www.mongodb.com/docs/manual/faq/concurrency/#how-granular-are-locks-in-mongodb-, 2022. Accessed: 2022-05-19.

[45] MongoDB - Indexes. https://www.mongodb.com/docs/manual/indexes/#footnote-b-tree, 2022. Accessed: 2022-05-22.

[46] GitHub public repository search - "Showing 53,417,874 available repository results". https://github.com/search?q=is:public, 2022. Accessed: 2022-05-21.

[47] GitHub REST API - Repositories. https://docs.github.com/en/rest/repos, 2022. Accessed: 2022-05-16.

[48] PyGithub. https://pygithub.readthedocs.io/en/latest/introduction.html, 2022. Accessed: 2022-05-16.

[49] GitHub Acceptable Use Policies - Information Usage Restrictions. https://docs.github.com/en/site-policy/acceptable-use-policies/github-acceptable-use-policies#7-information-usage-restrictions, 2022. Accessed: 2022-05-21.

[50] pycparser - parser for the C language, written in pure Python. https://github.com/eliben/pycparser, 2022. Accessed: 2022-05-22.

[51] javalang - pure Python library for working with Java source code. https://github.com/c2nes/javalang, 2022. Accessed: 2022-05-22.

[52] Haskell - Usage within education. https://wiki.haskell.org/Haskell_in_education, 2022. Accessed: 2022-01-20.

[53] GitHub REST API Resources - Rate Limits. https://docs.github.com/en/rest/overview/resources-in-the-rest-api#rate-limiting, 2022. Accessed: 2022-05-20.

[54] PyGithub - search_repositories function. https://pygithub.readthedocs.io/en/latest/github.html#github.MainClass.Github.search_repositories, 2022. Accessed: 2022-05-20.

[55] GitHub REST API - Search. https://docs.github.com/en/rest/search, 2022. Accessed: 2022-05-20.

[56] GitHub REST API - Search Repositories. https://docs.github.com/en/rest/search#search-repositories--parameters, 2022. Accessed: 2022-05-21.

[57] GitHub Terms of Service - API Terms. https://docs.github.com/en/site-policy/acceptable-use-policies/github-acceptable-use-policies#7-information-usage-restrictions, 2022. Accessed: 2022-05-21.

[58] GitHub REST API - Repository Contents. https://docs.github.com/en/rest/repos/contents#get-repository-content, 2022. Accessed: 2022-05-21.

[59] MongoDB Documentation - ObjectId. https://www.mongodb.com/docs/manual/reference/method/ObjectId/, 2022. Accessed: 2022-05-22.

[60] Screen - multiprocess terminal manager for Linux. https://www.gnu.org/software/screen/manual/screen.html#Overview, 2022. Accessed: 2022-05-22.

[61] 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*, pages 76–85, 2003.

[62] GNU - The C Preprocessor. https://gcc.gnu.org/onlinedocs/cpp/, 2022. Accessed: 2022-05-27.

[63] Microsoft Data Encoding Docs - Byte order marks. https://docs.microsoft.com/en-us/globalization/encoding/byte-order-mark, 2022. Accessed: 2022-05-27.

[64] Language.Haskell.Exts - Manipulating Haskell source: abstract syntax, lexer, parser, and pretty-printer. https://hackage.haskell.org/package/haskell-src-exts-1.17.1/docs/Language-Haskell-Exts.html, 2022. Accessed: 2022-05-25.

[65] Linux Manual Page - fork. https://man7.org/linux/man-pages/man2/fork.2.html, 2022. Accessed: 2022-06-03.

[66] Haskell.org - Type Classes and Overloading. https://www.haskell.org/tutorial/classes.html, 2022. Accessed: 2022-05-27.

[67] Haskell.org - Dollar Operator. https://wiki.haskell.org/$, 2022. Accessed: 2022-05-27.

[68] Haskell.org - Data.Maybe. https://hackage.haskell.org/package/base-4.16.1.0/docs/Data-Maybe.html, 2022. Accessed: 2022-05-28.

[69] MongoDB Driver - B-Tree Implementation. https://github.com/datacratic/mongo-cxx-driver/blob/master/src/mongo/db/btree.h, 2022. Accessed: 2022-05-29.

[70] MongoDB - Cursor Timeouts. https://www.mongodb.com/docs/v4.4/reference/method/cursor.noCursorTimeout/#session-idle-timeout-overrides-nocursortimeout, 2022. Accessed: 2022-06-02.

[71] MongoDB - Bulk Write Operations. https://www.mongodb.com/docs/manual/core/bulk-write-operations/, 2022. Accessed: 2022-06-02.

[72] MongoDB - BSON Document Size Limit. https://www.mongodb.com/docs/manual/reference/limits/#mongodb-limit-BSON-Document-Size, 2022. Accessed: 2022-06-02.

[73] Subprocess - POpen. https://docs.python.org/3/library/subprocess.html#subprocess.Popen, 2022. Accessed: 2022-06-03.

[74] MongoDB - $in operator. https://www.mongodb.com/docs/manual/reference/operator/query/in/, 2022. Accessed: 2022-06-05.

[75] Python heapq library - nlargest function. https://docs.python.org/3/library/heapq.html#heapq.nlargest, 2022. Accessed: 2022-06-06.

[76] Python heapq library - nlargest function. https://github.com/python/cpython/blob/d1e2e0e1b2af10ddecc5a6a0f9f4ab19ee8a0036/Lib/heapq.py#L397-461, 2022. Accessed: 2022-06-06.

[77] Python Multiprocessing library - Pool function. https://docs.python.org/3/library/multiprocessing.html#multiprocessing.pool.Pool, 2022. Accessed: 2022-06-09.

[78] LeetCode - Algorithmic problem platform to prepare for interviews. https://leetcode.com/, 2022. Accessed: 2022-06-10.

[79] GPL - General Public License. https://www.gnu.org/licenses/gpl-3.0.en.html, 2022. Accessed: 2022-01-20.

[80] Rosetta Code - Programming Chrestomathy Site. https://www.rosettacode.org/wiki/Rosetta_Code, 2022. Accessed: 2022-06-15.

[81] KalkiCode - Online Java to C++ Converter. https://kalkicode.com/ai/java-to-cplusplus-converter-online, 2022. Accessed: 2022-06-15.

[82] C++ to Java Converter. https://c-to-java-converter-free-edition.software.informer.com/, 2022. Accessed: 2022-06-15.

[83] R Barefield. A haskell-to-java translation tool. *Baruch College*, 2006.

[84] Yun Zhang, David Lo, Pavneet Singh Kochhar, Xin Xia, Quanlai Li, and Jianling Sun. Detecting similar repositories on github. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 13–23. IEEE, 2017.

[85] The LLVM Compiler Infrastructure. https://llvm.org/, 2022. Accessed: 2022-06-17.

[86] Renata Borovica-Gajić, Raja Appuswamy, and Anastasia Ailamaki. Cheap data analytics using cold storage devices. *Proceedings of the VLDB Endowment*, 9(12):1029–1040, 2016.

Appendix A
Appendix

[1] In thousands, e.g. 1,234,567 means 1,234,567,000

Repository Type	Number of repositories
Java	3,622
C	1,077
C++	1,492
Haskell	1,017
Total	7,208

Table A.1: Analysis of the number of crawled repositories

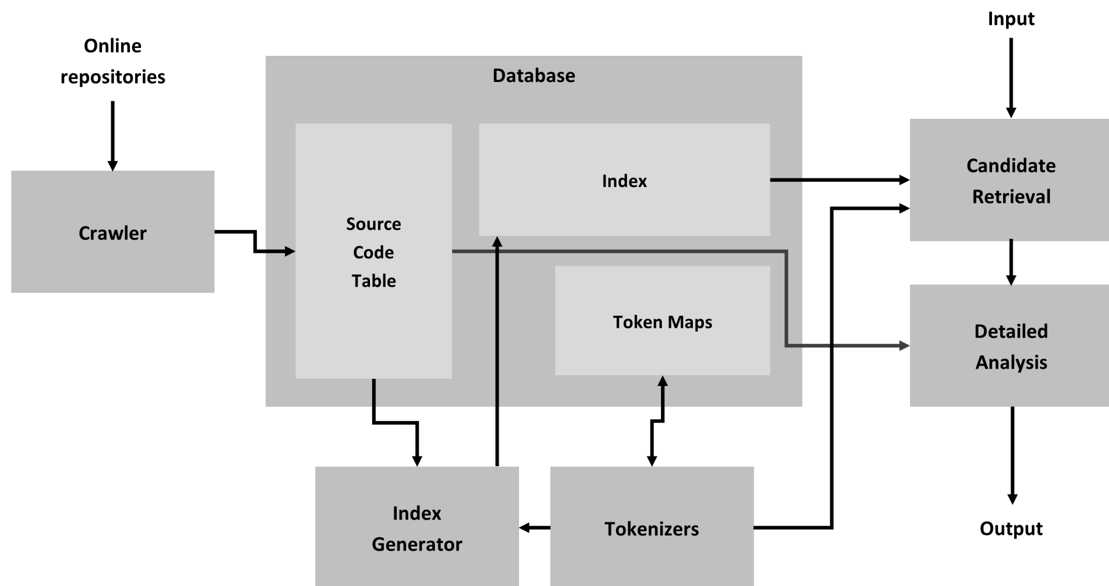

Figure A.1: Overview of the components within the plagiarism detection system. Arrows indicate one component being used by another.

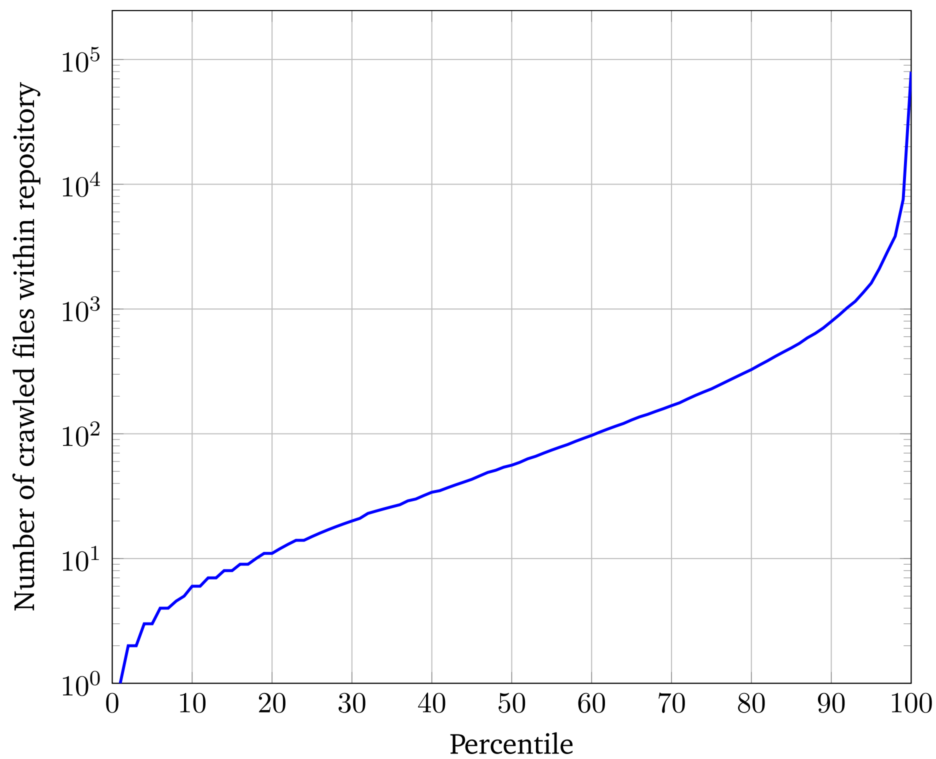

Figure A.2: Distribution of the number of crawled files within each crawled repository.

Number of files within repositories	Repository Type				All repositories
	Java	C	C++	Haskell	
Min	1	1	1	1	1
Max	52,370	52,328	79,816	26,782	79,816
Mean	381.8	808.9	676.0	129.3	470.9
Median	49.0	85.0	140.0	30.0	59.0

Table A.2: Analysis of the number of files within crawled repositories

Number of chars	File Type					All files
	Java	C	C++	Header	Haskell	
Min	0	0	0	0	0	0
Max	1,047,412	1,045,476	1,044,052	1,047,566	1,038,789	1,047,566
Mean	5,788.9	16,845.8	11,759.0	8,644.9	6,070.5	9,159.7
Median	2,628.4	7,038.0	4,057.0	2,831.0	2,778.0	3,230
Total[1]	7,669,625	8,604,295	6,336,875	7,685,564	791,647	31,088,006

Table A.3: Analysis of the number of characters within each crawled file

Number of lines	File Type					All files
	Java	C	C++	Header	Haskell	
Min	0	0	0	0	0	0
Max	38,526	47,981	196,613	59,331	21,879	196,613
Mean	156.8	555.3	344.7	224.4	159.5	264.4
Median	77.0	247.0	129.0	86.0	83.0	98.0
Total[1]	207,693	283,631	185,778	199,489	20,794	897,387

Table A.4: Analysis of the number of lines within each crawled file

Number of tokens	File Type					All files
	Java	C	C++	Header	Haskell	
Min	0	0	0	0	0	0
Max	320,285	405,326	655,510	468,218	64,702	655,510
Mean	722.8	2,681.8	1,861.8	964.0	529.0	1,255.6
Median	303.0	1036.0	609.0	251.0	229.0	369.0
Total[1]	956,525	1,351,048	986,463	823,967	62,226	4,180,231

Table A.5: Analysis of the number of tokens within each crawled file

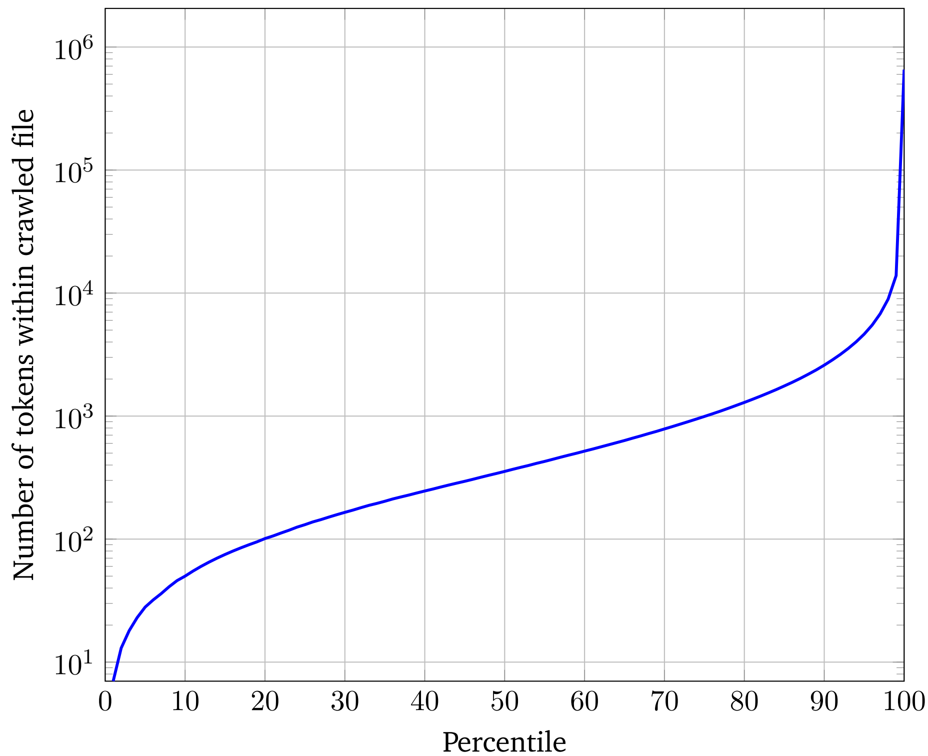

Figure A.3: Distribution of the number of tokens within each crawled file.

File type	Language tokens	*n*-gram value	Number of *n*-grams
Java	103	4	60,925
		5	208,567
		6	600,244
Haskell	421	4	134,679
		5	395,862
		6	943,824
C	127	5	291,626
C++	127	5	381,600
Header	127	5	337,367

Table A.6: Table showing the number of tokens and distinct *n*-grams per index.

n value	Similarity Percentile				*n*-grams generated
	50%	75%	90%	99%	
2	0.662	0.889	0.948	0.986	1,088
3	0.455	0.731	0.850	0.948	5,748
4	0.295	0.568	0.730	0.874	20,473
5	0.194	0.401	0.574	0.757	56,213
6	0.124	0.269	0.428	0.629	124,767
7	0.082	0.188	0.316	0.507	235,283
8	0.057	0.132	0.228	0.403	391,003
9	0.036	0.082	0.161	0.314	590,422

Table A.7: Table showing the similarities of all programs compared to an arbitrary program, from sample of 10,000 Java programs, for varying values of n and averaged over 10 programs.

Programs containing the same n-gram	n		
	4	5	6
Min	1	1	1
Max	1,289,364	1,247,581	1,245,156
Mean	3,613.8	1,292.2	540.8
Median	17	10	6

Table A.8: Table showing the number of distinct Java programs which contained the same n-gram.

Programs containing the same n-gram	n		
	4	5	6
Min	1	1	1
Max	89,265	65,767	56,325
Mean	138.6	59.7	29.5
Median	3	2	2

Table A.9: Table showing the number of distinct Haskell programs which contained the same n-gram.

Number of plagiarised files in source	Total source tokens	True rank of unmodified plagiarised sources						
		1	2	3	4	5	6	7
1	467	1	-	-	-	-	-	-
2	933	7,004	6,339	-	-	-	-	-
3	1,372	27,276	21,422	36,176	-	-	-	-
4	1,839	42905	28,481	52,763	50,534	-	-	-
5	2,201	55,130	37,693	65,707	63,695	>100k	-	-
6	2,571	78,020	60,072	91,794	93,475	>100k	>100k	-
7	2,902	86,070	65,394	>100k	>100k	>100k	>100k	>100k

Table A.10: Table showing the rank of multiple true candidates for sources containing multiple unmodified code sections, within the Java 4-gram index using no drop-off.

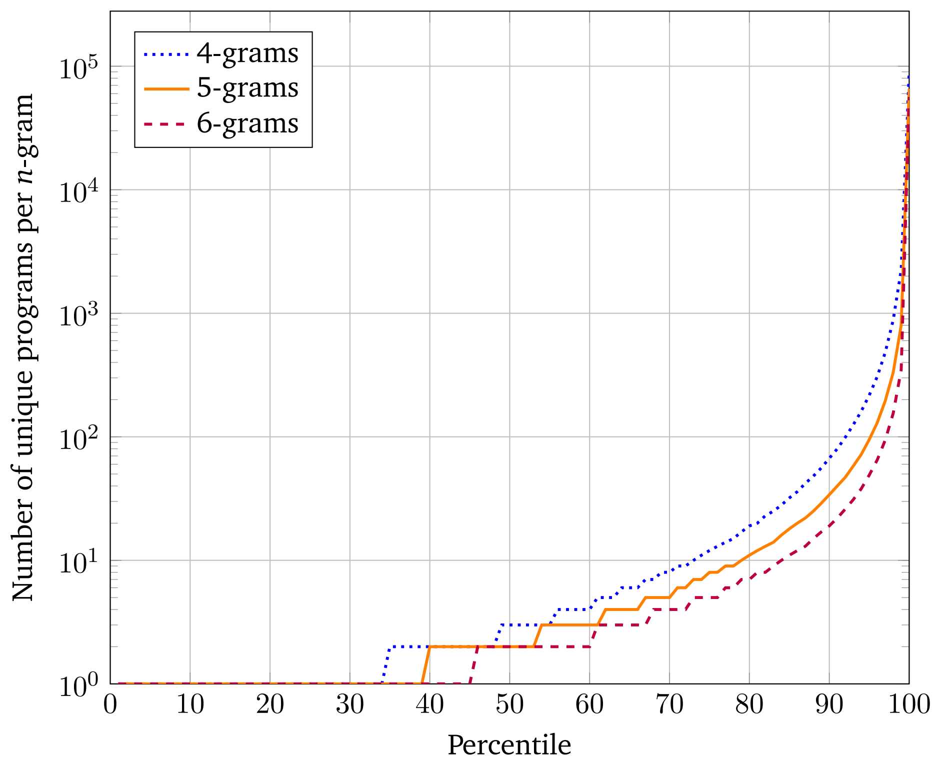

Figure A.4: Percentiles of the number of distinct Haskell programs containing the same *n*-gram.

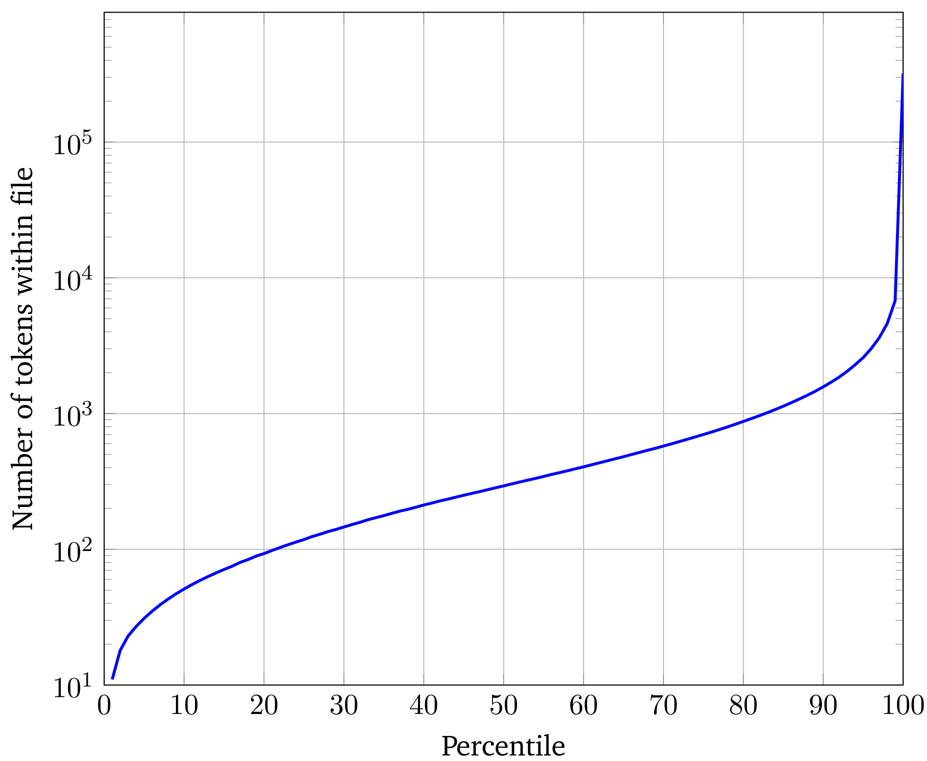

Figure A.5: Percentiles of the number of tokens within indexed Java files.

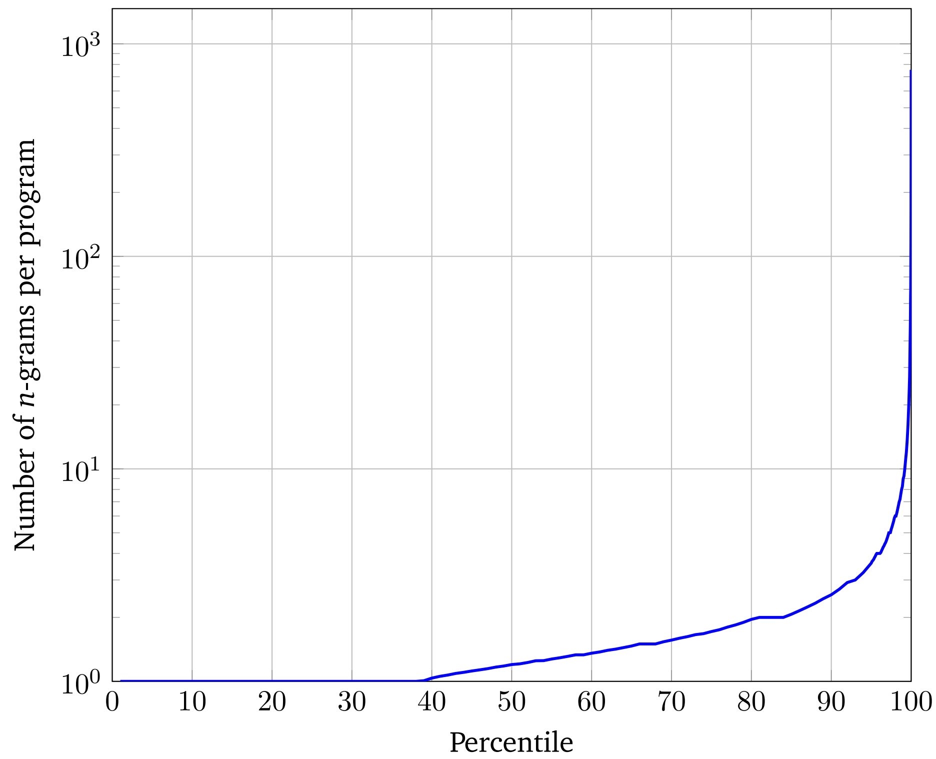

Figure A.6: Percentiles of the number of n-grams per program in which the n-gram occurs in, for the Java 5-gram index.

Number of plagiarised files in source	Total source tokens	True rank of unmodified plagiarised sources						
		1	2	3	4	5	6	7
1	467	1	-	-	-	-	-	-
2	933	1	2	-	-	-	-	-
3	1,372	1	2	3	-	-	-	-
4	1,839	2	1	12	3	-	-	-
5	2,201	190	127	420	212	2,477	-	-
6	2,571	1,038	915	1,817	1,360	6,941	6,349	-
7	2,902	2,023	1,812	3,174	2,594	10,141	9,234	13,988

Table A.11: Table showing the rank of multiple true candidates for sources containing multiple unmodified code sections, within the Java 6-gram index using square root drop-off.

Number of plagiarised files in source	Total source tokens	True rank of medium-level obfuscated plagiarised sources						
		1	2	3	4	5	6	7
1	467	1	-	-	-	-	-	-
2	933	1	2	-	-	-	-	-
3	1,372	2	4	1	-	-	-	-
4	1,839	169	17	10	191	-	-	-
5	2,201	1,131	108	146	916	30,576	-	-
6	2,571	3,008	1,027	1,201	4,509	56,574	10,828	-
7	2,902	4,999	2,271	2,348	7,272	67,332	14,911	21,561

Table A.12: Table showing the rank of multiple true candidates for sources containing multiple plagiarised code sections with medium plagiarism obfuscation performed, within the Java 6-gram index using square root drop-off.

Number of plagiarised files in source	Total source tokens	True rank of medium-level obfuscated plagiarised sources						
		1	2	3	4	5	6	7
1	467	1	-	-	-	-	-	-
2	933	1	2	-	-	-	-	-
3	1,372	2	4	1	-	-	-	-
4	1,839	79	11	9	88	-	-	-
5	2,201	617	64	83	515	24,285	-	-
6	2,571	1,892	577	679	3,098	47,017	7,957	-
7	2,902	3,494	1,457	1,473	5,357	56,887	11,495	17,143

Table A.13: Table showing the rank of multiple true candidates for sources containing multiple plagiarised code sections with medium plagiarism obfuscation performed, within the Java 6-gram index using $\frac{3}{5}$ power law drop-off.

Number of plagiarised files in source	Total source tokens	JPlag similarity assigned to unmodified plagiarised sources							Average candidate similarity
		1	2	3	4	5	6	7	
1	467	87.10	-	-	-	-	-	-	87.10
2	933	71.07	49.47	-	-	-	-	-	60.27
3	1,372	55.53	39.25	47.21	-	-	-	-	47.33
4	1,839	46.70	32.52	40.53	37.21	-	-	-	39.24
5	2,201	38.50	26.45	33.03	29.64	37.55	-	-	33.03
6	2,571	33.68	22.96	28.71	25.61	32.29	26.54	-	28.30
7	2,902	30.75	20.86	26.10	23.20	29.34	25.54	19.52	25.04

Table A.14: Table showing the similarity assigned to multiple true candidates by JPlag, after having been retrieved in the candidate set.

Number of plagiarised files in source	Total source tokens	JPlag similarity assigned to unmodified plagiarised sources							Average candidate similarity
		1	2	3	4	5	6	7	
1	473	60.10	-	-	-	-	-	-	60.10
2	1,039	47.66	43.44	-	-	-	-	-	45.55
3	1,541	46.31	37.95	39.49	-	-	-	-	41.25
4	2,121	38.90	31.67	32.34	31.59	-	-	-	33.60
5	2,627	33.57	27.21	27.59	27.84	26.73	-	-	28.59
6	3,052	31.51	23.92	30.17	24.64	23.53	27.98	-	26.96
7	3,447	29.91	22.68	28.54	23.41	22.31	26.12	14.29	23.89

Table A.15: Table showing the similarity assigned to multiple true candidates by JPlag, after having been retrieved in the candidate set. Candidates were subsections of the original files

Number of plagiarised files in source	Total source tokens	True rank of light-level obfuscated plagiarised sources						
		1	2	3	4	5	6	7
1	467	1	-	-	-	-	-	-
2	933	1	2	-	-	-	-	-
3	1,372	2	3	1	-	-	-	-
4	1,839	3	1	2	4	-	-	-
5	2,201	8	1	9	15	619	-	-
6	2,571	51	19	92	198	2,982	2,438	-
7	2,902	189	82	261	531	4,994	3,910	6,613

Table A.16: Table showing the rank of multiple true candidates for sources containing multiple plagiarised code sections with light plagiarism obfuscation performed, within the Java 6-gram index using linear drop-off.

Number of plagiarised files in source	Total source tokens	True rank of unmodified plagiarised C sources						
		1	2	3	4	5	6	7
1	378	1	-	-	-	-	-	-
2	806	5	3	-	-	-	-	-
3	1,268	11	6	1	-	-	-	-
4	1,709	36	7	1	4	-	-	-
5	2,184	3,165	277	47	155	16	-	-
6	2,556	21,051	5,821	2,012	3,899	1,272	24,094	-
7	3,017	75,700	37,176	20,326	29,718	15,817	81,301	20,695

Table A.17: Table showing the rank of multiple true candidates for sources containing multiple unmodified code sections, within the C 5-gram index using linear drop-off.

Number of plagiarised files in source	Total source tokens	True rank of unmodified plagiarised header sources						
		1	2	3	4	5	6	7
1	366	1	-	-	-	-	-	-
2	722	3	2	-	-	-	-	-
3	1,187	4	3	1	-	-	-	-
4	1,578	10	9	1	3	-	-	-
5	1,960	525	495	1	171	240	-	-
6	2,418	3,633	3,505	234	1,791	2,178	313	-
7	2,788	9,782	9,588	1,687	6,068	7,016	1,946	8,470

Table A.18: Table showing the rank of multiple true candidates for sources containing multiple unmodified code sections, within the header 5-gram index using linear drop-off.

Number of plagiarised files in source	Total source tokens	True rank of unmodified plagiarised C++ sources						
		1	2	3	4	5	6	7
1	299	1	-	-	-	-	-	-
2	713	3	1	-	-	-	-	-
3	998	3	1	6	-	-	-	-
4	1,377	5	1	8	4	-	-	-
5	1,656	16	1	45	4	69	-	-
6	2,006	783	2	1,679	9	2,212	45	-
7	2,390	6,904	32	10,791	252	12,686	948	202

Table A.19: Table showing the rank of multiple true candidates for sources containing multiple unmodified code sections, within the C++ 5-gram index using linear drop-off.

Number of plagiarised files in source	Total source tokens	True rank of unmodified plagiarised Haskell sources						
		1	2	3	4	5	6	7
1	254	1	-	-	-	-	-	-
2	530	2	1	-	-	-	-	-
3	719	2	1	3	-	-	-	-
4	878	2	1	3	6	-	-	-
5	1,059	2	1	4	99	14	-	-
6	1,420	4	2	246	1,393	490	1	-
7	1,669	44	16	551	3,013	1,602	1	73

Table A.20: Table showing the rank of multiple true candidates for sources containing multiple unmodified code sections, within the Haskell 6-gram index using linear drop-off.

Lines in indexed program	Tokens in indexed program	Plagiarised tokens in source file	True candidate rank	True candidate similarity	Candidate similarity threshold	Time taken (s)
1,092	4,639	174	1	0.582	0.112	1.64
2,647	13,575	381	153	0.125	0.082	1.14
2,827	15,266	438	138	0.118	0.051	0.97
1,533	20,017	356	1	0.474	0.162	1.92

Table A.21: Table showing the true ranks and similarities for plagiarised subsections of large programs, within the Haskell 5-gram index, using linear drop-off.

Original program language	Plagiarised program language	Tokens in plagiarised translated file	True candidate rank	True candidate similarity	Average candidate similarity	Candidate similarity threshold
Java	C++	576	1	0.650	0.465	0.420
		333	281	0.565	0.507	0.447
		480	356	0.273	0.249	0.214
C++	Java	794	1	0.671	0.412	0.366
		666	44	0.449	0.363	0.325
		395	2,535	0.488	0.553	0.504

Table A.22: Table showing the rank and similarity of true candidates for inter-lingual plagiarism detection, within the Java and C++ 5-gram indexes.