SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2007; **37**:151–175
Published online 12 September 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.750

Efficient plagiarism detection for large code repositories

Steven Burrows, S. M. M. Tahaghoghi[*,†]
and Justin Zobel

School of Computer Science and Information Technology,
RMIT University, Melbourne, Australia

SUMMARY

Unauthorized re-use of code by students is a widespread problem in academic institutions, and raises liability issues for industry. Manual plagiarism detection is time-consuming, and current effective plagiarism detection approaches cannot be easily scaled to very large code repositories. While there are practical text-based plagiarism detection systems capable of working with large collections, this is not the case for code-based plagiarism detection. In this paper, we propose techniques for detecting plagiarism in program code using text similarity measures and local alignment. Through detailed empirical evaluation on small and large collections of programs, we show that our approach is highly scalable while maintaining similar levels of effectiveness to that of the popular JPlag and MOSS systems. Copyright © 2006 John Wiley & Sons, Ltd.

Received 6 February 2005; Revised 27 February 2006; Accepted 24 March 2006

KEY WORDS: plagiarism detection; program code similarity; indexing; local alignment

1. INTRODUCTION

Digital content is easy to copy and manipulate, and preventing unsanctioned re-use is a major problem for academic institutions hoping to maintaining standards, and for businesses concerned about the legal implications of having unauthorized content in their products. For example, an Australian study of 287 Monash University and Swinburne University students [1] found that more than 70% of students at each institution admitted to cheating. Effective and efficient detection mechanisms are needed to manage this problem.

[*] Correspondence to: S. M. M. Tahaghoghi, School of Computer Science and Information Technology, RMIT University, GPO Box 2476V, Melbourne 3001, Australia.
[†] E-mail: saied@cs.rmit.edu.au

Contract/grant sponsor: RMIT School of Computer Science and Information Technology
Contract/grant sponsor: Australian Research Council

However, detecting electronic plagiarism has become increasingly difficult due to the large volume of available data, and easy access and sharing mechanisms. For example, the 'School Sucks' Web site[‡] claims to have 50 000 term papers, book reports, dissertations, and college essays available online. For computing subjects, there is an abundance of Web sites that provide source code for programs relevant to assessable tasks. A query to the Google[§] search engine for the system call `printf`, common to several popular programming languages, returned over four million query results[¶]. In academic environments, the problem is compounded by large cohorts: class sizes of 300 students or more are not uncommon.

The Merriam-Webster Online Dictionary [2] defines four definitions for the word 'plagiarize': 'To steal and pass off (the ideas or words of another) as one's own . . . To use (another's production) without crediting the source. . . . To commit literary theft. . . . To present as new and original an idea or product derived from an existing source'. We define code plagiarism as the unauthorized reuse of program structure and programming language syntax.

In this work, we discuss code plagiarism detection in the context of programming assignments in university courses. For large class sizes, manual detection of plagiarism is a slow and laborious process and therefore impractical. In a collection of 300 programs, there are 44 850 unique program pairs. Many subjects have multiple staff assessing student assignments, and it is rare for a single person to see all submissions. Consequently, many schools have adopted automated plagiarism-detection systems.

Moreover, the nature of the syllabus in some subjects makes it difficult to vary assignment specifications between offerings from year to year. A scalable plagiarism-detection method would allow staff to check submissions against those for previous subject offerings; we refer to this as *historical* plagiarism detection. It would also allow search across subjects that have substantially common content, but are perhaps separated due to the cohort of students enrolled. Arguably more importantly, a scalable search would allow the use of large collections of programs obtained by crawling the Web.

Besides being efficient, any practical system must of course also be highly accurate: university staff lack the time and resources to examine large numbers of false positives. Hence, we require an efficient, effective, and scalable solution that enables plagiarism detection for historical, multi-subject, and Web-based repositories.

While systems such as SIM, JPlag, and MOSS are effective at detecting plagiarism, they are designed to search for similar documents only within a single set of source code files; for example, those submitted for a single assessment task. JPlag [3] and SIM [4] search for matches in an exhaustive pairwise fashion, which is unscalable: JPlag actually enforces an upload limit of approximately 9.5 MB [5]. MOSS does not impose any explicit volume limits on uploads[‖] and uses an index to efficiently filter out unlikely matches [6]; however, it currently performs all processing in main memory.

In this paper, we present a novel approach to plagiarism detection for program source code using techniques adapted from text and genomic information retrieval. Our approach uses an index of

[‡] http://www.schoolsucks.com

[§] http://google.com

[¶] As of 25 February 2006.

[‖] From personal communication with MOSS author A. Aiken, 14 February 2006.

DOI: 10.1002/spe

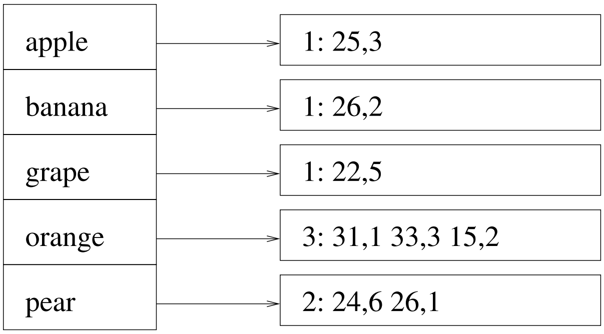


Figure 1. A simple inverted index with a lexicon of five elements. Each entry shows the number of documents that contain the element, the individual document identifiers, and the number of occurrences of the term in each document.

programs that is accessed by a search engine for each program query, and the results generated by this process are refined through multiple local alignment. We present results on one small and one large collection of C program source code files that show that our approach is effective, efficient, and scalable.

The remainder of this paper is organized as follows. In Section 2, we discuss background knowledge in search engines and local alignment. In Section 3, we review existing approaches to plagiarism detection and note their strengths and limitations. In Section 4, we describe our approach. In Section 5, we describe our experimental data and evaluation techniques, and present results in Section 6. We conclude in Section 7 with a summary of our findings and an outline of potential future work.

2. BACKGROUND

The plagiarism-detection techniques we introduce in this paper use search algorithms and local alignment to identify likely matches. In this section, we briefly describe the major components and functions of a search engine and follow with a discussion of *local alignment*, a technique commonly used in genomic information retrieval for finding matching sub-sequences. Witten *et al.* [7] describe search engine design in detail.

2.1. Search engines

The purpose of a search engine is to summarize large volumes of data for efficient retrieval, typically through the use of an *inverted index*. Figure 1 shows an inverted index. This is comprised of two main components: a *lexicon* and a set of *inverted lists*. The lexicon stores all distinct words in the collection, in our example these are names of fruits. The inverted lists hold selected statistics about each lexicon item including the *term frequency*, which indicates how many times a particular term occurs in the collection, and integer pairs that indicate which documents contain the term, and how frequently it occurs in each document. These are known as the *document identifier* and the *within-document frequency*.

Consider two examples. The term 'banana' has a document frequency of 1; this term occurs in one document. The inverted list indicates that it appears twice in `document 26`. Similarly, the term 'pear' appears in two documents; six times in `document 24` and once in `document 26`. This illustrates why a search engine is highly scalable for large collections of data: each term need only be stored once in the lexicon, the inverted lists are comprised of integers that can be processed rapidly, and the lists are highly compressible [7,8].

At query time, the system retrieves the inverted lists for the query terms and constructs a list of collection documents ranked by decreasing estimated relevance according to a *similarity measure*. Such a measure must consider several factors. First, documents that have more instances of a query term (have a higher *term frequency*) are likely to be more relevant. Second, lengthy documents are by nature likely to have a higher term frequency than shorter documents, so we should provide an advantage to shorter documents based on *inverse document length* [7]. Third, some terms are common, while some are rare. Since information content is proportional to rarity [9], it is also useful to weight terms based upon their rarity or their *inverse document frequency*.

A well-known similarity measure that employs term frequency, inverse document length, and inverse document frequency is the *cosine measure* [7]

$$cosine(Q, D_d) = \frac{1}{W_D W_Q} \cdot \sum_{t \in Q \cap D_d} w_{q,t} . w_{d,t}$$

$$W_D = \sqrt{\sum_{t=1}^{n} w_{d,t}^2}, \quad W_Q = \sqrt{\sum_{t=1}^{n} w_{q,t}^2}$$

$$w_{d,t} = 1 + \ln f_{d,t}, \quad w_{q,t} = \ln\left(1 + \frac{N}{f_t}\right)$$

where Q is the query, D_d is the document, t is a term in the query that also appears in document D_d, and N is the number of documents in the collection. W_D is the document weight, W_Q is the query weight, f_t is the collection frequency of the term, $f_{d,t}$ is the within-document frequency of the term, $w_{d,t}$ is the document-term weight, and $w_{q,t}$ is the query-term weight.

This measure, which has many variants, is based on the assumption that a document is a point in n-dimensional space, and that two documents are similar if the angle between their origin vectors is small. The variant shown is one commonly used for text information retrieval where users typically enter each query term only once. Hence, the query-term weight ($w_{q,t}$) does not explicitly incorporate the query-term frequency ($f_{q,t}$), which is assumed to be constant at 1. However, the query-term frequency is important in the context of our work, where each query is a list of program source code symbols. Hence, we calculate the query-term weights as:

$$w_{q,t} = (1 + \ln f_{q,t}). \ln\left(1 + \frac{N}{f_t}\right)$$

An alternative approach to similarity measurement is to calculate the likelihood that a document is relevant to the information need expressed in the query, as in the Okapi BM25 similarity measure.

This measure employs factors for term frequency, inverse document length, and inverse document frequency [10]

$$Okapi(Q, D_d) = \sum_{t \in Q} w_t . \frac{(k_1 + 1) f_{d,t}}{K + f_{d,t}} . \frac{(k_3 + 1) f_{q,t}}{k_3 + f_{q,t}}$$

$$w_t = \ln\left(\frac{N - f_t + 0.5}{f_t + 0.5}\right), \quad K = k_1 . \left((1 - b) + \frac{b . W_d}{W_D}\right)$$

where Q is the query, D_d is the document, t is a term in the query that also appears in document D_d, and N is the number of documents in the collection. W_d is the document length, W_D is the average document length, f_t is the collection frequency of the term, $f_{d,t}$ is the within-document frequency of the term, and $f_{q,t}$ is the within-query frequency of the term. The recommended values for the parameters are as follows: $k_1 = 1.2$, $k_3 = 1\,000$, and $b = 0.75$.

While repetition of a query term signifies increased similarity in text information retrieval the same is not true in plagiarism detection, where identical use is more significant. A key difference between the cosine and BM25 similarity measures is the way they weight frequently occurring terms. In the cosine measure, the weight accorded to a term can increase without bound. This is not the case with BM25, where the weight accorded to a term asymptotically approaches a fixed upper limit.

2.2. Local alignment

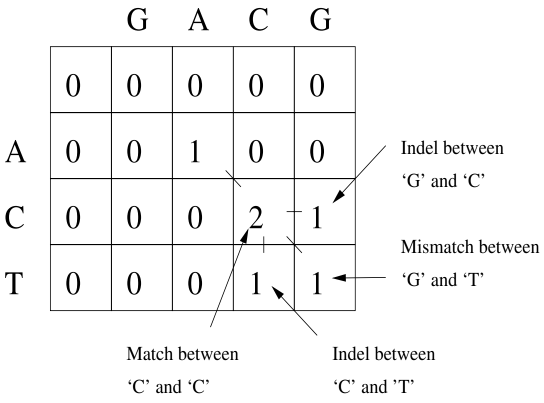
The *local alignment* [11] procedure is an approximate string matching technique commonly used in genomic information retrieval to identify the optimal alignment—the alignment with the largest possible overlap—between two DNA sequence pairs. Insertions, deletions, matches, and mismatches are all considered when computing an optimal alignment.

Sequences with highly similar regions of DNA could have similar *homology*, that is, they may share a similar evolutionary ancestry. The most widely used tool used to perform DNA local alignment is the Basic Local Alignment Search Tool (BLAST) [12]. In this paper, we investigate the application of local alignment to plagiarism detection.

Consider the local alignment of the two short sequences ‘GACG’ and ‘ACT’ presented in Figure 2. Local alignment can be calculated on two sequences s and t, of lengths l_1 and l_2 using a matrix of size $(l_1 + 1) \times (l_2 + 1)$ [11]. The local alignment matrix allows us to calculate an alignment score for every character position of two sequences, and in this way identify highly similar regions. For our example sequences of length three and four, we obtain a matrix of twenty cells. The first row and the first column of the matrix are initialised with zeros. We then compute the score for each cell, considering the maximum possible score resulting from a match, mismatch, insertion, or deletion between each possible pair of characters of the two sequences. Following a selected scoring scheme, we generally award a positive score (say 1) for any pair of characters from the two sequences that match, and apply a penalty (say -1) for each mismatch.

In our example, the character ‘C’ from ‘GACG’ and the character ‘C’ from ‘ACT’ match. This merits a one-point gain over the value in the matrix square immediately to the upper-left. Since that score is already one (due to the matching ‘A’ of each sequence), we obtain a score of two for this cell. Similarly, we can penalise any pair of characters that *mismatch*. We have a mismatch between the last character ‘G’ from ‘GACG’ and the character ‘T’ from ‘ACT’, and a one-point penalty is applied.

DOI: 10.1002/spe



		G	A	C	G
	0	0	0	0	0
A	0	0	1	0	0
C	0	0	0	2	1
T	0	0	0	1	1

Figure 2. A local alignment matrix for the sequences 'GACG' and 'ACT'. The arrows indicate a match, a mismatch, and two indels.

We may be able to find a better alignment of two sequences through character insertions or deletions, that is, *indels*. These attract a penalty, as they also represent a difference between the sequences. We have an indel between the last character 'G' from 'GACG' and the character 'C' from 'ACT'. This implies that we have either inserted a blank before the character 'T' of the sequence 'ACT', or deleted it. In our scoring scheme, an indel is penalised by one; a cell arrived at through an indel is assigned a score one less than the previous cell (to the left or above, depending on the path). In our example, we have reduced the score from the left.

In sequences with a large number of mismatches and indels, it is possible that a negative score will be generated. Given that we are most interested in the similarity of matching regions in local alignment we enforce a minimum score of zero, ensuring that interesting regions of similarity are not hidden by preceding regions of high dissimilarity.

The local alignment scoring metrics can be summarized as follows:

$$[i, j] = \max \begin{cases} [i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } s_i = t_j \\ [i-1, j-1] - 1 & \text{if } i, j > 0 \text{ and } s_i \neq t_j \\ [i, j-1] - 1 & \text{if } i, j > 0 \\ [i-1, j] - 1 & \text{if } i, j > 0 \\ 0 & \text{for all } i, j \end{cases}$$

The score increase in the first line of the formula corresponds to a match, while the score decreases in lines 2–4 represent the penalties for mismatches and indels respectively. The last line ensures a lower bound of zero for the alignment score. The *max* operator ensures that we always follow the optimal character alignment.

To find the optimal alignment of the two sequences, we traverse the matrix to locate the highest score in any cell. In Figure 2, the highest score is two. The *traceback path* (indicated by dashes) is followed until a zero is reached. This traceback path denotes the highest scoring preceding region to

DOI: 10.1002/spe

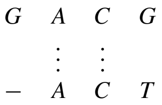
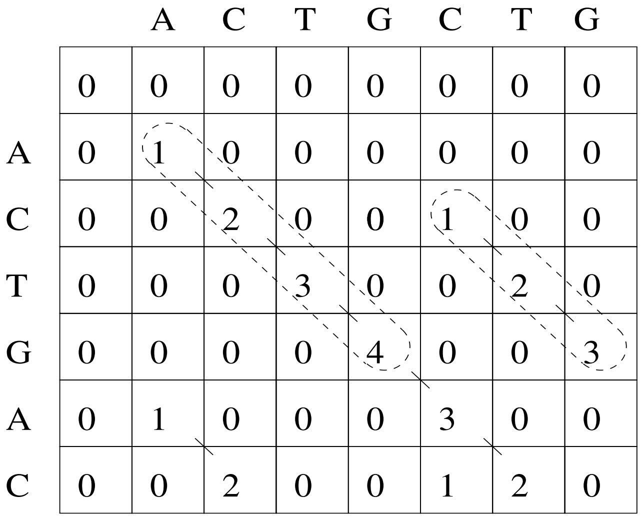


Figure 3. An optimal alignment ‘AC’ as found in the sequences ‘GACG’ and ‘ACT’. This alignment has been computed from the local alignment matrix in Figure 2.



		A	C	T	G	C	T	G
	0	0	0	0	0	0	0	0
A	0	1	0	0	0	0	0	0
C	0	0	2	0	0	1	0	0
T	0	0	0	3	0	0	2	0
G	0	0	0	0	4	0	0	3
A	0	1	0	0	0	3	0	0
C	0	0	2	0	0	1	2	0

Figure 4. Multiple local alignment between the sequences ‘ACTGAC’ and ‘ACTGCTG’. In this example, we use {match = 1, mismatch = −1}. Indels are ignored. We also employ a minimum match length of 3 to remove trivial matches. The alignment of these two sequences generates a score of 7.

the left, above, or to the upper-left of the current cell. The traceback path in our example represents the alignment ‘AC’ as shown in Figure 3.

When matching genomic sequences, it is not uncommon to encounter more than one optimal alignment or several regions of high similarity. The same occurs when comparing plagiarized program source code, where multiple segments of similarity may result from attempts at disguise. To process these, we apply the *multiple local alignment* technique of Morgenstern *et al.* [13] as used in the *DIALIGN* genomic information retrieval system. To ensure that the partial segments do not overlap, we ignore indels. We identify local regions of similarity by summing the optimal local alignment on each diagonal to produce a similarity score. To avoid noise from trivial matches, we impose a minimum threshold for the match length. In Figure 4, two regions of similarity above a threshold length of 3 produce an overall similarity score of 7.

DOI: 10.1002/spe

3. RELATED WORK

Existing plagiarism detection methods fall into two main categories: text-based and code-based. In this section, we explain why text-based approaches are not suitable for code-based plagiarism detection, and discuss attribute-oriented and structure-oriented approaches to this problem. We also discuss the SIM, JPlag, MOSS, and PlagiIndex code plagiarism detection systems in detail.

3.1. Text-based systems

There are two families of methods for text-based plagiarism detection: *ranking* [14] and *fingerprinting* [15]. Ranking has a text information retrieval heritage, with indexed collection items being ranked by estimated similarity to a query. Hoad and Zobel show that text similarity measures such as the cosine measure are best suited for *ad hoc* querying and do not adapt well to plagiarism detection, where plagiarized documents are likely to contain a similar number of occurrences of terms. They propose alternative *identity measures* to estimate similarity for this application; we have experimented with the variant shown to be the most effective (Variant Five):

$$\frac{1}{1+\ln(1+|f_d - f_q|)} \cdot \sum_{t \in q \cap d} \frac{N/f_t}{1+|f_{d,t} - f_{q,t}|}$$

where N is the number of documents in collection, f_t is the number of documents containing term t, f_d is the number of terms in document d, $f_{d,t}$ is the number of occurrences of term t in document d, f_q is the number of terms in query q, and $f_{q,t}$ is the number of occurrences of term t in query q.

In fingerprinting, documents are summarized as a compact collection of integers representing key aspects of a document [14]. These *fingerprints* are indexed, and comparable fingerprints are generated from queries to be searched against this index.

One of the best-known text-based plagiarism detection systems available is Turnitin**. As of February 2006, Turnitin claims to index more than 4.5 billion documents. This system does not explicitly cater for program source code, and, when provided source code to check, simply compares the English words. This relies on words such as function names, comments, and output statements being similar, and ignores coding syntax. This is a severe drawback, since it is the coding syntax that should be used for judging program similarity.

3.2. Attribute-oriented code-based systems

Attribute-oriented plagiarism detection systems evaluate key properties of program code. Donaldson *et al.* [16] describe a system that measures four attributes:

- the number of unique operators;
- the number of unique operands;
- the total number of occurrences of operators;
- the total number of occurrences of operands.

**http://www.turnitin.com

DOI: 10.1002/spe

The similarity of two programs is approximated by the differences between these attributes. Unfortunately, attribute-oriented systems are highly limited. Verco and Wise [17] report that attribute-oriented systems are most effective where very close copies are involved. However, a common method for disguising plagiarism is to add or remove redundant code. Since attribute-oriented systems consider all lines of code, the attribute scores of the two programs may be very different. Moreover, it is difficult to detect copied segments of code in larger programs.

3.3. Structure-oriented code-based systems

Structure-oriented systems [3,4,18] create lossy representations of programs and compare these to detect similarity. Easily modified elements, such as comments, whitespace, and variable names, are ignored.

Structure-oriented systems are relatively robust to the addition or removal of redundant statements because they perform local comparisons. To avoid detection, a student would need to significantly modify all parts of the program to reduce the longest matching program segments to a value below the minimum match length threshold of the plagiarism detection system. It can be argued that this amount of work is comparable to that needed to complete the program legitimately, and thus may deter plagiarism. Three of the most cited structure-oriented plagiarism detection systems are SIM [4], JPlag [3], and MOSS [18].

SIM [4] analyses programs written in Java, C, Pascal, Modula-2, Lisp, Miranda, and can also process plain text files. It applies string alignment techniques developed to detect similarity in DNA strings as discussed in Section 2.2. However, alignment is computationally expensive, and exhaustive application to all program pairs as in SIM is not scalable for large code repositories. SIM also performs all processing in main memory. While this plagiarism detection system is no longer actively supported, its source code is publicly available[††].

JPlag [3] is provided as a publicly accessible service[‡‡] that compares programs pair-by-pair to produce a report in HTML format listing highly similar files. It supports the Java, C#, C, C++, and Scheme languages, and can also compare plain text files.

The system first converts programs to a token representation and then uses the *greedy string tiling* algorithm [19] to find a set of maximal length substrings common to two programs. To avoid repetitively matching subsets of previously matched substrings, characters are marked off as each match is performed. Trivial matches shorter than a threshold are ignored. A similarity score is computed between the two strings that reflects the percentage of characters in common substrings.

For example, comparing the strings ‘ABCDEFGHIJ’ (string A) and ‘ABDEFLEFGH’ (string B) with a minimum match length of three returns two matches—‘DEF’ and ‘EFGH’—with sufficient length. After marking these seven characters in string B, we obtain ‘AB***L****’. Finally, the JPlag similarity measure is used to provide a similarity score between the two strings [3]:

$$sim(A, B) = \frac{2 \sum_{match(a,b,length)\in tiles} length}{|A| + |B|}$$

[††] http://www.cs.vu.nl/∼dick/sim.html
[‡‡] http://www.jplag.de

Prechelt *et al.* [3] note that the worst case complexity for this algorithm is $O((|A|+|B|)^3)$. In the best case, where the two strings have no characters in common, the complexity is reduced to $O((|A|+|B|)^2)$. Elements of the Karp–Rabin pattern matching algorithm [19,20] are used to further improve efficiency. The greedy string tiling algorithm is applied in an exhaustive fashion to all program pairs in JPlag [3].

MOSS [18] can process files containing Java, C, C++, Pascal, or Ada programs, or plain text. It is accessible as an online service, and allows the user to submit boilerplate code alongside the assignment submissions to avoid a large number of false positives. Internally, MOSS converts program source code into tokens, and uses the *robust winnowing* algorithm to select a subset of the token hashes as the document fingerprints [6].

When comparing a batch of files, MOSS creates an inverted index to map document fingerprints to documents and their positions within each document. Next, each program file is used as a query against this index, returning a list of documents in the collection having fingerprints in common with the query. This results in a score (number of matching fingerprints) for each document pair in the batch; these are sorted and the highest-scoring matches are reported to the user. Schleimer *et al.* [6] describe the robust winnowing algorithm, but details of the MOSS internal algorithms remain confidential.

3.4. Other work

The PlagiIndex [5] scheme supports plagiarism detection for large code repositories by using an inverted index. It ranks programs using the PlagiRank similarity measure, which is 'based on the perception that similar programs should contain similar number of occurrences of symbols (tokens)'. This makes sense because the nature of the cosine and BM25 similarity measures is to reward documents that have a high number of query terms rather than a similar number of terms between the query and the document. The PlagiRank similarity measure is defined as

$$PlagiRank(Q, D_d) = \frac{1}{W_D W_Q} . \sum_{t \in Q \cap D_d} \left(\ln\left(\max\left(\frac{f_{q,t}}{f_{d,t}}, \frac{f_{d,t}}{f_{q,t}} \right) \right) + 1 \right) . f_{q,t}$$

As discussed in Section 2.1, indexes are highly scalable for large collection sizes, and Chawla observed this for code indexing. He also reported that the PlagiRank metric was highly effective, and made preliminary use of local alignment. However, the reported experiments are limited in both scope and significance, and details of some collections and experiments are not documented.

Manber [21] introduces the *sif* program that uses fingerprints of substrings to find files that share a significant number of substrings. The author lists applications including file management, program reuse, and file synchronisation, and notes that while *sif* could be used for plagiarism detection, it would not be difficult to evade detection. Baker [22] discusses the program *dup*; that can detect exact or parameterized matches in a large software system to aid re-engineering and maintenance. Baker and Manber [23] describe how Java bytecode can be disassembled and processed with *siff* [sic], *dup*, and the Unix *diff* tool for applications including plagiarism detection.

Broder *et al.* define the mathematical notions of 'containment' and 'resemblance' between two documents, and use shingles to determine clusters of similarity within a large collection of documents retrieved from the Web [24,25]. They note that their approach has application to detection of plagiarism, but that it has limited security.

Irving [26] describes using local alignment for plagiarism detection in text, and considers its application to program source code. While effective, the reported approach relies on pairwise matching of documents, and the author notes that efficiency would need to be improved before it could be used for a large-scale collection.

Mozgovoy *et al.* [27] underline the importance of scalability, and employ a suffix array to index program content. However, they do not report any experimental results for their approach. We continue by describing our solution to the scalability problem.

4. SCALABLE PLAGIARISM DETECTION

We aim to improve the scalability of the code plagiarism detection process through the use of the inverted index, and use local alignment to maintain overall effectiveness comparable to the JPlag system. Our approach to plagiarism detection is to find high-ranking matches with an index, then post-process these using local alignment. The main steps are as follows:

1. tokenize all program source files into a format suitable for indexing;
2. construct an index of all files;
3. query the index using each program file in turn to identify candidate matches;
4. use local alignment to examine highly ranked files to refine the results.

We now describe our approach in detail.

4.1. Tokenization

In Section 2.1, we illustrated how the lexicon in an inverted index stores the distinct words in a collection. Programming code has little resemblance to written text: it contains specialized programming characters—such as punctuation characters and braces—that text search engines ignore (or *stop*) during indexing. Programming code is similarly unsuitable for direct use in local alignment. Therefore, we must represent code in a more suitable format for indexing and alignment.

It is desirable to have a token representing each significant characteristic of code. For example, branching and looping constructs are significant, and should be retained. However, while comments may provide useful forensic evidence, they are difficult to use in an automated tool, and are hence best ignored. We have adapted the tokenizing functionality of the SIM plagiarism detection system. Figure 5 shows an example C program—that we refer to as Sample Program 1—its constructs, and the corresponding token stream.

This stream is now in a suitable format for performing local alignment. However, it is not practical to index a contiguous token stream, so we generate an *n-gram* representation of the token stream, and index each n-gram separately. This provides an additional benefit besides easy indexing. The n-grams have been shown to support matching of partial query terms [28] for English text retrieval, and can be used for effective structural plagiarism detection in code [6]. Any change to the source code will only affect a few neighbouring n-grams, and a modified version of a copied program will have a substantial proportion of n-grams unchanged.

Consider the simple example of the words ‘cold’ and ‘colder’; these can be decomposed into the 4-gram sets {cold} and {cold, olde, lder}, respectively, by moving a *sliding window* of size $N = 4$

DOI: 10.1002/spe

```
1. #include <stdio.h>
2. int main(void) {
3.     int var;
4.     for (var=0; var<5; var++) {
5.         printf("%d\n", var);
6.     }
7.     return 0;
8. }
```

No.	Construct	Token
1	int	S
2	main	N
3	(	A
4	)	B
5	{	j
6	int	S
7	ALPHANUM	N
8	for	R
9	(	A
10	ALPHANUM	N
11	=	K
12	ALPHANUM	N
13	ALPHANUM	N
14	<	J
15	ALPHANUM	N
16	ALPHANUM	N
17	+	D
18	+	D
19	)	B
20	{	j
21	ALPHANUM	N
22	(	A
23	STRING	5
24	,	E
25	ALPHANUM	N
26	)	B
27	}	l
28	return	g
29	ALPHANUM	N
30	}	l

Figure 5. Sample Program 1—a small C program—its constructs, and its token representation. Here, 'ALPHANUM' represents a function or variable name, or a variable value, while 'STRING' is a sequence of characters enclosed by quotation marks.

across the string from left to right. We can similarly apply the sliding window technique to the token stream of Figure 5 and convert this token stream into a sequence of n-grams. For a token stream of t tokens, we require $t - n + 1$ n-grams, where n is the length of the n-gram. Chawla [5] identified the most appropriate size of n-grams for his PlagiIndex system to be four, and we use this size for our approach.

Figure 6 shows Sample Program 1, its token stream, and the sequence of 4-grams extracted from this token stream. The figure also shows Sample Program 2, a C program that is identical to the first, with the addition of a line of code at line 7. The token stream and 4-gram representation are included alongside.

4.2. Index construction

The second step is to create an inverted index of the n-grams. We use the Zettair* text search engine for this purpose. The n-grams of both programs in our running example are shown indexed in Figure 7. As discussed in Section 2.1 each element of our lexicon is stored only once, which results in a large space saving. This allows us to limit the search to only the relevant parts of the inverted index, rather than following the highly inefficient approach of exhaustively comparing all elements of all programs. Note also that we only need to build the index once, and can reuse it for any combination of queries on the collection. This allows practical indexing and search of large collections of submissions accumulated over time, or source code crawled from the Web. For applications where the collection varies over time, we must rebuild the index or use incremental update approaches [29–31]. We do not explore updates in this work.

*http://www.seg.rmit.edu.au/zettair/

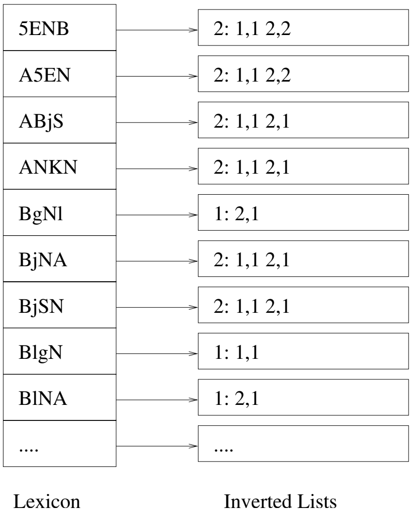
```
#include <stdio.h>
int main(void) {
   int var;
   for (var=0; var<5; var++) {
      printf("%d\n", var);
   }
   return 0;
}
```

```
Token Stream:
SNABjSNRANKNNJNNDDBjNA5ENBlgNl

4-grams:
SNAB NABj ABjS BjSN jSNR SNRA NRAN
RANK ANKN NKNN KNNJ NNJN NJNN JNND
NNDD NDDB DDBj DBjN BjNA jNA5 NA5E
A5EN 5ENB ENBl NBlg BlgN lgNl
```

```
#include <stdio.h>
int main(void) {
   int var;
   for (var=0; var<5; var++) {
      printf("%d\n", var);
   }
   printf("Value: %d\n", var);
   return 0;
}
```

```
Token Stream:
SNABjSNRANKNNJNNDDBjNA5ENBlNA5ENBgNl

4-grams:
SNAB NABj ABjS BjSN jSNR SNRA NRAN
RANK ANKN NKNN KNNJ NNJN NJNN JNND
NNDD NDDB DDBj DBjN BjNA jNA5 NA5E
A5EN 5ENB ENBl NBlN BlNA lNA5 NA5E
A5EN 5ENB ENBg NBgN BgNl
```

Figure 6. (Top) Sample Program 1, its token stream comprising the 30 tokens of Figure 5, and the resulting 27 4-grams. (Bottom) Sample Program 2, its token stream, and its 4-gram representation. Here, the 36 tokens produce 33 4-grams.



Lexicon	Inverted Lists
5ENB	2: 1,1 2,2
A5EN	2: 1,1 2,2
ABjS	2: 1,1 2,1
ANKN	2: 1,1 2,1
BgNl	1: 2,1
BjNA	2: 1,1 2,1
BjSN	2: 1,1 2,1
BlgN	1: 1,1
BlNA	1: 2,1
....	....

Figure 7. The *n*-gram index representation of the first nine *n*-grams of the sample programs of Figure 6.

Table I. Results of the program `0020.c` compared to an index of 296 programs. We find that program `0103.c` is highly similar to program `0020.c` indicating that these programs could be plagiarized. The remaining programs do not have significant regions of high similarity, and are unlikely to be plagiarized.

Rank	Query file	Index file	Raw score	Similarity score
1	`0020.c`	`0020.c`	369	100%
2	`0020.c`	`0103.c`	345	93%
3	`0020.c`	`0092.c`	189	51%
4	`0020.c`	`0151.c`	185	50%
5	`0020.c`	`0267.c`	168	45%
6	`0020.c`	`0150.c`	165	45%
7	`0020.c`	`0137.c`	159	43%
8	`0020.c`	`0139.c`	154	42%
9	`0020.c`	`0269.c`	129	35%
10	`0020.c`	`0241.c`	127	34%

4.3. Querying

To perform plagiarism detection, we use an *n*-gram representation of each program as a query against the index. The system returns a list of programs matching the query, sorted by calculated similarity. If the query program is itself indexed, we expect it to appear as the highest-ranking entry in the list. We assign a similarity score of 100% to this top match. The other programs are given a similarity score relative to this.

Our test data and anecdotal evidence indicates that students generally do not work in very large groups, and so we believe that it is sufficient to list only the top ten results for each query. For example, Table I presents the top ten results of one *n*-gram program file (`0020.c`) compared against an index of 296 programs. We can see that the file evaluated against itself generates the highest score, and we assign this the relative score of 100.00%. This score is ignored, but we use it to generate a relative similarity score for all other results. We see that program `0103.c` is very similar to program `0020.c` with a score of 93.34%. This is an early indication that these two programs could be plagiarized. The other programs have much lower scores, indicating that these are unlikely to be plagiarized. We repeat this process for every query file.

As can be seen from Table II, our program code queries are represented by hundreds or thousands of *n*-grams. In contrast, the typical query to a text search engine is very short: one study found that 91% of logged queries had five or fewer words [32].

The similarity measure we use is asymmetric; the similarity of a program A to a program B is not necessarily the same as the similarity of program B to program A. This asymmetry is a consequence

Table II. The number of lines of code and the number of tokens in the 296 programs of Collection A and the 61 540 programs of Collection B.

	Collection A		Collection B	
	Lines of code	Tokens	Lines of code	Tokens
Min	112	312	0	0
Max	1 095	3 942	2 628	74 467
Mean	446	1 461	169	563
Median	418	1 353	102	273
Total	132 146	432 561	10 384 257	34 630 414

of the Zettair optimization as a query-to-document search engine, not a document-to-document search engine[†]. If both similarity values are in the top 10 results, we use the higher of the two scores for final ranking. We applied this technique for the BM25, Cosine, PlagiRank, and Identity similarity measures.

4.4. Local alignment

The short n-grams used to index program source code provide high efficiency, but do not afford high selectivity. To address this, we perform additional processing using the local alignment approximate string matching technique. Since local alignment is computationally expensive, we apply it only to program pairs with a similarity score of more than 30%. In our sample data, this represents approximately two percent of all possible program pairs.

In Section 2.2, we discussed the basic local alignment algorithm with the default parameters of $\{\text{match} = 1, \text{mismatch} = -1, \text{indel} = -1\}$. In our experiments, we explore two aspects of the basic local alignment algorithm. First, we examine the effect of altering the default local alignment scoring parameters for match, mismatch and indel. Second, we examine the effect of computing multiple local alignments in the local alignment matrix as discussed in Section 2.2.

5. EVALUATION

In this section, we describe our data sets, the approach we use to employ the ground truth, and the evaluation measures we employ to compare competing approaches.

[†] From personal communication with Zettair contributor Nicholas M. Lester, 7 September 2004.

DOI: 10.1002/spe

We use two data collections for our experiments. The first is a collection of 296 student programs, Collection A, written in the C programming language for an assignment of an introductory programming subject. This assignment required students to implement a 'Student Adviser System' that would allow student and adviser details to be manipulated via a simple command-line interface. Students were supplied with a some boilerplate code, mainly comprising variable definitions and header comments; this formed a small part of the required final program.

The second collection, Collection B, consists of 61 540 C source code files archived from several years of student assignment submissions. We incorporated Collection A into this repository.

We use this collection to demonstrate the efficiency and effectiveness of our solution for a large code repository. Table II presents statistics about the two collections.

There are 296 program files in Collection A, forming 43 660 distinct program pairs. It is impractical to inspect each pair manually to identify cases of plagiarism and thus establish the ground truth. To address this issue, we use an automatic tool as a first-pass filter. While not perfect, this provides a reasonable baseline for evaluation of our results. We used the popular JPlag plagiarism detection system as the initial filter to identify all program pairs with a similarity score of at least 30%:30 program pairs met this criterion.

Each pair was then examined by the first author, who had written the assignment specification for the programming subject. Having an expert user evaluate the programs for plagiarism allowed good discrimination of plagiarism from innocent similarity, and also allowed the boilerplate code to be discounted.

Of the 30 programs pairs selected for detailed examination, eight pairs were judged to be plagiarized. The plagiarized program pair with the lowest score (38.9%) was substantially above the 30% threshold score, and we are satisfied that we have found all instances of plagiarism identified by JPlag. We also consider five additional plagiarized program pairs that were identified during preparatory experiments. Thus, our ground truth for Collection A has a total of 13 program pairs.

We have not established ground truth on Collection B, as its main purpose is to demonstrate the scalability of our approach. However, during our final experiment on Collection B, we identified six cases of historical plagiarism. In this instance, one program from Collection A, submitted in 2004, included high regions of similarity to six different programs submitted in previous years in Collection B. We included these six program pairs in our ground truth for the results of our final experiment.

Whale [33] reports that code plagiarism can often be identified from common traits. Among these are:

- identical function prototypes;
- identical magic numbers;
- clear slabs of structurally identical code;
- identical formatted strings in `printf` function calls;
- identical use of whitespace characters;
- similar use of branching constructs;
- consistent use of rare programming constructs;
- identical constant and variable names;
- distinctive commenting;
- same mistakes.

We add that students attempt to disguise plagiarism by:

- modifying comments, variable names, formatting, and output statements;
- replacing programming constructs with equivalents. For example, `switch`... `case` statements can be changed to `if`... `else` statements;
- re-ordering function definitions.

These techniques are not difficult to detect with a structure-oriented plagiarism detection system, as they do not have a high impact on the structure of a program. Interlingual code plagiarism, where program code is copied from one programming language to another—is yet another disguising technique that can be detected using structure-oriented approach [34].

To evaluate the effectiveness of alternative plagiarism detection schemes, we employ the *precision* and *recall* measures commonly used in information retrieval [35]. The precision P_r of a ranking method at a cut-off point r is defined as

$$P_r = \frac{\text{number of retrieved documents that are relevant}}{\text{number of retrieved documents}}$$

The recall R_r of a ranking method at a cut-off point r is defined as

$$R_r = \frac{\text{number of retrieved documents that are relevant}}{\text{number of relevant documents in collection}}$$

Consider the case where the user wishes to examine the top ten (r) results of a query. If six of these documents are relevant—or, in our application, are plagiarized—then the precision of this query is 60%. If there are 20 relevant documents in total, then the recall at ten documents is 30%.

All experiments were conducted on a 733 MHz Pentium III processor with 512 megabytes of RAM running Red Hat Linux 7.3 with a 2.4.18-18.7.xsmp kernel.

6. RESULTS

In this section, we discuss the results of experiments with various ranking algorithms used to identify candidate plagiarism. We examine the effectiveness of post-filtering with local alignment, and describe an experiment that demonstrates the scalability of our approach.

6.1. Index results

To identify candidate plagiarism, we used the Zettair search engine with four different ranking functions: cosine, BM25, identity, and PlagiRank. Figure 8 is an interpolated precision and recall graph of the effectiveness of the four ranking algorithms at various levels of recall, with results for the JPlag baseline and MOSS included for comparison.

The precision and recall scores for MOSS are poorer than those of JPlag, supporting our choice of baseline system. We experimented with two settings for the MOSS `-m` parameter, which controls the threshold at which MOSS ignores frequently appearing content. For example, with `-m` set to 10 (the default), content that appears in more than ten documents is considered to be common or 'boilerplate' content that can be reused legitimately, and associated documents are not reported by the system.

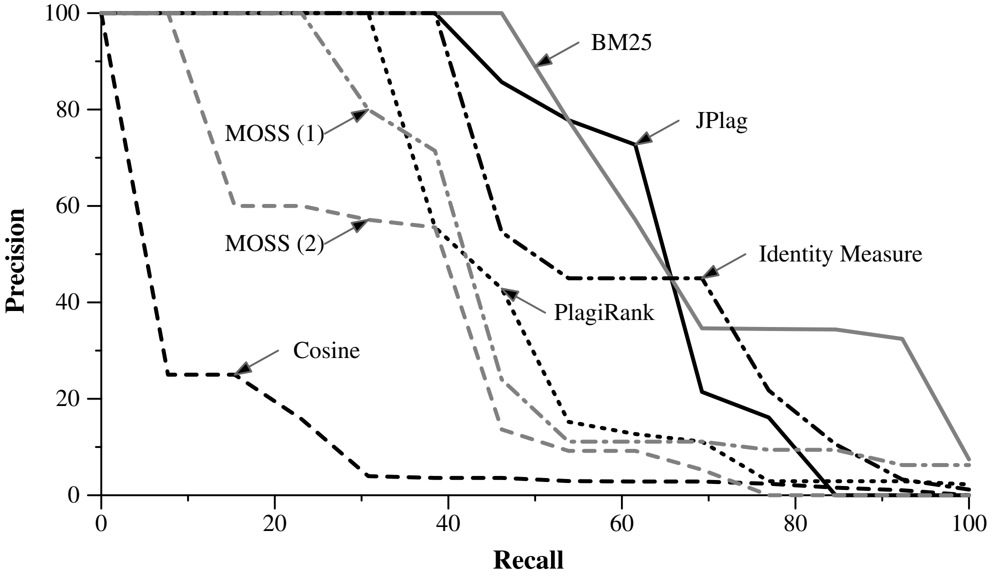


Figure 8. Interpolated precision-recall graph providing a comparison of the BM25, cosine, PlagiRank, and the identity similarity measures to our JPlag baseline on Collection A. We also include results for MOSS as a point of comparison.

We used two settings for this parameter in our experiment: the default value of 10, shown in the figure as MOSS (1), and the extreme value of 1 000 000, shown in the figure as MOSS (2). All matches are reported[‡] with the latter setting, producing results comparable to those produced by JPlag with its default settings.

For our collection, a base file was supplied to students and hence effectively ignoring any base file with a very large value of -m achieves an effect opposite to that we desire, and produces poorer results than the default -m parameter value. Using the default -m value and specifying the base file for Collection A did not produce any significant change in performance. The base file used was small—containing approximately 25 lines of mostly constant and variable definitions to clarify the desired structure of program code to the students.

The relatively poor performance of MOSS may be in part due to its lossy representation of the content of each document; only a subset of the fingerprints of each document are retained and indexed [6]. We also see that the PlagiRank and identity measures are not as effective as BM25. This is surprising, since these measures are designed for the specific task of plagiarism detection, while BM25 is optimized for general text retrieval.

[‡] MOSS submission script instructions: http://moss.cs.berkeley.edu/∼moss/general/scripts/moss20

DOI: 10.1002/spe

The BM25 similarity measure is the most effective, with the top six results all belonging to plagiarized pairs. In contrast, the cosine measure performs particularly poorly. We believe that the query term frequency—used in BM25 but missing from the Cosine measure—is especially important in this result. With queries of hundreds or thousands of n-grams, we can expect many duplicate n-grams, and it is important to appropriately handle those that appear very frequently in the query.

JPlag did not return a few relevant program pairs in the top 1000 results, leading to a 0% precision score above the 85% recall point. In practice, results at high recall levels are unlikely to of interest to a human reviewer given the reduced precision.

These results are encouraging given that there are 43 660 possible program pairs in our repository. This significantly reduces the number of program pairs that we must examine in detail. We found that the distribution of scores generated by Zettair using the BM25 measure is comparable to that of JPlag. Our lowest-ranking program returned a score of 40.4% when using the BM25 similarity measure. Therefore, a cut-off threshold of 30% (the same as the one we used for JPlag) is justified. This represents 810 program pairs to be post-filtered using local alignment.

6.2. Local alignment results

In our initial variant of local alignment, we used the scoring metrics of $\{\text{match} = 1, \text{mismatch} = -1, \text{indel} = -1\}$ as presented in Section 2.2. The score assigned to each program pair is taken to be the optimal alignment score from the local alignment matrix.

In our second variant of local alignment, we experimented with the match, mismatch and indel parameters to identify the optimal combination of these parameters. We found that increasing the match parameter value decreased effectiveness. The mismatch and indel penalty scores did not penalise less similar program pairs sufficiently, and produced many false positives. Increasing the mismatch and indel penalty scores increased the effectiveness of our results. The optimal combination of these parameters was $\{\text{match} = 1, \text{mismatch} = -3, \text{indel} = -2\}$.

In our third variant of local alignment, we considered multiple local alignments of two program token streams, instead of only their optimal alignment. We found that the optimal combination of parameters was $\{\text{match} = 1, \text{mismatch} = -3\}$, using a minimum match length threshold of 65. As described in Section 2.2, we ignore indels when performing multiple local alignment. While this threshold may appear excessive, it is appropriate given the high number of tokens—relative to the nine tokens of the JPlag minimum match length [3]—that we use. We also note that we do not perform exact matching, and still allow mismatch penalties. This accommodates both perfect and near-perfect matching of program regions, unlike JPlag, which only considers perfect matching regions. In our final variant of local alignment, we used multiple local alignments but considered all alignments above a threshold length on each diagonal. We found that the optimal combination of parameters remained the same.

Figure 9 shows the results of all variants of local alignment. Both variants of multiple local alignment produced the same results, and so only one plot is shown. All local alignment variants produced comparable results—not surprising since they are variations of same idea. At the conclusion of these experiments we had identified three new program pairs with evidence of highly similar program segments.

In the JPlag output, we find that the eight identified cases of plagiarism are ranked highly. However, it does not rank highly the five plagiarized pairs that we discovered through other means.

DOI: 10.1002/spe

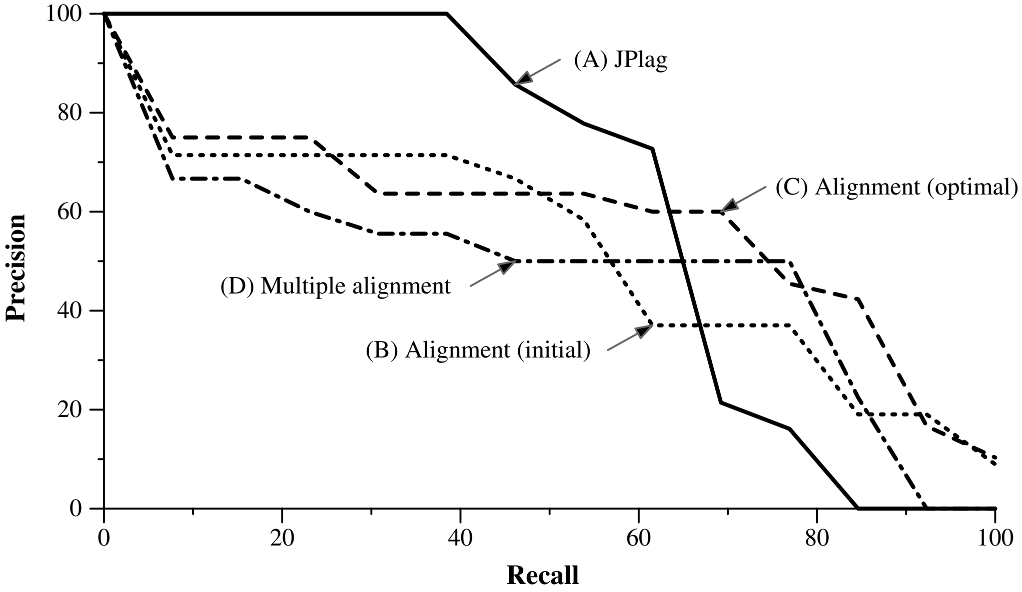


Figure 9. Interpolated precision/recall graph using Collection A providing a comparison of: (A) Our JPlag baseline; (B) our initial local alignment variant with parameters {match = 1, mismatch = −1 indel = −1}; (C) an optimal local alignment variant with parameters {match = 1, mismatch = −3 indel = −2}; (D) multiple local alignment taking into account all alignments on each diagonal of the local alignment matrix above a threshold length.

This suggests that JPlag works best where close matches are involved, and is less suitable for finding more distant matches. A possible explanation for the difference in these results is the underlying algorithms of the two systems. In the greedy string tiling approach of JPlag, the longest matches are identified first, and these matching regions are excluded when identifying later, smaller, matches. Local alignment also identifies these long matches; however, it also allows for imperfectly matching regions that contain short matches separated by mismatches or indels. For example, any detected single-token mismatch between two similar code segments causes them to be treated as separate—and less significant—sequences by JPlag. In contrast, when using the BM25 metric and n-grams, such a mismatch affects only the two neighbouring n-grams.

6.3. Scalability

In our final experiment, we demonstrate the efficiency, effectiveness, and scalability of our solution using the 61 540 programs in Collection B. We queried the large index with our original collection of 296 programs to ascertain whether reasonable levels of effectiveness could be maintained. The results are shown in Figure 10. We see that we are indeed able to maintain similar levels of

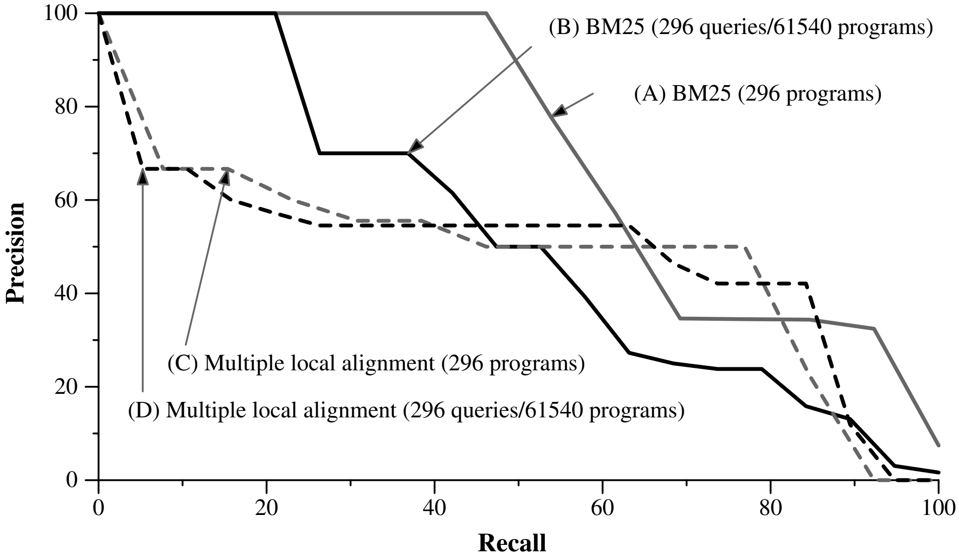


Figure 10. Interpolated precision/recall graph providing a comparison of: (A) BM25 similarity measure on Collection A; (B) BM25 similarity measure of results taken from Collection A used as queries on Collection B; (C) multiple local alignment on Collection A; (D) multiple local alignment of results taken from Collection A used as queries on Collection B.

effectiveness when using multiple local alignment as a second-pass filter for the BM25 similarity measure. BM25 on its own showed reduced precision at higher recall levels, though it continued to rank highly some of the more similar program pairs.

This graph has 19 points, whereas the experiments on Collection A contained 13 points. Multiple local alignment identified six cases of historical plagiarism on the large collection. These six pairs comprised of one program from our 296 program repository clearly matching segments of identical code from six programs in our 61 540 program repository. We suspect that these six programs shared the original source program and as a result they show significant similarities. We found that, in contrast to BM25, multiple local alignment consistently ranked these program pairs near the top.

This result is particularly pleasing because this kind of plagiarism cannot be found using current plagiarism detection techniques. As discussed in the introduction, it is generally possible to only check for plagiarism among submissions for a single assessment task. However, we have demonstrated that historical plagiarism detection is now practical. BM25 alone proved to be effective for nine program pairs, while post-filtering with multiple local alignment produced better results for the remaining ten program pairs.

To demonstrate the efficiency of the process, we compared the running times of processing each collection. Table III lists the running time corresponding to each of the four phases in our approach as discussed in Section 4: tokenization, index construction, querying, and local alignment. We see that Collection B is 207.9 times larger than Collection A. The time required for the tokenization and

Table III. The running times for the 296 program repository (Collection A) queried against an index of this repository, and for these same 296 programs queried against a 61 540 program repository (Collection B). We present the increase factor between the collection size and running times of the two collections.

	Programs	Tokenization time (s)	Construction time (s)	Querying time (s)	Alignment time (s)
Collection A	296	3.6	1.8	249.0	264.6
Collection B	61 540	1869.6	258.6	1441.8	723.6
Increase factor	207.9	312.7	90.3	5.7	2.6

construction phases is approximately proportional to the collection size. We expect that experiments with larger collections will confirm that this relationship is linear.

Our tokenization is based upon an existing plagiarism detection system (SIM), and we believe that our implementation can be greatly refined to reduce running time. A modest time saving is gained in the index construction phase, as we expect there to be more duplicate *n*-grams in a larger collection. Note that these two phases are the least significant, since the constructed index can be reused for any number of queries. If index updates are required, these can be performed offline or at times of low system activity.

The time taken for the querying phase is of most interest. We can see that a collection size increase of 207.9 times only results in a 5.7-fold increase in query time. This is very reasonable and clearly much better than a linear time increase. We expect the growth in size of the lexicon to be small: we are less likely to encounter large numbers of new *n*-grams as we progress further into a large collection. In our collections, we used 55 distinct tokens: for an *n*-gram size of 4, we can expect up to 55^4 (or approximately 9 million) entries in our lexicon. However, given the large number of duplicates, we expect the lexicon length to be significantly smaller. For example, Collection A used in our experiments contained 431 000 *n*-grams, 4500 of them distinct. As discussed in Section 2.1, we also expect the inverted lists to be highly compressible for larger collections. We expect the time required for the alignment phase to be unvarying across collection sizes, since we process only program pairs exceeding the 30% similarity threshold.

The running times we report here for the experimental framework are to indicate relative increases with collection size. We believe that a production version can exhibit much better absolute running times while maintaining its high effectiveness. By way of comparison, a complete run for Collection A using the network-based MOSS service takes approximately 30 seconds. The increase observed for the large repository is in part due to the non-optimal file access method used in the test code. Moreover, the absolute running times we report are inflated due to the large number (810 program pairs—those above the 30% similarity threshold) of programs we processed to underline the effectiveness of our approach. In practice, we expect that far fewer pairs would show sufficient similarity to be passed on to the second stage. The number of pairs is proportional to collection size, that is, we expect larger collections to return more candidate program pairs.

7. DISCUSSION AND FUTURE WORK

We have proposed an effective and scalable approach of using standard text search techniques to index all overlapping tokens extracted from computer programs for plagiarism detection in code. Importantly, we have compared the effectiveness of our approach to several state-of-the-art plagiarism detection systems for code; such a direct comparison has been absent from the literature [27,36].

We have shown that the BM25 similarity measure used for text retrieval can also be applied for plagiarism detection in code, and is indeed more effective than the cosine, identity, and PlagiRank measures, producing results comparable to those of the popular JPlag plagiarism detection system, and is notably better than JPlag for less obvious cases of plagiarism.

We have also shown that post-processing BM25 results using multiple local alignment is a highly scalable approach, maintaining precision for a large collection of program source code files, with only a minor increase in query time. The overhead of tokenizing and indexing a collection is proportional to collection size. Given that index updates are likely to be infrequent, this is likely to be of low importance in practice.

When testing our large collection using multiple local alignment, we identified six instances of historical plagiarism detection. This is an interesting result, as such cases cannot be detected with other current plagiarism detection techniques.

We see several promising areas of extension to this work. First, plagiarized programs are likely to have more than one segment of interest. While multiple local alignment did not produce results significantly better than single local alignment, we believe that further investigation of this scheme is justified.

Second, the granularity of source code tokenization is likely to be an influential parameter in the robustness of detection techniques, and further exploration is merited.

Third, it would be interesting to include additional similarity measures, such as those proposed by Broder [24,25], in further evaluation.

Fourth, while our approach automatically discounts frequently-occurring tokens, we see the need for a more detailed exploration of how best to handle boilerplate content that can be legitimately re-used.

Fifth, much plagiarism occurs using resources from the Web; techniques are needed to compile large repositories of code, including specialized Web crawlers to identify and collect program source code for indexing. The development of search engines such as Krugle[§] that are specifically designed to search for program source code is likely to attract research interest in such crawlers; however, it also portends an increase in the level of copying from the Web.

Finally, interactive query times and an intuitive user interface are critical to wide adoption; we plan to develop a more efficient implementation and add support displaying regions of interest to the user.

ACKNOWLEDGEMENTS

We thank Alexandra Uitdenbogerd for providing local alignment code, and the RMIT School of Computer Science and Information Technology for scholarship funding for the first author. This work was funded by the Australian Research Council.

[§] http://www.krugle.com

REFERENCES

1. Sheard J, Dick M, Markham S, Macdonald I, Walsh M. Cheating and plagiarism: Perceptions and practices of first year IT students. *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education*, Aarhus, Denmark, June 2002. ACM Press: New York, 2002; 183–187.
2. Merriam-Webster online dictionary. Merriam-Webster, Inc., Springfield, MA. http://www.m-w.com/ [February 2006].
3. Prechelt L, Malpohl G, Philippsen M. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science* 2002; **8**(11):1016–1038.
4. Gitchell D, Tran N. Sim: A utility for detecting similarity in computer programs. *Proceedings of the 30th SIGCSE Technical Symposium*, March 1999; 266–270.
5. Chawla M. An indexing technique for efficiently detecting plagiarism in large volumes of source code. *Honours Thesis*, RMIT University, Melbourne, Australia, October 2003.
6. Schleimer S, Wilkerson D, Aiken A. Winnowing: Local algorithms for document fingerprinting. *SIGMOD Conference on the Management of Data*, New York, June 2003. ACM Press: New York, 2003; 76–85.
7. Witten I, Moffat A, Bell T. *Managing Gigabytes: Compressing and Indexing Documents and Images* (2nd edn). Morgan Kaufmann: San Francisco, CA, 1999.
8. Trotman A. Compressing inverted files. *Information Retrieval* 2003; **6**(1):5–19.
9. Shannon CE. A mathematical theory of communication. *The Bell Systems Technical Journal* 1948; **27**:379–423, 623–656.
10. Robertson S, Walker S. Okapi/Keenbow at TREC-8. *Proceedings of the 8th Text Retrieval Conference (TREC-8)*, Gaithersburg, MD, November 1999. NIST, 1999; 151–162.
11. Smith T, Waterman M. Identification of common molecular subsequences. *Journal of Molecular Biology* 1981; **147**(1):195–197.
12. Altschul S, Gish W, Miller W, Myers E, Lipman D. Basic local alignment search tool. *Journal of Molecular Biology* 1990; **215**:403–410.
13. Morgenstern B, Frech K, Dress A, Werner T. DIALIGN: Finding local similarities by multiple sequence alignment. *Bioinformatics* 1998; **14**(3):290–294.
14. Hoad T, Zobel J. Methods for identifying versioned and plagiarised documents. *Journal of the American Society for Information Science and Technology* 2002; **54**(3):203–215.
15. Heintze N. Scalable document fingerprinting. *1996 USENIX Workshop on Electronic Commerce*, November 1996; 191–200.
16. Donaldson J, Lancaster A, Sposato P. A plagiarism detection system. *Proceedings of the 12th SIGCSE Technical Symposium on Computer Science Education*. ACM Press: New York 1981; 21–25.
17. Verco K, Wise M. Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. *Proceedings of Australian Conference on Computer Science Education*, Sydney, Australia, July 1996. ACM Press: New York, 1996; 81–88.
18. Bowyer K, Hall L. Experience using MOSS to detect cheating on programming assignments. *Proceedings of the 29th ASEE/IEEE Frontiers in Education Conference*, San Juan, Puerto Rico, November 1999. IEEE Computer Society: Los Alamitos, CA, 1999; 18–22.
19. Wise MJ. Running Karp-Rabin matching and greedy string tiling. *Technical Report TR 463*, School of Information Technologies, The University of Sydney, Sydney, Australia, March 1993.
20. Karp R, Rabin M. Efficient randomised pattern-matching algorithms. *IBM Journal of Research and Development* 1987; **31**(2):249–260.
21. Manber U. Finding similar files in a large file system. *Proceedings of the USENIX Winter 1994 Technical Conference*, San Francisco, CA, January 1994. ACM Press: New York, 1994; 1–10.
22. Baker B. On finding duplication and near-duplication in large software systems. *Proceedings of the 2nd Working Conference on Reverse Engineering*, Los Alamitos, CA, July 1995, Wills L, Newcomb P, Chikofsky E (eds.). IEEE Computer Society Press: Los Alamitos, CA, 1995; 86–95.
23. Baker B, Manber U. Deducing similarities in Java sources from bytecodes. *Proceedings of Usenix Annual Technical Conference*, Berkeley, IL, June 1998. ACM Press: New York, 1998; 179–190.
24. Broder A. On the resemblance and containment of documents. *Proceedings of Compression and Complexity of Sequences*, Positano, Italy, June 1998. IEEE Computer Society: Los Alamitos, CA, 1998; 21–29.
25. Broder A, Glassman S, Manasse M, Zweig G. Syntactic clustering of the Web. *Selected Papers from the 6th International Conference on World Wide Web*, Santa Clara, CA, 1997, Enslow P, Genesereth M, Patterson A (eds.). Elsevier Science: Amsterdam, 1997; 1157–1166.
26. Irving RW. Plagiarism and collusion detection using the Smith-Waterman algorithm. *Technical Report TR-2004-164*, University of Glasgow Computing Science Department Research Report, April 2004.
27. Mozgovoy M, Fredriksson K, White D, Joy M, Sutinen E. Fast plagiarism detection system. *Proceedings of the International Symposium on String Processing and Information Retrieval (SPIRE2005)*, Buenos Aires, Argentina, November 2005 (*Lecture Notes in Computer Science*, vol. 3772). Springer: Heidelberg, 2005; 267–270.

DOI: 10.1002/spe

28. Zobel J, Moffat A, Sacks-Davis R. Searching large lexicons for partially specified terms using compressed inverted files. *Proceedings of the 19th International Conference on Very Large Databases*, Dublin, Ireland, August 1993, Agrawal R, Baker S, Bell D (eds.). Morgan Kaufmann: San Francisco, CA, 1993; 290–301.
29. Chiueh T, Huang L. Efficient real-time index updates in text retrieval systems. *SUNY Stony Brook ECSL Technical Report ECSL-TR-66*, State University of New York, New York, April 1999.
30. Lester N, Zobel J, Williams HE. In-place versus re-build versus re-merge: Index maintenance strategies for text retrieval systems. *Proceedings of the Australasian Computer Science Conference*, Dunedin, NZ, January 2004, Estivill-Castro V (ed.). Australian Computer Society, 2004; 15–22.
31. Tomasic A, García-Molina H, Shoens K. Incremental updates of inverted lists for text document retrieval. *SIGMOD'94: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, MN, 1994. ACM Press: New York, 1994; 289–300.
32. Jansen B, Spink A, Bateman J, Saracevic T. Real life information retrieval: A study of user queries on the Web. *ACM SIGIR Forum* 1998; **32**(1):5–17.
33. Whale G. Detection of plagiarism in student programs. *Proceedings of the 9th Australian Computer Science Conference*, Canberra, Australia, January 1986 (*Australian Computer Science Communications*, vol. 8). Australian Computer Society, 1986; 231–241.
34. Arwin C, Tahaghoghi SMM. Plagiarism detection across programming languages. *Proceedings of the 29th Australasian Computer Science Conference*, Hobart, Australia, January 2006 (*Conferences in Research and Practice in Information Technology (CRPIT)*, vol. 48), Estivill-Castro V, Dobbie G (eds.). Australian Computer Society, 2006; 277–286.
35. Baeza-Yates R, Ribeiro-Neto B. *Modern Information Retrieval* (1st edn). Addison-Wesley, 1999.
36. Culwin F, MacLeod A, Lancaster T. Source code plagiarism in UK HE computing schools: Issues, attitudes and tools. *Technical Report SBU-CISM-01-01*, South Bank University School of Computing, Information Systems and Mathematics, September 2001.