# EFFICIENT AND EFFECTIVE PLAGIARISM DETECTION FOR LARGE CODE REPOSITORIES

Steven Burrows [1], Seyed M. M. Tahaghoghi [1] & Justin Zobel [1]

[1] School of Computer Science and Information Technology,
RMIT University

ABSTRACT: The copying of programming assignments is a widespread problem in academic institutions. Manual plagiarism detection is time-consuming, and current popular plagiarism detection systems are not scalable to large code repositories. While there are text-based plagiarism detection systems capable of handling millions of student papers, comparable systems for code-based plagiarism detection are in their infancy. In this paper, we propose and evaluate new techniques for code plagiarism detection. Using small and large collections of programs, we show that our approach is highly scalable while maintaining similar levels of effectiveness to that of JPlag.

## INTRODUCTION

Code plagiarism is the reuse of program structure and programming language syntax either from another student or from online resources. Plagiarism is a problem that universities across the world need to deal with. A joint study conducted on 287 Monash University and Swinburne University students [SDM+02] revealed that between 69.3% and 85.4% admitted to cheating. With such a high level of cheating, it is imperative that we have adequate mechanisms to detect plagiarism. In Computer Science, this problem is serious due to the abundance of web sites that have easily accessible program code.

For large class sizes, manual plagiarism detection is a slow and laborious process and therefore impractical. Many departments use automated plagiarism detection systems. Unfortunately, these are restricted by upper limits on the volume of programs that they can check. An efficient, effective, and scalable solution is needed. Such a solution would allow staff to search for plagiarism against large volumes of previous assignments, courses that have content in common, and very large collections of programs obtained from the Web.

Most previous approaches to plagiarism detection perform the task in an exhaustive pairwise fashion. This is the approach used in JPlag [PMP02]. This is not scalable to large code repositories. Our approach is to construct an index of student programs and query this index once for each program. We refine the results generated by this process using the local alignment approximate string matching technique. Our initial experiments conducted on a repository of 296 programs allowed us to identify the most efficient and effective indexing, querying, and local alignment techniques. Further experiments on a much larger repository of 61,540 programs revealed that the efficiency and effectiveness of our approach scales to large code repositories.

## BACKGROUND

The plagiarism detection techniques described in this paper use search engines and local alignment to identify likely matches. In this section, we introduce the underlying techniques of each.

### Search Engines

Arasu et al. [ACGM+01] describe a web search engine as a tool that allows users to submit a query as a list of keywords, and receive a list of possibly relevant web pages that contain most or all of the keywords. The purpose of a search engine is to index huge volumes of data in a compressed fashion for efficient retrieval. To facilitate efficient retrieval, search engines use an *inverted index*.

Figure 1A shows that an inverted index is comprised of two main components: a *lexicon* and a series of *inverted lists*. The lexicon stores all distinct words in the collection, in our example the names of fruits. For example, we can see that the term “banana” has a term frequency of 1, which tells us that

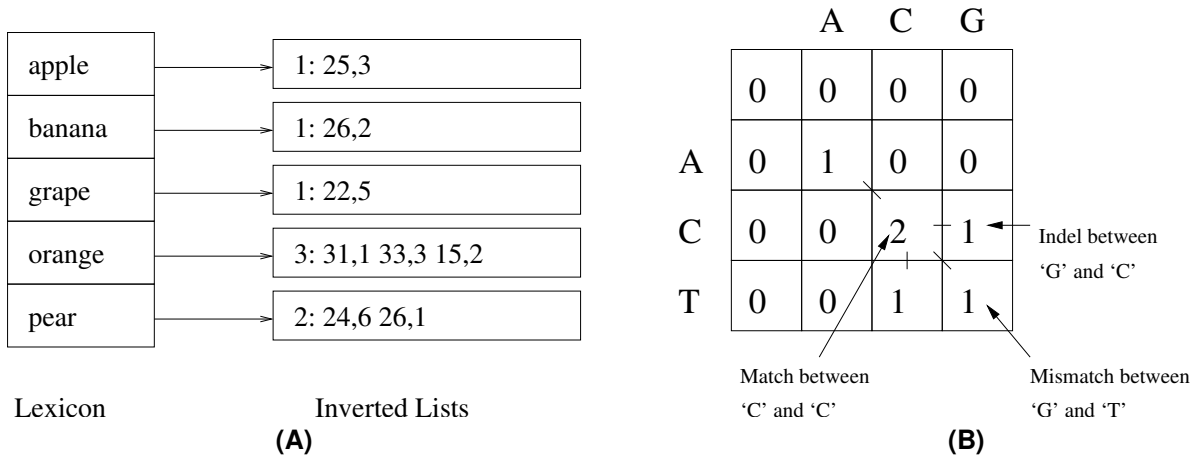


*Figure 1: (A) A simple inverted index comprising a lexicon of five terms and corresponding inverted lists. (B) A local alignment matrix for the sequences "ACG" and "ACT". The arrows indicate a match, a mismatch, and an indel.*

this term occurs in one document. The inverted list indicates that it appears twice in `document 26`. This index demonstrates why a search engines is highly scalable for large collections of data. Each term need only be stored once in our lexicon and the inverted lists comprise of integers that are highly compressible [WMB99].

The *query engine* is the component that the user most closely interacts with. The user requests information by providing a query string that the query engine parses. The query engine retrieves matching inverted lists for the terms the user provided. The *ranking engine* provides matching results to the user ranked by decreasing estimated relevance. Relevance is estimated using a similarity measure. Examples of similarity measures include Cosine [WMB99], Okapi BM25 [RW99], and PlagiRank [Cha03].

Local Alignment

*Local alignment* [SW81] is an approximate string matching technique. Local alignment is used in bioinformatics to find organisms of similar *homology*, that is, organisms with a similar evolutionary ancestry. In this paper, we investigate the application of local alignment to plagiarism detection.

Consider the local alignment of the two very short sequences "ACG" and "ACT" presented in Figure 1B. Local alignment can be calculated on two sequences $s$ and $t$, of lengths $l_1$ and $l_2$ using a matrix of size $(l_1 + 1) \times (l_2 + 1)$ [SW81]. In our example, the strings of length three produce a matrix with sixteen cells and the first row and column of the matrix are initialised with zeros.

When computing the score for each cell, we consider the maximum possible score resulting from a match, mismatch, insertion, or deletion between each pair of characters between the two sequences.

We award a score increase for any pair of characters in the local alignment matrix that *match*. In our example, the character 'C' from "ACG" 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 one, we obtain a score of two for this match. Similarly, we can penalise any pair of characters that *mismatch*. We have a mismatch between the character 'G' from "ACG" and the character 'T' from "ACT", and so the score is reduced by one. We can also optimise the alignment of two sequences by inserting or deleting additional characters. These attract a penalty, as they also represent changes to the sequences. We refer to insertions and deletions collectively as *indels*. We have an indel between the character 'G' from "ACG" and the character 'C' from "ACT". This effectively means that we have either deleted the character 'T' from the string "ACT" or inserted a blank before it. Indel scores reduce the score to the previous cell to the left or upwards by one depending on which sequence the indel penalty is applied to. 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 is generated. Given that we are only interested in the similarity of regions in local alignment, we enforce a minimum score of zero.

To find the optimal alignment, we traverse our local alignment matrix to locate the highest score at any cell. In our example in Figure 1B, the highest score is two. We then follow our traceback path (that has been indicated by dashes) until we reach a zero. This traceback path denotes the highest scoring preceding region to the left, above, or to the upper-left of the current cell. The traceback path in our example represents the alignment "AC".

It is not uncommon to have more than one optimal alignment or several regions of high similarity. We refer to this as *multiple local alignment*. Identifying multiple local alignments is much more difficult than identifying a single optimal alignment. Morgenstern et al. [MFDW98] propose that this problem can be overcome by ignoring indel penalties and just identify local regions of similarity in the diagonals of the local alignment matrix.

## RELATED WORK

In this section, we show that existing plagiarism detection methods fall into two main categories: text-based and code-based. We also discuss the two main classifications of code-based methods: attribute-oriented and structure-oriented.

### Text-based Systems

One of the largest text-based plagiarism detection systems available is Turnitin.[1] As of November 2004, Turnitin claims that their database exceeds 4.5 billion documents. The techniques used for text plagiarism are not the same as code plagiarism. Our experiments with an online text-based plagiarism detection tool showed that text-based systems ignore coding syntax in favour of comparing normal English words for plagiarism detection – a severe defect. Text-based systems are not discussed further for this reason.

### Attribute-oriented Code-based Systems

Attribute-oriented plagiarism detection systems measure properties of assignment submissions. The similarity of two programs is approximated by the differences between these attributes. Unfortunately, attribute-oriented systems are highly limited. Verco and Wise [VW96] report that attribute-oriented systems perform best in the detection of plagiarism where very close copies are involved. A common method of plagiarising is to add additional redundant code statements to a program. Given that attribute-oriented systems take all lines of code into account, the resulting attribute scores differ greatly. Moreover, it is difficult to detect copies code segments.

### Structure-oriented Code-based Systems

Structure-oriented systems [BH99, GT99, PMP02] create lossy representations of programs and compare these to detect plagiarism. These systems deliberately ignore easy-to-modify elements such as comments, additional white space, and variable names.

Structure-oriented systems are less susceptible to the addition of redundant statements that can mislead attribute-oriented systems because these systems perform local comparisons. To effectively 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 utilised plagiarism detection system. It can be argued that this amount of work is similar to that needed to complete the assignment legitimately, and hence students who know that structural plagiarism detection is employed are less likely to plagiarise. Three of the most popular structure-oriented plagiarism detection systems are JPlag [PMP02], MOSS [BH99], and SIM [GT99].

While these systems are effective at detecting plagiarism, they are limited by the volume of the code that they can process at any one time. JPlag is limited because its underlying comparison algorithm

[1] `http://www.turnitin.com`

```
   Source Code                          Token Descriptor  Token  4-Gram
   -----------                          ----------------  -----  ------
1. int main(void) {                     BEGIN             B      BVLA
2.    int var;                          VARIABLE_DEF      V      VLAM
3.    for (var=0; var<5; var++) {       BEGIN_LOOP        L      LAMR
4.       printf("%d\n", var);           APPLY             A      AMRE
5.    }                                 END_LOOP          M
6.    return EXIT_SUCCESS;              RETURN            R
7. }                                    END               E
```

*Figure 2: Demonstration of how to convert a simple C program into a token stream and 4-grams. Note that we only provide a simplistic example here; in our approach, we use a more fine-grained token set that generates 3 tokens per line of code on average.*

is applied exhaustively to all program pairs and the JPlag administrators enforce an upload limit of 9.5 Megabytes [Cha03]. MOSS is limited because all processing is done in main memory.[2] Our experiments demonstrated that this limitation exists in SIM as well. To the best of our knowledge, PlagiIndex [Cha03] is the only system that addresses this problem.

## SCALABLE PLAGIARISM DETECTION

Our approach to plagiarism detection is to find matches with an index, then use local alignment. The approach is as follows. First, we tokenise all assignment submissions into a format suitable for indexing. Second, we construct an index of all assignment submissions. Third, we query assignment submissions against the index to identify candidate plagiarism. Finally, we use local alignment on assignment submissions that were ranked highly during querying to refine our results. We now describe our approach in detail.

### Tokenisation

In our approach, we require our assignment submissions to be tokenised into two different formats. Earlier, we discussed that the lexicon in an inverted index can be used to store all distinct words in a collection, as search engines are used to index text. Programming code has little resemblance to written text; it contains large amounts of specialised programming characters. These characters are ignored by text search engines when an index is built. Therefore, we need to represent our assignment submissions in a more suitable format for indexing. Prior to building an index of programs, we need to transform programs into a format our indexer can understand.

Similarly, programming code is unsuitable for local alignment, so we need to tokenise our submissions into a second format for this task. It is desirable to have a single token to represent each programmatically significant piece of code. This reduces the file size while preserving program structure.

To prepare the code for local alignment, we substitute each programming construct with an alphanumeric character to form a contiguous token stream. This is demonstrated in Figure 2. We need to further modify this token stream for indexing in our search engine. We need to break it into smaller pieces, so that each can be indexed separately. To achieve this task, we generate an *N-gram* representation of our token stream. For example, consider our token stream from Figure 2 “BVLAMRE”; this can be decomposed into 4-grams: “BVLA VLAM LAMR AMRE”. These 4-grams were generated using the *sliding window* technique. The sliding window technique generates N-grams by moving a “window” of size $N$ across all parts of the string from left to right. In our example, we moved a window of size 4 across our token stream “BVLAMRE” to produce four 4-grams.

N-grams can be used for faster query pattern matching [WMB99] in search engines; an important aspect of information retrieval. Zobel et al. [ZMSD93] discuss that N-grams can be used for partial query matching, that is, where only parts of query words are supplied.

[2] Claim based on personal communication with MOSS author A. Aitken.

| Rank | Query File | Index File | Raw Score | Similarity Score |
|---|---|---|---|---|
| 1. | 0020.c | 0020.c | 369.45 | 100.00% |
| 2. | 0020.c | 0103.c | 344.85 | 93.34% |
| 3. | 0020.c | 0092.c | 189.38 | 51.26% |
| 4. | 0020.c | 0151.c | 185.05 | 50.09% |
| 5. | 0020.c | 0267.c | 167.82 | 45.43% |

*Table 1: Results of the program `0020.c` compared to an index of 296 programs.*

The use of N-grams is an appropriate method of performing structural plagiarism detection because any change to the source code only affects a few neighbouring N-grams. The modified version of the program still has a large percentage of unchanged N-grams, hence detecting plagiarism in this program remains plausible. Chawla [Cha03] found that the most appropriate size of N-grams for his PlagiIndex system is four, and we use this size for our approach.

Index Construction

The second step is to create an inverted index of these N-grams. We discussed a basic inverted index of fruit names earlier. An inverted index of N-grams requires our N-grams to be inserted into the lexicon and the corresponding inverted lists to be created.

Note that we only need to build the index once. We can reuse the same index for any combination of queries. This contrasts with JPlag, which computes the similarity of all assignment submissions exhaustively every time.

Querying

The next step of our approach is to query the index. Each query is an N-gram representation of a program. Each query returns the ten most similar programs matching the query program and these are organised from most similar to least similar. If the query program is one of the indexed programs, we would expect this result to produce the highest score. We assign a similarity score of 100% to this top match. All other programs are given a similarity score relative to the top score.

For example, Table 1 presents the top five results of one N-gram program file (`0020.c`) compared against a 296 program index. In this example, we can see that the file scored against itself generates the highest relative score of 100.00%. This score is used to generate a relative similarity score for all other results. We can also see that the 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 plagiarised. The other programs have much lower scores indicating that these are unlikely to be plagiarised. This process is repeated for every query file. When complete, we collate the results. All candidate matches are ranked from most similar to least similar. We modified the Zettair search engine[3] for all our indexing and querying tasks.

Local Alignment

In the final step, we use local alignment to evaluate the similarity of program pairs that were highly ranked in the previous step. Earlier, we discussed the basic local alignment algorithm with the default parameters match = $1$, mismatch = $-1$ and indel = $-1$. We use these parameters for our first implementation.

We also examine two refinements to 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.

The input to our local alignment process is the output of our query results from the previous step above a threshold similarity score. This information tells us which program pairs should be processed further. The results are presented to the user in order of most similar to least similar. As with any plagiarism detection system, manual verification of the top results is required to remove false positives.

[3] `http://www.seg.rmit.edu.au/zettair/index.php`

## EVALUATION

In this section, we describe our data sets and the assessment of which pairs of programs are deemed to be plagiarised. We also present techniques to measure the effectiveness of our results.

### Data Set

For our initial experiments, we chose a collection of 296 student programs written in the C programming language for an assignment of an introductory C programming course.

In our final experiment, we use a repository of 61,540 programs archived by the School of Computer Science and Information Technology at RMIT. This collection represents several years of previously submitted C programming assignments with our test collection of 296 programs included.

### Ground Truth

To establish the *ground truth* (knowledge of plagiarism), it is impractical to inspect each program pair manually. 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. To conduct our ground truth work, we examined all program pairs that had a JPlag similarity score of at least 30%. We found that there were thirty program pairs with at least this score in our collection and we judged eight pairs to be plagiarised.

We included five additional program pairs that were identified from the results of our initial experiments. This additional ground truth was established by examining the top thirty ranked program pairs in each experiment we conducted.

During our final experiment, we identified six cases of historical plagiarism. In this instance, one program from our 296 program collection of assignments submitted in 2004 included high regions of similarity to six different programs submitted in previous years. These were also included in our ground truth.

### Effectiveness Measurement

*Precision* and *recall* are common ways of measuring effectiveness [BYRN99]. We use these measures to evaluate our results. The precision of a ranking method is defined as the proportion of retrieved documents that are relevant. The recall of a ranking method is defined as the proportion of relevant documents that are retrieved.

## RESULTS

To identify candidate plagiarism, we tested three different ranking functions within the framework of our search engine. We adapted the Cosine, Okapi BM25, and PlagiRank similarity measures to our system. We found that Okapi performed better than PlagiRank and Cosine. The effectiveness of Okapi was comparable to that of JPlag.

We found that all implementations of single and multiple local alignment performed similarly. At the conclusion of these experiments, we had identified three new program pairs with evidence of highly similar program segments and we believe this approach has potential. Once again, the effectiveness was comparable to JPlag.

In our final experiment, we demonstrate the efficiency, effectiveness, and scalability of our solution using the 61,540 program repository.

We queried our large index with our original collection of 296 programs to examine whether similar levels of effectiveness could be maintained for both multiple local alignment and the Okapi similarity measure, and we present these results in Figure 3. We found that we were able to maintain very similar levels of effectiveness for our implementation of multiple local alignment demonstrating excellent scalability. We found that Okapi did not scale as well. However, it continues to rank some of the most similar program pairs very highly. We were pleased to discover that our implementation of multiple local alignment

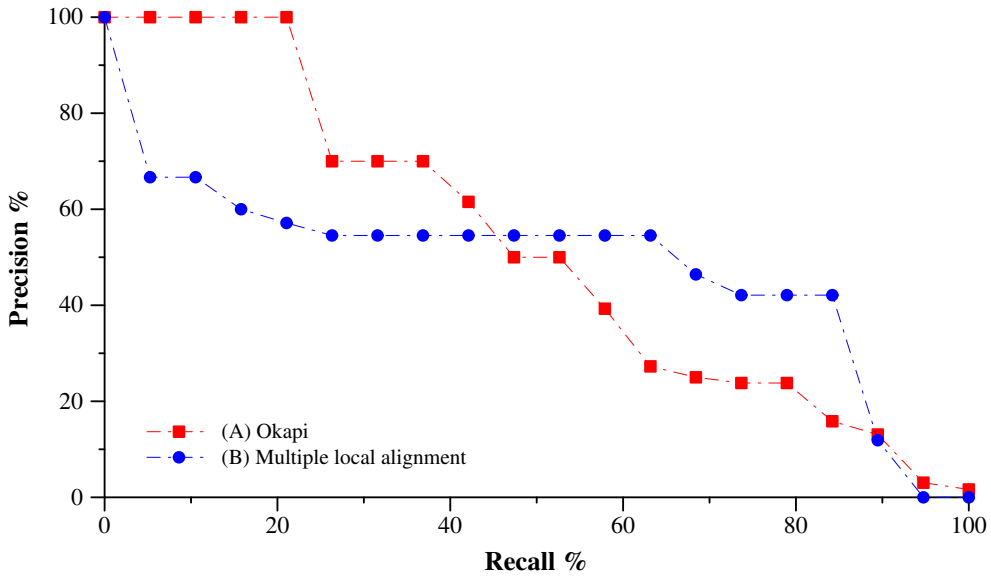


*Figure 3: Interpolated precision/recall graph providing a comparison of the Okapi similarity measure and multiple local alignment from results of 296 queries executed on the large index of 61,540 programs.*

identified six cases of historical plagiarism on the large collection accounting for these new results. These six pairs comprised of one program from our 296 program repository matching clear sections of identical code from six programs in our 61,540 program repository. We found that unlike Okapi, multiple local alignment ranked these program pairs consistently. This variability is due to noise generated from other parts of the programs. This result can be observed in Figure 3 between the 45% and 90% recall levels where multiple local alignment out-performs Okapi.

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, our system has demonstrated that we are now capable of performing historical plagiarism detection. In summarising the effectiveness of our results, we found that Okapi performed better for nine program pairs, and local alignment performed better for the remaining ten program pairs.

To demonstrate the efficiency of the process, we compared the running times of our original experiment with just 296 programs, and the running times of this experiment. These details are presented in Table 2. The time taken for the tokenisation and construction phases were proportionate to the collection size as expected. These phases are of of less importance if we reuse our token files and indexes. The time taken for the querying phase is of most interest. A collection size increase of 208 times only results in the querying phase taking 5.7 times longer. This can be attributed to a number of factors. Firstly, we expect the growth size of the lexicon to be small because we are less likely to encounter consistently large numbers of new N-grams as we progress further into the large collection. We also expect the inverted lists to be very highly compressible for larger collections. The running time for the local alignment phase is proportionate to the number of program pairs that were post-filtered.

## CONCLUSION

From our initial results, we found that the Okapi similarity measure and local alignment performed with similar levels of effectiveness to each other and that of JPlag. We found that JPlag ranked the most blatant cases of plagiarism well, but demonstrated poor effectiveness on less obvious cases of plagiarism compared to Okapi and local alignment. In terms of scalability, we demonstrated that very large increases in index size has a minor impact on querying times. Time taken to tokenise the collection and

| Collection | Tokenisation | Construction | Querying | Alignment |
|---|---|---|---|---|
| 296 | 6 sec | 3 sec | 255 sec | 281 sec |
| 61,540 | 1,876 sec | 271 sec | 1,443 sec | 726 sec |
| Increase | 312.7 | 90.3 | 5.7 | 2.6 |

*Table 2: The running times for the 296 program repository queried against an index of this repository, and for these same 296 programs queried against a 61,540 program repository. We present the increase factor between the collection size and running times of the two collections.*

to build the index, however, is proportionate to collection size, but this is of little importance if the token files and indexes are reused. Time taken to perform local alignment is almost constant; the total amount of time depends on the number of query results that are post filtered. When testing our large collection using local alignment, we identified six cases of historical plagiarism detection. Current popular plagiarism detection techniques are not scalable to handle this problem. We found that local alignment ranked all of these matches similarly making the identification of these programs easy. This was not the case for Okapi. We found that 4-gram program segments used to calculate similarity contained too much noise to produce consistent scores for these programs.

REFERENCES

[ACGM+01] A. Arasu, J. Cho, H. Garcia-Molina, A. Paepcke, and S. Raghavan. Searching the Web. In *ACM Transactions on Internet Technology*, volume 1, pages 2–43. ACM Press, 2001.

[BH99] K. Bowyer and L. Hall. Experience using MOSS to detect cheating on programming assignments. In *Proceedings of the 29th ASEE/IEEE Frontiers in Education Conference*, pages 18–22, San Juan, Puerto Rico, November 1999. IEEE.

[BYRN99] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, first edition, 1999.

[Cha03] M. Chawla. An indexing technique for efficiently detecting plagiarism in large volumes of source code. Honours thesis, RMIT University, Melbourne, Australia, October 2003.

[GT99] D. Gitchell and N. Tran. Sim: A utility for detecting similarity in computer programs. In *Proceedings of the 30th SIGCSE Technical Symposium*, pages 266–270, March 1999.

[MFDW98] B. Morgenstern, K. Frech, A. Dress, and T. Werner. Dialign: Finding local similarities by multiple sequence alignment. *Bioinformatics*, 14(3):290–294, April 1998.

[PMP02] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038, November 2002.

[RW99] S. Robertson and S. Walker. Okapi/Keenbow at TREC-8. In *Proceedings of the Eighth Text Retrieval Conference (TREC-8)*, pages 151–162, Gaithersburg, Maryland, November 1999.

[SDM+02] J. Sheard, M. Dick, S. Markham, I. Macdonald, and M. Walsh. Cheating and plagiarism: perceptions and practices of first year IT students. In *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education*, pages 183–187, Aarhus, Denmark, June 2002. ACM Press.

[SW81] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, March 1981.

[VW96] K. Verco and M. Wise. Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. In *Proceedings of Australian Conference on Computer Science Education*, pages 81–88, Sydney, Australia, July 1996.

[WMB99] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, second edition, 1999.

[ZMSD93] J. Zobel, A. Moffat, and R. Sacks-Davis. Searching large lexicons for partially specified terms using compressed inverted files. In R. Agrawal, S. Baker, and D. Bell, editors, *Proceedings of the 19th International Conference on Very Large Databases*, pages 290–301, Dublin, Ireland, August 1993. Morgan Kaufmann.