# Plagiarism and Collusion Detection using the Smith-Waterman Algorithm

Robert W. Irving

*Department of Computing Science, University of Glasgow,*
*Glasgow G12 8QQ, UK.*
*email:* rwi@dcs.gla.ac.uk
*telephone:* 44-141-330-4478
*fax:* 44-141-330-4913

**Abstract**

We investigate the use of variants of the Smith-Waterman algorithm to locate similarities in texts and in program source code, with a view to their application in the detection of plagiarism and collusion. The Smith-Waterman algorithm is a classical tool in the identification and quantification of local similarities in biological sequences, but we demonstrate that somewhat different issues arise in this different context, and that these factors can be exploited to yield significant speed-up in practice. We include empirical evidence to indicate the practicality of the approach and to illustrate the efficiency gains.

Key Words: plagiarism detection, string comparison, local similarity, Smith-Waterman algorithm.

## 1 Introduction and background

### Plagiarism and collusion

Plagiarism and collusion in students' assessed work are issues of increasing concern to the academic community as a whole. By *plagiarism* we mean the submission of part or all of another person's work as if it were ones own, without the knowledge of the author, and with intention to deceive. *Collusion*, on the other hand is the submission of work as ones own when (at least some of) that work has been done partly or wholly by another person, and that other person is party to the deception.

A variety of reasons have been proposed for the apparent increase in these forms of cheating, particularly the availability of electronic communication and internet access. But it is not our purpose here to pursue the reasons and the implications, but rather to consider the issue of detection, and in particular,

to describe one approach, seemingly not previously discussed in the literature, that has been successfully employed to identify likely cases of collusion.

In many disciplines, there is concern over Web-based plagiarism, whereby students use material, unattributed, from one or more sources on the World Wide Web. However the detection method that we describe here addresses only collusion, in which student A submits work wholly or partly done by student B, or the form of plagiarism in which student A somehow obtains a copy of the work of student B, without his knowledge, and submits it (or a thinly disguised version of it) as his own. Hence, our underlying assumption is that all of the source material that need be examined is directly available in the form of the students' submissions. This is sometimes referred to as *intra-corporal* detection [6], where attention is restricted to a given corpus of documents. (However, web-based plagiarism may well be detected in cases where two or more submissions contain possibly edited versions of material taken from the same web source.)

**Collusion detection**

The method that we propose can be applied to any form of textual material, such as essays, reports, etc. Unlike many existing techniques for collusion detection, it does not depend on statistical properties, such as counts of particular words, but rather on structural similarities between (parts of) texts. A special case involves collusion in computer programming assignments, which has been an area of major concern to computer science educators over many years. The only difference in the approach to this special case is the way in which the source material is parsed. Ordinary textual material will be parsed as a sequence of words, where the term *word* is given an appropriate precise meaning, whereas a computer program will be parsed as a sequence of lexical entities of the particular programming language. In either case, a student's submission is ultimately represented as a sequence of symbols over some finite alphabet; in the case of plain text the alphabet size is effectively the number of words available in the language, whereas in the case of computer programs it is (more or less) the number of different lexical entities.

For implementation purposes, we may represent the lexical entities by integers, or by characters from some sufficiently large alphabet, such as Unicode. So, for example, the text *A horse, a horse, my kingdom for a horse.* might be represented as the sequence $< 1\ 2\ 1\ 2\ 3\ 4\ 5\ 1\ 2 >$, with the correspondence between lexemes and integer tokens as shown in Table 1. Here we have chosen to discard punctuation symbols.

Once the initial lexical analysis phase has been carried out on all submissions in a corpus, the comparison phase involves only the resulting symbol sequences or *strings*. Hence any of a wide variety of methods that can be used to detect similarities in two strings might potentially be used as an aid to collusion detection.

When seeking to detect collusion, it is vital to have some insight into the methods that are typically used in attempts to disguise it.

In essays or reports, one party may 'borrow' one or more sections from another party. S/he may then alter these sections internally by making a variety

| | |
|---|---|
| a | 1 |
| horse | 2 |
| my | 3 |
| kingdom | 4 |
| for | 5 |

Table 1: A mapping of lexemes to tokens

of minor alterations that involve the insertion, deletion, or substitution of words, and the relative positions and separations of these sections within the document as a whole may then be subject to unpredictable changes.

In the case of computer programs, certain changes can easily be made without affecting the semantics — for example, changes in comments, character strings, layout and stylistic conventions, and systematic global editing of user-defined identifiers. Changes of this kind can easily be factored out at the lexical analysis stage. In addition, the relative order of program components — classes, subprograms, etc. — can be changed, just as is the case for sections or paragraphs in textual material.

Whatever changes are made in an attempt to disguise collusion, it can be expected that the corresponding token strings will contain one or more sections that are 'highly similar', and the occurrence of such sections will indicate a suspicion, either of collusion, or of the use of a common source of reference (attributed or not as the case may be). However, quantifying what is meant by 'highly similar' is a difficult matter. Ultimately, human judgment must be used to assess the significance of any degree of similarity identified algorithmically.

Our problem therefore can be viewed as that of finding 'highly similar' sections in strings over some fixed alphabet. This is a problem that is quite familiar in the domain of computational biology, where the identification of good 'local alignments' between strings that represent protein or DNA sequences is an everyday requirement [3]. The Smith-Waterman algorithm [8], a classical dynamic programming technique, is one of the key tools in that domain, and therefore it is natural to seek to adapt it to apply in this quite different context.

The issue of algorithm efficiency is of real practical significance here. If we wish to apply methods of string comparison effectively to a corpus of $n$ documents, then we have little option but to make all $\binom{n}{2} = n(n-1)/2$ pairwise document comparisons. When $n$ is, say, of the order of 300, this means almost 45000 pairwise comparisons.

**Structure of the paper**

The remainder of this paper is structured as follows. Section 2 is devoted to a description of the Smith-Waterman algorithm in the context of text comparison, and Section 3 to a discussion of how best to keep track of the significant similarities detected by the algorithm. In Section 4, we indicate how, in this context, the likely sparsity of the dynamic programming array can be exploited to improve both time and space efficiency. Section 5 presents the results of

some empirical investigations, and we summarise our results and conclusions in Section 6.

# 2 The Smith-Waterman algorithm in collusion detection

### Motivation and outline of the method

The Smith-Waterman algorithm [8] is a classical method of comparing two strings with a view to identifying highly similar sections within them. It is widely-used in finding good near-matches, or so-called local alignments, within biological sequences [3].

The basis of the method is a dynamic programming scheme, which we now describe. Throughout, we denote the lengths of the given strings $X$ and $Y$ by $m$ and $n$ respectively. If we imagine a portion $X'$ of string $X$ aligned with a portion $Y'$ of string $Y$, we wish to allocate a *score* that, in some sense, represents the 'goodness of fit' between $X'$ and $Y'$. Each matching symbol should make a positive contribution to that score, and each symbol that has to be inserted, deleted or substituted to transform $X'$ to $Y'$ should make a negative contribution.

Let $h$ be the (positive) contribution made by a symbol 'hit', $d$ the (negative) contribution made by a symbol insertion or deletion (an 'indel'), and $r$ the (negative) contribution made by replacing one symbol by another. A more general model is typically used in computational biology. Rather than a fixed positive score for a hit and a fixed negative score for a replacement, a scoring matrix is used, giving appropriate scores for all possible hits and replacements. In addition, a more complex model for 'gaps' is often used, the so-called affine gap model, which imposes a constant cost for opening a gap in the alignment and a different constant cost for extending that gap. Our methodology could be extended to that more general model, but in our context it is not clear how we would construct a scoring matrix, nor what appropriate gap costs would be, so we describe only the simpler model. Even for this simpler model, it is not immediately clear what the relative values of $h$, $d$ and $r$ should be; the most obvious option is to choose $h = d = r = 1$, and these values have been shown to work effectively in practice. (In much of the following discussion, and all of our later examples and empirical work, we will assume that $h = d = r = 1$.)

For example, if $X' =$ `abcbadbca` and $Y' =$ `abbdbda`, an optimal alignment has 6 hits, 2 indels, and 1 replacement, as shown in Figure 1, and we obtain a score of $6h - 2d - r$, or $6 - 2 - 1 = 3$ in the case where $h = d = r = 1$.

Our objective is to find significant near-matches between substrings of $X$ and $Y$, where 'significant' is defined in terms of some suitably chosen threshold score. What this threshold score should be depends on the context, and on the chosen values of $h$, $d$ and $r$. For example, a comparison of the texts of the Jane

Austen novels *Pride and Prejudice* and *Emma* (with $h = d = r = 1$) revealed only one near-match with a score of 9 or more (in fact, it is an exact match of the nine-word phrase 'I am sure I do not know who is'), indicating that, in the case of plain text, quite a low threshold may be appropriate. On the other hand, in the much more restrictive domain of program code, a substantially higher threshold is likely to be necessary.

The cumulative score of significant near-matches might be taken as an appropriate measure of overall similarity of the two strings in our application context, though more generally, any two strings containing at least one significant near-match might be regarded as worthy of further investigation. But, as we shall see, there are non-trivial issues to be considered when it comes to identifying an appropriate set of 'independent' significant near-matches.

**The algorithm in detail**

To formulate the classical Smith-Waterman dynamic programming scheme, we define $S_{ij}$ to be the maximum score obtainable by aligning a substring of $X$ ending at position $i$ with a substring of $Y$ ending at position $j$.

The standard recurrence relation for $S_{ij}$ is

$$S_{ij} = \begin{cases} S_{i-1,j-1} + h \text{ if } X(i) = Y(j) \\ \max(0, S_{i-1,j} - d, S_{i,j-1} - d, S_{i-1,j-1} - r) \text{ otherwise,} \end{cases}$$

subject to the initial conditions

$$S_{i,0} = S_{0,j} = 0 \text{ for all } i, j.$$

Notice that a negative score is impossible, since aligning the empty substrings ending at positions $i$ and $j$ yields a score of zero.

Application of this recurrence relation leads to a dynamic programming algorithm enabling the computation of the elements of the array $S$, for example in row by row order.

As is standard with dynamic programming schemes of this kind, we can use the idea of a traceback path to construct an optimal local alignment ending at position $i$ in $X$ and position $j$ in $Y$. For a given cell $(i, j)$ we define a *parent* cell as follows:

- if $S_{ij} = 0$ then $(i, j)$ has no parent;
- if $X(i) = Y(j)$ then $(i, j)$ has the parent $(i - 1, j - 1)$;
- in addition $(i, j)$ has as a parent any cell $(p, q) \in \{(i-1, j), (i, j-1)\}$ such that $S_{ij} = S_{pq} - d$, and/or cell $(i - 1, j - 1)$ if $S_{ij} = S_{i-1,j-1} - r$.

| a | b | c | b | a | d | b | c | a |
|---|---|---|---|---|---|---|---|---|
| a | b | - | b | - | d | b | d | a |

Figure 1: An optimal alignment of two substrings

So each cell containing a non-zero value has at least one parent, and may have as many as three.

For any cell $(i, j)$ for which $S_{ij} > 0$, we define a *traceback path* in the array to be any path obtained by starting from cell $(i, j)$, stepping successively from a cell to a parent cell, and terminating as soon as the next step in the path would reach a cell with a zero entry. Let $O_{ij} = (x_{ij}, y_{ij})$ be the final cell in a traceback path for cell $(i, j)$ (so that the value in this cell is necessarily equal to $h$). We call $O_{ij}$ an *origin* for cell $(i, j)$. Because parents need not be unique, a cell may have more than one traceback path and more than one origin. Any origin for cell $(i, j)$ specifies the starting points in $X$ and $Y$ respectively of a highest scoring local alignment ending at $X(i)$ and $Y(j)$.

**Identifying significant matches**

Suppose that we specify a threshold value $v$, and deem that any table entry $S_{ij} \geq v$ represents a potentially significant local alignment or match. It seems natural that, for a genuinely significant match, we should require a local maximality property, expressed in terms of the pre- and post-domination properties that we now introduce.

Let $(i, j)$ be a cell with a traceback path that passes through a cell $(p, q)$ with $S_{pq} \geq S_{ij}$. Cells $(i, j)$ and $(p, q)$ share an origin, say $(k, l)$, and the substrings $X(k \ldots p)$, $Y(l \ldots q)$ are prefixes of $X(k \ldots i)$ and $Y(l \ldots j)$ respectively, and give an alignment with a score that is at least as high. In such a case we say that cell $(i, j)$ is *pre-dominated* by cell $(p, q)$. (Alternatively, we could require that $S_{pq} > S_{ij}$; it is a moot point as to whether a longer alignment is more or less significant than a shorter alignment with the same score.)

Moreover, we say that a cell $(i, j)$ that is on a traceback path for cell $(p, q)$ where $S_{pq} > S_{ij}$, is *post-dominated*. Clearly the local alignment that is represented by a post-dominated cell can be extended to an alignment with a higher score.

Suppose, as seems reasonable, that any entry $\geq v$ in the Smith-Waterman array is viewed as representing a significant local match provided it is neither pre-dominated nor post-dominated.

We claim that this notion of a significant local match can be improved if we use a certain variant of the classical algorithm. In fact, it has recently been observed in the bioinformatics context [2, 11] that the Smith-Waterman algorithm can produce local alignments that are unsatisfactory because of intermediate sections that score very poorly, and this is the issue that we address here.

We begin with an example. Consider the two texts:

1. *Here is an example that very clearly illustrates how the Smith-Waterman algorithm can be adapted to deal effectively with the detection of collusion.*

2. *Here is an interesting example that very clearly illustrates how we can adapt the algorithm of Smith-Waterman to deal with the detection of collusion.*

The token sequences for these two texts can be represented as follows, where

we use the lower case letters as tokens (`a` = 'here', `b` = 'is', etc., and we represent 'Smith-Waterman' by the single token `k`).

1. `X = a b c d e f g h i j k l m n o p q r s j t u v`
2. `Y = a b c x d e f g h i y m z j l u k p q s j t u v`

Suppose that the rows of the dynamic programming array represent $X$ and the columns represent $Y$, and suppose that we set the threshold for a significant match to be 5. Figure 2 shows the resulting dynamic programming table, with only the non-zero entries displayed. (Here, as in all of our examples, we use $h = d = r = 1$.)

| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | a | b | c | x | d | e | f | g | h | i | y | m | z | j | l | u | k | p | q | s | j | t | u | v |
| 1 | a | **1** | | | | | | | | | | | | | | | | | | | | | | | |
| 2 | b | | **2** | 1 | | | | | | | | | | | | | | | | | | | | | |
| 3 | c | | 1 | **3** | **2** | 1 | | | | | | | | | | | | | | | | | | | |
| 4 | d | | | 2 | 2 | **3** | 2 | 1 | | | | | | | | | | | | | | | | | |
| 5 | e | | | 1 | 1 | 2 | **4** | 3 | 2 | 1 | | | | | | | | | | | | | | | |
| 6 | f | | | | | 1 | 3 | **5** | 4 | 3 | 2 | 1 | | | | | | | | | | | | | |
| 7 | g | | | | | | 2 | 4 | **6** | 5 | 4 | 3 | 2 | 1 | | | | | | | | | | | |
| 8 | h | | | | | | 1 | 3 | 5 | **7** | 6 | 5 | 4 | 3 | 2 | 1 | | | | | | | | | |
| 9 | i | | | | | | | 2 | 4 | 6 | **8** | **7** | **6** | **5** | 4 | 3 | 2 | 1 | | | | | | | |
| 0 | j | | | | | | | 1 | 3 | 5 | 7 | 7 | 6 | 5 | **6** | 5 | 4 | 3 | 2 | 1 | | 1 | | | |
| 1 | k | | | | | | | | 2 | 4 | 6 | 6 | 6 | 5 | **5** | 5 | 4 | 5 | 4 | 3 | 2 | 1 | | | |
| 2 | l | | | | | | | | 1 | 3 | 5 | 5 | 5 | 5 | 4 | **6** | 5 | 4 | 4 | 3 | 2 | 1 | | | |
| 3 | m | | | | | | | | | 2 | 4 | 4 | 6 | 5 | 4 | 5 | **5** | 4 | 3 | 3 | 2 | 1 | | | |
| 4 | n | | | | | | | | | 1 | 3 | 3 | 5 | 5 | 4 | 4 | 4 | **4** | 3 | 2 | 2 | 1 | | | |
| 5 | o | | | | | | | | | | 2 | 2 | 4 | 4 | 4 | 3 | 3 | **3** | 3 | 2 | 1 | 1 | | | |
| 6 | p | | | | | | | | | | 1 | 1 | 3 | 3 | 3 | 3 | 2 | 2 | **4** | 3 | 2 | 1 | | | |
| 7 | q | | | | | | | | | | | | 2 | 2 | 2 | 2 | 2 | 1 | 3 | **5** | 4 | 3 | 2 | 1 | |
| 8 | r | | | | | | | | | | | | 1 | 1 | 1 | 1 | 1 | 1 | 2 | **4** | 4 | 3 | 2 | 1 | |
| 9 | s | | | | | | | | | | | | | | | | | | 1 | 3 | **5** | 4 | 3 | 2 | 1 |
| 0 | j | | | | | | | | | | | | | | 1 | | | | | 2 | 4 | **6** | 5 | 4 | 3 |
| 1 | t | | | | | | | | | | | | | | | | | | | 1 | 3 | 5 | **7** | 6 | 5 |
| 2 | u | | | | | | | | | | | | | | | | | | | | 2 | 4 | 6 | **8** | 7 |
| 3 | v | | | | | | | | | | | | | | | | | | | | 1 | 3 | 5 | 7 | **9** |

Figure 2: The non-zero values of $S(i, j)$ for strings $X$ and $Y$

On applying the algorithm, we find that the largest value in the table is 9, and this value appears in the bottom right-hand corner of the table, in position (23,24). This is the only entry in the table that is $\geq 5$ and is neither pre-dominated nor post-dominated. A traceback path from this position shown in bold in the figure, leads all the way to cell (1,1), showing that the corresponding

```
a b c - d e f g h i - - - j k l m n o p q r s j t u v
a b c x d e f g h i y m z j - l u k - p q - s j t u v
```

Figure 3: Optimal local (and global) alignment of strings $X$ and $Y$

```
a b c - d e f g h i          p q r s j t u v
a b c x d e f g h i          p q - s j t u v
```

Figure 4: Two local alignments of strings $X$ and $Y$

match is between X(1...23) and Y(1...24) aligned as shown in Figure 3. The score of 9 comprises 18 hits, 7 indels, and 2 replacements.

Hence if, as seems natural, we select the largest $S$ value as representing the 'best' local alignment between the two strings, we obtain the match consisting of the entire two strings, with a score of 9. However, if instead we choose two separate local alignments consisting of the match between X(1...9) and Y(1...10), with a score of 8, and the match between X(16...23) and Y(18...24) with a score of 6, we obtain an overall score of 14 — see the two local alignments displayed in Figure 4. We would argue that this score more accurately reflects the similarities between the two strings, and that the original score of 9 substantially underestimates the cumulative significant local matches. Clearly, more extreme examples of this phenomenon are possible.

**A revised version of the algorithm**

To avoid the situation illustrated by this example, we introduce the notion of a *cut-off*. Define $M_{ij}$ to be the largest $S$ value contained in any cell on a traceback path from cell $(i, j)$, excluding $S_{ij}$ itself, with $M_{ij} = 0$ if $S_{ij} = 0$. Henceforth we consider only the case $h = d = r = 1$, though the ideas extend quite easily to the more general case. The following theorem gives a recurrence relation for $M_{ij}$.

**Theorem 2.1** *For each $i, j$ ($1 \le i \le m, 1 \le j \le n$),*

$$M_{ij} = \begin{cases} 0 \text{ if } S_{ij} = 0 \\ \max(S_{i-1,j-1}, M_{i-1,j-1}) \text{ if } X(i) = Y(j) \\ \max(M_{pq}, S_{pq}) \text{ otherwise} \end{cases}$$

*where, in the third case, the maximum is taken over those cells $(p, q) \in \{(i-1, j), (i, j-1), (i-1, j-1)\}$ that are parents of $(i, j)$.*

**Proof** The first case is a consequence of the definition. In the case where $X(i) = Y(j)$, the $M_{ij}$ value is at least equal to $S_{i-1,j-1}$, since cell $(i-1, j-1)$ is on every traceback path from $(i, j)$. However, every cell that is on a traceback path from $(i-1, j-1)$ is also on a traceback path from $(i, j)$, so $M_{ij}$ is at least equal to $M_{i-1,j-1}$.

When $X(i) \neq Y(j)$, the cells on a traceback path from $(i, j)$ are precisely the parent cells $(p, q)$ of $(i, j)$ together with all cells on traceback paths from these cells. Hence $M_{ij}$ is the maximum of the values contained in the parent cells and the values on any traceback path from these cells, from which the stated result follows. □

Suppose that $M_{ij} - S_{ij} \geq v$, where $v$ is our chosen threshold value, and that $(p, q)$ is a cell on a traceback path from $(i, j)$ such that $S_{pq} = M_{ij}$. Suppose that some cell $(k, l)$, with $k > i$ and $l > j$, for which $S_{kl} \geq v$, has a traceback path that passes through $(i, j)$, and therefore also one that passes through $(p, q)$. Then if $S_{kl} < S_{pq}$, cell $(k, l)$ is pre-dominated and therefore is not a candidate for a significant local alignment. On the other hand, if $S_{kl} \geq S_{pq}$, then the gain in value as we travel (in reverse direction) along the traceback path from $(i, j)$ to $(k, l)$ is $S_{kl} - S_{ij} \geq S_{pq} - S_{ij} = M_{ij} - S_{ij} \geq v$. It follows that, if we were to reset the value of $S_{ij}$ to zero — or in other words, conceptually discard the prefixes $X(1 \ldots i)$ and $Y(1 \ldots j)$, cell $(k, l)$ would still contain a value at least equal to the threshold.

This illustrates the concept of a cut-off. If, on evaluating $S_{ij}$, we find that $M_{ij} - S_{ij} \geq v$, then we simply reset the $S_{ij}$ value to zero (and of course this implies that the $M_{ij}$ value should also be reset to zero). For our example, the revised $S_{ij}$ values are shown in Figure 5. We now find that we have two values achieving the threshold that are neither pre- nor post-dominated, namely the value of 8 in cell $(9, 10)$ and the value of 6 in cell $(23, 24)$. The local alignments corresponding to these are precisely those shown in Figure 4.

# 3 Application of the algorithm

The question arises as to how we can most effectively and efficiently use our variant of the Smith-Waterman algorithm to obtain and present an appropriate set of matches between two source documents.

Firstly, we have found that the most useful form of output from the algorithm consists of annotated versions of the original two documents, with matching sections suitably highlighted — see Figure 10 later in the paper for an example of such a display. Hence we would not want any of the symbols that contribute to one match to contribute to another — in other words we would want to discount overlapping matches, since any overlap between matches would cause confusion in such a display. In reality significant overlaps between matches are unlikely, since their existence would imply a degree of repetition within one or both of the original documents, but we certainly cannot discount the possibility. Very small overlaps may well arise commonly, and our methodology must be able to deal with this.

In biological applications, overlapping matches are indeed of interest. The norm in that domain is to discount only matches that share a common cell in their traceback paths, and time and space efficient methods have been developed to generate significant matches under this constraint [4, 5, 10].

| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | a | b | c | x | d | e | f | g | h | i | y | m | z | j | l | u | k | p | q | s | j | t | u | v |
| 1 | a | **1** | | | | | | | | | | | | | | | | | | | | | | | |
| 2 | b | | **2** | 1 | | | | | | | | | | | | | | | | | | | | | |
| 3 | c | | 1 | **3** | **2** | 1 | | | | | | | | | | | | | | | | | | | |
| 4 | d | | | 2 | 2 | **3** | 2 | 1 | | | | | | | | | | | | | | | | | |
| 5 | e | | | 1 | 1 | 2 | **4** | 3 | 2 | 1 | | | | | | | | | | | | | | | |
| 6 | f | | | | | 1 | 3 | **5** | 4 | 3 | 2 | 1 | | | | | | | | | | | | | |
| 7 | g | | | | | | 2 | 4 | **6** | 5 | 4 | 3 | 2 | | | | | | | | | | | | |
| 8 | h | | | | | | 1 | 3 | 5 | **7** | 6 | 5 | 4 | 3 | | | | | | | | | | | |
| 9 | i | | | | | | | 2 | 4 | 6 | **8** | 7 | 6 | 5 | 4 | | | | | | | | | | |
| 0 | j | | | | | | | 1 | 3 | 5 | 7 | 7 | 6 | 5 | 6 | 5 | 4 | | | | | 1 | | | |
| 1 | k | | | | | | | | 2 | 4 | 6 | 6 | 6 | 5 | 5 | 5 | 4 | 5 | 4 | | | | | | |
| 2 | l | | | | | | | | | 3 | 5 | 5 | 5 | 5 | 4 | 6 | 5 | 4 | 4 | | | | | | |
| 3 | m | | | | | | | | | | 4 | 4 | 6 | 5 | 4 | 5 | 5 | 4 | | | | | | | |
| 4 | n | | | | | | | | | | | | 5 | 5 | 4 | 4 | 4 | 4 | | | | | | | |
| 5 | o | | | | | | | | | | | | 4 | 4 | 4 | | | | | | | | | | |
| 6 | p | | | | | | | | | | | | | | | | | | **1** | | | | | | |
| 7 | q | | | | | | | | | | | | | | | | | | | **2** | 1 | | | | |
| 8 | r | | | | | | | | | | | | | | | | | | | **1** | 1 | | | | |
| 9 | s | | | | | | | | | | | | | | | | | | | | **2** | 1 | | | |
| 0 | j | | | | | | | | | | | | | | 1 | | | | | | 1 | **3** | 2 | 1 | |
| 1 | t | | | | | | | | | | | | | | | | | | | | | 2 | **4** | 3 | 2 |
| 2 | u | | | | | | | | | | | | | | | | 1 | | | | | 1 | 3 | **5** | 4 |
| 3 | v | | | | | | | | | | | | | | | | | | | | | | 2 | 4 | **6** |

Figure 5: The non-zero revised values of $S(i, j)$ for strings $X$ and $Y$

In our setting, we might therefore seek the set of non-overlapping matches, all of which achieve the prescribed threshold score, and which have the maximum possible combined score. Unfortunately, this problem is known to be NP-hard [9]. We therefore adopt an alternative strategy which, in practice, is likely to yield at least a very close approximation to this optimum solution. Our strategy is a greedy one, whereby we first select a match corresponding to the largest $S$ value in the table. (Of course there may be more than one entry with the largest score, and for any one such entry, more than one origin and therefore more than one match — we break any such tie arbitrarily.)

Our strategy will be to discount any match that overlaps a match already chosen, but we must recognise that some part of such an overlapping match may constitute a non-overlapping match that achieves the threshold score, so we cannot merely discard overlapping matches without allowing for this possibility. The simplest way of doing this involves successive re-computation of all of the parts of the $S$ table not already accounted for by chosen matches. This can be conveniently achieved by replacing the substrings that contribute the

match by single dummy characters, and then forcing the entries in the row and column of the table corresponding to these dummy characters to be zero. This is summarised in the Algorithm of Figure 6.

**loop**
    compute the table of $S$ values;
**exit when** maximum $S$ value less than threshold;
    select match represented by largest $S$ value;
    replace matching substrings by a fixed row and column of zeros;
**end loop**;

Figure 6: Naive algorithm to find non-overlapping matches

We seek to improve on this naive algorithm by maintaining a *candidate set* during the computation of the table, selecting matches from this candidate set, and recomputing as little as possible, and only when forced to because of overlaps.

**Identifying significant matches — maintaining a candidate set**

As we generate the tables of $S$ and $M$ values, we need to decide, for each value $S_{ij}$ generated, whether it should be considered to represent a (potentially) significant match. The first requirement is that $S_{ij} > v$, where $v$ is the specified threshold value, but of course this is not a sufficient condition. Any cell $(i, j)$ for which $S_{ij} \leq M_{ij}$ is pre-dominated, and therefore is not a candidate, and furthermore this condition can be checked immediately since the $S$ and $M$ values are calculated essentially in parallel.

Moreover, no cell $(i, j)$ that is post-dominated should be regarded as a candidate, but we will not know until some time later, during the application of the dynamic programming algorithm, whether this is the case. Rather than develop a strategy for discarding post-dominated candidates at the earliest opportunity, we will merely add them to the candidate set, and allow them to be pruned later, along with overlapping candidates, once a candidate with maximum $S$ value has been selected.

On processing cell $(i, j)$ during the execution of the algorithm, if we find that $S_{ij} \geq v$ and that the cell is not pre-dominated, then we add cell $(i, j)$ to this candidate set. The candidate set can be conveniently represented, say, as a linked list.

**Overlapping candidates**

The final candidate set contains all cells that represent matches of value at least

equal to the threshold which are not pre-dominated. However, this may well include

- pairs $(i, j)$ and $(k, l)$ for which the corresponding substrings $X(x_{ij} \ldots i)$ and $X(x_{kl} \ldots k)$ of X, and/or the corresponding substrings $Y(y_{ij} \ldots j)$ and $Y(y_{kl} \ldots l)$ of Y overlap, and
- pairs $(i, j)$ and $(k, l)$ such that $(i, j)$ is post-dominated by $(k, l)$.

Here $(x_{ij}, y_{ij}) = O_{ij}$ is an origin for cell $(i, j)$.

It is traightforward to show that computation of an origin for each position $(i, j)$ may be achieved as follows:

$$x_{ij} = \begin{cases} x_{i-1,j-1} \text{ if } X(i) = Y(j), \text{ else} \\ i + 1, \text{ if } S_{ij} = 0, \text{ else} \\ x_{i-1,j} \text{ if } S_{ij} = S_{i-1,j} - 1, \text{ else} \\ x_{i,j-1} \text{ if } S_{ij} = S_{i,j-1} - 1, \text{ else} \\ x_{i-1,j-1} \text{ if } S_{ij} = S_{i-1,j-1} - 1, \end{cases}$$

and

$$y_{ij} = \begin{cases} y_{i-1,j-1} \text{ if } X(i) = Y(j), \text{ else} \\ j + 1, \text{ if } S_{ij} = 0, \text{ else} \\ y_{i-1,j} \text{ if } S_{ij} = S_{i-1,j} - 1, \text{ else} \\ y_{i,j-1} \text{ if } S_{ij} = S_{i,j-1} - 1, \text{ else} \\ y_{i-1,j-1} \text{ if } S_{ij} = S_{i-1,j-1} - 1. \end{cases}$$

Note that these particular recurrence relations give priority to a vertical step, then a horizontal step, and then a diagonal step, in the traceback path, whenever there is a choice. Other versions giving a different order of priority are equally valid, and in some case will give a different origin. This can be seen by a comparison of the strings `ababcd` and `babacd`.

Knowing the origin for each cell means that it is straightforward to select a largest valued entry from the candidate set, and to scan that set to identify and delete any overlapping candidates and any candidates post-dominated by it. In fact, it turns out that handling post-dominated candidates is straightforward, as they can be regarded merely as a special case of overlapping candidates.

However, as indicated earlier, we must allow for the possibility that some portion of an overlapping candidate, or some entry dominated by an overlapping candidate, may represent a genuine non-overlapping significant match. For example, consider the following strings:

1. `U = x y z a b c d b c e f`
2. `V = a b c d e f g h x y z a b c`

The table of $S$ values is shown in Figure 7.

Let us suppose that the threshold value is 3. Then on completion of the computation of the table, the candidate set contains the cells $(6, 14)$ with a value of 6, and $(7, 4)$, with a value of 4. In particular, cell $(11, 6)$ is pre-dominated

| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | a | b | c | d | e | f | g | h | x | y | z | a | b | c |
| 1 | x | | | | | | | | | 1 | | | | | |
| 2 | y | | | | | | | | | | 2 | 1 | | | |
| 3 | z | | | | | | | | | | 1 | 3 | 2 | 1 | |
| 4 | a | 1 | | | | | | | | | | 2 | 4 | 3 | 2 |
| 5 | b | | 2 | 1 | | | | | | | | 1 | 3 | 5 | 4 |
| 6 | c | | 1 | 3 | 2 | 1 | | | | | | | 2 | 4 | 6 |
| 7 | d | | | 2 | 4 | 3 | 2 | 1 | | | | | 1 | 3 | 5 |
| 8 | b | | 1 | 1 | 3 | 3 | 2 | 1 | | | | | | 2 | 4 |
| 9 | c | | | 2 | 2 | 2 | 2 | 1 | | | | | | 1 | 3 |
| 0 | e | | | 1 | 1 | 3 | 2 | 1 | | | | | | | 2 |
| 1 | f | | | | | 2 | 4 | 3 | 2 | 1 | | | | | 1 |

Figure 7: The non-zero values of $S(i, j)$ for strings $U$ and $V$

by $(7, 4)$ and so is not recorded. The best match, with value 6, is between $U(1 \ldots 6)$ and $V(9 \ldots 14)$ (and is, in fact, an exact match). The only other match represented in the candidate set, and not post-dominated by this maximum candidtae, has value 4 and is between $U(4 \ldots 7)$ and $V(1 \ldots 4)$ (again an exact match). This second match overlaps the first one (in $U$) and so must be pruned from the candidate set. However, there is a non-overlapping match of value 3 between $U(8 \ldots 11)$ and $V(2 \ldots 6)$, but cell $(11, 6)$ is pre-dominated by $(7, 4)$, and so is not in the candidate set.

To address this difficulty, we must allow for re-computation of parts of the table affected by the deletion of an overlapping match. Once such recomputation has taken place, and the candidate set has been suitably updated, the whole process will be iterated until the updated candidate set is empty.

Suppose that at some point during this process, the current maximum-valued candidate, say the $k$th significant match selected, represents cell $(i, j)$, with origin $(x_{ij}, y_{ij})$. So the matching substrings are $X_k = X(x_{ij} \ldots i)$ and $Y_k = Y(y_{ij} \ldots j)$. Then the characters in string $X$ not so far matched constitute $k + 1$ substrings (some of which may be empty) separated by matching substrings, and likewise for string $Y$ — let us refer to these as the *unmatched segments*. For recomputation purposes, we need consider only the table entries that correspond to the unmatched segments of $X$ on either side of $X_k$ or the unmatched segments of $Y$ on either side of $Y_k$ — see Figure 8 for an illustration in which the relevant rectangles are indicated by asterisks. This is because entries in any other rectangle of the array could not have a changed value on recomputation, and if pre-dominated in the recomputed table they must have been so in the original.

Remaining rectangles are of two kinds, namely
(i) those whose upper boundary is row $i + 1$ or whose left voundary is column $j + 1$;

(ii) those whose lower boundary is row $x_{ij}-1$ or whose right boundary is column $y_{ij}-1$.
In case (i), recomputation of the rectangle $R$ is necessary only if a candidate within $R$ was pruned from the candidate set. In case (ii) the corresponding condition is that the candidate's origin is within $R$. It is not hard to see that neither of these conditions can be satisfied by a post-dominated candidate, so that pruning such candidates as a special case of overlaps never gives rise to re-computation.

Clearly a maximum of $4k+1$ rectangles of entries in the array need be recomputed. We call such a portion of the array requiring re-computation an *essential* rectangle.

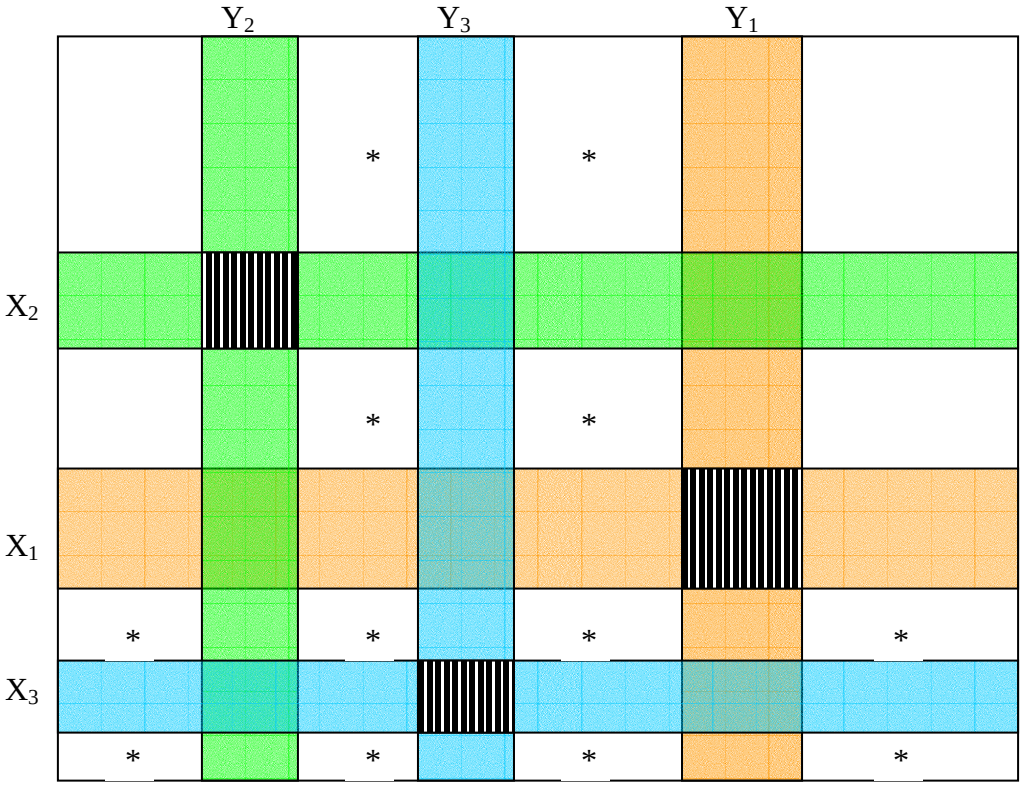


Figure 8: Rectangles corresponding to unmatched segments.

The sequence of unmatched segments in each of $X$ and $Y$ can be maintained in two lists, ordered according to position within the original string, and this enables easy identification of the essential rectangles. The recomputation for each such rectangle proceeds in an identical way to the main dynamic programming algorithm, with the candidate set being updated appropriately.

The complete algorithm at this point is summarised in Figure 9, in which we use $C$ to stand for the candidate set.

```
S_i0 := 0 for all i;
S_0j := 0 for all j;
C := ∅;
for i in 1...m loop
    for j in 1...n loop
        Calculate S_ij;
        Calculate M_ij;
        if M_ij − S_ij ≥ v then
            S_ij := 0;
            M_ij := 0;
        end if;
        if S_ij ≥ v and S_ij > M_ij then
            C := C ∪ {(i, j)};
        end if;
    end loop;
end loop;
while C is non-empty loop
    Select a largest valued match M from C;
    Add M to set of significant matches;
    Prune from C candidates that represent matches that overlap M;
    Identify essential rectangles of the table;
    Recompute essential rectangles, updating C as appropriate;
end loop;
```

Figure 9: Algorithm to compute significant local alignments

# 4 Space requirements

If we are to store all elements of the $S$ and $M$ tables simultaneously, then the space requirement for the algorithm is $\Omega(mn)$. This could be prohibitive in the case of long texts. We can apply the standard trick of retaining only the most recently evaluated entry in each column of each of these tables (assuming that our algorithm is proceeding row by row). This, of course, prevents us from following traceback paths to determine the precise alignment for each significant match.

However, in contrast to the biological setting, where knowledge of actual alignments is important, we would argue that in our context only the score and the extent of a match are required. Figure 10 gives a graphic illustration of the utility of such information. Human judgment must be used to evaluate the

real significance of a match, and it is not clear that this would be aided by precise knowledge of the alignment itself. In fact, the time-efficient linear-space approach of Huang and Miller [5] could be adapted to the present domain, and would thereby allow precise alignments to be generated without affecting the asymptotic worst-case complexity, but we do not pursue that avenue here.

**Exploiting sparsity**

The worst-case time complexity of an algorithm that computes all of the entries in the $S$ table cannot be better than $O(mn)$. However, in practical cases, we can expect a large proportion of the entries in the $S$ table to be zero, and we can exploit this sparsity to significantly speed up the algorithm in most cases. Such speed-up would not be expected in the traditional domain of the Smith-Waterman algorithm because of the small alphabet sizes in molecular biology (4 in the case of DNA and around 20 in the case of protein sequences). But for sequences derived from plain English text, we can expect an alphabet size of several thousand and a relatively small number of pairwise matches between tokens in the two sequences (see Section 5 below),

To exploit this sparsity, instead of recording the most recently computed entry in every column of the table, we record only those that are non-zero, each entry being accompanied by the index of the column that it comes from (and the corresponding $M$ value and origin). To achieve this, we have to identify, at each step, the next column of the current row that contains a non-zero entry, and compute that entry from the available information.

A non-zero entry in column $j$ of the current row, say row $i$, may arise in four ways, namely

- from a match between the characters $X(i)$ and $Y(j)$;
- from an entry with value $> 1$ in column $j$ of the previous row;
- from an entry with value $> 1$ in column $j - 1$ of the previous row.
- from an entry with value $> 1$ in column $j - 1$ of the the current row.

Hence each of these possibilities must be considered when deciding what the next relevant value of $j$ should be.

With careful implementation, this strategy enables the computation of all of the $s$ non-zero entries in the table to be carried out in $O(s + n \log n)$ time. This involves, among other things, setting up, for each token in string $X$ a *match-list*, a list of the positions in string $Y$ where this token occurs, thereby providing an efficient means of dealing with the first of the three criteria listed above. One way of achieving this efficiently is to sort each token sequence into token order (keeping a record of the original positions in the sequence), generate the matchlist for each token in $X$ by means of a single scan of the sorted sequence $Y$, and then restore the original sequence order of both $X$ and $Y$. The complexity of match-list construction is then dominated by the sorting, and so is $O(n \log n)$ (assuming $m \leq n$).

An additional trick that will have the effect of substantially reducing the number of spurious non-zero entries in the table without significantly affecting the outcome is to use a stop-list, i.e., a list of common words ('a', 'and', 'in', 'the', etc.) which can be removed from the texts before (or during) the lexical analysis phase. A significant match is likely to remain significant even after this pre-processing, provided we lower the threshold somewhat, and the reduction in the number of non-zero table entries (as well as the reduction in length of the strings) should give a worthwhile speed-up to the algorithm. This is confirmed by the empirical evidence presented in the next section.

# 5 Empirical evidence

An initial empirical investigation of the effectiveness of our approach was focused on text files. We conducted tests of two kinds, namely

- comparisons of a batch of 120 essays where some significant collusion and/or plagiarism was known to have taken place;
- comparisons of several pairs of longer texts where only relatively minor similarities were expected.

We implemented two different programs. The first (standard) made no attempt to exploit sparsity, merely calculating each entry in the dynamic programming table, but maintaining only the most recently computed entry in each column of the table, while the second (refined) used the techniques described in Section 4 to compute only the non-zero elements. Both versions maintained a candidate set in the form of a linked list, and used the greedy strategy to select matches from this list. Any candidates post-dominated by, or overlapping with, a selected match would be pruned from the list, as described earlier, and appropriate sections of the table would be recomputed (although an option was provided to suppress any such recomputation). An option was also provided to employ a stop-list, which contained 27 of the commonest words in English.

**Pairwise comparison of a batch of text files**

For these comparisons, we set a threshold value of 10 for significant matches, and flagged all pairs of files that reached a score of 100 or more, but when using a stop-list we lowered these values to 7 and 70 respectively. We ran the programs on the first 30 of the files, then the first 60, and finally on all 120, to observe the growth in the run-times. The overall average length of the files was 2234 tokens, and we discovered that 1.14% of the elements across all of the dynamic programming tables were non-zero. (These figures were reduced to 1731 and 0.67%, respectively, when the stop-list was used.) The results are summarised in Table 2. The final two columns in this table report the times taken by the standard and refined programs (Algorithms S and R) in each case. (The times are in seconds for programs run on a 2.6GHz PC with 512 Mb of RAM.) The table also reports the average score across all pairs of files, and

the number of pairs reaching the threshold score of 100 (or 70 when a stop-list was used). For each batch of files, the two rows show the results without and with the use of a stop-list, respectively. In view of the small proportion

| Files | Ave. length | % non-zero | Ave. score | threshold | Alg S | Alg R |
|---|---|---|---|---|---|---|
| 30 | 2172 | 1.15 | 34.6 | 29/435 | 39.1 | 5.9 |
| | 1689 | 0.66 | 29.2 | 32/435 | 23.7 | 3.4 |
| 60 | 2229 | 1.16 | 25.6 | 67/1770 | 171 | 25.1 |
| | 1727 | 0.68 | 22.4 | 81/1770 | 103 | 14.4 |
| 120 | 2234 | 1.14 | 23.5 | 179/7140 | 695 | 101 |
| | 1731 | 0.67 | 21.1 | 234/1770 | 421 | 58.2 |

Table 2: Pairwise comparison of a batch of text files

of non-zero elements in the dynamic programming table, we would expect a substantial speed-up in moving from the standard to the refined version. Of course, the speed-up achieved is reduced somewhat by the preprocessing of the strings, and by the significantly more complex algorithm required to compute successive non-zero elements, as compared to just successive elements, in the table. Any manipulations of the candidate list are common to both approaches. The table shows that a speed-up by a factor of about 7 was obtained in all cases. This makes the refined version a much more practical proposition for the comparison of large batches of files. As we would expect, the running times of both algorithms grow quadratically with the number of files.

We also carried out the same test after switching off the re-computation phase following the detection and removal of overlapping matches. The effect was only marginal - a speed-up of less than 5% for both programs, a small reduction in the average score, but a reduction of one in the number of pairs of files reaching the threshold score. The conclusion here is that the potential loss of information in switching off recomputation is not justified by the small efficiency gain.

**Comparison of some longer text files**

In this test, we compared (a) the texts of the four gospels in the New Testament of the King James edition of the Bible, and (b) the texts of the Jane Austen novels *Emma* and *Pride and Prejudice*. The expectation was that there would be some significant similarities and matches in (a), but few if any in (b).

Table 3 summarises the results obtained in pairwise comparisons of the gospels. These confirm the known fact that the first three, *Matthew*, *Mark* and *Luke*, have commonalities not shared with the fourth, *John*. Figure 10 illustrates a section of *Matthew* and a section of *Luke* with some significant matches highlighted.

For each pair, the table shows the number of matches reaching the threshold (of 12 in this case), the total score, the score of the largest match, and the percentage of non-zero entries, as well as the times for Algorithms S and R.

7:22: Then Jesus answering said unto them, Go your way, and tell John what things ye have seen and heard; how that the blind see, **the lame walk, the lepers are cleansed, the deaf hear, the dead are raised, to the poor the gospel is preached.**

7:23: **And blessed is he, whosoever shall not be offended in me.**

7:24: **And** when the messengers of John were departed, he began to speak unto the people *concerning John, What went ye out into the wilderness for to see? A reed shaken with the wind?*

7:25: *But what went ye out for to see? A man clothed in soft raiment? Behold, they* which are gorgeously apparelled, and live delicately, are in kings' courts.

7:26: But what went ye out for to see? A prophet? Yea, I say unto you, and much more than a prophet.

7:27: This is he, of whom it is written, Behold, I send my messenger before thy face, which shall prepare thy way before thee.

7:28: For I say unto you, Among those that are born of women there is not a greater prophet than John the Baptist: but he that is least in the kingdom of God is greater than he.

11:5: The blind receive their sight, and **the lame walk, the lepers are cleansed, and the deaf hear, the dead are raised up, and the poor have the gospel preached to them.**

11:6: **And blessed is he, whosoever shall not be offended in me.**

11:7: **And** as they departed, Jesus began to say unto the multitudes *concerning John, What went ye out into the wilderness to see? A reed shaken with the wind?*

11:8: *But what went ye out for to see? A man clothed in soft raiment? behold, they* that wear soft clothing are in kings' houses.

11:9: But what went ye out for to see? A prophet? yea, I say unto you, and more than a prophet.

11:10: For this is he, of whom it is written, Behold, I send my messenger before thy face, which shall prepare thy way before thee.

11:11: Verily I say unto you, Among them that are born of women there hath not risen a greater than John the Baptist: notwithstanding he that is least in the kingdom of heaven is greater than he.

Figure 10: Some significant matches between *Matthew* and *Luke*

| Docs | lengths | matches | score | max | % non-zero | Alg S | Alg R |
|---|---|---|---|---|---|---|---|
| Mt / Mk | 23729 / 15188 | 93 | 2298 | 105 | 1.65% | 7.67 | 0.75 |
| | 17180 / 10716 | 88 | 1538 | 68 | 0.85% | 3.64 | 0.20 |
| Mt / L | 23729 / 25988 | 82 | 2103 | 112 | 1.54% | 12.9 | 1.08 |
| | 17180 / 18407 | 79 | 1457 | 72 | 0.82% | 6.31 | 0.31 |
| Mt / J | 23729 / 19127 | 2 | 24 | 12 | 1.34% | 9.08 | 0.67 |
| | 17180 / 13477 | 0 | 0 | 0 | 0.72% | 4.58 | 0.19 |
| Mk / L | 15188 / 25988 | 63 | 1286 | 49 | 1.74% | 8.25 | 0.75 |
| | 10716 / 18407 | 52 | 826 | 40 | 0.91% | 3.92 | 0.20 |
| Mk / J | 15188 / 19127 | 1 | 12 | 12 | 1.46% | 5.80 | 0.47 |
| | 10716 / 13477 | 0 | 0 | 0 | 0.74% | 2.86 | 0.13 |
| L / J | 25988 / 19127 | 0 | 0 | 0 | 1.38% | 9.92 | 0.75 |
| | 18407 / 13477 | 0 | 0 | 0 | 0.73% | 4.91 | 0.20 |

Table 3: Pairwise comparison of the four gospels

Again the two rows for each comparison give figures without and with the use of a stop-list, respectively (using a threshold of 9 in the latter case).

For these comparisons, Algorithm R achieves a speed-up by a factor of between 10 and 14 (towards the higher end of this range for the comparisons involving *John*), rising to around 20 if a stop-list is used. Again suppressing the re-evaluation of parts of the table after an overlapping match gave very little gain in time, but occasionally caused a genuine non-overlapping match to be lost.

Table 4 summarises the results obtained in comparing *Emma* with *Pride and Prejudice*. Here, we carried out comparisons of the first eighth of each text, then the first quarter, then the first half, and finally the complete texts, to measure the growth of the running times. We used a threshold of 8, so that we at least obtained a few significant matches.

| Lengths | matches | score | max | % non-zero | Alg S | Alg R |
|---|---|---|---|---|---|---|
| 20248 / 15188 | 0 | 0 | 0 | 0.85% | 7.15 | 0.28 |
| 14605 / 11156 | 0 | 0 | 0 | 0.47% | 3.09 | 0.09 |
| 40495 / 31182 | 0 | 0 | 0 | 0.85% | 29.9 | 1.16 |
| 29137 / 22272 | 0 | 0 | 0 | 0.47% | 12.8 | 0.34 |
| 80990 / 62364 | 2 | 17 | 9 | 0.85% | 119 | 4.66 |
| 58343 / 44628 | 4 | 24 | 7 | 0.47% | 61.9 | 1.34 |
| 161980 / 124729 | 7 | 57 | 9 | 0.85% | 479 | 18.5 |
| 116636 / 89582 | 11 | 67 | 7 | 0.47% | 247 | 5.31 |

Table 4: A comparison of *Emma* with *Pride and Prejudice*

The table indicates that the running time of the algorithms show essentially quadratic growth in the text lengths. Algorithm R gives a speed-up by a factor of about 26, rising to over 40 when a stop-list is used.

The obvious question that arises from these results is why the speed-up obtained by exploiting sparsity is so much greater in the comparison of a single pair of files as compared to comparison of all pairs of a batch of files. The answer lies in the pre-processing needed to build the list of occurrences in one file of each token that appears in the other. In comparing two texts, say of length $n$, this pre-processing is done just once, and its complexity is dominated by sorting operations — hence is $O(n \log n)$. Suppose we instead compare $k$ files each of length $2n/k$ (so that the total length of text involved is the same). Then the pre-processing phase has to be done before each pairwise comparison — hence $\binom{k}{2}$ times — and this time each is effectively a sort of $2n/k$ items, leading to $O(kn \log(2n/k))$ complexity. Empirical evidence showed that, in the batch experiments, up to half of the total execution time was taken up by this pre-processing, whereas in the comparisons of just two texts, the proportion was almost negligible.

# 6 Summary and conclusion

We have described how the classical Smith-Waterman algorithm can be transported from its usual domain of molecular biology to that of comparing text documents (or computer programs) for significant local similarities, and how this can be used in practice as a powerful tool for the detection of plagiarism or collusion.

Further efficiency gains in adapting the algorithm to this context would be valuable. The implementation of Algorithm R described here is more than adequate, in practice, for dealing with a moderate number of texts on an occasional basis, but it might not be fast enough to form the basis of, say, an open web-based collusion detection service. In the biological domain, Smith-Waterman is widely regarded as being too slow for large-scale application, and this has led to the development of faster heuristic methods such as BLAST [1] and FASTA [7]. It would be interesting to explore appropriate heuristics in the area of text comparison.

# References


[1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. A basic local alignment search tool. *Journal of Molecular Biology*, 215:403 – 410, 1990.

[2] A.N. Arslan, O. Egecioglu, and P.A. Pevzner. A new approach to sequence comparison: normalized sequence alignment. *Bioinformatics*, 17:327 – 337, 2001.

[3] D. Gusfield. *Algorithms on strings, trees, and sequences. Computer Science and Computational Biology.* Cambridge University Press, 1997.

[4] X. Huang, R.C. Hardison, and W. Miller. A space-efficient algorithm for local similarities. *CABIOS*, 6:373 – 381, 1990.

[5] X. Huang and W. Miller. A time-efficient linear-space local similarity algorithm. *Advances in Applied Mathematics*, 12:337 – 357, 1991.

[6] T. Lancaster. *Effective and Efficient Plagiarism Detection*. PhD thesis, School of Computing, Information Systems and Mathematics, South Bank University, 2003.

[7] D.J. Lipman and W.R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227:1435 – 1441, 1985.

[8] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195 – 197, 1981.

[9] V.Bafna, B.Narayanan, and R.Ravi. Non-overlapping local alignments (weighted independent sets of axis-parallel rectangles). *Discrete Applied Mathematics*, 71:41 – 53, 1996.

[10] M.S. Waterman and M. Eggert. A new algorithm for best subsequence alignments with application to tRNA-tRNA comparisons. *Journal of Molecular Biology*, 197:723 – 728, 1987.

[11] Z. Zhang, P. Berman, and W. Miller. Alignments without low scoring regions. *Journal of Computational Biology*, 5:197 – 210, 1998.