String Search

Graham A Stephen[1]

October 1992

Technical Report TR-92-gas-01

School of Electronic Engineering Science
University College of North Wales
Dean Street, Bangor, Gwynedd, UK LL57 1UT.
(email: graham@sees.bangor.ac.uk)

[1] Supported by SERC research grant GR/G51565

Contents

List of Figures

Chapter 1

Introduction

It could be said of me that in this book I have only made a bunch of other men's flowers, providing of my own only the string that ties them together.
— Michel de Montaigne (1533-1592), *Essais*.

This report is concerned with a string searching problem which has arisen in the development of textual search methods for a proposed information processing system. The aim of the report is not to attempt to provide a definitive solution for the problem, although some ideas towards this end are presented, but rather to review existing string searching algorithms germane to the processing system in general and, in particular, to the specific problem. The term *string searching* is used here in a generic sense to encompass techniques used to locate particular substrings, evaluate longest common subsequences between two strings, compute the edit distance between two strings, and so on.

First of all, the problem itself is described — initially in a rather loose, informal manner, and then a stricter definition is considered.

A discussion of the development of string searching algorithms is presented, giving an overview of their capabilities and performance. This is then followed by a more detailed examination of the principal algorithms themselves.

Finally, the implications of these algorithms on the original problem are considered.

Definitions of some of the terms and notation used in this area may be found in the appendices.

Chapter 2

The Problem

> Every string which has one end also has another end.
> — FINAGLE'S FIRST FUNDAMENTAL FINDING
> (Robert Anton Wilson, *Schrödinger's Cat Trilogy*).

Stated rather loosely, the problem is this — *find the patterns which occur more than once in a given sequence of symbols.* A bit vague perhaps, so some further qualification would appear to be in order.

First of all, the sequence is textual and one-dimensional, but what about its component symbols? Limiting ourselves to the printable ASCII characters excluding whitespaces (ASCII 33-126), we could take strings of these characters to be the unitary symbols, and a whitespace-separated list of these atomic symbols to be the sequence. This, however, leads to an infinite alphabet of symbols. It might therefore be simpler — it certainly seems more intuitive — to use a symbol alphabet comprising the previously mentioned ASCII characters together with the space character. The sequence of the problem now becomes a string over this (finite) alphabet.

The next point to consider is what exactly is meant by *pattern*. If the objective is to identify repeating patterns so that any inherent redundancy in the sequence may be exploited via some suitable coding scheme, in order to increase the representational efficiency, then it might be appropriate to interpret *pattern* as *substring*. If, on the other hand, however, the intention is to analyse the input sequence for syntactical structures, such as discontinuous dependencies, then it might be more appropriate to treat *patterns* as *subsequences*. In any event, since a substring is a special case of a subsequence, the latter shall, for generality, be adopted as the *pattern* of the problem in this context.

It might, at this stage, be informative to pause to consider the total number of patterns in a sequence of length n, say. Firstly, consider the case of substrings. There are $(n - k + 1)$ substrings of length k from the sequence. The total number is therefore given by summing over k, yielding

$$\sum_{k=1}^{n} (n-k+1) \quad = \quad \sum_{k=1}^{n} k \quad = \quad \frac{(n+1)n}{2} \quad = \quad \frac{(n^2+n)}{2} \tag{2.1}$$

This shows that the total number of substrings is $O\left(n^2\right)$.

Turning our attention to the case of subsequences, we find that the upper bound is exponential. This is simply derived as follows. For a given subsequence, each of the n symbols of the original sequence may be in either of two states — namely included in the subsequence, or excluded from it. It then follows that there will be a total of 2^n subsequences, including the empty string, ϵ.

The next point to consider is what exactly is meant by *recurrence* of a subsequence in the sequence. Taking the approach that the recurring instances of the pattern should be totally distinct, it shall be stipulated that only disjoint repeating patterns are to be considered. For example, in the sequence `A A B`, the subsequence `A B` would not be counted as recurring, since its instances are not disjoint — they both share the same '`B`'.

Finally, having shown that the number of subsequences in a sequence is $O\left(2^n\right)$, it is a simple matter to show that, in the worst case, there can be an exponential number of disjoint recurring subsequences. Consider, for example, the *square*, of length $2n$, x^2, where x comprises n distinct symbols. (For finite alphabets, n will be bounded by the cardinality of the alphabet.) Sequence x^2 will then have $2^n - 1$ recurring subsequences, not counting ϵ. There is, however, only one *maximal* recurring subsequence, namely x. This is illustrated in the following example.

x = `A B C` $\qquad x^2$ = `A B C A B C`

The disjoint recurring subsequences are `A`, `B`, `C`, `A B`, `A C`, `B C`, and `A B C`, which are all themselves subsequences of the maximal recurring subsequence, `A B C`.

It would therefore seem sensible to impose the restriction of searching only for maximal disjoint recurring subsequences.

Having come thus far in the examination of the requirements, we are now in a position to put forward a stricter formulation of the problem. Before doing so, however, some of the notational conventions employed in this document shall be explained.

Firstly, a string x, of length, $|x| = m$, is given by $x_1 x_2 \ldots x_m$, where x_i represents the i^{th} component symbol of x.

The substring of x, $x_i x_{i+1} \ldots x_j$, where $i, j \leqslant m$ and $j \geqslant i$, shall be denoted by $x(i,j)$. For the case where $i > j$, the reversed substring shall be denoted explicitly by $x_R(i,j)$.

Example

x = `trismegistus`
$|x| = 12$

$$x(7,10) = \texttt{gist}$$
$$x_R(7,4) = \texttt{gems}$$

Normally, x shall be used to denote a pattern, and y a text string of length n, where $m \leqslant n$.

Problem Definition

```
A = { ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8
      9 : ; < = > ? @ A B C D E F G H I J K L M N O P
      Q R S T U V W X Y Z [ \ ] ^ _ ` a b c d e f g h
      i j k l m n o p q r s t u v w x y z { | } ~       }
C = A ∪ {' '}
```

Alphabet Variant #1

Each symbol is a non-empty string, or ***word***, over A. The symbol alphabet, W, is therefore infinite and is given by the positive closure of A,

$$W = A^+ \tag{2.2}$$

A whitespace character shall be used as a separator for the component symbols of a sequence, or ***sentence***, of words. The set of all sentences shall be denoted by S_W. This would be equal to W^+ (i.e. W itself) were it not for the requirement that the words be separated by a symbol $\notin A$. In fact, S_W is given by the positive closure of C.

Alphabet Variant #2

Each symbol is a character from C. The symbol alphabet is therefore equal to C, and a sequence shall be a non-empty string over C. The set of all sequences, S_C, will be given by

$$S_C = C^+ \tag{2.3}$$

Problem

Given the sequence $x \in S$, identify and locate maximal disjoint recurring subsequences of x.

Chapter 3

Overview of the String Searching Algorithms

> Attempt the end, and never stand to doubt;
> Nothing's so hard, but search will find it out.
> — Robert Herrick (1591-1674), *Seek and Find.*

The major string searching problems which have been addressed in the literature are the *string matching* problem, the *string distance* problem, the *longest common subsequence* problem, the *approximate string matching* problems, and the *longest repeated substring* problem. Definitions of these problems together with overviews of the various approaches developed to solve them are presented below.

3.1 String Matching

The *String Matching Problem* is usually formulated as follows.

> Given pattern string x, with $|x| = m$, and text string y, with $|y| = n$, where $m, n > 0$ and $m \leqslant n$, if x occurs as a substring of y then determine the position within y of the first occurrence of x, i.e. return the least value of i such that $y(i, i+m-1) = x(1, m)$. The problem is sometimes extended such that the positions of all the occurrences of x within y are to be found. Algorithms satisfying the former requirement may be extended easily to satisfy the latter.

A recent survey and comparison of string matching algorithms, including the recent contributions by Sunday (1990), has been conducted by Pirklbauer (1992). A comprehensive comparison of the time and space complexities of various algorithms is also provided by Gonnet and Baeza-Yates (1991). Not to be neglected are the reviews by Aho (1990, 1980) and Sedgewick (1983).

The intuitive, or *Brute Force*, approach to the problem is simply to attempt to match x with substrings from y at successive positions in the text string. This requires the input text string to be buffered since backtracking in the text is required in the event of an unsuccessful match. In the worst case, this method runs in $O(mn)$ time, but for practical applications, such as searching English text, the expected performance is $O(m+n)$.

A theorem about two-way deterministic pushdown automata, derived by Cook (1972), led to the result that there exists an algorithm solving the string matching problem in $O(m+n)$ time in the worst case. In fact, Rivest (1977) demonstrated that, in the worst case, any string matching algorithm must examine at least $n-m+1$ symbols from the text string. This shows that no solution to the string matching problem may have behaviour sublinear in n in the worst case.

In 1970 Knuth learned of Cook's theorem and from it derived a string matching algorithm. This was then modified by Pratt so that its running time was independent of the size of the symbol alphabet. The resulting algorithm was also independently arrived at, without knowledge of Cook's theorem, by Morris in 1969 for application in a text editor so as to avoid having to backtrack in the text string. The operation of their algorithm (Knuth, Morris and Pratt, 1977) is described briefly below.

The basic idea behind the algorithm is to avoid backtracking in the text string in the event of a mismatch by taking advantage of known information. The text string is processed sequentially from left to right. When a substring match attempt fails at symbol x_j, say, the previous $j-1$ text symbols will be known since they will be given by $x(1, j-1)$. This fact can be exploited to determine how far to shift the pattern to the right for the next match attempt. An auxiliary *next* table containing this shift information is computed in $O(m)$ time from the pattern before searching the string.

The worst case performance occurs for Fibonacci pattern strings, for which the algorithm runs in $O(m+n)$ time. Although the method performs well for highly self-repetitive patterns and text, this situation occurs only rarely in actual applications. Thus, in practice the Knuth-Morris-Pratt algorithm is not likely to be significantly faster than the Brute Force approach. It does, however, have the advantage of never having to backtrack in the input text, obviating the need for involved buffering schemes when reading from sequential text streams for example.

Pirklbauer (1992) has discussed a variation of the Knuth-Morris-Pratt algorithm using right to left comparison of the pattern and text, as in the Boyer-Moore approach described below. An improved average case performance was reported, but at the cost of sacrificing the strictly sequential nature of processing the input text.

A much faster string searching method was discovered in 1974 by Boyer and Moore, and also independently by Gosper. The algorithm was later presented in a revised form (Boyer and Moore, 1977) taking into account suggestions from B. Kuiper, D.E. Knuth and R.W. Floyd.

The speed of the Boyer-Moore algorithm is achieved by disregarding portions of the text string which cannot possibly participate in a successful match. For a large alphabet and small pattern, the expected number of symbol comparisons is about n/m, and is $O(m+n)$ in the worst case. In this method, the pattern is scanned across the text string from left to right, but for each trial position the symbol comparisons between pattern and text are performed from right to left. Thus, the first comparison is between x_m and y_m. If y_m does not occur anywhere in x, then there can be no match for x starting anywhere in the first m symbols of y. The next potential pattern position would therefore be $m+1$, and the next comparison would be between x_m and y_{2m}. The actual pattern shift is determined by taking the larger of the values from two precomputed auxiliary tables. One of these, the *skip* table, is indexed by the offending text symbol causing the mismatch (the *occurrence* heuristic), whilst the other is based on the *next* table of the Knuth-Morris-Pratt algorithm (the *match* heuristic). The clarified pattern preprocessing for the Boyer-Moore algorithm described in the postscript of Knuth, Morris and Pratt (1977) is actually incorrect, but has been rectified by Rytter (1980).

The worst case number of symbol comparisons necessary to determine *all* the occurrences of the pattern in the text has been a matter of some interest. In these circumstances, the Boyer-Moore algorithm requires $O(n+rm)$ time, where r is the number of matches found. Knuth, Morris and Pratt (1977) presented a modified version of the Boyer-Moore algorithm requiring at most $6n$ comparisons for the case where the pattern does not occur in the text. Guibas and Odlyzko (1977) improved this figure to $4n$, and conjectured a tighter bound of $2n$. Galil (1979) also put forward a variant of the algorithm, which was improved upon by Apostolico and Giancarlo (1986). Their version requires at most $2n$ symbol comparisons, regardless of the number of occurrences of the pattern in the text. This is at the cost of only slightly increased pattern preprocessing overheads, which are still linear in m. These methods do not, however, provide any significant improvement in the average case.

Horspool (1980) has shown that a simplified form of the Boyer-Moore algorithm, employing only a single auxiliary table — indexed by the mismatching text symbols, results in a performance comparable to that of the original version. In fact, Baeza-Yates (1989a) has observed that, based on both analytical and empirical studies, of the known algorithms the Boyer-Moore-Horspool method is the best in terms of average case performance for nearly all pattern lengths and alphabet sizes.

The average case performance of the Boyer-Moore-Horspool and the Knuth-Morris-Pratt algorithms, together with the Brute Force approach, has been analysed by Baeza-Yates (1989b). His analysis was for the case of finding *all* the pattern occurrences, and was based on random strings (i.e. concatenations of symbols chosen independently from the alphabet with a uniform probability distribution) for the pattern and text. Empirically validated tight bounds were derived for the expected number of symbol comparisons involved in the search methods. A hybrid algorithm, combining the Boyer-Moore and Knuth-Morris-Pratt approaches, was proposed. In practice, the hybrid performed slightly better than the Boyer-Moore-Horspool method for searches of English text.

For the Boyer-Moore approach, the average running time improves with larger alphabets or longer patterns. Improved performance can therefore be obtained by increasing the size of the alphabet. This technique has been considered by Knuth, Morris and Pratt (1977) and by Schaback (1988). Baeza-Yates (1989a) has proposed a simple alphabet transformation which leads to a practical implementation of a Boyer-Moore-Horspool variant. Essentially, this involves grouping k symbols of the pattern together as a *supersymbol.* This reduces the pattern size, $|x|$, to $m - k + 1$ and increases the size of the alphabet to $|C|^k$, where C is the original alphabet. This approach has been shown to be of practical use for small alphabets or large patterns, and experimental increases in speed of 50% have been reported (e.g. for $k = 2$, $|C| = 8$ and random text).

Baeza-Yates (1989a) also put forward suggestions for improving the efficiency of searching non-uniformly distributed texts, such as English. He considered the optimal ordering of pattern symbol comparisons, based on their probability distribution within the text. This may be obtained either by preprocessing the text or from *a priori* information about the text source. Rarer symbols could, for example, be compared before more frequent ones, thereby increasing the probability of an early detection of a mismatch.

Sunday (1990) has suggested using the symbol in the text immediately *following* the one that caused a mismatch to address the occurrence heuristic table of the Boyer-Moore algorithm. Using this approach, he developed three variations, each differing in the manner in which the order of the symbol comparisons between the pattern and the current text substring is determined. In the *Quick Search* algorithm, the comparisons are performed from left to right. The *Maximal Shift* algorithm orders the comparisons such that the distance to the next pattern position in the event of a mismatch is maximised. Finally, the *Optimal Mismatch* algorithm uses the idea, discussed by Beaza-Yates, as mentioned above, to compare statistically rarer symbols first. Smith (1991) has developed an adaptive version of Sunday's Optimal Mismatch technique with the advantage that it assumes no prior knowledge of the given text.

A family of algorithms based on the general structure of an optimised implementation of Boyer and Moore's (1977) method, involving a fast skip loop, has been investigated by Hume and Sunday (1991). Two new variants conforming to this structure have been developed — the *Tuned Boyer-Moore* and the *Least Cost* algorithms.

An alternative approach to the string matching problem involves the use of hashing functions. Such a method to test for the occurrence of a pattern within a text string, but not to find its location, was proposed by Harrison (1971). This technique is based on the comparison of string *signatures* and requires the text to be preprocessed. It would therefore be applicable to the searching of large static text files, for example.

Karp and Rabin (1987) also put forward an algorithm involving the use of hashing, but one which does not require preprocessing of the text to be searched. Their approach is similar to that of Brute Force searching, but rather than directly comparing the m-symbol strings at successive text positions, their respective signatures are compared. This reduces the task of comparing two strings to the simpler one of comparing two integers. A carefully chosen hash function allows the text substring signatures to be easily calculated in an incremental manner. The algorithm is not restricted to string matching and may be extended to multi-dimensional pattern matching, in image processing applications for example. In the worst case, which is extremely unlikely, the algorithm takes $O(mn)$ time, but the expected performance is $O(m + n)$.

The comparative performance of the various algorithms described above has been studied both theoretically and empirically by many investigators (e.g. Horspool, 1980; Smit, 1982; Baeza-Yates, 1989a; Davies and Bowsher, 1986; Pirklbauer, 1992). The overall conclusion of these studies is that, for general applications, the Boyer-Moore-Horspool is the best available algorithm. As noted earlier, this is the recommendation given for almost all pattern lengths and alphabet sizes by Baeza-Yates (1989a). In the extreme cases of very small patterns (i.e. $m \in [1, 3]$) and pattern lengths commensurate with the text length (i.e. $m \approx n$), the Brute Force approach is preferable. Horspool (1980) observed that his simplified variant gave an average performance comparable to that of the Boyer-Moore algorithm, and outperformed a Brute Force method coded using special purpose machine instructions for symbol searching when $m \geqslant 6$.

Despite its theoretical elegance, the non-backtracking Knuth-Morris-Pratt algorithm provides no significant speed advantage over the Brute Force method in practice. It may, however, be a good choice when $|C| \approx n$ (Baeza-Yates, 1989a), or when dealing with binary strings, i.e. when $|C| = 2$ (Davies and Bowsher, 1986). Although it may inspect text symbols more than once and may backtrack in the input string, the Boyer-Moore approach provides, on the whole, a significantly faster search method.

In terms of execution time, the Karp-Rabin hashing technique fares badly in competition with the other methods, but is extendible to higher dimensional problems. It may therefore be used for pattern matching in image processing applications, for example.

With regards to Sunday's variations of the Boyer-Moore approach, Pirklbauer (1992) recommends the Sunday Quick Search in addition to a simplified Boyer-Moore variant. However, he advises against the use of the Maximal Shift and Optimal Mismatch algorithms, on the grounds of the resulting length of code and excessive preprocessing loads, especially significant when searching for only the first occurrence of a pattern. Hume and Sunday (1991) report a significant improvement in performance over the original Boyer-Moore algorithm for their new methods. Again, however, preprocessing loads were not taken into account in the measurements.

Although providing a high performance, the degree to which the Boyer-Moore approach has been put into practice may have been curbed to a certain extent by conceptual difficulties in the preprocessing, particularly with the match heuristic. Horspool (1980) observed that "*many programmers may not believe that the Boyer and Moore algorithm (if they have heard of it) is a truly practical approach.*" Sedgewick (1983) also noted that "*both the Knuth-Morris-Pratt and the Boyer-Moore algorithms require some complicated preprocessing on the pattern that is difficult to understand and has limited the extent to which they are used.*" In fact, Morris discovered that after a few months his initial search implementation in a text editor, being too difficult to understand by other systems programmers, had been ruined by various gratuitous 'fixes' (Knuth, Morris and Pratt, 1977). Hume and Sunday (1991) add that "*partially because the best algorithms presented in the literature are difficult to understand and to implement, knowledge of fast and practical algorithms is not commonplace.*" Arguing for a more consistent approach to algorithm development, and against "*widespread chaotic algorithm presentation,*" Woude (1989) asserts that the Knuth-Morris-Pratt preprocessing should be no more difficult to understand than the actual search procedure itself. There have also been some attempts of late to popularise the Boyer-Moore approach, particularly in its simplified form (e.g. Menico, 1989). As a step towards elucidating this field, Hume and Sunday (1991) have presented a means of classifying the various string matching algorithms based on the Knuth-Morris-Pratt and Boyer-Moore approaches.

3.2 String Distance Metrics

The notion of a distance function, or metric, crops up in many diverse fields, and is often employed as a measure of the similarity between two vectors, or n-tuples. This allows the degree of correlation between two entities to be assessed, and, for example, has uses in pattern recognition applications such as template matching in image processing. Formally, a distance metric, d, is a function satisfying the following axioms.

$$\begin{array}{rrcll} \text{Nonnegative Property} & d(x,y) & \geqslant & 0 \ \forall x,y & (3.1) \\ \text{Zero Property} & d(x,y) & = & 0 \ \Leftrightarrow \ x=y & (3.2) \\ \text{Symmetry} & d(x,y) & = & d(y,x) \ \forall x,y & (3.3) \\ \text{Triangle Inequality} & d(x,z) & \leqslant & d(x,y)+d(y,z) \ \forall x,y,z & (3.4) \end{array}$$

In order to evaluate the distances between sequences, a sequence of length n may be regarded as a vector in R^n. For two vectors of the same dimensionality (sequences of the same length), with components of either discrete or continuous numerical values, common metrics are the chessboard distance ($\max_i \mid x_i - y_i \mid$), city-block distance ($\sum_i \mid x_i - y_i \mid$), and the familiar Euclidean distance ($\left[\sum_i (x_i - y_i)^2\right]^{1/2}$).

In string processing applications, however, it is often not appropriate to interpret the component symbols as numerical values, although they are normally assigned such for the convenience of coding, and strings to be compared are often of disparate length. The distance metrics employed in this area therefore tend to quantify the minimum cost of transforming one string into the other. In general, cost weights may be assigned to the individual editing operations involved in such a transformation, namely symbol substitution, insertion, and deletion. Note that the latter two operations are sometimes referred to collectively as *indels*.

For two strings of equal length, the *Hamming distance* (Hamming, 1982) between them is defined as the number of symbol positions at which differ. This is equivalent to the minimum cost of transforming the first string into the second, when only substitutions are permitted and are given a unity weighting. When not restricted to comparing equal length strings, insertions and deletions will also generally be required. Giving these the same weight as a substitution, the minimum total transformation cost is equal to one of the two distance metrics proposed by Levenshtein (1965). His other metric is equal to the minimum transformation cost when only indels are permitted. This is equivalent to assigning a cost of 1 to indels and 2 to substitutions, as the latter may be implemented by an insertion-deletion pair. The former metric shall henceforth be referred to as the *Levenshtein distance*, and the latter as the *edit distance*.

The use of such metrics as similarity measures between strings finds application in many different areas, such as quantifying degrees of homology in genetic sequences, for example. Note, however, that not all string distance functions are necessarily metrics. For example, when the cost of editing operations is a function of the particular symbols involved, the triangle inequality may not necessarily be satisfied. A comprehensive discussion of a host of application milieux, from molecular biology to speech processing, employing string distance functions is given by Sankoff and Kruskall (1983).

The remainder of this section gives an overview of the major algorithms developed to solve both the string distance problem, sometimes given under the guise of the file difference problem or the string-to-string correction problem, and also the related longest common subsequence problem. A recent review of these problems may be found in Aho (1990), and an overview of complexity results for common subsequence problems is presented by Hirschberg (1983). Again, a more detailed examination of the principal algorithms has been deferred until the next chapter. Before going any further, however, definitions of the problems shall first be given.

String Distance Problem

> Given strings x and y, with $|x| = m$, $|y| = n$, where $m, n > 0$, and a string distance metric, d, find $d(x, y)$. When given as the file difference problem, x and y represent two files, where x_i is the i^{th} line of x and y_j is the j^{th} line of y. The requirement is then to determine a minimal sequence of editing operations necessary to transform x to y.

Longest Common Subsequence Problem

> A subsequence of a string is obtained be eliminating zero or more symbols, not necessarily contiguous, from the string. A common subsequence of two strings is a string which occurs as a subsequence of both strings. A longest common subsequence of two strings is a common subsequence of the strings having maximal length, i.e. it is at least as long as any other common subsequence of the strings. Given two strings x and y, with $|x| = m$, $|y| = n$ and $m, n > 0$, find $|\text{lcs}(x, y)|$, where $\text{lcs}(x, y)$ is a longest common subsequence of x and y. It is sometimes also required to determine an $\text{lcs}(x, y)$ in addition to its length.

One method of evaluating the distance between two strings, based on a dynamic programming (Bellman and Dreyfus, 1962) technique, has been discovered independently by many different investigators. Sankoff and Kruskall (1983) credit the following authors with procedures all based on the same underlying principle — Vintsyuk (1968); Needleman and Wunsch (1970); Velichko and Zagoruyko (1970); Sakoe and Chiba (1970, 1971); Sankoff (1972); Reichert, Cohen and Wong (1973); Haton (1973); Wagner and Fischer (1974); and Hirschberg (1975).

The basic idea of the dynamic programming method is to evaluate successively the distance between longer and longer prefixes of the two strings until the final result is obtained. These partial results are computed iteratively and are entered in an $(m + 1) \times (n + 1)$ array, leading to $O(mn)$ time and space complexities. In Wagner and Fischer's (1974) method, an lcs may then be derived in a straightforward manner from the completed distance array via a structure known as a *trace*.

Hirschberg (1975) modified Wagner and Fischer's technique, preserving the $O(mn)$ time complexity but reducing the space requirement from a quadratic to a linear characteristic. In this method it is the length of the lcs of progressively longer string prefixes, rather than the distance between them, that is computed iteratively. The saving in space is effected by holding only two rows of the dynamic programming array in memory at any given time. This loss of the full array, however, leads to a more complicated method of extracting an lcs of the two strings. The technique employed is to bisect recursively the first string and obtain an lcs between each half and an appropriate substring of the second string. The recursion is performed until this lcs extraction becomes trivial. The final lcs of the complete strings is then obtained by concatenating those obtained for the recursively sliced string sections.

Hunt and McIlroy (1976) implemented Hirschberg's lcs algorithm for the unix `diff` command. This was found to work well for short inputs, but its quadratic time complexity resulted in significant degradation in execution speed as the lengths of the input files were increased. However, McIlroy later obtained a dramatic improvement in performance by employing the lcs algorithm developed by Hunt and Szymanski (1977).

Hunt and Szymanski's approach to extracting an lcs from two strings is equivalent to determining the longest monotonically increasing path in the graph composed of nodes (i, j) such that $x_i = y_j$. Whereas previous methods required quadratic time in all cases, their algorithm requires $O((r+n)\log n)$ time for equal length strings, where r is the total number of ordered pairs of positions at which the two strings match. In the worst case, every element of x could match every symbol in y, resulting in n^2 such ordered pairs and a complexity of $O(n^2 \log n)$. For many applications, however, such as the file difference problem where each line of a file is taken as an atomic symbol, r can be expected to be close to n, giving a performance of $O(n \log n)$ — a significant improvement over the quadratic time procedures.

Masek and Paterson (1980, 1983) developed a string distance algorithm based on Wagner and Fischer's dynamic programming approach which has a sub-quadratic time complexity, even in the worst case. The method does, however, require the alphabet to be finite and the cost weights to be integral multiples of a fixed real number. Their algorithm uses the 'Four Russians' technique (Arlazarov, Dinic, Kronod and Faradzev, 1970), and has a worst case execution time of $O(n^2/\log n)$ for equal length input strings. The basic idea of this method is to partition the distance matrix into a number of smaller submatrices. The lower and rightmost edges of each submatrix may then be calculated from those of adjacent submatrices, above and to the left of the current one, and from the appropriate substrings of x and y using a precomputed lookup table. Although asymptotically faster than the straightforward dynamic programming method, Masek and Paterson (1983) point out that their algorithm should only be used for long strings, and, for example, is actually faster to compute the edit distance between two binary strings only once the string lengths exceed 262418.

Szymanski has shown that alphabets of unrestricted size may be accommodated by a modified version of Masek and Paterson's algorithm, taking time $O(n^2(\log\log n)^2/\log n)$ — a behaviour still asymptotically better than quadratic (Hirschberg, 1983).

An alternative lcs algorithm, having a favourable performance for the case of similar strings, has been proposed independently by Ukkonen (1983, 1985a) and Myers (1986). The underlying principle of their methods is to search for a minimum-cost path in a graph. This graph is composed of the vertices of the distance matrix together with horizontal, vertical and diagonal edges corresponding to the dependencies between adjacent nodes. The required path in the graph starts at $(0, 0)$ and ends on (m, n).

In Ukkonen's method, unnecessary calculations in the distance matrix are avoided in a threshold test of the distance between the two strings. This process is repeated with successively larger threshold values until the test is successful, yielding the required string distance in $O(md)$ time. An actual editing sequence may then be recovered from the computed distance matrix values as in the Wagner-Fischer algorithm.

Myers' approach is to find the endpoints of successively longer furthest-reaching paths in the graphs until point (m, n) is reached. In the worst case, this takes $O((m+n)d)$ time, but Myers has shown that the expected case temporal complexity is $O(m+n+d^2)$. A linear space version of the algorithm and a variation with a worst case time of $O((m+n)\log(m+n) + d^2)$ have also been developed. The latter is based on suffix tree techniques (McCreight, 1976) and its improved asymptotic behaviour is of theoretical interest. It is not, however, a viable technique owing to the complex methods involved in its implementation.

Apostolico and Guerra (1987) have analysed both the Hirschberg and the Hunt-Szymanski methods of finding a longest common subsequence, and have devised cognate algorithms with certain improvements. For cases where $m \ll n$, their version of Hirschberg's algorithm will be faster than the original, as their version has a time complexity of $O(m^2 \cdot \min\{\log|C|, \log m, \log(2n/m)\})$, where C is the alphabet. As noted above, Hunt and Szymanski's algorithm becomes worse than quadratic in the pessimal case. However, Apostolico and Guerra have limited the worst case behaviour to a quadratic characteristic. The actual time bound of their revised version of the algorithm is $O(m \log n + t \log(2mn/t))$, where $t \leqslant r$ is the number of *dominant matches* between the two strings.

The complexity of the lcs problem, for particular computational models, has been the subject of some investigation. Wong and Chandra (1976) have derived bounds for the calculation of string distance in terms of the editing cost function when the symbol comparisons are restricted to only tests of equality. They have also shown that $\Omega(mn)$ symbol comparisons are required to find an lcs under this restriction.

Using a similar 'equal-unequal' decision tree model of computation, Aho, Hirschberg and Ullman (1976) derived bounds for the lcs problem in terms of the size of the symbol alphabet. They found that, in general, the worst case minimum number of comparisons required is $O(n^2)$ for equal length strings, and is $O(n \cdot |C|)$ for $n \geqslant |C|$, where C is the alphabet.

When a lexicographic symbol ordering is employed, allowing 'less than-equal to-greater than' symbol comparisons to be performed, Hirschberg (1978) has shown that the minimum number of such comparisons required is $O(n \log n)$.

To sum up, the salient points concerning the performance of the various approaches to the related string distance and lcs problems are as follows. The dynamic programming method of Wagner and Fischer is simple to implement, but is quadratic in both time and space. For large applications, Hirschberg's linear space version is therefore preferable. However, the Hunt-Szymanski algorithm has a significantly better performance for many applications, at the cost of a worse than quadratic temporal complexity in the worst case. The only known algorithm requiring sub-quadratic time in the worst case is that of Masek and Paterson. The additional computational overheads of this method, however, make it more efficient than the straightforward dynamic programming technique only for very long strings. For the case where the two strings are similar, i.e. when d is small, the methods of Ukkonen or Myers become attractive.

Extended Problems

A related problem, often arising in molecular biology, is that of determining which portions of two sequences are similar, or homologous. This problem is discussed by Sankoff and Kruskall (1983), who expand upon Smith and Waterman's (1981) method of finding such *local alignments*. In addition to minimising the distance between portions of the sequences, some means of maximising the portion length is also required, since a minimum distance of zero will always be possible for trivial portions. A variation of the dynamic programming method is therefore used in which matches are rewarded and substitutions and indels penalised. This is achieved by assigning a positive cost to the former and negative ones to the latter operations. However, the quadratic nature of the dynamic programming method makes it impractical for use with very large sequence databases. Consequently, various heuristic methods have been developed for this problem, such as those of Lipman and Pearson (1985) and Altschul *et al.* (1990).

The problem of determining the distance between two strings when transpositions, or swaps, of adjacent symbols are permitted as editing operations in addition to indels and substitutions has also been studied. This case, for example, allows transposition errors occurring during manual keyboard operation to be taken into account when calculating string distances. Despite the significant increase in the difficulty of analysing this situation, the problem is still soluble in quadratic time using an extension of the dynamic programming method (Lowrance and Wagner, 1975; Wagner, 1975, 1983).

A generalisation of the lcs problem is the N-lcs problem, where it is required to determine a longest subsequence common to a set of N strings. Denoting the fact that x is a subsequence of y by $x < y$, the problem may be stated formally as follows. Given a set of N strings, $X = \{x_1,\ x_2, \ldots,\ x_N\}$, find N-lcs(X), where N-lcs(X) is a maximal length sequence such that N-lcs$(X) < x_i$ for integer $i \in [1, N]$. Under this convention, the lcs problem discussed previously would be known as the 2-lcs problem.

For arbitrary N, Maier (1978) has shown that the problem of determining $|N\text{-lcs}|$ is NP-complete (Karp, 1972) for alphabets of size two and above. The extraction of an N-lcs for such purposes as data compression is therefore not viable, as any means for so doing will inevitably be intractable (at least as long as there remains no known polynomial-time solution to the class of NP-complete problems). For example, applying the dynamic programming approach to the N-lcs problem would require $O(n^N)$ time and space for strings of length n.

For cases where it is required to solve the N-lcs problem for constant N and strings of length n, Baeza-Yates (1991) has proposed a method which still requires $O(n^N)$ worst case time and space, but has a more favourable average case behaviour. His approach is to build a partial deterministic finite automaton, known as a Directed Acyclic Subsequence Graph (DASG), which can recognise all the subsequences of the strings in the set X. As the graph is being constructed, its edges are labelled with the number of strings sharing that transition. The lcs's of X may then be recovered by searching the graph for maximal sequences of edges labelled with N.

The DASG can be constructed in $O(N \mid C \mid n \log n)$ time and $O((N + \mid C \mid)n)$ space. We shall denote the number of matched points between the N strings by r, which is pessimally equal to, but on average small in comparison to, n^N. Traversing the graph may be accomplished in $O(r)$ time, and at most $O(r)$ space is required to obtain the set of lcs's of the strings. The temporal and spatial complexities of the overall procedure are thus $O(r + N \mid C \mid n \log n)$ and $O(r + (N + \mid C \mid)n)$, respectively. This time bound improves over that of a method due to Hsu and Du (1984) which has similar complexities.

Maier (1978) has also addressed the N-scs problem. This is the problem of finding the shortest common supersequence of a set of N strings. If $x < y$, then y is said to be a supersequence of x. The N-scs problem may thus be stated as follows. Given a set of N strings, $X = \{x_1,\ x_2, \ldots,\ x_N\}$, find N-scs(X), where N-scs(X) is a minimal length sequence such that N-scs$(X) > x_i$ for integer $i \in [1, N]$. Maier found that, for alphabet sizes $\geqslant 5$, the problem of finding $|N$-scs$|$ is also NP-complete, and has conjectured that this result may also hold for alphabet sizes $\geqslant 3$.

The problem of obtaining a heaviest common subsequence (hcs) of two strings has been examined by Jacobson and Vo (1992). An hcs(x, y) is a subsequence common to x and y having a maximal sum of weights for its component symbols for a given weight function. In general, this function may depend on both the symbols themselves and also on their positions in the original strings.

The determination of an hcs finds application in, for example, the updating of CRT screens. The line indels required to transform one screen into another may be derived from an lcs of the two screens. In the interests of minimising screen disruption, however, it is preferable to give precedence to closely aligned pairs of matched lines. A minimal distance weight function, which diminishes as the distance between the positions in their respective sequences of matching symbols increases, may therefore be employed to obtain an appropriate hcs. The computation of an hcs may be performed using an adaptation of the dynamic programming lcs algorithm, which has been used, for example, in the `curses` screen update library distributed with System V unix (Vo, 1986).

Jacobson and Vo (1992) present an hcs algorithm derived from one to compute a heaviest increasing subsequence (his). This, in turn, was derived from the Robinson-Schensted (Schensted, 1961) method of computing a longest increasing subsequence (lis). An lis is a maximal string subsequence whose components are strictly increasing, given some linear ordering of the alphabet, and an his is an increasing subsequence with maximal weight. Their algorithm is a generalisation of the Hunt-Szymanski method, and has the same temporal complexity. For position-independent weight functions, a specialised form of the hcs procedure becomes a generalisation of the Apostolico-Guerra lcs algorithm.

3.3 Approximate String Matching

A generalisation of the string matching problem, which involves finding substrings of a text string close to a given pattern string, is the *Approximate String Matching Problem*. This variation of the problem is important when errors are being taken into consideration, and, for example, finds application in the field of molecular biology (see Landau, Vishkin and Nussinov (1985) for a discussion of some such applications). The approximate string matching problem may be stated as follows.

> Given a pattern string x, with $|x| = m$, text string y, with $|y| = n$, where $m, n > 0$ and $m \leqslant n$, an integer $k \geqslant 0$ and a distance function d, find all the substrings, s, of y such that $d(x, s) \leqslant k$.

The task is thus to determine all the text substrings having a distance of at most k from the pattern, for the given distance function. When d is the Hamming distance the problem is known as string matching with *k mismatches*, and when d is the Levenshtein distance it is known as string matching with *k differences* (or sometimes *errors*).

A comprehensive overview of approximate string matching algorithms is presented by Galil and Giancarlo (1988), and a comparison of the complexities of certain algorithms is also provided by Gonnet and Baeza-Yates (1991).

The $O(mn)$ time brute force string matching algorithm may be adapted easily to cater for the k-mismatch problem by allowing up to k mismatches in the comparisons at each substring position within the text. However, as with the string matching problem, various more efficient approaches to both the k-mismatch and the k-difference problems have been devised.

Landau and Vishkin (1985, 1986a) have developed an algorithm for the k-mismatch problem running in $O(k(m \log m + n))$ time. Their approach is similar to the Knuth-Morris-Pratt string matching technique in that a table derived from preprocessing of the pattern is employed as the text string is examined from left to right, and known information is exploited in order to reduce the number of symbol comparisons required.

Landau and Vishkin (1985, 1986a) also remark that a k-mismatch algorithm due to Ivanov (1984), which has a temporal complexity of $O(f(k)(m + n))$, is faster than their algorithm for the case of very small k and comparable values of m and n. However, they highlight the abstruse nature of the function f and have established that it is bounded from below by 2^k. Excepting the above case, their algorithm is thus faster than Ivanov's, and furthermore has the appeal of its simplicity — they state that Ivanov's algorithm required in excess of forty journal pages to describe.

A variant of Landau and Vishkin's k-mismatch algorithm with improved pattern preprocessing has been developed by Galil and Giancarlo (1986, 1988). The overall time complexity of their version is $O(m \log m + kn)$.

Sellers (1980) has shown how the dynamic programming method of computing the distance between two strings may be adapted to solve the k-difference problem. Once a variation of the distance array has been evaluated, if, and only if, there is some value in the last row, $d_{m,j}$ say, such that $d_{m,j} \leqslant k$, then there is a substring, s, of y such that $d(x, s) \leqslant k$. The actual substring may then be recovered by tracing back through the dependency graph. The time required for this method is thus $O(mn)$.

A k-difference algorithm proposed by Ukkonen (1985b) comprises a preprocessing and a text analysis stage. A deterministic finite-state automaton is constructed in the former stage, which is then used to scan the input text in the latter in $O(n)$ time. However, this algorithm is not, in general, practicable owing to its exponential requirements for preprocessing time and space. (For alphabet C, the pattern analysis requires $O(m \mid C \mid \min\{3^m, 2^k \mid C \mid^k m^{k+1}\})$ time and $O((\mid C \mid + m) \min\{3^m, 2^k \mid C \mid^k m^{k+1}\})$ space.)

Landau and Vishkin have developed several k-difference algorithms based on Ukkonen's (1983, 1985a) efficient, $O(md)$ time method of computing inter-string distances. Preprocessing of the pattern in these algorithms permits known information to be exploited in the subsequent analysis of the text. Their (1985) algorithm successively tests text positions for the occurrence — with up to k differences — of the pattern, and runs in $O(m^2 + k^2 n)$ time and requires $O(m^2)$ space. The performance of this method was later (1988) improved so as to require $O(m)$ space and $O(k^2 n)$ time for a fixed size alphabet and $O(m \log m + k^2 n)$ time for unbounded alphabets. The improved pattern preprocessing of this method involves a suffix tree (constructed using Weiner's (1973) algorithm), and the text analysis stage involves finding *lowest common ancestors* (LCA) (Harel and Tarjan, 1984; Schieber and Vishkin, 1988) in the suffix tree. Landau and Vishkin (1986b, 1989) also discovered another k-difference algorithm involving the use of a suffix tree as a by-product of developing a parallel algorithm. The basis of this approach is to compute efficiently the dynamic programming string distance matrix, and the algorithm locates text positions at which k-difference occurrences of the pattern end. This method takes $O(kn)$ time for a fixed size alphabet and $O(n \log m + kn)$ time for unbounded alphabets.

Galil and Giancarlo (1988) discuss how Masek and Paterson's (1980, 1983) string distance algorithm may be adapted in order to solve the k-difference problem, resulting in an $O(mn/\log n)$ time computation. Recall that this approach requires the alphabet to be finite.

Taking stock of the k-difference algorithms mentioned above leads to the following observations. Although the exponential requirements of Ukkonen's method preclude its general application, it could conceivably be used for very small patterns or for small alphabets and very small values of k. For fixed size alphabets, it may be seen that if k is $\Omega(m/\log n)$, then the variation of Masek and Paterson's algorithm is asymptotically the fastest of the above methods. (The comment made previously concerning the performance of the Masek-Paterson algorithm — namely that it is actually faster than the straightforward dynamic programming method only for very large strings — should, however, be borne in mind.) If, on the other hand, k is $O(m/\log n)$, then Landau and Vishkin's (1986b,1989) algorithm becomes the method of choice. For the case of unbounded alphabets, their (1988) algorithm becomes faster if k is $O(\log^{\frac{1}{2}} m)$. However, the (1986b, 1989) method does have the advantage of being a significantly simpler algorithm, and is, of course, superior if k is $\Omega(\log^{\frac{1}{2}} m)$. Note, however, that for large k, i.e. when k is $\Theta(m)$, the performance becomes no better than Sellers' adaptation of the straightforward, quadratic time, dynamic programming approach.

One area in which approximate matching is relevant is the modelling of genetic sequences. DNA sequences are, for example, sometimes modelled as sequences of independent and identically distributed (i.i.d.) symbols from the alphabet {A, C, G, T}. The components of this alphabet represent occurrences of the nucleotides adenine, cytosine, guanine and thymine, respectively. Similarly, proteins are sometimes modelled as i.i.d. sequences of amino acids (e.g. Sibbald and White, 1987). The distribution of the length of the longest common substring of two independent random sequences has been examined by relating it to that of the longest run of heads in a sequence of tosses of a coin. Theoretical results have been obtained for exact matching (Arratia and Waterman, 1985a) and also for the cases of k-mismatches and k-differences (Arratia and Waterman, 1985b; Arratia, Gordon and Waterman, 1986).

A related string matching problem is that involving *don't care*, or wild card, symbols which match any single symbol, including another don't care. Pinter (1985) has addressed this problem for the case where only the pattern string contains don't cares. His algorithm is an extension of the Aho-Corasick (1975) string matching procedure. The latter is a generalisation of the Knuth-Morris-Pratt algorithm, and searches for multiple patterns in the text string in $O(m+n)$ time, where m is the sum of the lengths of the pattern strings. The preprocessing stage of this algorithm constructs in $O(m)$ time a pattern matching automaton, similar to a deterministic finite automaton but having two transition functions, from the set of patterns. This is then used to scan the text and search simultaneously for occurrences of the pattern strings in $O(n)$ time. The linearity of the method is preserved in Pinter's algorithm provided that the elements of an array of accumulators may be incremented in parallel; otherwise the temporal complexity becomes quadratic.

The case where both the pattern and text strings may contain don't care symbols has been examined by Fischer and Paterson (1974). For strings over a finite alphabet, C, they have demonstrated that this problem is equivalent to integer multiplication, and using Schönhage-Strassen (1971; see also Aho, Hopcroft and Ullman, 1974) multiplication have derived an indirect algorithm running in $O(n \log^2 m \log\log m \log \mid C \mid)$ time. The constant term of this complexity expression is not, however, inconsiderable, and the algorithm is thus only of practical import for very long strings. For the case of unbounded alphabets, there is as yet no known algorithm with performance superior to that of the quadratic brute force approach to string matching.

Special-purpose hardware

Various special-purpose hardware designs for string matching applications, such as text retrieval from large, unformatted document sets, have been developed. These have employed such techniques as those involving cellular arrays (Hollaar, 1979; Foster and Kung, 1980; Mukhopadhyay, 1980; Curry and Mukhopadhyay, 1983; Halaas, 1983; Lee and Mak, 1989), associative memories (Burkowski, 1982; Lee and Lochovsky, 1985), finite state automata (Hollaar, 1979; Haskin, 1981; Robert, 1982; Haskin and Hollaar, 1983), and dynamic programming (Hall and Dowling, 1980; Salton, 1980; Yianilos, 1983).

One particular CMOS VLSI device, developed fairly recently by NEC (Yamada *et al.*, 1987; Hirata *et al.*, 1988), combines finite state automaton (FSA) logic with a content addressable memory (CAM), and allows approximate string matching operations to be performed. A 512 character (16 bit) CAM may be configured to contain from 64 8-character pattern strings to 1 512-character pattern string. Serial input data, at a rate of 10 million characters/s, is fed through the CAM, which performs parallel matching and passes match signals to the FSA logic. In addition to exact string matching, an approximate matching mode allows occurrences of patterns with up to 1 difference to be found. There is also a variable length don't care mode, which searches for strings starting with a given prefix, ending with a given suffix, and having variable length intermediate sections. Finally, a mask mode permits wild cards in the input stream to be supported, which would, for example, allow unrecognised characters from an OCR unit to be coded and to match any symbol in a pattern string.

3.4 Longest Repeated Substring

The problem of finding the longest substring which occurs more than once in a given string is known as the *Longest Repeated Substring* problem. This may be formulated as follows.

> Given string y, with $\mid y \mid = n$, where $n > 0$, identify and locate the longest substring, x, occurring more than once in y. A longest repeated substring is thus the longest maximal length string, x, such that $y = uxvxw$ for some strings u, v and w with $|u| \geqslant 0$, $|v| \geqslant 0$ and $|w| \geqslant 0$.

A maximal repeated substring is one having at least two distinct matching occurrences in the given string, where the match may be extended no further in either direction. As an example, consider the string `PABCQRABCSABTU`. Substrings `A`, `B`, `C`, `AB`, `BC` and `ABC` are all repeated in the original string. Strings `AB` and `ABC` are maximal repeated substrings, and `ABC` is thus the longest repeated substring.

A brute force approach to the problem, requiring quadratic time, is as follows. An $n \times n$ match matrix, M, is constructed such that $M_{i,j} = 1$ if $y_i = y_j$, otherwise $M_{i,j} = 0$. With the exception of the main diagonal, contiguous diagonals of '1's in the matrix represent maximal recurring substrings, the longest of which is thus the longest repeated substring. Note that the matrix is symmetrical about the main diagonal, and thus only the upper half, say, (i.e. elements $M_{1 \ldots n-1, i+1 \ldots n}$) need be evaluated and processed.

However, the longest repeated substring may be found in linear time using one of several variations of the data structure proposed by Weiner (1973), which comprises a compact index to all distinct substrings of a string. This index has been studied widely under the following guises:

Patricia trees (Morrison, 1968)

Position trees (Aho, Hopcroft and Ullman, 1974; Majster and Reiser, 1980)

Suffix trees (McCreight, 1976; Rodeh, Pratt and Even, 1981; Crochemore, 1986)

Complete inverted files (Blumer *et al.*, 1984a, 1984b, 1985, 1987)

Subword trees (Apostolico, 1985)

An early, implicit manifestation of the index is to be found in Morrison's Patricia tree structure. It was, however, Weiner who pioneered the explicit index, for which a clean construction is presented by Chen and Seiferas (1985). Majster and Reiser's (1980) on-line construction allows the index to be built as the input string is read in a symbol at a time from left to right. Apostolico (1985) details several applications of the index (in the form of McCreight's suffix tree), such as finding longest common substrings and algorithms to perform linear-time sequential data compression. The basic idea of the latter is to interleave the construction of a suffix tree with the parsing of the text string into phrases. Rodeh, Pratt and Even (1981) have, for example, implemented the optimal universal data compression methods of Ziv and Lempel (1977) using suffix trees.

The construction of suffix trees and their application to this problem are described in more detail in the following chapter.

Finally, a variation of the problem involves finding all the maximal repeated substrings of a given string. Again, the quadratic time, brute force approach may be brought to bear on this problem. The use of suffix trees in solving this variation has, however, been investigated by Baker (1992). She has developed a software visualisation tool for finding occurrences of duplicated or related sections of code in large software systems. Lines of code, stripped of comments and white space, are taken to be the component string symbols, and a threshold on the minimum length of repeated substrings may be set in order to filter out trivial duplications. The method is based on McCreight's suffix tree construction, and the overall algorithm runs in $O(n+r)$ time, where r is the number of pairs of maximal repeated substrings found (which is normally small in comparison with n).

Chapter 4

Description of the Algorithms

Errors, like straws, upon the surface flow;
He would search for pearls must dive below.
— John Dryden (1631-1700), *All for Love*, Prologue.

This chapter is devoted to a more in-depth discussion of the major algorithms outlined in the overview of the previous chapter.

4.1 String Matching

4.1.1 Brute Force

In the Brute Force approach, x is compared with substrings $y(i, i+m-1)$ at successive positions, i, in the text string (for $1 \leqslant i \leqslant n-m+1$), until a match is found or the end of the text is reached.

As a maximum of m symbol comparisons may be performed for each position i in y, a total of $m(n-m+1)$ comparisons are required in the worst case. For this maximum to be incurred, the substring comparison for each position i must either first fail on the last (m^{th}) symbol comparison, or in the case of the final position ($i = n-m+1$) alternatively provide a successful match. This situation arises in the following example.

$$
\begin{aligned}
x &= \texttt{EEEEW} \\
y &= \texttt{EEEEEEEEEEEEEEEEEEEEEW}
\end{aligned}
$$

However, this type of case is likely to occur but seldom in practical applications of text searching. The expected performance is therefore about $O(m+n)$, since the majority of

```
i = 1        — initialise y index
j = 1        — initialise x index
while (i ⩽ n) and (j ⩽ m)
    if x_j = y_i
        i = i + 1
        j = j + 1
    else
        i = i - j + 2
        j = 1
if j > m     — check for successful match
    i = i - m
else
    i = 0
```

Figure 4.1: Brute Force string matching

the unsuccessful substring matching attempts are likely to fail very early on. The actual performance will, of course, depend on the statistical parameters of the particular pattern and text involved.

An implementation of the Brute Force string matching algorithm is given in the pseudocode of Figure 4.1. The pattern and text indices are initially both set to point to the start of their respective strings, and are both incremented as long as the symbols to which they point match. In the event of a symbol mismatch, j is reset to point to the start of the pattern, and i to the next potential pattern position in the text. If a match is found, then i is set to the starting position in y of the pattern, otherwise it is reset to zero to indicate that the text search has been unsuccessful.

4.1.2 Knuth-Morris-Pratt

The basic method of the Knuth-Morris-Pratt algorithm is similar to that of the Brute Force approach, except that the text index, i, is never decremented. This is achieved by shifting pattern x forward relative to text y, in the event of a mismatch, by an amount dependent on both the structure of the pattern and the position, j, within x at which the mismatch occurred. This shift information is obtained from an m-entry auxiliary table, *next*, precomputed from the pattern before searching the text.

The search algorithm is given in Figure 4.2. In the event of a mismatch between the text and the pattern at positions i and j respectively, then it is known that the previous $j - 1$ text symbols, i.e. substring $y(i - j + 1, i - 1)$, match the first $j - 1$ pattern symbols, i.e. prefix $x(1, j - 1)$. This information is exploited by the use of the *next* table, which contains values for the next pattern position, $next_j$, to be used after a mismatch at pattern position j.

In the inner loop j is set to $next_j$, which is equivalent to shifting the pattern $j - next_j$

```
i = 1      — initialise y index
j = 1      — initialise x index
while (i ⩽ n) and (j ⩽ m)
    while (j > 0) and (x_j ≠ y_i)
        j = next_j
    i = i + 1
    j = j + 1
if j > m
    i = i − m
else
    i = 0
```

NOTE that the **and** in the inner loop is a 'conditional and,' which only evaluates its right hand operand if its left hand one evaluates to true.

Figure 4.2: Knuth-Morris-Pratt string matching

symbol positions to the right relative to the text. This is performed repeatedly until either $j = 0$ or $x_j = y_i$. In the former case, none of x matches the current text substring; in the latter, the prefix $x(1, j)$ matches the substring $y(i - j + 1, i)$. The outer loop then resumes by incrementing the text and pattern indices. If a complete match is found, then i is set to the starting position of the occurrence of x in y, otherwise it is reset to 0.

The crucial part of the algorithm is the *next* table. Before considering it in its final form, a simplified version of the *next* table shall be examined. This is illustrated in the following example.

j	1	2	3	4	5	6	7	8
x_j	A	B	C	D	A	B	C	E
$next_j$	0	1	1	1	1	2	3	4

This shows that if, for example, matching between the above pattern and a given text string fails at $j = 8$, then the next pattern symbol to be compared with the current text symbol is x_4. This situation is shown below.

Match attempt fails at comparison of y_i and x_8

```
                          j = 8
pattern        A B C D A B C E
text       ... A B C D A B C D ...
                          i
```

Next comparison is between y_i and x_4

```
                          j = 4
pattern                   A B C D A B C E
text           . . . A B C D A B C D . . .
                                i
```

The entries of the *next* table may be derived from x as follows. For a mismatch between x_j and y_i, it is known that $x(1, j-1)$ matches the previous $j-1$ symbols of y. If there exists a proper prefix of substring $x(1, j-1)$, of maximal length $k-1$ say, i.e. $x(1, k-1)$, equal to a suffix of the same substring, i.e. $x(j-k+1, j-1)$, then the next pattern symbol to try is that obtained by shifting the pattern relative to the text until the prefix occupies the space previously occupied by the suffix. The next symbol to be compared with y_i is therefore the one immediately following the prefix, namely x_k. For the case of a mismatch at the first pattern symbol, $next_1$ is given the value of 0 so that the entire pattern is shifted past y_i.

To reiterate, the value of $next_j$ is given by the maximum k, less than j, such that $x(1, k-1)$ is a suffix of $x(1, j-1)$, i.e. $x(1, k-1) = x(j-k+1, j-1)$ (note that for the case of the prefix $x(1, k-1) = \epsilon$, k is taken to be 1). This may be found in the following manner. Compare $x(1, j-1)$ with itself by sliding one copy over the other from left to right. Start in the position such that the first comparison is between x_1 and x_2, and stop when either all the overlapping symbols match or there is none left. The overlapping characters then define the required prefix, and k is given by the size of this prefix plus 1.

Returning to the previous example pattern, the situation for $next_8$ is shown below.

```
                    A B C D A B C
x(1,7)      A B C D A B C
```

From the above, it may be seen that there are 3 overlapping symbols. In this case, the value of $next_8$ is therefore equal to 4.

Note that in the search procedure, just after it has been incremented, the value of j is the largest integer, less than or equal to i, such that $x(1, j-1) = y(i-j+1, i-1)$. This is exactly the requirement described earlier for the *next* values, when y is replaced by x. The method to generate the *next* table is therefore similar to the actual search procedure, with the exception that the pattern is being matched against itself. The pseudocode for this is given in Figure 4.3.

It was mentioned previously that the above formulation of the *next* table caters for a simplified case. This does not take into account all the available information, and advantage may still be taken of the actual symbol causing a mismatch. Consider the previous example. If there is a mismatch between y_i and x_7 (the second C), say, then the next comparison would be between

$j = 1$
$k = 0$
$next_1 = 0$ — special value for mismatch at x_1
while $j < m$
  while $(k > 0)$ **and** $(x_j \neq x_k)$
    $k = next_k$
  $j = j + 1$
  $k = k + 1$
  $next_j = k$

NOTE that the **and** in the inner loop is a 'conditional and.'

Figure 4.3: Initialisation of the *next* table (simplified)

y_i and x_{next_7}, i.e. x_3 (the first C). However, this comparison will also fail and its *next* value will shift the comparison to x_1. An improved *next* table would take the comparison directly to x_1 in the event of a mismatch at x_7. The modified requirement for $next_j$ is therefore that it is now equal to the greatest k less than j such that $x(1, k-1) = x(j-k+1, j-1)$ *and* $x_j \neq x_k$, and is equal to 0 if there is no such k. Using this definition, the new *next* table is given below.

j	1	2	3	4	5	6	7	8
x_j	A	B	C	D	A	B	C	E
$next_j$	0	1	1	1	0	1	1	4

This revised requirement is easily accommodated by modifying the $next_j$ assignment in the *next* table creation algorithm, giving the final version of Figure 4.4. The value of $next_k$ is now assigned to $next_j$ if $x_j = x_k$. This ensures that on a mismatch, a match is not next attempted between the current text symbol and another pattern symbol equal to the one just tried.

The search algorithm given in Figure 4.2 has the unusual feature that the total number of times that the assignment "$j = next_j$" in the inner loop is executed never exceeds that of the increment operation of i in the outer loop. The pattern is therefore shifted to the right a total of at most n times by the inner loop, giving a matching time of $O(n)$. Similarly, it may be shown that the *next* table initialisation stage is performed in $O(m)$ time. This gives an overall worst case time of $O(m+n)$, which incidentally is independent of the size of the symbol alphabet.

The worst case time is incurred for the task of matching a Fibonacci string pattern. Also, the maximum number of times that the inner loop is executed whilst still at the same position in the text has been shown to be $1 + \log_\phi m$, where ϕ is the golden ratio, equal to $(1+\sqrt{5})/2 \approx 1.618$ (Knuth, Morris and Pratt, 1977).

```
j = 1
k = 0
next_1 = 0      — special value for mismatch at x_1
while j < m
   while (k > 0) and (x_j ≠ x_k)
      k = next_k
   j = j + 1
   k = k + 1
   if x_j = x_k
      next_j = next_k
   else
      next_j = k
```

NOTE that the **and** in the inner loop is a 'conditional and.'

Figure 4.4: Initialisation of the *next* table

The average case performance of the algorithm for random strings has been analysed by Baeza-Yates (1989b). An upper bound for the expected number of symbol comparisons, A_n, was derived which improves upon an earlier result (Regnier, 1988). This is given below.

$$\frac{A_n}{n} \leqslant 1 + \frac{1}{|C|} + \frac{\lceil \log_\phi m \rceil - 1}{|C|\,3} + O\left(\frac{\log^2 m}{|C|^4}\right) \tag{4.1}$$

for $|C| > \lceil \log_\phi m \rceil$, where C is the alphabet. To give a numerical value to this bound, consider the case of a pattern length of 10 and an alphabet size of 256. This gives an upper bound for A_n of $\approx 1.004n$. Note that, with the same parameters and for large n, a virtually identical result is obtained for the average case of the Brute Force algorithm, for which the expected value of A_n for random strings is as follows (Baeza-Yates, 1989b).

$$A_n = \frac{|C|}{|C| - 1}\left(1 - \frac{1}{|C|^m}\right)(n - m + 1) + O(1) \tag{4.2}$$

Various implementation efficiency considerations were addressed by Knuth, Morris and Pratt (1977). For example, sentinels may be appended to the ends of the strings in order to reduce the requirement for index bound checking. Also, the *next* table may be used to compile a finite-state machine matcher for the pattern. Extensions to the problem, such as finding *all* the occurrences of the pattern in the text were also considered.

4.1.3 Boyer-Moore

When searching using the Boyer-Moore algorithm, the pattern is scanned across the text string from left to right, but the actual symbol comparisons between the pattern and the text are performed from right to left. The first comparison is therefore between x_m and y_m. In the event of a mismatch, if y_m does not occur at all in x, then the pattern may be shifted safely m places to the right. This is due to the fact that the possibility of x occurring in y starting at any of the first m positions has been ruled out. The next comparison would then be between x_m and y_{2m}.

Consider the case of a mismatch occurring between x_m and y_i, say. As mentioned above, if y_i does not appear anywhere in the pattern, then a shift of the pattern m places to the right may be effected. If, on the other hand, y_i does appear in the pattern and its rightmost occurrence is x_{m-s}, then the pattern may be shifted s places to the right, aligning x_{m-s} with y_i. Testing may be resumed by comparing x_m with y_{i+s}, i.e. the text index is incremented by s.

If a match is found between symbols x_m and y_i, then the corresponding preceding symbols from the pattern and the text are compared until either a complete match is obtained or a mismatch occurs. If a mismatch occurs at x_j, say, then the suffix, of length $m-j$, $x(j+1, m)$ will be equal to text substring $y(i-m+j+1, i)$, and $x_j \neq y_{i-m+j}$. If the rightmost occurrence in x of y_{i-m+j} is again x_{m-s}, say, then the pattern may be shifted $j-m+s$ places to the right, lining up x_{m-s} with y_{i-m+j}. Once more, the procedure may be continued by comparing x_m with its corresponding text symbol, in this case $y_{i+j-m+s}$. The increment in the text index from the mismatch position to that of the next comparison is therefore $(i+j-m+s) - (i-m+j) = s$. Note that this increment is the same as that for the previously considered case.

If x_{m-s} happens to be to the right of x_j, giving a negative value for $j-m+s$, there is nothing to be gained by aligning x_{m-s} with y_{i-m+j}, as this would involve a retrograde step. In these circumstances, therefore, the pattern may be shifted one place to the right, with the comparisons being resumed with symbols x_m and y_{i+1}. This requires a text index increment of $(i+1) - (i-m+j) = m+1-j$.

The increments, s, obtained from this *occurrence* heuristic are precomputed and stored in a *skip* table (denoted by $delta_1$ by Boyer and Moore (1977)). During searching, the table is indexed by the text symbol causing a mismatch, so a table the size of the symbol alphabet in use is required. The entry for symbol w is equal to $m-j$, where x_j is the rightmost occurrence of w in x, and equal to m if w does not occur at all in x, i.e.

$$skip[w] = \min\{s : s = m \text{ or } (0 \leqslant s < m \text{ and } x_{m-s} = w)\} \tag{4.3}$$

The *skip* table is simply initialised as follows for alphabet C.

for $w = 1$ **to** $|C|$
    $skip[w] = m$
for $j = 1$ **to** m
    $skip[x_j] = m - j$

From the above, it may be seen that the initialisation of the *skip* table is accomplished in $O(m + |C|)$ time.

The following example illustrates the operation of the *skip* table for the pattern `ABCDB`.

w	`A B C D E F G H . . .`
$skip[w]$	`4 0 2 1 5 5 5 5 . . .`

Mismatch occurs at $y_i =$ `A`

```
pattern          A B C D B
text       . . . L M N A B C D B . . .
                       i
```

Next comparison is between x_m and $y_{i+skip[\mathtt{A}]}$, i.e. y_{i+4}

```
pattern                A B C D B
text       . . . L M N A B C D B . . .
                       i       i + 4
```

For the case described earlier of a mismatch occurring after a partial match had been obtained, it is possible that a greater shift of the pattern than that given by the occurrence heuristic may be feasible. If this is so, then it is preferable to use the larger shift, which is obtained via the *match* heuristic. This is based on a table similar to that of the Knuth-Morris-Pratt algorithm. The idea behind this is that when the pattern is shifted, it must match all the symbols previously matched, and there must be a change in the pattern symbol at the text position that caused the mismatch. The latter condition was proposed independently by Kuipers (credited in Boyer and Moore (1977)) and by Knuth (in the postscript of Knuth, Morris and Pratt (1977)). It improves over Boyer and Moore's original heuristic in a manner analogous to that of the improvement of the *next* table of the Knuth-Morris-Pratt algorithm.

As mentioned earlier, the particular case under examination is when $x(j+1,m) = y(i-m+j+1,i)$ and $x_j \neq y_{i-m+j}$. If the suffix $x(j+1,m)$ also occurs in x as the substring $x(j+1-t, m-t)$, with $x_{j-t} \neq x_j$, and it is the rightmost such occurrence, then the pattern may be shifted t places to the right. This brings $x(j+1-t,m-t)$ into alignment with $y(i-m+j+1,i)$, and the search process may be resumed by comparing x_m with y_{i+t}. A *shift* table (denoted by $delta_2$ by Boyer and Moore (1977)) for the t shift information may be precomputed from the pattern and is indexed by the position in the pattern at which mismatch occurs. The value of $shift[j]$ is equal to a pattern shift, t, plus the extra shift required to restart the comparisons at the extreme right of the pattern. The value of t is the minimum such that when x_m is aligned with y_{i+t}, then assuming that $x_j \neq y_{i-m+j}$, pattern substring $x(j+1-t,m-t)$ will match text substring $y(i-m+j+1,i)$. Thus, the *shift* table entries are defined as follows.

$$\begin{aligned} shift[j] \; = \; & \min\{t+m-j \; : \; t \geqslant 1 \text{ and } (t \geqslant j \text{ or } x_{j-t} \neq x_j) \text{ and} \\ & ((t \geqslant k \text{ or } x_{k-t} = x_k) \;\; \text{for } j < k \leqslant m)\} \end{aligned} \tag{4.4}$$

The $shift[j]$ value is therefore equal to the required increment for the text index from the current position, y_{i-m+j}, to the next comparison position, y_{i+t}, i.e. $(i+t) - (i-m+j) = t+m-j$. The following example illustrates the *shift* values for the pattern `ABCDABC`.

x_j	`A B C D A B C`

```
i = m
j = m
while (j > 0) and (i ⩽ n)
   if y_i = x_j
      i = i − 1
      j = j − 1
   else
      — a mismatch
      i = i + max{skip[y_i], shift[j]}
      j = m
if j < 1
   i = i + 1
else
   i = 0
```

Figure 4.5: Boyer-Moore string matching

j	1	2	3	4	5	6	7
$m - j$	6	5	4	3	2	1	0
t	4	4	4	4	7	7	1
$shift[j]$	10	9	8	7	9	8	1

From the definition of $shift[j]$, it may be seen that $shift[j] \geqslant m + 1 - j$. Thus, in the case described earlier requiring a text index increment of $m + 1 - j$ rather than the relevant *skip* value, as the latter would involve a backwards pattern shift, it is sufficient to use the relevant *shift* table value. In the event of a mismatch while searching, the maximum of the text increments from the two heuristics described above should therefore be used. The algorithm for this method is given in Figure 4.5. The final value of i gives the position of the pattern in the text, with 0 indicating that the pattern was not found.

The *shift* table may be precomputed from the pattern as shown in Figure 4.6. This is the method presented by Aho (1990), originated by Knuth (Knuth, Morris and Pratt, 1977) and modified by Mehlhorn (Smit, 1982). The intermediate function, f, calculated in the algorithm is defined as follows: $f[m] = m + 1$, and for $1 \leqslant j < m$, $f[j] = \min\{i \ : \ j < i \leqslant m \text{ and } x(i+1, m) = x(j+1, j+m-i)\}$.

Knuth demonstrated that to locate all the occurrences of the pattern, the Boyer-Moore algorithm performs $O(n + rm)$ symbol comparisons, where r is the number of times the pattern appears in the text (Knuth, Morris and Pratt, 1977). He also showed that when the pattern is absent from the text, a total of $6n$ symbol comparisons is required. For large alphabets, the expected performance of the Boyer-Moore algorithm is sublinear, requiring about n/m symbol comparisons on average.

A variant of the Boyer-Moore algorithm developed by Apostolico and Giancarlo (1986) finds all of the occurrences of the pattern, performing at most $2n - m + 1$ symbol comparisons.

```
for i = 1 to m
    shift[i] = 2 * m - i
j = m
k = m + 1
while j > 0
    f[j] = k
    while (k ⩽ m) and (x_j ≠ x_k)
        shift[k] = min{shift[k], m - j}
        k = f[k]
    j = j - 1
    k = k - 1
for i = 1 to k
    shift[i] = min{shift[i], m + k - i}
j = f[k]
while k ⩽ m
    while k ⩽ j
        shift[k] = min{shift[k], j - k + m}
        k = k + 1
    j = f[j]
```

NOTE that the **and** in the second **while** loop is a 'conditional and.'

Figure 4.6: Initialisation of the *shift* table

The attendant preprocessing requirements are slightly more onerous, but are still linear in m. The method employed exploits knowledge of which pattern substrings matched which text substrings at previous trial positions, and uses the simpler form of the *shift* table definition.

There are many other variations of the Boyer-Moore approach to string matching. One of the more notable of these, the Boyer-Moore-Horspool algorithm, which involves simplified preprocessing, is described in the next section.

4.1.4 Boyer-Moore-Horspool

In practice, the match heuristic of the Boyer-Moore algorithm makes only a small contribution to the overall speed of the search. It does provide benefits for highly self repetitive patterns, but these seldom crop up in actual applications. The inclusion of the heuristic prevents the theoretical worst case performance from being $O(mn)$, as opposed to the actual linear behaviour.

Horspool (1980) developed a simplified form of the Boyer-Moore algorithm which dispenses with the match heuristic and relies solely on the occurrence heuristic. This reduces the preprocessing effort required, and also cuts down the space requirements from $O(m + |C|)$ to $O(|C|)$, for alphabet C. As the alphabet is usually fixed, the storage requirements will therefore normally be constant. Although the worst case performance is now $O(mn)$, this actually occurs only for rare pathological cases, and the expected performance is comparable to that of the full Boyer-Moore algorithm.

The *skip* table used in the Horspool variant is slightly different from the Boyer-Moore version, in that *skip*$[x_m]$ is assigned the value of m, and thus only x_1 to x_{m-1} need be used in the initialisation. In the Boyer-Moore table, the *skip* value for x_m is equal to 0, and a non-zero shift is obtained from the comparison to find the maximum of the match and occurrence heuristic values.

Horspool also noted that, in the event of a mismatch at x_j, say, the next pattern position to try may be that such that *any* of the $m - j$ matching text symbols is brought into alignment with the rightmost occurrence in the pattern of that particular symbol. By always using the text symbol corresponding to the last pattern symbol (x_m) to index the *skip* table, the situation described previously involving a backwards pattern shift may be avoided. This situation would otherwise have to be explicitly tested for. Recall that that situation is averted in the Boyer-Moore algorithm by virtue of the match heuristic. The Boyer-Moore-Horspool algorithm is depicted in Figure 4.7. Again, the pattern position is given by the final value of i, with 0 indicating that no match was found.

Baeza-Yates (1989a) has shown that the expected number of symbol comparisons, A_n, performed by the Boyer-Moore-Horspool algorithm for random strings is bounded from below as follows.

$$A_n \geqslant \frac{1 - \frac{1}{|C|^m}}{(|C| - 1)\left[1 - \left(1 - \frac{1}{|C|}\right)^m\right]} \; (n - m + 1) \tag{4.5}$$

For a pattern of length 10, the above relation gives lower bounds of 0.17, 0.11, and 0.10 times

— initialise *skip* table
for $w = 1$ **to** $|C|$
    $skip[w] = m$
for $j = 1$ **to** $m - 1$
    $skip[x_j] = m - j$
— search
$i = m$
$j = m$
while $(j > 0)$ **and** $(i \leqslant n)$
    $k = i$
    while $(j > 0)$ **and** $(y_k = x_j)$
        $k = k - 1$
        $j = j - 1$
    if $j > 0$
        — a mismatch
        $i = i + skip[y_i]$
        $j = m$
if $j < 1$
    $i = k + 1$
else
    $i = 0$

NOTE that the **and** in the inner **while** loop is a 'conditional and.'

Figure 4.7: Boyer-Moore-Horspool string matching

$(n - m + 1)$ for alphabet sizes of 10, 100, and 256 respectively. This shows that, for large alphabets, the lower bound is close to $(n - m + 1)/m$ in the average case, i.e. about n/m for $n \gg m$.

4.1.5 Sunday — Quick Search, Maximal Shift, and Optimal Mismatch

Sunday (1990) observed that in the Boyer-Moore algorithm, for a mismatch occurring when x_m is aligned with text symbol y_i, say, then the *next* text symbol, y_{i+1}, could be used rather than y_i to address the occurrence heuristic table. This is due to the fact that for the minimum pattern shift, i.e. one place to the right, y_{i+1} is part of the next text substring to be examined. Sunday states that in practice this should result in a shift greater than or equal to that of the Boyer-Moore *skip* value plus 1, on average. The speed advantage offered by this phenomenon, however, diminishes as the length of the pattern increases.

Smith (1991) has pointed out that there are circumstances under which the use of y_i rather than y_{i+1} is advantageous, as demonstrated in the example below for the pattern `ABCDE`. He therefore proposed using the maximum of the occurrence heuristic values for text symbols y_i and y_{i+1}.

Boyer-Moore *skip* table for the pattern `ABCDE`

w	A	B	C	D	E	F	G	H	...
$skip[w]$	4	3	2	1	0	5	5	5	...

Mismatch occurs at $y_i =$ `T`

```
pattern          A B C D E
text       . . . P Q R S T D U V W X . . .
                         i
```

Next pattern position based on $skip[y_i]$, i.e. $i + skip[\mathtt{T}]$

```
pattern                    A B C D E
text       . . . P Q R S T D U V W X . . .
                                   i + 5
```

Next pattern position based on $skip[y_{i+1}]$, i.e. $i + skip[\mathtt{D}] + 1$

```
pattern              A B C D E
text       . . . P Q R S T D U V W X . . .
                             i + 2
```

Sunday also noted that the symbol comparisons between the pattern and the current text substring may be performed in an arbitrary order, and used this fact in an effort to improve

```
— initialise skip table
for w = 1 to |C|
    skip[w] = m + 1
for j = 1 to m
    skip[x_j] = m + 1 − j
— search
i = 1
j = 1
while (j ⩽ m) and (i ⩽ n − m + 1)
    k = i
    while (j ⩽ m) and (y_k = x_j)
        k = k + 1
        j = j + 1
    if j ⩽ m
        — a mismatch
        i = i + skip[y_{i+m}]
        j = 1
if j ⩽ m
    i = 0
```

NOTE that the **and** in the inner **while** loop is a 'conditional and.'

Figure 4.8: Sunday Quick Search

the overall matching efficiency. He proposed three different variations of the Boyer-Moore search, all of which use y_{i+1} in the occurrence heuristic, but differ in the way in which the order of the symbol comparisons is determined. Sunday has conjectured that the linear worst case behaviour of the Boyer-Moore algorithm is preserved when the substring symbols are compared in an arbitrary order.

The simplest of the three is the Quick Search algorithm, in which the comparisons are performed strictly from left to right and only the occurrence heuristic is employed. Pseudocode for this method is given in Figure 4.8. Note that the ***skip*** table values are initialised to be 1 greater than those of the Boyer-Moore table to take into account the extra place in the pattern shifts.

In the other two methods — the Maximal Shift and the Optimal Mismatch algorithms — an additional array is used to store the pattern indices in the order that the symbol comparisons are performed at each trial pattern position. During searching, this array is therefore accessed sequentially so as to provide the desired order of symbol positions within the pattern.

In the Maximal Shift algorithm, the comparison order is chosen such that the match heuristic shift values are maximised. This is done by sorting the pattern symbols in descending order of their distance to their next leftward occurrence in the pattern, or, if there is none, to the start of the pattern. Thus, for a pattern containing no repeated symbols, the comparisons are performed in sequential order from right to left.

In the final algorithm, the Optimal Mismatch search, the probability of an early detection of a mismatch is maximised by comparing rarer symbols first. This requires the pattern symbols to be sorted in descending order of their frequency of occurrence in the text to be searched, necessitating an *a priori* knowledge of its statistics. Full implementations of the Maximal Shift and Optimal Mismatch algorithms, coded in C, may be found in Sunday (1990).

Sunday found the Optimal Mismatch version to be the fastest of his methods, and reported all three to be superior in average case performance to the Boyer-Moore algorithm — a speed advantage of 10-20% is reported for the Optimal Mismatch algorithm. The speed differentials, however, decreased with an increase in the pattern length, consistent with the behaviour obtained by using the text symbol following that causing a mismatch.

Sunday's algorithms have also been empirically compared with a simplified Boyer-Moore variant by Pirklbauer (1992). In this study, it was found that for long strings and for the case where preprocessing times were taken into account, the Maximal Shift and Optimal Mismatch procedures fared rather badly. Their preprocessing overheads were found to be significant, especially when searching for only the first occurrence of the pattern. The Quick Search algorithm was, however, recommended on the grounds of its performance, similar to that of the Boyer-Moore variant, and its ease of implementation.

The disadvantage of the Optimal Mismatch method, namely its requirement for foreknowledge of the input text statistics, has been addressed by Smith (1991). His adaptive approach starts with an arbitrary symbol comparison order, which is modified as the search proceeds. The algorithm dispenses with the match heuristic and relies solely on the occurrence heuristic. This language independent scheme was found to perform only slightly worse on average than the equivalent static language based search, i.e. a version of the Optimal Mismatch algorithm employing only the occurrence heuristic. The method used to adapt the ordering is to move the pattern symbol position that causes a mismatch either up one position or to the top of the ordering list, thereby giving it a higher precedence in future trials. As noted earlier, Smith's method also involves selecting the maximum of two occurrence heuristic shift values — one for the text symbol aligned with the end of the pattern, and the other for the following text symbol.

4.1.6 Hume and Sunday — Tuned Boyer-Moore and Least Cost

In discussing efficiency considerations, Boyer and Moore (1977) presented an implementation of their algorithm incorporating a 'fast' skip loop involving only the occurrence heuristic. This is used to swiftly discount non-match positions, and only after a successful match has been found for the first pattern symbol to be tested, i.e. x_m, is a 'slow' loop entered in which the other pattern symbols are compared with the text. If a mismatch occurs at this stage, then the maximum of the two heuristics is used to determine the next trial pattern position, and searching is resumed using the fast loop.

Hume and Sunday (1991) have presented a taxonomy of various cognate algorithms based on the structure of this form of the Boyer-Moore search method. Three distinct components of the procedure were considered, namely the fast skip loop, the slow matching loop, and the calculation of the pattern shift to the next trial position in the event of a mismatch whilst in the slow loop.

For each of these three components, various different techniques were considered. For example, initially testing for x_1, x_m, or the least frequent, in terms of the text symbol statistics, pattern symbol were examined for the skip loop. Various comparison orders were investigated for the matching loop — e.g. left to right, right to left, and Sunday's Maximal Shift and Optimal Mismatch orderings. With the exception of the Optimal Mismatch, these were optionally *guarded* by preceding the full match attempt with a single comparison involving a low frequency pattern symbol. The shift calculation stage tests involved the use of the Knuth-Morris-Pratt table, the Boyer-Moore heuristics, Sunday's shift tables, and combinations thereof. It was found, however, that the advantages of combining more than one heuristic were outweighed by the overhead incurred by computing the maximum of the different values.

From empirical tests, based on searches of a bible text, the most effective methods for each of the three stages were identified, and two new procedures were proposed, namely the Tuned Boyer-Moore and the Least Cost algorithms. These both use a guarded left to right comparison for the match loop and Sunday's version of the match heuristic for the shift calculation for a mismatch in the match loop. However, they differ in the methods used in the skip loop — the Tuned Boyer-Moore algorithm uses an optimised version of Boyer and Moore's original 'fast' loop, whereas the Least Cost algorithm initially seeks the pattern symbol incurring the lowest search cost. This cost is based on both the text symbol frequency distribution and the position of the symbol in the pattern, which influences the size of the pattern shift possible.

Implementations of the two algorithms in C are presented in Hume and Sunday (1991), where details of how full copies of the programs and the datasets employed may be obtained via internet *ftp* or email are also given.

A speed improvement by a factor of 4.5 over the original Boyer-Moore algorithm (not in its 'fast-loop' implementation) has been reported for the Tuned Boyer-Moore and Least Cost search procedures. Note that the latter method relies on *a priori* information of the text symbol statistics. Furthermore, it should be pointed out that, in common with Sunday (1990), the algorithms were compared for the case of searching for all occurrences of the pattern in the text, the preprocessing costs were not included in the measurements, and the tests were conducted using relatively small patterns ($m \leqslant 16$).

4.1.7 Harrison

Although not strictly solving the string matching problem as defined earlier, Harrison's approach is included here since it can provide an efficient method of eliminating text strings, with low probabilities of containing a given pattern, when there is no actual match. Also, note that this technique requires the text to be preprocessed.

The basic idea of this substring decision procedure involves representing strings with simpler data objects, which preserve some of their inherent properties. In this method, a string is represented by the set of its substrings of a fixed length k, say. A suitable hashing function is employed that maps the $\mid C \mid^k$ possible distinct substrings (where C is the alphabet) to integers in the interval $[1, p]$. A binary string of length p, known as a *hashed k-signature*, may then be used to represent this set of substrings — the i^{th} bit of the signature is set to '1' only if i is the hashed value of some member of the set of substrings.

A necessary, but not sufficient, condition for a pattern to occur as a substring of a text is that the pattern's signature, x_{sig}, is a subset of that of the text, y_{sig}. If a pattern substring appears nowhere in the text, then the pattern itself cannot be a substring of the text. This means that if x_{sig} has a '1' in a position where y_{sig} does not, then it can be concluded that x is not a substring of y. This test may be performed efficiently using logical operators on the two signatures as follows. A non-zero result from the bitwise *and* of x_{sig} and *not* y_{sig} indicates that there can be no possible match. On the other hand, however, in the event of a zero result, a rigorous search is still required to establish whether or not x is present in the text.

The method may therefore be used to rapidly discount text strings where there can be no match with the pattern. Each line of a large text file could, for example, be stored together with a hashed k-signature to facilitate later pattern searches.

4.1.8 Karp-Rabin

In the Karp-Rabin algorithm, comparison of m-symbol text substrings, $y(i, i+m-1)$, at successive positions, $1 \leqslant i \leqslant n-m+1$, with the pattern x, proceeds in a manner similar to that of the Brute Force approach. The difference being that rather than directly comparing $x(1, m)$ with $y(i, i+m-1)$, it is a single integer comparison that is performed. The integers involved are the hash values for the corresponding m-symbol strings.

The signatures for $x(1, m)$ and $y(1, m)$ are precomputed, and the use of a carefully selected hash function permits the signature for $y(i, i+m-1)$ to be derived easily from that for $y(i-1, i+m-2)$ as the scan of the text proceeds. The actual hash function used is given below.

$$h(k) = k \bmod q \tag{4.6}$$

where q is a large prime. For the calculation, the m-symbol strings are treated as m-digit integers in base c, where $c = \ |C|$ for alphabet C. The component symbols are therefore the (base c) digits of the integer. Adopting a big-endian convention, the most significant digit comes first. Thus, the number, k_i, corresponding to the text substring at position i is given by

$$k_i = y_i c^{m-1} + \ y_{i+1} c^{m-2} + \ \ldots \ + \ y_{i+m-1} \tag{4.7}$$

The number for the next text substring, $y(i+1, i+m)$, k_{i+1} is derived from k_i as follows

$$k_{i+1} = (k_i - y_i c^{m-1})c \ + \ y_{i+m} \tag{4.8}$$

The associative property of the mod function, i.e. $(a+b) \bmod c = (a \bmod c \ + \ b \bmod c) \bmod c$, allows the remainders to be taken after each stage of the calculation. This keeps the partial results small, and yields the same result as that that would be obtained by performing the mod operation only at the final stage of the calculation.

```
— initialisation
i = 1
found = false
d = c^(m-1) mod q
h_x = (x_1 * c^(m-1) + x_2 * c^(m-2) + ... + x_m) mod q
h_y = (y_1 * c^(m-1) + y_2 * c^(m-2) + ... + y_m) mod q
— search
while (i ⩽ n - m + 1) and (not found)
   if (h_x = h_y) and (x(1, m) = y(i, i + m - 1))
      — a successful match
      found = true
   else
      — calculate next signature value
      h_y = (h_y + c * q - y_i * d) mod q
      h_y = (h_y * c + y_(i+m)) mod q
      i = i + 1
if (i > n - m + 1)
   i = 0
```

NOTE that the **and** in the **if** statement is a 'conditional and.'

Figure 4.9: Karp-Rabin string matching

As the hash function values are not unique, the equality of two signatures is a necessary, but not sufficient, condition for the equality of their corresponding strings. Therefore, when a potential match is detected, the two strings must then be compared directly, to check for a signature *collision*. For a large hash table, i.e. a large value of q, the chances of such collisions are slight. One of the algorithms presented by Karp and Rabin (1987) further reduces the probability of a large number of collisions by randomly reselecting the prime q should a collision occur, reinitialising and then continuing with the search. The theoretical worst case involves m symbol comparisons being forced at each stage, giving an overall execution time of $O(mn)$. In practice, however, the signatures at each stage would be expected, on the whole, to be unequal. In these circumstances, each text position is processed in constant time, giving an $O(m+n)$ expected performance. The algorithm is given in Figure 4.9. Note that the initial calculation of h_x and h_y may be performed in the following manner.

```
h_x = 0
for j = 1 to m
   h_x = (h_x * c + x_j) mod q
```

The value of d, used in the incremental signature calculation, may also be initially computed iteratively, again exploiting the associativity of mod. In the calculation of the next signature value, $c * q$ is added to ensure that the partial result remains positive. On completion of the search procedure, i gives the position of the pattern in the text, or is equal to 0 when no match has been found.

The prime number q should be as large as possible, consistent with $(c+1)q$ not causing arithmetic overflow (this is the maximum partial result in the next signature computation). Sedgewick (1983) uses a value of 33554393 (for $c = 32$), and Pirklbauer (1992) suggests 8355967 (for $c = 256$). Gonnet and Baeza-Yates (1991) present an implementation of the algorithm obviating the mod operations by virtue of the implicit modular arithmetic of the target hardware.

4.2 String Distance and Longest Common Subsequence

4.2.1 Wagner-Fischer

In the dynamic programming method of computing the distance between two strings, the distances between longer and longer prefixes of the strings are successively evaluated from previous values until the final result is obtained. The precise details of this procedure are given below.

Let the distance between the prefixes of strings x and y, of lengths i and j respectively, be denoted by $d_{i,j}$, i.e.

$$d_{i,j} = d(x(1,i),\ y(1,j)) \tag{4.9}$$

Further, let the weighting for the cost of transforming symbol a into symbol b be denoted by $w(a,b)$. Thus, $w(a,b)$ is the cost of a symbol substitution if $a \neq b$, $w(a,\epsilon)$ is the cost of deleting a, and $w(\epsilon,b)$ is the cost of inserting b. Note that when d is the Levenshtein distance, we have the following.

$$\begin{aligned} w(a,\epsilon) &= 1 \\ w(\epsilon,b) &= 1 \\ w(a,b) &= 1 \ \text{ if } a \neq b \\ &= 0 \ \text{ otherwise} \end{aligned} \tag{4.10}$$

During the calculation procedure, the values of $d_{i,j}$ are recorded in an $(m+1)\times(n+1)$ array, and are computed using the following recurrence relation.

$$d_{i,j} = \min\{d_{i-1,j} + w(x_i,\epsilon),\ d_{i,j-1} + w(\epsilon,y_j),\ d_{i-1,j-1} + w(x_i,y_j)\} \tag{4.11}$$

This relationship is derived as follows. If the cost of transforming $x(1,i-1)$ into $y(1,j)$ is known, then that of transforming $x(1,i)$ into $y(1,j)$ may be obtained by adding the cost of deleting x_i. Similarly, if the cost of transforming $x(1,i)$ into $y(1,j-1)$ is known, then that of transforming $x(1,i)$ into $y(1,j)$ may be obtained by adding the cost of inserting y_j. Finally, if the cost of transforming $x(1,i-1)$ into $y(1,j-1)$ is known, then that of transforming $x(1,i)$ into $y(1,j)$ may be obtained by adding the cost of replacing x_i by y_j. Recalling that the distance $d_{i,j}$ is the minimum cost of transforming $x(1,i)$ into $y(1,j)$, it is necessary to select the cheapest of the three operations described above in order to arrive at the correct value for $d_{i,j}$.

Before performing the $d_{i,j}$ calculations, the array boundary conditions must be set up. For the first column of the array, the value of $d_{i,0}$ is equal to the sum of the costs of deleting the first i symbols of x. Similarly, for the first row, the value of $d_{0,j}$ is given by the sum of the costs of inserting the first j symbols of y. Thus, we have the following:

$$\begin{aligned} d_{0,0} &= 0 \\ d_{i,0} &= \sum_{k=1}^{i} w(x_i, \epsilon) \;\text{ for } 1 \leqslant i \leqslant m \\ d_{0,j} &= \sum_{k=1}^{j} w(\epsilon, j_k) \;\text{ for } 1 \leqslant j \leqslant n \end{aligned} \tag{4.12}$$

For the case of the Levenshtein distance, it therefore follows that $d_{i,0} = i$ and $d_{0,j} = j$. The calculation procedure is demonstrated in the following example which shows the array produced in the evaluation of the Levenshtein distance between the strings `preterit` and `zeitgeist`. From this, it may be seen that the distance between these strings, i.e. $d_{8,9}$, is equal to 6.

	j	0	1	2	3	4	5	6	7	8	9
i			z	e	i	t	g	e	i	s	t
0		0	1	2	3	4	5	6	7	8	9
1	p	1	1	2	3	4	5	6	7	8	9
2	r	2	2	2	3	4	5	6	7	8	9
3	e	3	3	2	3	4	5	5	6	7	8
4	t	4	4	3	3	3	4	5	6	7	7
5	e	5	5	4	4	4	4	4	5	6	7
6	r	6	6	5	5	5	5	5	5	6	7
7	i	7	7	6	5	6	6	6	5	6	7
8	t	8	8	7	6	5	6	7	6	6	6

The algorithm to compute the distance array, as developed by Wagner and Fischer, is given in Figure 4.10. It may be seen that the border initialisation stage comprises $1 + m + n$ assignments, and that the main loop is iterated mn times. The time complexity of the method is therefore $O(mn)$.

The editing sequence taking x to y may be obtained by way of a structure known as a *trace*. A trace from x to y may be visualised as comprising string x placed above string y, with edges joining symbols in x to symbols in y, such that any symbol is touched by at most one edge and no two edges cross each other. Representing the edge from x_i to y_j by the ordered pair of integers (i, j), a trace from x to y may be defined formally as a set of such ordered pairs satisfying the following conditions.

$$\begin{aligned} &(a) \quad 1 \leqslant i \leqslant m, \;\; 1 \leqslant j \leqslant n \\ &(b) \quad \text{for distinct edges } (i_1, j_1),\; (i_2, j_2) \\ &\qquad\quad i_1 \neq i_2 \text{ and } j_1 \neq j_2, \quad i_1 < i_2 \;\Leftrightarrow\; j_1 < j_2 \end{aligned} \tag{4.13}$$

— initialise array borders
$d_{0,0} = 0$
for $i = 1$ **to** m
  $d_{i,0} = d_{i-1,0} + w(x_i, \epsilon)$
for $j = 1$ **to** n
  $d_{0,j} = d_{0,j-1} + w(\epsilon, y_j)$
— calculate $d_{i,j}$
for $i = 1$ **to** m
  for $j = 1$ **to** n
    $d_{i,j} = \min\{d_{i-1,j} + w(x_i, \epsilon),\ d_{i,j-1} + w(\epsilon, y_j),\ d_{i-1,j-1} + w(x_i, y_j)\}$

Figure 4.10: Wagner and Fischer string distance calculation

An editing sequence may be obtained from a trace as follows. Symbols in x untouched by edges are deleted, whereas those untouched in y are inserted. For an edge in the trace, (i, j), if $x_i \neq y_j$, then x_i is replaced by y_j, otherwise no edit is required. Returning to the previous example, a least cost trace from `preterit` to `zeitgeist` is given below.

```
i        1 2 3 4 5 6 7 8
x_i      p r e t e r i t
         |   / |   \ |   \
y_j      z e i t g e i s t
j        1 2 3 4 5 6 7 8 9
```

A minimal cost trace, and hence a least cost editing sequence, from x to y may be obtained from the completed distance array. An algorithm to print out the ordered pairs of the trace is given in Figure 4.11. The procedure starts with $d_{m,n}$ and works back until either i or j equals 0. The method is to determine from which of its neighbours $d_{i,j}$ was derived, and to move to that position in the array. If $d_{i,j}$ was derived from $d_{i-1,j-1}$, then the ordered pair (i, j) is output — corresponding to an edge in the trace, i.e. either a substitution or a match depending on whether $x_i \neq y_j$ or not respectively. No output is produced for the case where $d_{i,j}$ was derived from either $d_{i-1,j}$ or $d_{i,j-1}$, as deletions and insertions correspond to symbols untouched by edges in the trace. Since either i or j, or both, is decremented on each pass of the loop, the maximum number of iterations is $m+n$. The least cost trace is thus determined in $O(m + n)$ time.

Returning once more to the previous example, the above procedure yields the following trace, T (in reverse order from that output by the algorithm).

$$T = \{(1,1),\ (3,2),\ (4,4),\ (5,6),\ (7,7),\ (8,9)\}$$

Note that it is now a fairly straightforward matter to obtain an $\mathrm{lcs}(x, y)$ from the least cost trace from x to y. The components of $\mathrm{lcs}(x, y)$ are the x_i symbols, or equivalently the y_j symbols, such that $(i, j) \in T$ and $x_i = y_j$. So, for our example, we have:

$$\mathrm{lcs}(x, y) = x_3x_4x_5x_7x_8 = y_2y_4y_6y_7y_9 = \texttt{eteit}$$

$i = m$
$j = n$
while $(i > 0)$ **and** $(j > 0)$
  if $d_{i,j} = d_{i-1,j} + w(x_i, \epsilon)$
    $i = i - 1$
  else if $d_{i,j} = d_{i,j-1} + w(\epsilon, y_j)$
    $j = j - 1$
  else
    print('(i, j)')
    $i = i - 1$
    $j = j - 1$

Figure 4.11: Wagner and Fischer least cost trace computation

Finally, a particular point made by Wagner and Fischer worth noting is that for spelling correction applications, it might be useful to have a cost weighting function for the edit $a \to b$, $w(a, b)$, dependent on the particular symbols a and b. For the standard `qwerty` keyboard, for example, it is much more likely that an `e` would be miskeyed as a `w` than, say, a `p`. Using this criterion, `wast` would, for example, be deemed to be closer to `east` than to `past`.

4.2.2 Hirschberg

Considering the distance array of the above method, it is apparent that the space requirement of the Wagner-Fischer approach is $O(mn)$. In order to avoid potential storage problems when dealing with long strings, Hirschberg (1975) developed a linear space version of the algorithm.

Hirschberg's objective was to determine an $\mathrm{lcs}(x, y)$ of two strings x and y. Rather than evaluate string distances in the dynamic programming array, it is therefore the lengths of longest common subsequences of progressively longer string prefixes that are determined. Let this quantity be denoted by $l_{i,j}$, i.e.

$$l_{i,j} = \ |\mathrm{lcs}(x(1, i), y(1, j))| \tag{4.14}$$

For a given metric, there is a fixed relationship between $l_{i,j}$ and $d_{i,j}$. Consider, for example, the edit distance, defined by the following cost weightings.

$$\begin{aligned} w(a, \epsilon) &= 1 \\ w(\epsilon, b) &= 1 \\ w(a, b) &= 2 \quad \text{if } a \neq b \\ &= 0 \quad \text{otherwise} \end{aligned} \tag{4.15}$$

From a minimal cost trace from x to y, it is apparent that the edit distance $d(x, y)$, is related to $|\mathrm{lcs}(x, y)|$ as derived below, where *del*, *ins* and *sub* are the number of symbol deletions, insertions and substitutions respectively.

$$d(x, y) \ = \ del \ + \ ins \ + \ 2sub$$

— initialise array borders
$l_{0,0} = 0$
for $i = 1$ **to** m
  $l_{i,0} = 0$
for $j = 1$ **to** n
  $l_{0,j} = 0$
— calculate $l_{i,j}$
for $i = 1$ **to** m
  for $j = 1$ **to** n
    if $x_i = y_j$
      $l_{i,j} = l_{i-1,j-1} + 1$
    else
      $l_{i,j} = \max\{l_{i-1,j},\ l_{i,j-1}\}$

Figure 4.12: Calculation of $|\,\mathrm{lcs}(x, y)\,|$

$$\begin{aligned} &= m - (|\,\mathrm{lcs}(x,y)\,| + sub) \ + \ n - (|\,\mathrm{lcs}(x,y)\,| + sub) \ + \ 2sub \\ &= m + n - 2\,|\,\mathrm{lcs}(x,y)\,| \end{aligned} \tag{4.16}$$

In this case, the relationship between $l_{i,j}$ and $d_{i,j}$ is therefore as follows.

$$l_{i,j} = (i + j - d_{i,j})/2 \tag{4.17}$$

Employing this transformation, the dynamic programming procedure for $l_{i,j}$ may be derived from that for $d_{i,j}$, and is given in Figure 4.12. Since the length of an lcs of any string and the empty string is 0, the array border values are given by $l_{i,0} = l_{0,j} = 0$. As with the $d_{i,j}$ calculations, the values of $l_{i,j}$ are built up from previous results. When at position (i, j), i.e. when considering prefixes $x(1, i)$ and $y(1, j)$, if $x_i = y_j$ then we obtain a new lcs by appending this symbol to the current lcs of $x(1, i-1)$ and $y(1, j-1)$, and thus $l_{i,j} = l_{i-1,j-1}+1$. Otherwise the length of the current partial lcs is simply the maximum of the previous neighbouring values.

The storage requirements of the above algorithm may be reduced by noting that only row $i-1$ of the matrix is required in the evaluation of row i. This fact is exploited in the algorithm given in Figure 4.13. This returns vector ll as output, where $ll_j = l_{m,j}$. A $2 \times (n + 1)$ array, h, is employed, whose rows 0 and 1 act as rows $i - 1$ and i of array l, respectively. Row 1 is therefore shifted up into row 0 prior to each new 'row i' calculation.

As before, the **if** statement is executed exactly mn times, giving a time complexity of $O(mn)$. The input and output arrays require $m+n+(n+1)$ locations, with $2(n+1)$ storage locations allocated to array h. The space complexity is therefore linear in m and n, i.e. $O(m + n)$. This method may be used in an algorithm to determine an actual $\mathrm{lcs}(x, y)$ in linear space, as described below.

The general idea is to divide recursively string x in half, and for each half, $x1$ and $x2$, find a corresponding prefix and suffix, $y1$ and $y2$, of y such that the lcs of $x1$ and $y1$ concatenated with that of $x2$ and $y2$ is equal to an lcs of the complete strings x and y, i.e.

$$\mathrm{lcs}(x1, y1) \cdot \mathrm{lcs}(x2, y2) = \mathrm{lcs}(x, y) \tag{4.18}$$

```
lcs_length(m, n, x, y, ll)
    — initialise
    for j = 0 to n
        h[1,j] = 0
    — calculate h[1,j]
    for i = 1 to m
        — shift row up
        for j = 0 to n
            h[0,j] = h[1,j]
        for j = 1 to n
            if x[i] = y[j]
                h[1,j] = h[0,j-1] + 1
            else
                h[1,j] = max{h[1,j-1], h[0,j]}
    — copy result to output vector
    for j = 0 to n
        ll[j] = h[1,j]
```

Figure 4.13: Reduced space calculation of $|\mathrm{lcs}(x, y)|$

The problem may thus be decomposed recursively until trivial subproblems are obtained.

Let the length of an lcs of the suffixes $x(i+1, m)$ and $y(j+1, n)$ be denoted by $l^*_{i,j}$, i.e.

$$l^*_{i,j} = \quad |\mathrm{lcs}(x(i+1, m), y(j+1, n))| \tag{4.19}$$

For $0 \leqslant j \leqslant n$, values of $l_{i,j}$ are the lengths of lcs's of the prefix $x(1, i)$ and various prefixes of y. Similarly, for $0 \leqslant j \leqslant n$, values of $l^*_{i,j}$ are the lengths of lcs's of the reversed suffix $x_R(m, i+1)$ and various prefixes of the reversed string y_R. The following theorem permits the appropriate prefix and suffix of y to be found when x is bisected, i.e. when i is taken to be $\lfloor m/2 \rfloor$. The theorem states that when x is divided into two parts at any point, then the maximum value, over all divisions of y into two sections, of the sum of the lengths of an lcs of the first sections of x and y and that of the latter sections is equal to the length of an lcs of the complete strings x and y.

Theorem

Defining M_i as

$$M_i = \max_{0 \leqslant j \leqslant n} \{l_{i,j} + l^*_{i,j}\} \tag{4.20}$$

then,

$$M_i = l_{m,n} \quad \text{for } 0 \leqslant i \leqslant m \tag{4.21}$$

Proof

Let

$$\begin{aligned} M_i &= l_{i,j} + l^*_{i,j} \quad \text{for } j = j_0, \text{ say} && (4.22) \\ s_{i,j_0} &\text{ be any lcs}(x(1,i), y(1,j_0)) && (4.23) \\ s^*_{i,j_0} &\text{ be any lcs}(x(i+1,m), y(j_0+1,n)) && (4.24) \end{aligned}$$

then $c = s_{i,j_0} \cdot s^*_{i,j_0}$ is a common subsequence of x and y, and has length M_i. Thus,

$$l_{m,n} \geqslant M_i \tag{4.25}$$

Let $s_{m,n}$ be any lcs(x, y), then $s_{m,n}$ is a subsequence of y that is equal to $s1 \cdot s2$, where $s1$ is a subsequence of $x(1,i)$ and $s2$ is a subsequence of $x(i+1,m)$. So, there exists a value of j, j_1 say, such that $s1$ is a subsequence of $y(1,j_1)$ and $s2$ is a subsequence of $y(j_1+1,n)$. The lengths of $s1$ and $s2$ satisfy the following.

$$\begin{aligned} |\,s1\,| &\leqslant |\,\text{lcs}(x(1,i), y(1,j_1))\,| \\ &\leqslant l_{i,j_1} \quad \text{from (4.14)} && (4.26) \\ |\,s2\,| &\leqslant |\,\text{lcs}(x(i+1,m), y(j_1+1,n))\,| \\ &\leqslant l^*_{i,j_1} \quad \text{from (4.19)} && (4.27) \end{aligned}$$

Thus we have,

$$\begin{aligned} l_{m,n} &= |\,s_{m,n}\,| \\ &= |\,s1\,| \ + \ |\,s2\,| \\ &\leqslant l_{i,j_1} + l^*_{i,j_1} \quad \text{from (4.26) and (4.27)} \\ &\leqslant M_i \quad \text{from (4.20)} && (4.28) \end{aligned}$$

From inequalities (4.25) and (4.28), we have

$$M_i = l_{m,n} \square \tag{4.29}$$

Figure 4.14 gives the overall algorithm, incorporating the above theorem, to determine lcs(x, y). String c is returned as output, and is equal to an lcs of input strings x and y. The procedure returns with either the empty string or a single-symbol lcs for a trivial problem. Otherwise, string x is sliced in half and the lengths of the lcs of its lower half and varying length prefixes of y, and that of its upper half and varying length suffixes of y are found. The theorem is then employed in order to find the first position in y, denoted here by k, such that the lcs of

```
lcs(m, n, x, y, c)
    — trivial case
    if n = 0
        c = ε
    else if m = 1
        if ∃j, 0 ⩽ j ⩽ n, such that x_1 = y_j
            c = x_1
        else
            c = ε
    — non-trivial case, split the problem
    else
        i = ⌊m/2⌋
        — calculate l_{i,j} and l*_{i,j} for 0 ⩽ j ⩽ n
        lcs_length(i, n, x(1,i), y(1,n), l1)
        lcs_length(m − i, n, x_R(m,i+1), y_R(n,1), l2)
        — find j such that l_{i,j} + l*_{i,j} = l_{m,n}
        M = max_{0⩽j⩽n}{l1_j + l2_{n−j}}
        k = min j such that l1_j + l2_{n−j} = M
        — solve simpler problems
        lcs(i, k, x(1,i), y(1,k), c1)
        lcs(m − i, n − k, x(i+1,m), y(k+1,n), c2)
        — concatenate results to produce output
        c = c1 · c2
```

Figure 4.14: Hirschberg calculation of $\mathrm{lcs}(x, y)$

the first half of x and $y(1,k)$ concatenated with that of the second half of x and $y(k+1,n)$ is equal to the desired lcs(x,y). It thus only remains to use this value of k in two recursive calls to the procedure to obtain the required subproblem lcs's, and then to concatenate them to provide the output.

The linear space characteristic of this algorithm may be demonstrated as follows. Strings x and y may be held in global storage, and substring parameters may be passed by transferring as arguments pointers to the start and end of the relevant substrings. As shown earlier, calls to *lcs_length* require temporary storage proportional to m and n. Not counting recursive calls, the storage requirement of procedure *lcs* is constant. It can be shown that during execution, there is a total of $2m-1$ call to *lcs*. The overall storage requirements are therefore linear in m and n, i.e. the space complexity of this method of determining an lcs is $O(m+n)$. Hirschberg (1975) also provided a temporal analysis of the execution of the algorithm and has shown its time complexity to be $O(mn)$.

4.2.3 Hunt-Szymanski

Hunt and Szymanski's method of extracting an lcs from two input strings, x and y, is equivalent to finding a maximal monotonically increasing path in the graph composed of points (i,j) such that $x_i = y_j$. This will be illustrated by an example once the actual method has been described.

For convenience, it shall be assumed that the lengths of the input strings are equal, i.e. $|\,x\,| \;=\; |\,y\,| \;= n$. The final algorithm may, however, be modified easily in order to accommodate unequal length strings. The number of ordered pairs of positions within x and y at which the respective symbols match, that is the cardinality of the set $\{(i,j)\ :\ x_i = y_j\}$ shall be denoted by r.

Crucial to the method is an array of $k_{i,l}$ values, defined as follows

$$k_{i,l} = \min j \text{ such that } l_{i,j} = l \tag{4.30}$$

where $l_{i,j}$ is defined in (4.14).

The value of $k_{i,l}$ thus gives the shortest length of prefix of y having a subsequence of length l in common with the prefix of x of length i. This is illustrated by considering the case where $x =$ `preterit` and $y =$ `zeitgeist`, for example, for which $k_{5,1} = 2$, $k_{5,2} = 4$, $k_{5,3} = 6$, and $k_{5,4}$ and $k_{5,5}$ are not defined.

Since, by definition, $k_{i,l}$ is a minimal value, it follows that the last symbol of a subsequence of length l common to the prefixes $x(1,i)$ and $y(1,k_{i,l})$ is equal to $y_{k_{i,l}}$. Thus, the length of the lcs of $x(1,i)$ and $y(1,k_{i,l}-1)$ is equal to $l-1$. Consequently, $k_{i,l-1} \leqslant k_{i,l}-1$, i.e. $k_{i,l-1} < k_{i,l}$. This shows that the values of $k_{i,l}$ in each row of the array must increase monotonically.

The values of k may be computed iteratively from previous values in the array. In particular, the value of $k_{i+1,l}$ may be derived from those of $k_{i,l-1}$ and $k_{i,l}$. But before showing how this is done, the range of values within which $k_{i+1,l}$ must fall shall first be examined.

If $x(1,i)$ and $y(1,k_{i,l})$ have a common subsequence of length l, then so too will $x(1,i+1)$ and $y(1,k_{i,l})$. Thus

$$k_{i+1,l} \leqslant k_{i,l} \tag{4.31}$$

By definition, $x(1,i+1)$ and $y(1,k_{i+1,l})$ have an lcs of length l. The lcs of prefixes one symbol shorter, namely $x(1,i)$ and $y(1,k_{i+1,l}-1)$, will therefore be shorter than l by at most one symbol. Thus, $k_{i,l-1} \leqslant k_{i+1,l}-1$, or

$$k_{i,l-1} < k_{i+1,l} \tag{4.32}$$

Combining (4.31) and (4.32) gives the following bounds for $k_{i+1,l}$

$$k_{i,l-1} < k_{i+1,l} \leqslant k_{i,l} \tag{4.33}$$

The actual rule for calculating $k_{i+1,l}$ from $k_{i,l-1}$ and $k_{i,l}$ is given below.

$$k_{i+1,l} = \begin{cases} \min j \text{ such that } x_{i+1} = y_j \text{ and } k_{i,l-1} < j \leqslant k_{i,l} \\ k_{i,l} \text{ if no such } j \text{ exists} \end{cases} \tag{4.34}$$

This may be shown to be correct as follows. Firstly, consider the case for which no suitable j exists. Since $k_{i+1,l}$ is a minimal value, the last component of any length-l subsequence common to $x(1,i+1)$ and $y(1,k_{i+1,l})$ must be equal to $y_{k_{i+1,l}}$. Recalling that $k_{i+1,l}$ must lie within the bounds given in (4.33), it follows that $y_{k_{i+1,l}} \neq x_{i+1}$, otherwise $k_{i+1,l}$ would be the required value of j. It then follows that the same length-l subsequence is common to $x(1,i)$ and $y(1,k_{i+1,l})$, implying that $k_{i,l} \leqslant k_{i+1,l}$. Combining this with (4.33) gives the result that, when no suitable j exists, $k_{i+1,l} = k_{i,l}$.

Secondly, consider the case where there is a j such that $x_{i+1} = y_j$ and $k_{i,l-1} < j \leqslant k_{i,l}$. String prefixes $x(1,i+1)$ and $y(1,j)$ will have a common subsequence of length l equal to the one of length $l-1$, common to $x(1,i)$ and $y(1,k_{i,l-1})$, with symbol x_{i+1} appended to the end (since $x_{i+1} = y_j$). Thus, $k_{i+1,l} \leqslant j$. It now remains to determine whether, as is desired, the equality holds, or whether the value of $k_{i+1,l}$ is actually less than our value of j. By assuming the latter and obtaining a contradiction, the former may be proven. We proceed as follows. Assume that $k_{i+1,l} < j$. Again, the last symbol of the length-l subsequence common to $x(1,i+1)$ and $y(1,k_{i+1,l})$ must be $y_{k_{i+1,l}}$. Since both $k_{i+1,l}$ and j are constrained to lie in the same interval (from (4.33) and (4.34)), we know that $y_{k_{i+1,l}} \neq x_{i+1}$ as j is the minimum value in the permitted range such that $y_j = x_{i+1}$ and we are assuming that $k_{i+1,l} < j$. It then follows that $x(1,i)$ and $y(1,k_{i+1,l})$ also have a common subsequence of length l, implying that $k_{i,l} \leqslant k_{i+1,l}$. Taken in conjunction with (4.33), this means that $k_{i,l} = k_{i+1,l}$. Our assumption was that $k_{i+1,l} < j$. Thus, from (4.34), $k_{i+1,l} < k_{i,l}$, which obviously contradicts the previous result. This shows that our value of j is, in fact, equal to the required $k_{i+1,l}$.

An algorithm to determine $|\,\mathrm{lcs}(x,y)\,|$ using the $k_{i,l}$ values is given in Figure 4.15. Vector kk is used as the i^{th} row of array k. Thus, at the start of each iteration of i, $kk_l = k_{i-1,l}$ for $0 \leqslant l \leqslant n$, and, on completion of the iteration, $kk_l = k_{i,l}$ for $0 \leqslant l \leqslant n$. Note that at the outset, kk_0 is initialised to 0 and the rest of kk to $n+1$, signifying undefined values of $k_{i,l}$. Only a single row is required for the calculations, since the values down a column either remain unaltered or are decreased and the values along a row remain in a monotonically increasing order.

— initialise
$kk_0 = 0$
for $l = 1$ **to** n
  $kk_l = n + 1$
— calculate $|\text{lcs}(x, y)|$
for $i = 1$ **to** n
  for $j = n$ **downto** 1
    if $x_i = y_j$
      find l such that $kk_{l-1} < j \leqslant kk_l$
      $kk_l = j$
print max$l$ such that $kk_l \neq n + 1$

Figure 4.15: Hunt-Szymanski calculation of $|\text{lcs}(x, y)|$

It is important that the j loop counts down from n to 1. Consider the case where x_i matches several symbols in y, y_{j_1}, y_{j_2}, $\ldots$, y_{j_p} say, such that $k_{i-1,l-1} < j_1 < j_2 < \ldots < j_p \leqslant k_{i-1,l}$. From (4.34), it can be seen that the correct value for $k_{i,l}$ is therefore j_1. By iterating downwards for j, $k_{i,l}$ will be assigned successively smaller values j_p, j_{p-1}, $\ldots$, j_1, ending in the final assignment of the required value. If, however, the iteration for j were carried out forwards, then $k_{i,l}$ would first of all be given the correct value but then $k_{i,l+1}$ would be set to j_2, $k_{i,l+2}$ to j_3, and so on up to $k_{i,l+p-1}$, which would not be correct.

At the end of the procedure, the value of $|\text{lcs}(x, y)|$ is given by the maximum l for which $k_{n,l}$ is defined. As the values in kk increase monotonically, the 'find' operation may be implemented in $O(\log n)$ time by performing a binary search. Since the **if** statement is executed exactly n^2 times, it follows that the algorithm runs in $O(n^2 \log n)$ time in the worst case. Note that for unequal length strings, the bound for the kk initialisation and the outer 'i' loop would be set to m rather than n, where $m < n$. The results of the procedure are demonstrated below, which shows the $k_{i,l}$ values for $x =$ `preterit` and $y =$ `zeitgeist`. It may be seen from the final row that $|\text{lcs}(\texttt{preterit}, \texttt{zeitgeist})| = 5$.

i	l	0	1	2	3	4	5	6	7	8
1	p	0	10	10	10	10	10	10	10	10
2	r	0	10	10	10	10	10	10	10	10
3	e	0	2	10	10	10	10	10	10	10
4	t	0	2	4	10	10	10	10	10	10
5	e	0	2	4	6	10	10	10	10	10
6	r	0	2	4	6	10	10	10	10	10
7	i	0	2	3	6	7	10	10	10	10
8	t	0	2	3	4	7	9	10	10	10

The exhaustive testing for matching symbols in the above algorithm is rather inefficient and may be avoided by preprocessing the strings so as to obtain a prior record of the positions at which they match. An array of pointers to linked lists, *matchlist*, may thus be constructed such that *matchlist*$[i]$ provides a linked list of positions, j, in descending order, at which

— create linked lists
for $i = 1$ **to** n
    $matchlist[i] = (j_1,\ j_2, \ldots,\ j_p)$ such that $j_1 > j_2 > \cdots > j_p$ and $x_i = y_{j_q}$ for $1 \leqslant q \leqslant p$
— initialise
$kk_0 = 0$
for $l = 1$ **to** n
    $kk_l = n + 1$
$link_0 =$ null
— calculate successive kk values
for $i = 1$ **to** n
    for j **in** $matchlist[i]$
        find l such that $kk_{l-1} < j \leqslant kk_l$
        if $j < kk_l$
            $kk_l = j$
            $link_l = newnode(i,\ j,\ link_{l-1})$
— extract lcs in reverse order
$l = \max l$ such that $kk_l \neq n + 1$
$pointer = link_l$
while $pointer \neq$ null
    print (i, j) pair pointed to by $pointer$
    advance $pointer$

Figure 4.16: Hunt-Szymanski calculation of lcs(x, y)

$x_i = y_j$. The same physical list may be used for repeated instances of a symbol in x. For example, the actual lists for the case of $x =$ `preterit` and $y =$ `zeitgeist` are as follows.

$$\begin{aligned}
matchlist[1] &= () \\
matchlist[2] &= () \\
matchlist[3] &= (6,\ 2) \\
matchlist[4] &= (9,\ 4) \\
matchlist[5] &= matchlist[3] \\
matchlist[6] &= () \\
matchlist[7] &= (7,\ 3) \\
matchlist[8] &= matchlist[4]
\end{aligned}$$

The final algorithm, which incorporates this improvement and which also extracts an lcs of the two strings, is given in Figure 4.16. The lcs is recovered by utilising a backtracking device employed whilst calculating the successive kk values. Whenever kk_l is defined, $link_l$ is set to point to the head of a list of (i, j) pairs defining a common subsequence of length l. This is implemented by using the *newnode* procedure, which creates a node containing the current (i, j) pair together with a pointer to the previous node in the list and returns a pointer to the node just created.

The preprocessing stage may be implemented by sorting the symbols of each string, keeping note of their original positions, and then merging the sorted strings to create the *matchlist*'s. By using the heapsort algorithm, this phase may be performed in $O(n \log n)$ time and $O(n)$ space. The initialisation stage takes $O(n)$ time. During the calculation of the kk values, the inner loop is iterated a total of r times. For each iteration, a binary search and a few other constant-time operations are performed. This stage therefore takes $O(r + r \log n)$ time and involves the creation of at most r new nodes. Finally, the actual extraction of an lcs takes $O(|\operatorname{lcs}(x, y)|)$ time, which is, of course, $O(n)$. The overall time and space complexities of the algorithm are therefore $O((r + n) \log n)$ and $O(r + n)$ respectively. Again, the more general case of unequal length strings is easily catered for by setting the bounds for the kk initialisation and the 'i' loops to m rather than n.

For our example of computing lcs(`preterit`, `zeitgeist`), the above procedure produces the following list of (i, j) pairs for the lcs (in reverse order).

$$(8,9),\ (7,7),\ (5,6),\ (4,4),\ (3,2)$$

This is illustrated below, where the nodes indicate the points (i, j) such that $x_i = y_j$. The lcs is formed by the symbols corresponding to the filled-in nodes, namely `eteit`, and is thus equivalent to a maximal monotonically increasing path in the graph.

	j	1	2	3	4	5	6	7	8	9
i		z	e	i	t	g	e	i	s	t
1	p									
2	r									
3	e		●				○			
4	t				●					○
5	e		○				●			
6	r									
7	i			○				●		
8	t				○					●

Apostolico and Guerra (1987) have developed a variation of the Hunt-Szymanski algorithm, which overcomes the latter's degradation in behaviour to worse than quadratic in the worst case. Apostolico and Guerra's variant has a temporal complexity of $O(m \log n + t \log(2mn/t))$, which is at worst $O(mn)$. Here $t \leqslant r$ is the number of dominant matches between the strings. A k-dominant match is an ordered pair (i, j) such that $x_i = y_j$, $l_{i,j} = k$, and it is the first such point occurring in the submatrix $l_{(1,\ldots,i),(1,\ldots,j)}$ ($l_{i,j}$ is defined in (4.14)). A match (i, j) is thus dominant if the occurrences in the appropriate string prefixes of every $\operatorname{lcs}(x(1, i), y(1, j))$ have x_i and y_j as their final symbols. Apostolico and Guerra's variant involves an efficient means of finding the dominant matches and employs data structures known as characteristic trees to represent the various lists required by the algorithm.

An upper bound, formalised by Jacobson and Vo (1992), for the total number of dominant matches, t, between two strings, is obtained as follows. The k-dominant matches may be sorted in increasing order of i to give $\{(i_1, j_1),\ (i_2, j_2),\ \ldots, (i_{t_k}, j_{t_k})\}$, where t_k denotes the number of k-dominant matches, and $i_1 < i_2 < \cdots < i_{t_k}$, $j_1 > j_2 > \cdots > j_{t_k}$. Since the

i's and j's are monotonically increasing and decreasing, respectively, we have the following inequalities, where $1 \leqslant p \leqslant t_k$.

$$\begin{aligned} i_p - i_1 &\geqslant p - 1 \\ j_p - j_{t_k} &\geqslant t_k - p \end{aligned} \tag{4.35}$$

The edit distance between prefixes $x(1,i)$ and $y(1,j)$, and the length of their lcs are related as follows.

$$\begin{aligned} d_{i_p,j_p} &= i_p + j_p - 2l_{i_p,j_p} \quad \text{from (4.16)} \\ &= i_p + j_p - 2k \end{aligned} \tag{4.36}$$

Since the match (i_1, j_1) is k-dominant, we have that $i_1 \geqslant k$, and also since match (i_{t_k}, j_{t_k}) is k-dominant, we have that $j_{t_k} \geqslant k$. Combining this with (4.36) gives the following.

$$\begin{aligned} d_{i_p,j_p} &\geqslant i_p + j_p - (i_1 + j_{t_k}) \\ &\geqslant (i_p - i_1) + (j_p - j_{t_k}) \\ &\geqslant (p-1) + (t_k - p) \quad \text{from (4.35)} \\ &\geqslant t_k - 1 \end{aligned} \tag{4.37}$$

Let the value of k having the greatest number of k-dominant matches be denoted by k'. An lcs(x,y) built using only dominant matches must incorporate one of these k'-dominant matches, (i_p, j_p), say. The distance between the complete strings x and y must then be greater than or equal to that between the prefixes $x(1,i)$ and $y(1,j)$, i.e.

$$\begin{aligned} d_{m,n} &\geqslant d_{i_p,j_p} \\ \Rightarrow d_{m,n} + 1 &\geqslant t_{k'} \quad \text{from (4.37)} \end{aligned} \tag{4.38}$$

Since $t_{k'}$ is a maximal number of dominant matches, i.e. $t_{k'} \geqslant t_l$ for $1 \leqslant l \leqslant l_{m,n}$, we have the following.

$$t \leqslant l_{m,n} \cdot t_{k'} \tag{4.39}$$

Combining this with inequality (4.38) gives the following final result for the upper bound for the total number of dominant matches.

$$t \leqslant l_{m,n}(d_{m,n} + 1) \tag{4.40}$$

The number of dominant matches can be much less than the total number of matches, especially when the distance between the strings is small. The worst case bound for t is obtained as follows. Substituting for $d_{m,n}$, from (4.16), gives the following.

$$t \leqslant l_{m,n}(m + n - 2l_{m,n} + 1) \tag{4.41}$$

Considering the case of equal length strings, this bound is maximised when the lcs length is $(2n+1)/4$, with a corresponding value of $(2n+1)^2/8$. Recall that the maximum number of matches is n^2. It may thus be seen that, in the worst case, the total number of matches is asymptotically twice that of the dominant matches.

4.2.4 Masek-Paterson

Masek and Paterson (1980) applied the approach of Arlazarov, Dinic, Kronod and Faradzev (1970) — the 'Four Russians' — to Wager and Fischer's string distance procedure to obtain an $O(n^2/\log n)$ time algorithm. Their resulting technique is described below.

The distance matrix, d, is broken up into a number of square submatrices with overlapping edge vectors. The (i, j, p) submatrix shall be defined as the $(p+1) \times (p+1)$ submatrix whose top left element is $d_{i,j}$. From the definition of the entries in the d matrix, (4.11), it may be seen that the values in an (i, j, p) submatrix are derived from its relevant substrings, $x(i+1, i+p)$ and $y(j+1, j+p)$, and from its initial vectors, i.e. its top row $(d_{i,j}, d_{i,j+1}, \ldots, d_{i,j+p})$ and its leftmost column $(d_{i,j}, d_{i+1,j}, \ldots, d_{i+p,j})$.

The first stage of the algorithm is to calculate the values in the final vectors, i.e. the bottom row and the rightmost column, of all the possible (i, j, p) submatrices of any d matrix for a given alphabet and cost function. To do this involves enumerating all the submatrices, requiring all the combinations of possible substrings and initial vectors to be listed. In order to enumerate all the length-p substrings, we must assume that the alphabet is finite. Furthermore, attempting to enumerate every possible initial vector may be prohibitive. However, restricting the edit costs to be integral multiples of some real number results in there being only a finite number of differences between consecutive values of d for all distance matrices using the same alphabet and cost function. It is thus more practicable to use these differential values, rather than the absolute distances. Defining a *step* as the difference between any two horizontally or vertically adjacent matrix elements leads to the following corollary of the rule for calculating $d_{i,j}$, (4.11).

$$
\begin{aligned}
(d_{i,j} - d_{i-1,j}) &= \min \left\{ \begin{array}{l} w(x_i, \epsilon), \\ d_{i,j-1} - d_{i-1,j} + w(\epsilon, y_j), \\ d_{i-1,j-1} - d_{i-1,j} + w(x_i, y_j) \end{array} \right\} \\
&= \min \left\{ \begin{array}{l} w(x_i, \epsilon), \\ w(\epsilon, y_j) + (d_{i,j-1} - d_{i-1,j-1}) - (d_{i-1,j} - d_{i-1,j-1}), \\ w(x_i, y_j) - (d_{i-1,j} - d_{i-1,j-1}) \end{array} \right\} \\
(d_{i,j} - d_{i,j-1}) &= \min \left\{ \begin{array}{l} d_{i-1,j} - d_{i,j-1} + w(x_i, \epsilon), \\ w(\epsilon, y_j), \\ d_{i-1,j-1} - d_{i,j-1} + w(x_i, y_j) \end{array} \right\} \\
&= \min \left\{ \begin{array}{l} w(x_i, \epsilon) + (d_{i-1,j} - d_{i-1,j-1}) - (d_{i,j-1} - d_{i-1,j-1}), \\ w(\epsilon, y_j), \\ w(x_i, y_j) - (d_{i,j-1} - d_{i-1,j-1}) \end{array} \right\}
\end{aligned} \tag{4.42}
$$

Each submatrix may thus be derived from the appropriate substrings, $x(i+1, i+p)$ and $y(j+1, j+p)$, a starting value $d_{i,j}$, and its initial step vectors — the top row $(d_{i,j+1} - d_{i,j}, \ldots, d_{i,j+p} - d_{i,j+p-1})$ and the first column $(d_{i+1,j} - d_{i,j}, \ldots, d_{i+p,j} - d_{i+p-1,j})$. The enumeration of all the possible submatrices may be achieved by enumerating all pairs of length-p strings over our finite alphabet, C, and all pairs of length-p step vectors.

for each pair of strings $u, v \in C^p$
  for each pair of length-p step vectors T, L
    — initialise initial step vectors
    for $i = 1$ **to** p
      $V_{i,0} = L_i$
      $H_{0,i} = T_i$
    — calculate step matrix entries
    for $i = 1$ **to** p
      for $j = 1$ **to** p

$$V_{i,j} = \min \left\{ \begin{array}{l} w(u_i, \epsilon), \\ w(\epsilon, v_j) + V_{i,j-1} - H_{i-1,j}, \\ w(u_i, v_j) - H_{i-1,j} \end{array} \right\}$$

$$H_{i,j} = \min \left\{ \begin{array}{l} w(u_i, \epsilon) + H_{i-1,j} - V_{i,j-1}, \\ w(\epsilon, v_j), \\ w(u_i, v_j) - V_{i,j-1} \end{array} \right\}$$

    — save results
    $R = (V_{1,p}, V_{2,p}, \ldots, V_{p,p})$
    $B = (H_{p,1}, H_{p,2}, \ldots, H_{p,p})$
    $store(R,\ B,\ L,\ T,\ u,\ v)$

Figure 4.17: Masek-Paterson submatrix preprocessing

The algorithm to precompute all the submatrices is given in Figure 4.17. The enumeration of the length-p strings and the length-p step vectors is in lexicographic order, assuming fixed orderings on C and on the finite set of possible step values. A step submatrix is evaluated for each pair of strings, u and v, and each pair of initial step vectors, T and L, according to (4.42). In order to facilitate later retrieval, a procedure *store* saves the final step vectors, B and R, of the submatrix determined by u, v, T and L. Two matrices of steps are computed during the procedure — V comprises the vertical step values, and H the horizontal ones.

As the innermost loop takes constant time and is iterated exactly p^2 times for each pair of strings and step vectors, each submatrix is computed in $O(p^2)$ time. The number of pairs of length-p strings over alphabet C is equal to $|C|^{2p}$. If s is the cardinality of the set of possible step values, which we will assume to be finite, then there are s^{2p} distinct pairs of length-p step vectors. There is therefore a total of $(s \cdot |C|)^{2p}$ different submatrices, giving an overall preprocessing time of $O(p^2(s \cdot |C|)^{2p})$.

It was previously stated that, for the restriction on the cost function, the set of possible step values was finite. This will now be examined a little more closely. It may be seen from the rule for $d_{i,j}$ and the boundary conditions, (4.11)-(4.12), that the step values are bounded, independently of the actual strings involved, as shown below.

$$\begin{array}{rcl} -I & \leqslant & (d_{i,j} - d_{i-1,j}) \leqslant D \\ -D & \leqslant & (d_{i,j} - d_{i,j-1}) \leqslant I \\ \text{where } D & = & \max\{w(a, \epsilon)\ :\ a \in C\}, \\ I & = & \max\{w(\epsilon, b)\ :\ b \in C\} \end{array} \tag{4.43}$$

— initialise first column of the leftmost step-submatrices
for $i = 1$ **to** m/p
  $P_{i,0} = (w(x_{(i-1)p+1}, \epsilon),\ w(x_{(i-1)p+2}, \epsilon), \ldots,\ w(x_{ip}, \epsilon))$
— initialise top row of the uppermost step-submatrices
for $j = 1$ **to** n/p
  $Q_{0,j} = (w(\epsilon, y_{(j-1)p+1}),\ w(\epsilon, y_{(j-1)p+2}), \ldots,\ w(\epsilon, y_{jp}))$
— lookup final step vectors for the submatrices
for $i = 1$ **to** m/p
  for $j = 1$ **to** n/p
    $P_{i,j},\ Q_{i,j} = fetch(P_{i,j-1},\ Q_{i-1,j},\ x((i-1)p+1, ip),\ y((j-1)p+1, jp))$
— sum the distance increments to obtain $d_{m,n}$
$d = 0$
for $i = 1$ **to** m/p
  $d = d + sum(P_{i,0})$
for $j = 1$ **to** n/p
  $d = d + sum(Q_{m/p,j})$

Figure 4.18: Masek-Paterson string distance calculation

The cost function is said to be *sparse* if every member of the set of cost weights, namely $\{w(a, \epsilon) : a \in C\} \cup \{w(\epsilon, b) : b \in C\} \cup \{w(a, b) : a, b \in C\}$, is an integral multiple of some constant r. It can be shown that, for finite alphabets, if the cost function is sparse, then the set of step values obtained in the submatrices is finite irrespective of the particular strings involved, thereby justifying this earlier assumption.

The algorithm to compute the string distance from the preprocessed submatrices is given in Figure 4.18. Assuming that $m \bmod p = n \bmod p = 0$, i and j step through the submatrices vertically and horizontally respectively. P is a matrix of length-p step-submatrix column vectors, such that $P_{i,j}$ is the rightmost column of submatrix (i, j), which is equal to the leftmost one of submatrix $(i, j+1)$ since adjacent $(p+1) \times (p+1)$ submatrices share a common border. Similarly, Q is a matrix of length-p step-submatrix row vectors, such that $Q_{i,j}$ is the bottom row of submatrix (i, j), equal to the top row of submatrix $(i + 1, j)$. The 0^{th} column and row, respectively, of P and Q are initialised with the steps corresponding to the distance array boundary conditions given in (4.12). Procedure $fetch$ returns the rightmost column and the bottom row of submatrix (i, j) by retrieving the precomputed values addressed by the leftmost column and top row of the submatrix together with the relevant substrings. Finally, the absolute distance between the two input strings, $d_{m,n}$, is obtained by accumulating the distance differences along a path from $d_{0,0}$ to $d_{m,n}$. Function sum, employed during this stage, returns the sum of the components of the vector supplied as its argument.

The initialisation stage of this algorithm runs in linear time, as does the final summing of the differences to obtain the required distance value. The main calculation stage involves $2mn/p^2$ length-p vector lookups and assignments, and thus takes $O(mn/p)$ time. It was noted earlier that the submatrix precomputations require $O(p^2 c^{2p})$ time, where $c = s \cdot |C|$. If p is taken as $\min\lfloor\{\log_c m, n\}/2\rfloor$, then the preprocessing time is asymptotically less than that required for the main computation. The overall distance calculation, including the preprocessing, therefore takes $O(mn/p)$ time. This bound is still valid when m and n are not exact multiples of p. This case is handled by appending dummy symbols not in the strings, α say, to x and y in order to bring their lengths up to integral multiples of p. The edit costs involving α are then set as follows.

$$\begin{aligned} w(\alpha,\epsilon) &= 0 \\ w(\epsilon,\alpha) &= 0 \\ w(a,\alpha) &= w(a,\epsilon) \quad \forall a \in C \\ w(\alpha,b) &= w(\epsilon,b) \quad \forall b \in C \end{aligned} \tag{4.44}$$

An actual minimal cost edit sequence from x to y may be obtained in a manner similar to the backtracking technique of Wagner and Fischer. Obtaining the required distance values for this involves either recalculating the submatrices crossed by the optimal edit path, or storing the complete submatrices during the preprocessing stage. Adopting the latter approach allows the edit sequence to be determined from the completed P and Q matrices and the precomputed submatrices in linear time, and requires $O(n \log^2 n)$ space for submatrix storage. Note that the P and Q matrices each require $(1 + m/p) \times (1 + n/p) \times p$ storage locations, which is $O(n^2/\log n)$. The overall space complexity of this method is therefore $O(n^2/\log n)$.

The algorithm may also be applied to the problem of determining $\mathrm{lcs}(x, y)$, by employing the cost function of the edit distance, given by (4.15). The relationship between $d(x, y)$ and $|\,\mathrm{lcs}(x, y)\,|$ given in (4.16) then permits the length of the lcs to be computed in $O(n^2/\log n)$ time. An actual lcs may then be recovered using a technique similar to that used to obtain an optimal edit sequence.

The computation of string distances using this method has been shown to be asymptotically faster than the quadratic method of Wagner and Fischer. Practical savings are, however, only obtained for very long strings. To illustrate this point, Masek and Paterson (1983) calculated that, for a binary alphabet and the edit distance cost function, superior performance is achieved only for strings with lengths in excess of 262418.

4.2.5 Ukkonen

Ukkonen's (1985) method of calculating the distance between two strings and obtaining a minimal edit sequence is based on the dynamic programming approach, but requires $O(dm)$ time and space, where d is the distance between strings x and y and m is the smaller of the two string lengths. The method is thus attractive when the inter-string distance is small, i.e. when the two strings are similar. The basic idea of the algorithm is to find the cheapest cost path in a directed graph in the $d_{i,j}$ matrix, and during the process unnecessary $d_{i,j}$ calculations are avoided.

In the following description, it is assumed, as before, that $m \leqslant n$. Recall that in the dynamic programming method, $d_{i,j}$ is derived from a minimisation involving its previously computed neighbours using relation 4.11. On completion of the calculation of $d_{m,n}$, a minimal cost edit sequence is then obtained by tracing a path composed of transitions that resulted in the minimum for the values of $d_{i,j}$ through the matrix.

The dependencies between the elements of the d matrix may be represented by directed arcs — an arc exists from (i,j) to (i',j') only if $d_{i',j'}$ has been obtained from $d_{i,j}$. The resulting *dependency graph* is a subgraph of the larger graph composed of all the nodes (i,j) and the vertical, horizontal, and (top left to bottom right) diagonal arcs connecting adjacent nodes. These arcs correspond to deletions, insertions, and substitutions or matches, respectively. The costs of these operations may therefore be associated with the relevant arcs, and thus the value of $d_{i,j}$ is given by the sum of the costs on any path from $(0,0)$ to (i,j). The arcs and their associated costs are as follows.

$$\begin{array}{rcll} d_{i-1,j} & \rightarrow & d_{i,j} & \quad w(x_i,\epsilon) \\ d_{i,j-1} & \rightarrow & d_{i,j} & \quad w(\epsilon,y_j) \\ d_{i-1,j-1} & \rightarrow & d_{i,j} & \quad w(x_i,y_j) \end{array}$$

The terminal values on a path from (i,j) to (i',j') in the dependency graph are related as follows, where d is the sum of the costs of the arcs in the path.

$$d_{i',j'} = d_{i,j} + d \tag{4.45}$$

The distance between the two strings, $d_{m,n}$, is thus the minimum total cost for a path from $d_{0,0}$ to $d_{m,n}$ in the dependency graph.

Dijkstra's algorithm could be used to find the cheapest path (Aho, Hopcroft and Ullman, 1974) in $O(mn\log mn)$ time. However, the topology of the graph lends itself to more efficient solutions, such as the dynamic programming method. It is further shown below how the number of entries of the d array that need be calculated may be reduced.

Only entries $d_{i,j}$ on some path from $(0,0)$ to (m,n) are relevant for the value of $d_{m,n}$. Note that this value is $O(n)$, since any such path has at most $m+n$ arcs. Consider the problem of ascertaining whether or not $d_{m,n}$ exceeds some threshold, h, say. From 4.11, it may be seen that the values of $d_{i,j}$ increase monotonically along any path in the dependency graph. Thus, if $d_{m,n} \leqslant h$, and if some $d_{i,j} > h$, then that $d_{i,j}$ cannot possibly be part of any path to (m,n).

Let w_{min} denote the minimum cost of all indels, that is

$$w_{min} \;=\; \min\{w(a,\epsilon),\; w(\epsilon,a) \;:\; a \in C\} \tag{4.46}$$

for alphabet C. For non-zero positive weights, $w_{min} > 0$. Also, let the diagonals of the distance matrix be numbered with integers, $k \in [-m,n]$, such that diagonal k consists of elements (i,j) for which $j-i=k$.

Consider any path from (i,j) to (i',j') in the dependency graph. Element (i,j) lies on diagonal $k=j-i$, and (i',j') on diagonal $k'=j'-i'$. If $k'-k \leqslant 0$, then the path contains at least $|\,k'-k\,|$ deletions (vertical arcs), and if $k'-k \geqslant 0$, at least $|\,k'-k\,|$ insertions (horizontal arcs). From 4.45, we then have the following.

$$d_{i',j'} \;\geqslant\; d_{i,j} + |\,(j'-i') \;-\; (j-i)\,| \cdot w_{min} \tag{4.47}$$

Thus, $d_{i,j} \geqslant |\,j-i\,| \cdot w_{min}$ for every (i,j) on a path from $(0,0)$ to (m,n). Also, since $d_{i,j} \leqslant d_{m,n}$ for every (i,j) on the path, we have the following.

$$|\,j-i\,| \;\leqslant d_{i,j}/w_{min} \leqslant d_{m,n}/w_{min} \tag{4.48}$$

It is thus sufficient to consider elements (i,j) in the diagonal band $-d_{m,n}/w_{min} \leqslant j-i \leqslant d_{m,n}/w_{min}$ in order to calculate $d_{m,n}$. This band may, in fact, be further narrowed as shown below.

Consider a path from $(0,0)$ to (m,n) in the dependency graph. This may be decomposed into the subsidiary paths from $(0,0)$ to (i,j) and from (i,j) to (m,n). From 4.47 we have the following.

$$\begin{aligned} d_{m,n} &\geqslant d_{0,0} + |j-i| \cdot w_{min} + |(n-m) - (j-i)| \cdot w_{min} \\ \Rightarrow d_{m,n} &\geqslant (|j-i| + |(n-m) - (j-i)|) \cdot w_{min} \end{aligned} \tag{4.49}$$

There are two cases for the relative sizes of i and j to consider, namely when $j \leqslant i$ and when $j \geqslant i$. For the former case, 4.49 becomes

$$\begin{aligned} d_{m,n} &\geqslant (-(j-i) + n - m - (j-i))w_{min} \\ \Rightarrow d_{m,n}/w_{min} - (n-m) &\geqslant -2(j-i) \end{aligned} \tag{4.50}$$

Taken together with the fact that $j-i$ is an integer $\leqslant 0$, this gives the following

$$-\left\lfloor \left(\frac{d_{m,n}}{w_{min}} - (n-m)\right)/2 \right\rfloor \leqslant j-i \leqslant 0 \tag{4.51}$$

For the case where $j \geqslant i$, there are two further possibilities to consider, namely when $n-m \geqslant j-i$ and when $n-m \leqslant j-i$. For the former case, 4.49 becomes

$$\begin{aligned} d_{m,n} &\geqslant (j-i+n-m-(j-i))w_{min} \\ \Rightarrow d_{m,n}/w_{min} &\geqslant n-m \end{aligned} \tag{4.52}$$

and for the latter case

$$\begin{aligned} d_{m,n} &\geqslant (j-i+j-i-(n-m))w_{min} \\ \Rightarrow d_{m,n}/w_{min} - (n-m) &\geqslant 2(j-i) - 2(n-m) \end{aligned} \tag{4.53}$$

Taken together with the fact that, in this case, $j-i$ is an integer $\geqslant 0$, this gives the following

$$\left\lfloor \left(\frac{d_{m,n}}{w_{min}} - (n-m)\right)/2 \right\rfloor + (n-m) \geqslant j-i \geqslant 0 \tag{4.54}$$

Combining (4.51) and (4.54) gives the following bounds for $j-i$ at points (i,j) lying on some path from $(0,0)$ to (m,n) in the dependency graph.

$$\begin{gathered} -p \leqslant j-i \leqslant n-m+p \\ \text{where } \ p = \left\lfloor \left(\tfrac{d_{m,n}}{w_{min}} - (n-m)\right)/2 \right\rfloor \end{gathered} \tag{4.55}$$

A consequence of (4.55) is that, when testing whether $d(x,y) \leqslant h$, the computation of $d_{i,j}$ values may be restricted to the diagonal band between the diagonals $-p$ and $n-m+p$, where $p = \lfloor (h/w_{min} - (n-m))/2 \rfloor$.

An algorithm to perform this threshold test is given in Figure 4.19. The procedure returns a negative value if the test fails, and returns the value of $d_{m,n}$ if this is less than or equal to the

```
distance_test(h)
  if h/w_min < n − m
    — trivial case
    return(−1)
  else
    — initialise
    p = ⌊(h/w_min − (n − m)) /2⌋
    d_{0,0} = 0
    for j = 1 to min{n, n − m + p}
      d_{0,j} = d_{0,j−1} + w(ε, y_j)
    — calculate d_{m,n}
    for i = 1 to m
      for j = max{0, i − p} to min{n, i + n − m + p}
        if j = 0
          d_{i,0} = d_{i−1,0} + w(x_i, ε)
        else
          d_{i,j} = min{d_{i−1,j} + w(x_i, ε), d_{i,j−1} + w(ε, y_j), d_{i−1,j−1} + w(x_i, y_j)}
    if d_{m,n} ⩽ h
      return(d_{m,n})
    else
      return(−1)
```

Figure 4.19: Ukkonen string distance threshold test

$h = (n - m + 1)w_{min}$
while $(d = distance\_test(h)) < 0$
$\quad h = 2h$

Figure 4.20: Ukkonen string distance calculation

threshold value. The test fails in the trivial case given by (4.52), as the distance must at least be equal to the difference in length of the strings times the minimum indel cost. For the non-trivial case, $d_{i,j}$ values are evaluated in the diagonal strip defined by (4.55). On completion of these computations, the final value of $d_{m,n}$ is compared with the threshold value.

For each of the $m+1$ rows, the number of matrix elements evaluated is at most $n-m+2p+1$, which is $O(h)$, as shown below.

$$\begin{aligned} p &= \lfloor (h/w_{min} - (n-m))/2 \rfloor \\ \Rightarrow 2p &\leqslant h/w_{min} - (n-m) \\ \Rightarrow n - m + 2p + 1 &\leqslant 1 + h/w_{min} \\ &= O(h) \end{aligned} \tag{4.56}$$

The threshold test therefore executes in $O(hm)$ time, and requires $O(hm)$ space if only the $d_{i,j}$ values in the diagonal band are stored. Note that if only the value of the string distance is required, then the space requirement could be reduced to $O(h)$ locations, as only the previous row is required in the computation of the current one (cf. Hirschberg's linear space algorithm).

In order to determine the actual distance between the two strings, the threshold test procedure may be called with successively increasing values of h until the test succeeds, as shown in Figure 4.20. The threshold is initially set to the minimum possible value of the distance plus w_{min}, and is successively doubled as long as the threshold test fails. On completion, the distance between the two strings is given by d.

Denoting the initial value of h by h_0, then the next value, h_1, equals $2h_0$, and the one after that, h_2, equals 2^2h_0, and so on. Thus, the r^{th} value, h_r, is equal to $2^r h_0$. If r calls to $distance\_test$ are required to determine $d(x, y)$, then the total time required is $O(m \sum_{k=0}^{r} h_k)$, i.e. $O(m(2h_r - 1))$ which is $O(mh_r)$. But, since $d > h_r/2$, the overall time complexity is $O(dm)$. An actual editing sequence from x to y may be obtained from the computed $d_{i,j}$ values on completion as in the Wagner-Fischer method.

Ukkonen also considered the special case when all the editing operation costs are equal (i.e. the calculation of the Levenshtein distance), for which the storage requirements may be reduced and a more efficient, $O(dm)$ time, direct method of computing $d(x, y)$ may be employed.

4.2.6 Heaviest Common Subsequence

Jacobson and Vo's (1992) hcs algorithm, derived from the Robinson-Schensted longest increasing subsequence (lis) algorithm (Schensted, 1961), is described below. The algorithms presented here are in a modified form so that the construction of the directed graphs used in the procedures is correct.

Firstly, the weight of a subsequence, s, common to strings x and y may be defined thus

$$W(s) = \sum_{p=1}^{|s|} f(i_p, j_p, s_p) \tag{4.57}$$

where (i_p, j_p) is a matched pair for x and y, i.e. $x_{i_p} = y_{j_p} = s_p$, and f is a weight function.

The dynamic programming method of computing an lcs may be adapted to evaluate an hcs. Let $W(\text{hcs}(x(1,i), y(1,j)))$ be denoted by $W_{i,j}$. The recurrence relation for $W_{i,j}$ is then given by

$$\begin{aligned} W_{i,j} &= \max\{W_{i-1,j}, W_{i,j-1}, W_{i-1,j-1} + f_{i,j}\} \qquad (4.58)\\ f_{i,j} &= \begin{cases} 0 & \text{if } x_i \neq y_j \\ f(i,j,x_i) & \text{if } x_i = y_j \end{cases} \end{aligned}$$

The above follows from the method of calculating the l matrix. When at position (i, j), i.e. when considering prefixes $x(1,i)$ and $y(1,j)$, if $x_i = y_j$, then a new common subsequence may be obtained by appending this symbol to the current common subsequence of $x(1, i-1)$ and $y(1, j-1)$, and the weight of the new subsequence is obtained by adding the weight of this match to that of the previous one. Otherwise position (i, j) does not alter the previous result. The weight for position (i, j) is then given by the heaviest of the previous neighbouring positions and, if applicable, the new derived weight.

The Robinson-Schensted algorithm to compute an lis of string y is given in Figure 4.21. L is an ordered list of pairs (a, k), where a is a string symbol and k is its position in the string. The following operations may each be performed in $O(\log n)$ time when L is implemented using a balanced tree structure.

$insert(L,\ a)$	–	insert object a into list L
$delete(L,\ a)$	–	delete object a from list L
$next(L,\ a)$	–	return the least object in L strictly greater than a
$prev(L,\ a)$	–	return the greatest object in L strictly less than a
$max(L)$	–	return the greatest object in L
$min(L)$	–	return the least object in L

When not defined, procedures *next*, *prev*, *max* and *min* all return a null value. The operations $next(L,$ null) and $prev(L,$ null) are taken to be equivalent to $min(L)$ and $max(L)$ respectively.

For each symbol in y, y_i, a is the symbol in L to which y_i may be appended whilst preserving the strictly increasing nature of the list. The element after a in L is then replaced by y_i. A directed graph of string symbols is created using array *node*. The elements of *node* are pointers to nodes created by procedure *newnode* and containing a symbol, its position in the string, and the address of a previous node. On completion, an lis of y may be recovered by tracing back through the graph starting from the maximal component of L.

```
— initialise list L
L = null
— calculate lis
for i = 1 to n
    (a, k) = prev(L, (y_i, 0))
    (b, l) = next(L, (a, k))
    if b ≠ null
        delete(L, (b, l))
    insert(L, (y_i, i))
    node_i = newnode((y_i, i), node_k)
```

Figure 4.21: Robinson-Schensted lis calculation

```
— initialise list L
L = null
— calculate his
for i = 1 to n
    (a, u, k) = prev(L, (y_i, 0, 0))
    (b, v, l) = next(L, (a, u, k))
    while ((b, v, l) ≠ null) and (u + f(i, y_i) ⩾ v)
        delete(L, (b, v, l))
        (b, v, l) = next(L, (b, v, l))
    if ((b, v, l) = null) or (y_i < b)
        insert(L, (y_i, u + f(i, y_i), i))
        node_i = newnode((y_i, u + f(i, y_i), i), node_k)
```

NOTE that the **and** in the **while** loop is a 'conditional and,' which only evaluates its right hand operand if its left hand one evaluates to true.

Figure 4.22: Jacobson-Vo his calculation

In order to calculate an his, it is necessary to keep track of the cumulative weights of the his's of the successively longer string prefixes as the computation progresses. The components of list L may thus be made triples (a, u, k), where a is a symbol of the input string, u is the total weight of the his ending with a, and k is the position of a in the string. L remains strictly monotonic in all three elements of its component triples. The ordering of L may therefore be maintained with reference to the lexicographic symbol ordering of the first element of each pair.

The algorithm to compute an his, based on the lis algorithm, is given in Figure 4.22. Again, a gives the largest symbol in L less than y_i, so that the latter may be appended to any increasing subsequence ending with a to form a new increasing subsequence. If there is no such a, then u is taken to be 0. The **while** loop ensures that the strict monotonicity of the second element of the component triples of L is maintained. The new triple is then inserted into L if this is consistent with the preservation of the strict increase of the first element of the triples. Array *node* is employed to construct a linked structure for the recovery of an actual his on completion of the process.

Consider the state of list L at the end of each iteration, denoted by L_i. For each component, (a, u, k) of L_i, an increasing subsequence of prefix $y(1, i)$ ending in a may be recovered by tracing back from a through the directed graph. It will be shown below that for every increasing subsequence, $s(1, k)$, of prefix $y(1, i)$, there exists an element of L_i, $b \leqslant s_k$, terminating an increasing subsequence in the graph at least as heavy as $s(1, k)$. The maximal symbol of L_i, together with the graph, thus provides an his of prefix $y(1, i)$. Consequently, the maximal element of L on completion of the process defines an his for the complete string y.

The above proposition is clearly true for $i = 1$. We may thus assume it to be true for iteration $i - 1$ and analyse the situation for iteration i. Let $s(1, k)$ be an increasing subsequence of prefix $y(1, i)$. From the foregoing, if $s_k = y_i$, then there is an element $b \leqslant s_{k-1}$ in L_{i-1} defining an increasing subsequence at least as heavy as $s(1, k - 1)$. Now, element b cannot be deleted from the list on this iteration, as $b < y_i$. At the end of the i^{th} iteration, y_i will either have been inserted into the list or will have already been present. Recalling that the cumulative weights in L are strictly increasing, it may be seen that the increasing subsequence defined by y_i satisfies the proposition.

Turning our attention to the situation where $s_k \neq y_i$, there are two cases to consider, namely when $s_k < y_i$ and when $s_k > y_i$. The proposition holds for the former case since the portion of the list preceding y_i is not altered by the i^{th} iteration. Thus only the case where $s_k > y_i$ remains to consider. Sequence $s(1, k)$ must then also be an increasing subsequence of $y(1, i-1)$. There will thus exist an increasing subsequence defined by some b in L_{i-1} at least as heavy as $s(1, k)$. If b is extant in L_i, then clearly the proposition holds. Otherwise, b must have been eliminated from the L during the i^{th} iteration, for which case the weight condition of the **while** loop ensures that the sequence defined by y_i is at least as heavy as that previously defined by b. The above proposition has thus been shown to be valid under all circumstances, and the algorithm therefore correctly calculates an his of the input string.

The main loop of the algorithm is iterated a total of n times, and each of its component operations takes at most $O(\log n)$ time. Note that the total number of iterations of the inner **while** loop is at most n, since each of the n elements of y may be inserted into, and deleted from, list L once at most. The temporal complexity of the his algorithm is therefore $O(n \log n)$.

The operation of the method may be illustrated by way of an example. The (a, u) components of the triples of list L for each step i are shown below for the case of $y =$ `zeitgeist`. Note that the lis of this string is `eigst`. However, the weights have been chosen to be equal to 1 for the first occurrence of a symbol in the string, and equal to 2 for the second occurrence. Under these conditions, increasing subsequence `eist` using the second `e` and `i` is heavier than lis `eigst`. On completion, the maximal element of L, `(t,7)`, is used to recover the path `e` $\leftarrow$ `i` $\leftarrow$ `s` $\leftarrow$ `t` from the directed graph.

i	1	2	3	4	5	6	7	8	9
y_i	z	e	i	t	g	e	i	s	t
$f(i, y_i)$	1	1	1	1	1	2	2	1	2
L_i	(z,1)	(e,1)	(e,1)	(e,1)	(e,1)	(e,2)	(e,2)	(e,2)	(e,2)

```
(i,2) (i,2) (g,2) (t,3) (i,4) (i,4) (i,4)
      (t,3) (t,3)             (s,5) (s,5)
                                    (t,7)
```

Directed graph

```
null  <---  (z,1)

null  <---  (e,1)  <---  (i,2)  <---  (t,3)
              |
               ----<---  (g,2)

null  <---  (e,2)  <---  (i,4)  <---  (s,5)  <--- (t,7)
```

If the matches (i, j) between strings x and y are listed in increasing order of i, and decreasing order of j for equal i values, then every common subsequence of x and y maps to an increasing subsequence of the sequence of j values. Conversely, a common subsequence may be derived from an increasing subsequence of j values. The descending order of j for constant i prevents multiple symbols of y matching a single symbol of x from being included in a subsequence. The example given below is for the strings `preterit` and `zeitgeist`. The underlined match indices show the lis of the j sequence, corresponding to the lcs of the two strings.

```
i        1 2 3 4 5 6 7 8
x_i      p r e t e r i t
             - - -   - -

j        1 2 3 4 5 6 7 8 9
y_j      z e i t g e i s t
           -   -   - -   -

Matches
i        3 3 4 4 5 5 7 7 8 8
j        6 2 9 4 6 2 7 3 9 4
           -   - -   -   -
```

In the above ordering of the matches, corresponding weights, $f(i, j, x_i)$, may be associated with the j values. An hcs of the strings may thus be computed by calculating the his of this sequence of j values. The method to do this, given in Figure 4.23, is a generalisation of the Hunt-Szymanski lcs algorithm. Note that only the (a, u) components of the triples of list L are shown here for clarity.

Array *position* contains ordered lists of the positions of the symbols occurring in y. This is thus similar to the *matchlist* of the Hunt-Szymanski algorithm. The latter was in descending order, whereas the lists of *position* are in ascending order. However, the correct ordering

— construct ordered lists of positions of symbols in y
for $i = 1$ **to** n
  $insert(position_{y_i},\ i)$
— initialise list L
$L =$ null
— calculate hcs
for $i = 1$ **to** m
  $j = max(position_{x_i})$
  while $j \neq$ null
    $(a, u) = prev(L,\ (j, 0))$
    $(b, v) = next(L,\ (a, u))$
    while $((b, v) \neq$ null$)$ **and** $(u + f(i, j, x_i) \geqslant v)$
      $delete(L,\ (b, v))$
      $(b, v) = next(L,\ (b, v))$
    if $((b, v) =$ null$)$ **or** $(j < b)$
      $insert(L,\ (j, u + f(i, j, x_i)))$
    $j = prev(position_{x_i},\ j)$

NOTE that the **and** in the **while** loop is a 'conditional and'

Figure 4.23: Jacobson-Vo hcs calculation

of the matches is achieved by stepping backwards through the j positions in $position_{x_i}$ for each i. The his algorithm is applied to the resulting sequence of j values. The corresponding hcs(x, y) may then be recovered on completion of the process from the directed graph, the construction of which has also been omitted from the algorithm for clarity.

The total length of the sequence of j values is equal to the number of matches, r. The processing time also depends on the size of list L, which is bounded by the length of the shorter of the two input strings. The overall running time is thus $O((r + m) \log n)$.

The following example serves to illustrate the computation of an hcs. As mentioned in the previous chapter, a minimal distance weight function may be applied in order to give preference in the selection of common subsequences to closely aligned matches. Consider the pair of strings `warfare` and `forewarn`. The longest common subsequences of this pair are shown below, together with their weights for the minimal distance weight function $8 - |i - j|$, which favours closely aligned matches. The total lcs weights are given in parentheses.

i	1 2 3 4 5 6 7
x_i	w a r f a r e
y_j	f o r e w a r n
j	1 2 3 4 5 6 7 8

x_i	w a r	w a r	w a r

i	1 2 3	1 2 6	1 5 6
j	5 6 7	5 6 7	5 6 7
$8-\|i-j\|$	4 4 4 (12)	4 4 7 (15)	4 7 7 (18)

x_i	r a r	f a r	f r e
i	3 5 6	4 5 6	4 6 7
j	3 6 7	1 6 7	1 3 4
$8-\|i-j\|$	8 7 7 (22)	5 7 7 (19)	5 5 5 (15)

It may thus be seen that `rar` is the heaviest lcs. The state of list L after each stage i in the computation of hcs(`warfare`, `forewarn`) for the above weight function is shown below, together with the resulting directed graph. Once again, only the (a, u) components of the list triples are given for simplicity.

i	1	2	3	4	5	6	7
x_i	w	a	r	f	a	r	e
L_i	(5,4)	(5,4)	(3,8)	(1,5)	(1,5)	(1,5)	(1,5)
		(6,8)	(7,12)	(3,8)	(3,8)	(3,10)	(3,10)
				(7,12)	(6,15)	(6,15)	(4,15)
						(7,22)	(7,22)

Directed graph

```
null  <---  (5,4)  <---  (6,8)   <---  (7,12)

null  <---  (3,8)  <---  (6,15)  <---  (7,22)

null  <---  (1,5)  <---  (3,10)  <---  (4,15)
```

On completion, the maximal element in L, namely $(7, 22)$, is used to retrieve path $3 \leftarrow 6 \leftarrow 7$ from the directed graph of j index values, corresponding to the subsequence `rar`.

A special case of the hcs problem is when the weights are position-independent, i.e. when f is a function only of the symbols. The hcs algorithm may be adapted to take advantage of this special case as discussed below.

When the weight function does not depend on position, the weights associated with the positions j in list $position_{x_i}$ are all equal. Consider the case of a given j being inserted into list L after a. Recalling that the j values from the position list are processed in descending order, if the next value of j tried is also greater than a, then the old j will be deleted from L and the new one inserted in its place, since they both have the same weight. This inefficiency may be avoided by going directly to the least j in $position_{x_i}$ greater than a. So, j may be assigned the value of $next(position_{x_i},\ a)$ after the assignments of (a, u) and (b, v).

— construct ordered lists of positions of symbols in y
for $i = 1$ **to** n
    $insert(position_{y_i},\ i)$
— initialise list L
$L =$ null
— calculate hcs
for $i = 1$ **to** m
    $j = max(position_{x_i})$
    while $j \neq$ null
        $(a, u) = prev(L,\ (j, 0))$
        $(b, v) = next(L,\ (a, u))$
        $j = next(position_{x_i},\ a)$
        while $((b, v) \neq$ null$)$ **and** $(u + f(x_i) \geqslant v)$
            $delete(L,\ (b, v))$
            $(b, v) = next(L,\ (b, v))$
            $insert(position_{y_b},\ b)$
        $insert(position_{y_b},\ b)$
        $insert(L,\ (j, u + f(x_i)))$
        $delete(position_{x_i},\ j)$
        $j = prev(position_{x_i},\ j)$

NOTE that the **and** in the **while** loop is a 'conditional and'

Figure 4.24: Jacobson-Vo hcs calculation for position-independent weights

If j is already in list L, then b is assigned the value of j. If the entry in L immediately preceding j is the same as that when j was inserted, then the algorithm will simply delete j from L and then reinsert it at the same point. Thus, when an element j is inserted into L, it may then be deleted from $position_{x_i}$, in order to circumvent any unnecessary duplication of effort. It must, however, be reinstated in its position list should it be removed from L or its predecessor in L be changed.

Finally, the condition for inserting j into L is not required when the weights are position-independent. The value of b cannot be less than j, since a is the greatest value in L less than j. Thus, $b \geqslant j$. For the case where $b > j$, j may be inserted into L between a and b without upsetting the list's monotonicity. If $b = j$, then j may safely replace b, since their weights are also the same. This then shows that j may be inserted unconditionally into L.

The hcs algorithm incorporating the above enhancements for the special case of position-independent weights is given in Figure 4.24, which, for clarity, again omits the construction of the directed graph and only depicts the (a, u) pairs of the component triples of list L. When the weights for all the symbols in the alphabet are equal, this reduces to the Apostolico-Guerra lcs algorithm.

4.3 Approximate String Matching

4.3.1 Landau-Vishkin k-mismatch

In Landau and Vishkin's (1985, 1986a) k-mismatch algorithm the text string is analysed using a 2-dimensional *pattern mismatch* table generated during a pattern preprocessing stage. The calculation of this table, $pm[1 \ldots m-1, 1 \ldots 2k+1]$, shall be considered in due course. Firstly, however, the text analysis stage shall be examined.

The analysis of the text is performed in a 2-dimensional *text mismatch* array, $tm[0 \ldots n - m, 1 \ldots k+1]$. On completion of the analysis, row i contains the positions within x of the first $k+1$ mismatches between strings $x(1, m)$ and $y(i+1, i+m)$. Thus, if $tm[i, v] = l$, then $y_{i+l} \neq x_l$ and this is the v^{th} mismatch between $x(1, m)$ and $y(i+1, i+m)$ from left to right. If the number of mismatches between x and substring $y(i+1, i+m)$, c say, is less than $k+1$, then the default value $m+1$ appears from location $c+1$ onwards, i.e.

$$tm[i, c+1] = tm[i, c+2] = \ldots = tm[i, k+1] = m+1$$

Note that a value of $m+1$ for $tm[i, k+1]$ indicates that text substring $y(i+1, i+m)$ differs from pattern x by at most k mismatches, and therefore provides a solution to the problem. This is illustrated in the following example. Consider the case where $x =$ `tram`, $y =$ `thetrippedtrap` and $k = 2$. It may be seen from the following text mismatch table that occurrences of the pattern with up to 2 mismatches are to be found starting at y_4 and y_{11}, namely substrings `trip` and `trap`.

	1	2	3
0	2	3	4
1	1	2	3
2	1	2	3
3	3	4	5
4	1	2	3
5	1	2	3
6	1	2	3
7	1	2	3
8	1	2	3
9	1	2	3
10	4	5	5

The text analysis algorithm is given in Figure 4.25. The pattern is scanned across the text from left to right one place at a time by the **for** loop. At iteration i, text substring $y(i+1, i+m)$ is compared with the pattern. The rightmost position in the text reached by previous iterations is given by j, i.e. it is the maximum value of $r + tm[r, k+1]$, where $0 \leqslant r < i$. If $i < j$, then procedure *merge* is called, which finds mismatches between $x(1, j-i)$ and $y(i+1, j)$ and sets b equal to the number found. If this does not exceed k, then procedure *extend* is invoked. This scans the text from y_{j+1} onwards, until it either finds $k+1$ mismatches or it reaches

```
— initialise
tm[0...n − m, 1...k + 1] = m + 1
r = 0
j = 0
— scan text
for i = 0 to n − m
    b = 0
    if i < j
        merge(i, r, j, b)
    if b < k + 1
        r=i
        extend(i, j, b)
```

Figure 4.25: Landau-Vishkin k-mismatch string matching

```
extend(i, j, b)
    while (b < k + 1) and (j − i < m)
        j = j + 1
        if y_j ≠ x_{j−i}
            b = b + 1
            tm[i, b] = j − i
```

Figure 4.26: Procedure *extend*

y_{i+m} with at most k mismatches, in which case a pattern occurrence has been found starting at y_{i+1}.

The operation of *extend* is straightforward, and the procedure is given in Figure 4.26. Pairs of symbols from the substrings $y(j+1, i+m)$ and $x(j-i+1, m)$ are compared, and, in the event of a mismatch, b is incremented and the text mismatch table is updated.

The operation of procedure *merge* shall now be examined. As noted earlier, this procedure finds mismatches between $x(1, j-i)$ and $y(i+1, j)$ and sets b equal to the number found. Thus, *merge* evaluates $tm[i, 1 \ldots b]$ for $b \leqslant k+1$, and in so doing uses information obtained previously. Firstly, recall that row r of the text mismatch table gives the mismatches obtained when the start of the pattern is aligned with y_{r+1} and that $r + tm[r, k+1]$ is the present rightmost position examined in the text. Mismatches in row r of tm lying to the right of y_i in the text are therefore relevant in the comparison of the pattern with the text starting from y_{i+1}. The appropriate values from the text mismatch table are thus $tm[r, q \ldots k+1]$, where q is the least integer such that $r + tm[r, q] > i$. However, account must be taken of the fact that these mismatches correspond to the start of the pattern being aligned with y_{r+1}, whereas the current pattern position is aligned with y_{i+1} — a difference of $i - r$ places.

The second source of information used by *merge* is thus the precomputed pattern mismatch table, pm, which gives the positions of the mismatches between the pattern and itself at various relative shifts. Row i of array $pm[1 \ldots m-1, 1 \ldots 2k+1]$ contains the positions within

x of the first $2k+1$ mismatches between the substrings $x(1, m-i)$ and $x(i+1, m)$, i.e. the overlapping portions of the two copies of the pattern for a relative shift of i. Thus, if $pm[i, u] = l$, then $x_{i+l} \neq x_l$ and this is the u^{th} mismatch between $x(1, m-i)$ and $x(i+1, m)$ from left to right. If the number of mismatches between these substrings, c say, is less than $2k+1$, then the default value $m+1$ appears from location $c+1$ onwards, i.e.

$$pm[i, c+1] = pm[i, c+2] = \ldots = pm[i, 2k+1] = m+1$$

It is thus row $i-r$ of the pattern mismatch table which is of interest in *merge*, and the relevant values are $pm[i-r, 1 \ldots t]$, where t is the rightmost mismatch in $pm[i-r, 1 \ldots 2k+1]$ such that $pm[i-r, t] < j-i+1$, since only the mismatches in substring $x(1, j-i)$ are required.

In order to see how the information mentioned above may be utilised in procedure *merge*, consider a text location p within the relevant range, i.e. $i+1 \leqslant p \leqslant j$. Position p may satisfy either, neither, or both of the following conditions.

A: with x_1 aligned with y_{r+1}, text position p corresponds to a previously detected mismatch between pattern and text, i.e. $y_p \neq x_{p-r}$, and this is mismatch number v, for some v such that $q \leqslant v \leqslant k+1$, i.e. $p - r = tm[r, v]$.

B: with two copies of the pattern, having a relative shift of $i-r$, aligned with the text such that their initial symbols lie above y_{r+1} and y_{i+1} respectively, text position p corresponds to a mismatch between the two patterns, i.e. $x_{p-r} \neq x_{p-i}$. This will be the u^{th} mismatch for this shift, for some u such that $1 \leqslant u \leqslant t$, i.e. $p - i = pm[i-r, u]$.

Recall that the question of importance at text position p is whether or not the text symbol there matches the corresponding pattern symbol when x_1 is aligned with y_{i+1} — does $y_p = x_{p-i}$? The bearing on this question of the possible situations with regards to the above two conditions shall now be examined.

Case 1: p satisfies neither Condition A nor Condition B. Thus, $y_p = x_{p-r}$ and $x_{p-r} = x_{p-i}$, so $y_p = x_{p-i}$. No explicit comparison is required between pattern and text as there must therefore be a match at this position.

Case 2: p satisfies only one of the two conditions, i.e. either Condition A or Condition B, but not both. In either case, $y_p \neq x_{p-i}$, which may be shown as follows. If only Condition A is satisfied, then $y_p \neq x_{p-r}$ and $x_{p-r} = x_{p-i}$, so $y_p \neq x_{p-i}$. If, on the other hand, only Condition B holds, then $y_p = x_{p-r}$ and $x_{p-r} \neq x_{p-i}$, so again $y_p \neq x_{p-i}$. Again, no explicit comparison is required since there must be a mismatch at text position p.

Case 3: both Condition A and Condition B hold. No conclusion concerning y_p and x_{p-i} may be drawn, requiring the two symbols to be compared directly.

In Case 2 and if a mismatch is found in Case 3, the mismatch count, b, must be incremented and $tm[i, b]$ updated by procedure *merge*.

```
merge(i, r, j, b)
   u = 1
   v = q
   while (b < k + 1) and (v < k + 2) and (i + pm[i - r, u] ⩽ j or tm[r, v] ≠ m + 1)
      if i + pm[i - r, u] > r + tm[r, v]
         — Case 2, Condition A
         b = b + 1
         tm[i, b] = tm[r, v] - (i - r)
         v = v + 1
      else if i + pm[i - r, u] < r + tm[r, v]
         —Case 2, Condition B
         b = b + 1
         tm[i, b] = pm[i - r, u]
         u = u + 1
      else
         — i + pm[i - r, u] = r + tm[r, v]
         — Case 3
         if x_{pm[i-r,u]} ≠ y_{i+pm[i-r,u]}
            b = b + 1
            tm[i, b] = pm[i - r, u]
         u = u + 1
         v = v + 1
```

Figure 4.27: Procedure *merge*

We are now in a position to describe procedure *merge*, which is given in Figure 4.27. As noted earlier, the relevant table values for *merge* are $pm[i-r, 1 \ldots t]$ and $tm[r, q \ldots k+1]$. Variables u and v are set initially to index the first elements of these two vectors respectively, and subsequently step through the vector components.

The conditions for terminating the procedure are as follows. Firstly, if $b = k+1$, then $k+1$ mismatches have been found for the pattern positioned such that x_1 is aligned with y_{i+1}, and the procedure may therefore be exited. Secondly, recalling that the rightmost text position of interest in *merge*, namely j, is equal to $r + tm[r, k+1]$, if $v = k+2$, then $tm[r, k+1]$ will have been used for the previous value of v, namely $v = k+1$, and position j must therefore have been passed. Hence, the procedure may also be exited in this case. Finally, the procedure may also be terminated if $i + pm[i-r, u] > j$ and $tm[r, v] = m+1$. If the latter part of this condition holds, then $r + tm[r, v]$ will equal j, as will corresponding sums for subsequent values of v up to $k+1$. In this case, the procedure may therefore be terminated if the former part of the above condition also holds, as this indicates that text position j has actually been passed.

It now remains to demonstrate that the number of mismatch positions in the pattern mismatch table is sufficient for *merge* to find all or, if there are more than $k+1$, the first $k+1$ mismatches for $y(i+1, j)$. This may be shown as follows. Condition A holds for at most $k+1$ text positions in the range $[i+1, j]$. Condition B holds for some unknown number of positions in the same interval. Row $i - r$ of the pattern mismatch table, $pm[i - r, 1 \ldots 2k + 1]$, contains at most $2k + 1$ mismatch positions between two copies of the pattern with a relative shift of $i - r$. If $pm[i - r, 2k + 1] \geqslant j - i$, then the table contains all the pattern-pattern mismatch positions for which Condition B holds for text positions in the interval $[i + 1, j]$. On the other hand, if $pm[i - r, 2k+1] < j - i$, then the table can provide $2k+1$ text positions in the range $[i+1, j-1]$ for which Condition B holds. Since $j = r + tm[r, k + 1]$, there are up to k text positions in the range $[i + 1, j - 1]$ for which condition A holds. In the worst case, there could thus be k positions for which Case 3 holds, and which would thus require explicit comparisons. This would then leave at least $k + 1$ positions satisfying Condition B but not Condition A (Case 2), which is sufficient to establish that there are at least $k + 1$ mismatches between pattern and text for the particular pattern alignment.

The temporal complexity of the text analysis shall now be examined. Excluding calls to procedures *merge* and *extend*, each of the $n - m + 1$ iterations of the text analysis loop takes constant time, giving a total of $O(n)$ time. The total number of operations performed by *extend* during all its calls is $O(n)$, since it inspects each text symbol once at most. On each call, *merge* operates on the vectors $pm[i - r, 1 \ldots 2k + 1]$ and $tm[r, 1 \ldots k + 1]$, which have a combined total of $3k + 2$ elements. The operations of *merge* may be accounted for by associating constant time operations with each of these entries, resulting in a running time for each invocation of $O(k)$. It may thus be seen that the overall running time of the text analysis is $O(nk)$.

All that remains now is to address the computation of the pattern mismatch table during the pattern preprocessing stage. Without loss of generality, it may be assumed that m is some power of 2. The preprocessing algorithm uses a partition of the set of the $m - 1$ rows of pm, $\{1, 2, \ldots, m - 1\}$, into the following $\log_2 m$ subsets

$$\{1\},\ \{2, 3\},\ \{4, 5, 6, 7\},\ \ldots,\ \{m/2, \ldots, m - 1\}$$

and has $\log_2 m$ stages. At stage l, where $1 \leqslant l \leqslant \log_2 m$, the rows of pm in set l are computed, where set l is $\{2^{l-1}, \ldots, 2^l - 1\}$.

The method used to compute the table is based on that used during the text analysis phase, and the algorithm for stage l is shown in Figure 4.28. At stage l, the inputs to the pattern analysis algorithm are the pattern substrings $x(1, m - 2^{l-1})$ and $x(2^{l-1} + 1, m)$, which correspond to the pattern and text in the text analysis respectively, and the sub-array $pm[1 \ldots 2^{l-1} - 1, 1 \ldots \min\{2^{\log_2 m - l} 4k + 1, m - 2^{l-1}\}]$ which contains the outputs from the previous $l - 1$ stages. (The number of elements in the rows of this sub-array shall be explained later.) The outputs of stage l are entered into $pm[2^{l-1} \ldots 2^l - 1, \min\{2^{\log_2 m - l} 2k + 1, m - 2^l\}]$. With the exception of stage $\log_2 m$, in which up to $2k + 1$ mismatches are found, it is thus up to $\min\{2^{\log_2 m - l} 2k + 1, m - 2^l\}$ mismatches which are to be found for each row of pm at stage l, rather than up to $k + 1$ as in the text analysis algorithm.

Using arguments similar to those of the previous examination of the correct operation of procedure *merge*, it may be shown that in order to find up to the requisite number of mismatches, $\min\{2^{\log_2 m - l} 4k + 1, m - 2^{l-1}\}$ positions for which Condition B holds are required at stage l, and $4k + 1$ such locations for the special case of stage $\log_2 m$.

For each stage, l, of the $\log_2 m$ stages of the pattern analysis, the **for** loop makes 2^{l-1} iterations

```
— initialise
pm[2^{l-1} ... 2^l - 1, 1 ... min{2^{log_2 m - l} 2k + 1, m - 2^l}] = m + 1
r = 2^{l-1}
j = 2^{l-1}
— calculate rows of pm in set l
for i = 2^{l-1} to 2^l - 1
    b = 0
    if i < j
        merge(i, r, j, b)
    if b < min{2^{log_2 m - l} 2k + 1, m - 2^l}
        r = i
        extend(i, j, b)
```

Figure 4.28: Landau-Vishkin pattern preprocessing

$(2^{l-1} \leqslant i \leqslant 2^l - 1)$. Excluding calls to *merge* and *extend*, each iteration takes constant time. For all the iterations of stage l, *extend* requires $O(m)$ time. It was shown earlier that each call of *merge* requires time proportional to the number of mismatches searched for. Thus, each call of *merge* takes $O(\min\{2^{\log_2 m - l}2k + 1, m - 2^l\})$, which is $O(2k2^{\log_2 m - l})$. The total time for stage l is therefore $O(m + 2^{l-1}(2k2^{\log_2 m - l})) = O(km)$. Summing this for all stages gives a total preprocessing time of $O(\sum_{l=1}^{\log_2 m} km) = O(km \log m)$. The overall time complexity of the pattern preprocessing and the text analysis is thus $O(k(n + m \log m))$.

4.3.2 Landau-Vishkin k-difference

Landau and Vishkin's (1986b, 1989) k-difference algorithm is based on an approach similar to Ukkonen's (1983, 1985a) dynamic programming string distance procedure. Before dealing with their algorithm, the dynamic programming method and a Ukkonen-style adaptation thereof shall firstly be described.

Recall that the elements, $d_{i,j}$, defined by (4.9), of the distance matrix give the distance between prefixes of the strings x and y, of lengths i and j respectively. In order to solve the k-difference problem, the distance matrix may be adapted such that $d_{i,j}$ represents the minimum distance between $x(1, i)$ and any substring of y ending at y_j (Sellers, 1980). This change may be accommodated simply by altering the boundary conditions (given by (4.12)) such that

$$d_{0,j} = 0 \quad \text{for } 0 \leqslant j \leqslant n \tag{4.59}$$

since the minimum distance between ϵ and any substring of y is 0.

The rest of the matrix is then computed as before using the Levenshtein distance edit costs, (4.10), and the recurrence relation for $d_{i,j}$, (4.11). On completion, any value not exceeding k in the final row indicates a position in the text where a substring having at most k differences from the pattern ends. This is illustrated in the example given below, which depicts the distance matrix for the case of $x =$ ABCDE and $y =$ ACEABPCQDEABCR. For the case of $k = 2$, it may be seen from row 5 of the matrix that occurrences of the pattern, with up to 2 differences, end at text positions 3, 10, 13 and 14. The corresponding text substrings for these positions are `ACE`, `ABPCQDE`, `ABC` and `ABCR`.

	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
i			A	C	E	A	B	P	C	Q	D	E	A	B	C	R
0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	A	1	0	1	1	0	1	1	1	1	1	1	0	1	1	1
2	B	2	1	1	2	1	0	1	2	2	2	2	1	0	1	2
3	C	3	2	1	2	2	1	1	1	2	3	3	2	1	0	1
4	D	4	3	2	2	3	2	2	2	2	2	3	3	2	1	1
5	E	5	4	3	2	3	3	3	3	3	3	2	3	3	2	2

Recall from Ukkonen's string distance algorithm that the matrix diagonals may be numbered with integers, $p \in [-m, n]$, such that diagonal p consists of elements (i, j) for which $j - i = p$. Let $r_{p,q}$ represent the largest row i such that $d_{i,j} = q$ and (i, j) is on diagonal p. Thus, q is the minimal number of differences between $x(1, r_{p,q})$ and any text substring ending at $y_{r_{p,q}+p}$, and furthermore $x_{r_{p,q}+1} \neq y_{r_{p,q}+p+1}$ since $r_{p,q} + 1$ would otherwise be a row larger than $r_{p,q}$ satisfying the required conditions.

Returning to the previous example, diagonal 9 of the distance matrix is shown below. It may be seen that the $r_{p,q}$ values for this are as follows — $r_{9,0} = 0$, $r_{9,1} = 4$ and $r_{9,2} = 5$.

	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
i			A	C	E	A	B	P	C	Q	D	E	A	B	C	R
0											0					
1	A											1				
2	B												1			
3	C													1		
4	D														1	
5	E															2

An $r_{p,q}$ value of m, for $q \leqslant k$, indicates that there is an occurrence of the pattern with up to k differences in the text ending at y_{m+p}. In order to solve the k-difference problem, it is thus sufficient to calculate the $r_{p,q}$ values for which $q \leqslant k$.

In order to explain how the $r_{p,q}$ values may be computed, consider the evaluation of the $d_{i,j}$ entries in the dynamic programming matrix. If $d_{i,j} = q$, this entry is on diagonal p, and i is the largest row satisfying these requirements, then by definition $r_{p,q} = i$. The value of $d_{i,j}$ will have been derived from one of the adjacent previous values in the matrix according to (4.11). It may thus have been obtained from one or more of the following possibilities.

1 $d_{i-1,j-1} = q$ and $x_i = y_j$

2 $d_{i-1,j-1} = q - 1$ and $x_i \neq y_j$

3 $d_{i,j-1} = q - 1$

— initialise
$r_{p,-1} = -1 \quad$ for $0 \leqslant p \leqslant n$
$r_{p,|p|-1} = |p| - 1,\ r_{p,|p|-2} = |p| - 2 \quad$ for $-(k+1) \leqslant p \leqslant -1$
$r_{n+1,q} = -1 \quad$ for $-1 \leqslant q \leqslant k$
— calculate $r_{p,q}$ values
for $q = 0$ **to** k
  for $p = -q$ **to** n
    $r = \max\{r_{p,q-1} + 1, r_{p-1,q-1}, r_{p+1,q-1} + 1\}$
    $r = \min\{r, m\}$
    while $(r < m)$ **and** $(r + p < n)$ **and** $(x_{r+1} = y_{r+1+p})$
      $r = r + 1$
    $r_{p,q} = r$
    if $r_{p,q} = m$ **then** there is a k-difference occurrence ending at y_{p+m}

Figure 4.29: Dynamic programming calculation of $r_{p,q}$ values

4 $\quad d_{i-1,j} = q - 1$

Given that the last occurrence of value $q-1$ in diagonal p has previously been found, then the last occurrence on the diagonal of value q may then be found by stepping through subsequent elements on the diagonal while $x_i = y_j$.

Note that the first possible k-difference pattern occurrence is at position $j = m - k$ in the text. This position in row m lies on diagonal $-k$, and thus diagonals $p < -k$ are of no interest in the computation. Also note that adjacent elements in the matrix may differ by only 0 or 1. The algorithm given in Figure 4.29 may be used to compute the $r_{p,q}$ values. An initial value for row r is obtained from previous values $r_{p,q-1}$, $r_{p-1,q-1}$ and $r_{p+1,q-1}$, and is then incremented successively by one at a time until the correct value of $r_{p,q}$ is reached.

The algorithm may be shown to calculate correctly the $r_{p,q}$ values as follows. Consider the case where $q = 0$ and the values of $r_{p,0}$ $(p \geqslant 0)$ are being calculated. From the boundary conditions, r is initialised to 0. The **while** loop finds that $x(1, r_{p,0}) = y(p+1, p + r_{p,0})$ and $x_{r_{p,0}+1} \neq y_{p+r_{p,0}+1}$, thus setting $r_{p,0}$ to the correct value. For diagonals $p < 0$, the correct values for $r_{p,|p|-1}$ and $r_{p,|p|-2}$ will have been set during the initialisation. Let $q = t$, and assume that all $r_{p,t-1}$ values are correct. Consider the calculation of $r_{p,t}$ $(p \geqslant -t)$. Variable r is set to the largest row on diagonal p such that $d_{r,r+p}$ can become value t by possibilities **2**, **3** or **4** described earlier. The **while** loop then steps through successive positions on the diagonal where the pattern and text symbols match to arrive at the correct value for $r_{p,t}$.

The algorithm computes the $r_{p,t}$ values on $n+k+1$ diagonals. Row variable r may be assigned at most m different values for each diagonal, leading to an $O(mn)$ computation time. The complexity is thus the same as that for the straightforward dynamic programming method described earlier. However, it shall now be shown how the framework of this method has been employed to develop an algorithm with improved performance.

The improvement in performance is achieved by altering the way in which the $r_{p,q}$ values are calculated. In the above method, variable r is incremented successively in unit steps until it reaches the correct value of $r_{p,q}$. However, by virtue of appropriate preprocessing, the value of $r_{p,q}$ may be found in constant time. The manner in which this may be achieved shall be described below, but first we turn our attention to the requisite preprocessing.

The preprocessing stage involves the construction of a *suffix tree*, which may be defined as follows. Consider string $s = s(1,l)$, where $s_l = \$$ is a special end marker, not part of the alphabet over which $s(1,l-1)$ is a string. Each suffix of s, $s(i,l)$ for $1 \leqslant i \leqslant l$, defines one of the l leaves of the suffix tree of s. All the edges of the tree are directed away from the root, and the out-degree of each node is either zero (for a leaf node) or $\geqslant 2$. If $s(i,i+p)$ is the longest common proper prefix of two suffixes $s(i,l)$ and $s(j,l)$, i.e. $s(i,i+p) = s(j,j+p)$ and $s_{i+p+1} \neq s_{j+p+1}$, then $s(i,i+p)$ defines an internal node of the tree. Two nodes representing a substring of s, b, and a proper prefix thereof, a, say, are connected by an edge only if there is no node representing a substring of s, c, say, such that c is a proper prefix of b and a is a proper prefix of c. The function of the special terminator symbol \$ is to separate in the tree suffixes $s(i,l-1)$ and $s(j,l-1)$ whenever one is a proper prefix of the other. Note that a suffix tree for a given string is unique up to isomorphism of graphs. Note that the suffix tree data structure is considered in greater detail in the following section of this chapter.

As an example, the suffix tree of the string `EWEW$` is depicted below, where each node is labelled with its defining substring together with a parenthesised node number.

```
root ---------------------- (1) $
|  |
|  |
|   --------- (2) W -------- (4) W$
|                 |
|                 |
|                  --------- (5) WEW$
|
|
 ----------- (3) EW ------- (6) EW$
                 |
                 |
                  --------- (7) EWEW$
```

In the construction of the tree, the substring defining a particular node may be represented by the length of the given substring and its start position in the string. These values for the suffix tree given above would then be as follows, where the subscripts represent the appropriate node number.

$$
\begin{aligned}
start_1 &= 5 & length_1 &= 1 \\
start_2 &= 4 & length_2 &= 1 \\
start_3 &= 3 & length_3 &= 2 \\
start_4 &= 4 & length_4 &= 2 \\
start_5 &= 2 & length_5 &= 4 \\
start_6 &= 3 & length_6 &= 3 \\
start_7 &= 1 & length_7 &= 5
\end{aligned}
$$

In the preprocessing stage, the suffix tree of the string $y\#x\$$, where $\#$ and $\$$ are symbols not in the alphabet over which x and y are strings, is constructed using Weiner's (1973) algorithm (which is described comprehensibly by Chen and Seiferas, 1985). This algorithm requires linear space and, for a fixed size alphabet, linear time. For unbounded alphabets, the algorithm may be adapted to run in $O(n \log c)$ time, where c is the number of distinct pattern symbols. The preprocessing stage thus requires $O(n)$ and $O(n \log m)$ time for the cases of constant and unbounded alphabets respectively. Alternative suffix tree preprocessing strategies are considered by Galil and Giancarlo (1988).

The algorithm of Figure 4.29 may now be modified in order to take advantage of the above preprocessing. Just before the **while** loop for diagonal p, r has been assigned a value such that $x(1, r)$ has been matched, up to q differences, with some text substring ending at y_{r+p}. The function of the **while** loop is then to find the largest value, h say, for which $x(r+1, r+h) = y(r+p+1, r+p+h)$. This is equivalent to finding the length of the longest common prefix of the suffixes $x(r+1, m)\$$ and $y(r+p+1, n)\#x\$$ of the preprocessed concatenated string. The role of the symbol $\#$ is to prevent the situation arising whereby an erroneous prefix consisting of symbols from both y and x might be considered. Defining $lca(r,p)$ as the lowest common ancestor (LCA) in the suffix tree of the leaves defined by the above suffixes, then the desired value of h is given by $length_{lca(r,p)}$. The lowest common ancestor algorithm of Harel and Tarjan (1984) or of Schieber and Vishkin (1988) may therefore be used in the k-difference algorithm as an alternative means of deriving the lengths of the requisite matching substrings.

The suffix tree has $O(n)$ nodes. A transformation, in linear time, of the tree is required by the LCA algorithms in order to support the determination of an LCA in $O(1)$ time. As noted before, $r_{p,q}$ values are calculated on $n+k+1$ diagonals. Furthermore, $k+1$ such values are required for each diagonal, leading to a total of $O(kn)$ LCA queries. This $O(kn)$ time computation is the most significant element in the overall calculation of the $r_{p,q}$ values. The total running time for this k-difference algorithm is therefore $O(kn)$ for a fixed size alphabet and $O(n \log m + kn)$ for unbounded alphabets.

Landau and Vishkin (1986b, 1989) have also developed a parallel version of the above algorithm. The suffix tree may be computed in $O(\log n)$ time using n processors (Landau, Schieber and Vishkin, 1987; Apostolico *et al.*, 1988). Schieber and Vishkin's (1988) parallel algorithm may then be used to preprocess the tree in $O(\log n)$ time using $n/\log n$ processors, allowing a single processor to perform a subsequent LCA query in $O(1)$ time. Employing $n+k+1$ processors thus permits all the required LCA queries to be performed in $O(k)$ time — a bound which may still be achieved by simulating the algorithm on n processors. The total time required is thus $O(\log n + k)$ for n processors. Landau and Vishkin also discuss how geometric decomposition of the problem can lead to an adaptation of the algorithm still requiring n processors, but running in $O(\log m + k)$ time.

4.4 Longest Repeated Substring

4.4.1 Brute Force

A brute force approach to the Longest Repeated Substring problem for the text string y involves an $n \times n$ match matrix, M. The elements of this matrix are defined as follows.

$$M_{i,j} = \begin{cases} 1 & \text{if } y_i = y_j \\ 0 & \text{otherwise} \end{cases} \tag{4.60}$$

The matrix for the example string `PABCQRABCSABTU` is given below, showing the symmetry about the main diagonal. It is thus only elements above, or only those below, the main diagonal that need be calculated. Diagonals of contiguous '1's in the matrix, with the exception of the main one, represent maximal recurring substrings within y.

	j	1	2	3	4	5	6	7	8	9	10	11	12	13	14
i		P	A	B	C	Q	R	A	B	C	S	A	B	T	U
1	P	1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	A	0	1	0	0	0	0	1	0	0	0	1	0	0	0
3	B	0	0	1	0	0	0	0	1	0	0	0	1	0	0
4	C	0	0	0	1	0	0	0	0	1	0	0	0	0	0
5	Q	0	0	0	0	1	0	0	0	0	0	0	0	0	0
6	R	0	0	0	0	0	1	0	0	0	0	0	0	0	0
7	A	0	1	0	0	0	0	1	0	0	0	1	0	0	0
8	B	0	0	1	0	0	0	0	1	0	0	0	1	0	0
9	C	0	0	0	1	0	0	0	0	1	0	0	0	0	0
10	S	0	0	0	0	0	0	0	0	0	1	0	0	0	0
11	A	0	1	0	0	0	0	1	0	0	0	1	0	0	0
12	B	0	0	1	0	0	0	0	1	0	0	0	1	0	0
13	T	0	0	0	0	0	0	0	0	0	0	0	0	1	0
14	U	0	0	0	0	0	0	0	0	0	0	0	0	0	1

From the above, it may be seen that in this example `ABC`, with occurrences $y(2,4)$ and $y(7,9)$, and `AB`, with occurrences $y(2,3)$, $y(7,8)$ and $y(11,12)$, are maximal recurring substrings. The longer of the two, namely `ABC`, is thus the longest repeated substring.

As mentioned earlier, only elements above the main diagonal, i.e. $M_{1\ldots n-1,i+1\ldots n}$, need be considered — a total of $n(n-1)/2$. The evaluation of the matrix therefore requires $O(n^2)$ time.

4.4.2 Suffix Trees

A more efficient solution to the problem involves the construction of a suffix tree of the text string. A description of this data structure was presented in the previous section, the salient points of which are reiterated below. This is followed by a description of McCreight's suffix tree algorithm, and finally the way in which the tree may be utilised in this application is examined.

It may be assumed that the given text string is terminated by a special end marker symbol, \$ say, occurring nowhere else in the string. If this is not the case, then such a symbol may simply be appended to y. The suffix tree for y has n leaves, each representing a different suffix of y, $y(i,n)$ for $1 \leqslant i \leqslant n$. All internal nodes have degree $\geqslant 2$, and represent longest common prefixes of suffixes of y. An edge connects two nodes representing a substring of y, b, and a proper prefix thereof, a, only if there exists no node representing a substring of y, c say, such that c is a proper prefix of b and a is a proper prefix of c. Associated with each edge is a substring which, when appended to that represented by its source node, yields the substring represented by its destination node. The substrings represented by sibling edges must therefore start with different symbols. The foregoing requirements ensure that the suffix tree for a given string is unique up to isomorphism of graphs.

McCreight's (1976) suffix tree construction, to which he refers as 'Algorithm M,' yields a data structure functionally equivalent to Weiner's (1973) substring index. It is, however, more space efficient than the original formulation, and a description of a version of the algorithm is given below.

Let Y denote the suffix tree for string y. Further, let Y_i represent the state of the tree at stage i of its construction. Y_0 is thus an empty tree with only a root node. The tree may be built by adding paths corresponding to suffixes of y one at a time starting with the longest and progressing to the shortest, i.e. suffixes $y(i,n)$ for $1 \leqslant i \leqslant n$. We therefore have the following process, where the *insert* operation inserts a path corresponding to suffix $y(i,n)$ into tree Y_{i-1}.

> **for** $i = 1$ **to** n
> $\quad Y_i = insert(Y_{i-1},\ y(i,n))$

Before proceeding any further, several terms shall first be defined.

locus of a string – the node representing the string in the suffix tree

extension of string u – any string having u as a prefix

extended locus of string u – the locus of the shortest extension of u that is represented in the suffix tree

contracted locus of string u – the locus of the longest prefix of u that is represented in the suffix tree

$head_i$ – the longest prefix of $y(i,n)$ which is also a prefix of $y(j,n)$ for some $j < i$. That is, $head_i$ is the longest prefix of $y(i,n)$ whose extended locus exists within Y_{i-1}.

$tail_i$ – the string such that $y(i,n) = head_i tail_i$. The requirement for the \$ terminator ensures that $tail_i$ is non-empty.

Consider the example of $y =$ `EWEW$`. The various stages in the construction of its suffix tree, Y, are shown in Figure 4.30. In order to insert suffix $y(i,n)$ into the tree, the extended locus of $head_i$ in Y_{i-1} is first located. A new internal node representing $head_i$ is then created, if necessary, and inserted into the former's incoming edge. Finally, a new edge is added from the locus of $head_i$ to a new leaf representing $y(i,n)$. The construction of Y_4, for example, is as follows. The new suffix, $y(4,n)$, is `W$`, $head_4$ is `W` — a prefix of the previous suffix `WEW$` — and $tail_4$ is thus \$. The edge connecting the root to the locus of `WEW$` is split and a new node for `W` is inserted, and lastly an edge is added from this new node to a new leaf for suffix `W$`.

Rather than storing an actual substring for each node of the tree, it is sufficient to represent the substring by a pair of integers indexing its start and end positions in the original string. Alternatively, the start position and the substring length may be employed. Individual nodes of the tree may thus be implemented as a pair of substring indices together with two node pointers — one pointing to the node's first child, the other to its next sibling. This arrangement for the example tree is illustrated in Figure 4.31, where null pointers are represented by

Y_0

```
root
```

Y_1

```
root ----- EWEW$
```

Y_2

```
root ----- EWEW$
   |
    ------ WEW$
```

Y_3

```
root ----- EW ----- EWEW$
   |        |
   |         ------ EW$
   |
    --------------- WEW$
```

Y_4

```
root ----- EW ----- EWEW$
   |        |
   |         ------ EW$
   |
    ------- W ----- WEW$
            |
             ------ W$
```

Y_5

```
root ----- EW ----- EWEW$
|  |        |
|  |         ------ EW$
|  |
|   ------- W ----- WEW$
|           |
|            ------ W$
|
 ------------------ $
```

Figure 4.30: Stages in the construction of the suffix tree for EWEW$

```
(0,0) * 0
      |
      |
      |
      v
     (1,2) * *-------------------> (2,2) * *---------> (5,5) 0 0
      EW   |                         W   |               $
           |                             |
           |                             |
           v                             v
          (1,5) 0 *-----> (3,5) 0 0     (2,5) 0 *-----> (4,5) 0 0
          EWEW$             EW$          WEW$             W$
```

Figure 4.31: Node implementation for EWEW$ suffix tree

0. Although not actually stored, the substring for each node has been included in the figure for convenience.

Once the extended locus of $head_i$ in Y_{i-1} has been found, the necessary modification of the tree at step i may be performed in constant time. So, if the search for the extended locus may also be completed in constant time, averaged over all the steps, then the suffix tree may be built in time linear in n. Auxiliary ***suffix links*** are employed in the data structure in order to do just this, and their use relies on the following relationship between $head_{i-1}$ and $head_i$.

If $head_{i-1} = az$, for some symbol a and some (possibly empty) string z, then z is a prefix of $head_i$. This may be proved as follows. If $head_{i-1} = az$, then $\exists j < i$ such that az is a prefix of both $y(i-1, n)$ and $y(j-1, n)$. String z is then a prefix common to $y(i, n)$ and $y(j, n)$, and is, by definition, therefore a prefix of $head_i$.

A suffix link is added in the tree from each internal node which is the locus of az, for some symbol a and string z, to the locus of z. Note that the latter is never within the subtree rooted at the former. These auxiliary links allow a short-cut search to be performed at stage i for the extended locus of $head_i$ starting from the locus of $head_{i-1}$, which will have been found at the previous stage. The implementation of the exemplar suffix tree with the suffix links for its internal nodes, namely the loci of EW and W, is shown in Figure 4.32.

The operation of the algorithm relies on the facts that (1) in Y_i, only the locus of $head_i$ can fail to have a valid suffix link, and that (2) in step i, the contracted locus of $head_i$ in Y_{i-1} (i.e. the locus of the longest prefix of $head_i$ in Y_{i-1}) is visited. At the outset, when $i = 1$, these two requirements obviously hold. At stage i of the algorithm, the following three substeps are carried out.

I three strings u, v and w are found such that

(a) $head_{i-1} = uvw$

(b) if the contracted locus of $head_{i-1}$ in Y_{i-2} is not the root, then it is the locus of uv. Otherwise, $v = \epsilon$

```
       ---------------------------------------
      |              suffix link              |
      v                                       |
   (0,0) * 0                                  |
          |    ---------------------------    |
          |   |        suffix link        |   |
          |   |                           |   |
          v   |                           v   |
    (1,2) * * *--------------------> (2,2) * * *---------> (5,5) 0 0
     EW         |                       W     |               $
                |                             |
                |                             |
                v                             v
          (1,5) 0 *-----> (3,5) 0 0     (2,5) 0 *-----> (4,5) 0 0
          EWEW$            EW$           WEW$              W$
```

Figure 4.32: Node implementation for EWEW$ suffix tree showing the suffix links

(c) $|u| \leqslant 1$, $u = \epsilon$ iff $head_{i+1} = \epsilon$

Since $head_{i-1} = uvw$, then, as shown earlier, vw is a prefix of $head_i$ (by (c), u is at most a single symbol). Thus, $head_i$ is given by vwz, for some (possibly empty) string z. A test on the emptiness of v is now performed, with the following appropriate actions.

- $v = \epsilon$ — the root node is designated as c (the locus of v), and the algorithm proceeds to **II**.
- $v \neq \epsilon$ — from (b), the locus of uv must have already existed in Y_{i-2}. From (1), the suffix link of this internal node must be defined in Y_{i-1}, since the node must have been added prior to step $i-1$. From (2), this node will have been visited in stage $i-1$. Its suffix link is then traversed to an internal node c (the locus of v) and the algorithm proceeds to **II**.

II ('rescanning') It was shown above that $head_i$ is given by vwz. By definition, the extended locus of vw thus exists in Y_{i-1}. There is therefore some downward sequence of edges from c (the locus of v) leading to the locus of some extension of vw. To rescan w, the child of c, e say, such that e is the locus of an extension, vp say, of v appended with the first symbol of w is found. If $|p| < |w|$, then a recursive rescan of q, where $w = pq$, is started from node e. If, on the other hand, $|p| \geqslant |w|$, then w is a prefix of p, and the extended locus of vw has been found, completing the rescan and taking time linear in the number of nodes visited. A new internal node is created as the locus of vw if one does not already exist, which is the case only if $z = \epsilon$. The locus of vw is designated by d, and the algorithm proceeds to **III**.

III ('scanning') If the suffix link of the locus of uvw (i.e. $head_{i-1}$) is currently undefined, then it is set to point to node d, ensuring the truth of (1). A downwards search starting from d to find the extended locus of vwz (i.e. of $head_i$) is then initiated. While rescanning, the length of w is already known as it has previously been scanned, whereas in scanning, the length of z is unknown. The downward search is thus dictated by the symbols of $tail_{i-1}$ (of which z is a prefix) one at a time from left to right. The extended locus of vwz is then found when this downward excursion may proceed no further. The previous node encountered on this journey is the contracted locus of $head_i$, ensuring the truth of (2). If necessary, a new internal node is created to be the locus of vwz. A new edge is then added from the locus of vwz to a new leaf representing suffix $y(i, n)$, thus completing stage i.

McCreight gives the following example to illustrate the above process. Consider step 14 in the construction of the suffix tree for y = `BBBBBABABBBAABBBBB$`. The tree Y_{13} is shown in Figure 4.33, where leaf nodes have been labelled with the relevant value of i for the suffix $y(i, 19)$ that they represent, and the relevant suffix links have been included. At step 14, suffix $y(14, 19)$, namely `BBBBB$`, is to be inserted into Y_{13}. Initially, uv is set to `AB` as this is the string represented by the contracted locus of $head_{13}$ in Y_{12}, so u = `A` and v = `B`. The value of $head_{13}$ (uvw) is equal to `ABBB`, so w must be `BB`. In **I**, v is non-empty, so the suffix link from the locus of uv (`AB`) is traversed to node c (the locus of `B`). From here, w is rescanned in **II**, encountering one intermediate node e and ending up at node d, the locus of `BBB`. In **III** a new suffix link is added from the locus of $head_{13}$ to node d, and the downward scan from d is then started using the symbols of $tail_{13}$ (`BB$`), finding that z = `BB`. A new internal node is then created as the locus of vwz ($head_{14}$), which in this case is `BBBBB`, and inserted into the incoming edge of the extended locus of vwz (the leaf for $y(1, 19)$). Finally, a new edge is added from the locus of `BBBBB` to a new leaf representing suffix $y(14, 19)$.

For stage i, let res_i denote the shortest suffix (i.e. $wz \cdot tail_i$) involved in the rescanning operation. For every intermediate node e visited whilst rescanning w, p is a non-empty string occurring in res_i, but not in res_{i+1}. For each intermediate node, p has a minimum length of 1, thus the maximum length of res_{i+1} is $\mid res_i \mid - \, int_i$, where int_i is the number of intermediate nodes encountered during the rescan of stage i. The overall total number of intermediate nodes visited during rescanning is thus $\sum_{i=1}^{n} int_i$, which is bounded from above by n. Also, in stage i, the number of symbols which must be scanned in order to locate $head_i$, i.e. $\mid z \mid$, is equal to $\mid head_i \mid - \mid head_{i-1} \mid + 1$. Summing over the complete algorithm gives $\sum_{i=1}^{n} \mid head_i \mid - \mid head_{i-1} \mid + 1 \; = head_n - head_0 + n = n$.

The above shows that the rescanning and scanning operations of the overall process consume at most time linear in n. It has also previously been shown that each of the n stages of the algorithm takes constant time, excluding that required for rescanning and scanning. The time taken by the entire algorithm is therefore linear in n. The running time of the algorithm is also linear in the size of the alphabet of y as each node may have up to $\mid C \mid$ children, for alphabet C.

Although efficient in space, the tree implementation depicted in Figure 4.32 is slow to search. As a compromise between space requirement and amenability to be searched rapidly, McCreight suggests an implementation in which the edges of the tree are encoded as entries in a hash table (using Lampson's hashing algorithm (see exercise 6.4.13 of Knuth, 1973)).

Having described the construction of the suffix tree, we are now in a position to examine how this data structure may be used in the solution of the longest repeated substring problem.

```
root ----- A ----- 12
  |        |                        ----------------
  |         ------ AB ----- 6      '                |
  |               ' |            '                  |
  |      suffix '   |          '                    |
  |      link '      ------ ABBB ----- 8            |
  |         '       (head_13)  |                    |
  |        |                    ------ 13           |
  |        |                                        |
  |        v                                        |
   ------- B ----- BA ----- 11                      | new
  (node c) |        |                               | suffix
           |         ------ BAB ------ 5            | link
           |                 |                      |
           |                  -------- 7            |
           |                                        |
           |    (node e)                            |
            ------ BB ----- BBA ------ 10           |
                    |        |                      |
                    |         -------- 4            |
                    |                               |
                    |            -------------------|
                    |           '
                    |         '
                    |        |
                    |        v
                     ------ BBB ------ BBBA ----- 9
                    (node d) |           |
                             |            ------- 3
                             |
                             |
                              -------- BBBB ----- 2
                                         |
                                          ------- 1
```

Figure 4.33: Y_{13} for string BBBBBABABBBAABBBBB$

```
  root ----- AB ------ 11
/  |  \       |
|  |  |        ------- ABC ----- 2
|  |  |                |
|  |  |                 -------- 7
|  |  |
|  |   ------ B ------ 12
|  |          |
|  |           ------- BC ------ 3
|  |                    |
|  |                     ------- 8
|  |
|   ------------------ C ------- 4
|                      |
|                       -------- 9
|
 --------------------- 1, 5, 6, 10, 13, 14, 15
```

Figure 4.34: Suffix tree for string PABCQRABCSABTU$

It is apparent that the longest substring represented by an internal node of Y is the longest repeated substring of y. It therefore suffices to keep track of the maximum length of the internal node substrings as the tree is being built, which, as we have seen, may be accomplished in linear time for a fixed alphabet.

The suffix tree for the previous example string of $y =$ PABCQRABCSABTU$ is shown in Figure 4.34. Again, leaf nodes have been labelled with the relevant value of i for their respective suffixes $y(i, 15)$. Also, for convenience, all leaves having no parent other than the root node have been grouped together at the end of a single edge. From the figure it may be seen that ABC is the longest substring represented by an internal node, and is thus the longest repeated substring of y.

The longest common substring of two strings, x and y, may also be determined in a similar fashion in $O(m+n)$ time for a fixed alphabet. In this case, the suffix tree for the string $x\#y\$$, where symbols $\#$ and \$ are not members of the alphabet over which x and y are strings, is constructed. Again, the required result is given by the maximal length substring represented by an internal node of the suffix tree.

Finally, Baker's (1992) approach to the problem of finding all the maximal repeated substrings, longer than some predetermined threshold value, of an input string involves searching the suffix tree of the string. The substring, u say, represented by the lowest common ancestor of two leaves in Y is the longest common prefix of the two suffixes represented by those leaves. It is thus a substring occurring at two distinct positions with differing right contexts in the input string. To determine whether u is a maximal repeated substring, it remains to establish whether the left contexts of its two occurrences in y also differ. If they do not, then u is merely a suffix of some longer recurring substring.

The necessary check on the left contexts may be performed by recursing over the tree, building up lists of suffixes grouped by left context and comparing these lists. For each internal node, lists of the positions of the suffixes represented by the leaves in the subtree rooted at the given internal node and sharing identical left contexts are created. Maximal repetitions are then identified by enumerating, at each subtree, pairings of positions from pairs of lists with distinct left contexts. For example, consider the suffix tree for the string `PABCQRABCSABTU$` (Figure 4.34). Looking for maximal repetitions of length 2 or greater, we find that `AB` is a maximal repeated substring (the left context of $y(11,15)$ is `S`, and those of $y(2,15)$ and $y(7,15)$ are `P` and `R` respectively); whereas `BC` is not (the left contexts of $y(3,15)$ and $y(8,15)$ are both `A`). By restricting this activity to only those internal nodes representing substrings longer than the preset threshold, the search of the tree may be performed in time linear in the number of pairs of maximal repetitions found, r. Taking into account the time required to construct the suffix tree gives an overall temporal complexity of $O(n+r)$ for a fixed alphabet. The space requirement is that of the suffix tree, which is linear in n. Baker has also extended this approach in order to find parameterised matches — useful, for example, for identifying similar sections of code in large computer programs.

Chapter 5

Back to the Original Problem

To be still searching what we know not by what we know ...
— John Milton (1608-1674), *The Doctrine and Discipline of Divorce.*

Having reviewed a variety of string searching algorithms in the foregoing chapters, we shall now examine in this chapter potential ways in which the original problem may be solved. Before doing so, however, it might first be appropriate to consider how the study of these algorithms is relevant to the tentative information processing system mentioned at the outset.

5.1 The System

The processing system in question has many claimed application areas, including information retrieval and the recognition of approximate patterns. An examination of efficient methods of retrieving substrings of a text is therefore relevant to the development of this system. For example, the Boyer-Moore-Horspool algorithm provides a fast and easily implemented method for searching a text for a given substring. The suffix tree data structure (described in §4.4.2) is also useful here, allowing multiple searches of a given text to be performed efficiently. It is also of interest to note in passing that this data structure permits linear time sequential data compression to be performed. (The intended operation of the processing system is based on a principle of data compression.)

The string distance and approximate string matching algorithms are clearly relevant to the recognition of approximate patterns, where occurrences up to some predefined distance away from a given pattern are to be identified. Furthermore, the previous work done with don't care symbols is particularly pertinent since support in the system for such wildcards is intended. Of singular relevance to the problem are the longest common subsequence and longest repeated substring algorithms, as we shall see in the next section.

A related problem, the solution of which is incidentally an objective of the system, is to find the patterns common to an arbitrary number of input strings. This is effectively a variation of the N-lcs problem. It is therefore interesting to recall that we have seen that the $|N\text{-lcs}|$ problem is NP complete (Maier, 1978).

5.2 The Problem

At the outset, it was stated that it seemed more intuitive to employ a finite alphabet, rather than allow arbitrary strings of ASCII characters represent the atomic symbols. As we have seen, the finite alphabet case certainly simplifies matters in many string searching applications. For example, this permits Boyer-Moore type matching techniques to be applied.

It was also initially discussed whether the *pattern* of the problem should be a substring or a subsequence. For the former case, it has been shown that the problem has been solved using suffix tree techniques. To reiterate, the longest repeated substring may be found in linear time by constructing the suffix tree of the text string (McCreight, 1976), and the maximal repetitions in the string may be found from its tree in $O(n+r)$ time overall, where r is the number of pairs of repetitions found (Baker, 1992). The latter case, namely that of repeated subsequences, is considered below.

It was originally stated that the problem was to find the maximal disjoint recurring subsequences of an input string. For the time being, it shall also be stipulated that overlapping subsequences are to be excluded. A description of a possible approach to solving this restricted problem is given below, and this is then followed by a discussion of a more efficient approach which also caters for the unrestricted problem.

If the second occurrence of a maximal recurring subsequence in string y starts at position i, then that subsequence may be found by calculating the longest common subsequence of $y(1, i-1)$ and $y(i, n)$. A possible approach to the problem would therefore be to compute $\text{lcs}(y(1, i-1), y(i, n))$ for $i = 2$ to n. Adopting a quadratic time lcs algorithm would result in the overall process requiring $O(n^3)$ time. Assuming the use of an $O(n^2)$ lcs algorithm provides a worst case bound for a practical implementation; the average case may however be more favourable. As we have previously seen, in addition to the quadratic time lcs algorithms (e.g. Hirschberg, 1975), there are also those exhibiting more attractive average case performance. We have, for example, Hunt and Szymanski's (1977) $O(n \log n)$ time algorithm, which degrades to $O(n^2 \log n)$ in the worst case, and Ukkonen's (1983, 1985a) $O(nd)$ time method, which is quadratic in the worst case. Recall also that the algorithm of Masek and Paterson (1980, 1983) is sub-quadratic in all cases, but at the cost of a prohibitively large constant of proportionality.

Under the above restriction on the subsequences, overlapping occurrences of a maximal recurring subsequence would not be found. Relaxing this restriction thus means that we must also be able to find interleaved disjoint occurrences of a maximal recurring subsequence, such as `ABC` in the string `AABBCC` (there are in fact 4 possible pairs of disjoint occurrences of `ABC` in this string: $y_1y_3y_5$, $y_2y_4y_6$; $y_1y_3y_6$, $y_2y_4y_5$; $y_1y_4y_5$, $y_2y_3y_6$; $y_1y_4y_6$, $y_2y_3y_5$). In general, when interleaved occurrences are permitted and when each symbol of a maximal recurring subsequence occurs exactly twice in the string, there may be up to $2^{l/2}$ distinct pairs of disjoint occurrences in the string of the maximal subsequence – giving an exponential number of pairs in the worst case since $l \leqslant \lfloor n/2 \rfloor$. When subsequence symbols occur more than twice in the string the potential number of pairs becomes even greater. Also note that there may be more than one longest recurring subsequence in a string. For example, both `ABC` and `DEF` are longest recurring subsequences in the string `PDEQFABCRSATBCDEUF`.

An approach to finding a single pair of occurrences of a maximal recurring subsequence of string y is to compute the heaviest common subsequence of y with itself, with the weight of a match (i, j) equal to 0 if $i = j$, and equal to 1 otherwise. As we have seen, this may be performed by using an adaptation of the dynamic programming technique, or by using Jacobson and Vo's (1992) hcs algorithm. The former requires $O(n^2)$ time, whereas the time requirement of the latter is $O((r + n) \log n)$ where r is the number of matches.

To illustrate the technique, the dynamic programming hcs weight array for the case of $y =$ `AABBCC` is given below. The array is generated using the recurrence relation 4.58 together with the above match weighting.

	j	0	1	2	3	4	5	6
i			A	A	B	B	C	C
0		0	0	0	0	0	0	0
1	A	0	0	1	1	1	1	1
2	A	0	1	1	1	1	1	1
3	B	0	1	1	1	2	2	2
4	B	0	1	1	2	2	2	2
5	C	0	1	1	2	2	2	3
6	C	0	1	1	2	2	3	3

An actual maximal recurring subsequence may then be recovered from the array by tracing back through its dependencies, as in the dynamic programming method for calculating inter-string edit distances.

Note that finding a maximal recurring subsequence is thus equivalent to finding a maximal monotonically increasing path in the graph composed of nodes corresponding to matches (i, j) where $y_i = y_j$ and $i \neq j$. This is reminiscent of the Hunt-Szymanski (1977) lcs algorithm, which could therefore be modified to perform the required task in the same asymptotic time as the Jacobson-Vo hcs algorithm. A pair of such paths corresponding to a distinct pair of occurrences of the maximal recurring subsequence `ABC` in the string `AABBCC` is shown below. The component nodes of one of the pair of disjoint paths are represented by the solid circles, and those of the other by the hollow ones. The actual pair of paths indicated below represent the pair of occurrences $y_1y_4y_5$ and $y_2y_3y_6$.

	j	1	2	3	4	5	6
i		A	A	B	B	C	C
1	A		●				
2	A	○					
3	B				○		
4	B			●			
5	C						●
6	C					○	

To conclude, the use of an hcs algorithm thus allows a maximal recurring subsequence of a string to be found, and this may be achieved in $O((r+n)\log n)$ time by employing the Jacobson-Vo method. Note, however, that finding all the possible disjoint pairs of occurrences of maximal recurring subsequences is not in general a viable proposition. This is equivalent to enumerating all the possible pairs of disjoint maximal monotonically increasing paths in the graph of the weighted matches. As intimated earlier, such a task is of exponential complexity in the worst case.□

* * *

Achilles (to the *Tortoise*): All that's left is to find out if the string has Buddha-nature or not. — Douglas R. Hofstadter, *Gödel, Escher, Bach.*

* * *

Acknowledgement

I am indebted to Dan Hirschberg and Phong Vo for suggesting approaches for the solution of the maximal repeated subsequence problem. The contributions of many other Internet respondents who provided numerous pointers to the literature are also gratefully acknowledged.

Appendix A

String Nomenclature

A glossary of terms in common use in string matching problems is presented below. The reader is referred to texts such as Aho, Sethi and Ullman (1986) for further explanation of these terms.

ALPHABET – any finite set of symbols, also known as a *character class*.

CONCATENATION – the concatenation of two strings x and y, xy, is the string obtained by appending y to x. The concatenation of two languages A and B, AB, is defined as

$$AB = \{xy \ : \ x \text{ is in } A, \ y \text{ is in } B\}$$

i.e. the set of all strings formed by concatenating any string from A with any one from B.

EMPTY STRING, ϵ – the string with zero length.

EXPONENTIATION – following on from an analogy between string concatenation and multiplication, exponentiation is defined as follows

$$x^0 = \epsilon, \quad x^i = x^{i-1}x \ \text{ for integer } i > 0$$

Thus, $x^1 = \epsilon x = x, \quad x^2 = xx, \quad x^3 = xxx \quad \ldots$

FIBONACCI STRING – a string defined by $f_1 = B$, $f_2 = A$, $f_n = f_{n-1}f_{n-2} \ \ n > 2$, e.g. f_5 = `ABAAB`.

HAMMING DISTANCE – a distance metric between two strings of equal length, equal to the number of symbol positions at which the two strings differ, e.g. the Hamming distance between `master` and `pastes` is 2.

KLEENE CLOSURE of a language A, A^* – the language formed by the union of zero and more concatenations of A, i.e.

$$A^* = \bigcup_{i=0}^{\infty} A^i$$

LANGUAGE – a set of strings over a given alphabet.

LENGTH of a string x, $|x|$ – the number of instances of symbols in x.

LEVENSHTEIN DISTANCE – a distance metric between two strings, not necessarily of the same length, given by the minimum number of symbol insertions, deletions and substitutions required to transform one string to the other, e.g. the Levenshtein distance between `zeitgeist` and `preterit` is 6.

POSITIVE CLOSURE of a language A, A^+ – the language formed by the union of one and more concatenations of A, i.e.

$$A^+ = \bigcup_{i=1}^{\infty} A^i$$

PREFIX of a string x – a string obtained by deleting zero or more symbols from the end of x, e.g. `jorm` is a prefix of `jormungand`. A *proper* prefix of x is a non-empty prefix of x not equal to x itself.

SQUARE – a string comprising a repeated non-empty substring, i.e. one of the form x^2, e.g. `dikdik`.

STRING over a given alphabet – a finite sequence of symbols drawn from the alphabet. The terms *sentence* and *word* are sometimes used synonymously with *string*. Note that both ϵ and x are prefixes, suffixes, substrings and subsequences of string x.

SUBSEQUENCE of a string x – a string obtained by deleting zero or more symbols, which need not be contiguous, from x, e.g. `nnnaa` is a subsequence of `ginnungagap`. A *proper* subsequence of x is a non-empty subsequence not equal to x itself.

SUBSTRING of a string x – a string obtained by deleting zero or more symbols from the beginning and end of x (i.e. by deleting a prefix and a suffix), e.g. `gna` is a substring of `ragnarok`. A *proper* substring of x is a non-empty substring not equal to x itself.

SUFFIX of a string x – a string obtained by deleting zero or more symbols from the beginning of x, e.g. `asill` is a suffix of `yggdrasill`. A *proper* suffix of x is a non-empty suffix not equal to x itself.

SUPERSEQUENCE of a string x – any string y such that x is a subsequence of y, e.g. `facetious` is a supersequence of `aeiou`.

UNION of languages A and B, $A \cup B$ – the language containing all the strings of both A and B, i.e.

$$A \cup B = \{x \ : \ x \text{ is in } A \text{ or } x \text{ is in } B\}$$

Appendix B

Asymptotic Notation

This appendix defines some of the notation commonly used in the study of asymptotics. A more thorough treatment of this subject may be found in Graham, Knuth and Patashnik (1990).

It is often useful to classify certain functions of n in terms of the relative speed with which they approach infinity for increasing values of n. The relational operator $\prec$, introduced by Paul du Bois-Reymond (1871) may be used for this purpose and is defined below.

$$f(n) \prec g(n) \quad \Leftrightarrow \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \tag{B.1}$$

The use of this operator is demonstrated in the following asymptotic hierarchy.

$$1 \;\prec\; \log n \;\prec\; n^a \;\prec\; n^b \;\prec\; b^n \;\prec\; n^n$$

where a and b are constants such that $0 \;<\; a \;<\; 1 \;<\; b$.

Another useful notation is Big Oh, introduced by Paul Bachman (1894), which provides a convenient means of expressing upper bounds. It is defined as follows.

$$f(n) = O(g(n)) \quad \Leftrightarrow \quad |\,f(n)\,| \;\leqslant\; c\,|\,g(n)\,| \quad \text{for some constant } c > 0 \tag{B.2}$$

Note that it has become customary to use the '=' sign in the above type of expression. It does not, however, stand for equality and should be read as 'is' (unidirectional) rather than 'equals' (symmetrical). It is sometimes implicitly assumed that the Big Oh relation is for sufficiently large n, i.e. as $n \to \infty$. In this case the strict condition is that

$$|\,f(n)\,| \;\leqslant\; c\,|\,g(n)\,| \quad \text{for } n \geqslant n_0 \tag{B.3}$$

for some constant n_0.

The corresponding notation for lower bounds is Big Omega. This is defined as follows.

$$f(n) = \Omega(g(n)) \quad \Leftrightarrow \quad |\,f(n)\,| \;\geqslant\; c\,|\,g(n)\,| \quad \text{for some constant } c > 0 \tag{B.4}$$

If $g(n)$ is a lower bound for $f(n)$, then $f(n)$ must be an upper bound for $g(n)$, i.e.

$$f(n) = \Omega(g(n)) \quad \Leftrightarrow \quad g(n) = O(f(n)) \tag{B.5}$$

Finally, Big Theta is used to indicate an exact order of growth, ie.

$$f(n) = \Theta(g(n)) \quad \Leftrightarrow \quad f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)) \tag{B.6}$$

* * *

I give you the end of a golden string;
Only wind it into a ball,
It will lead you in at Heaven's gate,
Built into Jerusalem's wall.
— William Blake (1757-1827)

References

Aho A.V., Hopcroft J.E., Ullman J.D. (1974) "*The design and analysis of computer algorithms*," Addison-Wesley, Reading, MA.

Aho A.V., Corasick M.J. (1975) "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, Vol. 18, No. 6, p. 333-40, June 1975.

Aho A.V., Hirschberg D.S., Ullman J.D. (1976) "Bounds on the complexity of the longest common subsequence problem," *Journal of the ACM*, Vol. 23, No. 1, p. 1-12, January 1976.

Aho A.V. (1980) "Pattern matching in strings," in Book R.V. (ed.) *Formal Language Theory*, p. 325-47, Academic Press, New York.

Aho A.V., Sethi R., Ullman J.D. (1986) *Compilers – Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA.

Aho A.V. (1990) "Algorithms for finding patterns in strings," Chapter 5 (p. 255-300) of Leeuwen J. van (ed.) *Handbook of Theoretical Computer Science*, Elsevier Science Publishers, Amsterdam.

Altschul S.F., Gish W., Miller W., Myers E.W., Lipman D.J. (1990) "Basic local alignment search tool," *Journal of Molecular Biology*, Vol. 215, p. 403-10.

Apostolico A. (1985) "The myriad virtues of subword trees," in Apostolico A., Galil Z. (eds.) *Combinatorial Algorithms on Words*, NATO ASI Series, Vol. F12, p. 85-96, Springer-Verlag, Berlin.

Apostolico A., Giancarlo R. (1986) "The Boyer-Moore-Galil string searching strategies revisited," *SIAM Journal on Computing*, Vol. 15, No. 1, p. 98-105, February 1986.

Apostolico A., Guerra C. (1987) "The longest common subsequence problem revisited," *Algorithmica*, Vol. 2, p. 315-36.

Apostolico A., Iliopoulos C., Landau G.M., Schieber B., Vishkin U. (1988) "Parallel construction of a suffix tree with applications," *Algorithmica*, Vol. 3, p. 347-65.

Arlazarov V.L., Dinic E.A., Kronod M.A., Faradzev I.A. (1970) "On economic construction of the transitive closure of a directed graph," (Russian) *Doklady Akademii*

Nauk SSR, Vol. 194, p. 487-8, also translated in *Soviet Math. Doklady*, Vol. 11, p. 1209-10, 1975.

Arratia R., Waterman M.S. (1985a) "Critical phenomena in sequence matching," *The Annals of Probability*, Vol. 13, No. 4, p. 1236-49.

Arratia R., Waterman M.S. (1985b) "An Erdős-Rényi law with shifts," *Advances in Mathematics*, Vol. 55, p. 13-23.

Arratia R., Gordon L., Waterman M. (1986) "An extreme value theory for sequence ma-
tching," *The Annals of Statistics*, Vol. 14, No. 3, p. 971-93.

Bachman P. (1894) *Die analytische Zahlentheorie*, Teubner, Leipzig.

Baeza-Yates R.A. (1989a) "Improved string matching," *Software — Practice and Experience*, Vol. 19, No. 3, p. 257-71, March 1989.

Baeza-Yates R.A. (1989b) "String searching algorithms revisited," *Lecture Notes in Computer Science*, Vol. 382, p. 75-96.

Baeza-Yates R.A. (1991) "Searching subsequences," *Theoretical Computer Science*, Vol. 78, No. 2, p. 363-76.

Baker B.S. (1992) "A program for identifying duplicated code," *Proceedings of the 24th Symposium on the Interface: Computer Science and Statistics*, College Station, Texas, 18-21 March 1992.

Bellman R., Dreyfus S. (1962) "*Applied Dynamic Programming*," Princeton University Press, Princeton NJ.

Blumer A., Blumer J., Ehrenfeucht A., Haussler D., McConnel R. (1984a) "Building a complete inverted file for a set of text files in linear time," *Proceedings of the 16th ACM Symposium on the Theory of Computing*, p. 349-58.

Blumer A., Blumer J., Ehrenfeucht A., Haussler D., McConnel R. (1984b) "Building the minimal DFA for the set of all subwords of a word on-line in linear time," in Paredaens J. (ed.), Proceedings of the 11th International Colloquium on Automata, Languages and Programming, *Lecture Notes in Computer Science*, Vol. 172, p. 109-18, Springer-Verlag, Berlin.

Blumer A., Blumer J., Ehrenfeucht A., Haussler D., Chen M.T., Seiferas J. (1985) "The smallest automaton recognizing the subwords of a word," *Theoretical Computer Science*, Vol. 40, No. 1, p. 31-56.

Blumer A., Blumer J., Haussler D., McConnel R., Ehrenfeucht A. (1987) "Complete inverted files for efficient text retrieval and analysis," *Journal of the ACM*, Vol. 34, No. 3, p. 578-95.

Bois-Reymond P. du (1871) "Sur la grandeur relative des infinis des fonctions," *Annali di Matematica pura e applicata*, series 2, Vol. 4, p. 338-53.

Boyer R.S., Moore J.S. (1977) "A fast string searching algorithm," *Communications of the ACM*, Vol. 20, No. 10, p. 762-72, October 1977.

Burkowski F.J. (1982) "A hardware hashing scheme in the design of a multiterm string comparator," *IEEE Transactions on Computers*, Vol. C-31, No. 9, p. 825-34, September 1982.

Chen M.T., Seiferas J. (1985) "Efficient and elegant subword-tree construction," in Apostolico A., Galil Z. (eds.) *Combinatorial Algorithms on Words*, NATO ASI Series, Vol. F12, p. 97-107, Springer-Verlag, Berlin.

Cook S.A. (1972) "Linear time simulation of deterministic two-way pushdown automata," *Information Processing*, Vol. 71, p. 75-80, North-Holland, Amsterdam.

Crochemore M. (1986) "Transducers and repetitions," *Theoretical Computer Science*, Vol. 45, p. 63-86.

Curry T., Mukhopadhyay A. (1983) "Realization of efficient non-numeric operations through VLSI," *Proceedings of VLSI '83*.

Davies G., Bowsher S. (1986) "Algorithms for pattern matching," *Software — Practice and Experience*, Vol. 16, No. 6, p. 575-601, June 1986.

Fischer M.J., Paterson M.S. (1974) "String-matching and other products," in Karp R.M. (ed.), *Complexity of Computation*, SIAM-AMS Proceedings, Vol. 7, p. 113-25.

Foster M.J., Kung H.T. (1980) "The design of special-purpose VLSI chips," *IEEE Computer*, Vol. 13, p. 26-40, January 1980.

Galil Z. (1979) "On improving the worst case running time of the Boyer-Moore string searching algorithm," *Communications of the ACM*, Vol. 22, No. 9, p. 505-8.

Galil Z., Giancarlo R. (1986) "Improved string matching with k mismatches," *Sigact News*, Vol. 17, p. 52-4.

Galil Z., Giancarlo R. (1988) "Data structures and algorithms for approximate string matching," *Journal of Complexity*, Vol. 4, p. 33-72.

Gonnet G.H., Baeza-Yates R. (1991) "Text algorithms," Chapter 7 (p. 251-88) of *Handbook of Algorithms and Data Structures in Pascal and C*, 2^{nd} edition, Addison-Wesley, Wokingham UK.

Graham R.L., Knuth D.E., Patashnik O. (1990) *Concrete Mathematics*, Addison-Wesley, Reading, MA (5^{th} printing with corrections, July 1990).

Guibas L.J., Odlyzko A.M. (1977) "A new proof of the linearity of the Boyer-Moore string searching algorithm," *Proceedings of the* 18^{th} *Annual IEEE Symposium on Foundations of Computer Science*, p. 189-95 (also *SIAM Journal on Computing*, Vol. 9, No. 4, p. 672-82, 1980).

Halaas A. (1983) "A systolic VLSI matrix for a family of fundamental search problem," *Integration VLSI Journal*, Vol. 1, No. 4, p. 269-82, December 1983.

Hall P.A.V., Dowling G.R. (1980) "Approximate matching," *Computing Surveys*, Vol. 12, No. 4, p. 381-402, December 1980.

Hamming R. (1982) "*Coding and Information Theory*," Prentice Hall, Englewood Cliffs NJ.

Harel D., Tarjan R.E. (1984) "Fast algorithms for finding nearest common ancestors," *SIAM Journal on Computing*, Vol. 13, No. 2, p. 338-55.

Harrison M.C. (1971) "Implementation of the substring test by hashing," *Communications of the ACM*, Vol. 14, No. 12, p. 777-9, December 1971.

Haskin R.L. (1981) "Special purpose processors for text retrieval," *Database Engineering*, Vol. 4, No. 1, p. 16-29, September 1981.

Haskin R.L., Hollaar L.A (1983) "Operational characteristics of a hardware-based pattern matcher," *ACM Transactions on Database Systems*, Vol. 8, No. 1, p. 15-40, March 1983.

Haton J.P. (1973) "*Contribution à l'analyse, paramétrisation et la reconnaissance automatique de la parole*," Thèse de doctorat d'état, Université de Nancy, Nancy France.

Hirata M., Yamada H., Nagai H., Takahashi K. (1988) "A versatile data string-search VLSI," *IEEE Journal of Solid-State Circuits*, Vol. 23, No. 2, p. 329-35, April 1988.

Hirschberg D.S. (1975) "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM*, Vol. 18, No. 6, p. 341-3, June 1975.

Hirschberg D.S. (1978) "An information theoretic lower bound for the longest common subsequence problem," *Information Processing Letters*, Vol. 7, p. 40-1.

Hirschberg D.S. (1983) "Recent results on the complexity of common subsequence problems," in Sankoff D., Kruskall J.B. (eds.) *Time warps, string edits, and macromolecules: the theory and practice of sequence comparison*, Chapter 12, p. 325-30, Addison-Wesley, Reading MA.

Hollaar L.A. (1979) "Text retrieval computers," *IEEE Computer*, Vol. 12, p. 40-50.

Horspool R.N. (1980) "Practical fast searching in strings," *Software — Practice and Experience*, Vol. 10, No. 6, p. 501-6.

Hsu W., Du M. (1984) "Computing a longest common subsequence for a set of strings," *BIT*, Vol. 24, p. 45-59.

Hume A., Sunday D. (1991) "Fast string searching," *Software — Practice and Experience*, Vol. 21, No. 11, p. 1221-48, November 1991.

Hunt J.W., McIlroy M.D. (1976) "*An algorithm for differential file comparison*," Computing Science Technical Report 41, AT&T Bell Laboratories, Murray Hill NJ.

Hunt J.W., Szymanski T.G. (1977) "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, Vol. 20, No. 5, p. 350-3, May 1977.

Ivanov A.G. (1984) "Distinguishing an approximate word's inclusion on Turing machine in real time," *Izv. Akademii Nauk SSSR Ser. Mat.*, Vol. 48, p. 520-68 (Russian).

Jacobson G., Vo K-P. (1992) "Heaviest increasing/common subsequence problems," *Proceedings of the Combinatorial Matching Conference, Tucson, Arizona, April 1992.*

Karp R.M. (1972) "Reducibility among combinatorial problems," in Miller R.E., Thatcher J.W. (eds.) *Complexity of Computer Computations*, p. 85-103, Plenum Press.

Karp R.M., Rabin M.O. (1987) "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, Vol. 31, No. 2, p. 249-60, March 1987.

Knuth D.E. (1973) "*The art of computer programming, Volume 3: Sorting and Searching,*" Chapter 6.3, p. 490-3, Addison-Wesley, Reading MA.

Knuth D.E., Morris J.H., Pratt V.R. (1977) "Fast pattern matching in strings," *SIAM Journal on Computing*, Vol. 6, No. 2, p. 323-50, June 1977.

Landau G.M., Vishkin U., Nussinov R. (1985) *"An efficient string matching algorithm with k differences for nucleotide and amino acid sequences,"* Technical Report TR-37/85, Department of Computer Science, Tel Aviv University.

Landau G.M., Vishkin U. (1985) "Efficient string matching in the presence of errors," *Proceedings of the 26th IEEE Symposium on the Foundations of Computer Science*, p. 126-36.

Landau G.M., Vishkin U. (1986a) "Efficient string matching with k mismatches," *Theoretical Computer Science*, Vol. 43, p. 239-49.

Landau G.M., Vishkin U. (1986b) "Introducing efficient parallelism into approximate string matching and a new serial algorithm," *Proceedings of the 18th ACM Symposium on the Theory of Computing*, p. 220-30.

Landau G.M., Schieber B., Vishkin U. (1987) "Parallel construction of a suffix tree," Proceedings of the 14th ICALP, *Lecture Notes in Computer Science*, Vol. 267, p. 314-25, Springer-Verlag, New York/Berlin.

Landau G.M., Vishkin U. (1988) "Fast string matching with k differences," *Journal of Computer and System Sciences*, Vol. 37, No. 1, p. 63-78.

Landau G.M., Vishkin U. (1989) "Fast parallel and serial approximate string matching," *Journal of Algorithms*, Vol. 10, p. 157-69.

Lee D., Lochovsky F. (1985) "Text retrieval machine," *Office Automation — Concepts and Tools*, section 14, Springer-Verlag, New York.

Lee K.C., Mak V.W. (1989) "Design and analysis of a parallel VLSI string search algorithm," *Lecture Notes in Computer Science*, Vol. 368, p. 215-29.

Levenshtein V.I. (1965) "Binary codes capable of correcting deletions, insertions, and reversals," (Russian) *Doklady Akademii Nauk SSR*, Vol. 163, No. 4, p. 845-8, also *Cybernetics and Control Theory*, Vol. 10, No. 8, p. 707-10, 1966.

Lipman D.J., Pearson W.R. (1985) "Rapid and sensitive protein similarity searches," *Science*, Vol. 227, No. 4693, p. 1435-41, 22 March 1985.

Lowrance R., Wagner R.A. (1975) "An extension of the string-to-string correction problem," *Journal of the ACM*, Vol. 22, No. 2, p. 177-83.

Maier D. (1978) "The complexity of some problems on subsequences and supersequences," *Journal of the ACM*, Vol. 25, No. 2, p. 322-36, April 1978.

Majster M.E., Reiser A. (1980) "Efficient on-line construction and correction of position trees," *SIAM Journal on Computing*, Vol. 9, No. 4, p. 785-807, November 1980.

Masek W.J., Paterson M.S. (1980) "A faster algorithm for computing string-edit distances," *Journal of Computer and Systems Sciences*, Vol. 20, No. 1, p. 18-31.

Masek W.J., Paterson M.S. (1983) "How to compute string-edit distances quickly," in Sankoff D., Kruskall J.B. (eds.) *Time warps, string edits, and macromolecules: the theory and practice of sequence comparison*, Chapter 14, p. 337-49, Addison-Wesley, Reading MA.

McCreight E.M. (1976) "A space-economical suffix tree construction algorithm," *Journal of the ACM*, Vol. 23, No. 2, p. 262-72, April 1976.

Menico C. (1989) "Faster string searches," *Dr. Dobb's Journal*, p. 74-5, July 1989.

Morrison D.R. (1968) "PATRICIA — practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM*, Vol. 15, No. 4, p. 514-34.

Mukhopadhyay A. (1980) "Hardware algorithms for string processing," *Proceedings of ICCC*, p. 508-11.

Myers E.W. (1986) "An $O(ND)$ difference algorithm and its variations," *Algorithmica*, Vol. 1, p. 251-66.

Needleman S.B., Wunsch C.D. (1970) "A general method applicable to the search for similarities in the amino-acid sequence of two proteins," *Journal of Molecular Biology*, Vol. 48, p. 443-53.

Pinter R.Y. (1985) "Efficient string matching with don't care patterns," in Apostolico A., Galil Z. (eds.), *Combinatorial Algorithms on Words*, NATO ASI Series, Vol. F12, p. 11-29, Springer-Verlag, Berlin.

Pirklbauer K. (1992) "A study of pattern-matching algorithms," *Structured Programming*, Vol. 13, p. 89-98, Springer Verlag New York.

Regnier M. (1988) "*Knuth-Morris-Pratt algorithm: an analysis*," INRIA, Rocquencourt, France (unpublished).

Reichert T.A., Cohen D.N., Wong A.K.C. (1973) "An application of information theory to genetic mutations and the matching of polypeptide sequences," *Journal of Theoretical Biology*, Vol. 42, p. 245-61.

Rivest R.L. (1977) "On the worst-case behaviour of string searching algorithms," *SIAM Journal on Computing*, Vol. 6, No. 4, p. 669-74.

Robert D.C. (1982) "A specialized computer architecture for text retrieval," *Proceedings of the 4th Workshop on Computer Architecture*, p. 51-9.

Rodeh M., Pratt V.R., Even S. (1981) "Linear algorithm for data compression via string matching," *Journal of the ACM*, Vol. 28, No. 1, p. 16-24, January 1981.

Rytter W. (1980) "A correct preprocessing algorithm for Boyer-Moore string-searching," *SIAM Journal on Computing*, Vol. 9, p. 509-12.

Sakoe H., Chiba S. (1970) "A similarity evaluation of speech patterns by dynamic programming," (Japanese) *Institute of Electronic Communications Engineering of Japan*, p. 136, July 1970.

Sakoe H., Chiba S. (1971) "A dynamic programming approach to continuous speech recognition," *1971 Proceedings of the International Congress of Acoustics, Budapest, Hungary*, Paper 20 C 13.

Salton G. (1980) "Automatic information retrieval," *IEEE Computer*, Vol. 13, p. 41-55. September 1980.

Sankoff D. (1972) "Matching sequences under deletion-insertion constraints, " *Proceedings of the National Academy of Sciences of the USA*, Vol. 69, p. 4-6.

Sankoff D., Kruskall J.B. (eds.) (1983) "*Time warps, string edits, and macromolecules: the theory and practice of sequence comparison*," Addison-Wesley, Reading MA.

Schaback R. (1988) "On the expected sublinearity of the Boyer-Moore algorithm," *SIAM Journal on Computing*, Vol. 17, No. 4, p. 648-58.

Schensted C. (1961) "Largest increasing and decreasing subsequences," *Canadian Journal of Mathematics*, Vol. 13, p. 179-91.

Schieber B., Vishkin U. (1988) "On finding lowest common ancestors: simplification and parallelization," *SIAM Journal on Computing*, Vol. 17, No. 6, p. 1253-62.

Schönhage A., Strassen V. (1971) "Schnelle Multiplikation grosser Zahlen," *Computing (Arch. Elektron. Rechnen)*, Vol. 7, p. 281-92.

Sedgewick R. (1983) "String searching," Chapter 19 (p. 241-55) of *Algorithms*, Addison-Wesley, Reading MA.

Sellers P.H. (1980) "The theory and computation of evolutionary distances: pattern recognition," *Journal of Algorithms*, Vol. 1, p. 359-73.

Sibbald P.R., White M.J. (1987) "How probable are antibody cross-reactions?," *Journal of Theoretical Biology*, Vol. 127, p. 163-9.

Smit G. de V. (1982) "A comparison of three string matching algorithms," *Software — Practice and Experience*, Vol. 12, p. 57-66.

Smith P.D. (1991) "Experiments with a very fast substring search algorithm," *Software — Practice and Experience*, Vol. 21, No. 10, p. 1065-74, October 1991.

Smith T.F., Waterman M.S. (1981) "Identification of common molecular subsequences," *Journal of Molecular Biology*, Vol. 147, p. 195-7.

Sunday D.M. (1990) "A very fast substring search algorithm," *Communications of the ACM*, Vol. 33, No. 8, p. 132-42, August 1990.

Ukkonen E. (1983) "On approximate string matching," Proceedings of the International Conference on Foundations of Computer Science, *Lecture Notes in Computer Science*, Vol. 158, p. 487-95, Springer-Verlag, Berlin.

Ukkonen E. (1985a) "Algorithms for approximate string matching," *Information and Control*, Vol. 64, p. 100-18.

Ukkonen E. (1985b) "Finding approximate patterns in strings," *Journal of Algorithms*, Vol. 6, No. 6, p. 132-7.

Velichko V.M., Zagoruyko N.G. (1970) "Automatic recognition of 200 words, " *International Journal of Man-Machine Studies*, Vol. 2, p. 223-34.

Vintsyuk T.K. (1968) "Speech discrimination by dynamic programming," *Cybernetics*, Vol. 4, No. 1, p. 52-7, also (Russian) *Kibernetika*, Vol. 4, No. 1, p. 81-8.

Vo K-P. (1986) "*More* <curses>*: the* <screen> *library*, Technical Report, AT&T Bell Laboratories.

Wagner R.A., Fischer M.J. (1974) "The string-to-string correction problem," *Journal of the ACM*, Vol. 21, No. 1, p. 168-73, January 1974.

Wagner R.A. (1975) "On the complexity of the extended string-to-string correction problem," *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*, p. 218-23, also in Sankoff D., Kruskall J.B. (eds.) *Time warps, string edits, and macromolecules: the theory and practice of sequence comparison*, Addison-Wesley, Reading MA, 1983.

Weiner P. (1973) "Linear pattern matching algorithm," *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, p. 1-11.

Wong C.K., Chandra A.K. (1976) "Bounds for the string editing problem," *Journal of the ACM*, Vol. 23, No. 1, p. 13-6, January 1976.

Woude J. van der (1989) "Playing with patterns, searching for strings," *Science of Computer Programming*, Vol. 12, No. 3, p. 177-90, Elsevier Science Publishers.

Yamada H., Hirata M., Nagai H., Takahashi K. (1987) "A high-speed string-search engine," *IEEE Journal of Solid-State Circuits*, Vol. SC-22, No. 5, p. 829-34, October 1987.

Yianilos P.N. (1983) "A dedicated comparator matches symbol strings fast and intelligently," *Electronics*, Vol. 56, No. 5, p. 113-7, December 1983.

Ziv J., Lempel A. (1977) "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, Vol. IT-23, p. 337-43, May 1977.