

Contents

1	Introduction	1
2	String Searching with Don't-care Symbols	3
2.1	Don't-cares in pattern only	3
2.2	Don't-cares in pattern and text	5
3	String Editing and Longest Common Subsequences	7
3.1	Longest Common Subsequences	9
3.2	Hirschberg's paradigm: finding antichains one at a time	13
3.3	Incremental antichain decompositions and the Hunt-Szymanski paradigm . . .	13
4	String Searching with Errors	16
5	Two Dimensional Matching	19
5.1	Searching with automata	20
5.2	Periods and witnesses in two dimensions	21
6	Tree matching	23
6.1	Exact tree searching	23
6.2	Tree editing	24
7	Research Issues and Summary	28
8	Defining Terms	29
9	Acknowledgements	32
10	Further Information	37

GENERAL PATTERN MATCHING

Alberto Apostolico, Purdue University and Università di Padova

1 Introduction

This chapter reviews combinatorial and algorithmic problems related to searching and matching of strings and slightly more complicated structures such as arrays and trees. These problems arise in a vast variety of applications and in connection with various facets of storage, transmission and retrieval of information. A list would include the design of structures for the efficient management of large repositories of strings, arrays and special types of graphs, fundamental primitives such as the various variants of exact and approximate searching, specific applications such as the identification of periodicities and other regularities, efficient implementations of ancillary functions such as compression and encoding of elementary discrete objects, etc. The main objective of studies in pattern matching is to abstract and identify some primitive manipulations, develop new techniques and efficient algorithms to perform them, both by serial and parallel or distributed computation, and implement the new algorithms.

Some initial pattern matching problems and techniques arose in the early Seventies in connection with emerging technologies and problems of the time, e.g., compiler design. Since then, the range of applications of the tools and methods developed in pattern matching has expanded to include text, image and signal processing, speech analysis and recognition, data compression, computational biology, computational chemistry, computer vision, information retrieval, symbolic computation, computational learning, computer security, graph theory and VLSI, etc. In little more than two decades, an initially sparse set of more or less unrelated results has grown into a considerable body of knowledge. A complete bibliography of string algorithms would contain more than 500 articles. A.V. Aho, [1990] references over 140 papers in his recent survey of string-searching algorithms alone; advanced workshops and schools, books and special issues in major journals have already been dedicated to the subject and more are planned for the future. The interested reader will find a few reference books and conference proceedings in the bibliography of this chapter.

While each application domain presents peculiarities of its own, a number of pattern matching primitives are shared, in nearly identical forms, within wide spectra of distant and diverse areas. For instance, searching for identical or similar substrings in strings is of paramount interest to software development and maintenance, philology or plagiarism detection in the humanities, inference of common ancestries in molecular genetics, comparison of geological evolutions, stereo matching for robot vision, etc. Checking the equivalence (i.e., identity up to a rotation) of circular strings finds use in determining the homology of organisms with circular genomes, comparing closed curves in computer vision, establishing the equivalence of polygons in computer graphics, etc. Finding repeated patterns, symmetries and cadences in strings is of interest to data compression, detection of recurrent events in symbolic dynamics, genome studies, intrusion detection in distributed computer systems, etc. Similar considerations hold for higher structures. In general, an intermediate objective of studies in pattern matching is to understand and characterize combinatorial structures and properties that are susceptible of exploitation in computational matching and searching on discrete elementary structures.

Most pattern matching issues are still subject to extensive investigation both within serial and parallel models of computation. This survey concentrates on sequential algorithms, but the reader is encouraged to explore for himself the rich repertoire of parallel algorithms developed in recent years. Most of these algorithms bear very little resemblance to their serial counterparts. Similar considerations apply to some algorithms formulated in a probabilistic setting.

Pattern matching problems may be classified according to a number of paradigms. One way is based on the type of structure (strings, arrays, trees, etc.) in terms of which they are posed. Another, is according to the model of computation used, e.g., RAM, PRAM, Turing Machine. Yet another one is according to whether the manipulations that one seeks to optimize need be performed on-line, off-line, in real time, etc. One could distinguish further between matching and searching and, within the latter, between exact and approximate searches, or vice versa. The classification used here is thus somewhat arbitrary. We assume some familiarity of the reader with exact string searching, both on- and off-line, which is covered in a separate chapter of this volume. We start by reviewing some basic variants of string searching

where occurrences of the pattern need not be exact. Next, we review algorithms for string comparisons. Then, we consider pattern matching on two dimensional arrays and finally on rooted trees.

2 String Searching with Don't-care Symbols

As already mentioned, we assume familiarity of the reader with the problem of **exact string searching**, in which we are interested in finding all the occurrences of a pattern string y into a textstring x . One of the natural departures from this formulation consists of assuming that a symbol can (perhaps only at some definite positions) match a small group of other symbols. At one extreme we may have, in addition to the symbols in the input alphabet Σ , a **don't care** symbol ϕ with the property that ϕ matches any other character in Σ . This gives rise to variants of string searching where, in principle, ϕ appears (i) only in the pattern, (ii) only in the text or (iii) both in pattern and text. There seems to be no peculiar result on variant (ii), whence we shall consider this as just a special case of (iii). The situation is different with variant (i), which warrants the separate treatment which is given next.

2.1 Don't-cares in pattern only

Fischer and Paterson [1973] and Pinter [1984] discuss the problem faced if one tried to extend the KMP string searching algorithm [Knuth et al., 1977] in order to accommodate don't cares in the pattern: the obvious transitivity on character equality, that subtends those and other exact string searching, is lost with don't cares. Pinter noted that a partial recovery is possible if the number and positions of don't cares is fixed. In fact, in this case one may resort to ideas used by Aho and Corasick [1975] in connection with exact multiple string searching and solve the problem within the same time complexity $O(n + m + r) \log |\Sigma|$ time, where $n = |x|$ is the length of the textstring, $m = |y|$ is the length of the pattern and r is the total number of occurrences of the fragments of the pattern that would be obtained by cleaving the latter at don't cares.

We outline Pinter's approach. Since the don't cares appear in fixed known positions, we

may consider the pattern decomposed into **segments** of Σ^+ , say, $\hat{y}_1, \hat{y}_2, \dots, \hat{y}_p$ and ϕ -**blocks** consisting of runs of occurrences of ϕ , respectively. Each \hat{y}_i can be treated as an individual pattern in a multiple pattern matching machine. Through the search, one computes for each \hat{y}_i a list of its occurrence in x . Let d_i be the known distance between the starting positions of \hat{y}_i and \hat{y}_{i+1} . We may now merge the occurrence list while keeping track of these distances, using the natural observation that if a match occurred starting at position j , then the \hat{y}_i 's will appear in the same order and inter-segment distance as they appear in the pattern. Here the merge process takes place after the search. To make his algorithm work in real time applications, Pinter used an array of counters, a data structure originally proposed by R.L. Rivest. Instead of merging lists, counters count from 0 to p while collecting evidence of a pattern occurrence. Specifically, the counting mechanism is as follows. Let the **offset** of a segment be the distance from the beginning of the pattern to the end of that segment. Whenever a segment match is detected ending at position j , then its offset f_j is subtracted from j , thus yielding the starting position $j - f_i$ of a corresponding candidate occurrence of the pattern. Next, the counter assigned to position $1 + (j - f_i) \bmod m$ is incremented by 1. Therefore, a counter initialized to zero reaches p iff the last m characters examined on the text matched the pattern. A check whether a counter has reached p is performed each time that counter is re-used.

Manber and Baeza-Yates [1991] consider the case where the pattern embeds a string of at most k don't cares, i.e., has the form $y = u\phi^i v$, where $i \leq k$, $u, v \in \Sigma^*$ and $|u| \leq m$ for some given k, m . Their algorithm is off-line in the sense that the text x is preprocessed to build the suffix array [Manber and Myers, 1990] associated with it. This operation costs $O(n \log |\Sigma|)$ time in the worst case. Once this is done, the problem reduces to one of efficient implementation of 2-dimensional orthogonal range queries (for these latter see. e.g., [Chazelle, 1988], [Willard, 1986]).

One more variant of string searching with don't care in pattern only is discussed in [Takeda, 1993]. Also Takeda's algorithm is based on the algorithm in [Aho et al. , 1975]. The problem is stated as follows. Consider a set $A = \{A_1, A_2, \dots, A_p\}$, where each $A_i \subseteq \Sigma$ is a nonempty set called a **picture** and pictures are mutually disjoint. While a don't care symbol matches all symbols, a picture matches a subset of the alphabet. For any pattern y , we have

now that $y \in (\Sigma \cup A)^+$. Then, given a set of patterns $Y = \{y^{(1)}, \dots, y^{(k)}\}$, the problem is to find all occurrences of $y^{(i)}$ for $i = 1, \dots, k$. Thus, when $A = \Sigma$, the problem reduces to plain string searching with don't cares. A pattern matching machine for such a family can be quite time consuming to build. Takeda managed to improve on time efficiency by saving on the number of explicit "goto" edges created in that machine.

Takeda's variant finds natural predecessors in an even more general class considered by K. Abrahamson [1987]. This latter paradigm applies to an unbounded alphabet Σ , as long as individual symbols have finite encodings. Let $P = \{P_1, P_2, \dots, P_k\}$ be a set of **pattern elements**, where each pattern element is a subset of Σ . There are **positive** and **negative** pattern elements. A positive element, is denoted by $\langle \sigma_1, \dots, \sigma_f \rangle$ and has the property of matching each one of the characters $\sigma_1, \sigma_2, \dots, \sigma_f$. A negative element is denoted by $[\sigma_1, \dots, \sigma_f]$ and will match any character of Σ except characters $\sigma_1, \sigma_2, \dots, \sigma_f$. A pattern $y \in P^+$ identifies now a family of strings, namely, all strings in the form $y_1 y_2 \dots y_m$ such that $y_i \in \Sigma$ is compatible with the element of P used to identify the i -th element of y . Using a time-space tradeoff proof technique due to Borodin, Abrahamson proved that the time-space lower bound on a subproblem with $n = 2m$ is $\Omega(m^2 / \log m)$.

By combining **divide and conquer** with an idea of Fischer and Paterson [1974] which will be discussed more thoroughly later, Abrahamson designed an algorithm taking time $O(N + M + n\hat{M}^{1/2} \log m \log^{1/2} m)$, where N and M denote the lengths of the encodings (e.g., in bits) of x and y respectively, and \hat{M} represents the number of distinct elements of Σ which are present in the pattern.

2.2 Don't-cares in pattern and text

In an elegant, landmark paper, Fischer and Paterson [1974] exposed the similarity of string searching to multiplication, thereby obtaining a number of interesting algorithms for exact string searching and some of its variants. It is not difficult to see that string matching problems can be rendered as special cases of a general linear product. Given two vectors X and Y , their **linear product** with respect to two suitable operations \otimes and \oplus , is denoted by $X \otimes_{\oplus} Y$, and is a vector $Z = Z_0 Z_1 \dots Z_{m+n}$ where $Z_k = \bigoplus_{i+j=k} X_i \otimes Y_j$ for $k = 0, \dots, m+n$. If we interpret

\oplus as the boolean \wedge and \otimes as the symbol equivalence \equiv , then a match of the reverse Y^R of Y , occurs ending at position k in X , where $m \leq k \leq n$, if $[X_{k-m} \dots X_k] \equiv [Y_m \dots Y_0]$, that is, with obvious meaning, if $(X \stackrel{\equiv}{\wedge} Y)_k = \text{TRUE}$. This observation brings string searching into the family of boolean, polynomial and integer multiplications leads quickly to an $O(n \log m \log \log m)$ time solution even in the presence of don't cares, provided that the size of Σ is fixed.

To see this, we show first that string searching can be regarded as a boolean linear product, i.e., one where \oplus is \vee and \otimes is \wedge . Let the textstring be specified as $x = x_0 x_1 x_2 \dots x_n$ and similarly let $y = y_0 y_1 y_2 \dots y_m$ be the pattern. Recall that we assume a finite alphabet Σ , and that both x and y may contain some don't cares. For each $\rho \in \Sigma$, define $H_\rho(X_i) = 1$ if $x_i = \rho$, and $H_\rho(X_i) = 0$ if $x_i \neq \rho$ or $x_i = \phi$. Assume now that the vector X corresponding to string x contains only symbol σ and ϕ while Y , corresponding to string y , contains only symbol $\tau \neq \sigma$ and ϕ , with both σ and $\tau \in \Sigma$. Then $\bigwedge_{i+j=k} X_i \equiv Y_j$ means that $\bigwedge_{i+j=k} \neg H_\sigma(X_i) \vee \neg H_\tau(Y_j) \iff \neg \bigvee_{i+j=k} H_\sigma(X_i) \wedge H_\tau(Y_j)$. The last term is a boolean product, whence such a product is not harder than string searching. On the other hand, $Z = X \stackrel{\equiv}{\wedge} Y = \neg \bigvee_{\sigma \neq \tau; \sigma, \tau \in \Sigma} H_\sigma(X) \bigwedge_{\vee} H_\tau(Y)$.

As is well known, the boolean product can be obtained by performing the polynomial product, in which \oplus is $+$ and \otimes is \times . For this, just encode TRUE and FALSE as 1 and 0, respectively. One way to compute the polynomial product is to embed the product in a single large integer multiplication. There are well known fast solutions for the latter problem. For the $\{0, 1\}$ string vectors X and Y , the maximum coefficient is $m + 1$, so if we choose r such that $2^r > m + 1$, compute the integers $X(2^r) = \sum_{i=0}^n X_i \cdot 2^{ri}$ and $Y(2^r) = \sum_{j=0}^m Y_j \cdot 2^{rj}$ and then multiply $X(2^r)$ and $Y(2^r)$, the result will be the product evaluated at 2^r . The consecutive blocks of length r in the binary representation of $Z(2^r)$ will give the coefficients of Z , which can be transferred back to the boolean product, and from there back to the string matching product. The Schönhage-Strassen [1971] algorithm multiplies an N -digit number by an M -digit number in time $O(N \cdot \log M \cdot \log \log M)$, for $N > M$, using a multi-tape Turing machine. For the string matching product, $N = nr = O(n \log m)$, $M = mr = O(m \log m)$, so that the problem is solved on that machine in time $O(n \log^2 m \log \log m)$. The algorithm as presented assumes that the alphabet finite. For any alphabet Σ of size polynomial in n , however, we can always encode

the two input strings in binary at a cost of a multiplicative factor $O(\log |\Sigma|)$, and then execute just two boolean products. This results in an extra $O(\log m)$ factor in the time complexity.

Note the adaptation of fast multiplication to string searching provides a basis for counting the mismatches generated by a pattern y at every position of a text x . This results from treating all symbols of Σ separately, and thus in overall time $O(n(|\Sigma|)\log^2 m \log \log m)$. This latter complexity is comparable to the above only for finite Σ . However, we shall see later that better bounds are achievable under this approach.

3 String Editing and Longest Common Subsequences

We now introduce three **edit operations** on strings. Namely, given any string w we consider the **deletion** of a symbol from w , the **insertion** of a new symbol in w and the **substitution** of one of the symbols of w with another symbol from Σ . We assume that each edit operation has an associated nonnegative real number representing the **cost** of that operation. More precisely, the cost of deleting from w an occurrence of symbol a is denoted by $D(a)$, the cost of inserting some symbol a between any two consecutive positions of w is denoted by $I(a)$ and the cost of substituting some occurrence of a in w with an occurrence of b is denoted by $S(a, b)$. An **edit script** on w is any sequence Γ of viable edit operations on w , and the cost of Γ is the sum of all costs of the edit operations in Γ .

Now, let x and y be two strings of respective lengths $|x| = n$ and $|y| = m \leq n$. The **string editing problem** for input strings x and y consists of finding an edit script Γ' of minimum cost that transforms y into x . The cost of Γ' is the **edit distance from y to x** . Edit distances where individual operations are assigned integer or unit costs occupy a special place. Such distances are often called Levenshtein distances, since they were introduced by Levenshtein [1966] in connection with error correcting codes. String editing finds applications in a broad variety of contexts, ranging from speech processing to geology, from text processing to molecular biology.

It is not difficult to see that the general (i.e., with unbounded alphabet and unrestricted costs) problem of edit distance computation is solved by a serial algorithm in $\Theta(mn)$ time and

space, through dynamic programming. Due to widespread application of the problem, however, such a solution and a few basic variants were discovered and published in literature catering to diverse disciplines (see, e.g., [Needleman and Wunsch, 1973], [Sankoff, 1972], [Sellers, 1974], [Wagner and Fischer, 1974]). In Computer Science, the problem was dubbed “the string-to-string correction problem”. The CS literature was possibly the last one to address the problem, but the interest in the CS community increased steadily in subsequent years. By the early 80’s, the problem had proved so pervasive, especially in biology, that a book by Sankoff and Kruskal [1983] was devoted almost entirely to it. Special issues of the Bulletin of Mathematical Biology and various other books and journals routinely devote significant portions to it.

An $\Omega(mn)$ lower bound was established for the problem by Wong and Chandra [1976] for the case where the queries on symbols of the string are restricted to tests of equality. For unrestricted tests, a lower bound $\Omega(n \log n)$ was given by Hirschberg [1978]. Algorithms slightly faster than $\Theta(mn)$ were devised by Masek and Paterson [1980], through resort to the so-called “Four Russians Trick”. The “Four Russians” are Arlazarov, Dinic, Kronrod, and Faradzev [1970]. Along these lines, the total execution time becomes $\Theta(n^2/\log n)$ for bounded alphabets and $O(n^2(\log \log n)/\log n)$ for unbounded alphabets. The method applies only to the classical Levenshtein distance metric, and does not extend to general cost matrices. To this date, the problem of finding either tighter lower bounds or faster algorithms is still open.

The criterion that subtends the computation of edit distances by dynamic programming is readily stated. For this, let $C(i, j)$, ($0 \leq i \leq |y|$, $0 \leq j \leq |x|$) be the minimum cost of transforming the prefix of y of length i into the prefix of x of length j . Let w_k denote the k th symbol of string w . Then $C(0, 0) = 0$, $C(i, 0) = C(i - 1, 0) + D(y_i)$ ($i = 1, 2, \dots, |y|$), $C(0, j) = C(0, j - 1) + I(x_j)$ ($j = 1, 2, \dots, |x|$), and

$$C(i, j) = \min\{C(i - 1, j - 1) + S(y_i, x_j), C(i - 1, j) + D(y_i), C(i, j - 1) + I(x_j)\}$$

for all i, j , ($1 \leq i \leq |y|$, $1 \leq j \leq |x|$). Observe that, of all entries of the C -matrix, only the three entries $C(i - 1, j - 1)$, $C(i - 1, j)$, and $C(i, j - 1)$ are involved in the computation of the final value of $C(i, j)$. Hence $C(i, j)$ can be evaluated row-by-row or column-by-column in $\Theta(|y||x|) = \Theta(mn)$ time. An optimal edit script can be retrieved at the end by backtracking

through the local decisions that were made by the algorithm.

A few important problems are special cases of string editing, including the **longest common subsequence** problem, **local alignment**, i.e., the detection of local similarities of the kind sought typically in the analysis of molecular sequences such as DNA and proteins, and some important variants of **string searching with errors**, or searching for approximate occurrences of a pattern string in a text string. As highlighted in the following brief discussion, a solution to the general string editing problem implies typically similar bounds for all these special cases.

3.1 Longest Common Subsequences

Perhaps the single most widely studied special case of string editing is the so-called longest common subsequence (LCS) problem. The problem is defined as follows. Given a string z over an alphabet $\Sigma = (i_1, i_2, \dots, i_s)$, a **subsequence** of z is any string w that can be obtained from z by deleting zero or more (not necessarily consecutive) symbols. The **longest common subsequence problem** for input strings $x = x_1x_2\dots x_n$ and $y = y_1y_2\dots y_m$ ($m \leq n$) consists of finding a third string $w = w_1w_2\dots w_l$ such that w is a subsequence of x and also a subsequence of y , and w is of maximum possible length. In general, string w is not unique.

Like the string editing problem itself, the LCS problem arises in a number of applications spanning from text editing to molecular sequence comparisons, and has been studied extensively over the past. Its relation to string editing can be understood as follows.

Observe that the effect of a given substitution can be always achieved, alternatively, through an appropriate sequence consisting of one deletion and one insertion. When the cost of a non-vacuous substitution (i.e., a substitution of a symbol with a different one) is higher than the global cost of one deletion followed by one insertion, then an optimum edit script will always avoid substitutions and produce instead y from x solely by insertions and deletions of overall minimum cost. Specifically, assume that insertions and deletions have unit costs, and that a cost higher than 2 is assigned to substitutions. Then, the pairs of matching symbols preserved in an optimal edit script constitute a longest common subsequence of x and y . It is

not difficult to see that the cost e of such an optimal edit script, the length l of an LCS and the lengths of the input strings obey the simple relationship: $e = n + m - 2l$. Similar considerations can be developed for the variant where matching pairs are assigned weights and a **heaviest** common subsequence is sought (see, e.g., Jacobson and Vo [1992]).

Lower bounds for the LCS problem are time $\Omega(n \log n)$ or linear time, according to whether the size s of Σ is unbounded or bounded [Hirschberg, 1978]. Aho, Hirschberg and Ullman [1976] showed that, for unbounded alphabets, any algorithm using only “equal-unequal” comparisons must take $\Omega(mn)$ time in the worst case. The asymptotically fastest general solution rests on the corresponding solution by Masek and Paterson [1980] to the string editing, hence takes time $O(n^2 \log \log n / \log n)$. Time $\Theta(mn)$ is achieved by the following dynamic programming algorithm from Hirschberg [1977].

Let $L[0\dots m, 0\dots n]$ be an integer matrix initially filled with zeroes.

The following code transforms L in such a way that $L[i, j]$ ($1 \leq i \leq m, 1 \leq j \leq n$) contains the length of an LCS between $y_1 y_2 \dots y_i$ and $x_1 x_2 \dots x_j$.

for $i = 1$ **to** m **do**

for $j = 1$ **to** n **do** **if** $y_i = x_j$ **then** $L[i, j] = L[i - 1, j - 1] + 1$

else $L[i, j] = \text{Max} \{L[i, j - 1], L[i - 1, j]\}$

The correctness of this strategy follows from the obvious relations:

$$\begin{aligned} L[i - 1, j] &\leq L[i, j] \leq L[i - 1, j] + 1; \\ L[i, j - 1] &\leq L[i, j] \leq L[i, j - 1] + 1; \\ L[i - 1, j - 1] &\leq L[i, j] \leq L[i - 1, j - 1] + 1. \end{aligned}$$

If only the length l of an LCS is desired, then this code can be adapted to use only linear space. If an LCS is required at the outset, then it is necessary to keep track of the decision made at every step by the algorithm, so that an LCS w can be retrieved at the end by backtracking. The early $\Theta(mn)$ time algorithm by Hirschberg [1978] achieved both a linear space bound and the production of an LCS at the outset, through a combination of dynamic programming and

divide-and-conquer. Subsequent approaches to the LCS problem achieve time complexities better than $\Theta(mn)$ in favorable cases, though a quadratic performance is always touched and sometimes even exceeded in the worst cases. These approaches exploit in various ways the **sparsity** inherent to the LCS problem. Sparsity allows us to relate algorithmic performances to parameters other than the lengths of the input. Some such parameters are introduced next.

The ordered pair of positions i and j of L , denoted $[i, j]$, is a **match** iff $y_i = x_j$, and we use r to denote the total number of matches between x and y . If $[i, j]$ is a match, and an LCS $w_{i,j}$ of $y_1y_2\dots y_i$ and $x_1x_2\dots x_j$ has length k , then k is the **rank** of $[i, j]$. The match $[i, j]$ is **k-dominant** if it has rank k and for any other pair $[i', j']$ of rank k either $i' > i$ and $j' \leq j$ or $i' \leq i$ and $j' > j$. A little reflection establishes that computing the k -dominant matches ($k = 1, 2, \dots, l$) is all that is needed to solve the LCS problem (see, e.g., [Apostolico and Guerra, 1987], [Hirschberg, 1977]). Clearly, the LCS of x and y has length l iff the maximum rank attained by a dominant match is l . It is also useful to define, on the set of matches in L , the following partial order relation \mathcal{R} : match $[i, j]$ precedes match $[i', j']$ in \mathcal{R} if $i < i'$ and $j < j'$. A set of matches such that in any pair one of the matches always precedes the other in \mathcal{R} constitutes a **chain** relative to the partial order relation \mathcal{R} . A set of matches such that in any pair neither match precedes the other in \mathcal{R} is an **antichain**. Then, the LCS problem translates into the problem of finding a longest chain in the **poset** (partially ordered set) of matches induced by \mathcal{R} (cf. [Sankoff and Sellers, 1973]). A decomposition of a poset into antichains is **minimal** if it partitions the poset into the minimum possible number of antichains (refer, e.g., to [Bogart, 1983]).

Theorem 1 (Dilworth [1950]) *A maximal chain in a poset P meets all antichains in a minimal antichain decomposition of P .*

In other words, the number of antichains in a minimal decomposition represents also the length of a longest chain. Even though it is never explicitly stated, most known approaches to the LCS problem in fact compute a minimal antichain decomposition for the poset of matches induced by \mathcal{R} . The k th antichain in this decomposition is represented by the set of all matches having rank k . For general posets, a minimal antichain decomposition is computed by flow

techniques [Bogart, 1983], although not in time linear in the number of elements of the poset. Most LCS algorithms that exploit sparsity have their natural predecessors in either [Hunt and Szymanski, 1977] or [Hirschberg, 1977]. In terms of antichain decompositions, the approach of Hirschberg [1977] consists of computing the antichains in succession, while that of Hunt and Szymanski [1977] consists of extending partial antichains relative to all ranks already discovered, one new symbol of y at a time. The respective time complexities are $O(nl + n \log s)$ and $O(r \log n)$. Thus, the algorithm of Hunt and Szymanski is favorable in very sparse cases, but worse than quadratic when r tends to mn . An important specialization of this algorithm is that to the problem of finding a longest ascending subsequence in a permutation of the integers from 1 to n . Here, the total number of matches is n , which results in a total complexity $O(n \log n)$. Resort to the “fat-tree” structures introduced by Van Emde Boas [1975] leads to $O(n \log \log n)$ for this problem, a bound which had been shown to be optimal by Fredman [1975].

		a	b	c	a	b	c	c	b	c
		1	2	3	4	5	6	7	8	9
c	1	0	0	1	1	1	1	1	1	1
b	2	0	1	1	1	2	2	2	2	2
a	3	1	1	1	2	2	2	2	2	3
d	4	1	1	1	2	2	2	2	2	3
b	5	1	2	2	2	3	3	3	3	3
b	6	1	2	2	2	3	3	3	4	4

Figure 1: Illustrating antichain decompositions

Fig. 1 illustrates the concepts introduced thus far, displaying the final L-matrix for the strings $x = cbadbb$ and $y = abcabccbc$. We use circles to represent matches, with bold circles denoting dominant matches. Dotted lines thread antichains relative to \mathcal{R} and also separate regions.

3.2 Hirschberg’s paradigm: finding antichains one at a time

We outline a $\Theta(mn)$ time LCS algorithm in which antichains of matches relative to the various ranks are discovered one after the other. Consider the *dummy* pair $[0, 0]$ as a “0-dominant match”, and assume that all $(k - 1)$ -dominant matches for some k , $0 \leq k \leq l - 1$, have been discovered at the expense of scanning the part of the L -matrix that would lie above or to the left of the antichain $(k - 1)$, inclusive. To find the k -th antichain, scan the unexplored area of the L -matrix from right to left and top-down, until a stream of matches is found occurring in some row i . The leftmost such match is the k -dominant match $[i, j]$ with smallest i -value. The scan continues at next row and to the left of this match, and the process is repeated at successive rows until all of the k -th antichain has been identified. The process may be illustrated like in Figure 2. The large circles denote “pebbles” used to intercept the matches in an antichain. Initially, the pebbles are positioned on the matches created in the last column by the ad-hoc wildcard symbol $\$$. Next, pebbles are considered in succession from the top, and each pebble is moved to the leftmost match it can reach without crossing a previously discovered antichain. Once all pebbles have been considered, those contributing to the new antichain are identified easily.

Note that for each k the list of $(k - 1)$ -dominant matches is enough to describe the shape of the antichain and also to guide the searches involved at the subsequent stage. Thus, also in this case linear space is sufficient if one wishes to compute only the length of w .

An efficient implementation of this scheme leads to the algorithm by Hirschberg [1977], which takes time $O(nl + n \log s)$ and space $O(d + n)$, where d is the number of dominant matches.

3.3 Incremental antichain decompositions and the Hunt-Szymanski paradigm

When the number r of matches is small compared to m^2 (or to the expected value of lm), an algorithm with running time bounded in terms of r may be advantageous. Along these lines,

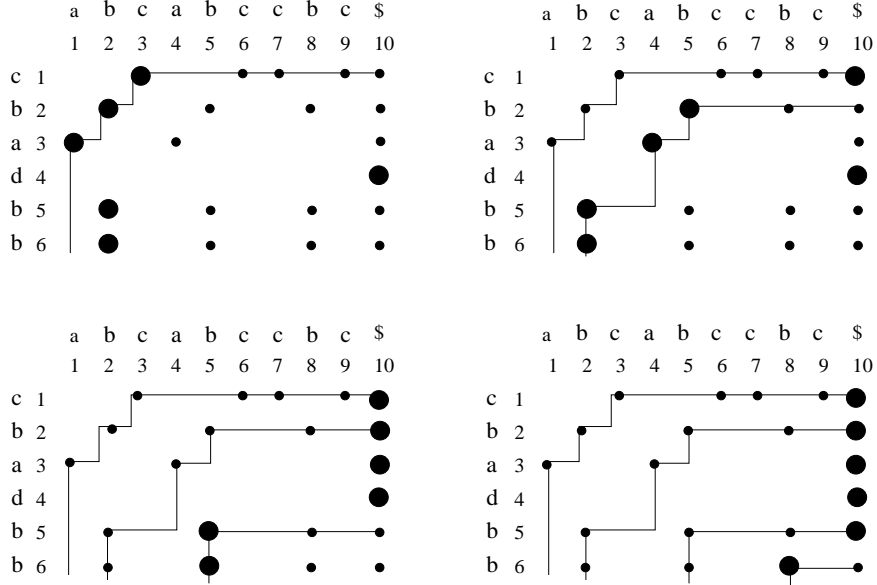


Figure 2: Hirschberg’s paradigm: discovering one antichain at a time. The positions occupied by the pebbles at the end of consecutive antichain constructions are displayed clockwise from left-top

Hunt and Szymanski [1977] set up an algorithm (HS) with a time bound of $O((n + r) \log n)$. This algorithm works by computing, row after row, the ranks of all matches in each row. The treatment of a new row corresponds thus to extending the antichain decomposition relative to all preceding rows. A same match is never considered more than once. On the other hand, the time required by HS degenerates as r gets close to mn . In these cases this algorithm is outperformed by the algorithm of Hirschberg [1977], which exhibits a bound of $O(\ln)$ in all situations.

Algorithm HS is reproduced below. Essentially, it scans the list of matching positions $MATCHLIST$ associated with the i -th row of L and considers those matches in succession, from right to left. For each match, HS decides whether it is a k -dominant match for some k through a binary search in the array $THRESH$ which maintains the leftmost previously discovered k -dominant match for each k . If the current match forces an update for rank k , then the contents of $THRESH[k]$ is modified accordingly. Observe that considering the matches in reverse order is crucial to the correct operation of HS . The details are found in the code below.

Algorithm “HS”: element array $y[1 : m], x[1 : n]$;
integer array $THRESH[0 : m]$; list array $MATCHLIST[1 : m]$;
pointer array $LINK[1 : m]$; pointer PTR ;
begin (*PHASE 1* : initializations)
 for $i = 1$ **to** m **do**
 begin
 $MATCHLIST[i] = \{j_1, j_2, \dots, j_p\}$
 such that $j_1 > j_2 > \dots > j_p$
 and $y_i = x_{j_q}$ *for* $1 \leq q \leq p$
 $THRESH[i] = n + 1$ *for* $1 \leq i \leq m$;
 end
 $THRESH[0] = 0$; $LINK[0] = null$;
 (*PHASE 2* : find k -dominant matches)
 for $i = 1$ **to** m **do**
 for j *on* $MATCHLIST[i]$ **do**
 begin *find* k *such that*
 $THRESH[k - 1] < j \leq THRESH[k]$;
 if $j < THRESH[k]$ **then**
 begin
 $THRESH[k] = j$;
 $LINK[k] = newnode(i, j, LINK[k - 1])$
 end
 end
 end
 (*PHASE 3* : recover LCS w in reverse order)
 $\hat{k} =$ *largest* k *such that* $THRESH[k] \neq n + 1$;
 $PTR = LINK[\hat{k}]$;
 while $PTR \neq null$ **do begin**
 print the match $[i, j]$ *pointed to by* PTR ;
 advance PTR **end**
end.

The total time spent by HS is bounded by $O((r + m)\log n + n \log s)$, where the $n \log s$ term is charged by the preprocessing needed to create the lists of matches. The space is bounded by $O(d + n)$. As mentioned, this is good in sparse cases but becomes worse than quadratic for dense r .

4 String Searching with Errors

In this section, we assume unit cost for all edit operations. Given a pattern y and a text x , the most general variant of the problem consists of computing, for every position of the text, the best edit distance achievable between y and any substring w of x ending at that position. It is not difficult to express a solution in terms of a suitable adaptation of the recurrence previously introduced in connection with string editing. The first obvious change consists of setting all costs to 1 except that $S(y_i, x_j) = 0$ for $y_i = x_j$. Thus, we have now, for all i, j , ($1 \leq i \leq |y|, 1 \leq j \leq |x|$),

$$C(i, j) = \min\{C(i - 1, j - 1) + 1, C(i - 1, j) + 1, C(i, j - 1) + 1\}.$$

A second change consists of setting the initial conditions so that $C(0, 0) = 0$, $C(i, 0) = i$ ($i = 1, 2, \dots, m$), $C(0, j) = 0$ ($j = 1, 2, \dots, n$). This has the effect of setting to zero the cost of prefixing y by any prefix of x . In other words, any prefix of the text can be skipped free of charge in an optimum edit script.

Clearly, the computation of the final value of $C(i, j)$ may proceed as in the general case, and it will still take $\Theta(|y||x|) = \Theta(mn)$ time. Note, however, that we are interested now in the entire last row of matrix C at the outset. Although we assumed unit costs, the validity of the method extends clearly to the case of general positive costs.

In practice, it is often more interesting to locate only those segments of x that present a high similarity with y under the adopted measure. Formally, given a pattern y , a text x and an integer k , this restricted version of the problem consists of locating all terminal positions of substrings w of x such that the edit distance between w and y is at most k . The recurrence

given above will clearly produce this information. However, there are more efficient methods to deal with this restricted case. In fact, a time complexity $O(kn)$ and even sublinear expected time are achievable. We refer to Landau and Vishkin [1986, 1988], Sellers [1974], Ukkonen [1985], Galil and Giancarlo [1988], Chang and Lawler [1990], for detailed discussions. In the following, we review some basic principles subtending an $O(kn)$ algorithm for string searching with k differences. Note that when k is a constant the corresponding time complexity is linear.

The crux of the method is to limit computation to $O(k)$ elements in each diagonal of the matrix C . These entries will be called **extremal** and may be defined as follows: a diagonal entry is d -extremal if it is the deepest entry on that diagonal to be given value d ($d = 1, 2, \dots, k$). Note that a diagonal might not feature any, say, 1-extremal entry, in which case it would correspond to a perfect match of the pattern. The identification of d -extremal entries proceeds from extension of entries already known to be $(d - 1)$ -extremal. Specifically, assume we knew that entry $C(i, j)$ is $(d - 1)$ -extremal. Then, any entry reachable from $C(i, j)$ through a unit vertical, horizontal or diagonal-mismatch step possibly followed by a maximal diagonal stream of matches is d -extremal at worst. In fact, the cost of a diagonal stream of matches is 0, whence the cost of an entry of the type considered cannot exceed d . On the other hand, that cost cannot be smaller than $d - 1$, otherwise this would contradict the assumption $C(i, j) = d - 1$. Let entries reachable from a $(d - 1)$ -extremal entry $C(i, j)$ through a unit vertical, horizontal or diagonal-mismatch step be called d -**adjacent**. Then the following program encapsulates the basic computations.

Algorithm “KERR” :

element array $x[1 : n], y[1 : m], C[0 : m; 0 : n]$; integer k

begin

(*PHASE 1* : initializations)

set first row of C to 0;

find the boundary set S_0 of 0-extremal entries by exact string searching;

(*PHASE 2* : identify k -extremal entries)

for $d = 1$ to k do

begin

```

    walk one step horizontally, vertically and (on mismatch) diagonally
    from each  $(d - 1)$ -extremal entry in set  $S_{(d-1)}$  to find  $d$ -adjacent entries;
    from each  $d$ -adjacent entry, compute the farthest  $d$ -valued
    entry reachable diagonally from it;
  end
for  $i = 1$  to  $n - m + 1$  do
  begin
    select lowest  $d$ -entry on diagonal  $i$ 
    and put it in the set  $S_d$  of  $d$ -extremal entries
  end
end.

```

It is easy to check that the algorithm performs k iterations in each one of which it does essentially a constant number of manipulations on each of the n diagonals. In turn, each one of these manipulations takes constant time except at the point where we ask to reach the farthest d -valued entry from some other entry on a same diagonal. We would know how to answer quickly that question if we knew how to handle the following query: given two arbitrary positions i and j in the two strings y and x , respectively, find the longest common prefix between the suffix of y that starts at position i and the suffix of x that starts at position j . In particular, our bound would follow if we knew how to process each query in constant time. It is not known how that could be done without preprocessing becoming somewhat heavy. On the other hand, it is possible to have it such that *all* queries have a cumulative amortized cost of $O(kn)$. This possibility rests on efficient algorithms for performing **lowest common ancestor** queries in trees. Space limitations do not allow us to belabor this point any further.

Note that the special case where insertions and deletions are forbidden is also solved by an algorithm very similar to the above and within the same time bound. This variant of the problem is often called **string searching with mismatches**. A probabilistic approach to this problem is implicit in [Chang and Lawler, 1990], one more is described in [Atallah et al., 1993]. When k cannot be considered a constant, an interesting alternative results from Abrahamson's

approach to multiple-value string searching.

Specifically, this algorithm of Abrahamson's combines divide and conquer with the idea of Fischer and Paterson [1974] which was discussed earlier. In divide-and-conquer, the problem is first partitioned into subproblems; these are then solved by ad-hoc techniques, and finally the partial solutions are combined. One possible way to "divide" is to take projections of the pattern into two complementary subsets, another is to split and handle separately the positive and negative portions of the pattern. We have already seen that the adaptation of fast multiplication to string searching leads to a time bound $O(n(|\Sigma|)\log^2 m \log \log m)$.

This performance is good for bounded Σ but quite poor when Σ is unbounded. In this latter case, however, some of the symbols must be very infrequent. Using this observation, Abrahamson designed a projection into $\Sigma' = \{\sigma \in \Sigma : \sigma \text{ occurs at most } z \text{ times in } y\}$ and the corresponding complement set Σ'' . The rare symbols can be handled efficiently by some direct match-counting, since they cannot produce more than zn matches in total. The frequent ones are limited in number to m/z and we can apply multiplication to each one of them separately. The overall result is time $O(nm/z \log^2 m \log \log m)$, which becomes $O(nm^{1/2} \log m \log \log^{1/2} m)$ if we pick $z = m^{1/2} \log m \log \log^{1/2} m$.

5 Two Dimensional Matching

The problem of matching and searching of two-dimensional objects arises in as many applications as there are ways to involve pictures and other planar representations and objects. Just like the full-fledged problem of recognizing the digitized signal of a spoken word in a speech finds a first rough approximation in string searching, the problem of recognizing a particular subject in a scene finds a first, simplistic model in the computational task that we consider in this section: locating occurrences of a small array into a larger one. Even at this level of simplification, this task is enough complicated already that we shall ignore such variants as those allowing for different shapes and rotations, variants that do not appear in the one dimensional searches.

Two-dimensional matching may be exact and approximate just like with strings, but

edit operations of insertion and deletion denature the structure of an array and thus may be meaningless in most settings. The literature on two-dimensional searching concentrates on exact matching, and so does the treatment of this section.

5.1 Searching with automata

In exact two-dimensional searching, the input consists of a “text” array $X[n \times n]$ and a “pattern” array $Y[m \times m]$. The output consists of all locations (i, j) in X where there is an occurrence of Y , where the word “occurrence” is to be interpreted in the obvious sense that $X_{i+k, j+l} = Y_{k+1, l+1}$, $0 \leq k, l \leq m - 1$.

The naive attack leads to an $O(n^2 m^2)$ solution for the problem. It is not difficult to reduce this down to $O(n^2 m)$ by resorting to established string searching tools. This may be seen as follows. Imagine to build a linear pattern y where each character consists of one of the consecutive rows of Y . Now, build similarly the family of textstrings $x_1^{(i)} x_2^{(i)} \dots x_{n-m+1}^{(i)}$ ($1 \leq i \leq n$) such that $x_j^{(i)}$ is the character for $X_{i, j} X_{i, j+1} \dots X_{i, j+m-1}$. Clearly, Y occurs at $X_{i, j}$ iff y occurs at $x_j^{(i)}$. If one could assume a constant cost for comparing a character of y with one of $x^{(i)}$, it would take $O(n)$ time by any of the known fast string searching to find the occurrences on y in each $x^{(i)}$. Hence, it would take optimal time $O(n^2)$ for the n strings in the global problem. Since comparing two strings of m characters each charges in fact m comparisons, then the overall bound becomes $O(n^2 m)$, as stated.

Automata-based techniques were developed along these lines by Bird [1977] and Baker [1978]. Later efforts exposed also a germane problem which came to be called “dictionary matching” and acquired some independent interest. Some details of such an automata-based two-dimensional searching are given next.

The main idea is to build on the distinct rows of the pattern Y the Aho-Corasick [1975] automaton for multiple string searching. Once this connection is made, it becomes possible to solve the problem at a cost of preprocessing time $O(m^2 \log |\Sigma|)$ (to build the automaton for at most m patterns with m characters each), and time $O(n^2 \log |\Sigma| + tocc)$ to scan the text. Here $tocc$, stands for total number of occurrences, i.e., is the size of the output. In multiple string

matching, the parameter *tocc* may play havoc with time linearity, since more than one pattern might end and thus have to be outputted at any given position. Here, however, the rows of Y are all of the same size, whence only one such row may occur at any given position.

5.2 Periods and witnesses in two dimensions

Automata-based approaches such as those just discussed result in time complexities that carry a dependence to alphabet size. This is caused by the branching of forward transitions that leave the states of the machine in multiple string searching. Single string searching is not affected by this problem. In fact, single string searching found quickly linear solutions without alphabet dependency. In contrast, several years elapse before alphabet dependency was eliminated from two-dimensional searching.

Alphabet dependency was eliminated in steps, first from the search phase only, and finally also from preprocessing. A key factor in the first step of progress was offered by a two-dimensional extension of the notion of a **witness**, a concept first introduced and used by U. Vishkin [1985] in connection with parallel exact string searching. It is certainly rare, and therefore quite remarkable, that a tool devised specifically to speed-up a *parallel* algorithm would find use in designing a better serial algorithm.

It is convenient to illustrate the idea of a witness on strings. Assume then to be given two copies of a pattern y , reciprocally aligned in such a way that the top copy is displaced, say, d positions ahead of the bottom one. A witness for d , if it exists, is any pair of mismatching characters that would prevent the two superimposed copies of y to coexist. Thus, if we were to be given two d -spaced, overlapping candidate occurrences of y on a text x , and a witness were defined for d , then at least one of the candidate occurrences of y in x will necessarily fail. One alternative way to regard a witness at d is as a counterexample to the claim that d is a period for y . The latter is a necessary, though not sufficient condition for having y occur twice, d positions apart.

The use of witnesses during the search phase presupposes preparation of appropriate tables. These tables essentially provide, for each d where this is true, a mismatch proving

the incompatibility of two overlapping matches at a distance of d . The notion of a witness generalizes naturally to higher dimensions. In two dimensions two witnesses tables were introduced by Amir, Benson and Farach [1994] as follows. Witness $Wit_{i,j}$ is any position (p, q) such that $X_{i+p,j+q}$ does not match $X_{i,j}$ or else it is 0. Note that, given an array W , there are essentially only two ways of superpositions one of the two copies onto the other. These consist, respectively, of shifting one of the copies towards the right and bottom or towards the right and top, of the other. These two families correspond to two witness tables that depend on whether $i < 0$ or $i \geq 0$. Amir, Benson and Farach [1994] showed how to build the witness table in time $O(m^2 \log |\Sigma|)$.

Once the table is available, the search phase is performed in two stages that are called respectively **candidate consistency testing** and **candidate verification**. The candidates are the positions of X , interpreted as top-left corners of potential occurrences of the pattern. At the beginning each position is a viable candidate. A pair of candidates is consistent if the pattern could be placed at both places without conflicting with the witness tables. The task of the first phase is to use the witness tables to remove one in each pair of inconsistent candidates. Clearly, one character comparison with the position of the text array that corresponds to the witness suffices to carry out this “duel” between the candidates. Note that a duel might rule out both candidates, however, eliminating one will do.

At the end of the consistency check we can verify the surviving candidates. A same text symbol could belong to several candidates, but all of these candidates must agree on that symbol. Thus, each position in the text can be labeled true or false according to whether or not it complies with what all participating candidates surrounding it prescribe for that position. Conversely, whenever a candidate covers a position of the text that is labeled as false, then that candidate can no longer survive. A procedure set up along these lines leads to an $O(n^2)$ search phase, within a model of computation in which character comparisons take constant time and only result in assessing whether the characters are equal or unequal.

The preprocessing in this approach is still dependent on the size of the alphabet. Alphabet independent preprocessing and overall linear time algorithm was achieved by Galil and Park [1992]. Like with strings, one may build an index structure based on preprocessing of the

text and then run faster queries off-line with varying patterns. Details can be found in, e.g., [Giancarlo and Grossi, 1995].

6 Tree matching

The discrete structures considered in this section are labeled, rooted trees, with the possible additional constraint that children of each node be ordered. Recall that a **tree** is any undirected, connected and acyclic graph. Choosing one of the vertices as the **root** makes the tree **rooted**, and fixing an order among the children of each node makes the tree **ordered**. Like with other classes of discrete objects, there is exact and approximate searching and matching of trees. We examine both of these issues next.

6.1 Exact tree searching

In exact tree searching, we are given two ordered trees, namely, a “pattern” tree P with m nodes and a “text” tree T with n nodes, and we are asked to find all occurrences of P in T . An occurrence of P in T is an ordered subtree P' rooted at some node ν of T such that P could be rigidly superimposed onto P' without any label mismatch or edge skip. The second condition means that the k -th child of a node of P matches precisely the k -th child of a node of T .

An $O(nm^{0.75}\text{polylog}(m))$ improvement over the trivial $O(mn)$ time algorithm was designed by Kosaraju [1992]. A faster, $O(n\sqrt{m}\text{polylog}(m))$ algorithm, is due to Dubiner, Galil and Magen [1994]. Their approach is ultimately reminiscent of Abrahamson’s pidgeon-hole approach to generalizations of string searching such as those examined earlier in our discussion. It is based on a combination of periodicity properties in strings and some techniques of tree partitioning that achieve succinct representations of long paths in the pattern.

Some notable variants of exact tree pattern matching arise in applications such as code generation and unification for logic programming and term-rewriting systems. In this context, a label can be a constant or a variable, where a variable at a leaf may match an entire subtree. In the most general setting, the input consists of a set S of patterns, rather than a single pattern, and of course of a text T . Early analyses and algorithms for the general problem are

due to Hoffman and O'Donnel [1982]. Two basic families of treatment descend, respectively, from matching the text tree from the root or from the leaves. The bottom-up approach is the more convenient of the two in the context of term rewriting systems. This approach is heavy on pattern preprocessing, where it may require exponential time and space, although essentially linear in the processing phase. Improvements and special cases are treated by Chase [1987], Cai, Paige and Tarjan [1992], and Thorup [1994].

6.2 Tree editing

The editing problem for unordered trees is NP-complete. However, much faster algorithms can be obtained for ordered trees. Early definitions and algorithms may be traced back to Selkow [1977] and Tai [1979]. In more recent years, the problem and some of its basic variants have been studied extensively by Shasha and Zhang and their co-authors. The outline given below concentrates on some of their work.

Let T be a tree of $|T| = n$ nodes, each node labeled with a symbol from some alphabet Σ . We consider three edit operations on T , consisting respectively of the deletion of a node ν from T (followed by the re-assignment of all children of ν to the node of which ν was formerly a child), the insertion of a new node along some consecutive arcs departing from a same node of T , and the substitution of the label of one of the nodes of T with another label from Σ . Like with strings, we assume that each edit operation has an associated non-negative real number representing the cost of that operation. We similarly extend the notion of edit script on T to be any consistent sequence Γ of edit operations on T , and define the cost of Γ as the sum of all costs of the edit operations in Γ . These notions generalize easily to any ordered **forest** of trees.

Now, let F and F' be two forests of respective sizes $|F| = n$ and $|F'| = m$. The **forest editing problem** for input F and F' consists of finding an edit script Γ' of minimum cost that transforms F into F' . The cost of Γ' is the edit distance from F to F' . When F and F' consist each of exactly one tree, then we speak of the **tree editing problem**.

A convenient way to visualize the editing of trees or forests is by means of a mapping of

nodes from one of two structures to the other. The map is represented by a set of links between node pairs (ν, ν') such that either these two nodes have precisely the same label –and thus node ν is exactly conserved as ν' – or else the label of ν gets substituted with that of ν' . Each node takes part in at most one link. The unaffected nodes of F (respectively, F') represent deletions (resp., insertions). A mapping defined along these lines has the property of preserving both ancestor-descendant and sibling orders. In other words, a link from a descendant of ν may only reach a descendant of ν' , and, similarly, links from two siblings to two others must not cross each other.

Early dynamic programming solutions for tree editing consume $\Theta(|F|^3|F'|^3)$ time. Much faster algorithms have been set up subsequently. Some other interesting problems are special cases of forest editing, including “tree alignment”, the “largest common subtree” problem, and the problem of “approximate tree matching” between a pattern tree and text tree. While any solution to the general tree editing problem implies similar bounds for all these special cases, some of the latter admit a faster treatment.

We review the criterion that subtends the computation of tree edit distances by dynamic programming after Zhang and Shaha [1989]. This leads to an algorithm with time bounded by the product of the squares of the sizes of the trees. A convenient preliminary step is to resort to a linear representation for the trees involved. The discussion on mappings suggests that such a representation consist of assigning to each node its ordinal number in the postorder visit of the tree. Let x and y be the strings representing the postorder visits of two trees T and T' . Then a prefix of, say, x will identify in general some forest of subtrees each rooted at some descendant of the root of T . Note that the leftmost leaf in the leftmost tree is denoted precisely x_1 . Let i_1 be the corresponding root, and let $forestdist(i, j)$ represent the cost of transforming the subforest of T corresponding to $x_1 \dots x_i$ into the subforest of T' corresponding to $y_1 \dots y_j$. Let $treedist(i, j)$ be the cost of transforming the tree rooted at x_i into the tree rooted at y_j . Then, in the most general case, these costs are dictated by the following recurrence:

$$forestdist(i, j) = \min \begin{cases} forestdist(i-1, j) + D(x_i) \\ forestdist(i, j-1) + I(y_j) \\ forestdist(l(i)-1, l(j)-1) + treedist(i, j) \end{cases}$$

Here $l(i)$ (resp., $l(j)$) is the index in x (resp., y) of the leftmost leaf in the subtree rooted at the node x_i (x_j). We leave the initialization conditions for an exercise. Note that *treedist* is little more than a notational convention, since it is a special case of *forestdist*, and thus is computed essentially through the same recursion. In fact, a recursion in the above form can be applied to any pair of substrings of x and y , with obvious meaning. In the special case where both forests consist of a single tree, i.e., x_i and y_j have x_1 and y_1 as their respective leftmost leaves, then *treedist*(i, j) becomes the substitution cost $S(x_i, y_j)$.

Building the algorithm around the above recurrence, and the subtended postorder visits, brings about an important advantage: each time that *treedist* is invoked, the main ingredients for its computation (namely, the pairwise distances of subtrees thereof) are already in place and thus need not be re-computed from scratch. We illustrate this point using $C(i, j)$, ($0 \leq i \leq |x|$, $0 \leq j \leq |y|$) as shorthand for *forestdist*. Observe that the recurrence above indicates that the value of $C(i, j)$ depends, in addition to the two neighboring values $C(i - 1, j)$ and $C(i, j - 1)$, on one generally more distant value $C(i', j')$. The pair (i', j') is called the *conjugate* of pair (i, j) . The following facts are easy to check.

Fact 1 *Every pair (i, j) has at most one conjugate.*

Fact 2 *If (i, j) has conjugate (i', j') , then, for any pair (k, l) with $i' \leq k \leq i$ and $j' \leq l \leq j$ we also have $i' \leq k' \leq i$ and $j' \leq l' \leq j$.*

Figuratively, Fact 1 states that each pair (i, j) of C is associated with exactly one (possibly empty) submatrix of C , with upper-left corner at the conjugate (i', j') of (i, j) (inclusive) and lower right corner at $(i - 1, j - 1)$ (inclusive). Fact 2 states that the submatrices defined by two pairs and their corresponding conjugates are either nested or disjoint.

Like in the case of string editing, the “close” interdependencies among the entries of the C -matrix induce an $(|x| + 1) \times (|y| + 1)$ “grid directed acyclic graph” (GDAG for short). String editing can be viewed as a shortest-path problem on a GDAG. To take care also of the interdependencies by conjugacy that appear in tree editing, however, the GDAG must be augmented by adding to the grid outerplanar edges connecting pairs of conjugate points.

Formally, an $l_1 \times l_2$ Augmented GDAG (or AGDAG) is a directed acyclic graph whose vertices are the $l_1 l_2$ points of an $l_1 \times l_2$ grid, and such that the only edges from point (i, j) are to grid points $(i, j + 1)$, $(i + 1, j)$, $(i + 1, j + 1)$ and $(i' - 1, j' - 1)$, where (i, j) is the conjugate of (i', j') . We refer to Figure 3 for an example. We make the convention of drawing the points such that point (i, j) is at the i -th row from the top and j -th column from the left. The top-left point is $(0, 0)$ and has no edge entering it (i.e., is a “source”), and the bottom-right point is (m, n) and has no edge leaving it (i.e., is a “sink”).

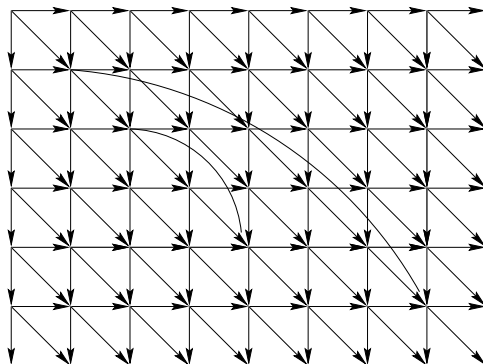


Figure 3: The upper-left corner of an AGDAG highlights the basic structure of such graphs: a grid with occasional outer-planar edges

We associate an $(|x| + 1) \times (|y| + 1)$ AGDAG G with the tree editing problem in the natural way: the $(|x| + 1)(|y| + 1)$ vertices of G are in one-to-one correspondence with the $(|x| + 1)(|y| + 1)$ entries of the C -matrix. We draw edges connecting a point to its neighbors in the planar grid of the AGDAG, while the edge that is incident on point $(i - 1, j - 1)$ from the unique conjugate of (i, j) , if the latter exists, are drawn outerplanar. Clearly, the cost of a grid edge from vertex (k, l) to vertex (i, j) is equal to $I(y_j)$ if $k = i$ and $l = j - 1$, to $D(x_i)$ if $k = i - 1$ and $l = j$, to $S(x_i, y_j)$ if $k = i - 1$ and $l = j - 1$. The cost of an outerplanar edge is the cost of the optimal solution to the submatrix associated with that edge. Thus, edit scripts that transform x into y or vice versa are in one-to-one correspondence to certain weighted paths of G that originate at the source (which corresponds to $C(0, 0)$) and end on the sink (which corresponds to $C(|x|, |y|)$). Specifically, in any such path horizontal or vertical edges can be traversed unconditionally, but the traversal of a diagonal edge from $(i - 1, j - 1)$

to (i, j) is allowed only if it follows the traversal of the outerplanar edge that is incident upon $(i - 1, j - 1)$ (if it exists). The details are left for an exercise.

7 Research Issues and Summary

The focus of this Chapter is represented by combinatorial and algorithmic issues of searching and matching with strings and other simple structures like arrays and trees. We have reviewed the basic variants of these problems, with the notable exception of exact string searching. The latter is definitely the primeval problem in the set, and has been devoted so much study to warrant a separate chapter in the present Handbook.

We started by reviewing, in Section 2, string searching in the presence of don't care symbols. In Section 3, we considered the problem of comparing two strings for similarity, under some basic sets of edit operations. This latter problem subtends the important variants of string searching where the occurrences of the pattern need not be exact, rather, they might be corrupted by a number of mismatches, and possibly by insertions and deletions of symbols as well. We abandoned the realm of one-dimensional pattern matching in Section 5, in which we highlighted the comparatively less battered topics of exact searching on two dimensional arrays. Finally, in Section 6, we reviewed exact and approximate searching on rooted trees.

As said at the beginning, most pattern matching issues are still subject to extensive investigation. Meanwhile, new problems and variants continue to arise in application areas that feature, in prominent position, the very information infrastructure under development. In most cases, the goal of current studies is to design better serial algorithms than those previously available. Parallel or distributed versions of the problems are also investigated. Typically, the solutions of such versions may be expected not to resemble in any significant way their serial predecessors. In fact (as exemplified by the previously encountered notion of a witness) they are more likely to expose novel combinatorial properties, some of which of intrinsic interest. Whether a problem be regarded within a serial, parallel, or distributed computational context, algorithms are also sought that display a good expected, rather than worst-case, performance. Relatively little work has been performed from this perspective, which requires often a thorough

re-examination of the problem and may result in a totally new line of attack, as experienced in such classical instances as the Boyer-Moore string searching algorithm and Quicksort.

An exhaustive list of specific open problems of Pattern Matching would be impossible. Here we limit mention to a few important ones.

For problems of searching with don't care, string editing, longest common subsequence and variations thereof, there are still wide and little understood gaps between the known, often trivial lower bounds and the efficiency of available algorithms. Likewise, relatively little is known in terms of nontrivial lower bounds for two-dimensional searches with mismatches, and also for both exact and approximate tree matching. Some general problems of fundamental nature remain unexplored across the entire board of pattern structures, problem variations, and computational models. Notable among these is the problem of preprocessing the "text" structure so that "patterns" presented on-line can be searched quickly thereafter. Such an approach has long been known to be elegantly and efficiently viable for exact searching on strings, but remains largely unexplored for approximate searches of all kind of patterns. The latter represent possibly the most recurrent queries in applications of molecular biology, information retrieval and other fields, so that progress in this direction would be valued enormously.

8 Defining Terms

Antichain: A subset of mutually incomparable elements in a partially ordered set.

Block: A sequence of don't care symbols.

Candidate consistency testing: The stage of two dimensional matching where it is checked whether a candidate occurrence of the pattern is checked against the "witness" table.

Candidate verification: The stage of two-dimensional searching where candidate occurrences of the pattern, not ruled out previously as mutually incompatible, are actually tested.

Chain: A linearly ordered subset of a partially ordered set.

D-adjacent: An entry reachable from a $(d - 1)$ -extremal entry through a unit vertical, horizontal or diagonal-mismatch step.

Divide and conquer: One of the basic paradigms of problem solving, in which the problem is decomposed (recursively) into smaller parts, solutions are then sought for the subproblems and finally combined in a solution for the whole.

Don't care: A “wildcard” symbol matching any other symbol of a given alphabet.

Edit operation: On a string, the operation of deletion, or insertion or substitution, performed on a single symbol. On a tree T , the deletion of a node ν from T followed by the re-assignment of all children of ν to the node of which ν was formerly a child, or the insertion of a new node along some consecutive arcs departing from a same node of T , or the substitution of the label of one of the nodes of T with another label from Σ . Each edit operation has an associated nonnegative real number representing its cost.

Edit distance: For two given strings, the cost of a cheapest edit script transforming one of the strings into the other.

Edit script: a sequence of viable edit operations on a string.

Exact string searching: The algorithmic problem of finding all occurrences of a given string usually called “the pattern” in another, larger “text” string.

Extremal: Some of the entries of the auxiliary array used to perform string searching. An entry is d -extremal if it is the deepest entry on its diagonal to be given value d .

Forest: A collection of trees.

Forest editing problem: The problem of transforming one of two given forests into the other by an edit script of minimum cost.

Linear product: For two vectors X and Y , and with respect to two suitable operations \otimes and \oplus , is a vector $Z = Z_0 Z_1 \dots Z_{m+n}$ where $Z_k = \bigoplus_{i+j=k} X_i \otimes Y_j$ ($k = 0, \dots, m+n$).

Local alignment: the detection of local similarities among two or more strings.

Longest (or heaviest) common subsequence problem: The problem of finding a maximum-length (or maximum weight) subsequence for two or more input strings.

Lowest common ancestor: The deepest node in a tree that is an ancestor of two given

leaves.

K-dominant match: A match $[i, j]$ having rank k and such that for any other pair $[i', j']$ of rank k either $i' > i$ and $j' \leq j$ or $i' \leq i$ and $j' > j$.

Match: The result of comparing two instances of a same symbol.

Minimal antichain decomposition: A decomposition of a poset into the minimum possible number of antichains.

Offset: The distance from the beginning of a string to the end of a segment in that string.

Pattern element: A positive (negative) pattern element is a “partial wildcard” presented as a subset of the alphabet Σ , with the symbols in the subset specifying which symbols of Σ are matched (mismatched) by the pattern element.

Picture: A collection of mutually disjoint subsets of an alphabet.

Poset: A set the elements of which are subject to a partial order.

Rank: For a given match, this is the number of matches in a longest chain terminating with that match, inclusive.

Segment: The substring of a pattern delimited by two don't cares or one don't care and one pattern boundary.

Sparsity: Used here to refer to LCS problem instances in which the number of matches is small compared to the product of the lengths of the input strings.

String editing problem: For input strings x and y , is the problem of finding an edit script of minimum cost that transforms y into x .

String searching with errors: Searching for approximate (e.g., up to a predefined number of symbol mismatches, insertions and deletions) occurrences of a pattern string in a text string.

String searching with mismatches: The special case of string matching with errors where mismatches are the only type of error allowed.

Subsequence: Of a string, is any string that can be obtained by deleting zero or more symbols

from that string.

Tree: A graph undirected, connected and acyclic. In a rooted tree, a special node is selected and called the root: the nodes reachable from a node by crossing arcs in the direction away from the root are the children of that node. In unordered rooted trees, there is no pre-set order among the children of a node. Assuming such an order makes the tree ordered.

Tree editing problem: The problem of transforming one of two given trees into the other by an edit script of minimum cost.

Witness: A mismatch of two symbols of string y at a distance of d is a “witness” to the fact that in no subject y could occur twice at a distance of exactly d positions (equivalently, that d cannot be a period of y).

9 Acknowledgements

This work was supported in part by NSF Grants CCR-9201078 and CCR-9700276, by NATO Grant CRG 900293, by the National Research Council of Italy, by British Engineering and Physical Sciences Research Council Grant GR/L19362. Xuyan Xu contributed to Section 2 through bibliographic searching and drafting. The Referees carried out a very careful scrutiny of the manuscript and made many helpful comments.

References

- ABRAHAMSON, K. [1987], “Generalized String Matching”, *SIAM. J. Comput* 16(6): 1039-1051.
- AHO, A.V. [1990], “Algorithms for finding patterns in strings”, *Handbook of Theoretical Computer Science* (J. van Leeuwen, Ed.), Elsevier, Amsterdam, 255-300.
- AHO, A.V., AND M.J. CORASICK [1975] “Efficient String Matching: An Aid to Bibliographic Search”, *CACM* 18(6): 333-340.
- AHO, A.V., D.S. HIRSCHBERG AND J.D. ULLMAN [1976], “Bounds on the complexity of the longest common subsequence problem”, *J. Assoc. Comput. Mach.* 23(1): 1-12.

- AMIR, A., G. BENSON, AND M. FARACH [1994], “An alphabet independent approach to two dimensional matching”, *SIAM J. Comp.* 23(2): 313-323.
- APOSTOLICO, A., S. BROWNE AND C. GUERRA [1992], “Fast linear space computations of longest common subsequences”, *Theoretical Computer Science*, 92(1): 3–17.
- APOSTOLICO, A. AND C. GUERRA [1987], “The longest common subsequence problem revisited”, *Algorithmica* 2: 315–336.
- ARLAZAROV, V. L., E. A. DINIC, M. A. KRONROD, AND I. A. FARADZEV [1970], “On economical construction of the transitive closure of a directed graph,” *Dokl. Akad. Nauk SSSR* 194: 487-488 (in Russian). English translation in *Soviet Math. Dokl.* 11(5): 1209-1210.
- ATALLAH, M.J., P. JACQUET AND W. SZPANKOWSKI [1993], “A Probabilistic Approach to Pattern Matching with Mismatches” *Random Structures and Algorithms* 4: 191-213.
- BAKER, T.P. [1978], “A technique for extending rapid exact-match string matching to arrays of more than one dimension”, *SIAM J. Comp* 7(4): 533–541.
- BIRD, R.S. [1977], “Two dimensional pattern matching”, *Information Processing Letters* 6(5): 168–170.
- CAI, J., R. PAIGE AND R. TARJAN [1992], “More efficient bottom-up multi-pattern matching in trees”, *Theoretical Computer Science* 106(1): 21-60.
- CHANG, W.I. AND E.L. LAWLER [1990], “Approximate string matching in sublinear expected time”, in *Proc. 31st Annual IEEE Symp. on Foundations of Vomputer Science*, St. Louis, MO., 116–124
- CHASE, D. [1987], “An improvement to bottom-up tree pattern matching” *Proceedings of the 14th Annual ACM Symp. on POPL*, 168-177.
- CHAZELLE, B. [1988], “A functional approach to data structures and its use in multidimensional searching”, *SIAM. J. Comput.* 17(3): 427-462.
- DILWORTH, R. P. [1950], “A decomposition theorem for partially ordered sets”, *Ann. Math.* 51, 161–165.

- DUBINER, M., Z. GALIL AND E. MAGEN [1994]. “Faster tree pattern matching”, *JACM* 14(2): 205-213.
- VAN EMDE BOAS, P. [1975], “Preserving order in a forest in less than logarithmic time,” *Proc. 16th FOCS*, 75-84.
- FISCHER, M.J. AND M. PATERSON [1973], “String Matching and Other Products”, in: Karp, R., ed., *Complexity of Computation, SIAM-AMS Proceedings 7*. 113-125.
- FREDMAN, M.L. [1975], “On Computing the Length of Longest Increasing Subsequences”, *Discrete Mathematics* 11: 29-35.
- GALIL Z. AND R. GIANCARLO [1988], “Data structures and algorithms for approximate string matching,” *Jour. Complexity* 4: 33-72.
- GALIL, Z. AND K. PARK [1990], “An improved algorithm for approximate string matching,” *SIAM Jour. Computing* 19(6): 989-999.
- GALIL, Z. AND K. PARK [1992], “Truly alphabet-independent two-dimensional pattern matching”, *Proc. 33rd Symposium on the Foundations of Computer Science (FOCS 92)*, 247-256.
- GIANCARLO, R., AND R. GROSSI [1995], “On the construction of classes of suffix trees for square matrices: Algorithms and applications,” *22nd Int. Colloquium on Automata, Languages, and Programming*, Z. Fulop and F. Gecseg eds, LNCS 944, 111-122.
- HIRSCHBERG, D.S. [1977], “Algorithms for the longest common subsequence problem”, *JACM* 24(4): 664-675.
- HIRSCHBERG, D. S. [1978], “An information theoretic lower bound for the longest common subsequence problem,” *Inform. Process. Lett.* 7(1): 40-41.
- HOFFMAN, C. AND J. O’DONNELL [1982]. “Pattern matching in trees”, *JACM* 29(1): 68-95.
- HUNT, J.W. AND T.G. SZYMANSKI [1977], “A fast algorithm for computing longest common subsequences”, *CACM* 20(5): 350-353.
- JACOBSON, G. AND K.P. VO [1992], “Heaviest increasing/common subsequence problems”, in *Combinatorial Pattern Matching, Proceedings of the Third Annual Symposium*, A. Apostolico, M. Crochemore, Z. Galil and U. Manber, Eds., Tucson, Arizona, 1992. Springer Verlag Lecture notes in Computer Science 644, 52-66.

- JIANG, T., L. WANG AND K. ZHANG [1994]. "Alignment of trees - an alternative to tree edit", *Proceedings of the Fifth Symposium on Combinatorial Pattern Matching*, 75-86.
- KNUTH, D.E., J.H. MORRIS AND V.R. PRATT [1977] "Fast pattern matching in strings", *SIAM. J. Comput.* 6(2): 323-350.
- KOSARAJU, S.R. [1992]. "Efficient tree pattern matching", *Proceedings of the 30th annual IEEE Symposium on Foundations of Computer Science*, 178-183.
- KUMAR, S.K. AND C.P. RANGAN [1987], "A linear space algorithm for the LCS problem", *Acta Informatica* 24: 353-362.
- LANDAU. G. M. AND U. VISHKIN [1986], "Introducing efficient parallelism into approximate string matching and a new serial algorithm", in *Proc. 18th Annual ACM STOC*, New York, 1986, 220-230.
- LANDAU, G. M. AND U. VISHKIN [1988], "Fast string matching with k differences," *Jour. Comp. and System Sci.* 37: 63-78.
- LEVENSHTAIN, V.I. [1966], "Binary codes capable of correcting deletions, insertions and reversals", *Soviet Phys. Dokl.* 10: 707-710.
- MANBER, U. AND R. BAEZA-YATES [1991], "An algorithm for string matching with a sequence of don't cares ", *Inform. Process. Lett.* 37(3): 133-136.
- MANBER, U. AND E.W. MYERS [1990], "Suffix Array: A new method for on-line string searches", in: *Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, CA (1990) 319-327.
- MASEK, W.J. AND M. S. PATERSON [1980], "A faster algorithm computing string edit distances", *J. Comput. System Sci.* 20(1): 18-31.
- MUTHUKRISHNAN, S. AND R. HARIHARAN [1995], "On the equivalence between the string matching with don't cares and the convolution", *Information and Computation* 122(1): 140-148.
- MYERS, E.W. [1986], "An $O(ND)$ difference algorithm and its variations", *Algorithmica* 1: 251-266.

- NEEDLEMAN, R.B. AND C.D. WUNSCH [1973], “A general method applicable to the search for similarities in the amino-acid sequence of two proteins”, *J. Molecular Bio.* 48: 443–453.
- PINTER, R. [1985], “Efficient string matching with don’t-care patterns”, in: Apostolico, A. and Galil, Z., eds., *Combinatorial Algorithms on Words*, Springer Verlag, NATO ASI Series 12, 11-29.
- SANKOFF, D.[1972], “Matching sequences under deletion-insertion constraints”, *Proc. Nat. Acad. Sci. U.S.A.* 69: 4–6.
- SANKOFF, D. AND P. H. SELLERS [1973], “Shortcuts, Diversions and Maximal Chains in Partially Ordered Sets”, *Discrete Mathematics* 4: 287–293.
- SELLERS, P.H. [1974], “The theory and computation of evolutionary distance”, *SIAM J. Appl. Math.* 26: 787–793.
- SELKOW, S.M. [1977]. “The tree-to-tree editing problem”, *Information Processing Letters* 6: 184-186.
- SHASHA, D., J.T.L. WANG AND K. ZHANG [1994]. “Exact and approximate algorithms for unordered tree matching”, *IEEE Trans. Systems, Man, and Cybernetics* 24(4): 668-678.
- SHASHA, D. AND K. ZHANG [1990]. “Fast algorithms for the unit cost editing distance between trees”, *J. Algorithms* 11: 581-621.
- SCHONHAGE, A. AND V. STRASSEN [1971], “Schnelle Multiplikation grosser Zahlen”, *Computing (Arch. Elektron. Rechnen)* 7: 281-292. MR 45 No. 1431
- TAI, K.C. [1979]. “The tree-to-tree correction problem”, *J. ACM* 26: 422-433.
- TAKEDA, M [1993], “A fast matching algorithm for patterns with pictures”, *Bull. Info. Cyber.* 25(3-4): 137-153.
- THORUP, M. [1994]. “Efficient Preprocessing of Simple Binary Pattern Forests”, *Proceedings of the 4th Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science* 824: 350-358.
- UKKONEN, E. [1985], “Finding approximate patterns in strings”, *Journal of Algorithms* 6: 132–137.

- VISHKIN, U. [1985], Optimal parallel pattern matching in strings, *Information and Control* 67(1-3): 91-113.
- WAGNER, R.A. AND M. J. FISCHER [1974], “The string to string correction problem”, *J. Assoc. Comput. Mach.* 21: 168–173.
- WILLARD, D.E. [1986], “On the application of sheared retrieval to orthogonal range queries”, in: *Proc. 2nd Annual ACM Symposium on Computational Geometry*, Yorktown Heights, NY, 80-89.
- WONG, C.K. AND A.K. CHANDRA [1976], “Bounds for the string editing problem”, *J. Assoc. Comput. Mach.* 23(1): 13–16.
- ZHANG, K. AND D. SHASHA [1989]. “Simple fast algorithms for the editing distance between trees and related problems”, *SIAM J. Computing* 18(6): 1245-1262.

10 Further Information

Most books on design and analysis of algorithms devote one or more chapters to Pattern Matching. Here, we limit mention to specialized sources.

The collection of essays *Combinatorics on Words*, published in 1982 by Addison Wesley under a fictitious editorship (M. Lothaire) contains most of the basic properties used in string searching, and more. An early attempt at unified coverage of string algorithmics is found in *Combinatorial Algorithms on Words*, edited by A. Apostolico and Z. Galil in 1985 for Springer-Verlag. *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*, edited by D. Sankoff and J. B. Kruskal (Addison-Wesley, 1983), represents still a valuable source for sequence analysis and comparison tools in computational biology and other areas. A few more volumes of recent years are, in order of appearance: *Text Algorithms* by M. Crochemore and W. Rytter (Oxford University Press, 1994), *String Searching Algorithms* by G.A. Stephen (World Scientific, 1994), *Pattern Matching Algorithms*, edited by A. Apostolico and Z. Galil (Oxford University Press, 1997), *Algorithms on Strings, Trees and Sequences* by D. Gusfield (Cambridge University Press, 1997). This last volume puts particular emphasis on issues arising in Computational Biology. A broader treatment of this field can be found

in *Introduction to Computational Biology* by M.S. Waterman (Chapman & Hall, 1995). *Data Compression, Methods and Theory* by J.A. Storer (Computer Science Press, 1988) describes applications of pattern matching to the important family of compression methods by “textual substitution”.

A rich bibliography on “words, automata and algorithms” is maintained by I. Simon of the University of São Paulo (Brazil). One on “sequence analysis and comparison” is maintained by William H. E. Day in Port Maitland, Canada. A collection of “pattern matching pointers” is currently maintained by S. Lonardi at <http://www.cs.purdue.edu/homes/stelo/pattern.html>.

Papers on the subject of Pattern Matching appear primarily in archival journals of Theoretical Computer Science, but important contributions are also found in journals of application areas such as computational biology (notably, *CABIOS* and *Journal of Computational Biology*) and various specialties of Computer Science (cf., e.g., *IEEE Transactions* on Information Theory, Pattern Recognition, Machine Intelligence, Software, etc.). Special issues have been dedicated to Pattern Matching by *Algorithmica* and *Theoretical Computer Science*. Papers on the subject are presented at most major conferences. The *International Symposia on Combinatorial Pattern Matching* have gathered yearly since 1990. Beginning in 1992, *Proceedings* have been published in the Lecture Notes in Computer Science Series of Springer-Verlag (serial numbers of volumes already published: 644, 684, 807, 937, 1075, 1264). Specifically flavored contributions appear also at conferences such as *RECOMB* (International Conference on Computational Molecular Biology), the *IEEE annual Data Compression Conference*, the *South American Workshop on String Processing*, and others.