Sequence Comparison: Some Theory and Some Practice

Imre Simon*
Instituto de Matemática e Estatística
Universidade de São Paulo
05508 São Paulo, SP, Brasil

April 2, 1988

Abstract

A brief survey of the theory and practice of sequence comparison is made focusing on diff, the UNIX[1] file difference utility.

1 Sequence comparison

Sequence comparison is a deep and fascinating subject in Computer Science, both theoretical and practical. However, in our opinion, neither the theoretical nor the practical aspects of the problem are well understood and we feel that their mastery is a true challenge for Computer Science.

The central problem can be stated very easily: find an algorithm, as efficient and practical as possible, to compute a longest common subsequence (lcs for short) of two given sequences[2].

As usual, a subsequence of a sequence is another sequence obtained from it by deleting some (not necessarily contiguous) terms. Thus, both `en␣pri` and `en␣pai` are longest common subsequences of `sequence␣comparison` and `theory␣and␣practice`.

*Part of this work was done while the author was visiting the Université de Rouen, in 1987. That visit was partially supported by FAPESP.

[1]UNIX is a trademark of Bell Laboratories

[2]The sequences we consider are usually called words. We avoid this terminology since it might lead to confusion because of the widespread misuse of the term subword (meaning a segment or a factor instead of a subsequence).

It turns out that this problem has many, many applications in many, many apparently unrelated fields, such as computer science, mathematics, molecular biology, speech recognition, gas chromatography, bird song analysis, etc. A comprehensive study of the role of the problem in these fields and how it is coped with in each of them can be found in the beautiful book of Sankoff and Kruskal [37].

In particular, in computer science the problem has at least two applications. The main one is a file comparison utility, nowadays universally called diff, after its popularization through the UNIX operating system. This tool is intensively used to discover differences between versions of a text file. In this role it is useful in keeping track of the evolution of a document or of a computer program. It is also used as a file compression utility, since many versions of a (long) file can be represented by storing one (long) version of it and many (short) scripts of transforming the stored version in the remaining ones. Another aplication in computer science is to approximate string matching used, for instance, in the detection of misspelled versions of names. It should be noted, however, that sequence comparison is not the main tool to solve this very important problem. For more details the reader is referred to [16].

An interesting aspect of the problem is that it can be solved by a simple and perhaps even intuitive 'folklore' algorithm based on a dynamic programming approach. This appealing algorithm has been discovered many, many times. Indeed, it has been discovered by engineers, by biologists and by computer scientists, in Russia, Japan, United States, France and Canada in the period 1968 to 1975. The first publication of the algorithm seems to be, according [37], in a 1968 paper by the russian engineer Vintsyuk [45].

The big challenge to computer science comes from the complexity of the folklore algorithm. Indeed, it requires time proportional to the product of the lengths of the sequences, and no essentially better practical algorithm is known. The question is to search for a possible algorithm which is simultaneously efficient and practical.

As far as we know, the existence of a linear algorithm has not been ruled out. Neither has been found a practical algorithm which worst case time complexity is better than $O(mn)$, where m and n are the lengths of the given sequences. There exists, however, an algorithm of time complexity $O(n^2/\log n)$ for pairs of sequences of length n over a fixed finite alphabet, discovered by Masek and Paterson [29]. This algorithm is not suitable for practical purposes but its existence is a hint that better algorithms than the ones in current use must exist.

All told, a very good start would be to find out whether or not there exists a linear time algorithm to compute a longest common subsequence of two given sequences over two letters. We get a more modest but still very interesting start by replacing "linear time" for "time complexity $O(n \log n)$". Here, the time bounds should be measured on a random access machine under the unit-cost model.

A weaker version, already answered by Masek and Paterson, has been proposed by D. E. Knuth in a technical report coauthored with V. Chvátal and D. A. Klarner in 1972 [9]:

```
Problem 35.   Greatest common substrings.
  It is possible to find the longest common subsequence of two
sequences of a´s and b´s in a time proportional to the product
of their lengths. Can one do better?
Note: aba is a subsequence of aabbbba.
```

Incidentally, this seems to be the first reference to the problem and to the folklore algorithm within Computer Science.

2 Some theory

We begin this section with the presentation of the folklore algorithm which is the starting point for most of the known algorithms to find an lcs of two given sequences.

Let $u = u_1 u_2 \cdots u_n$ and $v = v_1 v_2 \cdots v_m$ be sequences of lengths n and m over an alphabet A. We assume that each u_i and v_j is a letter in A. The folklore algorithm consists of computing the length $d(i,j)$ of an lcs of $u_1 \cdots u_i$ and $v_1 \cdots v_j$. This can be done by observing that $d(0,0) = 0$ and by repeatedly applying the formula:

$$d(i,j) = \begin{cases} 1 + d(i-1, j-1) & \text{if } u_i = v_j, \\ \max\{\, d(i-1,j), d(i,j-1) \,\} & \text{if } u_i \neq v_j. \end{cases}$$

Having computed the matrix d the length of an lcs of u and v is $d(n,m)$ and a common subsequence of length $d(n,m)$ can be computed easily proceeding backwards: from $d(n,m)$ to $d(0,0)$. Figure 1 shows an application of the folklore algorithm.

The time complexity of this algorithm is clearly $O(nm)$; actually this time does not depend on the sequences u and v themselves but only on their

	s	e	q	u	e	n	c	e	␣	c	o	m	p	a	r	i	s	o	n
t	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
h	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
e	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
o	0	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2
r	0	1	1	1	1	1	1	1	1	1	2	2	2	2	3	3	3	3	3
y	0	1	1	1	1	1	1	1	1	1	2	2	2	2	3	3	3	3	3
␣	0	1	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3
a	0	1	1	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3	3
n	0	1	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3	4
d	0	1	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3	4
␣	0	1	1	1	1	2	2	2	3	3	3	3	3	3	3	3	3	3	4
p	0	1	1	1	1	2	2	2	3	3	3	3	4	4	4	4	4	4	4
r	0	1	1	1	1	2	2	2	3	3	3	3	4	4	5	5	5	5	5
a	0	1	1	1	1	2	2	2	3	3	3	3	4	5	5	5	5	5	5
c	0	1	1	1	1	2	3	3	3	4	4	4	4	5	5	5	5	5	5
t	0	1	1	1	1	2	3	3	3	4	4	4	4	5	5	5	5	5	5
i	0	1	1	1	1	2	3	3	3	4	4	4	4	5	5	6	6	6	6
c	0	1	1	1	1	2	3	3	3	4	4	4	4	5	5	6	6	6	6
e	0	1	1	1	2	2	3	4	4	4	4	4	4	5	5	6	6	6	6

Figure 1: An example for the folklore algorithm

lengths. By choosing carefully the order of computing the $d(i,j)$'s one can execute the above algorithm in space $O(n+m)$. Even an lcs can be obtained within this time and space complexities but this requires a clever subdivision of the problem [19].

An important property of this dynamic programming algorithm is that it can easily be generalized to computing the minimum cost of editing u into v given (possibly different) costs of the operations: changing a letter into another and deleting or inserting a letter [46].

The literature contains a large number of variants of this algorithm, most of them are recorded in the bibliography. Just to give some idea about the various time bounds Table 1 lists a few results, where we assume that u and v have the same length n. In the table p denotes the length of the result (an lcs of u and v). Also, r denotes the number of matches, i.e. the number of pairs $(i,j) \in [1,n] \times [1,n]$ for which $u_i = v_j$.

Hunt and Szymanski(77) [25]	$O((r+n)\log n)$
Hirschberg(77) [20]	$O(pn+n\log n)$
Hirschberg(77) [20]	$O(((n+1-p)p\log n)$
Nakatsu et al.(82) [33]	$O(n(n-p))$
Hebrard(84) [17]	$O(pn)$

Table 1: Time complexities of some lcs algorithms

None of the algorithms in Table 1 have worst case time complexity better than $O(n^2)$, some are even worse. This can be seen by observing that the value of p varies between 0 and n, while that of r varies between 0 and n^2 and their average value, for pairs of sequences over a fixed alphabet, is proportional, respectively, to n and n^2. It is important to note, however, that, for particular cases, some of the algorithms might use considerably less time than in the worst case. A lively description, from a unified viewpoint, of two of the main algorithms and some variations can be found in [7], a recent paper by Apostolico and Guerra.

The most interesting theoretical result is that of Masek and Paterson [29]. Carefully using subdivision techniques similar to the ones used in the "Four russian's algorithm" they transformed the folklore algorithm into one with time complexity $O(n^2/\log n)$. This is indeed the only evidence that there exist faster algorithms than the folklore one.

Before talking about lower bounds we would like to clarify the model we think is appropriate for the lcs problem. First, the time complexity should be measured on a random access machine under the unit cost model. This seems to be the correct model because we are interested in practical algorithms and this is the closest we can get to existing computers. Second, the time complexity should be measured in terms of the total size, say t, of the input, instead of simply considering the length, say n, of the input sequences. This is a delicate point. For sequences over a known and fixed alphabet we can consider $t=n$ and this was the case considered until now. The other common assumption is to consider sequences over a potentially unbounded alphabet. In this case we assume that the letters are coded over some known, fixed and finite auxiliary alphabet; hence, to represent n different symbols we need size $t \in \Omega(n\log n)$. Thus, measuring complexity in terms of n or t turn out to be very different!

This model adjusts very well to the current philosophy of text files whenever each line is considered as a letter. In particular, this is the case of the file comparison utilities, our main example for the unbounded alphabet model. In essence we propose that in this case complexity should be measured in terms of the length of the files instead of their number of lines. The main consequence of this proposal is that it increases considerably, but within reasonable bounds, the world of linear algorithms: just for one example, sorting the lines of a text file can be done in linear time [2, 30].

The existing lower bounds for the complexity of the longest common subsequence problem [14, 1, 47, 21] are based on restricted models of computations and do not apply to the model we just proposed. This point seems to be responsible for a certain amount of confusion because sometimes the known lower bounds tend to be interpreted outside the model for which they were obtained. Indeed, as far as we are aware of, no known lower bound excludes the existence of a linear algorithm in the sense just outlined.

Another very interesting, apparently difficult and little developed area is the probabilistic analysis of the quantities envolved in the lcs problem. Let $f(n,k)$ be the average length of the lcs of two sequences of length n over an alphabet A of k letters (the uniform distribution on A^n is assumed). The function $f(n,k)$ has been explicitly computed for small values of n and k in [11]. On the asymptotic side it is known that for every k there exists a constant c_k such that

$$\lim_{n\to\infty} \frac{f(n,k)}{n} = \sup_n \frac{f(n,k)}{n} = c_k.$$

Thus, fixing the finite alphabet A, the length of an lcs of two random sequences in A^n is ultimately proportional to n.

The exact determination of c_k seems elusive and only lower and upper bounds are known for small values of k. Some results in this direction appear in Table 2. For more details, see [37, 13, 10].

k	lower bound	upper bound
2	0.76	0.86
5	0.51	0.67
10	0.40	0.54
15	0.32	0.46

Table 2: Some upper and lower bounds for c_k

An interesting conjecture was made by Sankoff and Mainville [37]:

$$\lim_{k\to\infty} \sqrt{k}c_k = 2.$$

We close this section mentioning five results related, in one way or another, to the lcs problem.

A very important subproblem is obtained by restricting the input to permutations (sequences in which each letter occurs at most once). This case was solved by Szymansky [42] in time $O(n \log n)$. Such an algorithm is also contained in work of Hunt and Szymanski [25] and that of Hunt and McIlroy [24]. It is an open problem whether or not the case of permutations can be done in linear time on the model we proposed.

A further restriction leads to yet another very important subproblem. This is obtained if we consider $(1, 2, \ldots, n)$ as one of the permutations, assuming, of course, the alphabet $[1, n]$. Then an lcs is just a longest increasing subsequence of the second permutation and this problem is part of a very rich theory of representations of the symmetric group using Young tableaux extensively studied by A. Young, G de B. Robinson, C. E. Schensted and M. P. Schützenberger. A survey focusing on the computational aspects can be found in Knuth's book [26] from which Fredman [14] extracted an algorithm to solve the longest increasing subsequence problem. His algorithm runs in time $O(n \log n)$ and he also derives $n \log n$ as a lower bound for the problem. But, beware, the lower bound does not apply to our model! Using the theory of Young tablaux one can compute the number of permutations of $[1, n]$ which has a longest increasing subsequence of any given length. Extensive calculations can be found in [8]. However, the expected value of the length of a longest increasing subsequence of a permutation of length n is not known but the data compiled in [8] indicate that this value is approximately $2\sqrt{n}$.

The third related problem is obtained if we look for a longest common segment of two sequences instead of a longest common subsequence. In [12] a linear algorithm was obtained to solve this problem.

The fourth related problem is obtained by considering mini-max duality: instead of looking for the longest common subsequence what about a shortest uncommon subsequence? In 1984 the author solved this problem with an algorithm of time complexity $O(|A| + |u| + |v|)$. More precisely, this (unpublished) linear algorithm computes a shortest sequence which distinguishes the sequences u and v over the alphabet A, that is to say, a shortest sequence which is a subsequence of exactly one of the unequal sequences u and v. For instance, consider `sequence␣comparison` and `theory␣and␣practice`: `eee` distinguishes them while `cc` does not. A shortest distinguisher is given by `d`.

The last related problem is a negative result (from our point of view). It was shown by Maier [28] that deciding whether or not a finite set of sequences has a common subsequence of length k is an NP-complete problem. Other related NP-complete problems can be found in [15].

3 Some practice

In this section we restrict ourselves to some practical aspects of file comparison utilities, focusing especially on the UNIX command diff.

A file comparison utility should determine the differences between two text files. But what is the difference of two text files? Indeed, our intuitive notion of such a difference is an elusive concept and it is difficult to define it. To cope with this it became common practice to consider entire lines as indivisible objects. Then it seems that the best results are obtained if one finds a longest common subsequence of lines and then anything not in this lcs is declared a difference.

The first (and still the best) file comparison utility was included in UNIX around 1976. Since then file comparison became a standard tool and with the proliferation of microcomputers many programs turned up. These are usually called diff, but most of them do not determine a true lcs; consequently they easily missynchronize. On the other hand, some of them are very fast and work well for many pairs of (real text) files. Many of these programs are based on [18].

One aspect of the file comparison programs for microcomputers is worth mentioning: their output is sometimes more suggestive (for a human) of the differences than the output of the original diff. Indeed, a good way of pinpoiting the differences seems to be a simultaneous listing of both files, indicating whether each line is common to both or exclusive to one of them. Long blocks of lines in the same class might be abbreviated by showing only their first and last lines. In contrast, the output of diff is thoroughly influenced by the intricacies of machine transformation of one file in another and this restricts, in our opinion, its potential as a tool for remembering or discovering the changes during the evolution of a file.

The algorithm actually used by diff is described by Hunt and McIlroy in [24]; its basic idea is attributed to unpublished work of H. S. Stone who generalized an $O(n \log n)$ solution of the most important particular case (the restriction of the problem to permutations) by T. G. Szymanski [42]. The resulting algorithm is very similar to the one in [25]; it is also described in

[3].

The first practical concession of diff is that it hashes the lines of the files. This is handy because it reduces significantly the volume of information to deal with. On the other hand, the hashing might introduce false matches caused by collisions; these are detected during the last phase when the computed lcs is checked in the files themselves. If false matches occur the corresponding lines are considered as differences. Consequently, it might happen that the reported common subsequence of lines is not a longest one. These events seems to be very rare in practice and the advantages of hashing greatly outweight its shortcomings.

The key concept in the algorithm is that of a k-candidate. Returning to our notations in the previous section, a *k-candidate* is a pair of positions (i, j) such that $u_i = v_j$ and

$$k = d(i,j) = d(i,j-1)+1 = d(i-1,j)+1 = d(i-1,j-1)+1.$$

It follows that every lcs of $u_1 \dots u_i$ and $v_1 \dots v_j$ is the concatenation of an lcs of $u_1 \dots u_{i-1}$ and $v_1 \dots v_{j-1}$ with the letter $u_i = v_j$. The set of k-candidates in Figure 1 is

$$(3,2),(9,6),(7,9),(4,11),(15,7),(11,9),(8,14),(5,15),$$
$$(19,8),(15,10),(12,13),(9,19),(14,14),(13,15),(17,16).$$

The basic strategy of the algorithm is to compute the set of all k-candidates and then collect an lcs from these (such an lcs clearly exists). The computation is done by performing a binary search, in a vector of at most n components, for certain matches, that is to say, pairs (i, j) for which $u_i = v_j$. Thus, r being the total number of matches this part of the algorithm takes time $O(r \log n)$ (we assume throughout that both input sequences have length n). The overall worst case time complexity of the algorithm is $O((r+n)\log n)$ and its space requirements are $O(q+n)$, where q is the total number of k-candidates encountered.

A key question is to investigate the total number q of candidates for particular pairs of sequences. This is interesting because $q \log n$ and q are lower bounds for the computing time and for the space requirements, once the present strategy is adopted. Unfortunately, there are pairs, such as $(abc)^n$ and $(acb)^n$ or $(abab)^n$ and $(abba)^n$ for which $q \in \Theta(n^2)$. Consequently, the derived upper bound can be obtained and the complexity of the algorithm is indeed $\Theta(n^2 \log n)$, that is the worst case behavior is even worse than that of the folklore algorithm.

The great advantage of this algorithm is that in the case of permutations the number r of matches is at most n, hence the algorthm works in time $O(n \log n)$. In actual practice the behavior of the algorithm is somewhere between these two bounds. Fortunately, most lines of true text files are either unique, or occur few times; hence, in practice this algorithm is definitely subquadratic! And this is why the algorithm works well even for long files (tens of tousands of lines).

One shortcoming of the algorithm of diff is that for families of pairs of sequences with $r \in \Theta(n^2)$ the running time is $\Theta(n^2 \log n)$ even if $q \in \Theta(n)$. There is at least one such family of files which occur in practice and for which diff behaves badly. These are files with many occurrences of one same line, say one fourth of the lines are blank. The easiest misbehavior of diff can be obtained by running it on sequences of the form $ab^n a$ and b^n [31].

Another shortcoming is that the computing time might depend on the order of specification of the sequences. Thus, computing the diff of $ab^{2n}a$ and b^n takes much longer than the diff of b^n and $ab^{2n}a$. The fact that the difference is not a symmetrical function does not justify this behavior because what dominates the running time is the computation of an lcs and an lcs does not depend on the order of the sequences.

Both these shortcomings disappear in an interesting variant discovered recently by Apostolico in [5]; see also [7]. This variant has time complexity $O((q+n) \log n)$ instead of $O((r+n) \log n)$. This is sufficient to guarantee an $O(n \log n)$ behavior, instead of $O(n^2 \log n)$, for files with only one frequent line, such as the example given above. However, the gain is obtained at the expense of complicated data structures, such as balanced binary search trees, and it is not clear whether the overhead of (always) using these structures is worth the time economy which is more accentuate only for special cases. Some experimentation might throw interesting light on this question.

A family which seems to defeat every known algorithm is given by pairs of random sequences over two letters. These seem to be the real “black sheeps” for sequence comparison; our luck is that they do not occur in practice very frequently. Or, do they? For instance, files that have many occurrences of two different lines in interlaced positions tend to behave as random sequences over two letters. These cases might arise in practice if we have blank lines with different indentations or two lines which occur frequently, such as the pairs **begin** and **end**.

Altogether, in spite of the excellence of diff, there seems to be ample space for a substantially better algorithm, if only it could be found! But we are hopeful that the proliferation of potentially equivalent quadratic algorithms is a sign that the ultimate word was not yet said.

Acknowledgements. The author thanks Christian Choffrut for inviting him to Rouen where this work began and Dominique Perrin for insisting that this survey be presented at the 1987 Ecole de Printemps at Ile d'Oléron. I also wish to thank Alair Pereira do Lago who helped with the programming tasks and with whom I maintained many interesting and helpful discussions on the subject of this paper.

References

[1] A. V. Aho, D. S. Hirschberg, and J. D. Ullman. Bounds on the complexity of the longest common subsequence problem. *J. ACM*, 23:1–12, 1976.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Pu. Co., Reading, MA, 1974.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Pu. Co., Reading, MA, 1983.

[4] L. Allison and T. I. Dix. A bit string longest common subsequence algorithm. *Inf. Process. Lett.*, 23:305–310, 1986.

[5] A. Apostolico. Improving the worst-case performance of the Hunt-Szymanski strategy for the longest common subsequence of two strings. *Inf. Process. Lett.*, 23:63–69, 1986.

[6] A. Apostolico. Remark on the Hsu-Du new algorithm for the longest common subsequence problem. *Inf. Process. Lett.*, 25:235–236, 1987.

[7] A. Apostolico and C. Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2:315–336, 1987.

[8] R. M. Baer and P. Brock. Natural sorting over permutation spaces. *Math. Comp.*, 22:385–410, 1968.

[9] V. Chvátal, D. A. Klarner, and D. E. Knuth. Selected combinatorial research problems. Technical Report STAN-CS-72-292, Computer Science Department, Stanford University, 1972.

[10] V. Chvátal and D. Sankoff. Longest common subsequences of random sequences. Technical Report STAN-CS-75-477, Computer Science Department, Stanford University, 1975.

[11] V. Chvátal and D. Sankoff. Longest common subsequences of two random sequences. *J. Appl. Prob.*, 12:306–315, 1975.

[12] M. Crochemore. Longest common factor of two words. In *Proceedings of CAAP'87, Pisa, Italy*, pages 26–36, 1987.

[13] J. Deken. Some limit results for longest common subsequences. *Discrete Math.*, 26:17–31, 1979.

[14] M. L. Fredman. On computing the length of longest increasing subsequences. *Discrete Math.*, 11:29–35, 1975.

[15] J. Gallant, D. Maier, and J. A. Storer. On finding minimal length superstrings. *J. Comput. Syst. Sci.*, 20:50–58, 1980.

[16] P. A. V. Hall and G. R. Dowling. Approximate string matching. *ACM Comput. Surv.*, 12:381–402, 1980.

[17] J. J. Hebrard. Distances sur les mots. Application à la recherche de motifs. Thèse de 3e cycle, Université de Haute-Normandie, 1984.

[18] P. Heckel. A technique for isolating differences between files. *Commun. ACM*, 21:264–268, 1978.

[19] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18:341–343, 1975.

[20] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24:664–675, 1977.

[21] D. S. Hirschberg. An information theoretic lower bound for the longest common subsequence problem. *Inf. Process. Lett.*, 7:40–41, 1978.

[22] D. S. Hirschberg and L. L. Larmore. The set lcs problem. *Algorithmica*, 2:91–95, 1987.

[23] W. J. Hsu and M. W. Du. New algorithms for the LCS problem. *J. Comput. Syst. Sci.*, 29:133–152, 1984.

[24] J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Technical Report #41, Computing Science, Bell Laboratories, 1976.

[25] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20:350–353, 1977.

[26] D. E. Knuth. *The Art of Computer Programming, Vol. 3, Sorting and Searching*. Addison-Wesley Pu. Co., Reading, MA, 1973.

[27] R. Lowrance and R. A. Wagner. An extension of the string-to-string correction problem. *J. ACM*, 22:177–183, 1975.

[28] D. Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25:322–336, 1977.

[29] W. J. Masek and M. S. Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20:18–31, 1980.

[30] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching.* Springer-Verlag, Berlin, 1984.

[31] W. Miller and E. W. Myers. A file comparison program. *Software - Practice and Experience*, 15:1025–1040, 1985.

[32] A. Mukhopadhyay. A fast algorithm for the longest common subsequence problem. *Information Sciences*, 20:69–82, 1980.

[33] N. Nakatsu, Y. Kambayashi, and S. Yajima. A longest common subsequence algorithm suitable for similar text strings. *Acta Inf.*, 18:171–179, 1982.

[34] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Molecular Biology*, 48:443–453, 1970.

[35] Y. Robert and M. Tchuente. A systolic array for the longest common subsequence problem. *Inf. Process. Lett.*, 21:191–198, 19885.

[36] D. Sankoff. Matching sequences under deletion/insertion constraints. *Proc. Nat. Acad. Sci. U.S.A.*, 69:4–6, 1972.

[37] D. Sankoff and J. B. Kruskal. *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison.* Addison-Wesley Pu. Co., Reading, MA, 1983.

[38] S. M. Selkow. The tree to tree editing problem. *Inf. Process. Lett.*, 6:184–186, 1977.

[39] P. H. Sellers. An algorithm for the distance between two finite sequences. *J. Comb. Th. A*, 16:253–258, 1974.

[40] P. H. Sellers. On the theory and computation of evolutionary distances. *SIAM J. Appl. Math.*, 26:787–793, 1974.

[41] P. H. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1:359–373, 1980.

[42] T. G. Szymanski. A special case of the maximal common subsequence problem. Technical Report TR-170, Computer Science Lab., Princeton University, 1975.

[43] W. F. Tichy. The string-to-string correction problem with block moves. *ACM Trans. Comput. Syst.*, 2:309–321, 1984.

[44] S. M. Ulam. Some combinatorial problems studied experimentally on computing machines. In S. K. Zaremba, editor, *Applications of Number Theory to Numerical Analysis*, pages 1–3, apa, 1972. Academic Press.

[45] T. K. Vintsyuk. Speech discrimination by dynamic programming. *Kibernetika*, 4:81–88, 1968.

[46] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21:168–173, 1974.

[47] C. K. Wong and A. K. Chandra. Bounds for the string editing problem. *J. ACM*, 23:13–16, 1976.