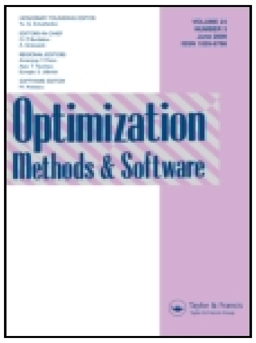

Algorithms for the longest common subsequence problem for multiple strings based on geometric maxima

Koji Hakata & Hiroshi Imai

Optimization Methods & Software, Vol. 10, pp. 233–260
Reprints available directly from the publisher
Photocopying permitted by license only

ALGORITHMS FOR THE LONGEST COMMON SUBSEQUENCE PROBLEM FOR MULTIPLE STRINGS BASED ON GEOMETRIC MAXIMA

KOJI HAKATA[a] and HIROSHI IMAI[b,*]

[a]*NEC, Japan;* [b]*Department of Information Science, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033, Japan*

Dedicated to Masao Iri on the occasion of his 65th birthday

(Received 19 May 1998; Revised 4 August 1998; In final form 24 August 1998)

Given two or more strings (for example, DNA and amino acid sequences), the longest common subsequence (LCS for short) problem is to determine the longest common subsequence obtained by deleting zero or more symbols from each string. The algorithms for computing an LCS between two strings were given by many papers, but there are few efficient algorithms for computing an LCS between more than two strings. This paper proposes a method for computing efficiently the LCS between three and more strings of small alphabet size, evaluates its theoretical time complexity, and estimates the computing time by computational experiments. Using this method, the LCS problem for eight strings of more than 120 length can be solved in about 40 min on a slow workstation.

Keywords: Longest common subsequence; dynamic programming; geometric maxima; computational geometry

1 INTRODUCTION

1.1 Background

This paper presents novel algorithms for the longest common subsequence (LCS for short) problem. There are both theoretical and practical reasons for considering this problem. The theoretical motivation is

* Corresponding author. Tel.: +81 3 3812 2111, ext. 4117. Fax:+81 3 5800 6933. E-mail: imai@is.s.u-tokyo.ac.jp.

to improve a bound on the complexity of this fundamental and useful problem, and practically the LCS problem is required to be solved because, although there is a demand for solving the LCS problem for many strings particularly in molecular biology, few efficient algorithms are known for this problem. Roughly speaking, an LCS is the largest common part among several strings. The formal definition of the LCS problem is described in Section 2. The LCS problem has been solved by using dynamic programming [13,26,30]. However, the dynamic programming algorithm is very time-and-space consuming for use with very large sequence databases, so an efficient and practical algorithm is needed for the LCS problem in which the number of input strings d is large.

The LCS problem is a common task in sequence analysis of DNA sequences and amino acid sequences of proteins, and has applications to genetics and molecular biology [27,29]. In such applications, the number of characters s is constant independent of the length of each string n. In fact, $s=4$ and 20 for DNA sequences and amino acid sequences, respectively. Also, in such applications, the LCS problem for many strings, i.e. for large d, arises, and it has been requested to devise an efficient algorithm for the problem. This problem also finds applications in computer science. For example, an LCS algorithm can compare files and detect their differences by finding what they have in common. It can be used to correct spelling errors and to compare sequences for similarity.

This paper presents an efficient algorithm for the LCS problem when the number of characters s is constant as in the above cases and the number of strings d is more than two. It can solve the LCS problem for larger number and longer length of strings, which could not be solved by the dynamic programming algorithm because of lack of time and memory. The LCS problem for d strings is referred as the d-LCS problem.

The 2-LCS problem was first studied by the molecular biologists [9,10], who found use of LCS in studying similar sequences of amino acids. Subsequently, together with other string problems, the complexity of the 2-LCS problem was analyzed by many computer scientists. Other cited applications include data compression [1,16], syntactic pattern recognition [11,22,23], where an LCS of the strings is used to denote a maximal common substructure of the objects represented by the strings.

1.2 Previous Works

We first explain the dynamic programming approach to solving the LCS problem, which is a basis of most algorithms for the problem. The basic idea of the dynamic programming method is to evaluate successively the distance between longer and longer prefixed strings until the final result is obtained. These partial results are computed iteratively and are entered in an $(n+1)^d$ array, leading to $O(n^d)$ time and space complexities.

For $d=2$, the input of the LCS problem is two strings A and B. For string $A=a_1a_2\cdots a_n$, $A[p,\ldots,q]$ is $a_pa_{p+1}\cdots a_q$. If we define $L[i,j]$ as the length of an LCS of $A[1,\ldots,i]$ and $B[1,\ldots,j]$, then $L[i,0]=L[0,j]=L[0,0]=0$ and for $1\le i,\ j\le n$

$$L[i,j]=\begin{cases} L[i-1,j-1]+1 & \text{if } A[i]=B[j],\\ \max\{L[i-1,\ j],\ L[i,j-1]\} & \text{otherwise.}\end{cases}$$

The boundary condition simply means that the length of LCS between any string and a null string is zero.

Wagner and Fischer [30] employ an array $U[0\ldots m,0\ldots n]$ such that the length of LCS of $A[1,\ldots,i]$ and $B[1,\ldots,j]$, i.e. $L[i,j]$, is kept in $U[i,j]$. With column 0 and row 0 of this array set to 0 initially, $U[i,j]$ are computed row by row according to the above recursion. The desired value is in $U[m,n]$. In their method, an LCS itself may then be derived in a straightforward manner from the completed distance array via a structure known as a trace. Clearly, the time complexity of their algorithm is $O(n^2)$. Hirschberg [13] modified Wagner and Fischer's technique, preserving the $O(n^2)$ time complexity but reducing the space requirement from a quadratic to a linear characteristic. Hunt and McIlory [16] implemented Hirschberg's LCS algorithm for the unix `diff` command. This was found to work well for short inputs, but its quadratic time complexity resulted in significant degradation in execution speed as the lengths of the input files were increased. However, McIlory later obtained a dramatic improvement in performance by employing the LCS algorithm developed by Hunt and Szymanski [17], which is explained later.

For $d=2$, various improvements to the simple $O(n^2)$ algorithm were made, although under plausible assumptions, the worst-case complexity of an LCS algorithm on strings over an infinite alphabet (or sufficiently large s) must be $O(n^2)$ time [1]. Masek and Paterson [24] gave an $O(n^2/\log n)$ algorithm for strings over a finite alphabet with modest restrictions on distance functions. However, this is faster than simple algorithms only for strings longer than about 200 000 characters. Therefore, another approach is needed to achieve the sub-quadratic time on the average. Many algorithms concentrate on efficient generation of matches or dominant matches (dominants), which are formally defined in Section 2.

Hirschberg's algorithm [14] repeatedly scans one of the input strings A, and generates dominants of each contour after each scan of A. Since each scan takes n steps in order to look for matching symbols in the other input string B and there are l contours where l is the length of an LCS, the total time complexity of the algorithm is no more than $O(n \log s + ln)$ where $n \log s$ is the preprocessing time.

Hunt and Szymanski's approach [17] to extracting an LCS from two strings is equivalent to determining the longest monotonically increasing path in the grid graph composed of nodes (i,j) such that $A[i] = B[j]$, which are called matches. Whereas previous methods required quadratic time in all cases, their algorithm requires $O((R + n) \log n)$ time, where R is the total number of ordered pairs of positions at which the two strings match, i.e. the number of matches. In the worst case, every element of a string A could match every symbol in a string B, resulting in n^2 such ordered pairs and a complexity of $O(n^2 \log n)$. For many applications, however, such as the file difference problem where each line of a file is taken as an atomic symbol, R can be expected to be close to n, giving a performance of $O(n \log n)$, a significant improvement over the quadratic time procedures.

Apostolico and Guerra [2] have analyzed both the Hirschberg's and the Hunt–Szymanski's methods of finding a longest common subsequence, and have devised cognate algorithms with certain improvements. For case where n_1 (the length of one of input strings) $\ll n_2$ (the length of the other input string), their version of Hirschberg's algorithm will be faster than the original, as their version has a time complexity of $O(n_1^2 \cdot \min\{\log s, \log n_1, \log(2n_2/n_1)\})$. As noted above, Hunt and Szymanski's algorithm becomes worse than quadratic in the pessimal case. However, Apostolico and Guerra have limited the worst-case behavior to a quadratic characteristic. The actual time bound of their revised version of the algorithm is $O(n_1 \log n_2 + D \log(n_1 n_2 / D))$, where D is the number of dominant matches between the two strings. Intuitively, the dominant matches are a layered set of geometric maxima among matches in the grid graph, and its number D is much smaller than the number R of matches in many cases. A rigorous definition is given in Section 2.

Chin and Poon [7] proposed to compute dominants in each contour from dominants with a lower value, instead of scanning elements of A and B one by one. Since each dominant can generate at most s dominants of the next contour, no more than $O(Ds)$ time will be required. Their algorithm takes $O(ns + \min(Ds, ln))$ time and $O(ns + D)$ space.

For $d = 3$, the dynamic programming strategy yields an $O(n^3)$ algorithm which computes the three-dimensional dynamic programming table L row by row. Some of the algorithms for two strings may be extended to this case, but the time complexity becomes higher than in the two-dimensional case. There have been few efficient algorithms for three or more strings which outperform the naïve dynamic programming algorithm.

It has been shown in [12,23] that the problem of determining the length of an LCS l for d input strings is NP-complete for variable-length strings and unbounded d, no matter whether the size of the alphabet is bounded or not. Further refined complexity analysis as well as an approximation algorithm is given in [6,8,20]. Therefore, it is very probable that every solution to the d-LCS problem in the worst case takes an amount of execution time which is an exponential function of the input size taking the lengths of the strings n and the number of strings d as parameters. Therefore the LCS problem for many strings is a very difficult problem, and the extraction of a d-LCS for such purposes as data compression is not viable, as any means for so doing will inevitably be intractable. However, for almost every conceivable application, it is also reasonable to assume that the number of the strings under consideration is bounded by some constant. In this sense, the straightforward generalized dynamic programming algorithm for solving the d-LCS problem already has a polynomial time (and space) complexity $O(n^d)$ as mentioned above, since d is now considered as a constant, although the complexity is unsatisfactory for practical use.

Hsu and Du [15] generalized the 2-string LCS problem to finding a longest common subsequence for a set of strings. Assuming a finite symbol set, they showed that their scheme requires a preprocessing time that grows linearly with the total length of input strings and a processing time that grows linearly with the number of strings d and the number of matches among them R. They achieved it by the formulation of the general LCS problem as identifying a longest path in some acyclic directed graph. The time complexity of their algorithm is $O(nsd + Rsd)$, and the space complexity is $O(nsd + R)$.

Baeza-Yates [5] has proposed a method, for cases where it is required to solve the d-LCS problem for constant d and strings of length n, which still requires $O(n^d)$ worst-case time and space, but has a more favorable average-case behavior. His approach is to build a partial deterministic finite automaton, known as a directed acyclic subsequence graph (DASG), which can recognize all the subsequences of the strings in the set $\{A_1, A_2, \ldots, A_d\}$. As the graph is being constructed, its edges are labeled with the number of strings sharing that transition. The LCSs may then be recovered by searching the graph for maximal sequences of edges labeled with d.

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

Irving and Fraser [19] presented two algorithms for the 3-string problem which are effective when the length l of the longest common subsequence is relatively large or very small. These algorithms can be generalized to the general d-LCS problem. The time complexities are $O(dn(n-l)^{d-1})$ and $O(dl(n-l)^{d-1} + dsn)$ for the d-LCS. It is shown via preliminary computational experiments that their algorithm is more efficient than Hsu and Du's algorithm. It is also claimed that the space complexity becomes a main bottleneck to solve large-size LCS problems for many strings, and hence a need for algorithms that use less space was pointed out.

1.3 Our Contributions

As pointed out by Aho *et al.* [1], the straightforward matrix-filling approach for the LCS problem has the disadvantage that it takes the same large amount of time on all input strings with the same lengths, since the algorithm makes no use of the inherent structures of the strings. Therefore, we consider an algorithm which makes use of the structures of the input strings. It is efficient on almost inputs. Moreover, although there are several algorithms [5,15] whose time complexity is linear in R for a constant $d \geq 3$ and a fixed alphabet, an algorithm [7] which runs in time linear in D is known only when the number of input strings d is equal to two. Hence, we consider an algorithm which takes time linear in D for $d = 3$.

In this paper, we present an efficient algorithm that, for three strings and small alphabet size, runs in time $O(ns + Ds \log s)$, where D is again the number of dominants between the three strings and $O(ns)$ is the preprocessing time. Whenever the dominant matches among the strings are scarce ($D \ll n^d$), as is the usual case, our method is very efficient in terms of both time and space. This algorithm is more efficient when the alphabet size s is small, which is the case in the applications mentioned in Section 1.1.

Furthermore, our algorithm can be generalized to the case for d strings ($d \geq 3$), and then the time and space complexities become $O(nsd + Dsd((\log^{d-3} n + \log^{d-2} s))$ and $O(nsd + D)$, respectively. The running time depends on the number of dominants D, and, since, roughly speaking, D becomes relatively small to n^d when d becomes large, our algorithm remains efficient for moderate size of d both in time and space.

The approach of our algorithm is as follows: since the d-dimensional dynamic programming table L is nondecreasing in every argument, we can draw contours on L to separate regions of different values. The entire matrix is specified by its contours (there are l contours), and the contours can be completely specified by their corner points, which are called *dominants* or dominant matches. Since only dominant matched symbols will constitute an LCS, an LCS can be obtained by finding the set of dominants, instead of filling all the $(n+1)^d$ entries in L.

Several algorithms for the 2-LCS problem compute some of the representation of the set of dominants. The worst-case time bound is lowered down to $O(ln)$ by Hirschberg's algorithm [14]. Unfortunately, one finds difficulties in generalizing the above algorithms to the general d-LCS problem, because these algorithms depend on certain properties unique to the special 2-string cases. They define easily the points (or contours) which have the same value on the dynamic programming table L row-by-row or column-by-column. But, when the LCS problem is generalized to higher dimensions, this contour-defining approach requires much work and is very likely to yield poor time performance and induce large space demand. Therefore a new approach which differs from those mentioned above is needed.

The basic idea of our algorithm is based on the algorithm of Chin and Poon, but more properties of dominant matches are utilized to enhance the efficiency of the algorithm. Our algorithm expands the candidates of next dominants which are few when the alphabet size is small, and then eliminates redundant matches from the candidates. The analysis and experiments show that the algorithm performs better than the simple dynamic programming.

Our extensions are based on computational-geometric results on geometric maxima. Computational geometry has been applied to various fields as in [18]. This paper presents its new application to a string problem, which looks non-geometric at a glance.

The paper proceeds as follows. Section 2 gives the definitions and the theorems for the algorithm. Section 3 gives the algorithm, and Section 4 gives the analysis of the algorithm. Section 5 mentions the higher-dimensional case. Section 6 presents the result of computational experiments. Section 7 concludes the paper.

2 PRELIMINARIES

The longest common subsequence problem is stated as follows: A string is a sequence of symbols (or characters). The set of all possible symbols is called the alphabet, denoted Σ. We assume that the alphabet is of bounded size s, that is, $|\Sigma| = s$. A *subsequence* of a string is obtained by deleting zero or more (not necessarily consecutive) symbols from the string. For example, "abdad" is a string, and "bad" is a subsequence of it. A *common subsequence* (CS in short) of a set of strings is a subsequence of all the strings. A *longest common subsequence* (LCS in short) is a subsequence with the maximal length. For example, "ad" is a CS of "abdcd" and "cacbd", while "abd" is the LCS of the two strings. Notice that a set of strings may possess more than one LCS in general. For example, both "abd" and "acd" are LCSs of "abdcd" and "cacbd".

The d-LCS problem is to find a single LCS for d arbitrary input strings. Formally, the d-LCS problem is as follows: Let $A_1, A_2, \ldots, A_d$ be d strings of length $n_1, n_2, \ldots, n_d$ on an alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_s\}$ of size s. Throughout this paper, we assume $n_1 = n_2 = \cdots = n_d = n$ only for convenience. A *subsequence* of A_i can be obtained by deleting zero or more symbols from A_i. String B is a *common subsequence* of $A_1, A_2, \ldots, A_d$ iff B is a subsequence of each A_i, $i = 1, 2, \ldots, d$. The *longest common subsequence* problem is to find a common subsequence of $A_1, A_2, \ldots, A_d$ of maximal length. In the d-string dynamic programming approach, we define an L-matrix for the d strings $A_1, A_2, \ldots A_d$ as an integer array $L[0 \ldots n_1, \ldots, 0 \ldots n_d]$ such that $L[p_1, p_2, \ldots, p_d]$ is the length of an LCS for $A_i[1, \ldots, p_i]$, $i = 1, 2, \ldots, d$.

Denote the ith coordinate of the position p in L by p_i. p is a *match* iff $A_1[p_1] = A_2[p_2] = \cdots = A_d[p_d]$, that is, a match of the symbols at some positions of the strings. For $d = 2$, a point (i, j) is called a match iff $A[i] = B[j]$. Let R be the total number of matches. A point p *dominates* a point q if $p_i \leq q_i$ for $i = 1, 2, \ldots, d$ (denoted by $p \preceq q$). Dominance relation $\preceq$ is a partial ordering for $d > 1$ defined on the set of all (dominant) matched points. Hsu and Du [15] represented the relation by an acyclic digraph having all arcs of length 1, and found an LCS corresponding to a maximal length path on the graph in time proportional to the number of all matched points R.

It is clear that a CS of $A_1, A_2, \ldots, A_d$ corresponds to a chain of dominance relations, and an LCS of the strings corresponds to a longest such chain. For a point set S, by $p \preceq S$ we mean that there is a point $q \in S$ such that $p \preceq q$. If $p_i < q_i$ for $i = 1, 2, \ldots, d$, we write $p \prec q$. In this case, a point q is said to be a *successor* of another point p. Alternatively, we say that p is a *predecessor* of q. A match p is a *dominant* or *k-dominant* iff $L[p] = L[p_1, p_2, \ldots, p_d] = k$ and $L[q] < k$ for all q such that $[0, 0, \ldots, 0] \preceq q \preceq p$ and $q \neq p$, that is, the dominants on the contour with value k. Let D be the total number of dominants. Since a point p is a dominant only if p is a match, $D \leq R$. Points p and q are *independent* iff $p \not\succeq q$ and $p \not\preceq q$. If p and q are k-dominants, then p and q are independent of each other.

LEMMA 1 *A match q is a $(k+1)$-dominant iff there is a k-dominant p with $p \prec q$, and there is no match r such that $p \prec r \preceq q$, $r \neq q$ for any k-dominant p with $p \prec q$.*

For convenience, if a successor of a point p corresponds to a symbol σ, we call it a *σ-successor* of p. Also, for any σ, we define ∞ (in the implementation it is represented by $n+1$) to be a σ-successor of any other point, while ∞ has no successor itself. Let $p_i(\sigma)$ be the position of the first σ in $A_i[p_{i+1} \dots n]$. For $p=[p_1, \dots, p_d]$, let $p(\sigma)$ be $[p_1(\sigma), p_2(\sigma), \dots, p_d(\sigma)]$, and we call $p(\sigma)$ the immediate σ-successor of a point p. $p(\Sigma)$ is $\{p(\sigma) \mid \sigma \in \Sigma\}$. Now, if the points q is the immediate σ-successor and q' is any non-immediate σ-successor to a point p, then q' can be ruled out (or dominated) by the immediate σ-successor q, because $q_i \leq q'_i$ for $1 \leq i \leq d$. Hence, out of all σ-successors of p, we need consider only the one (immediate) σ-successor which is not ruled out by any other σ-successor of p. By the above definition, the immediate σ-successor of a point p represents the next σ that is closest to p_j in string A_j for $1 \leq j \leq d$, whereas ∞ simply denotes that no such σ exists. In Section 3, we shall discuss an efficient method for generating an immediate successor for any point under consideration.

Define D^k as the set of k-dominants, that is, all the dominants on the contour with value k. The dominant is composed of l disjoint subsets $D^1, D^2, \dots, D^l$. Define $D^k(\sigma)$ as $\{p(\sigma) \mid p \in D^k\}$ and $D^k(\Sigma)$ as $\{p(\Sigma) \mid p \in D^k,\ \sigma \in \Sigma\}$. $D^k(\Sigma)$ is the set of candidates of $(k+1)$-dominants.

A point p in a set S is a *maximal* element of S, if $p \not\preceq S - \{p\}$. Given a set S of n points, the *maxima* problem consists of finding all the maximal elements of S. A *minimal* element and the *minima* problem are also defined similarly. The simplest maxima algorithm is that we pick out a point p from a set S one by one, and for each point p we examine whether $p \preceq S - \{p\}$ or not. This algorithm clearly takes $O(n^2)$ time where n is the number of points in S, because the check of $p \preceq S - \{p\}$ can be done in $O(n)$ time.

Efficient algorithms for computing the maxima are given in [21,25] as stated below.

THEOREM 1 Ref. [21,25] *For $d=2$, the maxima problem requires time $\Omega(n \log n)$. However, if S is already sorted, it can be solved in $O(n)$ time.*

In the case of $d=3$, using a proper data structure, we can check whether $p \not\preceq S - \{p\}$ in $O(\log n)$ time, and then we have the following theorem.

THEOREM 2 Ref. [21,25] *For $d=3$, the maxima problem can be solved in $O(n \log n)$ time.*

3 ALGORITHM FOR THREE STRINGS

3.1 Basic Properties

We now provide nice properties of dominants, which will be used in developing our efficient algorithm to LCS.

LEMMA 2 *For any match p, $p \prec p(\sigma)$, and there is no match r of σ such that $p \prec r \prec p(\sigma)$.*

Proof This lemma is evident from the definition of $p(\sigma)$, which is the immediate σ-successor of a point p.

LEMMA 3 *For any i, if $p_i < q_i$ then $p_i(\sigma) \leq q_i(\sigma)$.*

Proof Since $p_i < q_i$ and there is no match of σ between p_i and $p_i(\sigma)$, if $q_i < p_i(\sigma)$, there is also no match of σ between q_i and $p_i(\sigma)$, and hence $p_i(\sigma)$ is the ith coordinate of the immediate σ-successor of point p, that is, $p_i(\sigma) = q_i(\sigma)$. Otherwise, $p_i(\sigma) \leq q_i \leq q_i(\sigma)$.

THEOREM 3 *If D^k is sorted with respect to a coordinate, then for any $\sigma \in \Sigma$, $p(\sigma)$'s for $p \in D^k$ in the sorted order are again sorted in respect to the same coordinate.*

Proof This theorem is evident from Lemma 3. For points p and q in D^k with $p_i < q_i$, points $p(\sigma)$ and $q(\sigma)$ in $D^k(\sigma)$ have the property that $p_i(\sigma) \leq q_i(\sigma)$, so the order is preserved.

THEOREM 4 *For a pair of k-dominants p and q, if $p(\sigma)$ and $q(\sigma)$ are not independent, $p_i(\sigma) = q_i(\sigma)$ holds for some $i = 1, 2, \ldots, d$.*

Proof If $p_1 = q_1$, trivially $p_1(\sigma) = q_1(\sigma)$. Otherwise, without loss of generality, we may assume $p_1 < q_1$. Then, since $p_1 < q_1$ and the k-dominants are independent, for some $i \in \{2, \ldots, d\}$, $p_i > q_i$ holds. If $p_j > q_j$ does not hold for any j, then $p \preceq q$ and it leads to the contradiction to the fact that k-dominants p and q are independent, that is, $p \not\preceq q$. For this i, from Lemma 3 $p_i(\sigma) \geq q_i(\sigma)$. If $p(\sigma) \preceq q(\sigma)$, $p_i(\sigma) \leq q_i(\sigma)$ and hence $p_i(\sigma) = q_i(\sigma)$. If $p(\sigma) \succeq q(\sigma)$ then $p_1(\sigma) = q_1(\sigma)$, since $p_1(\sigma) \leq q_1(\sigma)$ from $p_1 < q_1$.

THEOREM 5 *Assume p, q are k-dominants. If $p(\sigma_i) \prec q(\sigma_j)$ for $i \neq j$, then $p(\sigma_j) \preceq q(\sigma_j)$.*

Proof Since $p \prec p(\sigma_i) \prec q(\sigma_j)$, $q(\sigma_j)$ is a match of σ_j in $S = \{r \mid p \prec r \preceq [n_1, n_2, \ldots, n_d]\}$, that is, $q(\sigma_j)$ is a σ_j-successor of p. Since $p(\sigma_j)$ is the immediate σ_j-successor, and hence $p(\sigma_j) \preceq q(\sigma_j)$.

THEOREM 6 *D^{k+1} is the minima of $D^k(\Sigma)$.*

Proof First we prove that D^{k+1} is a subset of $D^k(\Sigma)$. Assume that a match q is not in $D^k(\Sigma)$. Let σ be the character of q, i.e. $A_1[q_1]$. If q is a $(k+1)$-dominant, from Lemma 1, then there must exist $p \in D^k$ such that $p \prec q$. Then, for this p, from Lemma 3, $p(\sigma) \preceq q$ holds. Since $p(\sigma) \in D^k(\Sigma)$ and $q \notin D^k(\Sigma)$, $p(\sigma) \neq q$. Then, again from Lemma 1, q cannot be a $(k+1)$-dominant. Thus $D^{k+1} \subset D^k(\Sigma)$.

If $q \in D^k(\Sigma)$ is not the minima of $D^k(\Sigma)$, there is $r \in D^k(\Sigma)$ such that $p \prec r \preceq q, r \neq q$ for a point $p \in D^k$. From Lemma 1, q can not be a $(k+1)$-dominant.

This theorem is a main theorem in this paper, because all our algorithms are based on the approach in which a number of the minima problem are solved many times repeatedly. From now on, we only use the term 'minima', although the term 'maxima' is commonly used in computational geometry. Theorems 3–5 are needed to make our algorithm efficient when the number of strings d is more than two.

3.2 Basic Ideas

We present the algorithm A in section 3.3 which obtains an LCS of three input strings A_1, A_2 and A_3 in time $O(ns + Ds \log s)$ and space $O(ns + D)$. This section describes its basic ideas. The principle of the algorithm is to compute repeatedly the minima of D^k, as proved in Theorem 6. If we use simply Theorem 2 to obtain the minima of $D^k(\Sigma)$, $O(\,|\,D^k(\Sigma)\,|\log|\,D^k(\Sigma)\,|\,)$ time is needed and the linearity of D in the time complexity cannot be accomplished. To make the time complexity linear in D, we have to make full use of properties derived in the previous section.

The algorithm is based on three ideas. The first idea is due to Theorem 5. If $p(\sigma_i) \prec q(\sigma_j)$, $q(\sigma_j)$ cannot be a k-dominant. In this case, by Theorem 5, $p(\sigma_j) \preceq q(\sigma_j)$ and hence $q(\sigma_j)$ can be excluded from the candidates of $(k+1)$-dominants, i.e. $D^k(\Sigma)$, simply by checking the dominance relations among $D^k(\sigma_j)$, which contains both $p(\sigma_j)$ and $q(\sigma_j)$. That is, after we examined the dominance relations between $p(\Sigma)$ whose size is equal to the alphabet size s for each point $p \in D^k$, we have only to compute the minima of each $D^k(\sigma_j)$, not the whole $D^k(\Sigma)$.

The second idea is the use of Theorem 3. Every $D^k(\sigma)$ is already sorted in the same order as D^k in some coordinate, since $D^k(\sigma)$ is constructed from each point of D^k in order, and then by Theorem 1 the two-dimensional minima problem concerning $D^k(\sigma)$ can be solved in linear time $O(|D^k(\sigma)|)$. So if we can reduce the three-dimensional minima problem into the two-dimensional minima problem, we can achieve the time complexity linear in D. When we construct D^{k+1} from all $D^k(\sigma)$'s, we use the merge sort for sorting D^{k+1} in some coordinate, because each $D^k(\sigma)$ is already sorted in the same coordinate and the number of times of merge operations is less than the alphabet size s.

The third idea is as follows. For a point p in the d-dimensional space, define a point $p[i]$ in the $(d-1)$-dimensional space to be $(p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_d)$. Also, define $D^k(\sigma)[i,j]$ to be $\{p[i] \mid p \in D^k(\sigma), p_i = j\}$. From Theorem 4, if two points p and q whose symbol is σ are not independent, then for at least one coordinate i both p and q have the same position, say j, and then two points p and q are contained in $D^k(\sigma)[i,j]$. Therefore, we have only to compute the two-dimensional minima of $D^k(\sigma)[i,j]$ for $\sigma \in \Sigma$, $1 \leq i \leq 3$ and $1 \leq j \leq n$, instead of the three-dimensional minima of $D^k(\sigma)$. Since these sets can be sorted in linear time according to the coordinate x_1 or x_2 by virtue of the second idea, the computation for the two-dimensional minima can be performed in $O(|D^k(\sigma)[i,j]|)$ by Theorem 1.

3.3 Implementation

We now describe the algorithm. In the preprocessing stage, we construct a data structure $\mathrm{T}[\sigma_1 \ldots \sigma_s, 0 \ldots n, 1 \ldots d]$ to enumerate $p(\Sigma)$ efficiently for any given point p. $\mathrm{T}[\sigma, i, j]$ specifies the position of the first σ in $A_j[i+1, \ldots, n]$, that is, the third parameter j of T table specifies a string and the second parameter i specifies the position of this string. Therefore $\mathrm{T}[\sigma, p_i, i]$ stores $p_i(\sigma)$. If such a σ does not exist, $\mathrm{T}[\sigma, i, j] = n+1$. The procedure which computes the T table is as follows [7,15].

```
1.  for $1 \leq i \leq d$ do
      for $\sigma \in \Sigma$ do $\mathrm{T}[\sigma, n, i] := n+1$
      for $j := n-1$ to 0 do
          for $\sigma \in \Sigma$ do $\mathrm{T}[\sigma, j, i] := \mathrm{T}[\sigma, j+1, i]$
          $\mathrm{T}[A_i[j+1], j, i] := j+1$
```

See the following example. Recall that the ith position of a matched point corresponds to a position in string A_i, and $p(\sigma)$ is given by $q = (q_1, q_2, \ldots, q_d)$ where $q_j = \min\{i(>p_j) \mid A_j[i] = \sigma\}$. For strings $A_1 =$ "abcdbb" and $A_2 =$ "cbacbaa", a point $p = (3, 1)$ is a match point of 'c'. Since T['b', 3, 1] = 5 and T['b', 1, 2] = 2, we can find that a point (5, 2) is the p('b'). Observe that for a given position in a string, we need to know the next higher position that contains σ. Therefore, to avoid the searches on the strings whenever $p(\sigma)$ is needed, we rather prepare such next higher σ-positions for each position in a string.

The Algorithm A needs $O(ns)$ space for T table and $O(D)$ space for storing the position of dominants, and the parent pointer and the ordering number according to the coordinate x_2. If the symbols are not known beforehand, then with the use of balanced search tree techniques, the work of the construction of T table requires at most $O(n \log s)$ time, as it is reasonable to assume that the representation of the symbols are linearly ordered so that useful operations like test for ordering ($a \geq b$?) may be used, rather than test for mere equality ($a = b$?). It needs also a temporary working area of $O(D^k)$ size storing $D^k(\Sigma)$. The Algorithm A is as follows, where $M(S, d)$ is an algorithm which returns minima of points in S in the d-dimensional space.

```
ALGORITHM A($n, s, \{A_1, A_2, A_3\}$)
1.  compute the T table; $D^0 := \{[0,0,0]\}$; $k := 0$
2.  **while** $D^k$ not empty **do**
      $A := B := \phi$
2.1 **for** $p \in D^k$ **do**
        the parents of $p(\Sigma)$ are set to $p$
        $A := A \cup \mathrm{M}(p(\Sigma), 3)$
2.2 **for** $\sigma \in \Sigma$ **do**
        sort $D^k(\sigma)$ concerning the coordinate $x_1$
        sort $D^k(\sigma)$ concerning the coordinate $x_2$
        **for** $1 \leq i \leq 3$ **do**
          sort $D^k(\sigma)[i, j]$ for all $j \in \{q_i \mid q \in D^k(\sigma)\}$
            concerning the coordinate $x_1$ when $i = 2, 3$ and $x_2$ when $i = 1$
          **for** $j \in \{q_i \mid q \in D^k(\sigma)\}$ **do** $B := B \cup \mathrm{M}(D^k(\sigma)[i, j], 2)$
2.3 merge sort $\{D^k(\sigma) \mid \sigma \in \Sigma\}$ into $D^k(\Sigma) = \{D^k(\sigma) \mid \sigma \in \Sigma\}$ concerning $x_1$
    merge sort $\{D^k(\sigma) \mid \sigma \in \Sigma\}$ concerning $x_2$
2.4 $D^{k+1} :=$ extract $A \cap B$ from $D^k(\Sigma)$ keeping these orderings
2.5 $k := k + 1$
    **end-while**
```

```
3. pick a point $p$ in $D^{k-1}$
   while $k-1>0$ do
     output $p$ and set $p$ to the parent of $p$ by the parent pointer
     $k:=k-1$
   end-while
```

4 ANALYSIS

We first prove the validity of the algorithm in Section 4.1. Secondly we analyze its time and space complexities in Section 4.2.

4.1 Validity

THEOREM 7 *The algorithm A correctly computes the LCS of strings A_1, A_2 and A_3.*

Proof By Theorem 6, D^{k+1} is a subset of $D^k(\Sigma)$, which is computed in Steps 2.2 and 2.3. The remaining task is to exclude points from $D^k(\Sigma)$ which are dominated by other points in $D^k(\Sigma)$.

Dominance relations among points $p(\sigma_i)$ $(1 \le i \le s)$ in $D^k(\Sigma)$ for each $p \in D^{k-1}$ are all checked in Step 2.1. And then, we check dominance relations among $p(\sigma)$ $(p \in D^{k-1})$ for each σ in Step 2.2. By Theorem 4, if $p(\sigma)$ and $q(\sigma)$ are not independent, there is i such that $p_i(\sigma) = q_i(\sigma)$. Then $p[i], q[i] \in D^k(\sigma)[i, p_i(\sigma)] (= D^k(\sigma)[i, q_i(\sigma)])$, and dominance relations between $p[i]$ and $q[i]$ (and accordingly $p(\sigma)$ and $q(\sigma)$) can be checked in $\mathrm{M}(D^k(\sigma)[i, p_i(\sigma)], 2)$. By Theorem 5, we do not have to check dominance relations between $p(\sigma_i)$ and $q(\sigma_j)$ for $p \neq q \in D^{k-1}$ and $i \neq j$ in order to exclude dominated points from $D^k(\Sigma)$, because when $p(\sigma_i) \prec q(\sigma_j)$, $p(\sigma_j) \preceq q(\sigma_j)$ holds and we have already checked the relation between $p(\sigma_j)$ and $q(\sigma_j)$ in Step 2.2 as described above. The theorem follows.

4.2 Complexities

LEMMA 4 *If the sorted orders of D^k with respect to both x_1 and x_2 are at hand, we can sort $D^k(\sigma)$ in both x_1* and *x_2 in $O(|D^k|)$ time for each σ, and sort D^{k+1} in both x_1 and x_2 in time $O(|D^k| s \log s)$.*

Proof For the former, simply use Theorem 3, that is, if D^k is sorted in respect to x_1, $D^k(\sigma)$ is again sorted in respect to x_1 when each point of $D^k(\sigma)$ is computed from D^k in order, and it holds also for x_2. For the latter, we sort D^{k+1} with respect to x_1 and x_2 by merging s sorted lists of $D^k(\sigma)$ $(\sigma \in \Sigma)$. Each level of merges requires $O(|D^k|s)$ time, and the level of merges is $\lceil \log s \rceil$. Then the whole time needed to sort D^{k+1} is $O(|D^k|s \log s)$.

LEMMA 5 *If the sorted orders of $D^k(\sigma)$ in both x_1 and x_2 are given, all $D^k(\sigma)[i,j]$ for $j \in \{q_i \mid q \in D^k(\sigma)\}$ can be computed in the sorted order of x_1 for $i = 2, 3$ and in the sorted order of x_2 for $i = 1$ in $O(|D^k(\sigma)|)$ time. Further, $M(D^k(\sigma)[i,j], 2)$ can then be solved in $O(|D^k(\sigma)[i,j]|)$ time.*

Proof We adopt the following simple algorithm for each i to compute the sorted list of $D^k(\sigma)[i,j]$: Initially, an empty list is made to correspond to each $D^k(\sigma)[i,j]$ for each $j \in \{q_i \mid q \in D^k(\sigma)\}$; we then enumerate points p of $D^k(\sigma)$ in the sorted order of x_1 when $i = 2, 3$ and x_2 when $i = 1$ with adding $p[i]$ to the list of $D^k(\sigma)[i, p_i]$ in the same order. This obviously produces sorted lists for all $D^k(\sigma)[i,j]$ and takes $O(D^k(\sigma))$ time in total.

Then, in solving the two-dimensional minima problem $\mathrm{M}(D^k(\sigma)[i,j], 2)$, points in $D^k(\sigma)[i,j]$ are sorted in either x_1 or x_2, and hence applying Theorem 1 we obtain the lemma.

THEOREM 8 *The algorithm A requires time of $O(ns + Ds \log s)$, where n is the length of strings A_1, A_2 and A_3 and s is the number of different symbols that appear in strings A_1, A_2 and A_3 and D is the number of dominant matches, assuming that symbols can be compared in one time unit.*

Proof Step 1 is the preprocessing step that builds the T table, which takes time $O(ns)$. In Step 2, the outer loop is repeated l times. Step 2.1 loops $|D^k|$ times. Therefore, there should be $|D^1| + |D^2| + \cdots + |D^l| = D$ executions to compute $A := A \cup \mathrm{M}(p(\Sigma), 3)$. By Theorem 2 and $|p(\Sigma)| = s$, the computation of $\mathrm{M}(p(\Sigma), 3)$ requires $O(s \log s)$ time and then Step 2.1 requires $O(Ds \log s)$ time. By Lemma 5, the innermost loop of Step 2.2 takes time proportional to $\sum |D^k(\sigma)[i,j]| = |D^k(\sigma)| = |D^k|$. By the first half of Lemma 4, the other part of Step 2.2 requires $O(|D^k|s)$ time. In total, Step 2.2 takes $O(|D^k|s)$ time for each iteration and the whole work is $O(Ds)$ time. By the latter half of Lemma 4, Step 2.3 takes $O(Ds \log s)$ time over the whole algorithm. Step 2.4 requires $O(|D^k|s)$ time. After all, Step 2 takes $O(Ds \log s)$ time. Step 3 takes $O(l)$ time, which is less than $O(ns)$.

THEOREM 9 *The algorithm A requires space of $O(ns + D)$, where n is the length of strings A_1, A_2 and A_3 and s is the number of different symbols that appear in strings A_1, A_2 and A_3 and D is the number of dominant matches.*

Proof Step 1 is the preprocessing step that builds the T table of size $O(ns)$. Step 2 needs $O(D^k)$ for storing each $D^k(\sigma)$. This value is less than $O(D)$. Therefore the Algorithm A requires space which is described in the theorem.

5 EXTENSIONS TO THE CASE OF MORE THAN THREE STRINGS

Our Algorithm A for the LCS problem among three strings can be generalized to the problem for d strings ($d > 3$) by using an algorithm for the higher-dimensional minima problem. Concerning the minima problem in general dimension d, the following theorem is well known.

THEOREM 10 Ref. [21,25] *For $d \geq 3$, the minima of n points in the d-dimensional space can be computed in $O(dn \log^{d-2} n)$ time by a divide-and-conquer algorithm.*

The algorithm of Preparata and co-workers is based on a divide-and-conquer approach. It contemplates partitioning point sets according to the values of the coordinates $x_d, x_{d-1}, \ldots, x_2$, in turn. The first step of their minima algorithm is the equipartition of the given S on x_d into (S_1, S_2). We say that S is equipartitioned on x_j into (S_1, S_2) if S is split at the median of the x_j-coordinates of the points of S into (S_1, S_2). They then recursively apply the algorithm to the sets S_1 and S_2 and obtain the two sets, minima(S_1) and minima(S_2), of their respective minima. To combine the result of the recursive calls, they first note that all elements of minima(S_2) are also minima of S, while the elements of minima(S_1) are not necessarily so. Indeed, an element of minima(S_1) is a minimal element of S if and only if it is not dominated by any element of minima(S_2). Therefore in their algorithm the merge step of the divide-and-conquer is a little complicated, and they resort once again to the divide-and-conquer approach in order to accomplish the merge step. Descriptively, computation proceeds on the tracks of a double recursion: one on set sizes, the other on the dimension.

This theorem is too general for our purpose, because in our problem points are not general points in the d-dimensional space but are grid points in the

$$\overbrace{n \times n \times \cdots \times n}^{d \text{ times}} = n^d$$

grid. The coordinates of points can be considered as integers. For this case, we have the following theorem.

THEOREM 11 *For $d \geq 3$, the minima of n grid points in the d-dimensional N^d grid can be computed in $O(dn \log^{d-2}(\min\{n, N\}))$ time by using a modified divide-and-conquer algorithm.*

Proof We here just give a main idea. At the topmost recursion of the original divide-and-conquer algorithm in [21], the point set is equipartitioned with respect to a coordinate and the minima of each partitioned subset is recursively computed. Here, after $O(\log N)$ recursions, points in subproblems come to have the same coordinate, since they are grid points in the N^d grid. Therefore, the minima problem for this point subset is really a $(d-1)$-dimensional problem. At this point, instead of dividing this point subset further, we call the $(d-1)$-dimensional minima algorithm for it. Since there are several types of recursions in the original algorithm, we have to modify it to this grid case in a more elaborate manner, but through careful analysis the theorem can be shown.

Since the minima of n points can be obtained by almost the same algorithm in this theorem, we can develop an efficient LCS Algorithm A′ for d strings ($d > 3$) by generalizing the three-dimensional Algorithm A into A′, where in A′ no sorting is done to maintain D^k and the minima problem is solved from scratch by the specialized minima algorithm.

For this Algorithm A′, we have the following theorem.

THEOREM 12 *The LCS problem for d (≥ 3) strings of length n can be solved in time $O(nsd + Dsd(\log^{d-3}(\min\{\max_{\sigma,i,j,k} |D^k(\sigma)[i,j]|, n\}) + \log^{d-2} s)) = O(nsd + Dsd(\log^{d-3} n + \log^{d-2} s))$.*

Note that if the original minima algorithm in [21] is used in A′, the dominant term in the time complexity becomes $O(Dsd \log^{d-3}(\max |D^k(\sigma)[i,j]|))$, which becomes much worse than the bound in Theorem 12 if we only use a trivial bound of $|D^k(\sigma)[i,j]| \leq n^{d-1}$. Although we have not implemented the Algorithm A′, it is likely to be fairly faster than the dynamic programming algorithm, since the number D of dominants becomes much smaller than n^d as d becomes larger.

6 EXPERIMENTAL RESULTS

6.1 DNA Sequence ($s = 4$)

One important area in which the LCS problem is relevant is the modeling of genetic sequences. DNA sequences consist of the alphabet {A,C,G,T}, where the components of this alphabet represent occurrences of the nucleotides adenine, cytosine, guanine and thymine, respectively. DNA sequences are sometimes modeled as sequences of independent and identically distributed symbols from the alphabet {A,C,G,T} for theoretical analysis, although this model is too rough from the viewpoint of biology. Similarly, proteins are sometimes modeled as independent and identically distributed sequences of amino acid (e.g. [28]). The distribution of the length of the longest common substring of two independent random sequences has been examined by relating it to that of the longest run of heads in a sequence of tosses of a coin. Theoretical results have been obtained by Arratia and Waterman [3] and Arratia *et al.* [4].

We have run experiments for our algorithms on three strings over alphabet of size $s = 4$. Strings are randomly and independently generated from the alphabet. Both the simple $O(n^3)$ dynamic programming algorithm (to calculate only the length of an LCS) and our algorithms have been implemented in C, and tests are run on a Sun SPARC station ELC using the `times` function. Table I shows the running time of two LCS algorithms, the naive dynamic programming Algorithm DP and our Algorithm A.

TABLE I For $s = 4$, the running time (s) of DP, and the maximum ($A_{\max}$), average (A_{ave}) and minimum ($A_{\min}$) running time of our Algorithm A for 20 different test sets of three strings of length n

n	100	200	300	400	500	600	700
DP	5.3	43.7	160.5	360.1	701.0	1284.4	2043.3
$A_{\max}$	0.7	4.8	14.5	37.0	70.7	117.5	192.9
A_{ave}	0.6	4.2	13.0	33.7	62.4	107.2	174.2
$A_{\min}$	0.5	3.6	11.4	30.0	56.0	97.3	159.9

As is seen from the Table I, our algorithm runs more than 10 times faster than the DP algorithm. This is mainly because the number of dominants D is much smaller than the size n^3 of the matrix. We also confirmed this fact for the LCS problem of more than three strings by computational experiments. This indicates the usefulness of devising algorithms whose complexity depends mainly on D if there are many strings.

For higher dimension ($d > 3$), we run experiments using a simpler Algorithm B than the Algorithm A as follows, since for $d > 3$ we do not need the sorting in order to get the linearity of D. Step 1 (computing the T table) is the same as in Section 3.2. Also, as for a minima algorithm, we use an Algorithm $\mathrm{M}'(A, d)$, to be given later, which computes the minima of points in A in the d-dimensional space.

```
ALGORITHM B($n, S, \{A_1, \ldots, A_d\}$)
  1.  computing the T table; $D^0 = \{[0, \ldots, 0]\}$; $k := 0$
  2.  while $D^k$ not empty do
        $A := \phi$
2.1     for $p \in D^k$ do
          the parents of $p(\Sigma)$ are set to $p$
          $A := A \cup p(\Sigma)$
2.2     $D^{k+1} := \mathrm{M}'(A, d)$
2.3     $k := k + 1$
      end-while
  3.  pick a point $p$ in $D^{k-1}$
      while $k - 1 > 0$ do
        output $p$ and set $p$ to the parent of $p$ by the parent pointer
        $k := k - 1$
      end-while
```

The following Algorithm C is a little improved one using the property of dominants which is described in Theorem 5. This property is the same as the first idea described in Section 3.1. The Algorithm B computes the minima of $D^k(\Sigma)$, and on the other hand the Algorithm C computes the minima of each $D^k(\sigma_j)$. Specifically, in Step 2.2 of the Algorithm C, a set A of candidates for dominants are partitioned into a subset $A[\sigma]$ ($\sigma \in \Sigma$) of dominants whose matching character is σ. Although this improvement does not influence the theoretical time complexity so much, it makes our algorithm practical as seen in the results of the experiments. We also adopt another minima algorithm $\mathrm{M}''(S, d)$, to be give later, computing the minima among points in S in the d-dimensional space.

ALGORITHM C($n, S, \{A_1, \ldots, A_d\}$)

```
 1.  computing the T table; $D^0 = \{[0, \ldots, 0]\}$; $k := 0$
 2.  while $D^k$ not empty do
         $A := B := \phi$
2.0      for $\sigma \in \Sigma$ do
           $A[\sigma] := \phi$
2.1      for $p \in D^k$ do
           the parents of $p(\Sigma)$ are set to $p$.
           $A := A \cup M''(p(\Sigma), d)$
2.2      for $p \in A$ do
           $A$[matching character of $p$] $:= p$
2.3      for $\sigma \in \Sigma$ do
           $B := B \cup M''(A[\sigma], d)$
2.4      $D^{k+1} := B$
         $k := k + 1$
     end-while
 3.  pick a point $p$ in $D^{k-1}$
     while $k - 1 > 0$ do
         output $p$ and set $p$ to the parent of $p$ by the parent pointer
         $k := k - 1$
     end-while
```

We used also the simplest minima Algorithm M′ which is mentioned in Section 2, because the minima algorithm which is described in Section 5 is too complicated to achieve the efficiency for moderate n. The minima Algorithm M′ takes $O(dn^2)$ time to compute the minima of n points of d-dimensional space, and it is slower than $O(dn \log^{d-2} n)$ time algorithm which is used in the Algorithm A′ in Section 5, but for moderate n, say, about 100 or 500, there may be no great difference between these two algorithms.

ALGORITHM M′(S, d)

```
 1.  $A := \phi$ ($A$ is the set of minima of dimension $d$)
 2.  for $p \in S$ do
       $f := 1$
2.1    for $q \in S - \{p\}$ do
2.2        if $p \succeq q$ then $f := 0$
2.3    if $f = 1$ then $A := A \cup \{p\}$
 3.  return $A$
```

TABLE II For $s=4$ the length of strings n, and the running time (s) of DP, and the maximum ($B_{\max}$), the average (B_{ave}) and the minimum ($B_{\min}$) running time of our Algorithm B, and the average running time, C_{ave}, of our Algorithm C for at least 20 different test sets of five random strings of length n

n	10	20	30	40	50
DP	1.4	49.8	295.2	1376.6	5347.5
$B_{\max}$	0.0	0.1	1.0	5.9	30.5
B_{ave}	0.0	0.0	0.6	4.1	23.1
$B_{\min}$	0.0	0.0	0.3	1.8	17.5
C_{ave}	0.0	0.0	0.3	1.9	8.7

Actually, we use a little improved minima Algorithm M$''$ in our experiment, which is faster than the previous minima Algorithm M$'$, because it reduces the number of comparisons using an array of flags.

Algorithm M$''(S,d)$

```
1. $A := \phi$ ($A$ is the set of minima of dimension $d$)
   for $i := 1$ to $|S|$ do $f[i] := 1$
2. for $i := 1$ to $|S|$ do
       if $f[i] = 1$ then
         for $j := 1$ to $|S|$ do
           if $f[j] = 1$ then
             if $S[i] = S[j]$ and $j > i$ then $f[j] := 0$
             if $S[i] \neq S[j]$ and $S[i] \succeq S[j]$ then $f[i] := 0$
             if $S[i] \neq S[j]$ and $S[i] \preceq S[j]$ then $f[j] := 0$
         if $f[i] = 1$ then $A := A \cup \{S[i]\}$
3. return $A$
```

The results for $d=5$ are shown in Table II. The speedup ratio is higher than for the case of $d=3$. For small alphabets our algorithm clearly provides a considerable speedup. For example, when the length of input strings is 50, the dynamic programming algorithm needs about 90 min, but our algorithm takes only 23 s on average. Moreover, our improved algorithm needs only 9 s on average.

6.2 Amino Acid Sequence ($s=20$)

We have also performed experiments for the case of $s=20$, which is the case where the protein consists of 20 kinds of amino acids. Although in this case s becomes bigger than the case of DNA sequences and it influences the time complexity, our algorithm still outperforms the dynamic programming algorithm DP. This is because, as s becomes larger with $s \ll n$, the number of dominant matches D tends to be smaller, and the influence of the factor $O(s \log s)$ is likely to be relatively negligible. Since we adopt for experiments a little simpler algorithm whose time complexity is $O(ns + Ds^2)$, the influence of the alphabet size s appears, that is, for $s = 20$ our algorithm runs slightly slower than the case that $s = 4$ (see Table III).

We also run experiments for $d = 5$. As is seen from Table IV, when the number of strings d is five, our algorithm runs faster for $s = 20$ than for $s = 4$, unlike the results for $d = 3$. The reason why for $d > 3$ our algorithm becomes faster as the alphabet size s grows is that for $d > 3$ the number of dominant matches D becomes so smaller as s grows that the decrease of D cancels out the influence of the increase of s, as is shown in the next section. When $n = 100$ we cannot do the dynamic programming algorithm because of lack of memory.

We have done experiments for real amino acid sequences. Random sequences tend to have the same number of each character on average, and this case is favorable for our algorithm because in this case D

TABLE III For $s = 20$, the running time (s) of DP, and the maximum ($A_{\max}$), the average (A_{ave}) and the minimum ($A_{\min}$) running time of our Algorithm A for at least 20 different test sets of three random strings of length n

n	100	200	300	400	500	600	700
DP	5.8	43.3	154.0	345.7	677.6	1237.3	1972.9
$A_{\max}$	1.0	7.5	24.1	60.5	111.5	192.7	313.6
A_{ave}	0.8	6.6	22.8	57.3	111.4	187.1	287.6
$A_{\min}$	0.6	5.0	19.0	50.3	96.3	168.3	267.1

TABLE IV For $s = 20$ the length of strings n, and the running time (s) of DP, and the maximum ($B_{\max}$), the average (B_{ave}) and the minimum ($B_{\min}$) running time of our Algorithm B, and the average running time, C_{ave}, of our Algorithm C for at least 20 different test sets of five random strings of length n

n	10	20	30	40	50	100
DP	1.4	49.2	295.2	1376.6	5123.0	—
$B_{\max}$	0.0	0.0	0.0	0.0	0.3	70.0
B_{ave}	0.0	0.0	0.0	0.0	0.1	37.3
$B_{\min}$	0.0	0.0	0.0	0.0	0.0	13.3
C_{ave}	0.0	0.0	0.0	0.1	0.3	12.0

becomes relatively small. Therefore, we tried to do experiments for non-random sequences. If we use the dynamic programming algorithm for both Example 1 ($n=63$, $d=6$, $s=20$) and Example 2 ($n=132$, $d=8$, $s=20$), we cannot execute the algorithm due to lack of memory on our computer. In fact, Example 1 requires the dynamic programming table of about 189M entries, and Example 2 requires much more entries (about 92000T entries).

Example 1:

```
Copia17.6: ILDFHEKLLHPGIQKTTKLFGETYYFPNSQLLIQNIINECSICNLAKTEHRNTDMPTKTT
M-MULV : LLDFLHQLTHLSFSKMKALLERSHSPYYMLNRDRTLKNITETCKACAQVNASKSAVKQGTR
HTLV   : LTDALLITPVLQLSPAELHSFTHCGQTALTLQGATTTEASNILRSCHACRGGNPQHQMPRGHI
RSV    : VADSQATFQAYPLREAKDLHTALHIGPRALSKACNISMQQAREVVQTCPHCNSAPALEAGVN
MMTV   : ISDPIHEATQAHTLHHLNAHTLRLLYKITREQARDIVKACKQCVVATPVPHLGVN
SMRV   : ILTALESAQESHALHHQNAAALRFQFHITREQAREIVKLCPNCPDWGSAPQLGVN
```

Example 2:

```
HIV-1  :IEQLIKKEKVYLAWVPAHKGIGGNEQVDKLVSAGIRKILFLDGIDKAQDEHEKYHSNWRAMASDFNLPPVVAKEIVASCDKCQLKGEAMHGQVDCSPGIWQLDCTHLEGKVILVAVHVASGYIEAEVIPAET
SIVagm:IALMIQKQQIYLQWVPAHKGIGGNEEIDKLVSKGIRRVLFLEKIEEAQEKHERYHNNWKNLADTYGLPQIVAKEIVAMCPKCQIKGEPVHGQVDASPGTWQMDCTHLEKKVVIVAVHVASGFIEAEVEPRET
EIAV   :GQKFAQLIILQHHSNSRQPWDENKISQRGDKGFGSTGVFWVENIQEAQDEHENWHTSPKILARNYNIPREQARQIVRQCPICATYLPVPHLGVNPRGLLPNMIWQMDVTHYSEFGNLKYIHVSIDTFSGFLL
SRV-1  :YNRSIPFYIGHVRAHSGLPGPIAHGNQKADLATKTVASNINTNLESAQNAHTLHHLNAQTLKLMFNIPREQARQIVKQCPICVTYLPVPHLGVNPRGLFPNMIWQMDVTHYSEFGNLKYIHVSIDTFSGFLL
MPMV   :YNRSIPFYIGHVRAHSGLPGPIAQGNQRADLATKIVASNINTNLESAQNAHTLHHLNAQTLRLMFHITREQAREIVKLCPNCPDWGHAPQLGVNPRGLKPRVLWQMDVTHVSEFGKLKYVHVTVDTYSHFTF
MMTV   :LQRLIHKRQEKFYIGHIRGHTGLPGPLAQGNAYADSLTRILTALESAQESHALHHQNAAALRFQFALSRKEAREIVTQCQNCCEFLPTPHMGINPRGIRPLQMWQMDVTHIPSFGRLQYVHVSVDTCSGVMF
IAPH18:LNRRFPVFITHVRAHSGLPGPMSLGNDLADKATKLVATALSTHAQAAKEFHKRFHVTAETLRRRFNISMQQAREVVQTCPHCNSAPALEAGVNPRGLGPLQIWQTDFTLEPRMAPRSWLAVTVDTASSAIVV
RSV    :EDALSQRSAMAAVLHVRSHSEVPGFFTEGNDVADSQATFQAYPLREAKDLHTALHIGPRALSKACPNPRISAWDPRSPATLCETCQKLNPTGGGKMRTIQRGWAPNHIWQADITHYKYKQFTYALHVFVDTYSGA
```

Even if there is enough memory to make the dynamic programming algorithm available, too much time is necessary (Example 1, more than 70 h, and Example 2, probably more than 1 year). On the other hand, our Algorithm B needs only 1.92 s for Example 1, and about 40 minutes (2491.42 s) for Example 2.

We present below the result of the experiment for Example 1. The string `LTLLIECCAN` is one of the LCSs of Example 1. The string `C**C` is the pattern which has a biological meaning.

```
ILDFHEKL_____LHPGIQKTTK_LFGETYYFPNSQL_____LIQNI___IN_______E_______CSI__________________CNL___AKTEHR___NTDMPTKTT___
LLDFLHGL_____LDFLHG__TH_LSFSKMKALERSHSPYYMLNRDRTLKNIT_______ET______CKA__________________C_____AQV______NASKSAVKGGTR
_____________L_______TDAL__________________L________ITPULGLSPAELHSFH___CGGTALTLGGATTTEASNILRSCH____ACRGG____NDGHGMPRGHI_
VADSGATFGAYP_LREAKDLHTA_LHIGPRA___________LSKACN___ISMGGAR___EVVGT____CPH___________________CNS___APALEAGV_N__________
ISDPIHEATGAHTLHHLNAH_T__LHHLNAHT__________LRLLTK___ITR_______EGARDIVKACKQ___________________CVV___ATPVPHLGVN__________
T____________L_______TA_LESAGESHA_________LRFGFH___ITR_______EGAREIVKLCPN___________________CPDWGSAPGLGV___N__________
```

We probably can refine this result to be suitable for biological facts. In this case, the interval between one character and the next character of an LCS must have similar length for all input strings. The extended problem which takes consideration of this fact can be considered, and it is a future problem.

For Example 2, the obtained LCS is the string `HGNDSAQRALP-PGGPWQDTLVT` of length 22 and the number of dominant matches is 9447.

6.3 Estimation of D and R

We measured the number of dominant matches D and the number of matches R, which influence the time complexity of algorithms for the LCS problem. $D_{\max}$ means the maximum number of dominant matches for at least 20 random strings. $R_{\max}$ is the maximum number of matches for the same sample strings. D_{ave} and R_{ave} are the average numbers, and $D_{\min}$ and $R_{\min}$ are the minimum numbers. $l_{\max}$, l_{ave}, $l_{\min}$ are the lengths of the LCS gained. The expectation of R is n^d/s^{d-1}, because the probability that each point becomes a match is s/s^d. We were not able to estimate the expectation of D, so it is a future problem.

We first measure D for $d = 3$ and $n = 50$ changing s. In this case, n^d is equal to 125 000. As s becomes larger, D and R becomes smaller, although the ratio between D and R becomes larger. While for $s = 4$ the ratio between D and R is about 0.07, for $s = 64$ the ratio is about 0.45 (see Table V).

Secondly, we measure D for $d = 5$ and $n = 50$. When d becomes large, the ratio of decreasing of D becomes large, and D itself becomes considerably smaller than not only n^d but also R. The ratio between D and R is about 0.0015, which is very small. Hence, our algorithm may remain efficient for large d (see Table VI).

TABLE V For $d = 3$, the alphabet size s, the length of the LCS l, the number of dominant matches D and the number of matches R for 20 random sequences

s	2	4	8	16	32	64
$l_{\max}$	36	27	19	12	7	5
l_{ave}	34	24	16	11	6	3
$l_{\min}$	32	21	14	10	5	3
$D_{\max}$	823	693	379	120	46	22
D_{ave}	634	550	279	107	41	14
$D_{\min}$	443	417	201	86	34	11
$R_{\max}$	32100	9540	2298	651	229	53
R_{ave}	31020	7687	1918	486	130	31
$R_{\min}$	29550	6864	1689	372	83	22
n^d/s	62500	31250	15625	7813	3906	1953

TABLE VI For $d = 5$, the alphabet size s, the length of the LCS l, the number of dominant matches D and the number of matches R for 20 trials

s	2	4	8	16	32	64
$l_{\max}$	32	22	13	8	4	2
l_{ave}	30	19	11	6	2	1
$l_{\min}$	28	18	10	5	0	0
$D_{\max}$	4117	3137	908	209	39	4
D_{ave}	3156	1997	586	127	21	2
$D_{\min}$	2106	1428	240	82	8	0
$R_{\max}$	22008600	2130660	133344	13108	997	56
R_{ave}	19036495	1252797	80913	5349	321	16
$R_{\min}$	17798400	1024044	57360	3215	120	0
n^d/s	156250000	78125000	39062500	19531250	9765625	4882812

7 CONCLUSIONS

The LCS problem has been studied by a number of researchers and its complexity has been improved in different respects. We have presented a better solution when the number of strings is more than two. The time complexity of the 3-LCS problem for a fixed finite alphabet is $O(n + D)$. This gives a speedup over previous algorithms, such as the $O(n^3)$ dynamic programming algorithm.

Computational experiments have been done for random sequences and moderate alphabets. The case of $s = 4$ corresponds to the DNA sequence, and the case of $s = 20$ corresponds to the amino acid sequence. The results show that our algorithm is quite promising. The time complexity of our algorithm is a polynomial of the alphabet size s, but as seen in the results of experiments the influence of s can almost be ignored.

Regarding the case of $d \geq 4$, the time complexity of our algorithm is not linear in D but proportional to $D \log^{d-3} n$, although the results of experiments for $d = 5$ are satisfactory. It is left as a future problem to make it linear by making use of more properties of the LCS problem.

Acknowledgments

The authors would like to thank both referees for giving many useful comments on the original version of this paper. The work of the second author was supported in part by the Grant-in-Aid for Scientific

Research on Priority Areas, 'Genome Science', of the Ministry of Education, Science, Sports and Culture of Japan.

References

[1] A.V. Aho, D.S. Hirschberg and J.D. Ullman (1976) Bounds on the complexity of the longest common subsequence problem, *Journal of the Association for Computing Machinery*, **23**, 1–12.

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

[3] R. Arratia and M.S. Waterman (1985) Critical phenomena in sequence matching, *The Annals of Probability*, **13**, 1236–1249.

[4] R. Arratia, L. Gordon and M.S. Waterman (1990) The Erdös–Rényi Law in distribution, for coin tossing and sequence matching, *The Annals of Statistics*, **18**, 539–570.

[5] R.A. Baeza-Yates (1991) Searching subsequences, *Theoretical Computer Science*, **78**, 363–376.

[6] H.L. Bodlaender, R.G. Downey, M.R. Fellows and H.T. Wareham (1995), The parameterized complexity of sequence alignment and consensus, *Theoretical Computer Science*, **147**, 31–54.

[7] F.Y.L. Chin and C.K. Poon (1990), A fast algorithm for computing longest common subsequences of small alphabet size, *Journal of Information Processing*, **13**, 463–469.

[8] F. Chin and C.K. Poon (1994) Performance analysis of some simple heuristics for computing longest common subsequences, *Algorithmica*, **12**, 293–311.

[9] M.O. Dayhoff (1965) Computer aids to protein sequence determination, *Journal of Theoretical Biology*, **8**, 97–112.

[10] M.O. Dayhoff (1969) Computer analysis of protein evolution, *Scientific America*, **221**, 86–95.

[11] M.L. Fredman (1975) On Computing length of the longest increasing subsequences, *Discrete Mathematics*, **11**, 29–36.

[12] J. Gallant, D. Maier and J.A. Storer (1980) On finding minimal length superstrings, *Journal of Computer and System Sciences*, **20**, 50–58.

[13] D.S. Hirschberg (1975) A linear space algorithm for computing maximal common subsequences, *Communications of the ACM*, **18**, 341–343.

[14] D.S. Hirschberg (1977) Algorithms for the longest common subsequence problem, *Journal of the Association for Computing Machinery*, **24**, 664–675.

[15] W.J. Hsu and M.W. Du (1984) Computing a longest common subsequence for a set of strings, *BIT*, **24**, 45–59.

[16] J.W. Hunt and M.D. McIlory (1976) An algorithm for differential file comparison, Computing Science Technical Report 41, AT&T Bell Laboratories.

[17] J.W. Hunt and T.G.A. Szymanski (1977), A fast algorithm for computing longest common subsequences, *Communications of the ACM*, **20**, 350–353.

[18] M. Iri, T. Koshizuka, eds. (1986) *Computational Geometry and Geographical Information Processing* (in Japanese), bit annex, Kyoritsu Shuppan, Tokyo.

[19] R.W. Irving and C.B. Fraser (1992), Two algorithms for the longest common subsequence of three (or more) strings, *Combinatorial Pattern Matching* (Tucson, AZ, 1992), Lecture Notes in Computer Science (Springer-Verlag), **644**, 214–229.

[20] T. Jiang and M. Li (1995) On the approximation of shortest common supersequences and longest common subsequences, *SIAM Journal on Computing*, **24**, 1122–1139.

[21] H.T. Kung, F. Luccio and F.P. Preparata (1975), On finding the maxima of a set of vectors, *Journal of the Association for Computing Machinery*, **22**, 469–476.

[22] S.Y. Lu and K.S. Fu (1978) A sentence-to-sentence clustering procedure for pattern analysis, *IEEE Transactions on System and Man, and Cybernetics*, **8**, 381–389.
[23] D. Maier (1978) The complexity of some problems on subsequences and supersequences, *Journal of the Association for Computing Machinery*, **22**, 177–183.
[24] W.J. Masek and M.S. Paterson (1980), A faster algorithm computing string edit distances, *Journal of Computer and System Sciences*, **20**, 18–31.
[25] F.P. Preparata and M. Shamos (1985) *Computational Geometry*, Springer-Verlag.
[26] D. Sankhoff (1972) Matching sequences under deletion insertion constraints, *Proceedings of the National Academy of Sciences*, **69**, 4–6.
[27] D. Sankhoff and J.B. Kruskal (1983) Time warps, string edits and macromolecules: the theory and practice of sequence comparison, Addison-Wesley, Reading MA.
[28] P.R. Sibbald and M.J. White (1987) How probable are antibody cross-reactions?, *Journal of Theoretical Biology*, **127**, 163–169.
[29] T.F. Smith and M.S. Waterman (1981) Identification of common molecular subsequences, *Journal of Molecular Biology*, **147**, 195–197.
[30] R.A. Wagner and M.J. Fischer (1974) The string-to-string correction problem, *Journal of the Association for Computing Machinery*, **21**, 168–173.