アルゴリズム 27－2
（1992．5．21）

# 文字数が少ない場合の複数の文字列間の最長共通部分列問題

博田 浩司　今井 浩

東京大学理学部情報科学科

最長共通部分列問題とは、二つ以上の文字列（例えばDNAやアミノ酸配列）が与えられたとき各文字列から0個以上の文字を取り除いて得られる共通部分列で最長のものを決定する問題である。二つの文字列間の最長共通部分列を計算するアルゴリズムは多くの論文によって与えられているが、三つ以上の文字列に対してはまだ効率のいいアルゴリズムが見つかっていない。この論文では、文字数が少ない場合の三つの文字列間の最長共通部分列を効率良く計算する方法を提案し、その手間を理論的に評価するとともに、計算機実験によりその計算時間を定量的に評価した。

# THE LONGEST COMMON SUBSEQUENCE PROBLEM FOR SMALL ALPHABET SIZE BETWEEN MANY STRINGS

Koji Hakata　Hiroshi Imai

Department of Information Science, University of Tokyo

Given two or more strings (for example, DNA and amino acid sequences), the longest common subsequence (LCS) 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 is no efficient algorithm for computing an LCS between more than two strings. This paper proposes a method for computing efficiently the LCS between three strings of small alphabet size, evaluates its theoretical time complexity, and estimates the computing time by computational experiments.

# 1 Introduction

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 $s$. A *subsequence* of $A_i$ can be obtained by deleting zero or more (not necessarily consecutive) 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* (LCS) problem is to find a common subsequence $B$ of $A_1, A_2, \ldots, A_d$ of maximal length. The LCS problem is a common task in DNA sequence analysis, and has applications to genetics and molecular biology. Throughout this paper, $l$ is the length of an LCS. We assume $n_1 = n_2 = \cdots = n_d = n$ only for convenience.

For string $A = a_1 a_2 \cdots a_n$, $A[p \ldots q]$ is $a_p a_{p+1} \cdots a_q$. Define an $L$-matrix for the $d$ strings $A_1, A_2, \ldots, A_d$ as an integer array $L[0 \ldots n_1, 0 \ldots n_2, \cdots, 0 \ldots n_d]$ such that $L[p_1, p_2, \ldots, p_d]$ is the length of an LCS for $A_i[1 \ldots p_i], i = 1, 2, \ldots, d$. For $d = 2$, since $A_1[1 \ldots 0]$ and $A_2[1 \ldots 0]$ are empty strings, $L[i, 0] = L[0, j] = L[0, 0] = 0$. For $1 \leq i \leq n_1$ and $1 \leq j \leq n_2$, $L[i, j] =$ if$(A_1[i] = A_2[j])$ $L[i-1, j-1] + 1$ else $\max\{L[i-1, j], L[i, j-1]\}$.

Hirschberg [3] applied a dynamic programming strategy to derive an $O(n^d)$ algorithm that solves the problem by filling $L$ row by row. Since $L$ 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, and the contours can be completely specified by their corner points, which are called dominants. It is easy to see that there are $l$ contours and an LCS can be obtained by finding the set of dominants, instead of filling all the $(n+1)^d$ entries in $L$.

For $d = 2$, various improvements to the simple $O(n^2)$ algorithm were made and they concentrate on efficient generation of the dominants. One of the earliest variations was by Hirschberg [4], whose algorithm repeatedly scans $A_1$ and generates all the dominants of each contour after each scan. Since each scan takes $n$ steps and there are $l$ contours, the total time complexity of the algorithm is $O(n \log s + ln)$ where $n \log s$ is the preprocessing time. Hunt and Szymanski [5] attempted to find the dominants in $L$ row by row by binary search. The total time complexity is bounded by $O(n \log s + r \log n)$, where $r$ is the total number of matches between the two strings. Apostolico and Guerra [1] improved the time complexity to $O(n(\log s + \log n) + d \log(n^2/d)))$ with the use of a proper data structure, where $d(\leq r)$ is the total number of dominants between the two strings.

Chin and Poon [2] devised the algorithm which, instead of scanning $A_1$, generates the dominants in each contour from dominants with a lower value. Since each dominant can generate at most $s$ dominants of the next contour, no more than $O(ds)$ time will be required for this stage. It can be shown with some careful analysis that their algorithm takes $O(ns + \min(ds, lm))$ time and $O(ns + d)$ space, where $O(ns)$ is the preprocessing time.

We have devised an algorithm that, for three strings and small alphabet size, runs in time $O(ns + ds^2)$, where $d$ is again the number of dominants between the three strings and $O(ns)$ is the preprocessing time. The running time depends on the nature of the input. The basic idea is based on the algorithm of Chin and Poon, and our algorithm is a generalization of their algorithm, but more properties of dominant matches are utilized for efficiency of the algorithm.

In the first place, the algorithm expands the candidates of next dominants whose number is small when the alphabet size is small, and then eliminates redundant dominants from the candidates. It is observed by the analysis and experiments that the algorithm performs better than the simple dynamic programming.

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

## 2 Preliminaries

Denote the position $p$ of $L$ by $[p_1, p_2, \ldots, p_d]$. $p$ is a *match* iff $A_1[p_1] = A_2[p_2] = \cdots = A_d[p_d]$. Define a region $< p, q >$ in $L$, as the set of elements, $\{[r_1, r_2, \ldots, r_d] \mid p_i < r_i \leq q_i, i = 1, 2, \ldots, d\}$. Point $p$ *dominates* point $q$ if $p_i \leq q_i$ for $i = 1, 2, \ldots, d$ (denoted by $p \preceq q$ or $[p_1, p_2, \ldots, p_d] \preceq [q_1, q_2, \ldots, q_d]$). Denote $p_i < q_i$, for $i = 1, 2, \ldots, d$ by $p \prec q$. $p$ is a *k-dominant* iff $L[p] = k$ and $L[q] < k$ for all other elements $q$ in $< [0, 0, \ldots, 0], p >$. Point $p$ and $q$ are *independent* unless $p \succeq q$ or $p \preceq q$. $p$ is a dominant only if $p$ is a match, that is, $r$ (the number of matches) $\leq d$. Furthermore, if $p$ and $q$ are $k$-dominants, then $p$ and $q$ are independent of each other.

Given a set $S$ of $n$ points in $L$, $\preceq$ (the dominance relation set $S$) is clearly a partial order on $S$ for $d > 1$. A point $p$ in $S$ is a *maximal* element of $S$, if there does not exist $q$ in $S$ such that $p \neq q$ and $p \preceq q$. The *maxima* problem consists of finding all the maximal elements of $S$ under dominance $\preceq$. A *minimal* element and the *minima* problem are also defined in the same way. Kung, Luccio and Preparata [6] have proved that in the comparison-tree model, any algorithm that solves the maxima problem in two dimensions requires time $\Omega(n \log n)$.

Define $D^k$ as the set of $k$-dominants, that is, all the dominants on the contour with value $k$. Define $D_p^k$ as the set of all the $k$-dominants in $< p, [n_1, n_2, \ldots, n_d] >$. Thus the set of dominants is composed of $l$ disjoint subsets $D^1, D^2, \ldots, D^l$, where $l$ is the length of an LCS. Let $p_i(\sigma)$ be the position of the first $\sigma$ in $A_i[p_i \ldots n]$, and for $p = [p_1, \ldots, p_d]$, let $p(\sigma)$ be $[p_1(\sigma), p_2(\sigma), \ldots, p_d(\sigma)]$ and let $p(\Sigma)$ be $\{p(\sigma) \mid \sigma \in \Sigma\}$. Define $S^k$ as $\{p(\Sigma) \mid p \in D^{k-1}\}$, the set of candidates of $k$-dominants, and $S^k(\sigma)$ as $\{p \in S^k \mid A_1[p_1] = \sigma\}$.

**Lemma 1.** A match $q$ is a $(k+1)$-dominant iff $q$ satisfies that there is no match $r$ such that $p \prec r \prec q$ for any $k$-dominant $p$ such that $p \prec q$ (if such $p$ does not exist, $q$ is not a $(k+1)$-dominant).

**Proof.** This lemma is evident from the definition of dominants.

**Lemma 2.** 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 $\sigma$ between $p_i$ and $p_i(\sigma)$, if $q_i < p_i(\sigma)$, there is also no match of $\sigma$ between $q_i$ and $p_i(\sigma)$, and hence $p_i(\sigma) \leq q_i(\sigma)$. Otherwise, $p_i(\sigma) \leq q_i \leq q_i(\sigma)$.

**Lemma 3.** For any match $p$, $p \prec p(\sigma)$ and there is no match $r$ of $\sigma$ such that $p \prec r \prec p(\sigma)$.

**Proof.** This lemma is evident from the definition of $p(\sigma)$.

**Lemma 4.** $D^{k+1}$ is a subset of $S^{k+1}$.

**Proof.** Assume that a match $q$ is not in $S^{k+1}$. From Lemma 1, if $q$ is a $(k+1)$-dominant, then there must exist $p \in D^k$ such that $p \prec q$. Then, for the same $p$, $p(A_1[q_1]) \prec q$ holds. Again from Lemma 1, $q$ can not be a $(k+1)$-dominant.

**Theorem 1.** $D^{k+1}$ is the minima of $S^{k+1}$.

**Proof.** For $q \in S^{k+1}$ not in the minima of $S^{k+1}$, there exists some $r \in S^{k+1}$ such that for $p \in D^k$, $p \prec r \prec q$. From Lemma 1, $q$ can not be a $(k+1)$-dominant.

**Theorem 2.** Assume $p = [p_1, p_2, \ldots, p_d]$ and $q = [q_1, q_2, \ldots, q_d]$ are $k$-dominants and $p_1 < q_1$. If $p(\sigma)$ and $q(\sigma)$ are independent, at least one of $p_i(\sigma) = q_i(\sigma)(i = 1, 2, \ldots, d)$ holds.

**Proof.** From Lemma 3, $p(\sigma) \succ p$ and $q(\sigma) \succ q$. Since $p_1 < q_1$ and the $k$-dominants are independent of each other, $p_2 > q_2$ or $p_3 > q_3$ or $\cdots$ or $p_d > q_d$. Since $p(\sigma)$ and $q(\sigma)$ are independent, if $p(\sigma) \succeq q(\sigma)$ then from $p_1 < q_1$ and Lemma 2, $p_1(\sigma) \leq q_1(\sigma)$, then $p_1(\sigma) = q_1(\sigma)$. Otherwise, that is, when $p(\sigma) \preceq q(\sigma)$, for at least one of $i$, $p_i > q_i$ holds, then assume $p_i(\sigma) > q_i(\sigma)$. Since $p_i(\sigma) > p_i > q_i$, $r = [q_1(\sigma), \ldots, p_i(\sigma), \ldots, q_d(\sigma)]$ is a match of $\sigma$ such that $q \prec r \prec q(\sigma)$. It contradicts Lemma 3. Thus $p_i(\sigma) = q_i(\sigma)$.

**Theorem 3.** 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 $\sigma_j$ in $< p, [n_1, n_2, \ldots, n_d] >$. From the definition of $p(\sigma)$, $p(\sigma_j)$ is the first match of $\sigma_j$ in $< p, [n_1, n_2, \ldots, n_d] >$, then $p(\sigma_j) \preceq q(\sigma_j)$.

## 3 The algorithm

We present algorithm A which obtains an LCS $B$ of length $l$ of input strings $A_1, A_2$ and $A_3$ in time $O(ns + ds^2)$ and space $O(ns + d)$. The algorithm is based on an efficient representation of the $L$-matrix.

The algorithm is based on three ideas. The first idea is the use of Theorem 3. Due to this theorem, after we examined the dominance relation between $p(\sigma_x)$ for $x = 1, 2, \ldots, s$ and $p \in D^k$, we need not compute the maxima of the whole $S^{k+1}$, but only compute each maxima of $S^{k+1}(\sigma_x)$.

The second idea is the use of the fact that if the points of $D^k$ are sorted concerning each coordinate, then from Lemma 2 the points of $S^{k+1}(\sigma)$ are also sorted in the same order concerning each coordinate, and the three dimensional maxima problem on $S^{k+1}(\sigma)$ is reduced into the two dimensional maxima problem. Given a set $S$ of points in $E^2$ and a point $p$, if we can determine whether $p$ is the maxima of $S \cup \{p\}$ in O(1) time, then the three dimensional maxima problem can be solved in $O(|S^{k+1}(\sigma)|)$ time by scanning $S^{k+1}(\sigma)$ in the ascending order of $p_1$. But, in real, determining whether $p$ is the maxima of $S \cup \{p\}$ needs $O(|S|)$ time in

two dimensional case, and hence the three dimensional maxima problem requires $O(|S^{k+1}(\sigma)|^2)$ time.

The third idea is the use of Theorem 2. By this theorem, the two dimensional maxima problem is reduced into the one dimensional maxima problem, that is, the maximum problem of integers. Clearly, given a set $S$ of points in $E^1$ and a point $p$, we can determine whether $p$ is the maxima of $S \cup \{p\}$ in O(1) time, by keeping the maximum of $S$. Therefore, the original three dimensional maxima problem is solved in time $O(|S^{k+1}(\sigma)|) = O(|D^k|)$.

We have a data structure $\mathrm{bc}[\sigma_1 \ldots \sigma_s, 0 \ldots n, 1 \ldots d]$ to enumerate $S_p^k$. $\mathrm{bc}[\sigma, i, j]$ specifies the position of the first $\sigma$ in $A_j[i+1 \ldots n]$. If $\sigma$ does not exist in $A_j[i+1 \ldots n]$, $\mathrm{bc}[\sigma, i, j] = n+1$. Therefore $\mathrm{bc}[\sigma, i, j]$ store the position of the $[p_1(\sigma), p_2(\sigma), \ldots, p_d(\sigma)]$'s in $S_p$. For $d = 3$ (constant), bc needs $O(ns)$ space.

We have another data structure $\mathrm{cc}[0 \ldots n, 1 \ldots 3]$ to exclude superfluous points from $S^{k+1}(\sigma_x)$. For any pair of two points $p$ and $q$ (in $S^{k+1}$) which are not independent, by Theorem 2 for at least one coordinate $i$, $p_i = q_i$. Assume $p \in S$ precedes $q$ in the lexicographic order, that is, $p_1 < q_1$ or $p_1 = q_1, p_2 \leq q_2$. There are four cases. (1) $p_1 = q_1, p_2 = q_2$. Since $p$ precedes $q$ in the lexicographic order, $p_3 \leq q_3$. Then $q$ is not the minima and has to be removed from $S^{k+1}(\sigma_x)$. (2) $p_1 = q_1, p_2 < q_2$. If $p_3 \leq q_3$, then $q$ is not the minima. Otherwise, $q$ is the minima. (3) $p_1 < q_1, p_2 = q_2$. If $p_3 \leq q_3$, then $q$ is not the minima. Otherwise, $q$ is the minima. (4) $p_1 < q_1, p_3 = q_3$. If $p_2 \leq q_2$, then $q$ is not the minima. Otherwise, $q$ is the minima. For case (2), we have $\mathrm{cc}[0 \ldots n, 1]$ to keep track of the minimum of $p_3$ by far. For case (3), we use $\mathrm{cc}[0 \ldots n, 2]$, and for case (4), we use $\mathrm{cc}[0 \ldots n, 3]$.

We have an array $\mathrm{dd}[0 \ldots d]$ for storing the position of dominants, and the parent pointer and the ordering number according to the coordinate $x_2$. It needs $O(d)$ space. Another array $\mathrm{d}[0 \ldots ns, 1 \ldots 3]$ is a working area for storing $S^{k+1}$.

ALG A$(n, s, A_1, A_2, A_3)$

1. for $1 \leq x \leq s$ and $1 \leq i \leq 3$ do $\mathrm{bc}[\sigma_x, n, i] = n+1$
   for $j = n-1$ to 0
   for $1 \leq x \leq s$ and $1 \leq i \leq 3$ do $\mathrm{bc}[\sigma_x, j, i] = \mathrm{bc}[\sigma_x, j+1, i]$
   for $1 \leq i \leq 3$ do $\mathrm{bc}[A_i[j+1], j, i] = j+1$
2. put [0,0,0] into $D^0$ $k = 0$ $S^{k+1} = \phi$
   while $D^k$ not empty do

2.1 for all points $p$ of $D^k$ do
   $S^{k+1} = S^{k+1} \cup p(\Sigma)$
   the parent of $p(\sigma)$ is set to $p$

2.2 for all points $p$ of $D^k$ do
   for $1 \leq x \leq s$ and $1 \leq y \leq s$ do if $p(\sigma_x) \prec p(\sigma_y)$ then remove $p(\sigma_y)$ from $S^{k+1}$.

2.3 for $1 \leq x \leq s$ do
   for all points $j$ of $S^{k+1}(\sigma_x)$ do $\mathrm{cc}[j_1, 1] = \mathrm{cc}[j_2, 2] = \mathrm{cc}[j_3, 3] = n+1$

radix sort $S^{k+1}(\sigma_x)$
for all points $j$ of $S^{k+1}(\sigma_x)$
while $j_1$ and $j_2$ are the same as the previous $j_1$ and $j_2$
remove $j$ from $S^{k+1}(\sigma_x)$
end-while
if $j_2 \geq$cc$[j_3, 3]$ then remove $j$ from $S^{k+1}(\sigma_x)$ and cc$[j_3, 3] = j_2$
if $j_3 \geq$cc$[j_2, 2]$ then remove $j$ from $S^{k+1}(\sigma_x)$ and cc$[j_2, 2] = j_3$
if $j_3 \geq$cc$[j_1, 1]$ then remove $j$ from $S^{k+1}(\sigma_x)$ and cc$[j_1, 1] = j_3$

2.4 merge sort $\{S^{k+1}(\sigma) \mid \sigma \in \Sigma\}$ into $D^{k+1}$ according to the coordinate $x_1$

2.5 merge sort $\{S^{k+1}(\sigma) \mid \sigma \in \Sigma\}$ according to the coordinate $x_2$ (to compute the rank)

2.6 merge sort $\{S^{k+1}(\sigma) \mid \sigma \in \Sigma\}$ according to the coordinate $x_3$ (to compute the rank)

2.7 $k = k + 1$

end-while

3. pick an 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 of Algorithm

**Theorem 4.** The algorithm A correctly computes the LCS of strings $A_1, A_2$ and $A_3$.

**Proof.** By step 2.1, $S^{k+1}$ is precisely constructed. From Theorem 2 and 3, in step 2.2 and 2.3 the points in $S^{k+1}$ which is not the minima of $S^{k+1}$ are surely excluded.

**Theorem 5.** The algorithm A requires time of $O(ns+ds^2)$, 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 bc table, which takes time $O(ns)$. In Step 2, the outer loop repeats for $l$ times. Step 2.1 loops for $|D^k|$ times. Therefore, there should be $|D_1| + |D_2| + \cdots + |D^l| = d$ executions to compute $S = S \cup p(\Sigma)$. Since $|p(\Sigma)| = s$, step 2.1 requires at most $O(ds)$ time. Similarly, step 2.2 takes $O(ds^2)$ time, and step 2.3 takes $O(ds)$ time. Step 2.4, 2.5 and 2.6 take $O(s \log s|D^k|)$ time, since each level of merges requires $O(ds)$ time and the level of merges is $\log s$. And hence $O(ds \log s)$ time is required for step 2.4, 2.5 and 2.6 over the whole algorithm. After all, step 2 takes $O(ds^2)$ time. Step 3 takes $O(l)$ time.

## 5 Experimental Results

We have run experiments for the algorithm in the paper on uniform random strings over alphabet of size $s$=4. Programs are written in C, and tests are run on a Sun SPARC station ELC using the time command. Table 1 shows the running time of two LCS algorithms, the naive dynamic programming algorithm (DP) and our algorithm (A).

Table 1. The running time (s) of DP, and the maximum $A_{max}$, average $A_{ave}$ and minimum $A_{min}$ running time of our algorithm for 20 different test strings of length $n$.

| $n$ | 100 | 200 | 300 | 400 | 500 | 600 | 700 |
|---|---|---|---|---|---|---|---|
| DP | 6 | 54 | 189 | 423 | 826 | 1516 | 2423 |
| $A_{max}$ | 1 | 6 | 17 | 45 | 79 | 142 | 241 |
| $A_{ave}$ | 0.1 | 4.9 | 16.2 | 42.8 | 75.4 | 135.2 | 222.3 |
| $A_{min}$ | 0 | 4 | 15 | 39 | 71 | 123 | 213 |

## 6 Extensions

For the higher dimension, $d > 3$, the different method for solving the LCS problem may be needed. One method is to use the maxima algorithm, whose performance is presented by the following theorem.

**Theorem 6** [6]. The maxima of a set of $n$ points in $E^d$, $d \geq 2$, can be obtained in time $O(n \log^{d-2} n) + O(n \log n)$.

Using the result of the previous chapters, the LCS problem of dimension $d$ (that is, for $d$ strings) can be reduced to the maxima problem of dimension $d - 2$.

## 7 Conclusion

The LCS problem has been studied by a number of researchers and its complexity has been improved in diffirent respects. We have presented a better solution when the alphabet size is small and the number of strings are more than two. Computational experiments have been done for the case of $s = 4$, which corresponds to the DNA sequence. For the case of $s = 20$, that is, the case where the protein which consists of 20 kinds of amino acids is investigated, the number of dominant matches tends to be small, and the influence of the factor $O(s^2)$ is likely to be relatively negligible. Regarding the case of $d \geq 4$, more careful consideration is needed.

## Acknowledgment

This work is supported in part by the Grand-in-Aid for Scientific Research on Priority Areas, "Genome Informatics", of the Ministry of Education, Science and Culture of Japan.

## References

[1] Apostolico, A. and C. Guerra, The longest common subsequence problem revisited, *Algorithmica*, Vol.2, 1987, pp.315-336.

[2] Chin, F. Y. L. and C. K. Poon, A fast algorithm for computing longest common subsequences of small alphabet size, *J. of Info. Proc.*, Vol.13, No.4, 1990, pp.463-469.

[3] Hirschberg, D. S., A linear space algorithm for computing maximal common subsequences, *Comm. ACM*, Vol.18, 1975, pp.341-343.

[4] Hirschberg, D. S., Algorithms for the longest common subsequence problem, *J. ACM*, Vol.24, 1977, pp.664-675.

[5] Hunt, J. W. and T. G. A. Szymanski, A fast algorithm for computing longest common subsequences, *Comm. ACM*, Vol.20, 1977, pp.350-353.

[6] Kung, H. T., F. Luccio, and F. P. Preparata, On finding the maxima of a set of vectors, *J. ACM*, Vol.22, No.4, 1975, pp. 469-476.