Available online at www.sciencedirect.com

SCIENCE DIRECT®

Information Processing Letters 94 (2005) 131–135

Information Processing Letters

www.elsevier.com/locate/ipl

An almost-linear time and linear space algorithm for the longest common subsequence problem [☆]

J.Y. Guo [*], F.K. Hwang

Department of Applied Mathematics, National Chiaotung University, Hsinchu, Taiwan, ROC 30500

Received 12 July 2004; received in revised form 2 December 2004

Available online 27 January 2005

Communicated by Wen-Lian Hsu

Abstract

There are two general approaches to the longest common subsequence problem. The dynamic programming approach takes quadratic time but linear space, while the nondynamic-programming approach takes less time but more space. We propose a new implementation of the latter approach which seems to get the best for both time and space for the DNA application.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Algorithms; Primal-dual algorithm; Longest common subsequence

1. Introduction

Mutations in DNA arise naturally in an evolution process. These mutations include substitutions, insertions and deletions of nucleotides, leading to "editing" of DNA texts. A sequence comparison of two DNA sequences attempts to align the two sequences to minimize a function of these mutations. The most commonly used function is the so-called edit distance first introduced by Levenshtein [5] which simply counts the number of mutations. If substitutions are not allowed, then the alignment minimizing the edit distance will produce a longest common subsequence (LCS) of the two sequences. Note that the LCS problem had been studied by mathematicians for general sequences long before the edit distance was introduced for DNA sequences.

Assume that both sequences are of $\mathrm{O}(n)$ length. Needleman and Wunsch [6] gave an $\mathrm{O}(n^2)$ time and $\mathrm{O}(n^2)$ space dynamic programming algorithm for the LCS problem. Hirschberg [2] improved to $\mathrm{O}(n)$ space by using a divide-and-conquer technique. Later, Hunt and Szymanski [4], and Hirschberg [3], both noticed that not all steps in the dynamic-programming procedure need to be processed and they proposed more efficient nondynamic-programming algorithms. Hunt and Szymanski's algorithm was improved by Apos-

[☆] Research partially supported by ROC National Science council grant NSC 90-2115-M-009-007.

[*] Corresponding author.
E-mail address: davidguo@math.nctu.edu.tw (J.Y. Guo).

tolico [1] to require $O(n \log n)$ time and $O(n+l)$ space, where l denotes the number of matches between two sequences. Hirschberg's algorithm requires $O(Ln)$ time and $O(n+Ln)$ space, where L is the length of an LCS. Pevzner and Weterman [7] recognized that these algorithms can be cast into a primal-dual set-up. The derived primal-dual algorithm, as presented by Pevzner and Waterman, takes $O(l+Ln)$ time and $O(l+Ln)$ space. In this paper we give an $O(nL)$ time and $O(n)$ space implementation of the primal-dual algorithm.

2. The primal-dual algorithm

Let $I=\{I_1, I_2, \ldots, I_m\}$ and $J=\{J_1, J_2, \ldots, J_n\}$ denote two DNA sequences when $I_i, J_j \in \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$. Define $\mathcal{P}=\{(i,j)\colon I_i=J_j\}$. Assume $m=O(n)$. Then typically, $|\mathcal{P}|=O(n^2)$. This is the case if each nucleotide independently has probability $p_{\mathrm{A}}, p_{\mathrm{C}}, p_{\mathrm{G}}, p_{\mathrm{T}}$ of being A, C, G, T, respectively. We will also denote $\mathcal{P}=\{p_1, p_2, \ldots, p_l\}$ where each p_k is a pair (i_k, j_k). The partial order $\prec$ is defined by

$$p_x \prec p_y \quad \text{if } i_x < i_y,\ j_x < j_y.$$

The conjugate partial order $\prec^*$ is defined by

$$p_x \prec^* p_y \quad \text{if } i_x \leqslant i_y,\ j_x \geqslant j_y.$$

Let $\sqsubset$ denotes the partial order such that

$$p_x \sqsubset p_y \quad \text{if either } p_x \prec p_y \quad \text{or } p_x \prec^* p_y.$$

Pevzner and Waterman proved that $\sqsubset$ is a linear order $p_1 \sqsubset p_2 \sqsubset \cdots \sqsubset p_l$. Note that $|\sqsubset| = |\mathcal{P}| = l$ which is typically $O(n^2)$.

The algorithm, as presented in [7], assigns p_1, $p_2, \ldots$ one at a time (in order) to sets $C_1, C_2, \ldots$ such that the elements in a given C_k can be linearly ordered in $\prec^*$. Suppose that $p_1, p_2, \ldots, p_u$ have been assigned to $C_1, C_2, \ldots, C_v$. Let $p_1^*, p_2^*, \ldots, p_v^*$ denote the $\prec^*$-maximum elements of $C_1, C_2, \ldots, C_v$, respectively. Let k, $1 \leqslant k \leqslant v$, be the minimum index such that $p_k^* \prec^* p_{u+1}$. Assign p_{u+1} to C_k. If no such k exists, assign p_{u+1} to C_{v+1}. We also set a counter $b(p_{u+1})$ such that

$$b(p_{u+1}) = \begin{cases} 0 & \text{if } k=1, \\ p_{k-1}^* & \text{if } 2 \leqslant k \leqslant v, \\ p_v^* & \text{if } k \text{ does not exist.} \end{cases}$$

Note that if $b(p_{u+1}) \neq 0$, then $b(p_{u+1}) \not\prec^* p_{u+1}$. Suppose $p_1, p_2, \ldots, p_l$ are assigned to $C_1, C_2, \ldots, C_L$. Then L is the length of an LCS. An LCS can be backtracked from any element in C_L by using the b function. Once an LCS is identified, a corresponding (nonunique) alignment can be obtained by filling in between p_k and p_{k+1} the unmatched nucleotides from both sequence in an arbitrary order as long as being consistent with each sequence order.

The following example, taken from [7], illustrates the algorithm.

$$I = \overset{I_1}{\mathrm{T}}\overset{I_2}{\mathrm{G}}\overset{I_3}{\mathrm{C}}\overset{I_4}{\mathrm{A}}\overset{I_5}{\mathrm{T}}\overset{I_6}{\mathrm{A}}, \qquad J = \overset{J_1}{\mathrm{A}}\overset{J_2}{\mathrm{T}}\overset{J_3}{\mathrm{C}}\overset{J_4}{\mathrm{T}}\overset{J_5}{\mathrm{G}}\overset{J_6}{\mathrm{A}}\overset{J_7}{\mathrm{T}},$$

$$\mathcal{P} = \Big\{\overset{p_1}{(1,2)}, \overset{p_2}{(1,4)}, \overset{p_3}{(1,7)}, \overset{p_4}{(2,5)}, \overset{p_5}{(3,3)}, \overset{p_6}{(4,1)}, \overset{p_7}{(4,6)}, \overset{p_8}{(5,2)}, \overset{p_9}{(5,4)}, \overset{p_{10}}{(5,7)}, \overset{p_{11}}{(6,1)}, \overset{p_{12}}{(6,6)}\Big\}$$

$$\sqsubset\colon p_3, p_2, p_1, p_4, p_5, p_7, p_6, p_{10}, p_9, p_8, p_{12}, p_{11}.$$

For example, $p_3 \sqsubset p_2$ since $p_3 \prec^* p_2$, while $p_1 \sqsubset p_4$ since $p_1 \prec p_4$.

The assignment of p_{u+1}, $u=0,1,\ldots,11$, and $b(p_{u+1})$ are given in Table 1.

To find an LCS, we can start from p_{12} to obtain $p_{12} \succ p_9 \succ p_5 \succ p_1$ or from p_{10} to obtain $p_{10} \succ p_7 \succ p_5 \succ p_1$. Using the former, an optimal alignment can be

−TGCAT−A−
AT−C−TGAT

It takes $O(n+l)$ time and space to construct $\mathcal{P}$ and $O(l \log l)$ time to $\sqsubset$-order $\mathcal{P}$. It takes $O(lL)$ time and $O(l+L)$ space to construct $C_1, \ldots, C_L$.

Table 1

	C_1					C_2			C_3		C_4	
	p_3	p_2	p_1	p_6	p_{11}	p_4	p_5	p_8	p_7	p_9	p_{10}	p_{12}
$b(p_{u+1})$	0	0	0	0	0	p_1	p_1	p_6	p_5	p_5	p_7	p_9

3. An $O(nL)$ time and $O(n)$ space implementation

We construct a table X with 5 rows marked by j, A, C, G, T and $n+1$ columns marked by $n, n-1, \dots, 1, 0$ (the indices of J). Column n is empty. If index n is of nucleotide N, then column $n-1$ has entry n in row N and copies the other entries from column n. In general, if index j is of nucleotide N, the column $j-1$ has entry j in row N and copies the other entries from column j.

For example, for $J =$ ATCTGAT see Table 2.

It is easily verified that the entries in each row are nonincreasing in j. Next we construct a table Y with $L+1$ columns (L is unknown at the beginning) marked by $C_0, C_1, \dots, C_L$, and 6 rows marked by j, i, A, C, G, T. Along with table Y, we also set up a backtrack function b. At the beginning, only the C_0 column is filled with entries $0, 0, A(0), C(0), G(0), T(0)$, the last four entries from table X. Then we proceed with the indices of I one by one in order to construct Y. Suppose index 1 is of nucleotide N. Inspect row N in Y and we find only one index $T(0)$. Fill column C_1 with entries $T(0), 1, A(T(0)), C(T(0)), G(T(0)), T(T(0))$, and set $b(1, T(0)) = (0, 0)$.

Suppose we are dealing with index y of nucleotide N where C_k is the large x such that C_x is nonempty. By our construction, entries in row j of Y are increasing (easily observed after we finish describing the implementation). Hence entries in row A, C, G, T are nondecreasing. Inspect row N which, say, has entries $n_0 \leqslant n_1 \leqslant \cdots \leqslant n_k$ for $k \leqslant L$. For each n_i in the order from large to small, we do the following:

Let j_w denotes the value of j in column C_w, $0 \leqslant w \leqslant k$. Compare n_k with $j_k, j_{k-1}, \dots$ until the first column $C_{w(k)}$ such that $j_{w(k)} < n_k$. We fill the column $C_{w(k)+1}$ (or replace its entries) with $n_k, y, A(n_k), C(n_k), G(n_k), T(n_k)$. Set $b(y, n_k) = (i, j)$ where (i, j) is from $C_{w(k)}$. In general, suppose n_z has just filled the column $C_{w(z)+1}$ with $z, y, A(n_z), C(n_z), G(n_z), T(n_z)$. Let n_v be the next $n_i < n_z$. We compare n_v with $j_{w(z)}, j_{w(z)-1}, \dots$ until $C_{w(v)}$ is found. Set $b(y, n_v) = (i, j)$ where (i, j) is from $C_{w(v)}$.

We demonstrate this procedure by the example

$$\underset{\text{index}}{I} : \underset{1\,2\,3\,4\,5\,6}{\text{TGCATA}} \qquad \underset{\text{index}}{J} : \underset{1\,2\,3\,4\,5\,6\,7}{\text{ATCTGAT}}$$

We will fill in Y column by column until a column needs to be replaced, then we draw a new Y with the new column (see Tables 3 and 4).

Table 2

j	7	6	5	4	3	2	1	0
A	–	–	6	6	6	6	6	1
C	–	–	–	–	–	3	3	3
G	–	–	–	5	5	5	5	5
T	–	7	7	7	4	4	2	2

Table 3

	C_0	C_1	C_2	C_0	C_1	C_2	C_3	C_0	C_1	C_2	C_3	C_4
j	0	2	5	0	2	3	6	0	1	3	6	7
i	0	1	2	0	1	3	4	0	4	3	4	5
A	1	6	6	1	6	6	–	1	6	6	–	–
C	3	3	–	3	3	–	–	3	3	–	–	–
G	5	5	–	5	5	5	–	5	5	5	–	–
T	2	4	7	2	4	4	7	2	2	4	7	–

$b(1,2) = (0,0)$, $b(3,3) = (1,2)$, $b(5,7) = (4,6)$, $b(2,5) = (1,2)$, $b(4,6) = (3,3)$, $b(4,1) = (0,0)$.

Table 4

	C_0	C_1	C_2	C_3	C_4	C_0	C_1	C_2	C_3	C_4
j	0	1	2	4	7	0	1	2	4	6
i	0	4	5	5	5	0	6	5	5	6
A	1	6	6	6	–	1	6	6	6	–
C	3	3	3	–	–	3	3	3	–	–
G	5	5	5	5	–	5	5	5	5	–
T	2	2	4	7	–	2	2	4	7	7

$b(5,4) = (3,3)$, $b(6,6) = (5,4)$, $b(5,2) = (4,1)$, $b(6,1) = 0$.

Finally, take a pair (i, j) from any C_L column, we can trace an LCS with length L through the b function. In the above example, (6, 6) is a pair in C_4. From $b(6, 6) = (5, 4)$, $b(5, 4) = (3, 3)$, $b(3, 3) = (1, 2)$, we obtain the LCS: (I_1, J_2), (I_3, J_3), (I_5, J_4), (I_6, J_6). If we start from the pair (5, 7), then we have (I_1, J_2), (I_3, J_3), (I_4, J_6), (I_5, J_7).

We now prove that this procedure is indeed an implementation of the primal-dual algorithm. Note that we process the pairs in $\mathcal{P}$ in the lexicographical order of (i, j). So pairs with the same i, called the i-batch, are processed consecutively.

Suppose we are processing the i-batch, and $C_1, \ldots, C_k$ are nonempty. Let $(i_1, j_1), \ldots, (i_k, j_k)$ be the maximal pair in $C_1, \ldots, C_k$, respectively. Then $j_1 < j_2 < \cdots < j_k$.

It suffices to prove $j_w < j_{w+1}$. If (i_w, j_w) is processed before (i_{w+1}, j_{w+1}), then

$$i_w \leqslant i_{w+1} \quad \text{and} \quad j_w < j_{w+1}$$

or (i_{w+1}, j_{w+1}) would be assigned to C_w. If (i_w, j_w) is processed afterwards, and (i'_w, j'_w) was the maximal pair of C_w when (i_{w+1}, j_{w+1}) was processed, then

$$i_w \leqslant i'_w \leqslant i_{w+1} \quad \text{and} \quad j_w \leqslant j'_w < j_{w+1}.$$

Note that all pairs (i', j') processed before the i-batch have $i' < i$. Hence an i-pair can either ${}^*{\succ}(i', j')$, or be noncomparable, but not smaller. More specifically $(i, j)^*{\succ}(i_h, j_h)$ if and only if $j \leqslant j_h$. So an i-pair (i, j) joins C_h if and only if

$$j_{h-1} < j \leqslant j_h$$

and if $j > j_h$, then (i, j) starts a new C_{k+1}. Thus pairs in the i-batch are partitioned into several intervals where pairs in the same interval go to the same C_h. Also note that i-pairs are always comparable in $\prec^*$ since $j_1^* < j_2^* < \cdots < j_g^*$ implies

$$(i, j^*)^*{\succ}(i, j_2^*)^*{\succ} \cdots {}^*{\succ}(i, j_g^*).$$

So we only need to assign one pair (i, j) in each interval h to C_h where j is minimal among all i-pairs in the interval. It is easily verified that the (i, j) pair in column C_h of table Y is indeed the maximal pair (i_h, j_h) of C_h. So the entry in row N and column C_h gives the minimal index $x > j_h$ of a nucleotide of type N. Therefore, if N is the next nucleotide to be processed, then all the j-values of the maximal pairs in $C_1, \ldots, C_k$, (C_0 gives the overall minimum j) are provided by row N.

We now check the time complexity of this implementation. Table X can be constructed in $O(n)$ time. To construct the dynamic table Y, we need to go through the $O(n)$ elements of I. Since the entries in both row j and row N are ordered, starting from comparing the maximal entries of both row, each comparison eliminates one entry from further comparisons. Since there are at most $2L$ entries in the two rows, it takes $O(L)$-time to locate the entries $\{n_i\}$ of row N. Inserting the column of n_i (and possibly deleting a column) takes constant time. The backtrack function needs to be updated at most L times, and it takes constant time to update it. So processing each element of I takes $O(L)$ time, and the construction of table Y takes $O(nL)$ time. We have an $O(nL)$ time algorithm. It is also easily seen that tables X and Y can be constructed in $O(n)$ space.

4. Conclusions

For the LCS problem, the dynamic programming approach requires quadratic time but linear space, while the nondynamic-programming approach requires $O(n \log n)$ time or $O(Ln)$ time, which is almost linear when the length of an LCS is small compared to n, but more than linear space. We gave a nondynamic-programming implementation with $O(Ln)$ time and $O(n)$ space, efficient in both time and space.

Although our presentation is for a DNA sequence, the implementation is valid for any general sequence with, say, p alphabets. If p is treated as a variable, then the time complexity would be $O(n(L + p))$ and the space complexity $O(np)$. We may also drop the assumption that both sequences are of lengths of $O(n)$ order. If the lengths of the two sequences, $m < n$, are not equal, then either the time complexity would be $O(mp + nL)$ and the space complexity $O(mp)$, or m and n are interchanged in the above complexities.

References

[1] A. Apostolico, Improving the worst-case performance of the Hunt–Szymanski strategy for the longest common subsequence of two strings, Inform. Process. Lett. 23 (1986) 63–69.

[2] D.S. Hirschberg, A linear space algorithm for computing maximal common subsequences, Comm. ACM 18 (1975) 341–343.

[3] D.S. Hirschberg, Algorithms for the longest common subsequence problem, J. ACM 24 (1977) 664–675.
[4] J.W. Hunt, T.G. Szymanski, A fast algorithm for computing longest common subsequences, Comm. ACM 20 (1977) 350–353.
[5] V.I. Levenshtein, Binary codes capable of correcting deletions, insertions and reversals, Soviet Phys. Dokl. 6 (1966) 707–710.
[6] S.B. Needleman, C.D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, J. Mol. Biol. 48 (1970) 443–453.
[7] P.A. Pevzner, M.S. Waterman, Generalized sequence alignment and duality, Adv. Appl. Math. 14 (2) (1993) 139–171.