A New Algorithm for the Longest Common Subsequence Problem

Xuyu Xiang[a,b], *Dafang Zhang*[a], *Jiaohua Qin*[a,b]
[a]*School of Computer & Communication, Hunan University, Changsha, Hunan China, 410082*
[b]*Department of Computer, Hunan City University, Changsha, Hunan China, 413000*
xyuxiang@163.com, dfzhang@hnu.cn, qinjiaohua@163.com

Abstract

In order to find the longest common subsequence (LCS) as soon as possible, we, with the method of match pairs, propose the new algorithm of the sequence of DNA, which is efficient both in time and in space on the basis of the improved dynamic programming theorem.

1. Introduction

Mutations in DNA arise naturally in an evolution process. These mutations, which lead to the "editing" of DNA texts, include substitutions, insertions and deletions of nucleotides. The comparison of two DNA sequences attempts to align those two sequences to get the function these mutations. The most commonly used function is the so-called edit distance first introduced by Levenshtein[6], which simply counts the number of mutations. If substitutions are not allowed, the alignment minimizing the edit distance will produce a longest common subsequence (LCS) of the two sequences. And 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 O(n) length. Needleman and Wunsch[7] gave an O(n^2) time and O(n^2) space dynamic programming algorithm for the LCS problem. Hirschberg[2] improved it to O(n) space by using a divide-and-conquer technique. Later, Hunt and Szymanski[5], and Hirschberg[3], all 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 Apostolico[1] to 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(L·n) time and O(n+L·n) space, where L is the length of an LCS. Pevzner and Weterman[8] recognized that these algorithms can be performed by a primal-dual set-up. The derived primal-dual algorithm[4], as presented by Pevzner and Waterman, takes O(l+L·n) time and O(l+L·n) space. we propose a new algorithm, whose implementation takes an O(n·L) time and O(n) space.

2. The Improved Dynamic Programming Algorithm

The single stranded DNA is viewed as a linear sequence $a_1a_2\ldots a_n$ of nucleotide. The sequence is called the primary structure. Each a_i is identified with one of four basic nucleotides: A, C, G, T.

Definition 1[9] (Longest Common Subsequence)

An DNA sequence can be described as a string of the alphabet {A,C,G,T}. Given two DNA sequences $\boldsymbol{I}=\{i_1, i_2, \ldots, i_m\}$ and $\boldsymbol{J}=\{j_1, j_2, \ldots, j_n\}$, when i_i, $j_j \in$ {A,C,G,T}. the LCS problem is to find a largest subsequence $\boldsymbol{Z}=\{z_1, z_2, \ldots, z_L\}$, when $z_i \in$ {A,C,G,T}, satisfies $1 \le k \le L$, $z_k = i_{i_k} = j_{j_k}$ and $i_1 < i_2 < \cdots < i_L$, $j_1 < j_2 < \cdots < j_L$.

A brute-force approach to solve the LCS problem is to enumerate all subsequences of $\boldsymbol{I}$ and check each subsequence to see if it is also a subsequence of $\boldsymbol{J}$, keeping track the process of the discovery of the longest subsequence. Each subsequence of $\boldsymbol{I}$ corresponds to a subset of the indices {1, 2, ..., m} of $\boldsymbol{I}$. There are 2^m subsequences of $\boldsymbol{I}$, which make the approach require exponential time, so that it is impractical for long sequences.

The LCS problem has an optimal-substructure property, as the following theorem shows. As we shall see, the natural classes of subproblems correspond to pairs of "prefixes" of the two input sequences. To be precise, given a sequence $\boldsymbol{I}=\{i_1, i_2, \ldots, i_m\}$, we define the ith prefix of $\boldsymbol{I}$, for i= 0, 1, ..., m, as $\boldsymbol{I}_i=\{i_1, i_2, \ldots, i_i\}$ and $\boldsymbol{I}_0$ is the empty sequence.

Theorem 1[9] (Optimal substructure of an LCS)

Let $\boldsymbol{I}=\{i_1, i_2, \ldots, i_m\}$ and $\boldsymbol{J}=\{j_1, j_2, \ldots, j_n\}$ be sequences, and let $\boldsymbol{Z}=\{z_1, z_2, \ldots, z_L\}$ be any LCS of $\boldsymbol{I}$ and $\boldsymbol{J}$.

(1).If $i_m = j_n$, then $z_L = i_m = j_n$ and $\boldsymbol{Z}_{L-1}$ is an LCS of $\boldsymbol{I}_{m-1}$ and $\boldsymbol{J}_{n-1}$.

0-7695-3073-7/07 $25.00 © 2007 IEEE
DOI 10.1109/CIS.Workshops.2007.209

IEEE computer society

(2).If $i_m \neq j_n$, then $z_L \neq i_m$ implies that $\boldsymbol{Z}$ is an LCS of $\boldsymbol{I}_{m-1}$ and $\boldsymbol{J}$.

(3).If $i_m \neq j_n$, then $z_L \neq j_n$ implies that $\boldsymbol{Z}$ is an LCS of $\boldsymbol{I}$ and J_{n-1}.

Similarly, the natural classes of subproblems can also correspond to pairs of " Suffixes " of the two input sequences. To be precise, given a sequence $\boldsymbol{I}=\{i_1, i_2, ..., i_m\}$, we define the ith Suffixe of $\boldsymbol{I}$, for $i = 0, 1, ..., m$, as $\boldsymbol{I}_i=\{i_i, i_{i+1}, ..., i_m\}$ and $\boldsymbol{I}_0$ is the empty sequence.

Theorem 2 (The improved dynamic programming algorithm)

If $\boldsymbol{I}=\{i_1, i_2, ..., i_m\}$ and If $\boldsymbol{J}=\{j_1, j_2, ..., j_n\}$ are sequences and let $\boldsymbol{Z}=\{z_1, z_2, ..., z_L\}$ be some LCS of $\boldsymbol{I}$ and $\boldsymbol{J}$.

(1) If $i_1=j_1$ then $z_1=i_1$ and $\boldsymbol{Z}_2$ is an LCS of $\boldsymbol{I}_2$ and $\boldsymbol{J}_2$,

(2) If $i_1\neq j_1$ then $z_1\neq i_1$ and $\boldsymbol{Z}$ is an LCS of $\boldsymbol{I}_2$ and $\boldsymbol{J}$,

(3) If $i_1\neq j_1$ then $z_1\neq j_1$ and $\boldsymbol{Z}$ is an LCS of $\boldsymbol{I}$ and $\boldsymbol{J}_2$.

Proof: (1) If $z_1\neq i_1$, we could add $i_1=j_1$ to $\boldsymbol{Z}$ to get an LCS of length L+1. By contradiction, $z_1=i_1=j_1$. $|Z_2|=L$-1 and it must be an LCS of $\boldsymbol{I}_2$ and $\boldsymbol{J}_2$. In this case, $\boldsymbol{Z}_2$ is an LCS. if not , $\exists\ \boldsymbol{W}$, $|\boldsymbol{W}|>L$-1 , and it is a CS(common sequence) of $\boldsymbol{I}_2$ and $\boldsymbol{J}_2$, then we get a CS of $\boldsymbol{I}$ and $\boldsymbol{J}$ of the length greater than L by appending $i_1=j_1$. so It is a contradiction.

(2) If $z_1\neq i_1$, $\boldsymbol{Z}$ is a CS of $\boldsymbol{I}_2$ and $\boldsymbol{J}$. If $\exists\ \boldsymbol{W}$, $|\boldsymbol{W}|>L$, and it is a CS, then $\boldsymbol{W}$ would be a CS of $\boldsymbol{I}$ and $\boldsymbol{J}$. and it is also a contradiction.

(3) The same Proof as Proof (2) by reversing $\boldsymbol{I}$ and $\boldsymbol{J}$.

So the approach to find the LCS of $\boldsymbol{I}$ and $\boldsymbol{J}$ is as following:

(1) if $i_1=j_1$ find LCS of $\boldsymbol{I}_2$ and $\boldsymbol{J}_2$,

(2) if $i_1\neq j_1$

a) find LCS of $\boldsymbol{I}_2$ and $\boldsymbol{J}$,

b) find LCS of $\boldsymbol{I}$ and $\boldsymbol{J}_2$,

and take the larger one of 'a)' or 'b)'.

Thus we start with small problem, find LCS and develop our solution:

Let LCS($\boldsymbol{I}_i$, $\boldsymbol{J}_j$) be LCS of $\boldsymbol{I}_i$ and $\boldsymbol{J}_j$. Let

$\boldsymbol{I}_1=\{i_1,i_2,\ldots,i_m\}$, $\boldsymbol{I}_2=\{i_2,i_3,\ldots,i_m\}$, …, $\boldsymbol{I}_m=\{i_m\}$;

$\boldsymbol{J}_1=\{j_1,j_2,\ldots,j_n\}$, $\boldsymbol{J}_2=\{j_2,j_3,\ldots,j_n\}$, …, $\boldsymbol{J}_n=\{j_n\}$.

then

$$LCS(I_i,J_j)=\begin{cases}\{i_i\}\cup LCS(I_{i-1},J_{j-1}) & \text{if } i_i=j_j\\ \max\{LCS(I_{i-1},J_j),LCS(I_i,J_{j-1})\} & \text{others}\end{cases}$$

Algorithm 1: computing LCS($\boldsymbol{I}$, $\boldsymbol{J}$)

1 LCS($\boldsymbol{I}_i$, ϕ)=ϕ;
2 LCS(ϕ, $\boldsymbol{J}_j$)= ϕ;
3 for i=1 to m
4 for j=1 to n
5 if i_i=j_j
6 LCS($\boldsymbol{I}_i$, $\boldsymbol{J}_j$)={i_i}+LCS($\boldsymbol{I}_{i-1}$, $\boldsymbol{J}_{j-1}$)
7 else if length(LCS($\boldsymbol{I}_{i-1}$, $\boldsymbol{J}_j$))>=length(LCS($\boldsymbol{I}_i$, $\boldsymbol{J}_{j-1}$))
8 LCS($\boldsymbol{I}_i$, $\boldsymbol{J}_j$)=LCS($\boldsymbol{I}_{i-1}$, $\boldsymbol{J}_j$)
9 else
10 LCS($\boldsymbol{I}_i$, $\boldsymbol{J}_j$)=LCS($\boldsymbol{I}_i$, $\boldsymbol{J}_{j-1}$)
11 output LCS($\boldsymbol{I}_m$, $\boldsymbol{J}_n$)

Running Time=O(mn) because each table entry takes O(1) time and O(mn) space.

The following example, taken from [8], illustrates the algorithm in table 1.

$\boldsymbol{I}$= ATCTGAT, $\boldsymbol{J}$= TGCATA,

Table 1. **illustration of the improved dynamic programming algorithm**

	J	T	G	C	A	T	A
I	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
A	ϕ	ϕ	ϕ	ϕ	A	←A	A
T	ϕ	T	←T	←T	↑A	□AT	←AT
C	ϕ	↑T	↑T	□TC	←TC	↑AT	↑AT
T	ϕ	T	↑T	↑TC	↑TC	□TCT	←TCT
G	ϕ	↑T	□TG	↑TC	↑TC	↑TCT	↑TCT
A	ϕ	↑T	↑TG	↑TC	□TCA	↑TCT	□TCTA
T	ϕ	T	↑TG	↑TC	↑TCA	□TCAT	↑TCTA

3. The New Algorithm

From the Definition 1 and the Theorems, We can see that subproblem class is corresponding to match pair of the two input sequences. Given two DNA sequences $\boldsymbol{I}=\{i_1, i_2, ..., i_m\}$ and $\boldsymbol{J}=\{j_1, j_2, ..., j_n\}$, when i_i,j_j□{A,C,G,T}.

Definition 2 Let $\boldsymbol{I}=\{i_1, i_2, ..., i_m\}$ and $\boldsymbol{J}=\{j_1, j_2, ..., j_n\}$ denote two DNA sequences, let $i_i=j_j$, i=1,2,…,m; j=1,2,…,n. Define $\boldsymbol{P}=\{(i, j): i_i =j_j \}$. then (i, j) is the match pair.

This is the case that each nucleotide independently has probability p_A, p_C, p_G, p_T of being A, C, G, T, respectively. We will also denote $\boldsymbol{P}$ = $\{p_1, p_2, \ldots, p_l\}$ where each p_k is a pair (i_k, j_k).

Definition 3 Given two match pairs p_x, p_y, if $i_x<i_y$, $j_x<j_y$, define $p_x < p_y$. p_y is the subsequence of the p_x, or p_x is the pre-sequence of the p_y.

Definition 4 If $p_x < p_y$, $p_x < p_z$, p_y, p_z□$\boldsymbol{P}$, and $p_z< p_y$ doesn't exist, define the specific order <* to be $p_x<^* p_y$, p_y is the direct subsequence of the p_x, or p_x is the direct pre-sequence of the p_y.

Algorithm 2: computing LCS($\boldsymbol{I}$, $\boldsymbol{J}$)

1 construct $\boldsymbol{P}=\{(i, j): i_i =j_j \}$ in the order of $\boldsymbol{I}$;
2 construct $\boldsymbol{P}_1$, …, $\boldsymbol{P}_l$, where l denotes the number of matches between two sequences and $\boldsymbol{P}_k=\{p_z: p_z<^*$

p_{z+1}, z=k, k+1, …, l}; we do not write out the sets of the match pairs which have direct pre-sequence, because their sets must be subsets of their direct pre-sequence.

3 LCS($\boldsymbol{I}$, $\boldsymbol{J}$)= max{$\boldsymbol{P}_i$: length($\boldsymbol{P}_i$), i=1,2,…,l}.

The following example, taken from [8], illustrates the algorithm.

$\boldsymbol{I}$ = ATCTGAT, $\boldsymbol{J}$ = TGCATA,

$\boldsymbol{P}$={(1,4), (1,6), (2,1), (2,5), (3,3), (4,1), (4,5), (5,2), (6,4), (6,6), (7,1), (7,5)}.

$\boldsymbol{P}_1$={(1,4), (2,5), (6,6)}; $\boldsymbol{P}_2$={ (1,6)}; $\boldsymbol{P}_3$={ (2,1), (3,3), (4,5), (6,6)}; $\boldsymbol{P}_4$={ (2,1), (3,3), (6,4), (7,5)}; $\boldsymbol{P}_5$={ (4,1), (5,2), (6,4), (7,5)}; $\boldsymbol{P}_6$={ (4,1), (5,2), (6,6)}; $\boldsymbol{P}_7$={ (7,1)}.

$\boldsymbol{P}_3$ and $\boldsymbol{P}_4$ are the (2,1) sets, because (2,1) has two direct subsequences. The (3,3) set is not writen, because (3,3) has direct pre-sequence (2,1), the set is a sub-set of the (2,1) set. then we do not write these sets of the match pairs which have direct pre-sequence.

$\boldsymbol{P}_3$={ (2,1), (3,3), (4,5), (6,6)}, or $\boldsymbol{P}_4$={ (2,1), (3,3), (6,4), (7,5)} or $\boldsymbol{P}_5$={ (4,1), (5,2), (6,4), (7,5)} is the longest, then LCS($\boldsymbol{I}$, $\boldsymbol{J}$)= $\boldsymbol{P}_3$ or $\boldsymbol{P}_4$ or $\boldsymbol{P}_5$. Using the former, an optimal alignment can be

–TGCAT–A–

AT–C–TGAT

We can induce the following theorem from the definitions and the algorithm.

Theorem 3 Let $\boldsymbol{I}$={I_1, I_2, ..., I_m} and $\boldsymbol{J}$={J_1, J_2, ..., J_n} denote two DNA sequences, LCS($\boldsymbol{I}$, $\boldsymbol{J}$)= max{$\boldsymbol{P}_i$: length($\boldsymbol{P}_i$), i=1,2,…,l}.

Proof: Let one LCS($\boldsymbol{I}$, $\boldsymbol{J}$)={ $\boldsymbol{P}_1$, $\boldsymbol{P}_2$, …, $\boldsymbol{P}_L$ }, L=length(LCS($\boldsymbol{I}$, $\boldsymbol{J}$)), then p_k <* p_{k+1}, k=1,2,…, L-1. if not, ∃ $\boldsymbol{W}$, |$\boldsymbol{W}$|>L, and it is a CS(common sequence) of $\boldsymbol{I}$ and $\boldsymbol{J}$, we get a CS set of $\boldsymbol{I}$ and $\boldsymbol{J}$ of the length which is greater than L by appending $p_{k'}$ and $p_{k'}$ □$\boldsymbol{P}$, so the $\boldsymbol{W}$ set has three pairs p_k, $p_{k'}$, p_{k+1} and p_k <* p_k <* p_{k+1}. It is a contradiction.

4. An O(nL) time and O(n) space implementation

We construct a table 2 with 5 rows marked by j, A, C, G, T and n columns marked by 1, … , n−1, n, the number that a single nucleotide appears most frequently in the sequence. And the number of the columns stand for the position of a single nucleotide in the sequence, when it appears for the nth time. If it doesn't appear, fill no number in it.

For example, for $\boldsymbol{J}$ = ATCTGAT see Table 2.

We now check the time complexity of this implementation. Table 2 can be constructed in O(n) time and O(n) space.

In order to constructe $\boldsymbol{P}$, we need to go through the O(m) elements of $\boldsymbol{I}$. The number of the position of each element and the respective number of the rows in which the corresponding element appears in the table 1 can construct match pairs, which can be stored in linear space in O(l) space,

Table 2. **The position and frequency of the nucleotide in the sequence of $\boldsymbol{J}$**

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

Assume m=O(n). It takes O(n+l) time and space to construct $\boldsymbol{P}$. It takes O(lL) time and O(l+L) space to construct $\boldsymbol{P}_1$, …, $\boldsymbol{P}_l$.

5. 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 propose a nondynamic programming implementation with O(n+l) time and space whose algorithm is 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 are not equal and m<n, then either the time complexity would be O(mp+nL) and the space complexity O(mp), or m and n are interchanged in the complexities above.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (NSFC No. 60473031, 60673155), Scientific Research Fund of Education Department of Hunan Province, China (No.07C192).

References

[1] A. Apostolico, G. Rozenberg, A. Salomaa (Eds.), *Handbook of Formal Languages*, Springer-Verlag, Berlin, 1997, pp. 361–398.

[2] A. Apostolico, M.J. Atallah (Ed.), *Handbook of Algorithms and Theory of Computation*, CRC, Boca Raton, FL, 1998.
[3] L. Bergroth, H. Hakonen, T. Raita, *A survey of longest common subsequence algorithms*, SPIRE, A Coruña, Spain, 2000, pp. 39–48.
[4] J.Y. Guo , F.K. Hwang, "An almost-linear time and linear space algorithm for the longest common subsequence problem", *Information Processing Letters*, 94 (2005) 131 – 135
[5] D.S. Hirschberg, "Algorithms for the longest common subsequence problem", *J. ACM*, 24 (1977) 664–675.
[6] D.S. Hirschberg, A. Apostolico, Z. Galil (Eds.), *Pattern Matching Algorithms*, Oxford University Press, Oxford, 1997, pp. 123–141.
[7] D. Maier, "The complexity of some problems on subsequences and supersequences", *J. ACM* 25 (1978) 322–336.
[8] W.J. Masek, M.S. Paterson, "A faster algorithm computing string edit distances", *J. Comput. System Sci*,. 20 (1980) 18–31.
[9] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, http://www.cs.fsu.edu/~cop4531/slideshow/chapter16/16-3.html.