Available online at www.sciencedirect.com
SCIENCE DIRECT®
Information Processing Letters 93 (2005) 249–253
Information Processing Letters
www.elsevier.com/locate/ipl

A fast algorithm for computing a longest common increasing subsequence

I-Hsuan Yang [a], Chien-Pin Huang [a], Kun-Mao Chao [a,b,*]

[a] *Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan*
[b] *Institute of Networking and Multimedia, National Taiwan University, Taipei, Taiwan*

Received 21 May 2003; received in revised form 9 August 2004

Available online 2 December 2004

Communicated by P.M.B. Vitányi

Abstract

Let $A = \langle a_1, a_2, \ldots, a_m \rangle$ and $B = \langle b_1, b_2, \ldots, b_n \rangle$ be two sequences, where each pair of elements in the sequences is comparable. A common increasing subsequence of A and B is a subsequence $\langle a_{i_1} = b_{j_1}, a_{i_2} = b_{j_2}, \ldots, a_{i_l} = b_{j_l} \rangle$, where $i_1 < i_2 < \cdots < i_l$ and $j_1 < j_2 < \cdots < j_l$, such that for all $1 \leqslant k < l$, we have $a_{i_k} < a_{i_{k+1}}$. A longest common increasing subsequence of A and B is a common increasing subsequence of the maximum length. This paper presents an algorithm for delivering a longest common increasing subsequence in $O(mn)$ time and $O(mn)$ space.

Keywords: Algorithms; Computational biology; Longest common subsequence; Longest increasing subsequence

1. Introduction

The longest common subsequence (LCS) problem and the longest increasing subsequence (LIS) problem are both very classical problems in computer science. By using the dynamic programming technique, the LCS problem can be solved in $O(mn)$ time. Knuth [3] posed the question of whether a sub-quadratic algorithm for the LCS problem exists. Masek and Paterson [10] gave an algorithm that runs in $O(mn/\log n)$ time, where $n \leqslant m$ and the subsequences are drawn from a set of bounded size. Szymanski [13] proposed an $O((n+m)\log(n+m))$ algorithm for the special case in which no element appears more than once in an input sequence. Interested readers can refer to a recent survey by Bergroth et al. [1].

On the other hand, there is a rich history for the longest increasing subsequence problem as well, e.g., see [6,11]. Schensted [12] and Knuth [8] gave an $O(n \log n)$ time algorithm for this problem where the input is an arbitrary sequence of n numbers. For a special case in which the input sequence is a permutation of $\{1, 2, \ldots, n\}$, Hunt and Szymanski [7], and

* Corresponding author.
E-mail address: kmchao@csie.ntu.edu.tw (K.-M. Chao).

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2004.10.014

Bespamyatnikh and Segal [2] gave algorithms that run in $O(n \log\log n)$ time.

In this paper, we consider the longest common increasing subsequence (LCIS) problem, whose goal is to find a maximum-length common subsequence of the two sequences such that the subsequence is increasing. This problem might arise in the situation where we wish to find the longest set of MUMs whose sequences occur in ascending order in three or more genomic sequences [4,5]. Formally speaking, let A and B be two sequences $A = \langle a_1, a_2, \ldots, a_m\rangle$ and $B = \langle b_1, b_2, \ldots, b_n\rangle$, where $m \geqslant n$ and each pair of elements in the sequences is comparable. A common increasing subsequence of A and B is a subsequence $\langle a_{i_1} = b_{j_1}, a_{i_2} = b_{j_2}, \ldots, a_{i_l} = b_{j_l}\rangle$, where $i_1 < i_2 < \cdots < i_l$ and $j_1 < j_2 < \cdots < j_l$, such that for all $1 \leqslant k < l$, we have $a_{i_k} < a_{i_{k+1}}$. A longest common increasing subsequence of A and B is a common increasing subsequence of the maximum length. A straightforward $O(mn^2)$-time algorithm for this problem is to sort the shorter sequence, and then find a longest common subsequence among the two sequences and the sorted sequence. Here we give an algorithm that solves the longest common increasing subsequence problem in $O(mn)$ time and $O(mn)$ space.

2. The algorithm

This algorithm utilizes a folklore algorithm (see [9] for more details) which runs in $O(n \log n)$ time and $O(n)$ space for the LIS problem. Define an array $L[k]$ to be the smallest ending number of an increasing subsequence of length k, where $1 \leqslant k \leqslant n$. Assume that each entry in L is initially an infinite value. For each position, perform a binary search to update L and make a backtracking link to the former number. The largest k such that $L[k]$ contains non-infinite value is the length of a longest increasing subsequence. Tracing back from $L[k]$ along with the link we established before will deliver a longest increasing subsequence. Its running time and space is $O(n \log n)$ and $O(n)$, respectively. In this paper, the array L will be explored between every pair of the prefix substrings of the two sequences. Instead of using binary search, we use linear search by a given starting point to update the array L.

Let $L^i_j[k]$ be the smallest ending number of a longest common increasing subsequence of length k between the sequences $\langle a_1, a_2, \ldots, a_i\rangle$ and $\langle b_1, b_2, \ldots, b_j\rangle$, where $1 \leqslant i \leqslant m$ and $1 \leqslant j, k \leqslant n$. In order to keep the backtracking information, we need two more variables. Specifically, let $L\_index^i_j[k]$ $(1 \leqslant i \leqslant m, 1 \leqslant j, k \leqslant n)$ record the index pair (x, y) such that $L^i_j[k]$ is the element a_x and b_y, and let $Prev[i, j]$ $(1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n)$ record the index pair (x, y) such that there is a link from (i, j) to (x, y). A link will be made in the function *LIS_insert* and used in the backtracking process.

It can be shown that the difference between arrays L^i_j and L^{i-1}_j as well as between L^i_j and L^i_{j-1} is at most one entry, which will be proved later in Lemma 1. We therefore use a variable χ^i_j for keeping the index of the difference between L^i_j and L^{i-1}_j as follows.

$$\chi^i_j = \begin{cases} l & \text{if } L^i_j[l] \neq L^{i-1}_j[l] \text{ for some } l, \\ -1 & \text{if } L^i_j = L^{i-1}_j. \end{cases}$$

Variable χ^i_j is used while L^{i-1}_j and L^i_{j-1} are merged in the *mismatch* case.

For $i \geqslant 2$ and $j \geqslant 2$, L^i_j can be derived from L^{i-1}_j and L^i_{j-1}. There are two cases for computing L^i_j: the *match* case if $a_i = b_j$, and the *mismatch* case if $a_i \neq b_j$. In the *match* case, we do the following:

$$L^i_j = \text{Insert } a_i \text{ into } L^{i-1}_j,$$

and in the *mismatch* case, we do the following:

$$L^i_j = merge\big(L^i_{j-1}, L^{i-1}_j\big).$$

Here, *inserting* a number into L refers to the process of calling the function *LIS_insert*(L, *L_index*, *Prev*, a, p, i, j), which inserts an element a into L, makes a link $Prev[i, j]$ to the former number in L, and returns the insertion index. Variable p stores the index of the last

LIS_insert(array L, array *L_index*, array *Prev*, element a, integer p, integer i, integer j)

1. $x := p$
2. **while** $(L[x] < a)$ **do**
3. $\quad x := x + 1$
4. $L[x] := a$
5. $L\_index[x] := (i, j)$
6. **if** $(x \neq 1)$ **then**
7. $\quad Prev[i, j] := L\_index[x - 1]$
8. **return** x

Fig. 1. The function *LIS_insert*.

inserted point, and is used as the starting index for the next linear search (see Fig. 1).

In the *merging* process, $L_j^i = merge(L_{j-1}^i, L_j^{i-1})$ is defined to be for any $k \geqslant 1$, $L_j^i[k] = \min(L_{j-1}^i[k], L_j^{i-1}[k])$. Our algorithm first assigns L_j^{i-1} to L_j^i, and then compares $L_j^i[\chi_{j-1}^i]$ with $L_{j-1}^i[\chi_{j-1}^i]$:

$$L_j^i := L_j^{i-1},$$
$$L_j^i[\chi_{j-1}^i] := \min\{L_j^i[\chi_{j-1}^i], L_{j-1}^i[\chi_{j-1}^i]\}.$$

The resulting L_j^i is equal to $merge(L_{j-1}^i, L_j^{i-1})$, which will be proved later.

The algorithm for computing a longest common increasing subsequence is given in Fig. 2. One observation is that after finishing computing L_j^i, the array L_j^{i-1} can be discarded. Therefore, all the L_j in different rows can use the same memory space and thus avoid the time for copying L_j^{i-1} to L_j^i. Besides, χ_j^i can also be recycled to save space. Thus, L_j^i, χ_j^i, and $L\_index_j^i$ are denoted by L_j, χ, and L_index_j, respectively. Once this algorithm reaches line 17 of Fig. 2, the last non-infinite element of array L_n stores the LCIS's ending number. A longest common increasing subsequence can be delivered by tracing back along the links.

3. Correctness

Assume that L_j^{i-1} and L_{j-1}^i keep the correct ending numbers. For convenience, L_j^i is first assigned as L_j^{i-1}, and then combined with L_{j-1}^i. There are two cases: the mismatch case and the match case.

3.1. The mismatch case

In the mismatch case ($a_i \neq b_j$), we do the following:

$$L_j^i = merge\big(L_{j-1}^i, L_j^{i-1}\big).$$

INPUT: Two sequences $A = \langle a_1, a_2, \ldots, a_m\rangle$ and $B = \langle b_1, b_2, \ldots, b_n\rangle$, where $m \geqslant n$.

```
    //initialization
1.  for j = 1 to n do
2.    for k = 1 to n do
3.      L_j[k] := ∞
4.  for j = 1 to m do
5.    for k = 1 to n do
6.      Prev[j, k] := (−1, −1)
    //main program
7.  for i = 1 to m do
8.    χ := −1, p := 1    //default value
9.    for j = 1 to n do
10.     if (a_i = b_j) then    //the match case
11.       χ, p := LIS_insert(L_j, L_index_j, Prev, a_i, p, i, j)    //return the insertion position
12.     else    //the mismatch case
13.       if ((χ ≠ −1) and (L_{j−1}[χ] < L_j[χ])) then
14.         L_j[χ] := L_{j−1}[χ]
15.       else
16.         χ := −1
    //recover a longest common increasing subsequence in reverse order
17. x := the largest x such that L_n[x] ≠ ∞ (if x does not exist, print "NULL" and exit )
18. (y1, y2) := L_index_n[x]
19. print a_{y1}
20. while (Prev[y1, y2] ≠ (−1, −1)) do
21.   (y1, y2) := Prev[y1, y2]
22.   print a_{y1}
```

Fig. 2. The algorithm for computing a longest common increasing subsequence.

Any common increasing subsequence of length k in the sequences $\langle a_1, a_2, \ldots, a_i\rangle$ and $\langle b_1, b_2, \ldots, b_{j-1}\rangle$ or in the sequences $\langle a_1, a_2, \ldots, a_{i-1}\rangle$ and $\langle b_1, b_2, \ldots, b_j\rangle$ will still exist in the sequences $\langle a_1, a_2, \ldots, a_i\rangle$ and $\langle b_1, b_2, \ldots, b_j\rangle$. By the definition of $L^i_j[k]$, we have to choose the "smallest" one. Moreover, because $a_i \neq b_j$, there will be no more new elements added into the list, which means that a common increasing subsequence of length k will still be a common subsequence of length exactly k in the sequences $\langle a_1, a_2, \ldots, a_i\rangle$ and $\langle b_1, b_2, \ldots, b_j\rangle$.

Now, we prove that L^i_j computed by our algorithm is the same as *merge*(L^i_{j-1}, L^{i-1}_j).

Lemma 1. *For any $i, j > 1$ in the dynamic-programming table, there is at most one different entry between L^i_j and L^{i-1}_j (or between L^i_j and L^i_{j-1}). If the different entry is $L^i_j[k]$, then $L^i_j[k] < L^{i-1}_j[k]$ (or $L^i_j[k] < L^i_{j-1}[k]$).*

Proof. Between L^i_j and L^{i-1}_j, if a_i matches one of the letter in sequence $\langle b_1, b_2, \ldots, b_j\rangle$, we will insert at most one a_i into L^{i-1}_j and cause at most one entry smaller. Similar arguments hold for L^i_j and L^i_{j-1}. □

Lemma 2. *For any $i, j > 1$ in the dynamic-programming table, there are at most two different entries between L^i_{j-1} and L^{i-1}_j. If there are two differences, then one of them is at index χ^i_{j-1} such that $L^i_{j-1}[\chi^i_{j-1}] < L^{i-1}_j[\chi^i_{j-1}]$, and the other difference is at index λ, where $L^i_{j-1}[\lambda] > L^{i-1}_j[\lambda]$. If there is only one difference, it is at index χ^i_{j-1}.*

Proof. By Lemma 1, there is at most one different entry between L^{i-1}_j and L^{i-1}_{j-1}, and if this happens, let us assume $L^{i-1}_{j-1}[\lambda] > L^{i-1}_j[\lambda]$. There is also at most one different entry between L^{i-1}_{j-1} and L^i_{j-1}, and if this happens, let us assume $L^{i-1}_{j-1}[\chi^i_{j-1}] > L^i_{j-1}[\chi^i_{j-1}]$. If $\chi^i_{j-1} \neq \lambda$, then there are two differences and the lemma holds. If $\chi^i_{j-1} = \lambda$, then there is at most one different entry, which is recorded by χ^i_{j-1}. □

By Lemma 2, while merging L^{i-1}_j with L^i_{j-1}, we need to compare at most two different entries. There are four cases. First, if $L^{i-1}_{j-1} = L^i_{j-1}$ (i.e., $\chi^i_{j-1} = -1$) and $L^{i-1}_{j-1} = L^{i-1}_j$, then $L^{i-1}_j = L^i_{j-1}$. So we have $L^i_j = L^{i-1}_j = \mathit{merge}(L^i_{j-1}, L^{i-1}_j)$. Second, if $L^{i-1}_{j-1} = L^i_{j-1}$ (i.e., $\chi^i_{j-1} = -1$) and $L^{i-1}_{j-1} \neq L^{i-1}_j$ where $L^{i-1}_{j-1}[k] > L^{i-1}_j[k]$ (Lemma 1), then for all k, $L^{i-1}_j[k] \leqslant L^{i-1}_{j-1}[k] = L^i_{j-1}[k]$. It follows $L^i_j = L^{i-1}_j = \mathit{merge}(L^i_{j-1}, L^{i-1}_j)$. Third, if $L^{i-1}_{j-1} \neq L^i_{j-1}$ (i.e., $\chi^i_{j-1} \neq -1$) where $L^i_{j-1}[\chi^i_{j-1}] < L^{i-1}_{j-1}[\chi^i_{j-1}]$ and $L^{i-1}_{j-1} = L^{i-1}_j$, then for all $k \neq \chi^i_{j-1}$, $L^{i-1}_j[k] = L^{i-1}_{j-1}[k] = L^i_{j-1}$ and $L^i_{j-1}[\chi^i_{j-1}] < L^{i-1}_{j-1}[\chi^i_{j-1}] = L^{i-1}_j[\chi^i_{j-1}]$. We set $L^i_j = L^{i-1}_j$ and $L^i_j[\chi^i_{j-1}] = L^i_{j-1}[\chi^i_{j-1}]$. Thus L^i_j will be equal to *merge*(L^i_{j-1}, L^{i-1}_j). Finally, consider the case where $L^{i-1}_{j-1} \neq L^i_{j-1}$ (i.e., $\chi^i_{j-1} \neq -1$) and $L^{i-1}_{j-1} \neq L^{i-1}_j$. If there are one or two different entries between L^{i-1}_j and L^i_{j-1}, we let $L^i_j = L^{i-1}_j$ and $L^i_j[\chi^i_{j-1}] = \min(L^i_{j-1}[\chi^i_{j-1}], L^{i-1}_j[\chi^i_{j-1}])$. By Lemma 2, L^i_j is equal to *merge*(L^i_{j-1}, L^{i-1}_j). If there is no different entry between L^{i-1}_j and L^i_{j-1}, then $L^i_{j-1} = L^{i-1}_j$. Consequently, $L^i_j = L^{i-1}_j = \mathit{merge}(L^i_{j-1}, L^{i-1}_j)$.

3.2. The match case

In the match case ($a_i = b_j$), we do the following:

$L^i_j = \text{Insert } a_i \text{ into } L^{i-1}_j$.

For the two sequences $\langle a_1, a_2, \ldots, a_i\rangle$ and $\langle b_1, b_2, \ldots, b_j\rangle$, $a_i = b_j$ is the last "common" element. That means that it can be added to any common increasing subsequence with ending number smaller than a_i. Thus, inserting a_i into L^{i-1}_j will maintain the invariance of L^i_j. Whether a_i is already in L^{i-1}_j or not, the resulting L^i_j is correct because the insertion process will do nothing if the element is already in it.

Lemma 3. *In the same row L^i, the indices p of all insertion points are non-decreasing from the index $j = 1$ to $j = n$.*

Proof. Observe that, in the same row, every element to be inserted is equal to a_i. For all $k \geqslant 1$, from L^i_1 to L^i_n, by Lemma 1, each change between $L^i_j[k]$ and

$L^i_{j+1}[k]$ is always decreasing. Assume that the index of the last insertion point of L^i_j is p_1, it follows $L^i_j[1], L^i_j[2], \ldots, L^i_j[p_1 - 1]$ are smaller than a_i. In the next match case, this property still holds. Thus, the insertion point will not be in the range of 1 to $p_1 - 1$. □

Lemma 3 implies that we can perform linear search from the previously recorded index p.

4. Time and space complexity

There are two loops in the algorithm of Fig. 2: an inner loop **for** $j = 1$ **to** n (line 9) and an outer loop **for** $i = 1$ **to** m (line 7). During each execution of the outer loop body (the same index i), there are n steps with two cases: *mismatch* and *match*. In the *mismatch* case, only one comparison is required. In the *match* case, the procedure *LIS_insert* will do linear search from p to the insertion point and update p. By Lemma 3, p is non-decreasing for each outer loop, and $1 \leqslant p \leqslant n$. Therefore, for each outer loop, it takes $O(n)$ time. The overall time complexity is $O(mn)$.

As Fig. 2 shows, for all $1 \leqslant i \leqslant m$, L^i_j and $L\_index^i_j$ share the memory space L_j and $L\_index^i_j$, respectively, and thus requires $O(n^2)$ space. Variables $Prev[i, j]$ are used for the range $1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n$, which would require $O(mn)$ space. It follows that this algorithm runs in $O(mn)$ space.

5. Discussion

We presented an algorithm for computing a longest common increasing subsequence in quadratic time. Can one do it in sub-quadratic time?

Acknowledgements

We thank the reviewers for their helpful comments that improve the presentation of the paper. Kun-Mao Chao was supported in part by an NSC grant 92-2213-E-002-073 from the National Science Council, Taiwan.

References

[1] L. Bergroth, H. Hakonen, T. Raita, A survey of longest common subsequence algorithms, in: Proc. 7th Internat. Symp. on String Processing Information Retrieval (SPIRE'00), Spain, 2000, pp. 39–48.
[2] S. Bespamyatnikh, M. Segal, Enumerating longest increasing subsequences and patience sorting, Inform. Process. Lett. 76 (2000) 7–11.
[3] V. Chvatal, D.A. Klarner, D.E. Knuth, Selected combinatorial research problems, Technical Report, STAN-CS-72-292, Computer Science Department, Stanford University, 1972.
[4] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, S.L. Salzberg, Alignment of whole genomes, Nucleic Acids Res. 27 (1999) 2369–2376.
[5] A.L. Delcher, A. Phillippy, J. Carlton, S.L. Salzberg, Fast algorithms for large-scale genome alignment and comparison, Nucleic Acids Res. 30 (2002) 2478–2483.
[6] M.L. Fredman, On computing the length of longest increasing subsequences, Discrete Math. 11 (1975) 29–35.
[7] J. Hunt, T. Szymanski, A fast algorithm for computing longest common subsequences, Comm. ACM 20 (1977) 350–353.
[8] D.E. Knuth, Sorting and Searching, The Art of Computer Programming, vol. 3, Addison-Wesley, Reading, MA, 1973.
[9] U. Manber, Introduction to Algorithms—A Creative Approach, Addison-Wesley, Reading, MA, 1989.
[10] W.J. Masek, M.S. Paterson, A faster algorithm computing string edit distances, J. Comput. System Sci. 20 (1) (1980) 18–31.
[11] J. Matousek, E. Welzl, Good splitters for counting points in triangles, J. Algorithms 13 (1992) 307–319.
[12] C. Schensted, Longest increasing and decreasing subsequences, Canad. J. Math. 13 (1961) 179–191.
[13] T. Szymanski, A special case of the maximal common subsequence problem, Technical Report, TR-170, Computer Science Laboratory, Princeton University, 1975.