

Another Algorithm for Computing Longest Common Increasing Subsequence for Two Random Input Sequences

Yongsheng Bai *
Department of Biology
The University of Texas at Arlington
Box 19498
Arlington, TX 76019-0498, USA

Bob P. Weems
Department of Computer Science &
Engineering
The University of Texas at Arlington
Box 19015
Arlington, TX 76019-0015, USA

Abstract

We have previously developed several algorithms which deal with different input sequence scenarios respectively. In this paper, another sequential algorithm for computing Longest Common Increasing Subsequence for two random input sequences is reported. It runs in $O(m n \log p)$ time complexity and takes space of $O(2(m + n + 1) p)$, where m and n are the length for two input sequences respectively, and p is the length of the final Longest Common Increasing Subsequence. Parallelization allows us to further reduce time and space. This algorithm has significant applications in the area of Computational Biology in manipulating the genomic sequences.

Keywords: Algorithms; Longest increasing subsequence; Longest common subsequence; Longest common increasing subsequence.

1.0 Introduction

Bai and Weems [1][2] have previously published several algorithms in finding the *Longest Common Increasing Subsequence (LCIS)* among sequences. In this paper, the authors have presented another novel algorithm for computing LCIS.

The *longest upsequence* or *Longest Increasing Subsequence (LIS)* is the maximum-length increasing subsequence of the original sequence. A sequential algorithm has been developed by Gries [3] to identify the longest upsequence. In his sequential method, let $sequence[0 : n - 1]$ be the original sequence which contain n elements, where $n > 0$, the smallest value that ends an upsequence of length k of $sequence[0 : n - 1]$ is saved in a traceback sequence ($help[0 : k - 1]$). A detailed review about how the procedure of updating a traceback table works is illustrated by Bai and Weems [1]. Using this method, the execution time of the LIS calculation is $O(n)$ in the best case and to $O(n \log n)$ in the worst case.

The *Longest Common Subsequence (LCS)* is the maximum-length common subsequence for two or multiple sequences. Several LCS algorithms with different level of improvement on time and space are represented as the ones developed by Hirschberg [4][5], Hunt and Szymanski [6].

The *Longest Common Increasing Subsequence (LCIS)* is the maximum-length subsequence which must also meet both “common” and “non-decreasing” criteria. Like LIS and LCS, the LCIS can be varied. For example, if sequence Y_1 is 2, 0, 8, 7, 6, 7, 11, 2, 5 and sequence Y_2 is 0, 8, 4, 2, 6, 11, 3, 0; then either 2, 6, 11 or 0, 6, 11 can be the LCIS for these two sequences. As illustrated in the author’s previous papers, the LCIS can be found through designing different methods. This paper will address another approach of finding the LCIS for two random input sequences and illustrate the idea of extending LIS and LCS methods to solve the LCIS problem again.

2.0 Algorithm for Computing LCIS for two random input sequences

In this case, the algorithm was designed in a space-efficient manner. In particular, the LCIS for general input sequences can be solved with merge-free and anti-diagonal approaches. Specifically, the algorithm uses $m + n + 1$ tables. Each table (mtable[p][q][r]) used for slot [i][j] in the grid will also be used for all slots [i+k][j+k]. When slot [i+1][j+1] is filled based on the information of slot [i][j], a binary search method is used to update “mtable” if $X[i] = Y[j]$, where $X[i]$ and $Y[j]$ are snapshot values for two elements of two input sequences respectively. Otherwise, the elements at the same “level” of upper neighbor slot and left neighbor slot are merged through taking the smaller value. At the same time, the index value of this taken element is also carried to a new slot (In the design manner of this algorithm, the indices of elements in horizontal displayed sequence in the algorithm are taken). If the numbers of elements in two slots are different, the rest of elements in the longer sequence will be appended to the merged table. The grid is filled in an anti-diagonal manner, which is the scenario of being perpendicular to the traditional diagonals. The running time of this algorithm is $O(m n \log p)$ since the binary search is used in this case, and the space requirement is $O(2(m + n + 1) p)$ because each of the $m + n + 1$ tables needs $\theta(p)$ space, where p is the length of LCIS.

The traceback procedure is performed using the following method. First, the lower right corner slot is inspected. Next, the correspondent slot number for this element is found based on its index value. The slot number of its “predecessor” is identified by searching both sequences in reverse order and taking the first matched slot’s information, and recursively tracing up. All subsequent “predecessors” and the resulting “mtable” in the lower right corner slot consist of the final LCIS. The core pseudocode for the algorithm of dealing with this case is shown in Figure 1.

```

** mVector[0..m-1] and nVector[0..n-1] are two input sequences **

LCIS ()
0  nVector[w]=abs(rand())%100 +1 for i=0 to n-1           /*Generate the row sequence*/
1  mVector[w]=abs(rand())%100 +1 for i=0 to m-1           /*Generate the column sequence*/
2  i ← 0
   j ← 0
   m_Counter ← 0           /*Initialize the counter for counting the length of sequence*/
   p_Counter ← 1           /*Initialize the counter for counting the length of mtable*/
3  if n>m                   /*If the row sequence is longer*/
   begin
     repeat
3.1 if m_Counter ≤ m       /*If the column sequence has not been scanned through*/
     begin
       x ← m_Counter
       m_Column ← 0
       repeat
       if x=0 or m_Column =0           /*Start to fill in the marginal slots*/
       begin
         initialize margin elements
       endif
       else
       begin
         if X[x-1] = Y[m_Column-1]     /*If there is a matched value between two sequences*/
         begin
           binary_Search(x, m_Column) /*Find the insertion location in the mtable*/
         end
       end
       endif
       m_Column ← m_Column + 1
     end
     m_Counter ← m_Counter + 1
   end
   end

```

```

endif
else /*If there is not a matched element between two sequences at this position*/
begin
    merge(x, m_Column) /*Merge upper slot and left slot element into current slot*/
endelse
endelse
    x--
    m_Column++
until x<0
    m_Counter++
    am_Row ←m_Counter
endif
3.2 else /*If the column sequence has been scanned through*/
begin
3.2.1 if am_Row<n /*If the row sequence has not been scanned through*/
begin
    final_Row ←am_Row
    y ←0
repeat
if y=0 /*Start to fill in the marginal slots*/
begin
    initialize margin elements
endif
else
begin
if X[final_Row-1] =Y[y-1] /*If there is a matched value between two sequences*/
begin
    binary_Search(final_Row, y) /*Find the insertion location in the mtable*/
endif
else
begin
    merge(final_Row, y) /*Merge upper slot and left slot element into current slot*/
endelse
endelse
    final_Row--
    y++
until y≤m
    am_Row++
    final ←am_Row
endif
3.2.2 else /*If the row sequence has been scanned through*/
begin
if n-m =1
begin
    i ←am_Row
endif
else
begin
    i ←final
endelse
if d≤m

```



```

        endelse
        x--
        m_Columnn++
    until x<0
    m_Counter++
endif
4.2 else /*If the row sequence has been scanned through*/
begin
    if past≤m
    begin
        final_Row ←n
        s_Column ←past+1
        repeat
            if final_Row=0
            begin
                initialize all margin elements
            endif
            else
            begin
                if X[final_Row-1] = Y[s_Column-1]
                begin
                    binary_Search(final_Row, s_Column)
                endif
                else
                begin
                    merge(final_Row,s_Column)
                endelse
            endelse
            final_Row—
            s_Column++
            until final_Row<0 or s_Column>m
        repeat
        past++
    endif
endelse
    p_Counter++
    until p_Counter >m+n+1
endelse
5 if mTable[n][m][1].mdata = 999 /*If the final lower right corner element is not changed*/
begin
    length ←0
    no LCIS found
endif
else /*Start to trace back the LCIS*/
begin
    length ←mTable[n][m][1].kValue
    back_Trace()
endelse

```

Fig. 1: Finding LCIS for the random input sequences using merge-free and anti-diagonal approaches.

Both scenarios in which the number of rows are equal and less than, or larger than the number of columns cases in the above algorithm are considered. An example of finding LCIS of sequence 1 {5, 6, 1, 2, 3, 7} and sequence 2 {5, 6, 7, 1, 2, 3} and doing traceback is shown in Table 1.

Table 1. Finding LCIS for sequence 1 {5, 6, 1, 2, 3, 7} and sequence 2 {5, 6, 7, 1, 2, 3}

		5	6	7	1	2	3
	-1 999	-1 999	-1 999	-1 999	-1 999	-1 999	-1 999
5	-1 999	0 5	0 5	0 5	0 5	0 5	0 5
6	-1 999	0 5	0 5 1 6	0 5 1 6	0 5 1 6	0 5 1 6	0 5 1 6
1	-1 999	0 5	0 5 1 6	0 5 1 6	3 1 1 6	3 1 1 6	3 1 1 6
2	-1 999	0 5	0 5 1 6	0 5 1 6	3 1 1 6	3 1 4 2	3 1 4 2
3	-1 999	0 5	0 5 1 6	0 5 1 6	3 1 1 6	3 1 4 2	3 1 4 2 5 3
7	-1 999	0 5	0 5 1 6	0 5 1 6 2 7	3 1 1 6 2 7	3 1 4 2 2 7	3 1 4 2 5 3

Notes: 999 means that no common value occurs in that slot. -1 is the index of the element in the empty slot. If these values appear in a table slot, then the LCIS is empty up to this slot length. In each slot, the numbers in left column are indices; the numbers in right column are values. The grid is filled by anti-diagonals.

3.0 Summary

In this paper, the LCIS problem is solved using a novel, merge-free, and anti-diagonal filling approach. The algorithm can be parallelized to obtain improved running time using either MPI or p-thread methods. This algorithm is very useful in handling genomic sequences because of their random arranged features. Addressing the lower bound of running time for the algorithm in solving the LCIS problem is still an open issue.

4.0 References

- [1] Bai, Y. and B. P. Weems, “Finding Longest Common Increasing Subsequence for Two Different Scenarios of Non-random Input Sequences”. *Proc. of the 2005 International Conference on Foundations of Computer Science*, pp 362-366, 2005.
- [2] Bai, Y and B. P. Weems, “The Longest Common Increasing Subsequence Problem.” *Proc. of the 8th Joint Conference on Information Sciences*, pp 64-70. 2005.
- [3] D. Gries. “The Science of Programming.” *Springer-Verlag, New York*, 259–262, 1981.
- [4] D.S. Hirschberg. “Algorithms for the longest common subsequence problem.” *J. ACM*, 24 (4): 664–675, 1977.
- [5] D.S. Hirschberg. “A linear space algorithm for computing maximal common subsequences.” *Communications of the ACM*, 18 (6): 341–343, 1975.
- [6] J.W. Hunt and T.G. Szymanski. “A fast algorithm for computing longest common subsequences.” *Communications of the ACM*, 20 (5): 350–353, 1977.