AN IMPROVED ALGORITHM TO FIND THE LENGTH OF THE LONGEST COMMON SUBSEQUENCE OF TWO STRINGS*

Shufen Kuo
George R. Cross

Washington State University
Department of Computer Science
Pullman, WA 99164-1210

ABSTRACT

Let A and B be strings of common length n. Define $LLCS(A, B)$ to be the length of the longest common subsequence of A and B. Hunt and Szymanski presented an algorithm for finding $LLCS(A, B)$ with time complexity $O((r + n)logn)$, where r is the number of elements in the set $\{(i, j)|A[i] = B[j]\}$. In the worst case the algorithm has running time of $O(n^2logn)$. We present an improvement to this algorithm which changes the time complexity to $O(r + n(LLCS(A, B) + logn))$. Some experimental results show dramatic improvements for large n.

KEYWORDS

Analysis of Algorithms, Pattern Recognition, Artificial Intelligence

---
*

Supported in part by an equipment donation from Tektronix, Inc.

1. Introduction

The length of the longest common subsequence (abbreviated $LLCS$) of two strings strings can be used to measure the similarity between two strings. We are investigating algorithms to retrieve trademark names which are similar to a proposed trademark as part of a legal reasoning system which assesses trademark infringement.

There are several papers presenting algorithms to find the $LLCS$ of two strings. A generalization of the $LLCS$ problem is the string–to–string correction or string–editing problem. Wagner and Fischer[5] presented an algorithm for string–to–string correction problem in quadratic time and space, which can be applied to find the longest common subsequence of two strings. Hirschberg[1] presented a quadratic time and linear space algorithm. Hunt and Szymanski[2] presented a fast algorithm having running time of $O((r+n)logn)$ and space complexity $O(r+n)$, where r is the number of elements in the set $\{(i,j) \mid A[i] = B[j]\}$, n is the length of strings in question. Mukhopadhyay[3] also presented an algorithm with running time of the same complexity as Hunt and Szymanski's algorithm. On the average r is n^2/k , where $k = |\Sigma|$ and Σ is the alphabet. These algorithms are not very efficient when n is large and k is small. Nakatsu *et al.*[4] presented an $O(n(m-p))$ algorithm, where n and m are lengths of two strings, $m \leq n$, and p is the $LLCS$ of the two strings. When p is close to m, Nakatsu's algorithm runs in linear time.

We present a modified Hunt-Szymanski algorithm that shows a dramatic improvement in running time for large n. The organization of this paper is as follows: Section 2 fixes notation for the remainder of the paper, Section 3 reviews the original Hunt-Szymanski algorithm, Section 4 describes our improvements to the algorithm and Section 5 presents some experimental results.

2. Definitions and Notation

In this section, we define conventions, notation and terminology that will be used throughout this paper.

A string $C = c_1c_2...c_p$ is a *subsequence* of string $A = a_1a_2...a_m$ if C is formed by deleting $m - p$ (not necessarily adjacent) symbols from A. For example, *"cut"* is a subsequence of *"computer"*.

A string C is a *common subsequence* of strings A and B if C is a subsequence of A and also a subsequence of B. *CS* is the abbreviation of common subsequence.

A string C is a *longest common subsequence* of string A and string B if C is a common subsequence of A and B of maximal length. *LLCS(A,B)* is the abbreviation for the *length of the longest common subsequence* of the strings A and B. For example, $LLCS($*"wings"*, *"magics"*$) = 2$ with longest common subsequences *"is"* and *"gs."*

We denote the length of string A by $|A|$. $A[i]$ is the ith element of A and $A[i : j]$ denotes the substring $A[i]A[i+1] \dots A[j]$.

3. The Original Hunt-Szymanski Algorithm

Hunt and Szymanski[2] presented an algorithm for finding $LLCS(A, B)$. The Hunt-Szymanski algorithm has time complexity $O((r + n)logn)$, where r is the number of elements in the set $\{(i,j)|A[i] = B[j]\}$, and n is the length of strings. In the worst case, the algorithm has a running time of $O(n^2logn)$.

The key data structure needed by Hunt-Szymanski algorithm is an array of *threshold values* $T_{i,k}$ $(0 \leq i,\ k \leq n)$ defined by

$$T_{i,k} = min\{ j \mid A[1:i] \text{ and } B[1:j] \text{ contain a CS of length k} \}.$$

For example, given the strings $A = abcbdda$, $B = badbabd$, we have $T_{5,1} = 1$, $T_{5,2} = 3$, $T_{5,3} = 6$, $T_{5,4} = 7$ and $T_{5,5} =$ undefined. Once the array of threshold values $T_{i,k}$ is completely filled, the

$$LLCS(A, B) = max\{\, k \mid T_{n,k} \text{ is defined} \}.$$

It is clear that $T_{i,0} = 0$ for all $0 \leq i \leq n$, and $T_{0,k} =$ undefined for all $1 \leq k \leq n$. The following *Lemma* suffices to compute all the threshold values since we can get $T_{i+1,k}$ from $T_{i,k-1}$ and $T_{i,k}$ easily.

Lemma: For all $1 \leq i,\ k \leq n$

$$T_{i,k} = \begin{cases} min\{ j \mid A[i] = B[j] \text{ and } T_{i-1,k-1} < j \leq T_{i-1,k} \} \\ T_{i-1,k} \text{ if no such j exists} \end{cases}$$

Proof: See Hunt and Szymanski[2].

Using the *Lemma* we need only a vector $THRESH[0:n]$ to hold threshold values instead of a 2-dimensional array. First, the $THRESH$ vector is initialized as the first row of the T array, that is $T[0,k]$ $(0 \leq k \leq n)$. Once the ith row of the T array is filled up, *i.e.* $T[i,k]$ $(0 \leq k \leq n)$, we can overwrite the ith row to get the $(i+1)$th row by computing each $T_{i+1,k}$ from $T_{i,k-1}$ and $T_{i,k}$.

In order to search for characters in the string B which match $A[i]$, a list array $MATCHLIST[1:n]$ is needed for this search. $MATCHLIST[i] =< j_1, j_2, \ldots, j_p >$ such that $j_1 > j_2 > \ldots > j_p$, $A[i] = B[j_q]$ for $1 \leq q \leq p$, and p is the total number of characters in string B which is matched by $A[i]$. For the strings $A = abcbdda$, $B = badbabd$ the desired lists are

$$\begin{aligned}
MATCHLIST[1] &= <\ 5,\ 2\ > \\
MATCHLIST[2] &= <\ 6,\ 4,\ 1\ > \\
MATCHLIST[3] &= <> \\
MATCHLIST[4] &= MATCHLIST[2] \\
MATCHLIST[5] &= <\ 7,\ 3\ > \\
MATCHLIST[6] &= MATCHLIST[5] \\
MATCHLIST[7] &= MATCHLIST[1].
\end{aligned}$$

The Hunt-Szymanski algorithm is presented below in Figure 1. *Step 1* can be implemented by sorting each string while keeping track of each character's original position. The next step is to merge the sorted strings to create the $MATCHLIST$s.

Since the $THRESH$ vector is monotonically increasing we can utilize a binary search to implement the "find" operation (in *Step 3*) in time $O(logn)$. The time complexity is $O(rlogn + nlogn)$ because the total number of elements in $MATCHLIST$s is r.

```
    character string $A[1:n]$, $B[1:n]$;
    integer array $THRESH[0:n]$;
    list array $MATCHLIST[1:n]$;

{ Step 1: Build Linked Lists }

    for i := 1 step 1 until n do
      set $MATCHLIST[i] := < j_1, j_2, \ldots, j_p >$ such that
           $j_1 > j_2 > \ldots > j_p$ and
           $A[i] = B[j_q]$ for $1 \leq q \leq p$;

{Step 2: Initialize The $THRESH$ Array}

    $THRESH[0] := 0$;
    for i := 1 step 1 until n do
      $THRESH[i]$ := n + 1;   { $n + 1$ means undefined }

{Step 3: Compute Successive $THRESH$ Values}

    for i := 1 step 1 until n do
           { $THRESH[k] = T_{i-1,k}$ for all k }
      for j on $MATCHLIST[i]$ do
        begin
          find k such that $THRESH[k-1] < j \leq THRESH[k]$;
          if j < $THRESH[k]$ then
             $THRESH[k]$ := j;
        end;  { $THRESH[k] = T_{i,k}$ for all k }
      print the largest k such that $THRESH[k] \neq n + 1$
```

Figure 1: The Original Hunt-Szymanski Algorithm

4. The Improved Hunt-Szymanski Algorithm

Now we show how to improve the Hunt-Szymanski algorithm to have a running time of $O(r + n(LLCS(A, B) + logn))$. We assume string A and string B have the same length n. A central problem in the Hunt-Szymanski algorithm is that each list in the $MATCHLIST$ array must be in decreasing order. The following example explains this requirement.

Suppose

$$MATCHLIST[i] =< 52, 20, 17, 14, 13, 10, 8, 3, 1 >,$$

and before executing the ith iteration, the $THRESH$ array has the following contents:

index:	0	1	2	3	4	5	6	...
contents:	0	2	4	30	32	41	78	...

In *Step 3* of the Hunt-Szymanski algorithm, we want to find k such that

$$THRESH[k-1] < j \leq THRESH[k].$$

The first element, 52, in the ith list of $MATCHLIST$ fits $THRESH[6]$, and the next six elements in the ith list of $MATCHLIST$ all fit $THRESH[3]$. We want the smallest possible j, *i.e.* 8, to be in $THRESH[3]$ at the end of the process; thus, each list of $MATCHLIST$ should be in decreasing order.

The value in $THRESH[3]$ will be overwritten six times until the final value, the smallest, 8 is written to it. Thus, the algorithm is quite inefficient. If we keep the lists in increasing order then for each iteration it only takes at most one write for each slot in $THRESH$ array. Using the above example, we keep the list in increasing order:

$$< 1, 3, 8, 10, 14, 17, 20, 52, > .$$

The improved *Step 3* which we call *Step 3** is given in Figure 2.

For the above example, first 1 is written to $THRESH[1]$, then 3 is written to $THRESH[2]$, and then 8 is written to $THRESH[3]$. The third slot was only written once since 10, 13, 14, 17 and 20 are all less than or equal to *temp* and will be skipped.

In *Step 3**, we use linear search for finding k. However, k does not have to start from 0 for each j; the search starts from the slot in which previous write takes place. Each slot is scanned at most once for each i. Since $k \leq LLCS(A, B)$ holds for every k, and there are n iterations, we see that the inner loop is visited no more than $nLLCS(A, B)$ times. During this process the $MATCHLIST$ has been visited r times and the sort to build the $MATCHLIST$ needs $O(nlogn)$ time. Putting all of these together, we see that the improved algorithm has time complexity

$$O(r + nLLCS(A, B) + nlogn) = O(r + n(LLCS(A, B) + logn)).$$

{Step 3*: Compute Successive *THRESH* Values}

```
for i := 1 step 1 until n do
  begin
    temp := 0; k := 0;
    for each j in $MATCHLIST[i]$ do
      if j > temp then
         begin
           repeat k := k + 1
           until j $\leq THRESH[k]$;
           temp := $THRESH[k]$;
           $THRESH[k]$ := j
         end
  end;
```

Figure 2: Improved Section of Hunt-Szymanski Algorithm

5. Experimental Results

We performed some timings for Hunt-Szymanski Algorithm and the improved Hunt-Szymanski Algorithm for random inputs on a *Tektronix 4404* workstation (Motorola 68010-based) machine in Extended Common LISP. In Table 1, *time 1* and *time 2* are the mean running times for 100 pairs of random strings of length n using the original Hunt-Szymanski algorithm and then the improved Hunt-Szymanksi algorithm, respectively. We also report the corresponding standard deviations for these runs in the columns labelled *s.d.* The time recorded here is in internal time units of one thousandth of a second.

Table 1

Running Time Comparison in Thousandths of a Second

n	*time 1*	*s.d.*	*time 2*	*s.d.*
20	259	58	178	43
100	5,787	368	1,454	57
200	25,897	1,012	3,978	65
300	62,073	1,697	7,520	73
400	118,840	3,099	11,934	196

The data in the above table reveals that the new algorithm is very efficient when n is large.

References

(1) Hirschberg, D. S. June 1975. "A Linear space Algorithm for Computing Maximal common Subsequences". *Communications of the ACM*, vol.18 no.6; 341–343.

(2) Hunt, James W. and Szymanski, Thomas G. May 1977. "A Fast Algorithm for Computing Longest Common Subsequences". *Comm. ACM*, vol.20 no.5; 350–353.

(3) Mukhopadhyay, Amar. 1980. "A Fast Algorithm for the Longest–Common–Subsequence Problem". *Information Sciences*, vol.20; 69–82.

(4) Nakatsu, N. Kambayashi, Y. and Yajima, S. 1982. "A Longest Common Subsequence Algorithm Suitable for Similar Text Strings". *Acta Informatica*, vol.18; 171–179.

(5) Wagner, Robert A. and Fischer, Michael J. Jan. 1974. "The String–to–String Correction Problem". *J. ACM*, vol.21 no.1; 168–173.