1985

Improving the Worst Case Performance of the Hunt-Szymanski Strategy for the Longest Common Subsequence of Two Strings

Alberto Apostolico

Report Number:
85-542

TR # 542

IMPROVING THE WORST CASE PERFORMANCE OF THE HUNT-SZYMANSKI STRATEGY FOR THE LONGEST COMMON SUBSEQUENCE OF TWO STRINGS

ALBERTO APOSTOLICO

Computer Sciences Department
Purdue University
West Lafayette, Indiana 47907

ABSTRACT

Among the algorithms set up to date for finding the longest common subsequence of two strings, the one by Hunt and Szymanski exhibits the best known performance in favorable cases, but can be worse than any straightforward algorithm for a large variety of inputs. The new algorithm presented here pursues a schedule of primitive operations quite close to the one inherent to the Hunt-Szymanski strategy, but with substantially enhanced efficiency. In fact, the new algorithm improves on the former in two important respects. First, its worst case is never worse than linear in the product nm of the lengths of the two input strings. Second, its time bound does not always grow with the cardinality r of the set R of all pairs of matching positions of the input strings. Rather, it depends on the cardinality d of a specific subset of R, whose elements are called here *dominant matches*, and are elsewhere referred to as *minimal candidates*. This second improvement also appears of significance, since it seems that whenever r gets too close to mn, then this forces d to be linear in m. The new algorithm requires standard preprocessing, and makes use of finger-trees. In a forthcoming paper, it will be shown among other things that the same performance can be achieved with simpler and handier auxiliary data structures.

Key words and phrases: Design and analysis of algorithms, Longest common subsequence, Dictionaries , (a,b)-trees, Finger-trees, Efficient merging of linear lists.

1. Preliminaries.

We consider *strings* A, B, C, ... of *symbols* on an *alphabet* $\Sigma=(\sigma_1,\sigma_2, \cdots \sigma_s)$ of cardinality s. A string is identified by writing $A=a_1a_2...a_m$, with $a_i \in \Sigma$ $(i=1,2,...,m)$. The *length* of A is m. A string $C=c_1c_2...c_l$ is a *subsequence* of A if there is a mapping $F:[1,2,...,l] \rightarrow [1,2,...,m]$ such that $F(i)=k$ only if $c_i=a_k$ and F is monotone and strictly increasing.

Let $A=a_1a_2...a_m$ and $B=b_1b_2...b_n$ be two strings on Σ with $m \le n$. The string C is a *common subsequence* of A and B iff C is a subsequence of A and also a subsequence of B. The *longest common subsequence* (LCS, for short) *problem* for input strings A and B consists of finding a common subsequence C of A and B of maximum length. Note that C is not unique in general.

The search space where the LCS of A and B is sought is suitably represented by the integer matrix $L[1...m,1...n]$ where $L[i,j]$ $(1 \le i \le m, 1 \le j \le n)$ is the length of an LCS between $A[1:i]=A_i=a_1a_2...a_i$ and $B[1:j]=B_j=b_1b_2 \cdots b_j$.

The ordered pair of *positions* i and j of L, denoted $[i,j]$, is a *match* iff $a_i=b_j=\sigma_t$ for some t, $1 \le t \le s$. In the following, r will denote the number of distinct matches between A and B. If $[i,j]$ is a match, and an LCS C_{ij} of A_i and B_j has length k, then k is the *rank* of $[i,j]$. The match $[i,j]$ is *k-dominant* if it has rank k and for any other pair $[i',j']$ of rank k either $i' > i$ and $j' \le j$ or $i' \le i$ and $j' > j$. The total number of k–dominant matches in $L[i,j]$ will be denoted by d. Let l be the length of an LCS of A and B. It can be shown [HI] that for any $k \le l$ there must be at least one k-dominant match, and that, moreover, there is at least one LCS $C=c_1c_2 \cdots c_l$ such that c_k comes from a k-dominant match $(k=1,2,...,l)$. Thus, computing the k-dominant matches $(k=1,2,...,l)$ is all is needed to solve the LCS problem. For a large or a-priori unknown alphabet, and within the (fairly general) decision tree model of computation based on comparisons with outcome in [=, =!], the only lower bound that can be drawn for the LCS problem is $\Theta(mn)$ [AH].

However, it is easy to see [HI, HS] that once all k-dominant matches are available, then $O(m)$ time suffices to retrieve C. Most known approaches to the LCS problem require $\Theta(n+r)$ space. By contrast, the dynamic programming implementation presented in [HC] takes never more than $\Theta(n)$ space, though never less than $\Theta(nm)$ time.

As an illustration of the concepts introduced so far, Fig. 1 below displays the L-matrix for the strings $A = abcdbb$ and $B = cbacbaaba$: entries that correspond to matches are encircled. Emboldened circles circumscribe dominant matches and boundaries are traced to separate regions with constant L-entry.

		c	b	a	c	b	a	a	b	a
		1	2	3	4	5	6	7	8	9
a	1	0	0	1	1	1	1	1	1	1
b	2	0	1	1	1	2	2	2	2	2
c	3	1	1	1	2	2	2	2	2	2
d	4	1	1	1	2	2	2	2	2	2
b	5	1	2	2	2	3	3	3	3	3
b	6	1	2	2	2	3	3	3	4	4

Figure 1

A glance at Fig. 1 shows that the bold circles of our example are roughly one half of all circles. While it is obviously $d \le r$, the instinctive expectation for a general direct proportionality between r and d is soon to be defied. Indeed, consider the following two extreme instances, both offsprings of the initial assumption that it be $A = B$. In the first extreme, we also assume that A and B both represent some permutation of the integers: thus $d = r$, but also $d = n$. In the other extreme, we set instead $A = a^n$, i.e., both strings consist of n replicas of the same symbol a: thus $r = n^2$, but still $d = n$. This seems to suggest that, also in practice, the instances where it happens that d is linear in n, while r is not, may be frequently encountered.

As the starting point of our discussion, the algorithm presented in [HS] is reproduced below for the convenience of the reader.

```
Algorithm HS
element array A[1:m],B[1:n]; integer array THRESH[0:m]; list array MATCHLIST[1:m];
pointer array LINK[1:m]; pointer PTR;
begin
(STAGE 1: initializations)
  for i = 1 to m do
  set MATCHLIST[i] = { j1, j2, ..., jp } such that j1 > j2 ... > jp
              and A[i] = B[jq] for 1≤q≤p
  set:    THRESH[i] = n+1    for    1≤i≤m;    THRESH[0] = 0; LINK[0] = null;
(STAGE 2: find k-dominant matches)
  for i = 1 to m  do
   for j on MATCHLIST[i] do
     begin
       find k such that THRESH[k-1] < j ≤ THRESH[k];
       if j < THRESH[k] then
         begin THRESH[k] = j; LINK[k] = newnode(i, j, LINK[k-1]) end
     end
(STAGE 3: recover LCS C in reverse order)
  k = largest k such that THRESH[k]≠n+1;  PTR = LINK[k];
  while PTR≠null do
   begin print the match [i, j] pointed to by PTR; advance PTR  end
end.
```

The principle of operation of HS is transparent: by scanning the $MATCHLIST$ associated with the i–th row, the matches in that row are considered in succession, from right to left; through a binary search in the array $THRESH$, it is assessed whether the match being considered represents a k–dominant match for some k. In this case the contents of $THRESH[k]$ is suitably updated. We remark that considering the matches in reverse order is crucial to the correct operation of HS (the reader is referred to [HS] for details). The total time spent by HS is bounded by $O((r+m)logn + nlogs)$, where the $nlogs$ term is charged by the preprocessing. The space is bounded by $O(r+n)$. The time performance of HS is very good whenever r is comparable to n: in such (common) instances, the worst case time bound becomes in fact $O(rlogn) - O(nlogn)$. However, this performance degenerates as r gets close to mn: in these cases HS is outperformed by the algorithm in [HI], which exhibits a bound of $O(nl)$ in all situations (recall here that l is the length of C).

2. A Modified Paradigm

The objective of this section is to reformulate *HS* in such a way that it be easier for us to distill off possible sources of inefficiency. We present a first modified paradigm for the strategy in [HS], and then rearrange it in a harmless way. The efficient implementation of the final result of our discussion is deferred to the last section.

Our main modifications concern the second stage (finding k–dominant matches) of *HS* as presented above, although slight adjustments of the preprocessing are also required. The first innovation brought about by algorithm *HS* 1 below is in that it does not consider all the matches in each row. Rather, *HS* 1 maintains, for each symbol, its associated *active list* of matches, the matches of any such list being characterized by the fact that they are not current thresholds. The second innovation consists of spotting all and only the new dominant matches contributed by any given active list by performing a number of primitive 'dictionary' operations proportional to the number of these new dominant matches, i.e., independent of the current size of the active list involved.

Algorithm HS1

THRESH is the list of thresholds initially empty; the 'active' lists $AMATCHLIST[\sigma_t]$, $t=1,2,...s$ are initialized to coincide with the reverse of the corresponding *MATCHLIST*s of *HS*. The primitives *INSERT* and *DELETE* have the usual meaning, except they do nothing if the first argument is ∞ or the second argument is Λ. $SEARCH(key, LIST)$ returns (a pointer to) the smallest element in *LIST* which is larger than *key* (∞, if no such element exists). $SEARCH(\infty, LIST)$ returns ∞ without performing any action. The function $symb(character)$ returns the symbol of the alphabet Σ which coincides with *character*. By convention, $b_\infty = \Lambda$, and $AMATCHLIST(\Lambda) = \Lambda$. Finally, it is assumed without loss of generality that each symbol of the string A occurs at least once in B.

begin
for $i = 1$ **to** m **do**
 begin $\sigma_1 = symb(a_i)$; $j = first(AMATCHLIST[\sigma_1])$; $FLAG = true$;
   while $FLAG$ **do**
     begin
       1) $t = SEARCH(j, THRESH)$; $k = rank(t)$;
       2) **if** $t = \infty$ **then** $FLAG = false$;
       3) $INSERT(j, THRESH)$; $DELETE(t, THRESH)$;
       4) $LINK[k] = newnode(i, j, LINK[k-1])$;
       5) $\sigma_2 = symb(b_t)$;
       6) $DELETE(j, AMATCHLIST[\sigma_1])$; $INSERT(t, AMATCHLIST[\sigma_2])$;
       7) $j = SEARCH(t, AMATCHLIST[\sigma_1])$;
     end;
 end;
 retrieve C as per stage 3 of *HS*.

end.

To exemplify the operation of *HS* 1, we may refer to Fig.1 and interpret it as representing the product of *HS* 1 after it has processed the matches between $B = cbacbaaba$ and the symbols in the first six positions of $A = abcdbba...$ At this point, *THRESH* consists of $\{1,2,5,8\}$, and $\sigma_1 = A[7] = a$. At this particular stage of our example, it so happens that $AMATCHLIST[a] = \{3,6,7,9\}$ coincides with the reverse of $MATCHLIST[7]$, i.e., each occurrence of an a in B could become a new threshold. *HS* 1 starts by searching for '3' in *THRESH*, which returns the entry '5'. Since $5 \neq \infty$, then *THRESH* is updated so that it becomes now $\{1,2,3,8\}$ (line 3), and the original entry '5' is returned to $AMATCHLIST[B[5]] = AMATCHLIST[b]$, while '3' is deleted from $AMATCHLIST[a]$ (line 6). The algorithm now searches in $AMATCHLIST[a]$ for the old threshold value '5', and this search returns '6' (line 7). Thus this block terminates with $FLAG$ = *true*. When line 1 is executed next, it provokes the substitution, in $THRESH$, of the old entry '8' with the new entry '6', which is accompanied by the various list updates. The search of line 7 returns the entry '9' of $AMATCHLIST[a]$. As soon as line 1 is executed again, $FLAG$ is set to *false*. This will cause the exit from the *while* loop soon after the necessary updates have been performed (notice that some of the updates are dummy in this case, since $t=\infty$, and that the search of line 7 is gratuitous, since $FLAG$ was set to *false*). As the final result of the management of $A[7]$, $THRESH$ has become $\{1,2,3,6,9\}$, while $AMATCHLIST[a]$ shrunk to just $\{7\}$. On the other hand, $AMATCHLIST[b]$ was given back the matches '5' and '8'.

In general, the correctness of $HS1$ can be established as follows. First, observe that, even when $\sigma_1 = \sigma_2$, if t is replaced by j in $THRESH$ during the $i-th$ iteration of $HS1$, then t could never have to be reinserted in $THRESH$ within that iteration. Indeed, even if $[i,t]$ is a match, it cannot be a dominant match, since the fact that t was formerly in $THRESH$ implies that there is a match $[i',t]$ with $i' < i$ and having the same rank as $[i,j]$. The inner loop of $HS1$ exploits this observation (cfr. the definition of $SEARCH$) in conjunction with the following fact: letting t be the last item removed from $THRESH$, then the smallest entry of $AMATCHLIST[symb(a_i)]$ which is larger than t represents the leftmost new dominant match among those such matches which fall to the right of t. It is easy to check at this point that the outer loop of $HS1$ maintains the following conditions. After $HS1$ has performed the $i-th$ iteration of the outer loop:

1. The $k-th$ entry j_k of $THRESH$ is the smallest position in B such that there is a $k-$ dominant match between A_i and B.

2. $AMATCHLIST[\sigma_t], t=1,2,...,s$ contains all and only the occurrences of σ_t in B which are not currently in $THRESH$.

We are now ready to asses a time bound for $HS1$. The preprocessing involved in $HS1$ is quite similar to that in [HS]. The table *symb* is thought of as produced during such preprocessing, within the bound of $O(n\log s)$ charged by this latter. Thus each subsequent reference to this table can be assumed to take constant time. *HS* 1 takes at least $\Theta(m)$ time, since it considers each one of the m rows, in succession. If we add the convention that ∞ is appended at the end of each *AMATCHLIST* during initialization, then *HS* 1 spends constant time in handling any *trivial* row, i.e., any row whose *AMATCHLIST* is found to contain currently only ∞.

Theorem 1. In handling all nontrivial rows, *Algorithm HS* 1 performs $\Theta(d)$ searches, insertions and deletions.

Proof.

All the searches, insertions and deletions take place in the *while* loop (lines 1-7) controlled by *FLAG*. There is a fixed number of such primitives within these lines, whence it will do to show that *FLAG* is *true* exactly d times. With our assumptions, $AMATCHLIST[\sigma_1] = reverse(MATCHLIST[1])$ is not empty, and the first element on this list (i.e., the leftmost match in the form [1,j]) is a 1-dominant match, as well as the only dominant match in that list. By initialization, *FLAG* is *true* the first time it is tested. Since *THRESH* is empty at this time, lines (3,4) will be executed, whence the first 1-dominant match is recorded. The algorithm also proceeds to the update of the other lists involved, so that at next step the contents of such lists will be consistent. Moreover, since the *SEARCH* of line (1) returns ∞, then *FLAG* is set to the value *false*, which exhausts all manipulations involving matches of $A[1]$. In general, the first match on the *AMATCHLIST* associated with the nontrivial row corresponding to the $i-th$ character of A is certainly a k-dominant match for some k. Assume that a certain number of entries of such *AMATCHLIST* have been processed and that: (i) the number of times that *FLAG* was *true* equals the number of dominant matches detected so far, (ii) j identifies the last dominant match detected, and (iii) j is the only such match which has not been recorded yet. It is easy to see that *HS* 1: locates the displacement of this match in *THRESH* (line 1); switches *FLAG* to *false*, if appropriate (line 2); updates the lists and records this new dominant match in

LINK (lines 3-6), and finally probes into *AMATCHLIST*$[\sigma_1]$ seeking the next dominant match (line 7, meaningful only if *FLAG* is *true*). Thus *FLAG* is *true* exactly when conditions (i-iii) hold, that is, d times. □

The actual time bound of $HS1$ depends on the internal representation which is chosen for the various lists involved. If the lists are represented as priority queues such as 2-3 trees or *AVL* trees [ME], then $HS1$ runs in $O(dlogn + nlogs)$ time, inclusive of preprocessing, which reduces to $O(dloglogn + nlogs)$ if one uses a structure better fit to the manipulation of integers [VE]. This compares already favorably with the corresponding bounds in [HS], where r figures in the place of d. One interesting observation, however, is that the sequences of insertions in each list constitute in fact merges of sorted linear sequences. As is well known, efficient dynamic structures are available [BW,BT,ME] which support, say, the merging of two lists of sizes k and $f \geq k$ in time $O(klog(f/k) + k)$. This leads to speculate that the total time spent by $HS1$ for the mergings could be bounded by a form such as $O(mlogm + dlog(nm/d) + d)$. Unfortunately, it does not seem that the $O(klog(f/k) + k)$ bound still holds, with such structures, if deletions are intermixed with insertions in an uncontrolled way.

However, it turns out that the special case which is of interest here is indeed susceptible of efficient implemetantion through finger-trees. In a forthcoming paper of broader scope [AG], we also show that the same objective can be achieved at the expense of almost negligible complications, by appealing to simple properties of standard static trees.

We shall find it more convenient to apply our discussion to a modified version of $HS1$, which we now proceed to describe. This version, to which we will refer from now on when speaking of $HS1$, is obtained from the old version by performing only a few substitutions and additions. The basic observation here is that, as far as the correct management of any single row is concerned, only the information provided by the searches is needed on-line. Thus, each of the insertions and deletions which appear in lines (3) and (6) can be replaced by a recording of the fact that such operation has to take place before the algorithm can proceed to the next row. The recording process may consist of simply appending the primitive to be performed at the tail of a suitable batch queue. There are at least four and at most $s+3$ such queues, the most demanding case occurring when two *deletion* queues are dedicated to the the deletions to be performed from *THRESH* and the 'invariant' list *AMATCHLIST*$[\sigma_1]$, respectively, one *insertion* queue is dedicated to *THRESH*, and finally s *insertion* queues are dedicated to the various incarnations of *AMATCHLIST*$[\sigma_2]$. All these batches are executed before proceeding to the next row, and this is accompanied by the destruction of the queues. The reader is encouraged to check for himself that this rearrangement does not affect the correctness of our algorithm, nor its performance.

3. Finger-Trees

For the following discussion, we assume that S and Q are linearly ordered sets represented as finger-trees. For our purposes, a *finger-tree* is a *level-linked* (a,b)*-tree* $(b \geq 2a,\ a \geq 2)$ with fingers [ME]. A *finger* is simply a pointer to a leaf. A typical finger-tree can be obtained, for instance, from a standard (2,4) tree by adding links to each node in such a way that it becomes possible to reach, from that node, its father node, as well as its two neighbor nodes on the same level. Thus the resulting tree is traversable in any direction.

Our interest in finger-trees rests on the following facts from [ME] (cfr. also [BT, BW]). Let f and k, be the cardinalities of S and Q, respectively, and let $p_0 = 1$, and $p_1, p_2, \ldots, p_k$ be the positions of the elements of Q in $Q \cup S$. Finally, let $b_i = p_i - p_{i-1} + 1$, $i=1,2,\ldots k$. Consider the following three homogeneous series of k operations each (each series applies a chosen primitive to all the elements of Q, in an orderly fashion): (i) the search in S of each of the elements of Q, (ii) the insertion in S of each of the elements of Q (i.e., the construction of $S \cup Q$) (iii) the deletion from $U = S \cup Q$ of each of the elements of Q (i.e., the derivation of $U - Q$).

Lemma 1 [ME]. Each of the series (i-iii) can be implemented on finger-trees in time

$$O(k + log(f+k) + \sum_{i=1}^{k} logb_i).$$

We remark that the above bound holds just as well for other series of standard concatenable queue operations, and that Lemma 1 or similar results can be stated under the stronger assumption that $k \le f$ [BT,BW,ME]. However, Lemma 1 already gives us a handle for efficiently carrying out the searches and the batches of *insert* or *delete* operations involved in $HS1$ at each row. Thus we assume henceforth that all lists in $HS1$ are implemented as finger-trees. For the following theorem, we stipulate that $logx = max(logx, 1)$.

Theorem 2. $HS1$ requires $O(nlogs)$ preprocessing time and $O(mlogn + dlog(mn/d))$ processing time.

Proof.

It is easy to check that the preprocessing required by $HS1$ is basically the same as that required by HS, whence we can concentrate on the second time-bound. Let d_i denote the number of dominant matches which will be introduced as a result of handling row i. We can safely assume that $SEARCH(key, LIST)$ is engineered so as to take constant time if key is smaller or larger than any other element in $LIST$. Then, as seen in the discussion of Theorem 1, d_i nontrivial searches are performed on $THRESH$ while considering row i. Thus, by Lemma 1, the cost of all searches on this row is bounded, up to a multiplicative constant, by $log(m+d_i) + \sum_{k=1}^{d_i} logb_k$, where the *intervals* b_k are such that $\sum_{k=1}^{d_i} b_k \le 2m$, since $THRESH$ never contains more than m elements. It follows that, up to a multiplicative constant, the total cost on all rows is bounded by $mlogm + \sum_{k=1}^{d} logb_k$, where now $\sum_{k=1}^{d} b_k \le 2m^2 \le 2mn$. With this constraint, the previous sum is maximized by choosing all b_i equal, i.e., $b_i = 2nm/d$. The claimed bound then follows at once, since $mlogm \leq mlogn$. The same argument can be used to show that the desired bound holds for the batches of insertions and deletions performed on *THRESH*. We now turn to the insertions collectively performed on all the *AMATCHLIST* s invoked during the management of any single row. The key observation here is that the sum of the number of elements found in all such lists does never exceed n. Thus for each row, the total work spent on all lists can be bounded by an expression similar to that derived for the searches on *THRESH*, except that m is replaced by n. Similarly, the condition $\sum_{k=1}^{d} b_i \leq 2nm$ replaces the one that was used for *THRESH* when adding up the work involved in all rows. Thus the assertion holds for these insertions as well. The same observation, and an argument analogous to the above, leads to establish our bound for the searches and deletions involving *AMATCHLIST* $[\sigma_1]$. □

Acknowledgements

C. Guerra, K. Mehlhorn and R. Melhem read an earlier version of this paper and gave me many helpful suggestions.

REFERENCES

[AG] Apostolico, A. and C. Guerra. The Longest Common Subsequence Problem Revisited, typescript, Jan. 1985.

[AH] Aho, A.D., D.S. Hirschberg and J.D. Ullman. Bounds on the complexity of the maximal common subsequence problem, *JACM* 23, 1, 1-12 (1976).

[BT] Brown, M.R., and R.E. Tarjan. A representation of linear lists with movable fingers. *Proceedings of the* 10–*th STOC*, San Diego, Ca., 19-29 (1978).

[BW] Brown, M.R., and R.E. Tarjan. A fast merging algorithm, *JACM* 26, 2, 211-226 (1979).

[HI] Hirschberg, D.S. Algorithms for the longest common subsequence problem, *JACM* 24, 4, 664-675 (1977).

[HC] Hirschberg, D.S. A linear space algorithm for computing maximal common subsequences, *CACM* 18, 6, 341-343 (1975)

[HS] Hunt, J.W., and T.G. Szymanski. A fast algorithm for computing longest common subsequences, *CACM* 20, 5, 350-353 (1977).

[ME] Mehlhorn, K. Data Structures and Algorithms 1: Sorting and Searching, Springer-Verlag, EATCS Monographs on TCS (1984).

[VE] van Emde Boas, P. Preserving order in a forest in less than logarithmic time and linear space. *Inf Proc.Lett.* 6, 3, 80-82 (1977).