Programming Techniques

G. Manacher Editor

A Linear Space Algorithm for Computing Maximal Common Subsequences

D.S. Hirschberg
Princeton University

The problem of finding a longest common subsequence of two strings has been solved in quadratic time and space. An algorithm is presented which will solve this problem in quadratic time and in linear space.

Key Words and Phrases: subsequence, longest common subsequence, string correction, editing

CR Categories: 3.63, 3.73, 3.79, 4.22, 5.25

Introduction

The problem of finding a longest common subsequence of two strings has been solved in quadratic time and space [1, 3]. For strings of length 1,000 (assuming coefficients of 1 microsecond and 1 byte) the solution would require 10^6 microseconds (one second) and 10^6 bytes (1000K bytes). The former is easily accommodated, the latter is not so easily obtainable. If the strings were of length 10,000, the problem might not be solvable in main memory for lack of space.

We present an algorithm which will solve this problem in quadratic time and in linear space. For example, assuming coefficients of 2 microseconds and 10 bytes, for strings of length 1,000 we would require 2 seconds and 10K bytes; for strings of length 10,000 we would require a little over 3 minutes and 100K bytes.

String $C = c_1c_2\cdots c_p$ is a *subsequence* of string $A = a_1a_2\cdots a_m$ if and only if there is a mapping F: $\{1, 2, \ldots, p\} \rightarrow \{1, 2, \ldots, m\}$ such that $f(i) = k$ only if c_i is a_k and F is a monotone strictly increasing function (i.e. $F(i) = u$, $F(j) = v$, and $i < j$ imply that $u < v$).

String C is a *common subsequence* of strings A and B if and only if C is a subsequence of A and C is a subsequence of B.

The problem can be stated as follows: Given strings $A = a_1a_2\cdots a_m$ and $B = b_1b_2\cdots b_n$ (over alphabet Σ), find a string $C = c_1c_2\cdots c_p$ such that C is a common subsequence of A and B and p is maximized.

We call C an example of a *maximal common subsequence.*

Notation. For string $D = d_1d_2\cdots d_r$, D_{kt} is $d_kd_{k+1}\cdots d_t$ if $k \leq t$; $d_kd_{k-1}\cdots d_t$ if $k \geq t$. When $k > t$, we shall write $\hat{D}_{kt}$ so as to make clear that we are referring to a "reverse substring" of D.

$L(i, j)$ is the maximum length possible of any common subsequence of A_{1i} and B_{1j}.

$x\| y$ is the concatenation of strings x and y.

We present the algorithm described in [3], which takes quadratic time and space.

Algorithm A

Algorithm A accepts as input strings A_{1m} and B_{1n} and produces as output the matrix L (where the element $L(i, j)$ corresponds to our notation of maximum length possible of any common subsequence of A_{1i} and B_{1j}).

```
ALG A (m, n, A, B, L)
1.  Initialization: L(i, 0) ← 0 [i=0···m];
                    L(0, j) ← 0 [j=0···n];
2.  for i ← 1 to m do
    begin
3.  for j ← 1 to n do
    if A(i) = B(j) then L(i, j) ← L(i−1, j−1) + 1
                  else L(i, j) ← max{L(i, j−1), L(i−1, j)}
    end
```

Proof of Correctness of Algorithm A

To find $L(i, j)$, let a common subsequence of that length be denoted by $S(i, j) = c_1c_2\cdots c_p$. If $a_i = b_j$, we can do no better than by taking $c_p = a_i$ and looking for $c_1\cdots c_{p-1}$ as a common subsequence of length $L(i, j) - 1$ of strings $A_{1,i-1}$ and $B_{1,j-1}$. Thus, in this case, $L(i, j) = L(i - 1, j - 1) + 1$.

If $a_i \neq b_j$, then c_p is a_i, b_j, or neither (but not both). If c_p is a_i, then a solution C to problem (A_{1i}, B_{1j}) [written $P(i, j)$] will be a solution to $P(i, j - 1)$ since b_j is not used. Similarly, if c_p is b_j, then we can get a solution to $P(i, j)$ by solving $P(i - 1, j)$. If c_p is neither, then a solution to either $P(i - 1, j)$ or $P(i, j - 1)$ will suffice. In determining the length of the solution, it is seen that $L(i, j)$ [corresponding to $P(i, j)$] will be the maximum of $L(i - 1, j)$ and $L(i, j - 1)$. □

Research work was supported in part by NSF grant GJ-30126 and National Science Foundation Graduate Felolwship. Author's address: Department of Electrical Engineering, Princeton University, Princeton, NJ 08540.

Time and Space Analysis of Algorithm A

The **if** statement in Algorithm A will be executed exactly mn times. Input and output arrays require $m + n + (m + 1)(n + 1)$ locations. Thus Algorithm A requires $O(mn)$ time and $O(mn)$ space.

Algorithm B

In Algorithm A, the derivation of row i of matrix L $(L(i, 1), L(i, 2), \ldots, L(i, n))$ requires only row $i - 1$ of matrix L. Thus, a slight modification yields Algorithm B, which accepts as input strings A_{1m} and B_{1n} and produces as output vector LL where $LL(j)$ will have the value $L(m, j)$.

```
ALG B (m, n, A, B, LL)
1.  Initialization: K(1, j) ← 0 [j=0···n];
2.  for i ← 1 to n do
    begin
3.  K(0, j) ← K(1, j) [j=0···n];
4.  for j ← 1 to n do
    if A(i) = B(j) then K(1, j) ← K(0, j−1) + 1
                   else K(1, j) ← max{K(1, j−1), K(0, j)};
    end
5.  LL(j) ← K(1, j) [j=0···n]
```

Proof of Correctness of Algorithm B

Algorithm B is Algorithm A with $K(0, j)$ in statement 4 of ALG B having the same value as $L(i - 1, j)$ in statement 3 of ALG A and $K(1, j)$ receiving the same value as $L(i, j)$. We show this by induction on i.

For $i = 1$, $L(i - 1, j)$ is zero (initialized in statement 1 of ALG A). In ALG B, $K(0, j)$ received in statement 3 the value of $K(1, j)$, which was just initialized to zero in statement 1.

Assume $K(0, j)$ has the same value as does $L(i - 1, j)$. Then $K(1, j)$ receives the same value as $L(i, j)$ since the assignment statements within the inner loops of ALG A and ALG B are equivalent. For the next iteration, $K(0, j)$ receives (in statement 3 of ALG B) the value of $K(1, j)$ which has the value of $L(i, j)$ as shown above. □

Time and Space Analysis of Algorithm B

As in Algorithm A, the **if** statement in Algorithm B is executed exactly mn times. Input and output arrays require $m + n + (n + 1)$ locations. Local storage requires $2(n + 1)$ locations. Thus Algorithm B requires $O(mn)$ time and $O(m + n)$ space.

We shall show that using Algorithm B for appropriate substrings of A and B will enable us to recover a maximal common subsequence of A and B in linear space.

Define $L^*(i, j)$ to be the maximum length of common subsequences of $A_{i+1,m}$ and $B_{j+1,n}$.

We note that $L(i, j)$ $j = 0 \cdots n$ are the maximum lengths of common subsequences of A_{1i} and various prefixes of B_{1n}. We also note that $L^*(i, j)$ $j = 0 \cdots n$ are the maximum lengths of common subsequences of $\hat{A}_{m,i+1}$ and various prefixes of $\hat{B}_{n1}$. Choosing i to be $m/2$ and using the theorem below, we shall be able to determine a prefix B_1 of B which can be matched with the first half A_1 of A (and the corresponding suffix B_2 of B matched with the last half A_2 of A) such that a maximal common subsequence (mcs) of A_1 and B_1 concatenated with an mcs of A_2 and B_2 will be an mcs of A and B.

Define $M(i) = \max_{0 \le j \le n} \{L(i, j) + L^*(i, j)\}$.

THEOREM. *For* $0 \le i \le m$, $M(i) = L(m, n)$.

PROOF. Let $M(i) = L(i, j) + L^*(i, j)$ for some j. Let $S(i, j)$ be any maximal common subsequence of A_{1i} and B_{1j}; let $S^*(i, j)$ be any maximal common subsequence of $A_{i+1,m}$ and $B_{j+1,n}$. Then $C = S(i, j) \mid\mid S^*(i, j)$ is a common subsequence of A_{1m} and B_{1n} of length $M(i)$. Thus $L(m, n) \ge M(i)$.

Let $S(m,n)$ be any maximal common subsequence of A_{1m} and B_{1n}. $S(m,n)$ is a subsequence of B that is S_1 (a subsequence of A_{1i}) $\mid\mid$ S_2 (a subsequence of $A_{i+1,m}$). Then there exists j such that S_1 is a subsequence of B_{1j} and S_2 is a subsequence of $B_{j+1,n}$. By definition of L and L^*, $|S_1| \le L(i, j)$ and $|S_2| \le L^*(i, j)$. Thus $L(m,n) = |S(m, n)| = |S_1| + |S_2| \le L(i, j) + L^*(i, j) \le M(i)$. □

Algorithm C

We now apply the above theorem recursively to divide a given problem into two smaller problems until we obtain a trivial subproblem.

Algorithm C accepts as input strings A and B (of lengths m and n) and produces as output a common subsequence C of A and B that is of maximum length p.

```
ALG C (m, n, A, B, C)
1.  If problem is trivial, solve it:
    if n = 0 then C ← e   (e is the empty string)
    else if m = 1 then if ∃j≤n such that A(1) = B(j)
                          then C ← A(1)
                          else C ← e
2.  Otherwise, split problem:
    else begin i ← ⌊m/2⌋;
3.  Evaluate L(i, j) and L*(i, j) [j = 0···n]:
      ALG B (i, n, A_1i, B_1n, L1);
      ALG B (m−i, n, Â_n,i+1, B̂_n1, L2);
4.  Find j such that L(i, j) + L*(i, j) = L(m, n) using theorem:
      M ← max {L1(j) + L2(n−j)};
         0≤j≤n
      k ← min j such that L1(j) + L2(n−j) = M;
5.  Solve simpler problems:
      ALG C (i, k, A_1i, B_1k, C1);
      ALG C (m−i, n−k, A_i+1,m, B_k+1,n, C2);
6.  Give output:
      C ← C1 || C2;
    end
```

Proof of Correctness of Algorithm C

$L1(j)$ produced by the first call to ALG B in line 3 is equal to $L(i, j)$. This was shown in the proof of correctness of Algorithm B. Similarly, $L2(j)$ is equal to the maximum length of common subsequences (max lcs) of $\hat{A}_{m,i+1}$ and $\hat{B}_{n,n-j+1}$ by the proof of correctness of Algorithm B.

Thus

$$\begin{aligned} L2(n - j) &= \text{max lcs of } A_{m,i+1} \text{ and } \hat{B}_{n,j+1}, \\ &= \text{max lcs of } A_{i+1,m} \text{ and } B_{j+1,n}, \\ &= L^*(i, j). \end{aligned}$$

By our theorem, we can find k (as in line 4) such that $L(i, k) + L^*(i, k) = L(m, n)$. So there must exist solutions $C1$ and $C2$ to the subproblems (A_{1i}, B_{1k}) and $(A_{i+1,m}, B_{k+1,n})$ such that $C1 \,||\, C2$ will be a common subsequence of A and B of length $L(m, n)$. The solutions to the subproblems are obtained in line 5 and are added together in line 6 to obtain the final output. □

Time Analysis of Algorithm C

For $P(1, n)$ we look for a single match. For some constants c_1 and c_2 this is time-bounded by $c_1 \cdot n + c_2$.

For $P(2m,n)$, let operations on vectors that are linear in m or n be time-bounded by $c_3 \cdot m + c_4 \cdot n + c_5$. That leaves two calls to ALG B and two calls to ALG C. The calls to ALG B are bounded by $c_6 \cdot mn$ by time analysis of ALG B. Assume $P(m, n)$ is time-bounded by $d_1 \cdot mn + d_2$ $(d_1 \geq c_1, d_2 \geq c_2)$. Then the calls to ALG C will be time-bounded by $d_1 \cdot mk + d_2$ and $d_1 \cdot m(n - k) + d_2$. Thus a total time-bound T for $P(2m,n)$ will be

$$T = (d_1 + c_6) \cdot mn + c_3 \cdot m + c_4 \cdot n + c_5 + 2d_2.$$

For $n \geq 1$, $T \leq (d_1 + c_6 + c_3 + c_4 + c_5 + d_2) \cdot mn + d_2$. For $n = 0$, let $T \leq d_2$. Then to be consistent with our assumption on the time-bound of $P(m, n)$, we must have $d_1 + c_6 + c_3 + c_4 + c_5 + d_2 \leq 2d_1$, which is realizable by letting $d_1 = c_6 + c_3 + c_4 + c_5 + d_2$.

Thus Algorithm C has an $O(mn)$ time bound.

Space Analysis of Algorithm C

We assume that vectors A and B are in common storage and substrings can be transferred as arguments by giving initial and final locations.

Then, during execution, the calls to ALG B use temporary storage which is linear in m and n (see space analysis of Algorithm B). It is seen that, exclusive of recursive calls to ALG C, ALG C uses a constant amount of memory space. There are $2m - 1$ calls to ALG C (proven below), and so ALG C requires memory space proportional to m and n, i.e. $O(m + n)$ space.

Proof That There Are 2m — 1 Calls to ALG C

Let $m \leq 2^r$. If r is zero, then m is one, and there are $2 \cdot 1 - 1 = 1$ call to ALG C.

Assume that for $m \leq 2^r = M$ there are $2m - 1$ calls to ALG C. For $m' \leq 2^{r+1} = 2M$, i will be set equal to at most M in line 2. There will be two calls to ALG C with first parameters m_1 and m_2 such that $m_1 + m_2 = m'$ and both m_1 and m_2 are at most M. By assumption, each of these calls will generate a total of $2m_i - 1$ calls to ALG C. Adding in the initial call results in a total of: $(2m_1 - 1) + (2m_2 - 1) + 1 = 2(m_1 + m_2) - 1 = 2m' - 1$ calls. □

Algorithm C can be modified to find the edit distance between two strings (as defined in [3]). In this case we would seek to minimize $D(m, n)$, the cost of a trace from A_{1m} to B_{1n}. The corresponding statement of our theorem would be: for all i,

$$D(m, n) = \min_{0 \leq j \leq n} \{D(i, j) + D^*(i, j)\}.$$

The modified Algorithm C would split problems in half by the above theorem, using a modified Algorithm B to evaluate $D(i, j)$ and $D^*(i, j)$, and call itself recursively.

Received May 1974; revised November 1974

References

1. Chvatal, V., Klarner, D.A., and Knuth, D.E. Selected combinatorial research problems. STAN-CS-72-292, Stanford U., (June 1972), 26.
2. Private communication from D. Knuth to J.D. Ullman.
3. Wagner, R.A., and Fischer, M.J. The string-to-string correction problem. *J. ACM 21*, 1 (Jan. 1974), 168–173.
4. Aho, A. V., Hirschberg, D.S., and Ullman, J.D. Bounds on the complexity of the longest common subsequence problem. Proc. 15th Ann. Symp. on Switching and Automata Theory, 1974, pp. 104–109.