Information Processing Letters 23 (1986) 305–310
North-Holland

3 December 1986

A BIT-STRING LONGEST-COMMON-SUBSEQUENCE ALGORITHM

Lloyd ALLISON and Trevor I. DIX

Department of Computer Science, University of Western Australia, Nedlands, Western Australia 6009

Communicated by M.A. Harrison
Received 7 October 1985
Revised 17 December 1985

A longest-common-subsequence algorithm is described which operates in terms of bit or bit-string operations. It offers a speedup of the order of the word-length on a conventional computer.

Keywords: Longest common subsequence, edit distance, bit string

1. Introduction

The *longest-common-subsequence* (LCS) problem is to find the maximum possible length of a common subsequence of two strings, "a" of length $|a|$ and "b" of length $|b|$. Usually, an actual LCS is also required. For example, using the alphabet A, C, G, and T of genetic *bases*, an LCS of "GCTAT" and "CGATTA" is "GTT" of length three.

Here, an algorithm which requires $O(|a| \times |b|)$ operations on single bits or $O(\lceil |a|/w \rceil \times |b|)$ operations on w-bit computer words or $O(|b|)$ operations on $|a|$-bit bit-strings, for a fixed finite alphabet, is presented. Although falling into the same complexity class as simple LCS algorithms, if w is greater than any additional multiplicative cost, this algorithm will be faster. If $|a| \leqslant w$, the algorithm is effectively linear in $|b|$. (An alphabet larger than $|b|$ can effectively be reduced to $|b|$ by sorting "b" in $O(|b| \times \log |b|)$ time and using index positions in the sorted string.)

The LCS problem is related to the *edit-distance* [11] or *evolutionary-distance* problem [9,10], where the minimum cost of editing string "a" to string "b" must be found. The elementary edit operations are to insert or delete one character or to change one character. There is a cost function, $d(\cdot, \cdot)$, which can be extended to strings in the obvious way. A common choice for the elementary edit costs is

$$
\begin{aligned}
&d(\alpha, \alpha) = 0, \\
&d(\alpha, \beta) = 2 \quad \text{if } \alpha \neq \beta \qquad \text{(cost of changing character } \alpha \text{ to } \beta\text{)}, \\
&d(\alpha, -) = 1 \qquad\qquad\quad \text{(cost of deleting } \alpha\text{)}, \\
&d(-, \alpha) = 1 \qquad\qquad\quad \text{(cost of inserting } \alpha\text{)},
\end{aligned}
$$

and then $d(a, b) = |a| + |b| - 2 \times |\mathrm{LCS}(a, b)|$.

The LCS problem and the edit-distance problem find applications in computer science and molecular biology [8,9,10,11,12]. An LCS algorithm can compare two files and, by finding what they have in common, compute their differences. It can be used to correct spelling errors and to compare two genetic sequences for homology (similarity).

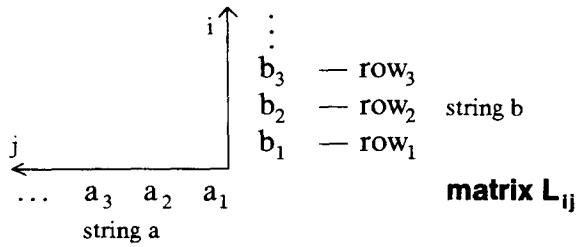
Under plausible assumptions [1,13], the worst-case complexity of an LCS or edit-distance algorithm on strings over an infinite alphabet must be $O(|a| \times |b|)$ time. Masek and Paterson [5] give an $O(|a| \times |b|/$

log(min{|a|, |b|})) edit-distance algorithm for strings over a finite alphabet with modest restrictions on $d(\cdot, \cdot)$. However, this is faster than simple algorithms only for strings longer than about 200 000 characters. For special cases there are faster algorithms: for similar strings [3,6], and for strings with few matching characters [4], but their worst-case running time is at least $O(|a| \times |b|)$.

The algorithm presented here certainly has complexity $O(|a| \times |b|)$ for a finite alphabet, but asymptotically the improvement in speed is close to w. The algorithm is insensitive to the number of matches between strings and to their similarity.

1.1. Matrix L

Conventionally a matrix L_{ij} is defined as follows:



L_{ij} equals the length of an LCS of $a_{1,\ldots,j}$ and $b_{1,\ldots,i}$,

$$L_{ij} = \begin{cases} 1 + L_{i-1,j-1} & \text{if } a_j = b_i, \\ \max\{L_{i-1,j}, L_{i,j-1}\} & \text{otherwise.} \end{cases}$$

This leads to simple dynamic-programming algorithms [8] upon which this one is based. A reflected representation of matrix L is typically used; the above representation is chosen to make the reading of certain bit-strings easier.

L has many interesting properties:

$$L_{i-1,j-1} \leqslant L_{i,j-1}, L_{i-1,j} \leqslant L_{ij}, \qquad |L_{ij} - L_{i-1,j-1}| \leqslant 1.$$

The rows and columns of L_{ij} contain ascending values for increasing i and j. This prompted Hunt and Szymanski [4] and Hirschberg [3] to 'contour' L and develop fast algorithms for special cases.

2. Bit-string algorithm

The values in the rows of L increase by at most one. This makes it possible to represent the information in L by a bit-matrix, as shown in Fig. 1,

$$L_{ij} = \sum_{k=1,\ldots,j} M_{ik}.$$

Row_i has either the same number of bits set or one more bit set than the previous row, row_{i-1}. New bits are set towards the left of a row. The length of an LCS of "a" and "b" is the number of bits in the top row. Let 1's in a particular row tend to drift the right as we look (up) at the next row. This is because when more of "b" is used, an LCS of a given length can be found using no more of, and possibly less of, "a". The 1's mark the contour lines of matrix L.

#bits																		
10	0	1	0	1	1	0	0	0	1	0	1	1	1	1	1	1	T	string b
10	1	0	0	1	1	0	0	0	1	0	1	1	1	1	1	1	G	
9	0	0	0	1	1	0	0	0	1	0	1	1	1	1	1	1	T	
9	0	1	0	0	1	0	0	0	1	0	1	1	1	1	1	1	T	
8	0	0	1	0	0	0	1	0	0	0	1	1	1	1	1	1	C	
7	0	0	0	0	1	0	0	0	1	0	0	1	1	1	1	1	T	— row_{11}
7	1	0	0	0	0	0	0	1	0	0	0	1	1	1	1	1	A	— row_{10}
7	1	0	0	0	0	1	0	0	0	0	0	1	1	1	1	1	G	
6	0	0	0	0	0	1	0	1	0	0	0	0	1	1	1	1	A	
5	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	1	A	
5	0	0	0	0	1	0	0	0	0	0	0	0	1	1	1	1	T	
4	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1	T	
3	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	C
3	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	G	
2	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	A	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	T **matrix M_{ij}**
	G	T	C	T	T	A	C	A	T	C	C	G	T	T	C	G		

string a

Fig. 1.

2.1. *Alphabet-strings*

Each letter in the alphabet defines a bit-string when it is compared with the elements of "a":

a:	G	T	C	T	T	A	C	A	T	C	C	G	T	T	C	G
A-string:	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
C-string:	0	0	1	0	0	0	1	0	0	1	1	0	0	0	1	0
G-string:	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
T-string:	0	1	0	1	1	0	0	0	1	0	0	0	1	1	0	0

Precomputing these *alphabet-strings* contributes $O(|\text{alphabet}| \times \lceil |a|/w \rceil + |a|)$ to the time complexity. For a fixed alphabet, this is $O(|a|)$; for a nonfixed alphabet, this could be $O(|a| \times |b|)$ at worst. If the alphabet is small ($|\text{alphabet}| \ll |b|$), the contribution to the total time can be ignored.

2.2. *Matrix M*

To calculate row_i, we use the *ith* character in string "b", b_i, to select the b_i-string from the set of alphabet-strings. The 1's in row_{i-1} 'cut' the b_i-string into segments:

row_{10} :	1	0	0	0	0	0	0	1	0	0	0	1	1	1	1	1
T-string:	0	1	0	1	1	0	0	0	1	0	0	0	1	1	0	0

Each segment extends from the position of a 1 in row_{i-1} rightward to the position to the left of the next 1. If the left-hand bit of row_{i-1} is zero, the left-most segment extends from the left of the b_i-string to the position left of the first 1. Row_i is formed by setting only the rightmost 1 bits in each segment of the b_i-string. If a segment is all zero, the bit defining the left end of the segment should be set in row_i (that is

the segmenting bit in row_{i-1}). This can be ensured by first or-ing row_{i-1} into the b_i-string.

T-string $\vee$ row_{10}:

	1	1	0	1	1*	0	0	1	1*	0	0	1*	1*	1*	1*	1*
row_{11}:	0	0	0	0	1	0	0	0	1	0	0	1	1	1	1	1

* indicates a right-most 1.

It may be convenient to imagine an invisible 1 at position $|a|+1$ for a leftmost segment which is zero, but this is unnecessary for the algorithm.

A 1 in row_{i-1} marks the shortest piece of "a" that can be used to make an LCS of a certain length with $b_{1,\ldots,i-1}$. Bringing b_i into use, the best that can be done to extend that LCS is to use the first b_i in "a" to the left of the 1, if possible. This is marked by the rightmost 1 bit in the next segment to the left.

In more detail, let $x = row_{i-1} \vee b_i$-string, for example, $x = row_{10} \vee$ T-string:

x:	1	1	0	1	1	0	0	1	1	0	0	1	1	1	1	1

Each segment of x can be read as a binary number. There is a 'folk-technique' that decrementing a binary number changes the low-order bits [7] up to and including the least-significant 1. The correct decrement for each segment can be found by a logical left-shift of row_{i-1} with a 1 carry-in:

x:	1	1	0	1	1	0	0	1	1	0	0	1	1	1	1	1
$-$	0	0	0	0	0	0	0	1	0	0	0	1	1	1	1	1
	1	1	0	1	0	1	1		1	0	1	1	$\bar{0}$	$\bar{0}$	$\bar{0}$	$\bar{0}$	$\bar{0}$

A nonequivalence of the result and x sets the changed bits:

	0	0	0	0	1	1	1	0	1	1	1	1	1	1	1	1

And-ing this with x gives the rightmost bits in each segment:

	0	0	0	0	1	0	0	0	1	0	0	1	1	1	1	1

This is row_i, row_{11} in this example.

In summary:

$$row_i = x \wedge ((x - (row_{i-1} \ll 1)) \neq x), \quad \text{where } x = row_{i-1} \vee b_i\text{-string}$$

with

$\vee \wedge \neq$: bit-string operations,
$\ll$: logical left-shift bit-string; right-hand bit set,
$-$: $|a|$-bit integer subtraction.

The bit-string operations can be done in units of a word-length. The shift and subtraction can also be done one word at a time, taking care that the carry-out-bit and the borrow-bit are propagated.

The number of bits set in a word can be counted in $O(\log_2 w)$ time; this is attributed to D. Muller in 1954 in [7]. The number of 1's in the final row can therefore be calculated in $O(\lceil |a|/w \rceil \times \log_2 w)$ time.

2.3. An LCS

An LCS, as opposed to just its length, can be obtained in at least two ways. Hirschberg's recursive technique [2] can be used to find an LCS in linear space at the cost of slowing the algorithm by a factor of two.

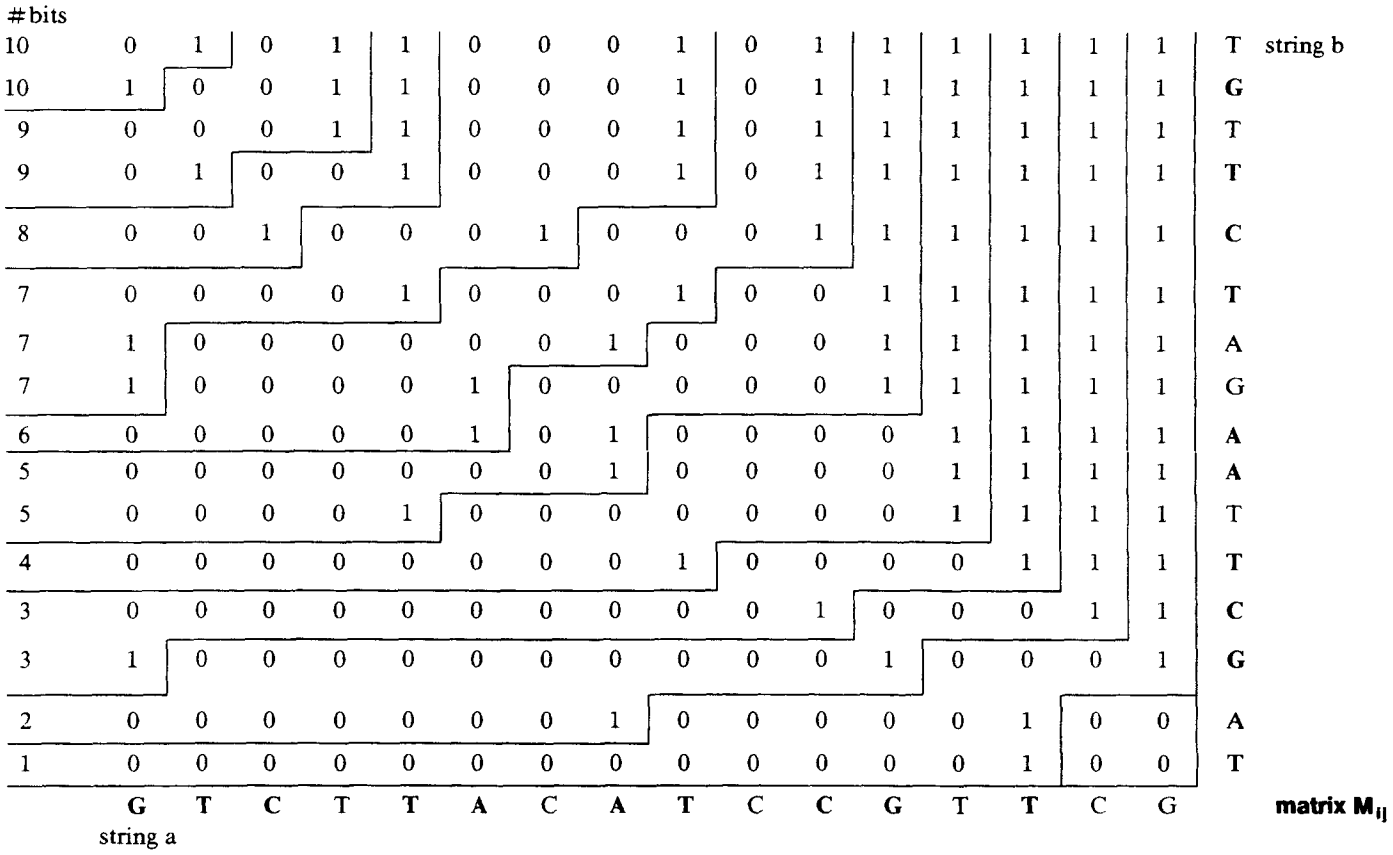


Fig. 2.

Alternatively, all rows of M can be kept, the space required is $|a| \times |b|$ bits, $\lceil |a|/w \rceil \times |b|$ words. This is quite practical for strings of the order of 1000 characters. Then, an LCS can be recovered by finding the 'corners' of the contours in L which stand out as patterns in M (see Fig. 2). This takes $O(|a| + |b|)$ time. The emboldened characters in Fig. 2 indicate an LCS.

3. Simulation results

Both the simple $O(|a| \times |b|)$ algorithm [2,8] and the bit-string algorithm to calculate the length of an LCS have been implemented in C and run on a VAX 11/750 with a 32-bit word. For random strings and an alphabet of size 4 the resulting speedups were

$\|a\| \times \|b\|$	Time ratio (simple/bit)
32 × 32	6
64 × 64	10
100 × 100	11
500 × 500	25
1000 × 1000	26
4000 × 4000	27

For comparison, an alphabet of size 256 resulted in

$\|a\| \times \|b\|$	Time ratio (simple/bit)
32×32	2
64×64	5
100×100	6
500×500	19
1000×1000	21
4000×4000	27

A similar number of operations are in the inner loop of each algorithm. The bit-string algorithm is well suited to optimization since it works across rows of matrix M and alphabet-strings. For small alphabets, the bit-string algorithm clearly provides a considerable speedup.

4. Conclusions

The time complexity of the bit-string LCS algorithm on a computer with w-bit words for a fixed finite alphabet is $O(\lceil |a|/w \rceil \times |b|)$. This gives a speedup over simple $O(|a| \times |b|)$ algorithms. The time does not depend on properties of "a" and "b". For random strings and moderate alphabets, the bit-string LCS algorithm will be faster than the special case algorithms.

The algorithm might also be programmed on a vector computer if carry-out and borrow bits can be propagated from word to word in a vector-shift and a vector-subtract operation. It could be 'pipelined' in a diagonal fashion on a vector or parallel machine because the least significant bit (or word) of row_{i+1} can be calculated as soon as the least significant bit (or word) of row_i has been calculated.

References

[1] A.V. Aho, D.S. Hirschberg and J.D. Ullman, Bounds on the complexity of the longest common subsequence problem, J. ACM 23 (1) (1976) 1–12.

[2] D.S. Hirschberg, A linear space algorithm for computing maximal common subsequences, Comm. ACM 18 (6) (1975) 431–433.

[3] D.S. Hirschberg, Algorithms for the longest common subsequence problem, J. ACM 24 (4) (1977) 664–675.

[4] J.W. Hunt and T.G. Szymanski, A fast algorithm for computing longest common subsequences, Comm. ACM 20 (5) (1977) 350–353.

[5] W.J. Masek and M.S. Paterson, How to computer string-edit distances quickly, in: D. Sankoff and J.B. Kruskal, eds., Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison (Addison-Wesley, Reading, MA, 1983).

[6] N. Nakatsu, Y. Kambayashi and S. Yajima, A longest common subsequence algorithm suitable for similar text strings, Acta Informatica 18 (1982) 171–179.

[7] E.M. Reingold, J. Nievergelt and N. Deo, Combinatorial Algorithms: Theory and Practice (Prentice-Hall, Englewood Cliffs, NJ, 1977).

[8] D. Sankoff and J.B. Kruskal, eds., Time Warps, String Edits and Macromolecules:. The Theory and Practice of Sequence Comparison (Addison-Wesley, Reading, MA, 1983).

[9] P.H. Sellers, On the theory and computation of evolutionary distances, SIAM J. Math. 26 (4) (1974) 787–793.

[10] P.H. Sellers, The theory and computation of evolutionary distances: Pattern recognition, J. Algorithms 1 (4) (1980) 359–373.

[11] R.A. Wagner and M.J. Fischer, The string-to-string correction problem, J. ACM 21 (1) (1974) 168–173.

[12] M.S. Waterman, General methods of sequence comparison, Bull. Math. Biology 46 (4) (1984) 473–500.

[13] C.K. Wong and A.K. Chandra, Bounds for the string editing problem, J. ACM 23 (1) (1976) 13–16.