An all-substrings common subsequence algorithm

C.E.R. Alves\textsuperscript{a}, E.N. Cáceres\textsuperscript{b}, S.W. Song\textsuperscript{c}

\textsuperscript{a}Faculdade de Tecnologia e Ciências Exatas, Universidade São Judas Tadeu, São Paulo, SP, Brazil
\textsuperscript{b}Departamento de Computação e Estatística, Universidade Federal de Mato Grosso do Sul, Campo Grande, MS, Brazil
\textsuperscript{c}Departamento de Ciência da Computação, Universidade de São Paulo, São Paulo, SP, Brazil

Received 31 May 2005; received in revised form 31 January 2006; accepted 20 May 2007
Available online 4 September 2007

Abstract

Given two strings $A$ and $B$ of lengths $n_a$ and $n_b$, $n_a \leq n_b$, respectively, the all-substrings longest common subsequence (ALCS) problem obtains, for every substring $B'$ of $B$, the length of the longest string that is a subsequence of both $A$ and $B'$. The ALCS problem has many applications, such as finding approximate tandem repeats in strings, solving the circular alignment of two strings and finding the alignment of one string with several others that have a common substring. We present an algorithm to prepare the basic data structure for ALCS queries that takes $O(n_a n_b)$ time and $O(n_a + n_b)$ space. After this preparation, it is possible to build a matrix of size $O(n_b^2)$ that allows any LCS length to be retrieved in constant time. Some trade-offs between the space required and the querying time are discussed. To our knowledge, this is the first algorithm in the literature for the ALCS problem.

© 2007 Elsevier B.V. All rights reserved.

MSC: 68W05; 68W40

Keywords: All substring common subsequence problem; String processing

1. Introduction

The computation of patterns in strings is a fundamental problem in Molecular Biology and many other areas [17]. In this paper we consider a variant of the longest common subsequence (LCS) problem. Consider a string of symbols from a finite alphabet. A substring of a string is any fragment of contiguous symbols of the string. A subsequence of a string is obtained by deleting zero or more symbols from the original string, at any position. Given strings $A$ and $B$, $|A| = n_a$ and $|B| = n_b$, the LCS problem obtains the longest subsequence common to both. It is an important problem with applications in DNA sequence comparison, data compression, pattern matching, etc. [13,16]. A generalization of the LCS problem is the all-substrings longest common subsequence (ALCS) problem that obtains the lengths of all the LCSs for string $A$ and all substrings of $B$. The ALCS problem is a restricted form of the all-substrings alignment problem [2,15]. It has many applications, such as finding approximate tandem repeats in strings [15], solving the circular alignment of two strings [12,15] and finding the alignment of one string with several others that have a common substring [10].
Sequential algorithms for the LCS problem have been proposed in [6,9,14]. The basic LCS problem can be solved in \(O(n_a n_b)\) time. There is, however, no sequential algorithm for the ALCS problem in the literature. Schmidt [15] presents an algorithm to solve the string edit problem to find the optimal weighted alignments between substrings of \(A\) and substrings of \(B\). Based on [15], we present an algorithm to solve the ALCS problem that takes \(O(n_a n_b)\) time and \(O(n_b)\) space. A short version of this paper appeared in [5].

2. The ALCS problem

Strings of symbols will be denoted by upper-case letters, such as \(A\) and \(B\). Individual symbols of a string will be identified by the string name (lower-case) with a subscripts index. The indices start at 1, for the first symbol. The length of the string will be denoted by \(n\) subscripted by the lower-case string name. For example, \(A = a_1 a_2 a_3 \ldots a_{n_a - 1} a_{n_a}\). A substring is denoted by the upper-case letter of the original string and indices that indicate the initial (subscripted) position and final (superscripted) position of the symbols in the original string. For example, \(A^5_3 = a_3 a_4 a_5 a_6\).

Given two strings \(A\) and \(B\), the LCS problem finds the longest string \(C\) that is a subsequence of \(A\) and \(B\). Fig. 1 illustrates the alignment of strings. Each string is placed in a row, in such a way that no column contains two different symbols. Each column may contain one blank space, though. Columns with equal symbols have a score of 1, the others have score 0. The array with the greatest possible sum of column scores gives the LCS.

LCS algorithms are based on dynamic programming or primal–dual techniques and present a worst-case time complexity of \(O(n_a n_b)\). See [7,13,14,16] for surveys.

For two strings \(A\) and \(B\), \(|A| = n_a\) and \(|B| = n_b\), the ALCS problem solves the LCS problem for \(A\) and all substrings of \(B\), that is, solve the LCS for all pairs \((A, B^j_i)\), where \(1 \leq i \leq j \leq n_b\). Often we are more interested in the lengths rather than the subsequences proper.

The grid directed acyclic graph (GDAG) for the ALCS problem has \(n_a + 1\) rows and \(n_b + 1\) columns of vertices. Labeling rows from 0 to \(n_a\) and the columns from 0 to \(n_b\), each vertex of a GDAG \(G\) is identified by \(G(i, j)\), where \(0 \leq i \leq n_a\) and \(0 \leq j \leq n_b\). The arcs in this GDAG have weights 0 or 1, defined as follows. Fig. 2 shows an example, with the diagonal arcs of weight 0 omitted:

- (diagonal arcs) for \(1 \leq i \leq n_a\) and \(1 \leq j \leq n_b\) there is an arc from \(G(i - 1, j - 1)\) to \(G(i, j)\) with weight 1 if \(a_i = b_j\), 0 otherwise;

\[
\begin{array}{cccccccc}
A & x & y & w & w & y & x & - & - & w \\
B & x & - & w & w & y & y & z & - \\
\text{score} & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 5
\end{array}
\]

Fig. 1. Example of a solution to the LCS Problem. For strings xywwyxw and xwwxyzx the longest common subsequence is xwwyx.

Fig. 2. GDAG for the ALCS problem. The strings being compared are \(A = yxxyzyzx\) and \(B = yxxyzxyxyzx\). The diagonal arcs shown have weight 1, those with weight 0 are omitted.
• (vertical arcs) for \(0 \leq i \leq n_a\) and \(1 \leq j \leq n_b\) there is an arc from \(G(i, j - 1)\) to \(G(i, j)\) with weight 0;
• (horizontal arcs) for \(1 \leq i \leq n_a\) and \(0 \leq j \leq n_b\) there is an arc from \(G(i - 1, j)\) to \(G(i, j)\) with weight 0.

Denote the vertices at the top row of \(G\) by \(G(i)\) and the ones at the bottom row by \(F_G(i)\).

To solve the LCS problem, one looks for the highest weighted path from \(G(0, 0)\) to \(G(n_a, n_b)\), whose total weight is the length of the longest common subsequence between \(A\) and \(B\). This path gives us the actual subsequence: each vertical arc represents a deletion in \(A\), each horizontal arc represents a deletion in \(B\) and each diagonal arc (of weight 1) represents matched symbols in \(A\) and \(B\). A dynamic programming technique is used in the LCS algorithm by Hirschberg [8] that requires \(O(n_a n_b)\) time and \(O(n_a + n_b)\) space.

For the ALCS problem we are interested in more paths than just the one from \(G(0, 0)\) to \(G(n_a, n_b)\). We define \(C_G(i, j)\) as follows.

**Definition 2.1** \((C_G(i, j))\). For \(0 \leq i \leq j \leq n_b\), \(C_G(i, j)\) is the largest total weight of a path from \(T_G(i)\) to \(F_G(j)\) in \(G\).

\(C_G(i, j)\) represents the length of the LCS of \(A\) and the substring \(B^{i+1}_{i+1}\). When \(i = j\), there is no such substring and the only path from \(T_G(i)\) to \(F_G(j)\) is formed exclusively by vertical arcs and has total weight 0. When \(i > j\), \(B^{i+1}_{i+1}\) also does not exist and we adopt \(C_G(i, j) = 0\).

The values of \(C_G(i, j)\) have the following interesting properties.

**Properties 2.1.**
1. For all \(i \) \((0 \leq i \leq n_b)\) and all \(j \) \((1 \leq j \leq n_b)\) we have \(C_G(i, j) = C_G(i, j - 1)\) or \(C_G(i, j) = C_G(i, j - 1) + 1\).
2. For all \(i \) \((1 \leq i \leq n_b)\) and all \(j \) \((0 \leq j \leq n_b)\) we have \(C_G(i - 1, j) = C_G(i, j)\) or \(C_G(i - 1, j) = C_G(i, j) + 1\).
3. For all \(i\) and all \(j\) \((1 \leq i < j \leq n_b)\) if \(C_G(i - 1, j) = C_G(i - 1, j - 1) + 1\) then \(C_G(i, j) = C_G(i, j - 1) + 1\).

**Proof.** We prove Properties 2.1(1) and 2.1(3). The proof of 2.1(2) is similar.

Naturally, \(C_G(i, j)\) and \(C_G(i, j - 1)\) are integers. When \(i \geq j\) we have \(C_G(i, j) = C_G(i, j - 1) = 0\). When \(i < j\), looking at the paths that start at vertex \(T_G(i)\) it becomes clear that \(C_G(i, j) \geq C_G(i, j - 1)\), since the best path to vertex \(F_G(j - 1)\) can be extended to \(F_G(j)\) by a horizontal arc with weight 0. On the other hand, as all arcs with weight 1 are diagonal arcs, it is clear that \(C_G(i, j) - C_G(i, j - 1) \leq 1\). That is because a path from \(T_G(i)\) to \(F_G(j - 1)\) can be created based on the best path to \(F_G(j)\), by eliminating the vertices in column \(j\) and completing the path to \(F_G(j - 1)\) with vertical arcs. At most one diagonal is eliminated.

To prove 2.1(3), we assume that \(C_G(i - 1, j) = C_G(i - 1, j - 1) + 1\) for a certain pair \((i, j)\), \(1 \leq i < j \leq n_b\). Let \(C_1\) be the leftmost path from \(T_G(i - 1)\) to \(F_G(j)\) with total weight \(C_G(i - 1, j)\). \(C_1\) is such that no path from \(T_G(i - 1)\) to \(F_G(j)\) with the same weight has any vertex to the left of a vertex of \(C_1\) in the same row. Let \(C_2\) be the leftmost path from \(T_G(i)\) to \(F_G(j - 1)\) with total weight \(C_G(i, j - 1)\). It is easy to show that such leftmost paths exist and that they have a unique common subpath with at least one vertex. Let \(C_{1a}\) and \(C_{1b}\) be the subpaths of \(C_1\) that do not belong to \(C_2\), \(C_{1a}\) is the subpath starting at \(T_G(i - 1)\) and \(C_{1b}\) is the subpath ending at \(F_G(j)\). Let \(C_{2a}\) and \(C_{2b}\) be subpaths of \(C_2\) defined analogously. The common subpath is denoted \(C\). See Fig. 3.

Naturally, the weight of \(C_{1a}\) must be equal to the weight of \(C_{2b}\) plus 1. Otherwise we would have a path from \(T_G(i - 1)\) to \(F_G(j - 1)\) (formed by \(C_{1a}\), \(C\) and \(C_{2b}\)) with the same weight as \(C_1\), which contradicts our hypothesis. Therefore, the path composed of \(C_{2a}\), \(C\) and \(C_{1b}\) will have a total weight equal to the weight of \(C_2\) plus 1, which implies \(C_G(i, j) > C_G(i, j - 1)\). By 2.1(1) we conclude that \(C_G(i, j) = C_G(i, j - 1) + 1\). □

Property 2.1(1) indicates that for a fixed value of \(i\), the values of \(C_G(i, 0), C_G(i, 1), \ldots, C_G(i, n_b)\) form a non-decreasing sequence that can be defined implicitly, by indicating just the values of \(j\) for which \(C_G(i, j) > C_G(i, j - 1)\). This fact is used in the CREW-PRAM algorithm for the LCS problem of Lin and Lu [11] who have introduced the following definition of the cost matrix \(D_G\).

**Definition 2.2** \((D_G)\). Let \(G\) be the GDAG for the ALCS problem with strings \(A\) and \(B\). For \(0 \leq i \leq n_b\), \(D_G(i, 0) = i\) and for \(1 \leq k \leq n_a\), \(D_G(i, k)\) indicates the value of \(j\) such that \(C_G(i, j) = k\) and \(C_G(i, j - 1) = k - 1\). If there is no such value, \(D_G(i, k) = \infty\).
This definition implies $\text{DG}(i, j) \leq n_a$. We define $\text{DG}(i, 0) = i$ instead of 0 to simplify the presentation of the algorithm. We denote $\text{DG}_i$ the row $i$ of $\text{DG}$. As an example, Figs. 4 and 5 show $\text{CG}(i, j)$ and $\text{DG}(i, k)$ corresponding to the GDAG of Fig. 2.

From the values of $\text{DG}(i, k)$ we can obtain the values of $\text{CG}(i, j)$. Furthermore, the following properties can be derived from Property 2.1(3) (see also [11]) and indicate that the information contained in $\text{DG}$ can be represented in $O(n_a + n_b)$ space, instead of $O(n_a n_b)$ space by using the straightforward representation.

**Properties 2.2.** (1) If $k_1 < k_2$ and $\text{DG}(i_1, k_1) \neq \infty$ then $\text{DG}(i_1, k_1) < \text{DG}(i_2, k_2)$.

(2) If $i_1 < i_2$ then $\text{DG}(i_1, k) \leq \text{DG}(i_2, k)$.

(3) If $\text{DG}(i_1, k_1) = j_1$ and $j_1 \neq \infty$ then for all $i, i_1 \leq i \leq j_1$, there is $k$ such that $\text{DG}(i, k) = j_1$. In other words, if $j_1$ appears in row $\text{DG}_i$ then it also appears in all the rows up to row $\text{DG}_i$.

(4) If $\text{DG}(i_1, k_1) = \text{DG}(i_2, k_2) = j_1$ then for all $i, i_1 \leq i \leq i_2$, there is $k$ such that $\text{DG}(i, k) = j_1$. In other words, if $j_1$ appears in rows $\text{DG}_i$ and $\text{DG}_i$ then it also appears in rows between $\text{DG}_i$ and $\text{DG}_i$.

**Proof.** If $k_1 < k_2$ and $j_1 = \text{DG}(i, k_1) \neq \infty$ then $\text{CG}(i, j_1) = k_1$. $\text{CG}(i, j)$ does not decrease with $j$, so if there is $j$ such that $\text{CG}(i, j) = k_2$ then $j > j_1$. If there is no such $j$ then $\text{DG}(i, k_2) = \infty$. In either case we have $\text{DG}(i, k_1) < \text{DG}(i, k_2)$, thus proving (1).
If $i_1 < i_2$, by Property 2.1(2) $C_G(i_2, j) = C_G(i_1, j)$ for all $j$. If $D_G(i_1, k) > D_G(i_2, k) \neq \infty$ for some value of $k$, then for $j = D_G(i_2, k)$ we have $C_G(i_2, j) = k > C_G(i_1, j)$, a contradiction. Thus (2) follows.

The proof of (3) is by induction on $i$. The presence of $j_1$ in $D^j_G$ follows from Definition 2.2. Now let us suppose that for a certain value of $i$ ($i_1 \leq i < j_1 \neq \infty$) we have $D_G(i, k) = j_1$ for some value of $k$ (the specific value of $k$ is not important). This means that $C_G(i, j_1) = k = C_G(i, j_1 - 1) + 1$. By Property 2.1(3) we have $C_G(i + 1, j_1) = C_G(i + 1, j_1 - 1) + 1$, which means that there is some $k$ such that $D_G(i + 1, k) = j_1$, concluding the inductive step. This step applies while the condition for Property 2.1(3) ($i + 1 < j_1$) is true. So $j_1$ appears in every row from $D^i_G$ to $D^{i+1}_G$.

Property (4) can be easily derived from (3), since $D_G(i, k)$ can only be $j_1$ if $i \leq j_1$. □

Properties 2.2(2) and 2.2(3) are used to reduce the space necessary to represent $D_G$, because they show that two consecutive rows of $D_G$ are similar. We now state this more precisely, as follows.

Properties 2.3. For $0 \leq i < n_b$,

(1) There is one unique finite element of $D^i_G$ that does not appear in $D^{i+1}_G$, which is $D_G(i, 0) = i$.

(2) There is at most one finite element of $D^{i+1}_G$ that does not appear in $D^i_G$.

Proof. Property (1) follows directly from Property 2.2(3). If $j_1$ appears in row $D^i_G$, then it appears in all the following rows, up to row $j_1$. As $D_G(i, 0) = i$, this is the last row in which $i$ appears. All the other elements from $D^i_G$ are larger than $i$ (by Property 2.2(1)) and also appear in $D^{i+1}_G$.

By Property 2.2(2) we have that $D^{i+1}_G$ cannot have more finite elements than $D^i_G$. As only one finite element from $D^i_G$ does not appear in $D^{i+1}_G$, there is at the most one new element in $D^{i+1}_G$. □

Properties 2.3(1) and 2.3(2) indicate that we can transform one row of $D_G$ into the next row by the removal of one element (the first one) and the insertion of another element (finite or not).

This suggests a representation for $D_G$ in $O(n_a + n_b)$ space. This representation is composed of $D^0_G$ (the first row of $D_G$), that has size $n_a + 1$, and a vector $V_G$ that has size $n_b$, defined below.
Definition 2.3 \((V_G)\). For \(1 \leq i \leq n_b\), \(V_G(i)\) is the value of the finite element that is present in row \(D_G^i\) but not in row \(D_G^{i-1}\). If there is no such element, then \(V_G(i) = \infty\).

Fig. 6 shows \(V_G\) for the GDAG of Fig. 2. It is instructive to compare the three representations, namely, \(C_G\) (Fig. 4), \(D_G\) (Fig. 5) and \(V_G\) (Fig. 6). Figs. 5 and 6 show that \(D_0^G\) (first row of \(D_G\)) and \(V_G\) can be used together to reconstruct \(D_G\). It is easy to see that the elements of \(D_0^G\) are the integers from 0 to \(n_b\) which are not present in \(V_G\), so \(V_G\) suffices to reconstruct \(D_G\).

3. The ALCS algorithm

We return to the GDAG \(G\) and present some additional properties that are important to the design of the proposed ALCS algorithm.

For each vertex of the GDAG \(G\) we need to determine some information about the distance between this vertex and each vertex on the top row of \(G\). For this we define \(C_G^l(i, j)\) as follows.

Definition 3.1 \((C_G^l(i, j))\). For \(0 \leq l \leq n_a\), \(0 \leq i \leq n_b\) and \(0 \leq j \leq n_b\), \(C_G^l(i, j)\) is the total weight of the path with the highest total weight between vertices \((0, i)\) and \((l, j)\), if there is such path. If there is no such path (in the case that \(i > j\)), \(C_G^l(i, j) = 0\).

This definition is an extension of definition of \(C_G(i, j)\) that deals with the internal vertices of \(G\) (notice that \(C_G(i, j) = C_G^{n_a}(i, j)\)). Now we have the following properties, that deal with neighboring vertices in \(G\).

Properties 3.1. For all \(l (0 \leq l \leq n_a)\):

1. For all \(i (0 \leq i \leq n_b)\) and all \(j (1 \leq j \leq n_b)\) we have \(C_G^l(i, j) = C_G^l(i, j - 1)\) or \(C_G^l(i, j) = C_G^l(i, j - 1) + 1\).
2. For all \(i\) and all \(j (0 < i < j \leq n_b)\) we have that if \(C_G^l(i-1, j) = C_G^l(i-1, j - 1) + 1\) then \(C_G^l(i, j) = C_G^l(i, j - 1) + 1\).

The proof of these properties is omitted, for it is similar to the proof of Property 2.1. The inclusion of \(l\) in these properties does not affect their nature.

Properties 3.1 are related to the differences between distances to two internal vertices of \(G\) that are neighbors on the same row. The following properties are related to internal vertices that are neighbors on the same column.
Properties 3.2. For all $l$ ($1 \leq l \leq n_a$):

1. For all $i$ ($0 \leq i \leq n_b$) and all $j$ ($0 \leq j \leq n_b$) we have $C_G^l(i, j) = C_G^{l-1}(i, j)$ or $C_G^l(i, j) = C_G^{l-1}(i, j) + 1$.

2. For all $i$ and all $j$ ($0 \leq i \leq j \leq n_b$) we have that if $C_G^l(i - 1, j) = C_G^{l-1}(i - 1, j)$ then $C_G^l(i, j) = C_G^{l-1}(i, j)$.

The proof of these properties is also omitted. By Properties 3.1 and 3.2, we can encode variations in the weights of the paths to internal vertices of the GDAG in an efficient way. If we pick a vertex on the upper row of $G$, say $T_G(i)$, and determine the best possible paths from it to two neighbor vertices on a same row, say $(l, j - 1)$ and $(l, j)$, these paths will have the same weight if $i$ is below certain limit. For values of $i$ equal or above this limit, the weight of the path to $(l, j)$ will be larger (by 1) than the weight of the path to $(l, j - 1)$. The value of this limit depends on $l$ and $j$ and will be called $i_h(l, j)$.

Similarly, the weights of the best paths from $T_G(i)$ to neighbor vertices on a column, $(l - 1, j)$ and $(l, j)$, differ by 1 if $i$ is below a certain limit $i_v(l, j)$. If $i$ is on or above this limit, the paths have the same weight.

Formally, we define $i_h(l, j)$ and $i_v(l, j)$ as follows.

Definition 3.2 ($i_h(l, j)$). For all $(l, j)$, $0 \leq l \leq n_a$ and $1 \leq j \leq n_b$, $i_h(l, j)$ is the smallest value of $i < j$ such that $C_G^l(i, j) = C_G^l(i, j - 1) + 1$. If there is no such $i$, $i_h(l, j) = j$.

Definition 3.3 ($i_v(l, j)$). For all $(l, j)$, $1 \leq l \leq n_a$ and $1 \leq j \leq n_b$, $i_v(l, j)$ is the smallest value of $i \leq j$ such that $C_G^l(i, j) = C_G^{l-1}(i, j)$.

The algorithm is based on the determination of the limits $i_h$ and $i_v$ for all the vertices of the GDAG. The first value of $i$ for which there is a $k$ such that $D_G(i, k) = j$ is $i_h(n_a, j)$. For values of $i$ above this limit, $j$ continues to appear in $D_G^l$, which means that there is a difference between the best paths from $T_G(i)$ to $(n_a, j - 1)$ and $(n_a, j)$.

Besides $i_h$ and $i_v$, for each vertex we need to determine two limits, $i_1$ and $i_2$, explained below. Given a vertex $(l, j)$, with $0 < l \leq n_a$ and $0 < j \leq n_b$, the best paths from the top of the GDAG to it must have $(l, j - 1)$, $(l - 1, j - 1)$ or $(l - 1, j)$ as the next-to-last vertex. Since two best leftmost paths cannot cross, we have the following Property 3.3, illustrated in Fig. 7.

Property 3.3. For all $(l, j)$, $1 \leq l \leq n_a$ and $0 < j \leq n_b$, there are values $i_1$ and $i_2$ such that the best (leftmost) path from $T_G(i)$ to $(l, j)$ has as the next-to-last vertex:

- $(l, j - 1)$ if $0 \leq i < i_1$,
- $(l - 1, j - 1)$ if $i_1 \leq i < i_2$,
- $(l - 1, j)$ if $i_2 \leq i < j$.

The determination of the four aforementioned limits ($i_h$, $i_v$, $i_1$ and $i_2$) is done vertex by vertex, by sweeping the GDAG row by row (or column by column). To compute the limits for a vertex $(l, j)$ it is necessary to know only $i_h(l - 1, j)$ and $i_v(l, j - 1)$, for these values indicate differences between paths to vertices adjacent to $(l, j)$. We will call these vertices candidate vertices. Two distinct cases must be considered separately.
Therefore, \( i_1 = i_2 = i_h(l-1, j) \), \( i_v(l, j) = i_h(l-1, j) \) and \( i_v(l, j) = i_v(l, j-1) \).

**Case 1:** If \( a_i \neq b_j \), that is, the arc from \((l-1, j-1)\) to \((l, j)\) has weight 0, this arc can be ignored and the next-to-last vertex of any path to \((l, j)\) cannot be \((l-1, j-1)\). In this case, \( i_l = i_2 \). There are two possibilities from this point:

**Case 1(i).** \( i_v(l, j-1) < i_h(l-1, j) \): there are three ranges of values to consider for \( i \) in the choice of the next-to-last vertex in the path to \((l, j)\):

- For \( 0 \leq i < i_v(l, j-1) \) the best path from \( T_G(i) \) to \((l, j-1)\) is better than the best path to the other two candidates, so the chosen vertex is \((l, j-1)\).
- \( i_v(l, j-1) < i < i_h(l-1, j) \): the best paths to each of the three candidates have the same weight, so the chosen candidate is again \((l, j-1)\) (which gives the leftmost path).
- \( i_h(l-1, j) \leq i < j \): the best path from \( T_G(i) \) to \((l-1, j)\) is better than the path to the other two candidates, so \((l-1, j)\) is chosen.

**Case 1(ii).** \( i_v(l, j-1) > i_h(l-1, j) \): there are again three ranges of values to consider for \( i \):

- For \( 0 \leq i < i_v(l-1, j) \): the chosen vertex is \((l-1, j)\);
- \( i_v(l-1, j) \leq i < i_h(l-1, j) \): the best paths from \( T_G(i) \) to \((l, j-1)\) and to \((l-1, j)\) have the same weight (the path to \((l-1, j-1)\) has a lower weight). The leftmost vertex \((l, j-1)\) is chosen.
- \( i_v(l, j-1) \leq i < j \): the chosen vertex is \((l-1, j)\).

So we have \( i_1 = i_2 = i_v(l, j-1) \), \( i_h(l, j) = i_v(l, j-1) \) and \( i_v(l, j) = i_h(l-1, j) \).

In Case 1 we have \( i_1 = i_2 = i_v(l, j) = \max(i_v(l, j-1), i_h(l-1, j)) \) and \( i_v(l, j) = \min(i_v(l, j-1), i_h(l-1, j)) \).

**Case 2:** If \( a_l = b_j \), that is, the arc from \((l-1, j-1)\) to \((l, j)\) has weight 1. We have to consider the three candidates, but \((l-1, j-1)\) has the advantage of being connected to \((i, j)\) through an arc of weight 1. In fact, as we search for the leftmost path among the best ones to \((i, j)\), no path will pass through \((l-1, j)\) except the one from \( T_G(j) \), and so we have \( i_2 = j \).

When \( i < i_v(l, j-1) \) the best path from \( T_G(i) \) to \((l, j-1)\) has a lower weight than the one to \((l-1, j-1)\), compensating the weight 1 of the arc from \((l-1, j-1)\) to \((l, j)\). So, \( i_1 = i_v(l, j-1) \) and \( i_h(l, j) = i_v(l, j-1) \). For similar reasons, we have \( i_v(l, j) = i_h(l-1, j) \).

This reasoning applies to the vertices that are not on the upper and left borders of the GDAG. For the vertices on the upper border we have \( i_h(0, j) = j \) (1 \( \leq j \leq n_b \)) and for the vertices on the left border we have \( i_v(l, 0) = 0 \) (1 \( \leq l \leq n_a \)). The other values are not important.

The values of \( i_h(n_a, j) \) with 1 \( \leq j \leq n_b \) (corresponding to the vertices on the lower border of the GDAG) represent the answer for our basic algorithm. We will call \( I_G \) the vector that contains these values. From this vector we also obtain \( D^0_G \) and \( V_G \).

At first we make \( D^0_G(0) = 0 \). Then, if for a certain \( j \) we have \( I_G(j) = i_h(n_a, j) = 0 \), this means that \( j \) is an element of \( D^0_G \). If \( I_G(j) = i, 1 \leq i \leq j \), this means that the first row of \( D_G \) in which \( j \) appears is \( D_i \), so \( V_G(i) = j \).

**Algorithm 1** The ALCS Algorithm

**Require:** Strings \( A = a_1a_2 \ldots a_{n_a} \) and \( B = b_1b_2 \ldots b_{n_b} \).

**Ensure:** Vectors \( I_G, D^0_G \) and \( V_G \) related to the strings \( A \) and \( B \).

1. for \( j \leftarrow 0 \) to \( n_b \) do
2. \( i_h(0, j) \leftarrow j \)
3. end for
4. for \( l \leftarrow 0 \) to \( n_a \) do
5. \( i_v(l, 0) \leftarrow 0 \)
6. end for
7. for \( l \leftarrow 1 \) to \( n_a \) do
8. for \( j \leftarrow 1 \) to \( n_b \) do
9. if \( a_l \neq b_j \) then
10. \( i_h(l, j) \leftarrow \max(i_v(l, j-1), i_h(l-1, j)) \)
11. \( i_v(l, j) \leftarrow \min(i_v(l, j-1), i_h(l-1, j)) \)
12. end if
13. end for
14. end for
15. end for
12:     else
13:         \( i_h(l, j) \leftarrow i_v(l, j - 1) \)
14:         \( i_v(l, j) \leftarrow i_h(l - 1, j) \)
15:     end if
16: end for
17: end for
18: for \( j \leftarrow 1 \) to \( n_b \) do
19:     \( V_G(j) \leftarrow \infty \)
20: end for
21: \( D^0_G(0) \leftarrow 0 \)
22: \( i \leftarrow 1 \)
23: for \( j \leftarrow 1 \) to \( n_b \) do
24:     if \( I_G(j) = 0 \) then
25:         \( D^0_G(i) \leftarrow j \)
26:         \( i \leftarrow i + 1 \)
27:     else
28:         \( V_G(I_G(j)) \leftarrow j \)
29:     end if
30: end for
31: for \( l \leftarrow i \) to \( n_a \) do
32:     \( D^0_G(l) \leftarrow \infty \)
33: end for

Algorithm 1 makes this procedure explicit. Notice that the limits \( i_1 \) and \( i_2 \) are not necessary for the determination of \( D^0_G \) and \( V_G \), so these limits are not included in the algorithm. However, if it is necessary to make possible the fast retrieval of the actual paths between \( T_G(i) \) and \( F_G(j) \) (for any \( i \) and \( j \)), then the determination and storage of the limits \( i_1 \) and \( i_2 \) are needed. These limits allow the reconstruction of the path, starting at \( F_G(j) \) and going back to \( T_G(i) \), in time \( O(n_a + n_b) \). Theorem 1 summarizes the main result.

An example of the results of this algorithm can be seen in Fig. 8, where the values of \( i_h \) and \( i_v \) are shown for each vertex of the GDAG of Fig. 2. The results in the last row of the figure (related to \( i_h(n_a, j) \)) lead correctly to the results shown in Figs. 5 and 6.

**Theorem 1.** Given two strings \( A \) and \( B \) of lengths \( n_a \) and \( n_b \), respectively, Algorithm 1 runs in \( O(n_a n_b) \) time and \( O(n_a + n_b) \) space (or \( O(n_a n_b) \) to allow the retrieval of the actual best paths).

**Proof.** The execution time is \( O(n_a n_b) \) and results from the nested loops from line 7 to line 17 of Algorithm 1. In the case where only the distances are important, we can discard the values of \( i_h(l, j) \) after the row \( l + 1 \) has been processed. So, to use the values \( i_h(l, j) \), \( O(n_b) \) space is required. The value of \( i_v(l, j) \) must be kept only for the calculation of \( i_v(l, j + 1) \), so the space required by the values of \( i_v(l, j) \) is only \( O(1) \).

The storage of the resulting vectors \( I_G \) (space \( O(n_b) \)), \( D^0_G \) (space \( O(n_a) \)) and \( V_G \) (space \( O(n_b) \)) requires \( O(n_a + n_b) \) space.

If the retrieval of the paths is required, the necessary space becomes \( O(n_a n_b) \), for the storage of \( i_1 \) and \( i_2 \) for all vertices. □

4. **Querying \( C_G \)**

The previous algorithm builds vectors \( I_G \), \( D^0_G \) and \( V_G \). To find the actual values of matrix \( C_G \) we may have to build other data structures or spend an extra time in each query.

From the definition of \( i_h \), we know that \( I_G(j) \) is the smaller value of \( i \) for which \( C_G(i, j) = C_G(i, j - 1) + 1 \). Given \( C_G(i, j - 1) \) we can obtain \( C_G(i, j) \) (or vice versa) in \( O(1) \) time, by checking if \( I_G(j) > i \). Therefore all values of a single row of \( C_G \) may be obtained sequentially in \( O(1) \) time per element, starting with \( C_G(i, i) = 0 \).
To obtain $C_G(i, j)$ from $C_G(i - 1, j)$ (or vice versa), one has to know if $i$ occurs in $I_G$ before or at position $j$. If it does, $C_G(i, j) = C_G(i - 1, j)$, otherwise $C_G(i, j) = C_G(i - 1, j)$. Another vector of size $n_b$, indicating the position of any number in $I_G$, may be built in $O(n_b)$ time (notice that any number larger than 0 can appear only once in $I_G$). With this vector, all elements of a single column of $C_G$ may be obtained in $O(1)$ time per element.

If fast ($O(1)$ time) random accesses to $C_G$ are necessary, the previous results permit the construction of $C_G$ in $O(n^2 b)$ time/space.

In some applications (as the one presented in [4,3,1]), the matrix $D_G$ is more relevant. From $D_G^0$ and $V_G$ it is possible to build in $O(n_b \sqrt{n_a})$ time/space a data structure that allows queries for values of $D_G$ in constant time. Details of this data structure are presented in the previous references. It basically uses the fact that the rows of $D_G$ are very similar and can be divided into subvectors of size close to $\sqrt{n_a}$. Subvectors of distinct rows of $D_G$ may overlap in memory.

With this last data structure, random queries to $C_G$ may be answered in $O(\log n_a)$ time, using binary search. This is useful when the full $C_G$ cannot be stored, but $O(n_b \sqrt{n_a})$ space is reasonable.

5. Conclusion

We consider the ALCS (all-substring longest common subsequence) problem for strings $A$ and $B$, $|A| = n_a$ and $|B| = n_b$. The basic algorithm takes $O(n_a n_b)$ time and $O(n_a + n_b)$ space and, depending on the queries that will be necessary, extra $O(n^2 b)$ time and space are required.

Although the ALCS was the problem under consideration, actually the GDAG, and not the original strings, is used. Other problems may be approached by the techniques presented here. For example, alignments where matches between symbols are given by some non-transitive relation may be used instead of LCS.
The algorithm presented here can be easily adapted to solve the problem incrementally, that is, presenting the solution for a certain prefix \( A \) based on the solution for a smaller prefix. Other adaptations, including new data structures more suitable for certain applications, may appear in the future.

Acknowledgment

We thank the anonymous referees for their helpful comments.

References