Journal of Mathematical Sciences, Vol. 158, No. 5, 2009

FASTER SUBSEQUENCE RECOGNITION IN COMPRESSED STRINGS

A. Tiskin*

UDC 519.16

Computation on compressed strings is one of the key approaches to processing massive data sets. We consider local subsequence recognition problems on strings compressed by straight-line programs (SLP)*, which is closely related to Lempel–Ziv compression. For an* SLP*-compressed text of length* $\overline{m}$*, and an uncompressed pattern of length* n*, Cégielski et al. gave an algorithm for local subsequence recognition running in time* $O(\overline{m}n^2 \log n)$*. We improve the running time to* $O(\overline{m}n^{1.5})$*. Our algorithm can also be used to compute the longest common subsequence between a compressed text and an uncompressed pattern in time* $O(\overline{m}n^{1.5})$*; the same problem with a compressed pattern is known to be* NP*-hard. Bibliography:* 22 *titles.*

1. Introduction

Computation on compressed strings is one of the key approaches to processing massive data sets. It has long been known that certain algorithmic problems can be solved directly on a compressed string, without first decompressing it; see [4, 11] for references.

One of the most general string compression methods is compression by straight-line programs (SLP) [16]. In particular, SLP compression captures the well-known LZ and LZW algorithms [21, 22, 20]. Various pattern matching problems on SLP-compressed strings have been studied; see, e.g., [4] for references. Cégielski et al. [4] considered subsequence recognition problems on SLP-compressed strings. For an SLP-compressed text of length $\overline{m}$, and an uncompressed pattern of length n, they gave several algorithms for global and local subsequence recognition, running in time $O(\overline{m}n^2 \log n)$.

In this paper, we improve on the results of [4] as follows. First, we describe a simple folklore algorithm for global subsequence recognition on an SLP-compressed text, running in time $O(\overline{m}n)$. Then, we consider the more general partial semi-local longest common subsequence (LCS) problem, which consists in computing implicitly the LCS between a compressed text and every substring of an uncompressed pattern. The same problem with a compressed pattern is known to be NP-hard. For the partial semi-local LCS problem, we propose a new algorithm, running in time $O(\overline{m}n^{1.5})$. Our algorithm is based on the partial highest-score matrix multiplication technique presented in [18]. We then extend this method to the several versions of local subsequence recognition considered in [4], for each obtaining an algorithm running in the same asymptotic time $O(\overline{m}n^{1.5})$.

This paper is a sequel to the papers [17, 18]; we recall most of their relevant material here for completeness.

2. Subsequences in a compressed text

We consider strings of characters from a fixed finite alphabet, denoting string concatenation by juxtaposition. Given a string, we distinguish between its contiguous *substrings*, and not necessarily contiguous *subsequences*. Special cases of a substring are *a prefix* and *a suffix* of a string. Given a string a of length m, we use the *take/drop notation* of [19], $a \uparrow k$, $a \downarrow k$, $a \upharpoonright k$, $a \downharpoonright k$, to denote, respectively, its prefix of length k, suffix of length $m-k$, suffix of length k, and prefix of length $m-k$. For two strings $a = \alpha_1\alpha_2 \dots \alpha_m$ and $b = \beta_1\beta_2 \dots \beta_n$ of lengths m and n, respectively, the *longest common subsequence* (LCS) *problem* consists in computing the length of the longest string that is a subsequence both of a and b. We will call this length the *LCS score* of the strings.

Let T be a string of length m (typically large). The string T will be represented implicitly by a *straight-line program* (SLP) of *length* $\overline{m}$, which is a sequence of $\overline{m}$ *statements*. Each statement r, $1 \le r \le \overline{m}$, has either the form $T_r = \alpha$, where α is an alphabet character, or the form $T_r = T_s T_t$, where $1 \le s,\ t < r$. We identify every symbol T_r with the string it represents; in particular, we have $T = T_{\overline{m}}$. Note that $m \ge \overline{m}$, and that in general the uncompressed text length m can be exponential in the SLP-compressed text length $\overline{m}$.

Our goal is to design efficient algorithms on SLP-compressed texts. While we do not allow text decompression (since, in the worst case, this could be extremely inefficient), we will assume that standard arithmetic operations on integers up to m can be performed in constant time. This assumption is necessary, since the counting version of our problem produces a numerical output that may be as high as $O(m)$. The same assumption on the computation model is made implicitly in [4].

*Department of Computer Science, University of Warwick, Coventry, United Kingdom, e-mail: tiskin@dcs.warwick.ac.uk.

Published in *Zapiski Nauchnykh Seminarov POMI*, Vol. 358, 2008, pp. 282–300. Original article submitted June 10, 2007.
1072-3374/09/1585-0759

The LCS problem on uncompressed strings is a classical problem; see, e.g., [7, 9] for the background and references. Given input strings of lengths m, n, the LCS problem can be solved in time $O\left(\frac{mn}{\log(m+n)}\right)$, assuming m and n are reasonably close [12, 6]. The LCS problem on two SLP-compressed strings is considered in [11], and proven to be NP-hard. In this paper, we consider the LCS problem on two input strings, one of which is SLP-compressed and the other uncompressed. This problem can be regarded as a special case of computing the edit distance between a context-free language given by a grammar of size $\overline{m}$, and a string of size n. For this more general problem, Myers [13] gives an algorithm running in time $O(\overline{m}n^3 + \overline{m}\log\overline{m} \cdot n^2)$.

From now on, we will assume that a string T (the *text string*) of length m is represented by an SLP of length $\overline{m}$, and that a string P (the *pattern string*) of length n is represented explicitly. Following [4, 11], we study the problem of recognizing in T subsequences identical to P, which is closely related to the LCS problem.

Definition 1. *The* (global) subsequence recognition problem *consists in deciding whether the string T contains the string P as a subsequence.*

The subsequence recognition problem on uncompressed strings is a classical problem, considered, e.g., in [1] as the "subsequence matching problem." The subsequence recognition problem on an SLP-compressed text is considered in [4] as Problem 1, with an algorithm running in time $O(\overline{m}n^2 \log n)$.

In addition to global subsequence recognition, it is useful to consider text subsequences locally, i.e., in substrings of T. In this context, we will call the substrings of T *windows*. We will say that a string a contains a string b *minimally* as a subsequence if b is a subsequence in a, but not in any proper substring of a. Even with this restriction, the number of substrings in T containing P minimally as a subsequence may be as high as $O(m)$, so just listing them all may require time exponential in $\overline{m}$. The same is true if, instead of minimal substrings, we consider all substrings of T of a fixed length. Therefore, it is sensible to define local subsequence recognition as a family of counting problems.

Definition 2. *The* minimal-window subsequence recognition problem *consists in counting the number of windows in a string T containing a string P minimally as a subsequence.*

Definition 3. *The* fixed-window subsequence recognition problem *consists in counting the number of windows of a given length w in a string T containing a string P as a subsequence.*

The minimal-window and fixed-window subsequence recognition problems on uncompressed strings are considered in [8] as "episode matching problems" (see also [5] and references therein). The same problems on an SLP-compressed text and an uncompressed pattern are considered in [4] as Problems 2 and 3 (a special case of 2) and 4. Additionally, the same paper considers the *bounded minimal-window subsequence recognition problem* (counting the number of windows in T of length at most w containing P minimally as a subsequence) as Problem 5. For all these problems, the paper [4] gives algorithms running in time $O(\overline{m}n^2 \log n)$.

3. Semi-local longest common subsequences

In this section and the next, we recall the algorithmic framework developed in [17, 18]. This framework is subsequently used to solve the compressed subsequence recognition problems introduced in the previous section.

In [17], we introduced the following problem.

Definition 4. *The* all semi-local LCS problem *consists in computing the LCS scores on substrings of strings a and b as follows:*

- *the* all string-substring LCS problem*: a against every substring of b;*
- *the* all prefix-suffix LCS problem*: every prefix of a against every suffix of b;*
- *symmetrically, the* all substring-string LCS problem *and the* all suffix-prefix LCS problem*, defined as above but with the roles of a and b exchanged.*

It turns out that this is a very natural and useful generalization of the LCS problem.

In addition to standard integer indices $\ldots, -2, -1, 0, 1, 2, \ldots$, we use *odd half-integer* indices $\ldots, -\frac{5}{2}, -\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \ldots$. We denote

$$[i:j] = \{i, i+1, \ldots, j-1, j\}, \quad \langle i:j\rangle = \left\{i+\tfrac{1}{2}, i+\tfrac{3}{2}, \ldots, j-\tfrac{3}{2}, j-\tfrac{1}{2}\right\}.$$

To denote infinite intervals of integers and odd half-integers, we will use $-\infty$ for i and $+\infty$ for j where appropriate. For both interval types $[i:j]$ and $\langle i:j\rangle$, we call the difference $j-i$ the *length* of the interval.

We will make extensive use of finite and infinite matrices, with integer elements and integer or odd half-integer indices. A *permutation matrix* is a (0,1)-matrix containing exactly one nonzero in every row and every column. An *identity matrix* is a permutation matrix I such that $I(i,j) = 1$ if $i = j$, and $I(i,j) = 0$ otherwise. Each of these definitions applies to both finite and infinite matrices.

From now on, instead of "index pairs corresponding to nonzeros," we will write simply "nonzeros," where this does not lead to confusion. A finite permutation matrix can be represented by its nonzeros. When we deal with an infinite matrix, it will typically have a finite nontrivial core, and will be trivial (e.g., equal to the infinite identity matrix) outside of this core. An infinite permutation matrix with finite nontrivial core can be represented by its core nonzeros.

Let D be an arbitrary numerical matrix with indices ranging over $\langle 0 : n\rangle$. Its *distribution matrix*, with indices ranging over $[0 : n]$, is defined by

$$d(i_0, j_0) = \sum D(i,j), \quad i \in \langle i_0 : n\rangle,\ j \in \langle 0 : j_0\rangle,$$

for all $i_0, j_0 \in [0 : n]$. We have

$$D(i,j) = d\Big(i - \tfrac{1}{2}, j + \tfrac{1}{2}\Big) - d\Big(i - \tfrac{1}{2}, j - \tfrac{1}{2}\Big) - d\Big(i + \tfrac{1}{2}, j + \tfrac{1}{2}\Big) + d\Big(i + \tfrac{1}{2}, j - \tfrac{1}{2}\Big).$$

When a matrix d is the distribution matrix of D, the matrix D is called the *density matrix* of d. The definitions of distribution and density matrices extend naturally to infinite matrices. We will only deal with distribution matrices where all elements are defined and finite.

We will use the term *permutation-distribution matrix* as an abbreviation of "the distribution matrix of a permutation matrix."

4. Algorithmic techniques

The rest of this paper is based on the framework for the all semi-local LCS problem developed in [17, 18]. For completeness, we recall most background definitions and results from [17], omitting the proofs.

4.1. Dominance counting. It is well known that an instance of the LCS problem can be represented by a dag (directed acyclic graph) on an $m \times n$ grid of nodes, where character matches correspond to edges scoring 1, and mismatches to edges scoring 0.

Definition 5. *Let $m, n \in \mathbb{N}$. An* alignment dag *G is a weighted dag, defined on the set of nodes $v_{l,i}$, $l \in [0 : m]$, $i \in [0 : n]$. The edge and path weights are called* scores. *For all $l \in [1 : m]$, $i \in [1 : n]$,*

- *the horizontal edge $v_{l,i-1} \to v_{l,i}$ and the vertical edge $v_{l-1,i} \to v_{l,i}$ are both always present in G and have score 0;*
- *the diagonal edge $v_{l-1,i-1} \to v_{l,i}$ may or may not be present in G; if present, it has score 1.*

Given an instance of the all semi-local LCS problem, its corresponding alignment dag *is the $m \times n$ alignment dag where the diagonal edge $v_{l-1,i-1} \to v_{l,i}$ is present if and only if $\alpha_i = \beta_j$.*

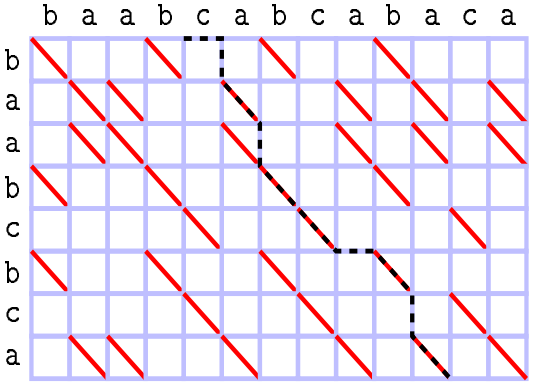

Fig. 1. An alignment dag and a highest-scoring path.

Figure 1 shows the alignment dag corresponding to the strings a = "baabcbca", b = "baabcabcabaca" (an example borrowed from [2]).

Common string-substring, suffix-prefix, prefix-suffix, and substring-string subsequences correspond, respectively, to paths of the following form in the alignment dag:

$$v_{0,i} \rightsquigarrow v_{m,i'}, \quad v_{l,0} \rightsquigarrow v_{m,i'}, \quad v_{0,i} \rightsquigarrow v_{l',n}, \quad v_{l,0} \rightsquigarrow v_{l',n}, \tag{1}$$

where $l, l' \in [0:m]$, $i, i' \in [0:n]$. The length of each subsequence is equal to the score of its corresponding path.

The solution to the all semi-local LCS problem is equivalent to finding the score of a highest-scoring path of each of the four types (1) between every possible pair of endpoints.

To describe our algorithms, we need to modify the definition of an alignment dag by embedding the finite grid of nodes into an infinite grid.

Definition 6. *Given an $m \times n$ alignment dag G, its* extension G^+ *is an infinite weighted dag, defined on the set of nodes $v_{l,i}$, $l, i \in [-\infty:+\infty]$, and containing G as a subgraph. For all $l, i \in [-\infty:+\infty]$,*

- *the horizontal edge $v_{l,i-1} \rightarrow v_{l,i}$ and the vertical edge $v_{l-1,i} \rightarrow v_{l,i}$ are both always present in G^+ and have score 0;*
- *when $l \in [1:m]$, $i \in [1:n]$, the diagonal edge $v_{l-1,i-1} \rightarrow v_{l,i}$ is present in G^+ if and only if it is present in G; if present, it has score 1;*
- *otherwise, the diagonal edge $v_{l-1,i-1} \rightarrow v_{l,i}$ is always present in G^+ and has score 1.*

An infinite dag that is an extension of some (finite) alignment dag will be called an extended alignment dag. *When a dag G^+ is the extension of a dag G, we will say that G is the* core *of G^+. Relative to G^+, we will call the nodes of G* core nodes.

By using the extended alignment dag representation, the four path types (1) can be reduced to a single type, corresponding to the all string-substring (or, symmetrically, substring-string) LCS problem on an extended set of indices.

Definition 7. *Given an $m \times n$ alignment dag G, its* extended highest-score matrix *is the infinite matrix defined by*

$$A(i,j) = \text{max score}\,(v_{0,i} \rightsquigarrow v_{m,j}), \quad i, j \in [-\infty:+\infty], \tag{2}$$

where the maximum is taken across all paths between the given endpoints in the extension G^+. If $i = j$, we have $A(i,j) = 0$. By convention, if $j < i$, then we let $A(i,j) = j - i < 0$.

In Fig. 1, the highlighted path has score 5, and corresponds to the value $A(4,11) = 5$, which is equal to the LCS score of the string a and the substring b' = "cabcaba".

In this paper, we will deal almost exclusively with extended (i.e., finitely represented, but conceptually infinite) alignment dags and highest-score matrices. From now on, we omit the term "extended" for brevity, always assuming it by default.

The maximum path scores for each of the four path types (1) can be obtained from the highest-score matrix (2) as follows:

$$\begin{aligned}
&\text{max score}\,(v_{0,j} \rightsquigarrow v_{m,j'}) = A(j,j'),\\
&\text{max score}\,(v_{i,0} \rightsquigarrow v_{m,j'}) = A(-i,j') - i,\\
&\text{max score}\,(v_{0,j} \rightsquigarrow v_{i',n}) = A(j,m+n-i') - m + i',\\
&\text{max score}\,(v_{i,0} \rightsquigarrow v_{i',n}) = A(-i,m+n-i') - m - i + i',
\end{aligned}$$

where $i, i' \in [0:m]$, $j, j' \in [0:n]$, and the maximum is taken across all paths between the given endpoints.

Definition 8. *An odd half-integer point $(i,j) \in \langle -\infty:+\infty\rangle^2$ is called* A-critical *if*

$$A\Big(i+\tfrac{1}{2}, j-\tfrac{1}{2}\Big) + 1 = A\Big(i-\tfrac{1}{2}, j-\tfrac{1}{2}\Big) = A\Big(i+\tfrac{1}{2}, j+\tfrac{1}{2}\Big) = A\Big(i-\tfrac{1}{2}, j+\tfrac{1}{2}\Big).$$

In particular, the point (i,j) is never A-critical for $i > j$. When $i = j$, the point (i,j) is A-critical if and only if $A\Big(i-\frac{1}{2}, j+\frac{1}{2}\Big) = 0$.

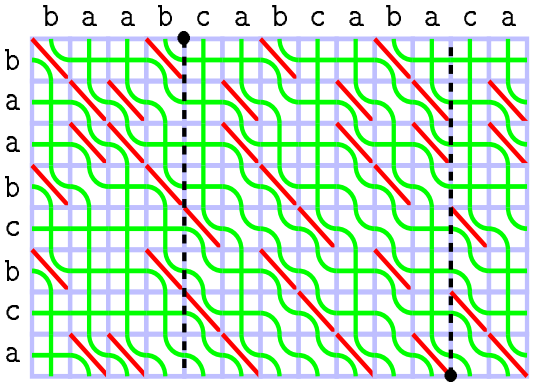

FIG. 2. An alignment dag and the seaweeds.

Corollary 1. *Let $i, j \in \langle -\infty : +\infty \rangle$. For each i (respectively, j), there exists exactly one j (respectively, i) such that the point (i, j) is A-critical.*

Figure 2 shows the alignment dag of Figure 1 along with the critical points. In particular, every critical point (i, j) with $i, j \in \langle 0 : n \rangle$ is represented by a *seaweed*[1], originating between the nodes $v_{0,i-\frac{1}{2}}$ and $v_{0,i+\frac{1}{2}}$ and terminating between the nodes $v_{m,j-\frac{1}{2}}$ and $v_{m,j+\frac{1}{2}}$. The remaining seaweeds, originating or terminating at the sides of the dag, correspond to critical points (i, j) with either $i \in \langle -m : 0 \rangle$ or $j \in \langle n : n + m \rangle$ (or both). In particular, every critical point (i, j) with $i \in \langle -m : 0 \rangle$ (respectively, $j \in \langle n : m + n \rangle$) is represented by a seaweed originating between the nodes $v_{-i-\frac{1}{2},0}$ and $v_{-i+\frac{1}{2},0}$ (respectively, terminating between the nodes $v_{m+n-j-\frac{1}{2},n}$ and $v_{m+n-j+\frac{1}{2},n}$).

It is convenient to consider the set of A-critical points as an infinite permutation matrix. For all $i, j \in \langle -\infty : +\infty \rangle$, we define

$$D_A(i,j) = \begin{cases} 1 & \text{if } (i,j) \text{ is } A\text{-critical,} \\ 0 & \text{otherwise.} \end{cases}$$

We denote the infinite distribution matrix of D_A by d_A, and consider the following simple geometric relation.

Definition 9. *A point (i_0, j_0)* dominates[2] *a point (i, j) if $i_0 < i$ and $j < j_0$.*

Informally, the dominated point is "below and to the left" of the dominating point in the highest-score matrix[3]. Clearly, for an arbitrary integer point $(i_0, j_0) \in [-\infty : +\infty]^2$, the value $d_A(i_0, j_0)$ is the number of (odd half-integer) A-critical points it dominates.

The following theorem shows that the set of critical points uniquely defines a highest-score matrix, and gives a simple formula for recovering the matrix elements.

Theorem 1 ([17])**.** *For all $i_0, j_0 \in [-\infty : +\infty]$, we have*

$$A(i_0, j_0) = j_0 - i_0 - d_A(i_0, j_0).$$

In Fig. 2, critical points dominated by the point $(4, 11)$ are represented by seaweeds whose both endpoints (and therefore the whole seaweed) fit between the two vertical lines corresponding to the index values $i = 4$ and $j = 11$. Note that there are exactly two such seaweeds, and that $A(4, 11) = 11 - 4 - 2 = 5$.

By Theorem 1, a highest-score matrix A is represented uniquely by an infinite permutation matrix D_A with odd half-integer row and column indices. We will call the matrix D_A the *implicit representation* of A. From now on, we will refer to the critical points of A as nonzeros (i.e., ones) in its implicit representation.

Recall that outside the core, the structure of an alignment graph is trivial: all possible diagonal edges are present in the off-core subgraph. This property carries over to the corresponding permutation matrix.

[1]This imaginative term was suggested by Yu. V. Matiyasevich.

[2]The standard definition of dominance requires $i < i_0$ instead of $i_0 < i$. Our definition is more convenient in the context of the LCS problem.

[3]Note that these concepts of "below" and "left" are relative to the highest-score matrix, and have no connection to the "vertical" and "horizontal" directions in the alignment dag.

Definition 10. *Given an infinite permutation matrix D, a* core *of D is a square (possibly semi-infinite) submatrix defined by an index range $[i_0 : j_0] \times [i_1 : j_1]$ with $j_0 - i_0 = j_1 - i_1$ (as long as both these values are defined) such that for all off-core elements $D(i, j)$, we have $D(i, j) = 1$ if and only if $j - i = j_0 - i_0$ and $j - i = j_1 - i_1$ (in each case, as long as the right-hand side is defined).*

Informally, the off-core part of a matrix D has nonzeros on the off-core extension of the main diagonal of the core.

The following statements are an immediate consequence of the definitions.

Corollary 2. *A core of an infinite permutation matrix is a (possibly semi-infinite) permutation matrix.*

Corollary 3. *Given an alignment dag A as described above, the corresponding permutation matrix D_A has a core of size $m + n$, defined by $i \in \langle -m, n \rangle$, $j \in \langle 0, m + n \rangle$.*

In Fig. 2, the set of critical points represented by the seaweeds corresponds precisely to the set of all core nonzeros in D_A. Note that there are $m + n = 8 + 13 = 21$ seaweeds in total.

Since only core nonzeros need to be represented explicitly, the implicit representation of a highest-score matrix can be stored as a permutation of size $m + n$. From now on, we will assume this as the default representation of such matrices.

By Theorem 1, the value $A(i_0, j_0)$ is determined by the number of nonzeros in D_A dominated by (i_0, j_0). Therefore, an individual element of A can be obtained explicitly by scanning the implicit representation of A in time $O(m + n)$, counting the dominated nonzeros. However, existing methods of computational geometry allow us to perform this *dominance counting* procedure much more efficiently, as long as preprocessing of the implicit representation is allowed.

Theorem 2 ([17])**.** *Given the implicit representation D_A of a highest-score matrix A, there exists a data structure that*

- *has size $O\Big((m + n) \log(m + n)\Big)$;*
- *can be built in time $O\Big((m + n) \log(m + n)\Big)$;*
- *allows to query an individual element of A in time $O\Big(\log^2(m + n)\Big)$.*

4.2. Highest-score matrix multiplication. A common pattern in many problems on strings is partitioning the alignment dag into alignment subdags. Without loss of generality, consider a partitioning of an $(M + m) \times n$ alignment dag G into an $M \times n$ alignment dag G_1 and an $m \times n$ alignment dag G_2, where $M \geq m$. The dags G_1, G_2 share a horizontal row of n nodes, which is simultaneously the bottom row of G_1 and the top row of G_2; the dags also share the corresponding $n - 1$ horizontal edges. We will say that the dag G is the *concatenation* of the dags G_1 and G_2. Let A, B, C denote the highest-score matrices defined, respectively, by the dags G_1, G_2, G. Our goal is, given the matrices A, B, to compute the matrix C efficiently. We call this procedure the *highest-score matrix multiplication.*

Definition 11. *Let $n \in \mathbb{N}$. Let A, B, C be arbitrary numerical matrices with indices ranging over $[0 : n]$. The* $(\min, +)$-product *$A \odot B = C$ is defined by*

$$C(i, k) = \min_j \Big(A(i, j) + B(j, k) \Big), \quad i, j, k \in [0 : n].$$

Lemma 1 ([17])**.** *Let D_A, D_B, D_C be permutation matrices with indices ranging over $\langle 0 : n \rangle$, and let d_A, d_B, d_C be their respective distribution matrices. Let $d_A \odot d_B = d_C$. Given the nonzeros of D_A, D_B, the nonzeros of D_C can be computed in time $O\Big(n^{1.5}\Big)$ and memory $O(n)$.*

Lemma 2 ([17])**.** *Let D_A, D_B, D_C be permutation matrices with indices ranging over $\langle -\infty : +\infty \rangle$. Let D_A (respectively, D_B) have semi-infinite core $\langle 0 : +\infty \rangle^2$ (respectively, $\langle -\infty : n \rangle^2$). Let d_A, d_B, d_C be the respective distribution matrices, and assume $d_A \odot d_B = d_C$. We have*

$$D_A(i, j) = D_C(i, j) \qquad \text{for} \quad i \in \langle -\infty : +\infty \rangle, \quad j \in \langle n : +\infty \rangle, \tag{3}$$

$$D_B(j, k) = D_C(j, k) \qquad \text{for} \quad j \in \langle -\infty : 0 \rangle, \quad k \in \langle -\infty : +\infty \rangle. \tag{4}$$

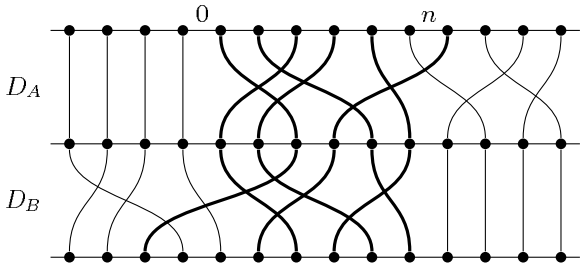

FIG. 3. An illustration of Lemma 2.

Equations (3)–(4) *cover all but* n *nonzeros in each of* D_A, D_B, D_C. *These remaining nonzeros have* $i \in \langle 0 : +\infty \rangle$, $j \in \langle 0 : n \rangle$, $k \in \langle -\infty : n \rangle$. *Given the* n *remaining nonzeros in each of* D_A, D_B, *the* n *remaining nonzeros in* D_C *can be computed in time* $O\left(n^{1.5}\right)$ *and memory* $O(n)$.

The above lemma is illustrated by Fig. 3. Three horizontal lines represent the index ranges of i, j, k, respectively. The nonzeros in D_A (respectively, D_B) are shown by top-to-middle (respectively, middle-to-bottom) seaweeds; thin seaweeds correspond to the nonzeros covered by (3)–(4), and thick seaweeds to the remaining nonzeros. By Lemma 2, the nonzeros in D_C covered by (3)–(4) are represented by thin top-to-bottom seaweeds. The remaining nonzeros in D_C are not represented explicitly, but can be obtained from the thick top-to-middle and middle-to bottom seaweeds by Lemma 1.

4.3. Partial highest-score matrix multiplication. In certain contexts, e.g., when $m \gg n$, we may not be able to solve the all semi-local LCS problem, or even to store its implicit highest-score matrix. In such cases, we may wish to settle for the following asymmetric version of the problem.

Definition 12. *The* partial semi-local LCS problem *consists in computing the LCS scores on substrings of* a *and* b *as follows:*

- *the* all string-substring LCS problem*: a against every substring of* b*;*
- *the* all prefix-suffix LCS problem*: every prefix of* a *against every suffix of* b*;*
- *the* all suffix-prefix LCS problem*: every suffix of* a *against every prefix of* b.

In contrast with the all semi-local LCS problem, the comparison of substrings of a *against* b *is not required.*

Let A be the highest-score matrix for the all semi-local LCS problem. Given the implicit representation of A, the corresponding *partial implicit representation* consists of all nonzeros $A(i,j)$ with either $i \in \langle 0 : n \rangle$ or $j \in \langle 0 : n \rangle$ (equivalently, $(i,j) \in \langle 0 : n \rangle \times \langle 0 : +\infty \rangle \cup \langle -\infty : n \rangle \times \langle 0 : n \rangle$). All such nonzeros are core; the number of these nonzeros is at least n and at most $2n$ (note that the size of the partial implicit representation is therefore independent of m). The minimum (respectively, maximum) number of nonzeros is attained when all (respectively, none of) these nonzeros are contained in the submatrix defined by $(i,j) \in \langle 0 : n \rangle \times \langle 0 : n \rangle$.

Theorem 3. *Given the partial implicit representation of a highest-score matrix* A*, there exists a data structure that*

- *has size* $O(n \log n)$*;*
- *can be built in time* $O(n \log n)$*;*
- *allows to query an individual element of* A*, corresponding to an output of the partial semi-local LCS problem, in time* $O(\log^2 n)$.

Proof. Similarly to the proof of Theorem 2, the structure in question is a 2D range tree built on the set of nonzeros in the partial implicit representation of A. □

The following lemma gives an equivalent of highest-score matrix multiplication for partially represented matrices.

Lemma 3. *Consider the concatenation of alignment dags as described in Sec.* 4.2*, with highest-score matrices* A, B, C. *Given the partial implicit representations of* A, B, *the partial implicit representation of* C *can be computed in time* $O\left(n^{1.5}\right)$ *and memory* $O(n)$.

Proof. Let $D'_A(i,j) = D_A(i-M,j)$, $D'_B(j,k) = D_B(j,k+m)$, $D'_C(i,k) = D_B(i-M,k+m)$ for all i,j,k, and define d'_A, d'_B, d'_C accordingly. It is easy to check that $d'_A \odot d'_B = d'_C$ if and only if $d_A \odot d_B = d_C$. The matrices D'_A, D'_B, D'_C satisfy the conditions of Lemma 2, therefore all but n of the core nonzeros in the required partial implicit representation can be obtained by (3)–(4) in time and memory $O(n)$, and the remaining n core nonzeros can be obtained in time $O(n^{1.5})$ and memory $O(n)$. □

5. The algorithms

5.1. Clobal subsequence recognition and LCS. We now return to the problem of subsequence recognition introduced in Sec. 2. A simple efficient algorithm for global subsequence recognition in an SLP-compressed string is not difficult to obtain, and has been known in folklore[4]. For convenience, we generalize the output of the problem: instead of a Boolean value, the algorithm will return an integer.

Algorithm 1 (Global subsequence recognition).

Input: string T of length m, represented by an SLP of length $\overline{m}$; string P of length n, represented explicitly.

Output: the integer k giving the length of the longest prefix of P that is a subsequence of T. The string T contains P as a subsequence if and only if $k = n$.

Description. The computation is performed recursively as follows.

Let $T = T'T''$ be the SLP statement defining the string T. Let k' be the length of the longest prefix of P that is a subsequence of T'. Let k'' be the length of the longest prefix of $P \downarrow k'$ that is a subsequence of T''. Both k' and k'' can be found recursively. We have $k = k' + k''$.

The base of the recursion is $\overline{m} = m = 1$. In this case, the value $k \in \{0,1\}$ is determined by a single character comparison.

Cost analysis. The running time of the algorithm is $O(\overline{m}k)$. The proof is by induction. The running times of the recursive calls are, respectively, $O(\overline{m}k')$ and $O(\overline{m}k'')$. The overall running time of the algorithm is $O(\overline{m}k') + O(\overline{m}k'') + O(1) = O(\overline{m}k)$. In the worst case, this is $O(\overline{m}n)$. □

We now address the more general partial semi-local LCS problem. Our approach is based on the technique introduced in Sec. 4.3.

Algorithm 2 (Partial semi-local LCS).

Input: string T of length m, represented by an SLP of length $\overline{m}$; string P of length n, represented explicitly.

Output: the partial implicit highest-score matrix on the strings T, P

Description. The computation is performed recursively as follows.

Let $T = T'T''$ be the SLP statement defining the string T. Given the partial implicit highest-score matrices for each of T' and T'' against P, the partial implicit highest-score matrix of T against P can be computed by Lemma 3.

The base of the recursion is $\overline{m} = m = 1$. In this case, the matrix coincides with the full implicit highest-score matrix, and can be computed by a simple scan of the string P.

Cost analysis. By Lemma 3, each implicit matrix multiplication runs in time $O(n^{1.5})$ and memory $O(n)$. There are $\overline{m}$ recursive steps in total, therefore all the matrix multiplications combined run in time $O(\overline{m}n^{1.5})$ and memory $O(n)$. □

Note that the above algorithm, as a special case, provides an efficient solution for the LCS problem: the LCS score for T against P can easily be queried from the output of the algorithm by Theorem 2.

The running time of Algorithm 2 should be contrasted with the standard uncompressed LCS algorithms, running in time $O\left(\frac{mn}{\log(m+n)}\right)$ (see [12, 6]), and with the NP-hardness of the LCS problem on two compressed strings (see [11]).

5.2. Local subsequence recognition. We now show how the partial semi-local LCS algorithm of the previous section can be used to provide local subsequence recognition.

Algorithm 3 (Minimal-window subsequence recognition).

Input: string T of length m, represented by an SLP of length $\overline{m}$; string P of length n, represented explicitly.

Output: the number of windows in T containing P minimally as a subsequence.

[4]The author is grateful to Y. Lifshits for pointing this out.

Description. The algorithm runs in two phases.

First phase. Using Algorithm 2, we compute the partial implicit highest-score matrix for every SLP symbol against P. For each of these matrices, we then build the data structure of Theorem 3.

Second phase. For brevity, we will call a window containing P minimally as a subsequence a *P-episode window*. The number of P-episode windows in T is computed recursively as follows.

Let $T = T'T''$ be the SLP statement defining the string T. Let m', m'' be the (uncompressed) lengths of the strings T', T''. Let r' (respectively, r'') be the number of P-episode windows in T' (respectively, T''), computed by recursion.

We now need to consider the $n-1$ possible prefix-suffix decompositions $P = (P \upharpoonright n')(P \upharpoonleft n'')$, for all $n', n'' > 0$ such that $n' + n'' = n$. Let l' (respectively, l'') be the length of the shortest suffix of T' (respectively, prefix of T'') containing $P \upharpoonright n'$ (respectively, $P \upharpoonleft n''$) as a subsequence. The value of l' (respectively, l'') can be found, or its nonexistence established, by binary search on the first (respectively, second) index component of nonzeros in the partial implicit highest-score matrix of T' (respectively, T'') against P. At every step of the binary search, we make a suffix-prefix (respectively, prefix-suffix) LCS score query by Theorem 3. We call the interval $[m' - l' : m' + l'']$ a *candidate window*.

It is easy to see that if a window in T is P-episode, then it is either contained within one of T', T'', or is a candidate window. Conversely, a candidate window $[i, j]$ is P-episode unless there is a smaller candidate window $[i_1, j_1]$ with either $i = i_1 < j_1 < j$, or $i < i_1 < j_1 = j$. Given the set of all candidate windows sorted separately by the lower endpoints and the higher endpoints, this test can be performed in overall time $O(n)$. Let s be the resulting number of distinct P-episode candidate windows. The overall number of P-episode windows in T is equal to $r' + r'' + s$.

The base of the recursion is $m < n$. In this case, no windows of length n or more exist in T, so none can be P-episode.

Cost analysis.

First phase. As in Algorithm 2, the main data structure can be built in time $O(\overline{m}n^{1.5})$. The additional data structure of Theorem 2 can be built in time $\overline{m} \cdot O(n \log n) = O(\overline{m}n \log n)$.

Second phase. For each of $n-1$ decompositions $n' + n'' = n$, the binary search performs at most $\log n$ suffix-prefix and prefix-suffix LCS queries, each taking time $O(\log^2 n)$. Therefore, each recursive step runs in time $2n \cdot \log n \cdot O(\log^2 n) = O(n \log^3 n)$. There are $\overline{m}$ recursive steps in total, therefore the whole recursion runs in time $O(\overline{m}n \log^3 n)$. It is possible to speed up this phase by reusing data between different instances of binary search and LCS query; however, this is not necessary for the overall efficiency of the algorithm.

The overall computation cost is dominated by the cost of building the main data structure in the first phase, equal to $O(\overline{m}n^{1.5})$. □

Algorithm 4 (Fixed-window subsequence recognition).

Input: string T of length m, represented by an SLP of length $\overline{m}$; string P of length n, represented explicitly; window length w.

Output: the number of windows of length w in T containing P as a subsequence.

Description.

First phase. As in Algorithm 3.

Second phase. For brevity, we will call a window of length w containing P as a subsequence a *(P, w)-episode window*. The number of (P, w)-episode windows in T is computed recursively as follows.

Let $T = T'T''$ be the SLP statement defining the string T. Let m', m'' be the (uncompressed) lengths of the strings T', T''. Let r' (respectively, r'') be the number of (P, w)-episode windows in T' (respectively, T''), computed by recursion.

We now need to consider the $w-1$ windows that span the boundary between T' and T'', corresponding to the strings $(T' \upharpoonright w')(T'' \upharpoonleft w'')$, for all $w', w'' > 0$ such that $w' + w'' = w$. We call an interval $[m' - w' : m' + w'']$ a *candidate window*. In contrast with the minimal-window problem, we can no longer afford to consider every candidate window individually, and will therefore need to count them in groups of "equivalent" windows.

Let (i, j) (respectively, (j, k)) be a nonzero in the partial highest-score matrix of T' (respectively, T'') against P. We will say that such a nonzero is *covered* by a candidate window $[m' - w' : m' + w'']$ if $i \in \langle -m' : -m' + w' \rangle$ (respectively, $k \in \langle m'' + n - w : m'' + n \rangle$). We will say that two candidate windows are *equivalent* if they cover the same set of nonzeros both for T' and T''.

Since the number of nonzeros for each of T', T'' is at most n, the defined equivalence relation has at most $2n$ equivalence classes. Each equivalence class corresponds to a contiguous segment of values w' (and, symmetrically, w''), and is completely described by the two endpoints of this segment. Given the set of all the nonzeros, the endpoint description of all the equivalence classes can be computed in time $O(n)$.

For each equivalence class of candidate windows, either none or all of them are (P, w)-episode; in the latter case, we will call the whole equivalence class (P, w)-*episode*. We consider each equivalence class in turn, and pick from it an arbitrary representative candidate window $[m' - w' : m' + w'']$. Let l' (respectively, l'') be the length of the longest prefix (respectively, suffix) of P contained in $T' \upharpoonright w'$ (respectively, $T'' \upharpoonright w''$) as a subsequence. The value of l' (respectively, l'') can be found by binary search on the second (respectively, first) index component of nonzeros in the partial implicit highest-score matrix of T' (respectively, T'') against P. At every step of the binary search, we make a suffix-prefix (respectively, prefix-suffix) LCS score query by Theorem 3.

It is easy to see that the current equivalence class is (P, w)-episode if and only if $l' + l'' \geq n$. Let s be the total size of (P, w)-episode equivalence classes. The overall number of (P, w)-episode windows in T is equal to $r' + r'' + s$.

The base of the recursion is $m < w$. In this case, no windows of length w or more exist in T, so none can be (P, w)-episode.

Cost analysis. As in Algorithm 3, the total cost is dominated by the cost of the first phase, equal to $O(\overline{m}n^{1.5})$. □

The bounded minimal-window subsequence recognition problem can be solved by a simple modification of Algorithm 3, discarding all candidate windows of length greater than w. Furthermore, in addition to counting the windows, Algorithms 3 and 4 can both be easily modified to report all the respective windows at the additional cost of $O(output)$.

6. Conclusions

We have considered several subsequence recognition problems for an SLP-compressed text against an uncompressed pattern. First, we mentioned a simple folklore algorithm for the global subsequence recognition problem, running in time $O(\overline{m}n)$. Relying on the previously developed framework of semi-local string comparison, we then gave an algorithm for the partial semi-local LCS problem, running in time $O(\overline{m}n^{1.5})$; this includes the LCS problem as a special case. A natural question is whether the running time of partial semi-local LCS (or just LCS) can be improved to match global subsequence recognition.

We have also given algorithms for the local subsequence recognition problem in its minimal-window and fixed-window versions. Both algorithms run in time $O(\overline{m}n^{1.5})$, and can be easily modified to report all the respective windows at the additional cost of $O(output)$. Again, a natural question is whether this running time can be further improved.

Another classical generalization of both the LCS problem and local subsequence recognition is *approximate matching* (see, e.g., [14]). Here, we look for substrings in the text that are close to the pattern in terms of the *edit distance*, with possibly different costs charged for insertions/deletions and substitutions. Once again, we can formulate it as a counting problem (the *k-approximate matching problem*): counting the number of windows in T that have edit distance at most k from P. This problem is considered on LZ-compressed strings (essentially, a special case of SLP-compression) in the paper [10], which gives an algorithm running in time $O(\overline{m}nk)$. It would be interesting to see if this algorithm can be improved by using the ideas of the current paper.

REFERENCES

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1976).
2. C. E. R. Alves, E. N. Cáceres, and S. W. Song, "An all-substrings common subsequence algorithm," *Electr. Notes Discr. Math.*, **19**, 133–139 (2005).
3. J. L. Bentley, "Multidimensional divide-and-conquer," *Comm. ACM*, **23**, No. 4, 214–229 (1980).
4. P. Cégielski, I. Guessarian, Y. Lifshits, and Y. Matiyasevich, "Window subsequence problems for compressed texts," in: *Proceedings of CSR, Lect. Notes Comp. Sci.*, **3967** (2006), pp. 127–136.
5. P. Cégielski, I. Guessarian, and Y. Matiyasevich, "Multiple serial episodes matching," *Inform. Process. Lett.*, **98**, No. 6, 211–218 (2006).
6. M. Crochemore, G. M. Landau, and M. Ziv-Ukelson, "A subquadratic sequence alignment algorithm for unrestricted score matrices," *SIAM J. Comp.*, **32**, No. 6, 1654–1673 (2003).

7. M. Crochemore and W. Rytter, *Text Algorithms*, Oxford University Press (1994).
8. G. Das, R. Fleischer, L. Gasieniec, D. Gunopulos, and J. Kärkkäinen, "Episode matching," in: *Proceedings of CPM, Lect. Notes Comp. Sci.*, **1264** (1997), pp. 12–27.
9. D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press (1997).
10. J. Kärkkäinen, G. Navarro, and E. Ukkonen, "Approximate string matching on Ziv–Lempel compressed text," *J. Discr. Alg.*, **1**, 313–338 (2003).
11. Y. Lifshits and M. Lohrey, "Querying and embedding compressed texts," in: *Proceedings of MFCS, Lect. Notes Comp. Sci.*, **4162** (2006), pp. 681–692.
12. W. J. Masek and M. S. Paterson, "A faster algorithm computing string edit distances," *J. Comp. System Sci.*, **20**, 18–31 (1980).
13. G. Myers, "Approximately matching context-free languages," *Inform. Process. Lett.*, **54**, 85–92 (1995).
14. G. Navarro, "A guided tour to approximate string matching," *ACM Comp. Surv.*, **33**, No. 1, 31–88 (2001).
15. F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer (1985).
16. W. Rytter, "Application of Lempel–Ziv factorization to the approximation of grammar-based compression," *Theor. Comp. Sci.*, **302**, No. 1–3, 211–222 (2003).
17. A. Tiskin, "Semi-local longest common subsequences in subquadratic time," *J. Discr. Alg.*, **6**, No. 4, 570–581 (2008).
18. A. Tiskin, "Semi-local string comparison: Algorithmic techniques and applications," *Math. Comput. Sci.*, **1**, No. 4, 571–603 (2008).
19. B. W. Watson and G. Zwaan, "A taxonomy of sublinear multiple keyword pattern matching algorithms," *Sci. Comput. Programm.*, **27**, No. 2, 85–118 (1996).
20. T. A. Welch, "A technique for high-performance data compression," *Computer*, **17**, No. 6, 8–19 (1984).
21. G. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transact. Inform. Theory*, **23**, 337–343 (1977).
22. G. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Transact. Inform. Theory*, **24**, 530–536 (1978).