Math.comput.sci. 1 (2008), 571–603

1661-8270/040571-33, *published online* May 23, 2008
DOI 10.1007/s11786-007-0033-3

Mathematics in Computer Science

Semi-local String Comparison: Algorithmic Techniques and Applications

Alexander Tiskin

Abstract. Given two strings, the longest common subsequence (LCS) problem consists in computing the length of the longest string that is a subsequence of both input strings. Its generalisation, the all semi-local LCS problem, requires computing the LCS length for each string against all substrings of the other string, and for all prefixes of each string against all suffixes of the other string. We survey a number of algorithmic techniques related to the all semi-local LCS problem. We then present a number of algorithmic applications of these techniques, both existing and new. In particular, we obtain a new all semi-local LCS algorithm, with asymptotic running time matching (in the case of an unbounded alphabet) the fastest known global LCS algorithm by Masek and Paterson. We conclude that semi-local string comparison turns out to be a useful algorithmic plug-in, which unifies, and often improves on, a number of previous approaches to various substring- and subsequence-related problems.

Mathematics Subject Classification (2000). Primary 68W05; Secondary 68R15.

Keywords. String algorithms; longest common subsequence; semi-local string comparison.

1. Introduction

Given two strings a, b of lengths m, n respectively, the (global) longest common subsequence (LCS) problem consists in computing the length of the longest string that is a subsequence of both a and b. In [48], we defined this problem's generalisation, the all semi-local LCS problem, where the LCS is computed for each string against all substrings of the other string, and for all prefixes of each string against all suffixes of the other string. The all semi-local LCS problem arises naturally in the context of LCS computations on substrings. We define the problem and give other preliminaries in Section 2.

The author acknowledges the support of The University of Warwick's DIMAP (the Centre for Discrete Mathematics and its Applications) during this work.

In previous papers, we developed an approach for the solution of the all semi-local LCS problem, based on an intimate geometric relationship between its output matrix (that we call a "highest-score matrix") and a certain kind of infinite permutation matrices. Exploiting this relationship, we proposed several algorithmic techniques involving fast multiplication of highest-score matrices in the $(\min, +)$-semiring. We review these techniques in Section 3.

In Section 4, we present a number of algorithmic applications of our techniques, both existing and new. These applications fall into two broad subcategories, based on the overall algorithm structure: either iterative, or divide-and-conquer.

Classical algorithms for the global LCS problem are based on the iterative approach. In particular, the dynamic programming algorithm by Wagner and Fischer [50] runs in time $O(mn)$. The algorithm by Masek and Paterson [35] runs in time $O(\frac{mn \log\log n}{\log n})$ for $m \leq n$ and reasonably close m and n, assuming the RAM model with word size at least $O(\log\log n)$; this running time can be improved to $O(\frac{mn}{\log n})$ if the alphabet size is bounded by a constant.

The iterative approach to the all semi-local LCS problem is covered in this paper by Subsections 4.1–4.5. In particular, in Subsection 4.1 we give an improved (faster and simpler) all semi-local LCS algorithm. In contrast with the algorithm of [48], the new algorithm is independent of the highest-score matrix multiplication method. Instead, it uses small-block precomputation directly, and runs in time $O(\frac{mn \log\log n}{\log n})$ for $m \leq n$ and reasonably close m and n, under the same assumptions on the RAM model, both for bounded and unbounded alphabet size. Thus, in the case of an unbounded alphabet, the running time of our algorithm matches the running time of the global LCS algorithm by Masek and Paterson, while in the case of a bounded (constant-sized) alphabet, the running time of our algorithm remains unchanged, and the algorithm by Masek and Paterson runs faster by a factor of $O(\log\log n)$.

By direct application of our algorithm, in Subsection 4.2 we obtain an improved algorithm for the cyclic LCS problem, and in Subsection 4.3 for the longest repeated subsequence problem.

Extending the paradigms of incremental [27, 30] and fully-incremental [23] string comparison, in Subsection 4.4 we define block-incremental string comparison, where each of the two input strings can be extended on-line by either appending or prepending a block of characters of variable length l. We give an algorithm that, following some off-line precomputation, updates the LCS data structure on-line in time $O(ml^{0.5})$ (respectively, $O(nl^{0.5})$) when input string a (respectively, b) is kept fixed.

In Subsection 4.5, we consider the paradigm of common-substring comparison [14, 31], where a text string of length n is compared against an unspecified number of pattern strings, that may share a common substring of length $l \leq n$. The goal is, given the text, to preprocess the common substring so as to minimise the LCS computation time for each occurrence of the common substring in the patterns. Papers [14, 31] give an algorithm for the global version of the problem.

This algorithm, following some preprocessing in time $O(nl)$, takes time $O(n)$ for each occurrence of the common substring in a pattern. We extend this algorithm to the more general semi-local version of the problem. Our algorithm, after the same asymptotic amount of preprocessing, takes time $O(n)$ for the first occurrence of the common substring in a pattern, and time $O(nl^{0.5})$ for each subsequent occurrence in the same pattern. In particular, if the common substring only occurs in every pattern string once, our algorithm improves on the algorithm of [14, 31] in functionality, without any increase in the asymptotic running time.

The (broadly defined) divide-and-conquer approach is covered by Subsections 4.6–4.12. In Subsection 4.6, we describe the all semi-local LCS algorithm from [48], running in time $O(\frac{mn}{\log^{0.5} n})$ for $m \leq n$ and reasonably close m and n. Although this algorithm is in general slower than the new iterative algorithm from Subsection 4.1, it illustrates well the divide-and-conquer approach, which is applicable to a wide range of other algorithmic problems. In particular, Subsection 4.7 describes an efficient algorithm given in [47, 49] for all semi-local LCS on permutations. Direct applications of this algorithm, also given in [47, 49], include an improved algorithm for the cyclic LCS problem on permutations, described in Subsection 4.8, and for the maximum clique problem on a circle graph, described in Subsection 4.9.

In Subsection 4.10, we propose a new method for local string comparison in windows, which has the potential to serve as a highly sensitive alternative to the standard "dot plot" method of biological string comparison. In contrast with the "dot plot" method, which only accounts for character substitutions, our method accounts for character insertions, deletions and substitutions. We describe the new method in terms of the window-window and window-substring LCS problems, and give an algorithm running in time $O(mn)$.

Subsection 4.11 is concerned with gene assembly from candidate exons, which is an important problem in computational biology. Among the most successful approaches to this problem is *spliced alignment*, proposed by Gelfand et al., which scores different candidate exon chains within a DNA sequence of length m by comparing them to a known related gene sequence of length n, $m = \Theta(n)$. Gelfand et al. gave an algorithm for spliced alignment running in time $O(n^3)$. Kent et al. considered sparse spliced alignment, where the number of candidate exons is $O(n)$, and proposed an algorithm for this problem running in time $O(n^{2.5})$. We describe our result from [45], which is based on a generalisation of window-substring string comparison, and gives an improved algorithm for sparse spliced alignment running in time $O(n^{2.25})$.

In Subsection 4.12, we consider the LCS problem and the local subsequence recognition problem for a compressed text of length $\bar{m}$ against an uncompressed pattern of length n. Cégielski et al. [11] gave an algorithm for local subsequence recognition running in time $O(\bar{m}n^2 \log n)$. We describe our result from [46], which gives a new LCS algorithm and an improved local subsequence recognition algorithm for a compressed text against an uncompressed pattern, both running in time $O(\bar{m}n^{1.5})$.

As already apparent to the reader, the aim of this paper is twofold: on one hand, to survey the techniques and applications of semi-local string comparison that have appeared in the author's previous papers; on the other hand, to introduce a number of new techniques and applications. At the end, we conclude that semi-local string comparison turns out to be a useful algorithmic plug-in, which unifies, and often improves on, a number of previous approaches to various substring- and subsequence-related problems.

2. Semi-local longest common subsequences

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. For two strings $a = \alpha_1\alpha_2 \dots \alpha_m$ and $b = \beta_1\beta_2 \dots \beta_n$ of lengths m, n respectively, the *longest common subsequence* (*LCS*) *problem* consists in computing the length of the longest string that is a subsequence of both a and b. We will call this length the *LCS score* of the strings.

In [48], we introduced the following problem.

Definition 1. The *all 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;
- 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 generalisation of the LCS problem. In the rest of this paper, we present several algorithmic techniques for the all semi-local LCS problem, and some of its applications.

A traditional distinction, especially in computational biology, is between global (full string against full string) and local (all substrings against all substrings) comparison. Our problem lies in between, hence the term "semi-local". Many string comparison algorithms output either a single optimal comparison score across all local comparisons, or a number of local comparison scores that are "sufficiently close" to the globally optimal. In contrast with this approach, we require to output all the locally optimal comparison scores.

In addition to standard integer indices $\dots, -2, -1, 0, 1, 2, \dots$, we use *odd half-integer* indices $\dots, -\frac{5}{2}, -\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \dots$. For two numbers i, j, we write $i \trianglelefteq j$ if $j - i \in \{0, 1\}$, and $i \triangleleft j$ if $j - i = 1$. We denote

$$[i : j] = \{i, i+1, \dots, j-1, j\}$$

$$\langle i : j \rangle = \left\{ i + \frac{1}{2}, i + \frac{3}{2}, \dots, j - \frac{3}{2}, j - \frac{1}{2} \right\}$$

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 $j - i$ interval *length*.

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 non-trivial core, and will be trivial (e.g., equal to an infinite identity matrix) outside of this core. An infinite permutation matrix with finite non-trivial core can be represented by its core nonzeros.

Given a permutation matrix D and a set of its columns R, the permutation submatrix *induced by* R is obtained by deleting from D all columns not belonging to R, and then deleting from the remaining submatrix all zero rows. A permutation submatrix induced by a set of rows is defined analogously.

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

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

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

$$D(i,j) = d\left(i - \frac{1}{2}, j + \frac{1}{2}\right) - d\left(i - \frac{1}{2}, j - \frac{1}{2}\right) - d\left(i + \frac{1}{2}, j + \frac{1}{2}\right) + d\left(i + \frac{1}{2}, j - \frac{1}{2}\right)$$

When matrix d is a distribution matrix of D, 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 "distribution matrix of a permutation matrix".

3. Algorithmic techniques

The rest of this paper is based on the framework for the all semi-local LCS problem developed in [48], which refines the approach of [5,42]. For completeness, we include most background definitions and results from [48].

3.1. 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 non-matches to edges scoring 0.

Definition 2. 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 $l \in [1 : m]$, $i \in [1 : n]$:

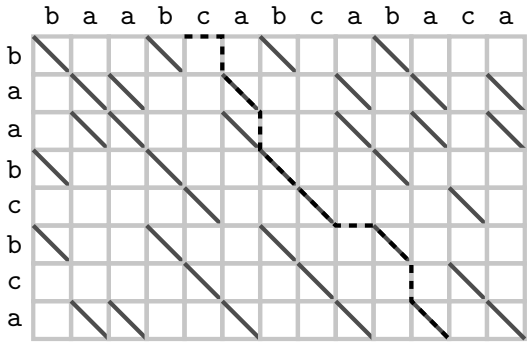

FIGURE 1. An alignment dag and a highest-scoring path.

- horizontal edge $v_{l,i-1} \rightarrow v_{l,i}$ and vertical edge $v_{l-1,i} \rightarrow v_{l,i}$ are both always present in G and have score 0;
- diagonal edge $v_{l-1,i-1} \rightarrow v_{l,i}$ 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 an $m \times n$ alignment dag, where the diagonal edge $v_{l-1,i-1} \rightarrow v_{l,i}$ is present, iff $\alpha_i = \beta_j$.

Figure 1 shows the alignment dag corresponding to strings $a =$ "`baabcbca`", $b =$ "`baabcabcabaca`" (an example borrowed from [5]).

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. This is also equivalent to finding the corresponding shortest distances in an undirected graph, obtained from the alignment dag by assigning length 1 to vertical and horizontal edges, length 0 to diagonal edges, and ignoring edge directions. Thus, the problem is related to classical shortest path problems. In particular, given a planar graph on mn nodes with nonnegative edge lengths, the algorithm of [28] builds in time $O(mn \log(m+n))$ a data structure of size $O(mn \log(m+n))$, that allows to query in time $O(\log(m+n))$ the distance between any node on the graph boundary and any other node. Assuming $n = \Theta(m)$, this data structure is redundant for the all semi-local LCS problem, since in this case the solution can be represented explicitly in size $O(mn)$ with query time $O(1)$. However, the data structure of [28] applies to the general range of m and n, and

to the more general string-prefix and string-suffix LCS problems, generalising the corresponding algorithm of [42].

We aim to obtain a more efficient solution to the all semi-local LCS problem by exploiting the special structure of the alignment dag. To describe our algorithms, we need to modify the definition of the alignment dag by embedding the finite grid of nodes into in an infinite grid.

Definition 3. 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 $l, i \in [-\infty : +\infty]$:

- horizontal edge $v_{l,i-1} \to v_{l,i}$ and vertical edge $v_{l-1,i} \to v_{l,i}$ are both always present in G^+ and have score 0;
- when $l \in [1 : m]$, $i \in [1 : n]$, diagonal edge $v_{l-1,i-1} \to v_{l,i}$ is present in G^+ iff it is present in G; if present, it has score 1;
- otherwise, diagonal edge $v_{l-1,i-1} \to v_{l,i}$ 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 dag G^+ is the extension of 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 4. Given an $m \times n$ alignment dag G, its *extended horizontal* (respectively, *vertical*) *highest-score matrix*[1] is an infinite matrix defined by

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

$$A^*(i,j) = \max score(v_{i,0} \rightsquigarrow v_{j,n}) \qquad i, j \in [-\infty : +\infty] \tag{3}$$

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 Figure 1, the highlighted path has score 5, and corresponds to the value $A(4, 11) = 5$, which is equal to the LCS score of string a and 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. For most of this subsection, we will concentrate on the properties of horizontal highest-score matrices, referring to them simply as "highest-score matrices".

[1]These matrices are called "DIST matrices", e.g., in [14, 42], and "score matrices" in [49]. We have chosen a different terminology to reflect better the score-maximising nature of the matrix elements, and to avoid confusion with pairwise score matrices used in comparative genomics (see, e.g., [25]).

By symmetry, vertical highest-score matrices will have analogous properties. We assume $i, j \in [-\infty : +\infty]$, unless indicated otherwise.

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

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

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

Theorem 1. *A highest-score matrix has the following properties:*

$$A(i, j) \trianglelefteq A(i-1, j); \tag{4}$$

$$A(i, j) \trianglelefteq A(i, j+1); \tag{5}$$

$$\textit{if} \quad A(i, j+1) \triangleleft A(i-1, j+1), \quad \textit{then} \quad A(i, j) \triangleleft A(i-1, j); \tag{6}$$

$$\textit{if} \quad A(i-1, j) \triangleleft A(i-1, j+1), \quad \textit{then} \quad A(i, j) \triangleleft A(i, j+1). \tag{7}$$

Proof. A path $v_{0,i-1} \rightsquigarrow v_{m,j}$ can be obtained by first following a horizontal edge of score 0: $v_{0,i-1} \rightarrow v_{0,i} \rightsquigarrow v_{m,j}$. Therefore, $A(i, j) \leq A(i-1, j)$. On the other hand, any path $v_{0,i-1} \rightsquigarrow v_{m,j}$ consists of a subpath $v_{0,i-1} \rightsquigarrow v_{l,i}$ of score at most 1, followed by a subpath $v_{l,i} \rightsquigarrow v_{m,j}$. Therefore, $A(i, j) \geq A(i-1, j) - 1$. We thus have (4) and, by symmetry, (5).

A crossing pair of paths $v_{0,i} \rightsquigarrow v_{m,j}$ and $v_{0,i-1} \rightsquigarrow v_{m,j+1}$ can be rearranged into a non-crossing pair of paths $v_{0,i-1} \rightsquigarrow v_{m,j}$ and $v_{0,i} \rightsquigarrow v_{m,j+1}$. Therefore, we have *the Monge property*:

$$A(i, j) + A(i-1, j+1) \leq A(i-1, j) + A(i, j+1)$$

Rearranging the terms

$$A(i-1, j+1) - A(i, j+1) \leq A(i-1, j) - A(i, j)$$

and applying (4), we obtain (6) and, by symmetry, (7). □

The properties of Theorem 1 are symmetric with respect to i and $n - j$. Alves et al. [5] introduce the same properties but do not make the most of their symmetry. We aim to exploit symmetry to the full.

Corollary 1. *A highest-score matrix has the following properties:*

$$\begin{aligned}
&\textit{if} \quad A(i, j) \triangleleft A(i-1, j), \quad \textit{then} \quad A(i, j') \triangleleft A(i-1, j') \quad \textit{for all} \quad j' \leq j; \\
&\textit{if} \quad A(i, j) = A(i-1, j), \quad \textit{then} \quad A(i, j') = A(i-1, j') \quad \textit{for all} \quad j' \geq j; \\
&\textit{if} \quad A(i, j) \triangleleft A(i, j+1), \quad \textit{then} \quad A(i', j) \triangleleft A(i', j+1) \quad \textit{for all} \quad i' \geq i; \\
&\textit{if} \quad A(i, j) = A(i, j+1), \quad \textit{then} \quad A(i', j) = A(i', j+1) \quad \textit{for all} \quad i' \leq i.
\end{aligned}$$

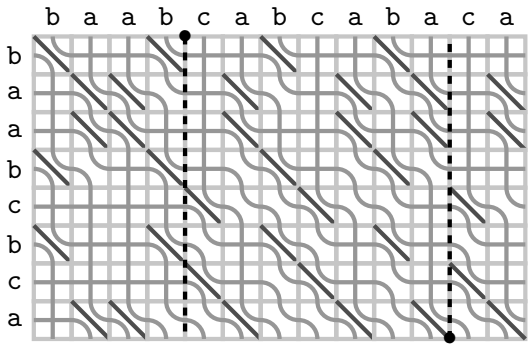

FIGURE 2. An alignment dag and the seaweeds.

Proof. These are the well-known properties of matrix A and its transpose A^T being *totally monotone*. In both pairs, the properties are each other's contrapositive, and follow immediately from Theorem 1. □

Informally, Corollary 1 says that the inequality between the corresponding elements in two successive rows (respectively, columns) "propagates to the left (respectively, downwards)", and the equality "propagates to the right (respectively, upwards)". Recall that by convention, $A(i,j) = j - i$ for all index pairs $j < i$. Therefore, we always have an inequality between the corresponding elements in successive rows or columns in the lower triangular part of matrix A. If we fix i and scan the set of indices j from $j = -\infty$ to $j = +\infty$, an inequality may change to an equality at most once. We call such a value of j *critical* for i. Symmetrically, if we fix j and scan the set of indices i from $i = +\infty$ to $i = -\infty$, an inequality may change to an equality at most once, and we can identify values of i that are critical for j. Crucially, for all pairs (i,j), index i will be critical for j if and only if index j is critical for i. To capture this property, which is central to our method, we give the following definition.

Definition 5. An odd half-integer point $(i,j) \in \langle -\infty : +\infty \rangle^2$ is called *A-critical*, if

$$A\left(i+\frac{1}{2}, j-\frac{1}{2}\right) \lhd A\left(i-\frac{1}{2}, j-\frac{1}{2}\right) = A\left(i+\frac{1}{2}, j+\frac{1}{2}\right) = A\left(i-\frac{1}{2}, j+\frac{1}{2}\right)$$

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

Corollary 2. *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.*

Proof. By Corollary 1 and Definition 5. □

Figure 2 shows the alignment dag of Figure 1 along with the critical points. In particular, every critical point (i,j), where $i,j \in \langle 0 : n \rangle$, is represented by

a *seaweed*[2], 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), where either $i \in \langle -m : 0\rangle$ or $j \in \langle n : n+m\rangle$ (or both). In particular, every critical point (i,j), where $i \in \langle -m : 0\rangle$ (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 $v_{m+n-j+\frac{1}{2},n}$). For the purposes of this subsection, the specific layout of the seaweeds between their endpoints is not important. However, this layout will become meaningful in the context of the algorithms described in the next section.

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 6. Point (i_0,j_0) *dominates*[3] 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[4]. 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 defines uniquely a highest-score matrix, and gives a simple formula for recovering the matrix elements.

Theorem 2. *For all* $i_0,j_0 \in [-\infty : +\infty]$, *we have*

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

Proof. Induction on $j_0 - i_0$. Denote $d = d_A(i_0,j_0)$.
Induction base. Suppose $i_0 \geq j_0$. Then $d = 0$ and $A(i_0,j_0) = j_0 - i_0$.
Inductive step. Suppose $i_0 < j_0$. Let d' denote the number of critical points in $\langle i_0 : n\rangle \times \langle 0 : j_0 - 1\rangle$. By the inductive hypothesis, $A(i_0,j_0-1) = j_0 - 1 - i_0 - d'$. We have two cases:

1. There is a critical point $(i, j_0 - \frac{1}{2})$ for some $i \in \langle i_0 : n\rangle$. Then $d = d' + 1$ and $A(i_0,j_0) = A(i_0,j_0-1) = j_0 - i_0 - d$ by Corollary 1.

[2]This imaginative term was suggested by Yu. V. Matiyasevich.
[3]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.
[4]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.

2. There is no such critical point. Then $d = d'$ and $A(i_0, j_0) = A(i_0, j_0-1)+1 = j_0 - i_0 - d$ by Corollary 1.

In both cases, the theorem statement holds for $A(i_0, j_0)$. □

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

There is a close connection between Theorem 2 and the canonical structure of general Monge matrices described in [10].

By Theorem 2, 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 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.

Definition 7. Given an infinite permutation matrix D, its *core* is a square (possibly semi-infinite) submatrix defined by the index range $[i_0 : j_0] \times [i_1 : j_1]$, where $j_0 - i_0 = j_1 - i_1$ (as long as both these values are defined), and such that for all off-core elements $D(i, j)$, we have $D(i, j) = 1$ iff $j - i = j_0 - i_0$ 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 matrix D has nonzeros on the off-core extension of the core's main diagonal.

The following statements are an immediate consequence of the definitions.

Corollary 3. *A core of an infinite permutation matrix is a (possibly semi-infinite) permutation matrix.*

Corollary 4. *Given an alignment dag A as described above, the corresponding permutation matrix D_A has core of size $m+n$, defined by $i \in \langle -m, n\rangle$, $j \in \langle 0, m+n\rangle$.*

In Figure 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 2, 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.

The following theorems are derived from two relevant geometric results, one classical and one recent. In view of algorithmic applications, we extend the following theorem from [48] to cover, in addition to ordinary matrix element queries, also *batch queries* returning a set of elements in a row (or column) of the matrix.

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

- *has size $O((m+n)\log(m+n))$;*
- *can be built in time $O((m+n)\log(m+n))$;*
- *allows to query an individual element of A in time $O(\log^2(m+n))$;*
- *allows to query r consecutive elements in a row (or column) of A in time $O(r+\log^2(m+n))$.*

Proof. The structure in question is a 2D range tree [9] (see also [38]), built on the set of core nonzeros in D_A. There are $m+n$ such nonzeros, hence the total number of nodes in the tree is $O((m+n)\log(m+n))$. A dominance counting query on the set of core nonzeros can be answered by accessing $O(\log(m+n)^2)$ of the tree nodes. A dominance counting query on the set of off-core nonzeros can be answered by a simple constant-time index calculation (note that the result of such a query can only be non-zero when the query is made outside the core subgraph of the alignment dag). The sum of the above two dominance counting queries provides the total number of nonzeros dominated by the query point (i_0, j_0). The value $A(i_0, j_0)$ can now be obtained by Theorem 2.

Given a batch query of r consecutive elements in a row (or column) of A, the first element is obtained as described above, in time $O(\log^2(m+n))$. For each subsequent element, the count of dominated nonzeros is changed by at most 1, and can be updated in constant time by a single index comparison. Therefore, the whole batch query can be answered in time $O(r+\log^2(m+n))$. □

Theorem 4. *Given the implicit representation D_A of a highest-score matrix A, there exists a data structure which*

- *has size $O(m+n)$;*
- *allows to query an individual element of A in time $O(\frac{\log(m+n)}{\log\log(m+n)})$.*

Proof. As above, but the range tree is replaced by the asymptotically more efficient data structure of [24]. □

While the data structure used in Theorem 4 provides better asymptotics, the range tree used in Theorem 3 is simpler, requires a less powerful computation model, and is more likely to be practical. Therefore, we will be using Theorem 3 as our main technique for efficient semi-local LCS queries.

We conclude this subsection by formulating yet another previously unexploited symmetry of the all semi-local LCS problem. This time, we consider both the horizontal highest-score matrix A as in (2), and the vertical highest-score matrix A^* as in (3). We show a simple one-to-one correspondence between the

implicit representations of A and A^*, allowing us to switch easily between these representations.

Theorem 5. *We have* $D_A(i,j) = D_{A^*}(-i, m+n-j)$.

Proof. Straightforward case analysis based on Definition 5. □

3.2. 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 dag G is the *concatenation* of dags G_1 and G_2. Let A, B, C denote the highest-score matrices defined respectively by dags G_1, G_2, G. Our goal is, given matrices A, B, to compute matrix C efficiently. We call this procedure *highest-score matrix multiplication*.

Highest-score matrix multiplication can be performed naively in time $O((M+n)^3)$ by standard matrix multiplication over the $(\max,+)$-semiring. By exploiting the Monge property of the matrices, the time complexity of highest-score matrix multiplication can be reduced to $O((M+n)^2)$, which is optimal if the matrices are represented explicitly. However, a further reduction in the time complexity of highest-score matrix multiplication is possible, by using the implicit matrix representation and algorithmic ideas introduced in Subsection 3.1.

The implicit representation of matrices A, B, C consists of respectively $M+n$, $m+n$, $M+m+n$ core nonzeros. Alves et al. [5] use a similar representation; however, for their algorithm, n nonzeros per matrix are sufficient. They describe a procedure equivalent to highest-score matrix multiplication for the special case $m=1$. By iterating this procedure, they obtain a string-substring LCS algorithm running in time $O(mn)$. In a generalised form, the main technique of [5] can be stated as follows.

Lemma 1 (Generalised from [5]). *Consider the concatenation of alignment dags as described above, with highest-score matrices A, B, C. Given the implicit representations of A, B, the implicit representation of C can be computed in time $O(M+mn)$ and memory $O(M+n)$.*

The following results from [48] improve on Lemma 1 for $n = o(m^2)$.

Definition 8. 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

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

Lemma 2 ([48]). *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*

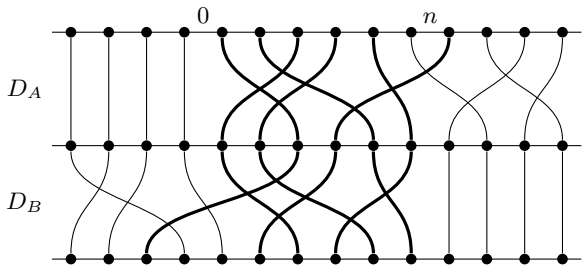

FIGURE 3. An illustration of Lemma 3.

$d_A \odot d_B = d_C$. Given the nonzeros of D_A, D_B, the nonzeros of D_C can be computed in time $O(n^{1.5})$ and memory $O(n)$.

Lemma 3 ([48]). *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*

$$D_A(i,j) = D_C(i,j) \qquad \text{for} \quad i \in \langle -\infty : +\infty \rangle\,, \quad j \in \langle n : +\infty \rangle \tag{8}$$

$$D_B(j,k) = D_C(j,k) \qquad \text{for} \quad j \in \langle -\infty : 0 \rangle\,, \quad k \in \langle -\infty : +\infty \rangle \tag{9}$$

Equations (8)–(9) 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 $O(n^{1.5})$ and memory $O(n)$.

The above lemma is illustrated by Figure 3. Three horizontal lines represent respectively the index ranges of i, j, k. The nonzeros in D_A (respectively, D_B) are shown by top-to-middle (respectively, middle-to-bottom) seaweeds; thick seaweeds correspond to the nonzeros covered by (8)–(9), and thin seaweeds to the remaining nonzeros. By Lemma 3, the nonzeros in D_C covered by (8)–(9) 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 2.

Lemma 4 ([48]). *Consider the concatenation of alignment dags as described above, with highest-score matrices A, B, C. Given the implicit representations of A, B, the implicit representation of C can be computed in time $O(M+n^{1.5})$ and memory $O(M+n)$.*

Note that Lemma 4 improves on Lemma 1, if and only if $n = o(m^2)$. In the rest of this subsection, we obtain an even better, unconditional improvement on Lemma 1, which is applicable in particular if $m \ll n$.

The algorithm of Lemma 4 employs a subroutine for multiplying infinite permutation-distribution matrices, described by Lemma 3. The subroutine works

in the special case where both multiplicands have semi-infinite core. We now give a subroutine for the complementary special case, where one multiplicand's core is unbounded, and the other's is finite.

Lemma 5. *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 trivial infinite core $\langle -\infty, +\infty \rangle^2$ (respectively, finite core $\langle 0 : n \rangle^2$). Let d_A, d_B, d_C be the respective distribution matrices, and assume $d_A \odot d_B = d_C$. We have*

$$D_A(i,j) = D_C(i,j) \qquad \text{for} \quad i \in \langle -\infty : +\infty \rangle\,, \quad j \in \langle -\infty : 0 \rangle \cup \langle n : +\infty \rangle \tag{10}$$

Equation (10) *cover all but n nonzeros in each of D_A, D_B, D_C. These remaining nonzeros have $i \in \langle -\infty : +\infty \rangle$, $j, k \in \langle 0 : n \rangle$. Given the n remaining nonzeros in each of D_A, D_B, the n remaining nonzeros in D_C can be computed in time $O(n^{1.5})$ and memory $O(n)$.*

Proof. It is straightforward to check equality (10), by (2) and Definition 8. Informally, each nonzero of D_C appearing in (10) is obtained as a direct combination of a core nonzero of D_A and an off-core nonzero of D_B. All remaining nonzeros of D_A and D_B are core, and determine collectively the remaining nonzeros of D_C. However, this time the direct one-to-one relationship between nonzeros of D_C and pairs of nonzeros of D_A and D_B need not hold.

Observe that all the off-core nonzeros of D_B with $j \in \langle -\infty : 0 \rangle$ are dominated by each of the remaining nonzeros of D_B. Furthermore, none of the nonzeros of D_B with $j \in \langle n : +\infty \rangle$ are dominated by any of the remaining nonzeros of D_B. Hence, the nonzeros of D_A appearing in (10) cannot affect the computation of the remaining nonzeros of D_C. We can therefore simplify the problem by looking for the remaining nonzeros of D_C in an induced submatrix. More precisely, let D'_A (respectively, D'_B, D'_C) be a permutation submatrix of D_A (respectively, D_B, D_C), induced by the set of columns with $j \in \langle 0 : n \rangle$ (respectively, $k \in \langle 0 : n \rangle$ for both D'_B, D'_C). We define d'_A, d'_B, d'_C accordingly. The index order is preserved in each submatrix, so the dominance relation is not affected.

It is easy to check that $d'_A \odot d'_B = d'_C$, iff $d_A \odot d_B = d_C$. Matrices D'_A, D'_B, D'_C satisfy the conditions of Lemma 2. Therefore, given the set of nonzeros of D'_A, D'_B, the set of nonzeros of D'_C can be computed in time $O(n^{1.5})$ and memory $O(n)$. □

The above lemma is illustrated by Figure 4, using the same conventions as Figure 3.

We are now able to give a new algorithm for highest-score matrix multiplication, which is an unconditional improvement on Lemma 1.

Lemma 6. *Consider the concatenation of alignment dags as described above, with highest-score matrices A, B, C. Given the implicit representations of A, B, the implicit representation of C can be computed in time $O(M + m^{0.5}n)$ and memory $O(M + n)$.*

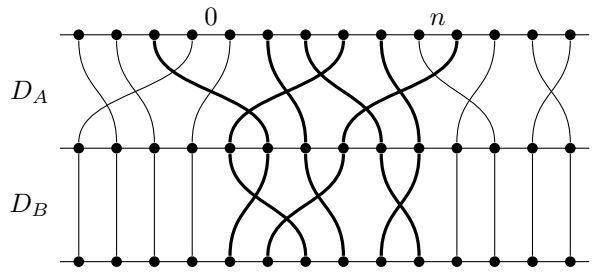

FIGURE 4. An illustration of Lemma 5.

Proof. By Lemma 5, all but n core nonzeros of D_C can be obtained in time and memory $O(M+n)$. We now show how to obtain the remaining core nonzeros in time $O(m^{0.5}n)$, instead of time $O(n^{1.5})$ given by Lemma 5.

The main idea is to decompose matrix d_B into a $(\min,+)$-product of permutation-distribution matrices with small core. The decomposition is described in terms of density matrices, and proceeds recursively. In each recursive step, we define infinite permutation matrices D'_B, D''_B, that are obtained from the density matrix D_B as follows.

Recall that core nonzeros in D_B belong to the range $\langle -m : n\rangle \times \langle 0 : m+n\rangle$. Intuitively, the idea is to split the range of each index into two blocks:

$$\langle -m : n\rangle = \left\langle -m : \frac{n}{2}\right\rangle \cup \left\langle \frac{n}{2} : n\right\rangle$$
$$\langle 0 : m+n\rangle = \left\langle 0 : \frac{n}{2}\right\rangle \cup \left\langle \frac{n}{2} : m+n\right\rangle$$

Note that the splits are not uniform, and that among the resulting four rectangular blocks in D_B, the block $\langle \frac{n}{2} : n\rangle \times \langle 0 : \frac{n}{2}\rangle$ cannot contain any nonzeros. We process the remaining three blocks individually, gradually introducing nonzeros in matrices D'_B, D''_B until they become permutation matrices. Matrix D'_B (respectively, D''_B) will have core $\langle -m : \frac{n}{2}\rangle \times \langle 0 : m+\frac{n}{2}\rangle$ (respectively, $\langle \frac{n}{2} : m+n\rangle \times \langle m+\frac{n}{2} : 2m+n\rangle$).

First, we consider all nonzeros in D_B at $(j,k) \in \langle -m : \frac{n}{2}\rangle \times \langle 0 : \frac{n}{2}\rangle$. For every such nonzero, we introduce a nonzero in D'_B at (j,k). We also consider all nonzeros in D_B at $(j,k) \in \langle \frac{n}{2} : n\rangle \times \langle \frac{n}{2} : m+n\rangle$. For every such nonzero, we introduce a nonzero in D''_B at $(m+j, m+k)$.

Now consider all nonzeros in D_B at $(j,k) \in \langle -m : \frac{n}{2}\rangle \times \langle \frac{n}{2} : m+n\rangle$. There are exactly m such nonzeros; we denote them by (j_0,k_0), (j_1,k_1), $\ldots$, (j_{m-1},k_{m-1}), where $j_0 < j_1 < \cdots < j_{m-1}$. For each nonzero (j_t,k_t), we introduce a nonzero in D'_B at $(j_t, \frac{n+1}{2}+t)$, and a nonzero in D''_B at $(\frac{n+1}{2}+t, m+k_t)$.

Finally, we introduce the off-core nonzeros in D'_B, D''_B at (j,k), $k-j=m$. The recursive step is completed.

Let d'_B, d''_B be the distribution matrices of D'_B, D''_B. Let $d^*_B = d'_B \odot d''_B$, and $d^*_C = d_A \odot d^*_B = d_A \odot d'_B \odot d''_B$, and define D^*_B, D^*_C accordingly. By the construction of the decomposition of d_B, matrices D_B and D^*_B (as well as d_B

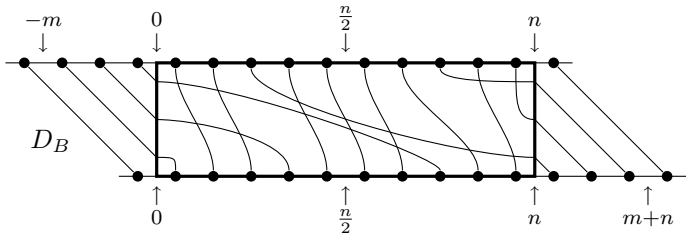

FIGURE 5. Proof of Lemma 6: the original matrix D_B.

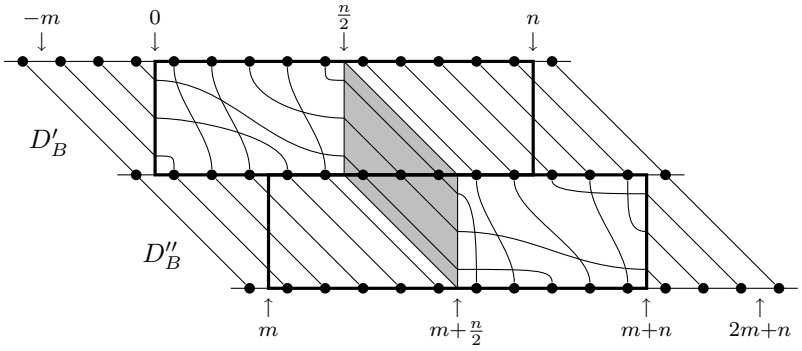

FIGURE 6. Proof of Lemma 6: the decomposition of D_B.

and d^*_B) are related by a simple shift: for all (i,k), $i,k \in \langle -\infty, +\infty \rangle$, we have $D_B(j,k) = D^*_B(j,k+m)$. Consequently, matrices D_C and D^*_C are related by a similar shift: for all (i,k), $i,k \in \langle -\infty, +\infty \rangle$, we have $D_C(i,k) = D^*_C(i,k+m)$.

The described decomposition process continues recursively, as long as $n \geq m$. The problem of computing matrix d_C is thus reduced, up to an index shift, to n/m instances of multiplying permutation-distribution matrices. In every instance, one of the multiplied matrices has core of size $O(m)$. By Lemma 5, the non-trivial part of every such multiplication can be performed in time $O(m^{1.5})$ and memory $O(m)$. The trivial parts of all these multiplications can be combined into a single scan of the nonzeros of each of D_A, D_B, and can therefore be performed in time and memory $O(M+n)$. Hence, the whole computation can be performed in time $O(M + (n/m) \cdot m^{1.5}) = O(M + m^{0.5}n)$ and memory $O(M+n)$. □

The decomposition of matrix D_B in the proof of Lemma 6 is illustrated by Figures 5, 6. The rectangle corresponding to D_B is split into two half-sized rectangles, corresponding to D'_B and D''_B. Each of the new rectangles is completed to a full-sized rectangle by trivial extension; then, the rectangles are arranged vertically with a shift by m. The seaweeds that do not cross the partition are

preserved by the construction, up to a shift by m. The seaweeds that do cross the partition are also preserved up to a shift by m, by passing them through a parallelogram-shaped "buffer zone". Note that this construction makes the latter class of seaweeds uncrossed in D'_B, and preserves all their original crossings in D''_B.

3.3. 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 9. 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 an implicit representation of A, the corresponding *partial implicit representation* consists of all nonzeros $A(i,j)$, where 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; their number is at least n and at most $2n$ (note that the size of a 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 6. *Given the partial implicit representation of a highest-score matrix A, there exists a data structure which*

- *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 3, 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 7. *Consider the concatenation of alignment dags as described in Subsection 3.2, 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 $O(n^{1.5})$ 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 $d'_A \odot d'_B = d'_C$, iff $d_A \odot d_B = d_C$. Matrices D'_A, D'_B, D'_C satisfy the conditions of Lemma 3, therefore all but n of the core nonzeros in the required partial implicit representation can be obtained by (8)–(9) in time and memory $O(n)$, and the remaining n core nonzeros in time $O(n^{1.5})$ and memory $O(n)$. □

4. Algorithmic applications

4.1. Semi-local LCS: the iterative algorithm

We now describe our new approach to the all semi-local LCS problem. First, we give a simple algorithm, running in overall time $O(mn)$, matching the running time of the standard dynamic programming algorithm for the (global) LCS problem (see, e.g., [15,21]). Our algorithm is similar to the string-substring LCS algorithm of [5], but is adapted to solve a more general problem than in [5], and is expressed in the new terminology and notation.

Algorithm 1 (All semi-local LCS, simple iteration).
Input: strings a, b of length m, n, respectively.
Output: implicit highest-score matrix on strings a, b.
Description. We build the output matrix D incrementally. The matrix is conceptually infinite, and is initialised as a shifted identity matrix: $D(i,j) = I(i,j-m)$ for all $i,j \in \langle -\infty : +\infty \rangle$. Such a matrix corresponds to the trivial alignment dag, where all possible diagonal edges are present.

We now perform a sequence of incremental transformations on the initial trivial alignment dag. We scan the dag from top to bottom and from left to right, in each step removing a diagonal edge from the current dag, if the corresponding diagonal edge is absent from the alignment dag of the input strings.

Let D be the implicit highest-score matrix corresponding to the current state of the dag. An incremental transformation corresponding to an index pair $l, i \in [1 : n]$ is reflected in matrix D as follows. Let $j_0 = i+m-l-\frac{1}{2}$ and $j_1 = i+m-l+\frac{1}{2}$. Let $i_0, i_1 \in \langle -m : m+n \rangle$ be respectively the row indices of the nonzeros in columns j_0 and j_1. We assign

if $i_0 < i_1$ and $\alpha_l \neq \beta i$ then
$D(i_0,j_0) \leftarrow 0\,; \quad D(i_0,j_1) \leftarrow 1\,; \quad D(i_1,j_0) \leftarrow 1\,; \quad D(i_1,j_1) \leftarrow 0\,;$
otherwise D unchanged

By Definition 5, this assignment maintains the invariant "the current state of matrix D is the implicit highest-score matrix for the current state of the alignment dag". Therefore, the final state of matrix D provides the required output.
Cost analysis. For every pair (l,i), the incremental transformation runs in constant time. Therefore, the total computation work is $O(mn)$, assuming the RAM model with word size at least $O(\log \log n)$.

The memory cost is dominated by storing the input and the core of current implicit highest-score matrix. Therefore, the total memory cost is $O(m+n)$. □

The above algorithm has a simple interpretation in terms of the seaweeds of Figure 2. Each seaweed is traced across the alignment dag and keeps straight by default. However, the seaweed bends away, if it about to meet a diagonal dag edge, or another seaweed *which is has previously crossed*. Therefore, two given seaweeds are allowed to cross at most once. Notice that the same property holds for the highest-scoring paths in the dag. The layout of the seaweeds in Figure 2 corresponds precisely to the sequence of states of Algorithm 1.

Algorithm 1 can now be modified to achieve subquadratic running time, using the method of small-block precomputation originating in [7]. Using this method, Masek and Paterson [35] gave an algorithm for the (global) LCS problem running in time $O(\frac{mn \log\log n}{\log n})$ for an unbounded alphabet (as observed, e.g., in [1]), and in time $O(\frac{mn}{\log n})$ for a finite (bounded) alphabet[5]. We now give an algorithm with similar running time, solving the more general semi-local LCS problem. This algorithm also improves on, and supersedes, our subquadratic semi-local LCS algorithm from [48].

From now on, we assume without loss of generality that $m \leq n$. We will also assume that m and n are reasonably close, so that $m = \Omega(\frac{\log n}{\log\log n})$. We do not make any assumptions on the size of the alphabet, except that the characters can be compared in constant time. We call two strings of equal length *isomorphic*, if one can obtained from the other by a permutation of the alphabet.

Algorithm 2 (All semi-local LCS, iteration with block precomputation).
Input, output: as in Algorithm 1; we let $t = \frac{\log n}{4 \cdot \log\log n}$, where the logarithms are base 2, and assume that $t \leq m \leq n$.
Description. As in Algorithm 1, we build the output matrix incrementally, by performing a sequence of transformations on the initial trivial alignment dag. This time, we scan the dags from top to bottom and from left to right in blocks of size $t \times t$, in each step replacing a trivial block of the current dag by the corresponding block in the alignment dag of the input strings.

Let D be the matrix corresponding to the current dag. Consider a dag block indexed by $[l-t, l] \times [i-t, i]$, where $l \in [t : m]$, $i \in [t : n]$. An incremental transformation corresponding to this block is reflected in matrix D as follows. Consider the $2t \times 2t$ permutation submatrix of D, induced by the column range $\langle i+m-l-t : i+m-l+t \rangle$. The transformed matrix D is obtained by replacing this permutation submatrix with a new permutation submatrix on the same row and column sets. The replacement permutation submatrix is a function of the original one, and of the substrings $\alpha_{l-t+1} \ldots \alpha_l$ and $\beta_{i-t+1} \ldots \beta_i$. Therefore, for each of the $(2t)!$ possible original permutation matrices, and for each of at most $(2t)^{2t}$ possible non-isomorphic substring pairs, the replacement permutation matrix can

[5]Assuming unit-cost RAM, both the algorithm of [35] and our algorithm can be speeded-up by another factor of $O(\log n)$ (see, e.g., [52]).

be precomputed in advance. The precomputation of each individual replacement matrix can be performed by Algorithm 1.

Given an implicit representation of matrix D and a set of $2t$ columns, the set of rows in the corresponding induced permutation submatrix can clearly be found in time $O(t)$. An additional requirement necessary for efficient searching of replacement submatrices is that the row indices of each induced submatrix are obtained in ascending order. This requirement is trivially satisfied at the start of the computation, and is maintained in every block processing step inductively as follows. As before, consider the block $[l-t, l] \times [i-t, i]$, where $l \in [t : m]$, $i \in [t : n]$. By the inductive hypothesis, as an input to processing this block we have two permutation submatrices, induced respectively by the column ranges $\langle i+m-l-t : i+m-l\rangle$ and $\langle i+m-l : i+m-l+t\rangle$; the row indices of each submatrix are in ascending order. The first (respectively, second) of these submatrices is obtained as the output of the block processing step for the block immediately to the left (respectively, above) of the current block (if such a predecessor block exists). The two sequences of row indices can be merged in time $O(t)$ into one sequence, still in ascending order, defining the required permutation submatrix induced by the column range $\langle i+m-l-t : i+m-l+t\rangle$, with row indices in ascending order. The precomputed replacement matrix can now be found and substituted for the original one in time $O(t)$; the sequence of row indices is unchanged and is still in ascending order. We now scan this sequence in order to split it into two subsequences, each still in ascending order, corresponding in the new matrix to row indices of the two permutation submatrices, induced as before by the the column ranges $\langle i+m-l-t : i+m-l\rangle$ and $\langle i+m-l : i+m-l+t\rangle$. The first (respectively, second) of these submatrices will serve as the input to the block processing step for the block immediately below (respectively, to the right) of the current block (if such a successor block exists). The overall block processing time is thus $O(t)$.

The described block processing procedure maintains the invariant "current matrix D is the implicit highest-score matrix for the current dag". Therefore, the final state of matrix D provides the required output.

Cost analysis. In the precomputation stage, there are at most $(2t)! \cdot (2t)^{2t}$ problem instances, each of which costs $O(t^2)$. Therefore, the total cost of the precomputation is at most is $(2t)! \cdot (2t)^{2t} \cdot O(t^2) = o((2t)^{4t}) = o(2^{4t \cdot \log(2t)}) = o(2^{4 \cdot \frac{\log n}{\log\log n} \cdot \log\log n}) = o(2^{\log n}) = o(n)$.

In the main computation stage, there are $\frac{mn}{t^2}$ block processing steps, each of which costs $O(t)$. Therefore, the total computation work is $\frac{mn}{t^2} \cdot O(t) = O(\frac{mn \cdot \log\log n}{\log n})$, under the same assumptions on the RAM model as in Algorithm 1. □

In the case of an unbounded alphabet, the running time of Algorithm 2 matches the running time of the global LCS algorithm by Masek and Paterson [35]. In the case of a bounded (constant-sized) alphabet, the running time of Algorithm 2 remains unchanged, while the running time of the algorithm by Masek and Paterson can be improved by a factor of $O(\log\log n)$.

Note that the dag cells in Algorithm 1 and dag blocks in Algorithm 2 can be processed in any order consistent with the grid-like left-to-right and top-to-bottom data dependency pattern. While the most obvious processing order choice is lexicographic (either by rows or by columns), one can also apply the recursively defined order of [17] to obtain a cache-efficient version of both algorithms.

4.2. Cyclic LCS

Given strings a, b of length m, n respectively, the *cyclic LCS* problem consists in computing the length of the longest string that is a subsequence of a and of some cyclic shift of b (or, equivalently, of a cyclic shift of a and a cyclic shift of b). Cyclic string comparison has been considered in [30, 33, 42]. Papers [30, 42] give algorithms that solve the cyclic LCS problem in worst-case time $O(mn)$. By running Algorithm 2 on strings a and bb (a concatenation of string b with itself), and then performing n string-substring LCS queries for a against every substring of bb of length n, the cyclic LCS problem can now be solved in time $O(\frac{mn \cdot \log\log n}{\log n})$.

4.3. Longest repeated subsequence

Given a string a of length n, the *longest repeated subsequence* problem consists in computing the length of the longest subsequence of a that is a concatenation of two identical strings. This problem has been considered under the name "longest tandem scattered subsequence problem" in [29], where an algorithm running in time $O(n^2)$ was given. By running Algorithm 2 on string a against itself, and then performing $n-1$ prefix-suffix LCS queries for every possible non-trivial prefix-suffix decomposition of a, the longest repeated subsequence problem can now be solved in time $O(\frac{n^2 \cdot \log\log n}{\log n})$.

4.4. Incremental LCS

The problem of *incremental LCS* was introduced in [27, 30]. It consists in computing the LCS for a fixed text string against a pattern string of variable length, which can be modified on-line by either appending or prepending a character. An extension of this problem, called *fully-incremental LCS*, was introduced in [23]. In this extension, both input strings can be modified on-line in a similar fashion. In both versions of the problem, the goal is to maintain a data structure that will store the LCS score for the input strings, and will allow efficient on-line updates of this score's value.

We denote by a, b the current state of each input string, and by m, n its respective current size. Papers [27,30] give incremental LCS algorithms with worst-case update time $O(m)$ (the length of the fixed string). Paper [23] extends this result to a fully-incremental LCS algorithm with worst-case update time $O(m)$ (respectively, $O(n)$) when input string a (respectively, b) is kept fixed.

The above algorithms can be matched by a straightforward generalisation of Algorithm 1. Intuitively, our dynamic data structure consists of the endpoints of all the seaweeds in the current state of the alignment dag. Prepending or appending

a character to string a (respectively, b) corresponds to adding a new row of cells along the top or bottom (respectively, left or right) boundary of the dag's core.

We can further generalise fully-incremental string comparison by considering block updates instead of single-character updates. In this new version of the problem, that we call *block-incremental string comparison*, both input strings can be extended on-line by either appending or prepending a block of characters from a pre-specified set of *admissible blocks*. The set of admissible blocks is known in advance, and off-line pre-processing of this set is allowed.

Consider an individual block update, and let l be the corresponding block length. Such an update can be done naively as l single-character updates, giving the block update time $O(ml)$ (respectively, $O(nl)$) when input string a (respectively, b) is kept fixed. The method of highest-score matrix multiplication described in Subsection 3.2 provides the following improvement. The set of admissible blocks is preprocessed off-line by computing the implicit highest-score matrix for each input string against every admissible block. The preprocessing runs in time $O((m+n)\cdot L)$, where L is the total length of the admissible blocks. Given the implicit highest-score matrices, an individual block update can be performed by Lemma 6 in time $O(ml^{0.5})$ (respectively, $O(nl^{0.5})$) when input string a (respectively, b) is kept fixed.

The same techniques generalise further to the case where the dynamic data structure is required to support, in addition to the global LCS, also semi-local LCS queries. By Theorem 3, a data structure allowing efficient semi-local LCS score queries can be maintained at an extra time $O((m+n)\log(m+n))$ per update, under both single-character and block updates.

4.5. Common-substring LCS

The paradigm of *common-substring comparison* was introduced by Landau and others in [14, 31]. We compare a text string of length n against an unspecified number of pattern strings. The pattern strings may share a common substring of length l; we assume $l \leq n$. The location(s) of the common substring in each of the patterns is (are) known in advance. The goal is, given the text, to preprocess the common substring so as to minimise the LCS computation time for each occurrence of the common substring in the patterns. The remainder of the algorithm's running time may only depend on the text, and on the parts of the patterns outside the occurrences of the common substring.

The problem can be solved naively by computing the LCS length for the text against each of the patterns, ignoring the common-substring structure. The resulting algorithm does no preprocessing, and runs in time $O(nl)$ for each occurrence of the common substring.

An improved algorithm is given in [14, 31]. This algorithm, following some preprocessing in time $O(nl)$, runs in time $O(n)$ for each occurrence of the common substring.

Once again, here we consider a more general problem of *semi-local common-substring comparison*. As in ordinary semi-local string comparison, string-substring, substring-string, prefix-suffix and suffix-prefix LCS length queries are allowed between the text and each of the patterns.

Similarly to the global version of the problem, the semi-local version can be solved naively by computing the implicit highest-score matrix for the text against each of the patterns, ignoring the common-substring structure. The resulting algorithm does no preprocessing, and runs in time $O(nl)$ for each occurrence of the common substring.

The method described in Subsection 3.2 provides the following algorithm for semi-local common-substring comparison, which is in fact a special case of the block-incremental semi-local LCS algorithm from the previous subsection. We preprocess the common substring is in time $O(nl)$, obtaining the implicit highest-score matrix for the text against the common substring. For every pattern string, the implicit highest-score matrix can now be built incrementally, starting from an arbitrary occurrence of the common substring. Each incremental update takes time $O(nl^{0.5})$ per occurrence of the common substring in the pattern, and $O(n)$ per pattern character outside any such occurrence. Overall, the algorithm takes time $O(n)$ for the first occurrence of the common substring in a pattern, and time $O(nl^{0.5})$ for each subsequent occurrence in the same pattern. In particular, if the common substring only occurs in every pattern string once, our algorithm improves on the algorithm of [14, 31] in functionality, without any increase in the asymptotic running time.

4.6. Semi-local LCS: the divide-and-conquer algorithm

For completeness, we now describe a semi-local LCS algorithm from [48], based on the divide-and-conquer approach. While the algorithm is asymptotically slower than Algorithm 2, it provides a useful pattern for designing algorithms for various other substring- and subsequence-related problems.

As before, we assume without loss of generality that $m \leq n$. For simplicity, we also assume that m and n are close: $n/2 \leq m \leq n$. First, we give a simple algorithm, based on the fast highest-score matrix multiplication procedure of Lemma 4, and running in overall time $O(mn)$, matching the running time of Algorithm 1.

Algorithm 3 (All semi-local LCS, simple divide-and-conquer).
Input: strings a, b of length m, n, respectively; we assume $n/2 \leq m \leq n$.
Output: implicit extended highest-score matrix on strings a, b.
Description. The computation proceeds recursively, partitioning the longer of the two current strings into a concatenation of two strings of equal length (within ± 1 if string length is odd). Given a current partitioning, the corresponding implicit highest-score matrices are multiplied by Lemma 4. Note that we now have two nested recursions: the main recursion of the current algorithm, and the inner recursion of the algorithm underlying Lemmas 2–4.

In the process of main recursion, the algorithm alternates between partitioning string a and string b. Therefore, we will need to convert the implicit representation of a horizontal highest-score matrix into a vertical one, and vice versa. This can be easily achieved by Theorem 5.

The base of the main recursion is $m = n = 1$.

Cost analysis. Consider the main recursion tree. The computation work in the tree is dominated by the bottom level, which consists of $O(mn)$ instances of implicit highest-score matrix multiplication of size $O(1)$. Therefore, the total computation cost is $O(mn)$.

The main recursion tree can be evaluated depth-first, so that the overall memory cost is dominated by the top level of the main recursion, running in memory $O(n)$. □

The above algorithm can now be easily modified to achieve subquadratic running time. As Algorithm 2, the new algorithm is based on precomputing an exhaustive small-block lookup table.

Algorithm 4 (All semi-local LCS, divide-and-conquer with block precomputation).
Input, output: as in Algorithm 3.

Description. Consider an all semi-local LCS problem on strings of size $t = \frac{1}{2} \cdot \log_\sigma m$, where σ is the size of the alphabet. All possible instances of this problem are precomputed by Algorithm 1 (or by the algorithm of [5]). After that, the computation proceeds as in Algorithm 1. However, the main recursion is cut off at the level where block size reaches t, and the precomputed values are used as the recursion base.

Cost analysis. In the precomputation stage, there are σ^{2t} problem instances, each of which costs $O(t^2)$. Therefore, the total cost of the precomputation is $\sigma^{2t} \cdot O(t^2) = \frac{1}{4} \cdot m(\log_\sigma m)^2 = O(\frac{mn}{\log^{1/2} m})$.

Consider the main recursion tree. The computation work in the tree is dominated by the cut-off level, which consists of $O(mn/t^2)$ instances of implicit highest-score matrix multiplication of size $O(t)$. Therefore, the total computation cost is $mn/t^2 \cdot O(t^{1.5}) = O(\frac{mn}{t^{1/2}}) = O(\frac{mn}{\log^{1/2} m})$. □

4.7. Semi-local LCS in permutations

In [49], we considered a restriction of the all semi-local LCS problem to strings that are permutations of a given set of size n. The resulting problem is equivalent to finding all-substring longest increasing subsequences (LIS) in a permutation. The same problem was previously considered in [2], where an algorithm running in time $O(n^2 \log n)$ is given. Our algorithm from [49] is based on recursive application of implicit highest-score matrix multiplication, and runs in time $O(n^{1.5})$.

A related problem of computing the *complete* LIS in every substring of a *fixed size* is studied in [4, 12]. In particular, paper [12] gives an algorithm that runs in time proportional to the size of the output (i.e., the combined lengths of all the output subsequences), which can be as high as $\Theta(n^2)$. In contrast, our algorithm

from [49] only computes the lengths instead of the complete LIS; however, this is done for substrings of *every possible size*.

4.8. Cyclic LCS in permutations

The cyclic LCS problem on permutation strings is equivalent to the LIS problem on a circular string. This problem is considered in paper [3], which gives a Monte Carlo algorithm running in time $O(n^{1.5} \log n)$ with small error probability. Without loss of generality, we consider the computation of the LCS score for all rotations of an input permutation string a against the identity permutation. The cyclic LCS problem on permutations can now be solved as follows

- run our algorithm from [49] on the identity permutation string against string a;
- run the algorithm of Lemma 2, obtaining the implicit highest-score matrix for the identity permutation string against string aa (the concatenation of a with itself);
- perform n string-substring LCS queries.

The above computation runs in deterministic time $O(n^{1.5})$.

4.9. Maximum cliques in circle graphs

A *circle graph* [16, 20] is defined as the intersection graph of a set of chords in a circle, i.e., the graph where nodes correspond to the chords, and two nodes are adjacent iff the corresponding chords intersect. It has long been known that the maximum clique problem on a circle graph on n nodes is solvable in polynomial time [18]. The best previous algorithms for this problem [6, 22, 36, 40] run in time $O(n^2)$ when the input graph is dense. In [49], we show that by running our semi-local LCS algorithm on a pair of permutations defined by the input set of chords, and then performing n prefix-suffix LCS queries on these permutations, the problem of finding a maximum clique in a circle graph can be solved in time $O(n^{1.5})$.

4.10. Window-local LCS

So far, we have only considered global and semi-local string comparison. We now give an approach to local string comparison – the variant that is the most important biologically, but also the most difficult. We consider a version of local string comparison that is restricted to fixed-length substrings in either one, or both input strings. Given a fixed parameter w, we call a substring of length w *a window* in the corresponding string.

String comparison in windows is one of the most traditional forms of local string comparison. Probably the earliest such paradigm is *dot plots* (also known as *dot matrices*), introduced in [34]. In addition to numerical data, dot plots provide a convenient visualisation of string comparison. In the context of dot plots, comparing strings in windows is usually referred to as *filtering*. The standard filtering method compares every window of a against every window of b in terms of their *Hamming score*, i.e., the count of matching characters along the main diagonal of

the windows' Cartesian product. A Hamming-filtered dot plot can be computed in time $O(mn)$ by the algorithm of [34,37]. This algorithm is implemented in various software packages (see, e.g., [13,39,44]).

Numerous other paradigms of local string comparison have been proposed. The *Smith–Waterman algorithm* [43] allows one to obtain the highest-scoring pair across all pairs of the inputs' substrings, or, more generally, all substring pairs scoring above a certain threshold. The threshold is dependent on the scoring scheme: for example, for the LCS score, the algorithm only provides the trivial global comparison, so the method is generally only useful for edit scores with relatively high costs of character insertion and deletion, and relatively low cost of substitution. A significant drawback of the Smith–Waterman algorithm is that it generally favours long, less precise substring alignments over short, more precise ones (see, e.g., [8]). In contrast, by providing all the local scores between windows of the input strings, the dot plot method gives the user more flexibility to select the biologically significant substring alignments. However, the Hamming scoring scheme used by the method is less sensitive than the LCS score or the general edit score used by Smith–Waterman.

Our approach can be seen as a refinement of the dot plot method and a complement to the Smith–Waterman method. As in the dot plot method, we compute all window-window comparison scores between the input strings. However, instead of the Hamming score, our method is based on the LCS score. Therefore, it is potentially more sensitive, since it compares the window substrings in terms of insertions, deletions and substitutions, instead of just substitutions. The method can be further extended to use more general edit scoring schemes.

We describe the new method in terms of computing the LCS lengths on substrings of a and b as follows:

- the *window-window* (respectively, *window-substring*) *LCS problem*: every window of a against every window (respectively, substring) of b.

In the rest of this subsection, we argue that these problems can be solved in time $O(mn)$, matching the asymptotic running time of both the Hamming-scored dot plot and the Smith–Waterman methods.

Note that the solution of the window-substring LCS problem can be represented implicitly in space $O(mn)$ by giving the highest-score matrix for each window of a against b. An individual window-substring LCS length query can be performed on this data structure in time $O(\log^2 n)$ (or even $O(\frac{\log n}{\log\log n})$ with a higher multiplicative constant). The same data structure can be used to obtain the explicit solution of the window-window LCS problem in time $O(mn)$. Thus, we will treat both problems simultaneously.

Algorithm 5 (Window-window and window-substring LCS).
Input: strings a, b of length m, n, respectively; window length w.
Output: implicit highest-score matrix for every window of a against full b.
Description. For simplicity, we assume that m is a power of 2. We call an interval of the form $[k \cdot 2^s : (k+1) \cdot 2^s]$, $k, s \in \mathbb{Z}$, as well as the corresponding substring

of a, *canonical*. In particular, all individual characters of a are canonical substrings. Every substring of a can be decomposed into a concatenation of $O(\log m)$ canonical substrings.

In the following, by processing an interval we mean computing the implicit highest-score matrix for the corresponding substring of a against b.

First phase. Canonical intervals are processed in a balanced binary tree, in order of increasing length. Every interval of length $2^0 = 1$ is canonical, and is processed by a simple scan of string b. Every canonical interval of length 2^{s+1} is processed as a concatenation of two already processed half-sized canonical intervals of length 2^s.

Second phase. We represent each window $[i, j]$, $j - i = w$, by an odd half-integer *prescribed point* $(i, j) \in \langle 0 : m\rangle^2$. We then proceed by partitioning the square range $\langle 0 : m\rangle^2$ recursively into regular half-sized square blocks.

Consider an $h \times h$ block $\langle i_0 - h : i_0\rangle \times \langle j_0 : j_0 + h\rangle$. The computation is organised so that when a recursive call is made on this block, either we have $i_0 \geq j_0$, or the interval $[i_0 : j_0]$ is already processed.

We can establish in time $O(1)$ the number of prescribed points contained in the current block. If this number is zero, no further computation on the block or recursive partitioning is performed. Otherwise, we have $j - i \in \{-h, 0, h, 2h, \ldots\}$. If $j - i = -h$, then the intervals $[i_0 - h : j_0]$, $[i_0 : j_0 + h]$ have length 0, and the interval $[i_0 - h : j_0 + h]$ is canonical. If $j - i \geq 0$, we process the intervals $[i_0 - h : j_0]$, $[i_0 : j_0 + h]$, $[i_0 - h : j_0 + h]$. Each of these intervals can be processed by Lemma 6, appending and/or prepending a canonical interval of length h to the already processed interval $[i_0 : j_0]$. We then perform further partitioning of the block, and call the procedure recursively on each of the four subblocks.

The base of the recursion is $h = 1$. At this point, we process all 1×1 blocks containing a prescribed point, which is equivalent to processing the original windows. The computation is completed.

Cost analysis. In both the first and the second phase, the computation is dominated by the cost of the bottom level of the computation tree, equal to $O(m) \cdot O(n) = O(mn)$.

Storing the implicit highest-score matrices for all $O(m)$ canonical intervals requires memory $O(mn)$. However, not all these matrices need to be stored simultaneously. By processing and consuming the canonical intervals incrementally within a sliding window of length w, the memory cost can be reduced to $O(wn)$. □

Note that the asymptotic running time of the algorithm is independent of window length w.

By a constant-factor blow-up of the alignment dag, our algorithm can be extended from the LCS score to the more general edit score, where the insertion, deletion and substitution costs are any constant rationals.

4.11. Sparse spliced alignment

Assembling a gene from candidate exons is an important problem in computational biology. Several alternative approaches to this problem have been developed over

time. One of the most successful approaches is *spliced alignment* by Gelfand et al. [19] (see also [21]), which scores different candidate exon chains within a DNA sequence by comparing them to a known related gene sequence. In this method, the two sequences are modelled respectively by strings a, b of lengths m, n; we assume that $m = \Theta(n)$. A subset of substrings in string a are marked as candidate exons. The comparison between sequences is made by string alignment. The algorithm for spliced alignment given in [19] runs in time $O(n^3)$.

In general, the number of candidate exons k may be as high as $O(n^2)$. The method of *sparse spliced alignment* makes a realistic assumption that, prior to the assembly, the set of candidate exons undergoes some filtering, after which only a small fraction of candidate exons remains. Kent et al. [26] give an algorithm for sparse spliced alignment that, in the special case $k = O(n)$, runs in time $O(n^{2.5})$. For higher values of k, the algorithm provides a smooth transition in running time to the dense case $k = O(n^2)$, where its running time $O(n^3)$ is asymptotically equal to the algorithm of [19].

In [45], we improve on the results of [26], by proposing an algorithm for sparse spliced alignment that, in the case $k = O(n)$, runs in time $O(n^{2.25})$, and also provides a smooth transition in running time to the dense case. Our approach is based on a new method of *quasi-local string comparison*, generalising the window-substring LCS problem of Subsection 4.10. Another ingredient of our algorithm is a fast procedure for *highest-score matrix-vector multiplication*, which complements the highest-score matrix multiplication algorithm of Lemma 6.

4.12. Compressed subsequence recognition

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 [11,32] for references.

One of the most general string compression methods is compression by straight-line programs (SLP) [41]. In particular, SLP compression captures the well-known LZ and LZW algorithms [51,53,54]. Various pattern matching problems on SLP-compressed strings have been studied; see, e.g., [11] for references. Cégielski et al. [11] considered subsequence recognition problems on SLP-compressed strings. For an SLP-compressed text of length $\bar{m}$, and an uncompressed pattern of length n, they gave several algorithms for global and local subsequence recognition, running in time $O(\bar{m}n^2 \log n)$.

In [46], we improve on the results of [11] as follows. First, we describe a simple folklore algorithm for global subsequence recognition on an SLP-compressed text, running in time $O(\bar{m}n)$. Then, we consider the more general semi-compressed LCS problem, which consists in computing implicitly the LCS between the compressed text and every substring of the uncompressed pattern. This problem is still a special case of the string-to-CFL edit distance problem; the same problem with a compressed pattern is known to be NP-hard. For the semi-compressed LCS problem, we propose a new algorithm, running in time $O(\bar{m}n^{1.5})$. Our algorithm

is based on the partial highest-score matrix multiplication technique presented in Subsection 3.3. We then extend this method to the several versions of local subsequence recognition considered in [11], for each obtaining an algorithm running in the same asymptotic time $O(\bar{m}n^{1.5})$.

5. Conclusions

We have surveyed a number of existing and new algorithmic techniques and applications related to semi-local string comparison. Our approach unifies a substantial number of previously unrelated problems and techniques, and in many cases allows us to match or improve existing algorithms. It is likely that further development of this approach will give it even more scope and power.

A number of questions related to the semi-local string comparison framework remain open. In particular, it is not yet clear whether the framework can be extended to arbitrary real costs, or to sequence alignment with non-linear gap penalties.

In summary, semi-local string comparison turns out to be a useful algorithmic plug-in, which unifies, and often improves on, a number of previous approaches to various substring- and subsequence-related problems.

Acknowledgement

This paper was conceived in a discussion with Gad Landau. The author thanks an anonymous referee, whose comments helped to improve the paper.

References

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

[2] M. H. Albert, R. E. L. Aldred, M. D. Atkinson, H. P. van Ditmarsch, B. D. Handley, C. C. Handley and J. Opartny. Longest subsequences in permutations. *Australasian Journal of Combinatorics*, 28:225–238, 2003.

[3] M. H. Albert, M. D. Atkinson, D. Nussbaum, J.-R. Sack and N. Santoro. On the longest increasing subsequence of a circular list. *Information Processing Letters*, 101:55–59, 2007.

[4] M. H. Albert, A. Golynski, A. M. Hamel, A. López-Ortiz, S. S. Rao and M. A. Safari. Longest increasing subsequences in sliding windows. *Theoretical Computer Science*, 321:405–414, 2004.

[5] C. E. R. Alves, E. N. Cáceres and S. W. Song. An all-substrings common subsequence algorithm. *Electronic Notes in Discrete Mathematics*, 19:133–139, 2005.

[6] A. Apostolico, M. J. Atallah and S. E. Hambrusch. New clique and independent set algorithms for circle graphs. *Discrete Applied Mathematics*, 36:1–24, 1992.

[7] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod and I. A. Faradzev. On economical construction of the transitive closure of an oriented graph. *Soviet Mathematical Doklady*, 11:1209–1210, 1970.

[8] A. N. Arslan, Ö. Eğecioğlu and P. A. Pevzner. A new approach to sequence comparison: Normalized sequence alignment. *Bioinformatics*, 17(4):327–337, 2001.

[9] J. L. Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, 1980.

[10] R. E. Burkard, B. Klinz and R. Rudolf. Perspectives of Monge properties in optimization. *Discrete Applied Mathematics*, 70(2):95–161, 1996.

[11] P. Cégielski, I. Guessarian, Y. Lifshits and Y. Matiyasevich. Window subsequence problems for compressed texts. In *Proceedings of CSR*, volume 3967 of *Lecture Notes in Computer Science*, pages 127–136, 2006.

[12] E. Chen, L. Yang and H. Yuan. Longest increasing subsequences in windows based on canonical antichain partition. *Theoretical Computer Science*, 378(3):223–236, 2007.

[13] CLC bio. *CLC Protein Workbench: User Manual*, 2006.

[14] M. Crochemore, G. M. Landau, B. Schieber and M. Ziv-Ukelson. Re-use dynamic programming for sequence alignment: An algorithmic toolkit. In *String Algorithmics*, volume 2 of *Texts in Algorithmics*. King's College Publications, 2004.

[15] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

[16] S. Even and A. Itai. Queues, stacks and graphs. In *Theory of Machines and Computations*, pages 71–86. Academic Press, 1971.

[17] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *Proceedings of the 19th ACM ICS*, 2005.

[18] F. Gavril. Algorithms for a maximum clique and a maximum independent set of a circle graph. *Networks*, 3:261–273, 1973.

[19] M. S. Gelfand, A. A. Mironov and P. A. Pevzner. Gene recognition via spliced sequence alignment. *Proceedings of the National Academy of Sciences of the USA*, 93(17):9061–9066, 1996.

[20] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Number 57 in Annals of Discrete Mathematics. Elsevier, second edition, 2004.

[21] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[22] W.-L. Hsu. Maximum weight clique algorithms for circular-arc graphs and circle graphs. *SIAM Journal on Computing*, 14(1):224–231, 1985.

[23] Y. Ishida, S. Inenaga, A. Shinohara and M. Takeda. Fully incremental LCS computation. In *Proceedings of FCT*, volume 3623 of *Lecture Notes in Computer Science*, pages 563–574, 2005.

[24] J. JáJá, C. Mortensen and Q. Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In *Proceedings of the* 15*th ISAAC*, volume 3341 of *Lecture Notes in Computer Science*, pages 558–568, 2004.

[25] N. C. Jones and P. A. Pevzner. *An introduction to bioinformatics algorithms*. Computational Molecular Biology. The MIT Press, 2004.

[26] C. Kent, G. M. Landau and M. Ziv-Ukelson. On the complexity of sparse exon assembly. *Journal of Computational Biology*, 13(5):1013–1027, 2006.

[27] S.-R. Kim and K. Park. A dynamic edit distance table. *Journal of Discrete Algorithms*, 2(2):303–312, 2004.

[28] P. N. Klein. Multiple-source shortest paths in planar graphs. In *Proceedings of the 16th ACM-SIAM SODA*, pages 146–155, 2005.

[29] A. Kosowski. An efficient algorithm for the longest tandem scattered subsequence problem. In *Proceedings of the 11th SPIRE*, volume 3246 of *Lecture Notes in Computer Science*, pages 93–100, 2004.

[30] G. M. Landau, E. W. Myers and J. P. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582, 1998.

[31] G. M. Landau and M. Ziv-Ukelson. On the common substring alignment problem. *Journal of Algorithms*, 41(2):338–359, 2001.

[32] Y. Lifshits and M. Lohrey. Querying and embedding compressed texts. In *Proceedings of MFCS*, volume 4162 of *Lecture Notes in Computer Science*, pages 681–692, 2006.

[33] M. Maes. On a cyclic string-to-string correction problem. *Information Processing Letters*, 35:73–78, 1990.

[34] J. V. Maizel and R. P. Lenk. Enhanced graphic matrix analysis of nucleic acid and protein sequences. *Proceedings of the National Academy of Sciences of the USA*, 78(12):7665–7669, 1981.

[35] W. J. Masek and M. S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20:18–31, 1980.

[36] S. Masuda, K. Nakajima, T. Kashiwabara and T. Fujisawa. Efficient algorithms for finding maximum cliques of an overlap graph. *Networks*, 20:157–171, 1990.

[37] C. Mueller, M. M. Dalkilic and A. Lumsdaine. High-performance direct pairwise comparison of large genomic sequences. *IEEE Transactions on Parallel and Distributed Systems*, 17(8):764–772, 2006.

[38] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Texts and Monographs in Computer Science. Springer, 1985.

[39] P. Rice, I. Longden and A. Bleasby. EMBOSS: The European molecular biology open software suite. *Trends in Genetics*, 16(6):276–277, 2000.

[40] D. Rotem and J. Urrutia. Finding maximum cliques in circle graphs. *Networks*, 11:269–278, 1981.

[41] W. Rytter. Application of Lempel–Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1–3):211–222, 2003.

[42] J. P. Schmidt. All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM Journal on Computing*, 27(4):972–992, 1998.

[43] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[44] E. L. L. Sonnhammer and R. Durbin. A dot-matrix program with dynamic threshold control suited for genomic DNA and protein sequence analysis. *Gene*, 167(1–2):1–10, 1996.

[45] A. Tiskin. Faster exon assembly by sparse spliced alignment. Technical Report 0707.3409, arXiv.

[46] A. Tiskin. Faster subsequence recognition in compressed strings. *Journal of Mathematical Sciences*. To appear.

[47] A. Tiskin. Longest common subsequences in permutations and maximum cliques in circle graphs. Submitted, available from `http://www.dcs.warwick.ac.uk/~tiskin/pub`; a revised version of [49].

[48] A. Tiskin. Semi-local longest common subsequences in subquadratic time. *Journal of Discrete Algorithms*. To appear.

[49] A. Tiskin. Longest common subsequences in permutations and maximum cliques in circle graphs. In *Proceedings of CPM*, volume 4009 of *Lecture Notes in Computer Science*, pages 271–282, 2006.

[50] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

[51] T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.

[52] S. Wu, U. Manber and G. Myers. A subquadratic algorithm for approximate limited expression matching. *Algorithmica*, 15:50–67, 1996.

[53] G. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.

[54] G. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24:530–536, 1978.

Alexander Tiskin
Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom
e-mail: `tiskin@dcs.warwick.ac.uk`

Received: March 12, 2007.

Revised: October 8, 2007.

Accepted: November 5, 2007.