Journal of Discrete Algorithms 6 (2008) 570–581

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Semi-local longest common subsequences in subquadratic time

Alexander Tiskin

Department of Computer Science, University of Warwick, Coventry CV4 7AL, United Kingdom

ARTICLE INFO

Article history:
Available online 20 August 2008

Keywords:
String algorithms
Longest common subsequence
Semi-local string comparison

ABSTRACT

For two strings a, b of lengths m, n, respectively, the longest common subsequence (LCS) problem consists in comparing a and b by computing the length of their LCS. In this paper, we define a generalisation, called "the all semi-local LCS problem", where each string is compared against all substrings of the other string, and all prefixes of each string are compared against all suffixes of the other string. An explicit representation of the output lengths is of size $\Theta((m+n)^2)$. We show that the output can be represented implicitly by a geometric data structure of size $O(m+n)$, allowing efficient queries of the individual output lengths. The currently best all string–substring LCS algorithm by Alves et al., based on previous work by Schmidt, can be adapted to produce the output in this form. We also develop the first all semi-local LCS algorithm, running in time $o(mn)$ when m and n are reasonably close. Compared to a number of previous results, our approach presents an improvement in algorithm functionality, output representation efficiency, and/or running time.

1. Introduction

Given two strings a, b of lengths m, n, respectively, the longest common subsequence (LCS) problem consists in comparing a and b by computing the length of their LCS. In this paper, we define a generalisation, called "the all semi-local LCS problem", where each string is compared against all substrings of the other string, and all prefixes of each string are compared against all suffixes of the other string. The all semi-local LCS problem arises naturally in the context of LCS computations on substrings. It is closely related to local sequence alignment (see e.g. [10,12]) and to approximate string matching (see e.g. [9,18]).

A standard approach to string comparison is representing the problem as an alignment dag (directed acyclic graph) of size $\Theta(mn)$ on an $m \times n$ grid of nodes. The basic LCS problem, as well as its many generalisations, can be solved by dynamic programming on this dag in time $O(mn)$ (see e.g. [9,10,12,18]). It is well known (see e.g. [2,16] and references therein) that all essential information in the alignment dag can in fact be represented by a data structure of size $O(m+n)$. In this paper, we expose a rather surprising (and to the best of our knowledge, previously unnoticed) connection between this linear-size representation of the string comparison dag, and a standard computational geometry problem known as dominance counting.

If the output lengths of the all semi-local LCS problem are represented explicitly, the total size of the output is $\Theta((m+n)^2)$, corresponding to m^2+n^2 possible substrings and $2mn$ possible prefix–suffix pairs. To reduce the storage requirements, we allow the output lengths to be represented implicitly by a smaller data structure that allows efficient retrieval of individual output values. Using previously known linear-size representations of the string comparison dag, retrieval of an individual output length typically requires scanning of at least a constant fraction of the representing data structure, and therefore takes time $O(m+n)$. By exploiting the geometry connection, we show that the output lengths can be rep-

E-mail address: tiskin@dcs.warwick.ac.uk.

1570-8667/$ – see front matter
doi:10.1016/j.jda.2008.07.001

resented by a set of $m+n$ grid points. Individual output lengths can be obtained from this representation by dominance counting queries. This leads to a data structure of size $O(m+n)$, that allows to query an individual output length in time $O(\frac{\log(m+n)}{\log\log(m+n)})$, using a recent result by JáJá et al. [11]. The described approach presents a substantial improvement in query efficiency over previous approaches.

It has long been known [8,17] that the (global) LCS problem can be solved in subquadratic[1] time $O(\frac{mn}{\log(m+n)})$ when m and n are reasonably close. Alves et al. [2], based on previous work by Schmidt [20], proposed an all string–substring (i.e. restricted semi-local) LCS algorithm that runs in time $O(mn)$. In this paper, we propose the first all semi-local LCS algorithm, which runs in subquadratic time $O(\frac{mn}{\log^{0.5}(m+n)})$ when m and n are reasonably close. This improves on [2] simultaneously in algorithm functionality, output representation efficiency, and running time.

A preliminary version of this paper appeared as [21].

2. Previous work

Although our generic definition of the all semi-local LCS problem is new, several algorithms dealing with similar problems involving multiple substring comparison have been proposed before. The standard dynamic programming approach can be regarded as comparing all prefixes of each string against all prefixes of the other string. Papers [2,7,13–16,20] present several variations on the theme of comparing substrings (prefixes, suffixes) of two strings. In [13,15], the two input strings are revealed character by character. Every new character can be either appended or prepended to the input string. Therefore, the computation is performed essentially on substrings of subsequent inputs. In [16], multiple strings sharing a common substring are compared against a common target string. A common feature in many of these algorithms is the use of linear-sized string comparison dag representation, and a suitable merging procedure that "stitches together" the representations of neighbouring dag blocks to obtain a representation for the blocks' union. As a consequence, such algorithms could be adapted to work with our new, potentially more efficient geometric representation, without any increase in asymptotic time or memory requirements.

3. 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 both of a and b.

We define a generalisation of the LCS problem, which we call the *all semi-local LCS problem*. It consists in computing the LCS lengths 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 by considering this combination of problems rather than each problem separately, the algorithms can be greatly simplified.

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*[2] 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

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

To denote infinite intervals of integers and odd half-integers, we will use $-\infty$ for i and $+\infty$ for j where appropriate.

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*

[1] The term "subquadratic" throughout this paper refers to the case $m=n$.

[2] It would be possible to reformulate all our results using only integers. However, using odd half-integers helps to make the exposition simpler and more elegant.

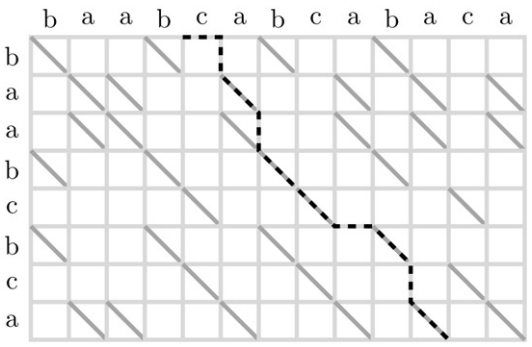

Fig. 1. An alignment dag and a highest-scoring path.

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 both to finite and infinite matrices.

A finite permutation matrix can be represented by its nonzeros' index set. 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' index set.

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

$$d_A(i_0,j_0)=\sum D_A(i,j), \quad i\in\langle i_0:n\rangle,\ \ j\in\langle 0:j_0\rangle, \tag{1}$$

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

$$D_A(i,j)=d_A\big(i-\tfrac{1}{2},j+\tfrac{1}{2}\big)-d_A\big(i-\tfrac{1}{2},j-\tfrac{1}{2}\big)-d_A\big(i+\tfrac{1}{2},j+\tfrac{1}{2}\big)+d_A\big(i+\tfrac{1}{2},j-\tfrac{1}{2}\big).$$

When matrix d_A is a distribution matrix of D_A, matrix D_A is called the *density matrix* of d_A. The definitions of distribution and density matrices extend naturally to infinite matrices. We will only deal with distribution matrices where the sum (1) is always finite.

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

4. Alignment dags and highest-score matrices

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. To describe our algorithms, we need a slightly extended version of this representation, where the finite grid of nodes is embedded in an infinite grid.

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

- horizontal edge $v_{i,j-1}\to v_{i,j}$ and vertical edge $v_{i-1,j}\to v_{i,j}$ are both always present in G and have score 0;
- diagonal edge $v_{i-1,j-1}\to v_{i,j}$ 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 $v_{i-1,j-1}\to v_{i,j}$ is present, iff $\alpha_i=\beta_j$. Fig. 1 shows the alignment dag corresponding to strings $a=$ "baabcbca", $b=$ "baabcabcabaca" (an example borrowed from [2]).

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

$$v_{0,j}\rightsquigarrow v_{m,j'},\quad v_{i,0}\rightsquigarrow v_{m,j'},\quad v_{0,j}\rightsquigarrow v_{i',n},\quad v_{i,0}\rightsquigarrow v_{i',n}, \tag{2}$$

where $i,i'\in[0:m]$, $j,j'\in[0:n]$. 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 (2) between every possible pair of endpoints. (Since the graph is acyclic, this is also equivalent to finding the score of the corresponding lowest-scoring path in an alignment dag where all the scores are negated. Thus, the problem is related to classical shortest path problems.)

Definition 2. Given an $m\times n$ alignment dag G, its *extension* G^+ is an infinite weighted dag, defined on the set of nodes $v_{i,j}$, $i,j\in[-\infty:+\infty]$ and containing G as a subgraph. For all $i,j\in[-\infty:+\infty]$:

- horizontal edge $v_{i,j-1} \to v_{i,j}$ and vertical edge $v_{i-1,j} \to v_{i,j}$ are both always present in G^+ and have score 0;
- when $i \in [1:m]$, $j \in [1:n]$, diagonal edge $v_{i-1,j-1} \to v_{i,j}$ is present in G^+ iff it is present in G; if present, it has score 1;
- otherwise, diagonal edge $v_{i-1,j-1} \to v_{i,j}$ 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 (2) 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 3. Given an $m \times n$ alignment dag G, its *extended horizontal* (respectively, *vertical*) *highest-score matrix*[3] is an infinite matrix defined by

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

$$A^*(i,j) = \max score(v_{i,0} \rightsquigarrow v_{j,n}), \quad i,j \in [-\infty:+\infty], \tag{4}$$

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

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

The maximum path scores for each of the four path types (2) can be obtained from the extended horizontal highest-score matrix (3) as follows:

$$\max score(v_{0,j} \rightsquigarrow v_{m,j'}) = A(j,j'),$$

$$\max score(v_{i,0} \rightsquigarrow v_{m,j'}) = A(-i,j') - i,$$

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

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

where $i,i' \in [0:m]$, $j,j' \in [0:n]$, and the maximum is taken across all paths between the given endpoints. The same maximum path scores can be obtained analogously from the extended vertical highest-score matrix (4).

For most of this section, we will concentrate on the properties of extended horizontal highest-score matrices, referring to them simply as "extended highest-score matrices". By symmetry, extended vertical highest-score matrices will have analogous properties. We assume $i,j \in [-\infty:+\infty]$, unless indicated otherwise.

Theorem 4. *An extended highest-score matrix has the following properties*:

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

$$A(i,j) \trianglelefteq A(i,j+1); \tag{6}$$

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

$$\text{if } A(i-1,j) \triangleleft A(i-1,j+1), \quad \text{then } A(i,j) \triangleleft A(i,j+1). \tag{8}$$

Proof. A path $v_{0,i-1} \rightsquigarrow v_{m,j}$ can be obtained by first following a horizontal edge of score 0: $v_{0,i-1} \to v_{0,i} \rightsquigarrow v_{m,j}$. Therefore, $A(i,j) \leqslant A(i-1,j)$. 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, $A(i,j) \geqslant A(i-1,j) - 1$. We thus have (5) and, by symmetry, (6).

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*:

$$A(i,j) + A(i-1,j+1) \leqslant A(i-1,j) + A(i,j+1).$$

Rearranging the terms

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

and applying (5), we obtain (7) and, by symmetry, (8). □

[3] These matrices are called "DIST matrices", e.g., in [7,20], and "score matrices" in [21,22]. 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. [12]).

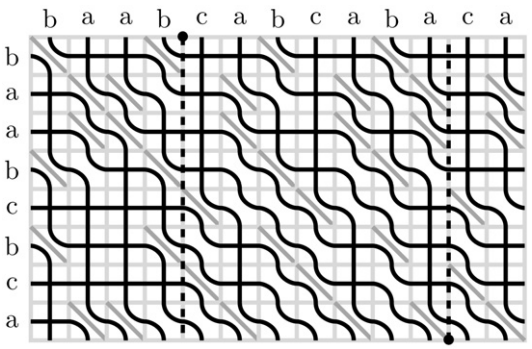

Fig. 2. An alignment dag and the non-trivial critical points.

The properties of Theorem 4 are symmetric with respect to i and $n-j$. Alves et al. [2] introduce the same properties but do not make the most of their symmetry. We aim to exploit symmetry to the full.

Corollary 5. *An extended highest-score matrix has the following properties*:

$$\text{if } A(i,j) \lhd A(i-1,j), \quad \text{then } A(i,j') \lhd A(i-1,j') \text{ for all } j' \leqslant j;$$
$$\text{if } A(i,j) = A(i-1,j), \quad \text{then } A(i,j') = A(i-1,j') \text{ for all } j' \geqslant j;$$
$$\text{if } A(i,j) \lhd A(i,j+1), \quad \text{then } A(i',j) \lhd A(i',j+1) \text{ for all } i' \geqslant i;$$
$$\text{if } A(i,j) = A(i,j+1), \quad \text{then } A(i',j) = A(i',j+1) \text{ for all } i' \leqslant i.$$

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 4. □

Informally, Corollary 5 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 propose the following definition.

Definition 6. An odd half-integer point $(i,j) \in \langle -\infty : +\infty \rangle^2$ is called *A-critical*, if

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

In particular, 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 7. *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 5 and Definition 6. □

Fig. 2 shows the alignment dag of Fig. 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" curve,[4] 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 curves, originating or terminating at the sides of the dag, correspond to critical points (i,j), where either $i \notin \langle 0 : n \rangle$ or $j \notin \langle 0 : n \rangle$.

[4] For the purposes of this illustration, the specific layout of the curves between their endpoints is not important.

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 8. Point (i_0, j_0) *dominates*[5] 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.[6] 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 critical point set defines uniquely a highest-score matrix, and gives a simple formula for recovering matrix elements.

Theorem 9. *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 $i_0 \geqslant j_0$. 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 5.
(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 5.

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

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

There is a close connection between Theorem 9 and the canonical structure of general Monge matrices (see e.g. [6]).

Recall that outside the core, the structure of an extended alignment graph is trivial: all possible diagonal edges are present in the non-core subgraph. This gives rise to the following property.

Corollary 10. *Let $i < -m$ or $j > m + n$. Point (i, j) is A-critical iff $j - i = m$ (and therefore $j < 0$ or $i > n$).*

Proof. By Definitions 2, 6. □

We will call A-critical points described by the above corollary *trivial*. Conversely, we have the following definition.

Definition 11. An A-critical point (i, j) is *non-trivial*, iff $i \in \langle -m, n \rangle$, $j \in \langle 0, m + n \rangle$.

Corollary 12. *There are exactly $m + n$ non-trivial A-critical points.*

Proof. We have $A(-m, m + n) = m$. On the other hand, $A(-m, m + n) = 2m + n - d_A(-m, m + n)$ by Theorem 9. Hence, the total number of non-trivial A-critical points is $d_A(-m, m + n) = m + n$. □

In Fig. 2, the set of critical points represented by the curves is precisely the set of all non-trivial critical points. Note that there are $8 + 13 = 21$ curves in total.

Since only non-trivial critical points need to be represented explicitly, an extended highest-score matrix can be represented by a critical point data structure of size $O(m + n)$. We will refer to this data structure as the *implicit representation* of A.

[5] 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.

[6] 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.

By Theorem 9, the value $A(i_0, j_0)$ is determined by the number of A-critical points 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 critical points. 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.

Theorem 13. *Given the implicit representation of an extended 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 $O(\log(m+n)^2)$.*

Proof. The structure in question is a 2D range tree [5] (see also [19]), built on the set of non-trivial A-critical points. There are $m+n$ such points, hence the total number of nodes in the tree is $O((m+n)\log(m+n))$. A dominance counting query on the set of non-trivial A-critical points can be answered by accessing $O(\log(m+n)^2)$ of the tree nodes. A dominance counting query on the set of trivial A-critical points can be answered by a simple constant-time index calculation (note that the result of such a query can only be nonzero when the query point lies outside the core subgraph of the extended alignment dag). The sum of the above two dominance counting queries provides the total number of A-critical points dominated by the query point (i_0, j_0). The value $A(i_0, j_0)$ can now be obtained by Theorem 9. □

Theorem 14. *Given the implicit representation of an extended 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 [11]. □

While the data structure used in Theorem 14 provides better asymptotics, the range tree used in Theorem 13 is simpler, requires a less powerful computation model, and is more likely to be practical.

We conclude this section by formulating yet another previously unexploited symmetry of the all semi-local LCS problem, which will also become a key ingredient of our all semi-local LCS algorithm. This time, we consider both the horizontal highest-score matrix A as in (3), and the vertical highest-score matrix A^* as in (4). 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 15. *Point (i, j) is A-critical, iff point $(-i, m+n-j)$ is A^*-critical.*

Proof. Straightforward case analysis based on Definition 6. □

5. Fast highest-score matrix multiplication

Our solution of the all semi-local LCS problem and related problems follows a divide-and-conquer approach, which refines the framework for the string–substring LCS problem developed in [2,20]. In this section, we describe the key subroutine, that will constitute the "conquer" step of our algorithms.

Following the divide-and-conquer approach, the alignment dag is partitioned recursively 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 $M \geqslant m$. 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 extended highest-score matrices defined respectively by dags G_1, G_2, G. In every recursive call 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. We show that 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 Section 4.

The implicit representation of matrices A, B, C consists of respectively $M+n$, $m+n$, $M+m+n$ non-trivial nonzeros. Alves et al. [2] 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 $O(mn)$. The algorithm produces a data structure of size $O(n)$, which

can be easily converted into the implicit representation of the output matrix. By adding a post-processing phase based on Theorems 13, 14, this algorithm can be adapted to produce a query-efficient output data structure.

In generalised form, the main technique of [2] can be stated as follows.

Lemma 16. *(Generalised from [2].) Consider the concatenation of alignment dags as described above, with extended 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 $O(M+n)$.*

Our new algorithm is based on a novel highest-score matrix multiplication procedure, which improves on Lemma 16, as long as $n = o(m^2)$. The main subroutine of our algorithm can be regarded as a variant of standard $(\min,+)$ matrix multiplication.

Definition 17. Let $n \in \mathbb{N}$. Let A, B, C be arbitrary numerical matrices with indices ranging over $[0:n]$. The $(\min,+)$-*product* $A \odot B = C$ is defined by

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

It is straightforward to check that $(\min,+)$-multiplication is associative. Moreover, if each of matrices A, B is a permutation–distribution matrix with indices ranging over $[0:n]$, then their $(\min,+)$-product is a permutation–distribution matrix with the same index range. In other words, the set of permutation–distribution matrices forms a submonoid in the multiplicative monoid of the $(\min,+)$-semiring of integer matrices over a given index range.

We now describe an efficient $(\min,+)$-multiplication algorithm for permutation–distribution matrices, which is the key lemma of this paper.

Lemma 18. *Let D_A, D_B, D_C be permutation matrices with indices ranging over $\langle 0:n\rangle$, and let d_A, d_B, d_C be their respective distribution matrices. Let $d_A \odot d_B = d_C$. Given the set of nonzero elements' index pairs in each of D_A, D_B, the set of nonzero elements' index pairs in D_C can be computed in time $O(n^{1.5})$ and memory $O(n)$.*

Proof. For a function f and a predicate P defined on a variable i, notation "$\mathrm{any}_{i:P(i)} f(i)$" will denote the value $f(i)$, where index i is chosen arbitrarily from the set $\{i : P(i)\}$. This is analogous to the use of "$\min_{i:P(i)} f(i)$" to denote the minimum of a function on a given index set.[7]

By Definition 17, we have

$$d_C(i,k) = \min_j\big(d_A(i,j) + d_B(j,k)\big), \quad i,j,k \in [0:n]. \tag{9}$$

We proceed by partitioning the square index pair range $\langle 0:n\rangle^2$ recursively into regular half-sized square blocks. For each block, we establish the number of nonzero elements of D_C contained in that block, and perform further recursive partitioning as long as this number is greater than 0.

Consider an $h \times h$ block

$$\langle i_0 - h : i_0\rangle \times \langle k_0 : k_0 + h\rangle.$$

The nonzero of D_C in this block will be determined by the nonzeros of D_A in $\langle i_0 - h : i_0\rangle \times \langle 0:n\rangle$, and the nonzeros of D_B in $\langle 0:n\rangle \times \langle k_0 : k_0 + h\rangle$. We call such nonzeros of D_A and D_B *relevant*. For the current block, there are exactly h relevant nonzeros in each of D_A, D_B.

For any $j \in [0:n]$, let $\delta_A(j)$ (respectively, $\delta_B(j)$) denote the number of relevant nonzeros of D_A in $\langle i_0 - h : i_0\rangle \times \langle 0 : j\rangle$ (respectively, of D_B in $\langle j : n\rangle \times \langle k_0 : k_0 + h\rangle$):

$$\delta_A(j) = d_A(i_0 - h, j) - d_A(i_0, j), \qquad \delta_B(j) = d_B(j, k_0 + h) - d_B(j, k_0).$$

Sequence δ_A is non-strictly monotonically increasing from $\delta_A(0) = 0$ to $\delta_A(n) = h$. Sequence δ_B is non-strictly monotonically decreasing from $\delta_B(0) = h$ to $\delta_B(n) = 0$.

As the block size h gets smaller, sequences δ_A, δ_B contain fewer and fewer distinct values. We represent these sequences compactly by storing, for every $d \in [-h:h]$, the values

$$\Delta_A(d) = \text{any } \delta_A(j), \qquad \Delta_B(d) = \text{any } \delta_B(j),$$
$$M(d) = \min\big(d_A(i_0, j) + d_B(j, k_0)\big),$$

[7] In fact, "min" (or "max") can always be used instead of "any" on a finite index set; however, such usage could be misleading when "any" happens to be sufficient.

where "any" and "min" are taken across all $j : \delta_A(j) - \delta_B(j) = d$. When the set of such j is empty, the corresponding values $\Delta_A(d)$, $\Delta_B(d)$, $M(d)$ are undefined and omitted from further computations. Sequence Δ_A is non-strictly monotonically increasing from $\Delta_A(-h) = 0$ to $\Delta_A(h) = h$ (ignoring the undefined values). Sequence Δ_B is non-strictly monotonically decreasing from $\Delta_B(-h) = h$ to $\Delta_B(h) = 0$ (again ignoring the undefined values). Sequences Δ_A, Δ_B can be computed in time $O(h)$ by a single scan of the relevant nonzero sets of D_A and D_B.

The computation is organised so that when a recursive call is made on the current block, sequence M is already computed at a higher level of recursion. At the top level, this sequence is computed in time $O(n)$ by a scan of the complete nonzero sets of D_A and D_B (all of which are relevant at the top recursion level). At lower levels of recursion, sequence M is recomputed in time $O(h)$ by a procedure that will be described below. From sequences Δ_A, Δ_B, M, the following values can be found in time $O(h)$:

$$d_C(i_0, k_0) = \min M(d),$$
$$d_C(i_0 - h, k_0) = \min\big(\Delta_A(d) + M(d)\big),$$
$$d_C(i_0, k_0 + h) = \min\big(M(d) + \Delta_B(d)\big),$$
$$d_C(i_0 - h, k_0 + h) = \min\big(\Delta_A(d) + M(d) + \Delta_B(d)\big),$$

where "min" is taken across all $d \in [-h : h]$ for which $\Delta_A(d)$, $\Delta_B(d)$, $M(d)$ are defined. The number of nonzeros of D_C in the current block can then be determined as

$$d_C(i_0 - h, k_0 + h) - d_C(i_0 - h, k_0) - d_C(i_0, k_0 + h) + d_C(i_0, k_0).$$

If the above value is nonzero, the recursion proceeds by partitioning the current block of size h into four subblocks of size $h/2$. The relevant nonzero sets of D_A and D_B are split accordingly, each into two subsets of size $h/2$. Let $i'_0 \in \{i_0, i_0 - \frac{h}{2}\}$, $k'_0 \in \{k_0, k_0 + \frac{h}{2}\}$, and consider each of the four half-sized subblocks $\langle i'_0 - \frac{h}{2} : i'_0\rangle \times \langle k'_0, k'_0 + \frac{h}{2}\rangle$. Let δ'_A, δ'_B, M' denote the sequences defined for the current subblock analogously to sequences δ_A, δ_B, M for the parent block. For every $d \in [-h : h]$, let

$$\Delta^*_A(d) = \text{any } \delta'_A(j), \qquad \Delta^*_B(d) = \text{any } \delta'_B(j),$$

where "any" is taken across all $j : \delta_A(j) - \delta_B(j) = d$. When the set of such j is empty, $\Delta^*_A(d)$, $\Delta^*_B(d)$ are left undefined. Sequence Δ^*_A is non-strictly monotonically increasing from $\Delta^*_A(-h) = 0$ to $\Delta^*_A(h) = h/2$ (ignoring the undefined values). Sequence Δ^*_B is non-strictly monotonically decreasing from $\Delta^*_B(-h) = h/2$ to $\Delta^*_B(h) = 0$ (again ignoring the undefined values). Similarly to Δ_A, Δ_B, sequences Δ^*_A, Δ^*_B can be computed in time $O(h)$ by a single scan of the relevant nonzero sets of D_A and D_B. In each of the four subblocks, values $M'(d')$ for all $d' \in [-\frac{h}{2} : \frac{h}{2}]$ can now be obtained from sequence M by

$$M'(d') = \min M(d), \qquad \text{for } i'_0 = i_0,\ k'_0 = k_0,$$
$$M'(d') = \min\big(\Delta^*_A(d) + M(d)\big), \qquad \text{for } i'_0 = i_0 - \tfrac{h}{2},\ k'_0 = k_0,$$
$$M'(d') = \min\big(M(d) + \Delta^*_B(d)\big), \qquad \text{for } i'_0 = i_0,\ k'_0 = k_0 + \tfrac{h}{2},$$
$$M'(d') = \min\big(\Delta^*_A(d) + M(d) + \Delta^*_B(d)\big), \qquad \text{for } i'_0 = i_0 - \tfrac{h}{2},\ k'_0 = k_0 + \tfrac{h}{2},$$

where "min" is taken across all $d : \Delta^*_A(d) - \Delta^*_B(d) = d'$ for which $\Delta^*_A(d)$, $\Delta^*_B(d)$, $M(d)$ are defined. Note that sequence M' is obtained purely from the sequences δ'_A, δ'_B and M; in particular, the scan of matrices d_A, d_B is not required. For each of the four subblocks, every value $M(d)$ contributes to exactly one value $M'(d')$, therefore the above computation can be done in time $O(h)$.

The base of the recursion is $h = 1$. At this point, we establish all 1×1 blocks of D_C containing a nonzero, which is equivalent to establishing the nonzeros of D_C themselves. The computation is completed.

The recursion tree has maximum degree 4, height $\log n$, and n leaves corresponding to the nonzeros of D_C.

Consider the top-to-middle levels of the recursion tree. In each level from the top down to the middle level, the maximum number of nodes increases by a factor of 4, and the maximum amount of computation work per node decreases by a factor of 2. Hence, the maximum amount of work per level increases in geometric progression, and is dominated by the middle level $\frac{\log n}{2}$.

Consider the middle-to-bottom levels of the recursion tree. Since the tree has n leaves, each level contains at most n nodes. In each level from the middle down to the bottom level, the maximum amount of computation work per node still decreases by a factor of 2. Hence, the maximum amount of work per level decreases in geometric progression, and is again dominated by the middle level $\frac{\log n}{2}$.

Thus, the computation work in the whole recursion tree is dominated by the maximum amount of work in the middle level $\frac{\log n}{2}$. This level has at most n nodes, each requiring at most $O(n)/2^{\frac{\log n}{2}} = O(n^{1/2})$ work. Therefore, the overall computation cost of the recursion is at most $n \cdot O(n^{1/2}) = O(n^{1.5})$.

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 next step is extending the algorithm of Lemma 18 to multiplying infinite permutation–distribution matrices. We consider the special case where both multiplicands have semi-infinite core.

Lemma 19. *Let D_A, D_B, D_C be permutation matrices with indices ranging over $\langle -\infty : +\infty \rangle$, such that*

$$D_A(i,j) = I(i,j), \quad \text{for } i,j \in \langle -\infty : 0 \rangle,$$
$$D_B(j,k) = I(j,k), \quad \text{for } j,k \in \langle n : +\infty \rangle.$$

Let d_A, d_B, d_C be their respective distribution matrices. Let $d_A \odot d_B = d_C$. We have

$$D_A(i,j) = D_C(i,j), \quad \text{for } i \in \langle -\infty : +\infty \rangle,\ j \in \langle n : +\infty \rangle, \tag{10}$$
$$D_B(j,k) = D_C(j,k), \quad \text{for } j \in \langle -\infty : 0 \rangle,\ k \in \langle -\infty : +\infty \rangle. \tag{11}$$

Given the set of all n remaining nonzero elements' index pairs in each of D_A, D_B, i.e. the set of all nonzero elements' index pairs (i,j) in D_A and (j,k) in D_B with $i \in \langle 0 : +\infty \rangle$, $j \in \langle 0 : n \rangle$, $k \in \langle -\infty : 0 \rangle$, the set of all n remaining nonzero elements' index pairs in D_C can be computed in time $O(n^{1.5})$ and memory $O(n)$.

Proof. It is straightforward to check equalities (10)–(11) by (1) and Definition 17. Informally, each nonzero of D_C appearing in (10)–(11) is obtained as a direct combination of a nonzero of D_A and a nonzero of D_B, exactly one of which is trivial. All remaining nonzeros of D_A and D_B are non-trivial, 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 none of the nonzeros of D_A and D_B appearing in (10)–(11) can be dominated by any of the remaining nonzeros of D_A and D_B. Hence, the nonzeros appearing in (10)–(11) cannot affect the computation of the remaining nonzeros of D_C. We can therefore simplify the problem by eliminating all half-integer indices i, j, k that correspond to nonzero index pairs (i,j) and (j,k) appearing in (10)–(11), and then renumbering the remaining indices i, k, so that their new range becomes $\langle 0 : n \rangle$ (which is already the range of j after the elimination). More precisely, we define permutation matrices D'_A, D'_B, D'_C, with indices ranging over $\langle 0 : n \rangle$, as follows. Matrix D'_A is obtained from D_A by selecting all rows i with a nonzero $D_A(i,j)$, $j \in \langle 0 : n \rangle$, and then selecting all columns that contain a nonzero in at least one (in fact, exactly one) of the selected rows. Matrix D'_B is obtained from D_B by selecting all columns k with a nonzero $D_B(j,k)$, $j \in \langle 0 : n \rangle$, and then selecting all rows that contain a nonzero in at least one (in fact, exactly one) of the selected columns. Matrix D'_C is obtained from D_C by selecting all rows and columns with a nonzero $D_C(i,k)$, $i \in \langle 0 : +\infty \rangle$, $k \in \langle -\infty, n \rangle$. We define d'_A, d'_B, d'_C accordingly. The index order is preserved by the above matrix transformation, so the dominance relation is not affected. Both the matrix transformation and its inverse can be done in time and memory $O(n)$.

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 18. Therefore, given the set of nonzero index pairs of D'_A, D'_B, the set of nonzero index pairs of D'_C can be computed in time $O(n^{1.5})$ and memory $O(n)$. □

The above lemma is illustrated by Fig. 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) curves. Thick top-to-bottom curves correspond immediately to a subset of nonzeros in D_C. Thin top-to-bottom curves can be used to determine the remaining nonzeros in D_C by application of Lemma 18.

We are now able to formulate our alternative to Lemma 16.

Lemma 20. *Consider the concatenation of alignment dags as described above, with extended 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 $O(M + n)$.*

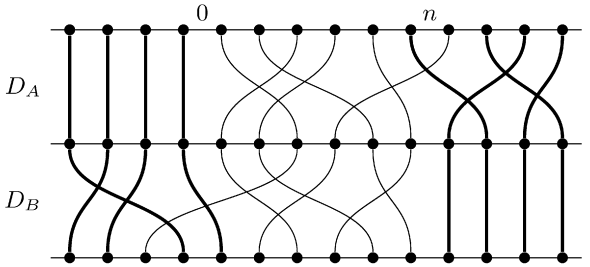

Fig. 3. An illustration of Lemma 19.

Proof. Let $D'_A(i,j) = D_A(i-M,j)$, $D'_B(j,k) = D_B(j,k+m)$, $D'_C(i,k) = D_C(i-M,k+m)$ 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 19, therefore $M+m$ of the non-trivial C-critical points can be obtained by (10)–(11) in time and memory $O(M+m) = O(M)$, and the remaining n non-trivial C-critical points in time $O(n^{1.5})$ and memory $O(n)$. □

Lemma 20 is formulated for arbitrary values of M, m (as long as $M \geqslant m$). However, it only improves on Lemma 16 in the case $n = o(m^2)$, which is sufficient to obtain the claimed results. A generalisation of Lemma 20 giving an unconditional improvement on Lemma 16 will be considered elsewhere.

6. All semi-local LCS computation

We now describe our all semi-local LCS algorithm. From now on, we assume without loss of generality that $n \leqslant m$. We will also assume that m and n are reasonably close, so that $(\log m)^c \leqslant n$ for some constant c, to be specified separately. First, we give a simple algorithm, based on the fast highest-score matrix multiplication procedure of Lemma 20, and running in overall time $O(mn)$. We then modify the algorithm to achieve running time $o(mn)$.

Algorithm 1 *(All semi-local LCS, basic version).*

Input: strings a, b of length m, n, respectively; we assume $\log m \leqslant n \leqslant m$.
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 20. Note that we now have two nested recursions: the main recursion of the algorithm, and the inner recursion of Lemma 20.

In the process of main recursion, the algorithm may (and typically will, as long as the current values of m and n are sufficiently close) alternate 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 15.

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

Cost analysis. Consider the main recursion tree. The computation work in the top $\log(m/n)$ levels of the tree is at most $\log(m/n) \cdot O(m) + (m/n) \cdot O(n^{1.5}) = O(mn)$. The computation work in the remaining $2\log n$ levels of 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 work 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 the claimed subquadratic running time, using an idea originating in [4] and subsequently applied to string comparison by [17].

Algorithm 2 *(All semi-local LCS, full version).*

Input, output: as in Algorithm 1; we assume $(\log m)^{5/2} \leqslant n \leqslant m$.
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 [2]). 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 $\sigma^{2t} \cdot O(t^2) = \frac{1}{4} \cdot m(\log_\sigma m)^2 = O(\frac{mn}{\log^{1/2}(m+n)})$.

Consider the main recursion tree. The computation work in the top $\log(m/n)$ levels of the tree is at most $\log(m/n) \cdot O(m) + (m/n) \cdot O(n^{1.5}) = O(\frac{mn}{\log^{1.5} m}) + O(\frac{mn}{\log^{5/4} m}) = o(\frac{mn}{\log^{1/2}(m+n)})$. The computational work in the remaining $2\log(n/t)$ levels of 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 work is $mn/t^2 \cdot O(t^{1.5}) = O(mn/t^{1/2}) = O(\frac{mn}{\log^{1/2}(m+n)})$.

7. Conclusions

We have presented a new approach to the all semi-local LCS problem. Our approach results in a significantly improved output representation, and yields the first subquadratic algorithm for the problem, with running time $O(\frac{mn}{\log^{1/2}(m+n)})$ when m and n are reasonably close.

An immediate open question is whether the time efficiency of our algorithm can be improved even further, e.g., to match the (global) LCS algorithms of [8,17] with running time $O(\frac{mn}{\log(m+n)})$. Our algorithm, as well as the algorithms of [8,17], assume constant alphabet size. For this class of algorithms, it is possible to remove this restriction at the price of an extra $O((\log\log n)^2)$ factor in the running time (see e.g. [1,3]).

Another interesting question is whether our algorithm can be adapted to more general string comparison. The *edit distance problem* concerns a minimum-cost transformation between two strings, with given costs for character insertion, deletion and substitution. The LCS problem is equivalent to the edit distance problem with insertion/deletion cost 1 and substitution cost 2 or greater. By a constant-factor blow-up of the alignment dag, our algorithm can solve the all semi-local edit distances problem, where the insertion, deletion and substitution edit costs are any constant rationals. It remains an open question whether this can be extended to arbitrary real costs, or to sequence alignment with non-linear gap penalties.

Finally, our technique appears general enough to be able to find applications beyond semi-local comparison. In particular, could it be applied in some form to the biologically important case of fully local (i.e. every substring against every substring) comparison?

References

[1] A.V. Aho, D.S. Hirschberg, J.D. Ullman, Bounds on the complexity of the longest common subsequence problem, Journal of the ACM 23 (1976) 1–12.
[2] C.E.R. Alves, E.N. Cáceres, S.W. Song, An all-substrings common subsequence algorithm, Electronic Notes in Discrete Mathematics 19 (2005) 133–139.
[3] A. Apostolico, String editing and longest common subsequences, in: Handbook of Formal Languages, vol. 2, Springer-Verlag, 1997, pp. 361–398.
[4] V.L. Arlazarov, E.A. Dinic, M.A. Kronrod, I.A. Faradzev, On economical construction of the transitive closure of an oriented graph, Soviet Mathematical Doklady 11 (1970) 1209–1210.
[5] J.L. Bentley, Multidimensional divide-and-conquer, Communications of the ACM 23 (4) (1980) 214–229.
[6] R.E. Burkard, B. Klinz, R. Rudolf, Perspectives of Monge properties in optimization, Discrete Applied Mathematics 70 (1996) 95–161.
[7] M. Crochemore, G.M. Landau, B. Schieber, M. Ziv-Ukelson, Re-use dynamic programming for sequence alignment: An algorithmic toolkit, in: String Algorithmics, in: Texts in Algorithmics, vol. 2, King's College Publications, 2004.
[8] M. Crochemore, G.M. Landau, M. Ziv-Ukelson, A subquadratic sequence alignment algorithm for unrestricted score matrices, SIAM Journal on Computing 32 (6) (2003) 1654–1673.
[9] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, 1994.
[10] D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, 1997.
[11] J. JáJá, C. Mortensen, Q. Shi, Space-efficient and fast algorithms for multidimensional dominance reporting and counting, in: Proceedings of the 15th ISAAC, in: Lecture Notes in Computer Science, vol. 3341, Springer, 2004, pp. 558–568.
[12] N.C. Jones, P.A. Pevzner, An introduction to bioinformatics algorithms, in: Computational Molecular Biology, The MIT Press, 2004.
[13] S.-R. Kim, K. Park, A dynamic edit distance table, Journal of Discrete Algorithms 2 (2004) 303–312.
[14] G.M. Landau, E. Myers, M. Ziv-Ukelson, Two algorithms for LCS consecutive suffix alignment, in: Proceedings of the 15th CPM, in: Lecture Notes in Computer Science, vol. 3109, Springer, 2004, pp. 173–193.
[15] G.M. Landau, E.W. Myers, J.P. Schmidt, Incremental string comparison, SIAM Journal on Computing 27 (2) (1998) 557–582.
[16] G.M. Landau, M. Ziv-Ukelson, On the common substring alignment problem, Journal of Algorithms 41 (2001) 338–359.
[17] W.J. Masek, M.S. Paterson, A faster algorithm computing string edit distances, Journal of Computer and System Sciences 20 (1980) 18–31.
[18] G. Navarro, A guided tour to approximate string matching, ACM Computing Surveys 33 (1) (2001) 31–88.
[19] F.P. Preparata, M.I. Shamos, Computational Geometry: An Introduction, Texts and Monographs in Computer Science, Springer, 1985.
[20] 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) (1998) 972–992.
[21] A. Tiskin, All semi-local longest common subsequences in subquadratic time, in: Proceedings of CSR, in: Lecture Notes in Computer Science, vol. 3967, Springer, 2006, pp. 352–363.
[22] A. Tiskin, Longest common subsequences in permutations and maximum cliques in circle graphs, in: Proceedings of CPM, in: Lecture Notes in Computer Science, vol. 4009, Springer, 2006, pp. 271–282.