Semi-local string comparison:
Algorithmic techniques and applications

Alexander Tiskin\textsuperscript{1}

May 12, 2010

\textsuperscript{1}Department of Computer Science, University of Warwick, Coventry CV4 7AL, United Kingdom. Research supported by the Centre for Discrete Mathematics and Its Applications (DIMAP), University of Warwick, and by the Royal Society Leverhulme Trust Senior Research Fellowship.
Abstract

The longest common subsequence (LCS) problem is a classical problem in computer science. The semi-local LCS problem is a generalisation of the LCS problem, arising naturally in the context of string comparison. Apart from playing an important role in string algorithms, this problem turns out to have surprising connections with computational geometry, algebra, graph theory, as well as applications in computational molecular biology. Our work gives a survey of recent results that expose these connections. We conclude that semi-local string comparison is a fascinating problem and a powerful algorithmic technique, which unifies and improves on a number of previous approaches.
## Contents

1 Introduction 3  
1.1 Overview ........................................ 3  
1.2 Points and matrices ..................................... 5  
1.3 Distribution and density matrices .......................... 8  
1.4 Permutation and unit-Monge matrices ...................... 9  

2 Matrix distance multiplication 13  
2.1 Distance multiplication monoids .......................... 13  
2.2 Seaweed braids ........................................... 17  
2.3 Distance multiplication algorithms ........................ 19  
2.4 Bruhat order ............................................. 27  

3 Semi-local string comparison 31  
3.1 Semi-local LCS .......................................... 31  
3.2 Alignment dags and score matrices ........................ 33  
3.3 Score matrix composition .................................. 40  

4 The seaweed method 46  
4.1 The seaweed algorithm .................................... 46  
4.2 The micro-block speedup ................................... 49  
4.3 Incremental LCS and semi-local LCS ........................ 52  
4.4 Common-substring LCS and semi-local LCS .................. 54  
4.5 Cyclic LCS .............................................. 55  
4.6 Longest repeating subsequence ............................. 55  

5 Weighted string comparison 57  
5.1 Weighted scores and edit distances ......................... 57  
5.2 Approximate pattern matching .............................. 60  

6 Periodic string comparison 64  
6.1 The periodic seaweed algorithm ............................ 64  
6.2 Tandem alignment ........................................ 67
Chapter 1

Introduction

1.1 Overview

The longest common subsequence (LCS) problem is a classical problem in computer science. Given two strings $a$, $b$ of lengths $m$, $n$ respectively, the LCS problem asks for the length of the longest string that is a subsequence of both $a$ and $b$. This length is called the strings’ LCS score. We refer the reader to monographs [42, 63] for the background and further references.

The semi-local LCS problem is a generalisation of the LCS problem, arising naturally in the context of string comparison. Given two strings $a$, $b$ as before, the semi-local LCS problem asks for the LCS score of each string against all substrings of the other string, and of all prefixes of each string against all suffixes of the other string. In this work, we survey a number of algorithmic techniques related to the semi-local LCS problem, and present some algorithmic applications of these techniques.

The rest of this chapter contains the necessary preliminaries. In Section 1.2, we establish the basic terminology and notation of points and matrices. In Section 1.3, we give the key definitions of distribution and density matrices. In Section 1.4, we introduce our main algorithmic tool: a special class of integer matrices, called simple unit-Monge matrices, which are obtained as distribution matrices of permutation matrices, and can be represented implicitly by a dominance counting data structure, such as a range tree.

In Chapter 2, we describe our main algorithmic techniques. In Section 2.1, we introduce matrix distance multiplication, and study its algebraic properties in the classes of Monge and simple unit-Monge matrices. In Section 2.2, we present an algebraic formalism for distance multiplication of simple unit-Monge matrices, called seaweed braids. In Section 2.3, we give an efficient algorithm for distance multiplication of implicit simple unit-Monge matrices (or, equivalently, seaweed braids). In Section 2.4, we describe an application of this algorithm to deciding comparability in the
Bruhat partial order on permutations.

In Chapter 3, we develop an application of our techniques to the semi-local LCS problem. In Section 3.1, we formally define the semi-local LCS problem and related concepts. In Section 3.2, we introduce alignment dags and highest-score matrices. Exploiting the algebraic framework of unit-Monge matrices, in Section 3.3 we obtain an efficient algorithm for highest-score matrix composition.

In Chapter 4, we introduce a conceptually simple algorithm for the semi-local LCS problem, called the seaweed algorithm, and show a number of its applications. In Section 4.1, we describe the seaweed algorithm itself, and in Section 4.2, its slightly faster version, obtained by micro-block precomputation. In Sections 4.3 and 4.4, we apply the seaweed algorithm to solving the incremental and the common-substring versions of the LCS and semi-local LCS problems. In Section 4.5 we give an algorithm for the cyclic LCS problem, and in Section 4.6 for the longest repeating subsequence problem. All our algorithms match, improve on, and/or generalise existing algorithms for these problems.

In Chapter 5, we generalise our techniques from LCS scores to arbitrary rational-weighted alignment scores and edit distances. In Section 5.1, we introduce the main concepts of weighted alignment, and describe an approach to rational-weighted alignment via the blow-up technique. In Section 5.2, we use the framework of weighted alignment to obtain algorithms for several versions of the approximate pattern matching problem.

In Chapter 6, we describe an extension of the seaweed algorithm that allows efficient semi-local comparison of two input strings, one of which is periodic. In Section 6.1, we describe the periodic seaweed algorithm itself. By application of the periodic seaweed algorithm, in Section 6.2 we obtain new algorithms for the tandem LCS problem and the tandem cyclic alignment problem, improving on existing algorithms in running time.

In Chapter 7, we consider the semi-local LCS problem restricted to permutation strings of length $n$. In particular, Section 7.1 gives an algorithm for the semi-local LCS problem on permutation strings. By direct application of this algorithm, in Section 7.2 we obtain an improved algorithm for the cyclic LCS problem on permutations. Further applications include improved algorithms for the longest pattern-avoiding subsequence problem, given in Section 7.3, and for the longest $k$-increasing and $k$-modal subsequence problems, given in Section 7.4. In Section 7.5, we consider the maximum clique problem in a circle graph represented by an interval model of size $n$. By application our semi-local LCS algorithm on permutations, we obtain new algorithms for this problem, both for general and sparse circle graphs, achieving a substantial improvement on existing algorithms in running time. In Section 7.6, we describe an application of these algorithms to the problem of finding exact and approximate commonly structured patterns in linear graphs.
In Chapter 8, we apply the semi-local LCS problem to compressed string comparison. Our goal is to obtain efficient algorithms that work on compressed strings without first decompressing them. In Section 8.1, we introduce the grammar compression (GC) framework, that generalises the classical LZ78 and LZW methods. In Section 8.2, we describe a folklore algorithm for global subsequence recognition on GC-strings. In Section 8.3, we give an efficient algorithm for the three-way semi-local LCS problem on GC-strings. By application of this algorithm, in Section 8.4 we obtain an algorithm for local subsequence recognition in GC-strings, and in Section 8.5 an algorithm for threshold approximate matching in GC-strings; both these algorithms improve on the existing ones in running time.

In Chapter 9, we consider applications of our techniques that aim to reach beyond semi-local string comparison, with the ultimate goal of efficient fully-local comparison. In Section 9.1, we introduce the window-substring and window-window LCS problems, and give an algorithm for these problems. This algorithm provides a refinement for the standard dot plot method, by allowing efficient window-window string comparison based on the LCS score, rather than the less sensitive Hamming score. In Section 9.2, we introduce the quasi-local LCS problem, which generalises the semi-local, window-substring and window-window LCS problems, and give an efficient algorithm for this problem. By application of this algorithm, in Section 9.3 we obtain an algorithm for sparse spliced alignment under an arbitrary rational edit distance metric, which improves on existing algorithms for this problem.

Some results presented in this work appeared incrementally in the author’s publications [121, 122, 124, 123, 125, 86, 126]. The aim of this work is to consolidate these results, unifying the terminology and notation. However, a number of results have not been published before, and are original to this work.

Summarising the presented results, 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.

1.2 Points and matrices

For indices, we will use either integers, or odd half-integers:

\[
\{\ldots, -2, -1, 0, 1, 2, \ldots\}
\]

\[
\{\ldots, -\frac{5}{2}, -\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \ldots\}
\]

For ease of reading, odd half-integer variables will be indicated by hats (e.g. \(\hat{i}, \hat{j}\)). Ordinary variable names (e.g. \(i, j\), with possible subscripts or super-
scripts), will normally denote integer variables, but can sometimes denote a variable that may be either integer, or odd half-integer.

We denote integer and odd half-integer intervals by

\[ [i : j] = \{i, i + 1, \ldots, j - 1, j\} \]
\[ \langle i : j \rangle = \{i + \frac{1}{2}, i + \frac{3}{2}, \ldots, j - \frac{3}{2}, j - \frac{1}{2}\} \]

Observe that in this notation, both an integer and an odd half-integer interval is defined by integer endpoints. To denote infinite intervals of integers and odd half-integers, we will use \(-\infty\) for \(i\) and \(+\infty\) for \(j\) where appropriate, so e.g. \([-\infty : +\infty]\) denotes the set of all integers, and \((-\infty : +\infty)\) the set of all odd half-integers. For finite intervals \([i : j]\) and \(\langle i : j \rangle\), we call the difference \(j - i\) interval length.

When dealing with pairs of numbers, we will often use geometric language and call them points. We will write

\[(i_0, j_0) \ll (i_1, j_1) \quad \text{if} \quad i_0 < i_1 \text{ and } j_0 < j_1\]
\[(i_0, j_0) \leq (i_1, j_1) \quad \text{if} \quad i_0 < i_1 \text{ and } j_0 > j_1\]

We will call these strict partial orders \(\ll\)- and \(\leq\)-dominance. When visualising points, we will use the matrix indexing convention: the first coordinate in a pair increases downwards, and the second coordinate rightwards. Hence, the visual convention of the dominated point lying “below and to the left” of the dominating point, which is standard in computational geometry, corresponds in this work to \(\leq\)-dominance.

We use standard terminology for geometric dominance and other partial orders. In particular, a set of elements forms a chain, if they are pairwise comparable, and an antichain, if they pairwise incomparable. Note that a \(\ll\)-chain is a \(\leq\)-antichain, and vice versa. An element in a partially ordered set is minimal (respectively, maximal), if, in terms of the partial order, it does not dominate (respectively, is not dominated by) any other element in the set. All minimal (respectively, maximal) elements in a partially ordered set form an antichain.

A function of an integer argument will be called unit-monotone increasing (respectively, decreasing), if in every successive pair of values, the difference between the successor and the predecessor is either 0 or 1 (respectively, 0 or \(-1\)).

We will make extensive use of vectors and matrices with integer (occasionally, rational or real) elements, and with integer or odd half-integer indices. We regard a vector or matrix as a one- (respectively, two-) argument function, so we can speak e.g. about unit-monotone increasing matrices.

We will sometimes consider matrices where one or both index range are non-consecutive sets of integers or odd half-integers; however, index ranges will always be assumed to be linearly ordered. Given two index ranges \(I,\)
it will be convenient to denote their Cartesian product by \((I \mid J)\). We extend this notation to Cartesian products of intervals:

\[
\begin{align*}
[i_0 : i_1 \mid j_0 : j_1] &= ([i_0 : i_1] \mid [j_0 : j_1]) \\
\langle i_0 : i_1 \mid j_0 : j_1 \rangle &= (\langle i_0 : i_1 \rangle \mid \langle j_0 : j_1 \rangle)
\end{align*}
\]

Given index ranges \(I, J\), a vector over \(I\) is indexed by \(i \in I\), and a matrix over \((I \mid J)\) is indexed by \(i \in I, j \in J\). A vector or matrix is nonnegative, if all its elements are nonnegative.

The matrices we consider can be implicit, i.e. represented by a compact data structure that supports access to every matrix element in a specified (small, but not necessarily constant) time. If the query time is not given, it is assumed by default to be constant.

When considering matrices over non-consecutive index ranges, we will occasionally perform operations on such matrices as if they were over consecutive intervals. This will have the following meaning: we remap the ranges to consecutive intervals preserving the linear order within each range, then we perform a matrix operation, and finally we remap the intervals back to the original ranges.

We will use function notation for indexing matrices, e.g. \(A(i, j)\). We will use straightforward notation for selecting subvectors and submatrices: for example, given a matrix \(A\) over \([0 : n] \mid [0 : n]\), we denote by \(A[i_0 : i_1 \mid j_0 : j_1]\) the submatrix defined by the given sub-intervals. A star * will indicate that for a particular index, its whole range is selected implicitly, e.g. \(A[* \mid j_0 : j_1]\).

We will denote by \(A^T\) the transpose of matrix \(A\). We will also denote by \(A^R\) the matrix obtained from \(A\) by counterclockwise 90-degree rotation. For a matrix \(A\) over \([0 : n] \mid [0 : n]\) (or \([0 : n] \mid [0 : n]\)), we have \(A^T(i, j) = A(j, i)\), \(A^R(i, j) = A(j, n - i)\) for all \(i, j\).

Given matrices \(A'\) over \((I' \mid J')\) and \(A''\) over \((I'' \mid J'')\), where \((I' \mid J') \cap (I'' \mid J'') = \emptyset\), their disjoint union is the matrix \(A' \sqcup A''\) over \((I' \cup I'') \mid J' \cup J''\), defined by

\[
(A' \sqcup A'')(i, j) = \begin{cases} 
A'(i, j) & \text{if } i \in I', j \in J' \\
A''(i, j) & \text{if } i \in I'', j \in J'' \\
0 & \text{otherwise}
\end{cases}
\]

The matrix disjoint union operation is associative, and can therefore be used for more than two matrices.
1.3 Distribution and density matrices

Definition 1 Let $D$ be a matrix over $\langle i_0 : i_1 | j_0 : j_1 \rangle$. Its *distribution matrix* $D^\Sigma$ over $[i_0 : i_1 | j_0 : j_1]$ is defined by

$$D^\Sigma(i,j) = \sum_{\hat{i} \in \langle i_0 : i_1 \rangle, \hat{j} \in \langle j_0 : j_1 \rangle} D(\hat{i}, \hat{j})$$

for all $i \in [i_0 : i_1], j \in [j_0 : j_1]$. □

Definition 2 Let $A$ be a matrix over $[i_0 : i_1 | j_0 : j_1]$. Its *density matrix* $A^\square$ over $\langle i_0 : i_1 | j_0 : j_1 \rangle$ is defined by

$$A^\square(i,j) = A(\hat{i}, \hat{j} + \frac{1}{2}) - A(\hat{i} - \frac{1}{2}, \hat{j} - \frac{1}{2}) -$$
$$A(\hat{i} + \frac{1}{2}, \hat{j} + \frac{1}{2}) + A(\hat{i} - \frac{1}{2}, \hat{j} + \frac{1}{2})$$

for all $\hat{i} \in \langle i_0 : i_1 \rangle, \hat{j} \in \langle j_0 : j_1 \rangle$. □

The definitions of distribution and density matrices extend naturally to matrices over an infinite index range, as long as the sum in Definition 1 is defined.

Note that for any matrix $D$ as above, and for all $\hat{i}, \hat{j}$, we have

$$(D^\Sigma)^\square(\hat{i}, \hat{j}) = D(\hat{i}, \hat{j})$$

Also, for any matrix $A$ as above, and for all $i, j$, we have

$$(A^\square)^\Sigma(i,j) + b(j) + c(i) = A(i,j)$$

where $b = A(i_1, *)$ is the bottom row vector, and $c = A(\ast, j_0)$ is the leftmost column vector of $A$. An important special case is when $b, c$ are both zero vectors, as in the following definition.

Definition 3 Matrix $A$ will be called *simple*, if $(A^\square)^\Sigma = A$. □

Equivalently, a (finite) matrix is simple, if and only if its entries in the leftmost column and the bottom row are all zeros.

The following classes of matrices play a fundamental role in optimisation theory (see [26] for an extensive survey), as well as in graph and string algorithms.

Definition 4 Matrix $A$ is called *totally monotone*, if

$$A(i, j) > A(i, j') \Rightarrow A(i', j) > A(i', j')$$

for all $i \leq i', j \leq j'$. □
Definition 5 Matrix $A$ is called a Monge matrix, if
\[ A(i, j) + A(i', j') \leq A(i, j') + A(i', j) \]
for all $i \leq i'$, $j \leq j'$. Equivalently, matrix $A$ is a Monge matrix, if $A$ is nonnegative. Matrix $A$ is called an anti-Monge matrix, if $-A$ is Monge.

It is easy to see that Monge matrices form a subclass of totally monotone matrices. By Definition 5, a matrix is Monge, if and only if its density matrix is nonnegative. This condition is equivalent to the canonical structure theorem for Monge matrices, given by Burkard et al. [26].

1.4 Permutation and unit-Monge matrices

Definition 6 A permutation (respectively, subpermutation) matrix is a zero-one matrix containing exactly one (respectively, at most one) nonzero in every row and every column.

Typically, permutation and subpermutation matrices will be indexed by (not necessarily consecutive) odd half-integers. Given sets $I$, $J$ of odd half-integers with $|I| = |J|$, a zero-one matrix $P$ over $(I \mid J)$ is a permutation matrix, if and only if
\[ \sum_{j \in J} P(i, j') = 1 \quad \sum_{i' \in I} P(i', j) = 1 \]
for all $i \in I$, $j \in J$. An identity matrix is a permutation matrix $Id$ over an interval range $\langle i_0 : i_1 \mid j_0 : j_1 \rangle$, such that $Id(i, j) = 1$, iff $i = j$. More generally, an offset indentity matrix is a permutation matrix $Id_h$ over an interval range $\langle i_0 : i_1 \mid j_0 : j_1 \rangle$, where $j_0 - i_0 = j_1 - i_1 = h$, such that $Id_h(i, j) = 1$, iff $j - i = h$. We have $Id_0 = Id$ for any compatible index range. Clearly, an identity or offset indentity matrix can be represented implicitly in constant space and with constant query time. When dealing with identity and offset indentity matrices, we will often omit their index ranges, where they are clear from the context.

When dealing with (sub)permutation matrices, we will write “nonzeros” for “index pairs corresponding to nonzeros”, as long as this does not lead to confusion. We will normally assume that a (sub)permutation matrix with $n$ nonzeros is given implicitly by a compact data structure of size $O(n)$, that allows constant-time access to each nonzero both by the row and by the column index.

Given a permutation matrix $P$ over $(I \mid J)$, and a set $I' \subseteq I$, we will denote by $P(I' \mid \cdot)$ the permutation submatrix row-induced by $I'$, i.e. the permutation submatrix obtained by deleting from $P$ all columns in $I \setminus I'$.
and then deleting from the remaining submatrix all zero rows. A column-induced permutation submatrix \( P(\cdot | J') \) is defined analogously. Both these operations can be implemented in linear time by a sweep of the nonzeros of matrix \( P \).

The following subclasses of Monge matrices play a crucial role in this work.

**Definition 7** A square matrix \( A \) is called a *unit-Monge* (respectively, *subunit-Monge*) matrix, if \( A^\Sigma \) is a permutation (respectively, subpermutation) matrix. Matrix \( A \) is called a *unit-anti-Monge* (respectively, *subunit-anti-Monge*) matrix, if \( -A \) is unit-Monge (respectively, subunit-Monge).

By Definitions 5, 7, any subunit-Monge matrix is unit-Monge, and any unit-Monge matrix is Monge. Similar inclusions hold for (sub)unit-anti-Monge matrices.

Matrices that are both simple and unit-Monge will be our main tool for the rest of this work. Note that such matrices are unit-monotone decreasing (respectively, increasing) in the first (respectively, second) index. Furthermore, a square matrix \( A \) is simple unit-Monge, if and only if \( A = P^\Sigma \), where \( P \) is a permutation matrix.

**Example 1** The following matrix is simple unit-Monge:

\[
\begin{bmatrix}
0 & 1 & 0 \\
1 & 0 & 0 \\
0 & 0 & 1
\end{bmatrix}^\Sigma =
\begin{bmatrix}
0 & 1 & 2 & 3 \\
0 & 1 & 1 & 2 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0
\end{bmatrix}
\]

A permutation matrix \( P \) of size \( n \) can be regarded as an implicit representation of the simple unit-Monge matrix \( P^\Sigma \). Geometrically, a value \( P^\Sigma(i, j) \) is the number of (odd half-integer) nonzeros in matrix \( P \) that are \( \preceq \)-dominated by the (integer) point \((i, j)\). An individual element \( P^\Sigma(i, j) \) can be queried in time \( O(n) \) by a sweep of the nonzeros of \( P \), counting those that are \( \preceq \)-dominated by the query point \((i, j)\). This procedure is known as geometric dominance counting.

Using methods of computational geometry, matrix \( P \) can be preprocessed to allow element queries on \( P^\Sigma(i, j) \) much more efficiently than by a direct linear sweep.

**Theorem 1** Given a (sub)permutation matrix \( P \) of size \( n \), 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 the simple (sub)unit-Monge matrix \( P^\Sigma \) in time \( O(\log^2 n) \).

\[\square\]
Figure 1.1: A permutation matrix and the corresponding range tree

\begin{proof}
The required structure is a two-dimensional range tree [15] (see also [107]), built on the set of nonzeros in \( P \). There are at most \( n \) nonzeros, hence the total number of nodes in the tree is \( O(n \log n) \). A dominance counting query on the set of nonzeros can be answered by accessing \( O(\log^2 n) \) of the tree nodes.
\end{proof}

\begin{example}
Figure 1.1 shows a \( 4 \times 4 \) permutation matrix, with nonzeros indicated by green\(^1\) bullets, and the corresponding range tree.
\end{example}

The bounds given by Theorem 1 can be improved by employing more efficient data structures. Successive improvements in the efficiency of orthogonal range counting (which includes dominance counting as a special case) were obtained by Chazelle [34], and by JáJá et al. [72]. The currently most efficient data structure, given by Chan and Pătraşcu [30], has size \( O(n) \), can be built in time \( O(n(\log \log \log n)^{-1/2}) \), and answers a dominance counting query in time \( O(\frac{\log n}{\log \log \log n}) \). However, the standard range tree data structure employed by Theorem 1 is simpler, requires a less powerful computation model, and is more likely to be practical. Therefore, we will be using Theorem 1 as our main technique for implicit representation of simple (sub)unit-Monge matrices.

In addition to ordinary element queries described by Theorem 1, we will also deal with \textit{incremental queries}, which are given an element of an implicit simple (sub)unit-Monge matrix, and return the value of an adjacent element. This kind of query can be answered directly from the (sub)permutation matrix, without any preprocessing.

\begin{theorem}
Given a (sub)permutation matrix \( P \) of size \( n \), and the value \( P^\Sigma(i, j) \), \( i, j \in [0 : n] \), the values \( P^\Sigma(i \pm 1, j) \), \( P^\Sigma(i, j \pm 1) \), where they exist, can be queried in time \( O(1) \).
\end{theorem}

\(^1\)For colour illustrations, the reader is referred to the online version of this work. If the colour version is not available, all references to colour can be ignored.
Proof. Let $P$ be a permutation matrix; a generalisation to subpermutation matrices is straightforward. Consider a query of the type $P^\Sigma(i+1, j)$; other query types are obtained by symmetry. Let $j \in \{0 : n\}$ be such that $P(i + \frac{1}{2}, j) = 1$; value $j$ can be obtained from the permutation representation of $P$ in time $O(1)$. We have

$$P^\Sigma(i + 1, j) = P^\Sigma(i, j) - \begin{cases} 1 & \text{if } \hat{j} < j \\ 0 & \text{otherwise} \end{cases}$$

Incremental queries described by Theorem 2 can be used to answer batch queries, returning a set of elements in a row, column or diagonal of an implicit simple (sub)unit-Monge matrix. In particular, all elements in a given row, column or diagonal of matrix $P^\Sigma$ can be obtained by a sequence of incremental queries in time $O(n)$, and a subset of $r$ consecutive elements in time $O(r + \log^2 n)$. 


Chapter 2

Matrix distance multiplication

2.1 Distance multiplication monoids

We will make extensive use of the \((\min, +)\)-semiring on integer or real numbers, where the operators \(\min\) and \(+\), denoted by \(\oplus\) and \(\odot\), play the role of addition and multiplication, respectively. This semiring is often called distance (or tropical) algebra; for an extensive review of this and related topics, see e.g. Rote [113], Gondran and Minoux [61]. Our techniques are based on matrix-vector and matrix-matrix multiplication in the distance algebra.

**Definition 8** Let \(A\) be a matrix over \([i_0 : i_1 | j_0 : j_1]\). Let \(b\), \(c\) be vectors over \([j_0 : j_1]\) and \([i_0 : i_1]\) respectively. The matrix-vector distance product \(A \odot b = c\) is defined by

\[
c(i) = \bigoplus_{j \in [j_0 : j_1]} (A(i, j) \odot b(j)) = \min_{j \in [j_0 : j_1]} (A(i, j) + b(j))
\]

for all \(i \in [i_0 : i_1]\).

**Definition 9** Let \(A, B, C\) be matrices over \([i_0 : i_1 | j_0 : j_1], [j_0 : j_1 | k_0 : k_1], [i_0 : i_1 | k_0 : k_1]\) respectively. The matrix distance product \(A \odot B = C\) is defined by

\[
C(i, k) = \bigoplus_{j \in [j_0 : j_1]} (A(i, j) \odot B(j, k)) = \min_{j \in [j_0 : j_1]} (A(i, j) + B(j, k))
\]

for all \(i \in [i_0 : i_1], k \in [k_0 : k_1]\).

Like any multiplication of matrices over a semiring, matrix distance multiplication is associative. The set of all square matrices with elements in
over a given index range forms a monoid with respect to distance multiplication. The identity element in this monoid is the matrix

\[ \text{Id}⊙(i,j) = \begin{cases} 0 & \text{if } i = j \\ +\infty & \text{otherwise} \end{cases} \]

The distance multiplication monoid also has an annihilator \( O⊙ \), where \( O⊙(i,j) = +\infty \) for all \( i, j \). For all \( A \), we have

\[ A ⊗ \text{Id} = \text{Id} ⊗ A = A \quad A ⊗ O⊙ = O⊙ ⊗ A = O⊙ \]

It is well-known that the set of all Monge matrices is closed under distance multiplication. It is slightly more surprising, but crucial for our method, that the same is also true for the set of all simple (sub)unit-Monge matrices.

**Theorem 3** Let \( A, B, C \) be matrices, such that \( A ⊗ B = C \). If \( A, B \) are Monge (respectively, simple unit-Monge, simple subunit-Monge), then \( C \) is also Monge (respectively, simple unit-Monge, simple subunit-Monge).

**Proof** First, let \( A, B \) be Monge matrices. Let \( i \leq i', k \leq k' \). By definition of matrix distance multiplication, we have

\[
C(i,k') = \min_{j^*} (A(i,j^*) + B(j^*,k'))
\]

\[
C(i',k) = \min_{j^*} (A(i',j^*) + B(j^*,k))
\]

Let \( j, j' \) respectively be the values of \( j^* \) on which these minima are attained. Suppose \( j \leq j' \). We have

\[
C(i,k) + C(i',k') = \min_{j^*} (A(i,j^*) + B(j^*,k)) + \min_{j^*} (A(i',j^*) + B(j^*,k')) \leq
\]

\[
(A(i,j) + B(j,k)) + (A(i',j') + B(j',k')) = (A(i,j) + A(i',j')) + (B(j,k) + B(j',k')) \leq (A is Monge)
\]

\[
(A(i,j') + A(i',j)) + (B(j,k) + B(j',k')) = (A(i,j') + B(j',k')) + (A(i',j) + B(j,k)) \leq (A is Monge)
\]

\[
C(i,k') + C(i',k)
\]

The case \( j \geq j' \) is treated symmetrically by the Monge property of \( B \). Hence, matrix \( C \) is Monge.

Now, let \( A, B \) be simple unit-Monge matrices. We have \( A = P_A^\Sigma \), \( B = P_B^\Sigma \). It is easy to check that matrix \( C \) is simple. Let \( C = P_C^\Sigma \); we have to show that \( P_C \) is a permutation matrix. Without loss of generality, let \( P_A, P_B, P_C \) be over \( \langle 0 : n \mid 0 : n \rangle \). Clearly, matrices \( C \) and \( P_C \) are both integer.
Furthermore, matrix $C$ is Monge by the previous argument, and therefore matrix $P_C$ is nonnegative.

For any $i \in [0 : n]$, we have
\[
C(i, 0) = \min_j (P_A^{\Sigma}(i, j) + P_B^\Sigma(j, 0)) = \min_j (P_A^{\Sigma}(i, j) + 0) = 0
\]
\[
C(i, n) = \min_j (P_A^{\Sigma}(i, j) + P_B^\Sigma(j, n)) = \min_j (P_A^{\Sigma}(i, j) + n - j) = n - i
\]
where the minimum is attained respectively at $j = 0$ and $j = n$. Therefore, for all $i \in [0 : n]$, we have
\[
\sum_k P_C(i, k) = \text{(definition of } \Sigma \text{ and } \bullet) \leq n
\]
\[
\sum_k \left( C(i + \frac{1}{2}, \hat{k} - \frac{1}{2}) - C(i - \frac{1}{2}, \hat{k} - \frac{1}{2}) - C(i + \frac{1}{2}, \hat{k} + \frac{1}{2}) + C(i - \frac{1}{2}, \hat{k} + \frac{1}{2}) \right) = 0
\]
\[
C(i + \frac{1}{2}, 0) - C(i - \frac{1}{2}, 0) - C(i + \frac{1}{2}, n) + C(i - \frac{1}{2}, n) = 0 - 0 - (n - i - \frac{1}{2}) + (n - i + \frac{1}{2}) = 1
\]
Symmetrically, for all $\hat{k} \in [0 : n]$, we have
\[
\sum_i P_C(i, \hat{k}) = 1
\]
Taken together, the above properties imply that matrix $P_C$ is a permutation matrix. Therefore, $C$ is a simple unit-Monge matrix.

Finally, let $A$, $B$ be simple subunit-Monge matrices. We have $A = P_A^\Sigma$, $B = P_B^\Sigma$, where $P_A$, $P_B$ are subpermutation matrices. As before, let $C = P_C^\Sigma$; we have to show that $P_C$ is a subpermutation matrix. Without loss of generality, let $P_A$, $P_B$, $P_C$ be over $\langle 0 : n' \mid 0 : n'' \rangle$, $\langle 0 : n'' \mid 0 : n'' \rangle$, $\langle 0 : n' \mid 0 : n'' \rangle$, respectively. Suppose that for some $i$, row $P_A(i, *)$ contains only zeros. Then, it is easy to check that row $P_C(i, *)$ also contains only zeros. Symmetrically, a zero column $P_B(*, \hat{k})$ results in a zero column $P_C(*, \hat{k})$. After deleting all zero rows from $P_A$, all zero columns from $P_B$, and all the corresponding zero rows and columns from $P_C$, the equality $P_A^\Sigma \circ P_B^\Sigma = P_C^\Sigma$ still holds; however, matrix $P_C$ is not necessarily a permutation matrix, and may not even be square.

Without loss of generality, we may now assume that $n' \leq n''$, $n'' \leq n''$, and that subpermutation matrix $P_A$ (respectively, $P_B$) does not have any zero rows (respectively, zero columns). Therefore, matrices $P_A$, $P_B$ have exactly $n'$ and $n''$ nonzeros, respectively. We now extend matrix $P_A$ to a permutation matrix $\tilde{P}_A$ over $\langle n' - n'' : n' : n'' \rangle$, by adding $n'' - n'$ rows with appropriately placed nonzeros at the top of the matrix. Similarly, we extend matrix $P_B$ to a permutation matrix $\tilde{P}_B$ over $\langle 0 : n'' : 0 : n'' \rangle$, by
adding $n'' - n'$ columns with appropriately placed nonzeros at the right-
hand side of the matrix. Let $\hat{P}_A \boxdot \hat{P}_B = \hat{P}_C$, where $\hat{P}_C$ is a permutation
matrix over $\langle n' - n'' : n' | 0 : n'' \rangle$. We have

$$P_C = P_A \boxdot P_B = \hat{P}_C(0 : n' | 0 : n'')$$

Hence, matrix $P_C$ is a subpermutation matrix, and the original matrix $C$ is
a simple subunit-Monge matrix. ■

The distance multiplication identity matrix $Id \otimes$ can be formally considered
a Monge matrix (since all indeterminate expressions of the form $+\infty - \infty$
in its density matrix can be considered nonnegative). Therefore, the set of
all square Monge matrices forms a submonoid in the distance multiplication
monoid of general matrices.

The set of all simple unit-Monge matrices over a given index range also
forms a monoid with respect to distance multiplication. The identity element
in this monoid is the distribution matrix of the identity permutation matrix
$Id^\Sigma$:

$$Id^\Sigma(i, j) = \begin{cases} j - i & \text{if } i \leq j \\ 0 & \text{otherwise} \end{cases}$$

The simple unit-Monge matrix monoid also has an annihilator $(Id^R)^\Sigma$ (recall
$Id^R$ is the matrix obtained by 90-degree rotation of $Id$). For all $P$, we have

$$P^\Sigma \otimes Id^\Sigma = Id^\Sigma \otimes P^\Sigma = P^\Sigma$$

$$P^\Sigma \otimes (Id^R)^\Sigma = (Id^R)^\Sigma \otimes P^\Sigma = (Id^R)^\Sigma$$

Theorem 3 gives us the basis for performing distance multiplication of
simple (sub)unit-Monge matrices implicitly, by taking the density (sub)permutation
matrices as input, and producing a density (sub)permutation matrix as out-
put. It will be convenient to introduce special notation for implicit distance
multiplication of this kind.

**Definition 10** Let $P$ be a (sub)permutation matrix. Let $b, c$ be vectors.
The implicit matrix-vector distance product $P \boxdot b = c$ is defined by $P^\Sigma \circ b = c$.

**Definition 11** Let $P_A, P_B, P_C$ be (sub)permutation matrices. The implicit
matrix distance product $P_A \boxdot P_B = P_C$ is defined by $P_A^\Sigma \circ P_B^\Sigma = P_C^\Sigma$.

The set of all permutation matrices over $\langle 0 : n | 0 : n \rangle$ is therefore a monoid
with respect to implicit distance multiplication $\boxdot$, with identity element $Id$
and annihilator $Id^R$.

**Example 3** In Figure 2.1, Subfigure 2.1a shows a triple of $6 \times 6$ permutation
matrices $P_A, P_B, P_C$, with nonzeros indicated by green bullets, such that
$P_A \boxdot P_B = P_C$. □
2.2 Seaweed braids

Further understanding of the distance multiplication monoid of simple unit-Monge matrices can be gained by the following construction. Given a permutation matrix $P$ over $(I \mid J)$, we represent the indices in sets $I$ and $J$ by nodes on two parallel lines in the Euclidean plane, respecting the order of indices within each set. We represent every nonzero $P(\hat{i}, \hat{j}) = 1$ by connecting node $\hat{i} \in I$ with node $\hat{j} \in J$ by a continuous monotone line called a seaweed. We call the resulting configuration a seaweed braid. Unless $P$ is the identity matrix $Id$, some of the seaweeds in the seaweed braid will have to cross. However, the seaweeds are drawn so that any “unnecessary” crossings are avoided; in other words, a given pair of seaweeds can only cross at most once.

Consider the implicit distance product $P_A \boxtimes P_B = P_C$, where $P_A$, $P_B$, $P_C$ are permutation matrices over $(I \mid J)$, $(J \mid K)$ and $(I \mid K)$, respectively. We represent the indices in sets $I$, $J$, $K$ by nodes on three parallel lines, and the nonzeros of the input matrices $P_A$, $P_B$ by two seaweed braids connecting the corresponding nodes. A seaweed braid for the output matrix $P_C$ can be obtained as follows. First, we remove the nodes representing the index set $J$. At each removed node $\hat{j} \in J$, we join up the two incident seaweeds, which represent nonzeros $P_A(\hat{i}, \hat{j}) = 1$ and $P_B(\hat{j}, k) = 1$. We now have a seaweed configuration between nodes of $I$ and nodes of $K$. However, some seaweed pairs in this configuration may cross twice. We now “comb” the seaweeds by running through all their crossings, respecting the top-to-bottom partial order of the crossings from $I$ to $K$. For each crossing, we check whether the two crossing seaweeds have previously crossed above the current point. If
this is the case, then we undo the current crossing by cutting it out of the configuration, and replacing it by two non-crossing seaweed pieces. After all the crossings have been processed, the resulting configuration is a seaweed braid representing the output matrix $P_C$.

**Example 4** In Figure 2.3, Subfigure 2.1b shows the seaweed braids representing the implicit matrix distance product in Subfigure 2.1a.

Seaweed braids can be formalised algebraically as follows. The seaweed monoid $T_n$ is a finitely presented monoid on $n$ generators $id, g_1, g_2, \ldots, g_{n-1}$. Generator $id$ is the identity element, which corresponds to a seaweed braid where all the seaweeds are parallel. Each of the remaining generators $g_t$ corresponds to a seaweed braid where all the seaweeds are parallel, except a pair of neighbouring seaweeds in positions $t - \frac{1}{2}$ and $t + \frac{1}{2}$, which do cross. In matrix notation, the identity generator $id$ corresponds to the simple unit-Monge matrix $Id^\Sigma$, and each generator $g_t$ corresponds to a simple unit-Monge matrix $P^\Sigma_t$, where an elementary transposition matrix $P_t$ is a permutation matrix defined by $P_t(i, j) = 1$ iff $i = j \notin \{t - \frac{1}{2}, t + \frac{1}{2}\}$ or $\{i, j\} = \{t - \frac{1}{2}, t + \frac{1}{2}\}$. Concatenation of words in the generators corresponds to the composition of seaweed braids. The presentation of monoid $T_n$ consists of the idempotence relations

$$g_t^2 = g_t \quad t \in [1 : n - 1] \quad (2.1)$$

and the braid relations

$$g_t g_u = g_u g_t \quad t, u \in [1 : n - 1], u - t \geq 2 \quad (2.2)$$

$$g_t g_u g_t = g_u g_t g_u \quad t, u \in [1 : n - 1], u - t = 1 \quad (2.3)$$

Intuitively, relations (2.1) express the seaweeds’ double crossing property; relations (2.2) express the commutativity of independent seaweed crossings (note that pairs of crossing with $|t - u| \leq 1$ are not independent, so their corresponding generators do not commute); and relations (2.3) give two equivalent expressions for a local crossing of three seaweeds.

We are now able to establish a formal connection between distance multiplication of simple unit-Monge matrices and the seaweed monoid.

**Theorem 4** The distance multiplication monoid of $n \times n$ simple unit-Monge matrices is isomorphic to the seaweed monoid $T_n$.

**Proof** It is straightforward to check that any simple unit-Monge matrix $P^\Sigma$ can be decomposed into a distance product of matrices $P^\Sigma_t$ for various values of $t$; this can be visualised as drawing a seaweed configuration for $P$, and decomposing it into individual seaweed crossings. Hence, matrices $P^\Sigma_t$ serve as generators for the distance multiplication monoid of simple unit-Monge matrices. By using the defining relations of the seaweed monoid, it
is also straightforward to check that multiplication in both monoids agrees on the generators. By associativity of multiplication, this implies that multiplication in monoids agrees on all the elements, therefore the two monoids are isomorphic.

The classical positive braid monoid (see e.g. [77, Section 6.5]) on generators $id, g_1, g_2, \ldots, g_{n-1}$, is defined by the braid relations (2.2)–(2.3) alone. Therefore, the seaweed monoid is isomorphic to the quotient of the positive braid monoid by the idempotence relations (2.1). The seaweed monoid has been previously introduced as the 0-Hecke monoid (of a symmetric group) by Fomin and Greene [54], and by Buch et al. [24]. A generalisation of the seaweed monoid is given by 0-Hecke monoids of Coxeter groups, also known as Coxeter monoids, which arise naturally as subgroup monoids in groups.

The theory of Coxeter monoids can be traced back to Bourbaki [23], and has been developed by Tsaranov [128] and Richardson and Springer [111]. The contents of this and the following sections can be regarded as the first step in the algorithmic study of Coxeter monoids.

### 2.3 Distance multiplication algorithms

In this section, we show that distance multiplication of Monge and simple unit-Monge matrices can be performed much more efficiently than a naive implementation of the definitions, by exploiting the special properties of the matrices. For simplicity, we only consider square matrices, although some of the results generalise to rectangular ones.

We begin with matrix-vector distance multiplication. For generic, explicitly represented matrices, the fastest method for matrix-vector distance multiplication of size $n$ is by direct application of Definition 8 in time $O(n^2)$. For implicit Monge matrices, the running time can be substantially reduced by an application of a classical row minima searching algorithm by Aggarwal et al. [1] (see also [58]), often nicknamed the “SMAWK algorithm”.

**Lemma 1 ([1])** Let $A$ be an $n \times n$ implicit totally monotone matrix, where each element can be queried in time $q$. The problem of finding the (say, leftmost) minimum element in every row of $A$ can be solved in time $O(qn)$.

**Proof** We give a sketch of the proof; for details, see [1, 58].

Let $A'$ be an implicit $\frac{n}{2} \times n$ matrix, obtained by taking every other row of $A$. Clearly, at least $\frac{n}{2}$ columns of $A'$ do not contain any of its leftmost row minima. The key idea of the algorithm is to eliminate such columns in an efficient process, based on the total monotonicity property.

Let $A'$ be over $[i_0 : i_1 | j_0 : j_1]$, $i_1 - i_0 = \frac{n}{2}$, $j_1 - j_0 = n$. The column elimination procedure builds a “staircase” of matrix entries that belong to yet uneliminated columns, but are already known not to contain a leftmost
\[ i \leftarrow i_0 \quad j \leftarrow j_0 \quad j' \leftarrow j_0 + 1 \]

while \( j' \leq j_1 \):

\[
\begin{align*}
\text{case } A(i, j) &\leq A(i, j') : \\
\text{case } i < i_1 &: \quad i \leftarrow i + 1 \quad j \leftarrow j' \\
\text{case } i = i_1 &: \quad \text{eliminate column } j' \\
\quad j' &\leftarrow j' + 1 \\
\text{case } A(i, j) &> A(i, j') : \\
\quad \text{eliminate column } j \\
\text{case } i = i_0 &: \quad j \leftarrow j' \quad j' \leftarrow j' + 1 \\
\text{case } i > i_0 &: \quad i \leftarrow i - 1 \quad j \leftarrow \max\{k : k \text{ uneliminated and } < j}\end{align*}
\]

Table 2.1: Elimination procedure of Lemma 1.

row minimum. In every iteration, the “staircase” is either extended by one “step”, or a whole column is eliminated from the matrix, reducing the “staircase” by one “step”. The decision depends on the comparison of two elements \( A(i, j), A(i, j') \) at the tip of the “staircase”. Following the comparison and possible column elimination, indices \( i, j, j' \) are updated to point to the tip of the updated (extended or reduced) “staircase”. The full elimination procedure is given in Table 2.1. By storing indices of uneliminated columns in an appropriate dynamic data structure, such as a doubly-linked list, a single iteration of this procedure can be implemented to run in time \( O(q) \). The whole elimination procedure runs in time \( O(qn) \), and eliminates \( n^2 \) columns.

Let \( A'' \) be the \( \frac{n}{2} \times \frac{n}{2} \) matrix obtained from \( A' \) by deleting the eliminated columns. We call the algorithm recursively on \( A'' \). Given the output of the recursive call, which provides the leftmost row minima of \( A' \), it is now straightforward to fill in the leftmost minima in the remaining rows of \( A \) in time \( O(qn) \). Thus, a single recursive call runs in time \( O(qn) \). The amount of work halves in every level of recursion, therefore the overall running time is \( O(qn) \).

Example 5 Figure 2.2 gives a snapshot of the two non-boundary cases of the elimination algorithm in the proof of Lemma 1. Each vertical dotted line represents an arbitrary number of consecutive eliminated columns. Dark-shaded cells represent the “staircase” of matrix entries that belong to yet uneliminated columns, but are known not to contain any leftmost row minima. The current elements \( A(i, j), A(i, j') \) at the tip of the “staircase” are shown by white circles.

Subfigure 2.2a shows the case \( A(i, j) \leq A(i, j'), \ i < i_1 \), and Subfigure 2.2b the case \( A(i, j) > A(i, j'), \ i > i_1 \). In both cases, the light-shaded cells represent the new entries that become known not to contain any leftmost row minima. In Subfigure 2.2a, these new entries extend the “staircase”
by one “step”. In Subfigure 2.2b, these new entries result in the elimination of a whole column, reducing the “staircase” by one “step”. In both cases, the new positions for elements $A(i, j)$, $A(i, j')$ are shown by black bullets.

**Theorem 5** Let $A$ be an $n \times n$ implicit Monge matrix, where each element can be queried in time $q$. Let $b, c$ be $n$-vectors, such that $A \odot b = c$. Given vector $b$, vector $c$ can be computed in time $O(qn)$.

**Proof** Let $\tilde{A}(i, j) = A(i, j) + b(j)$. Matrix $\tilde{A}$ is an implicit Monge matrix, where each element can be queried in time $q + O(1)$. The problem of computing the product $A \odot b = c$ is equivalent to searching for row minima in matrix $\tilde{A}$, which can be solved in time $O(qn)$ by Lemma 1.

Let us now restrict our attention to implicit unit-Monge matrices. By Theorem 1, an element of such a matrix (represented by an appropriate data structure) can be queried in time $q = O(\log^2 n)$. By plugging this query time into Theorem 5, we obtain immediately an algorithm for implicit matrix-vector distance multiplication, running in time $O(n \log^2 n)$. A more careful analysis of the proof of Lemma 1 shows that its required matrix elements can be obtained, instead of the standalone element queries of Theorem 1, by more efficient incremental queries of Theorem 2, with $q = O(1)$. As a result, we obtain the following algorithm.

**Lemma 2** Let $A$ be an $n \times n$ implicit (sub)unit-Monge matrix over $[i_0 : i_1 \mid j_0 : j_1]$, represented by permutation matrix $P = A^{\square}$ and vectors $b = A(i_1, \ast)$, $c = A(\ast, j_0)$. The problem of finding the (say, leftmost) minimum element in every row of $A$ can be solved in time $O(n)$.

**Proof** We follow the algorithm outlined in the proof of Lemma 1. We claim that under the conditions of the current lemma, every matrix element query in the algorithm can be performed as an incremental query of Theorem 2. Consider the column elimination procedure of Lemma 1. At the end of a loop iteration, we have just updated the values of $i, j, j'$. For the next
iteration, we need to query $A(i, j)$, $A(i, j')$ with the new index values. Given
the old value of $A(i, j')$, these new values can be obtained by two incremental
queries in time $O(1)$, except where the previous iteration has gone through
the case $A(i, j) > A(i, j')$, $i > i_0$. In this case, the new value $A(i, j')$ can
still be obtained by an incremental query from the old $A(i, j')$. To query
the new value $A(i, j)$, we observe that the new $i$ is the old $i - 1$, and the
new $j$ is the index of the rightmost uneliminated column to the left of the
old $j$. If the new $i$ is equal to $i_0$, querying $A(i, j)$ is trivial. Otherwise, the
value $A(i - 1, j)$ has already been queried at some point in the past, and
therefore $A(i, j)$ can be obtained from $A(i - 1, j)$ by an incremental query
in time $O(1)$.

Now consider the row fill-in procedure. Here, the elements of matrix $A$
are queried consecutively either by row, or by column. Let $A(i, j)$ be the
current element. The next element to be queried is either $A(i, j + 1)$, or
$A(i + 2, j + 1)$. In both cases, this can be achieved by an incremental query
in time $O(1)$.

Thus, every element query can be performed in time $O(1)$. Therefore,
the overall running time is $O(n)$. ■

Example 6 In Figure 2.2, the incremental queries within the elimination
algorithm in the proof of Lemma 2 are shown by arrows. Note that no
incremental query can cross a vertical dotted line, since every such line
represents an arbitrary number of eliminated columns.

Theorem 6 Let $P$ be an $n \times n$ (sub)permutation matrix. Let $b, c$ be
vectors, such that $P \Box b = c$. Given the nonzeros of $P$ and the full vector $b$,
vector $c$ can be computed in time $O(n)$. ■

Proof By Theorem 5 and Lemma 2.

We now consider matrix-matrix distance multiplication. For generic ma-
trices, direct application of Definition 9 gives an algorithm for matrix dis-
tance multiplication of size $n$, running in time $O(n^3)$. Slightly subcubic
algorithms for this problem have also been obtained. The fastest currently
known algorithm is by Chan [31], running in time $O\left(\frac{n^3(\log \log n)^3}{\log^5 n}\right)$.

For Monge matrices, distance multiplication can be easily performed in
quadratic time.

Theorem 7 Let $A, B, C$ be $n \times n$ matrices, such that $A$ is Monge, and
$A \odot B = C$. Given matrices $A, B$, matrix $C$ can be computed in time and
memory $O(n^2)$. ■

Proof The problem of computing the product $A \odot B = C$ is equivalent to $n$
instances of the matrix-vector product $A \odot b = c$, where $b$ (respectively, $c$) is
a column of $B$ (respectively, $C$). Every one of these instances can be solved
in time $O(n)$ by Theorem 5, so the overall running time (and therefore also memory) is $n \cdot O(n) = O(n^2)$.

When matrices are represented explicitly, the running time in Theorem 7 is clearly optimal. However, for implicit simple unit-Monge matrices, the distance multiplication time can be reduced further. In [121, 124], we gave an algorithm running in time $O(n^{1.5})$. In [127], we improved upon this as follows.

**Theorem 8** Let $P_A$, $P_B$, $P_C$ be $n \times n$ (sub)permutation matrices, such that $P_A \boxtimes P_B = P_C$. Given the nonzeros of $P_A$, $P_B$, the nonzeros of $P_C$ can be computed in time $O(n \log n)$.

**Proof** Let $P_A$, $P_B$, $P_C$ be permutation matrices over $\langle 0 : n \rangle$. The algorithm is defined by recursion on $n$.

Recursion base: $n = 1$. The computation is trivial.

Recursive step: $n > 1$. Assume without loss of generality that $n$ is even. Informally, the idea is to split the range of index $j$ in the definition of matrix distance product (Definition 9) into two subranges of size $\frac{n}{2}$. For each of these subranges of $j$, we use the sparsity of the input permutation matrix $P_A$ (respectively, $P_B$) to partition the range of index $i$ (respectively, $k$) into two disjoint, not necessarily contiguous, subsets of size $\frac{n}{2}$. We then call the algorithm recursively on the two resulting half-sized subproblems, and use the two returned half-sized permutation matrices to reconstruct the output permutation matrix $P_C$, relying on the Monge properties of the respective distribution matrices.

We now describe the recursive step in more detail. We have

$$P_A^\Sigma \odot P_B^\Sigma = P_C^\Sigma$$

Let

$$P_{A,lo} = P_A(\ast | 0 : \frac{n}{2}) \quad P_{B,lo} = P_B(0 : \frac{n}{2} | \ast) \quad P_{A,lo}^\Sigma \odot P_{B,lo}^\Sigma = P_{C,lo}^\Sigma$$

$$P_{A,hi} = P_A(\ast | \frac{n}{2} : n) \quad P_{B,hi} = P_B(\frac{n}{2} : n | \ast) \quad P_{A,hi}^\Sigma \odot P_{B,hi}^\Sigma = P_{C,hi}^\Sigma$$

In the first subproblem, matrices $P_{A,lo}$, $P_{B,lo}$ are rectangular (respectively, $n \times \frac{n}{2}$ and $\frac{n}{2} \times n$) subpermutation matrices, each with $\frac{n}{2}$ nonzeros. Recall from the proof of Theorem 3 that a zero row (respectively, column) in $P_{A,lo}$, $P_{B,lo}$ corresponds to a zero row (respectively, column) in their implicit distance product $P_{C,lo}$. Therefore, we can delete all zero rows and columns from $P_{A,lo}$, $P_{B,lo}$, $P_{C,lo}$, obtaining, after appropriate index remapping, three $\frac{n}{2} \times \frac{n}{2}$ permutation matrices. Consequently, the first subproblem can be solved by first performing a linear-time index remapping (corresponding to the deletion of zero rows and columns from $P_{A,lo}$, $P_{B,lo}$), then making a recursive
call on the resulting half-sized problem, and then performing an inverse index remapping (corresponding to the reinsertion of the zero rows and columns into \( P_{C,lo} \)). The second subproblem can be solved analogously.

Since the nonzeros in the two subproblems have disjoint index ranges, the sum \( P_{C,lo} + P_{C,hi} \) is an \( n \times n \) permutation matrix. We have

\[
P^\Sigma_C(i, k) = \min_{j \in [0:n]} \left( P^\Sigma_A(i, j) + P^\Sigma_B(j, k) \right) = \min \left( \min_{j \in [0, \frac{n}{2}]} \left( P^\Sigma_A(i, j) + P^\Sigma_B(j, k) \right), \min_{j \in [\frac{n}{2}, n]} \left( P^\Sigma_A(i, j) + P^\Sigma_B(j, k) \right) \right)
\]

for all \( i, k \in [0:n] \). The first argument in the above expression can now be rewritten as

\[
\min_{j \in [0, \frac{n}{2}]} \left( P^\Sigma_A(i, j) + P^\Sigma_B(j, k) \right) = \min_{j \in [0, \frac{n}{2}]} \left( P^\Sigma_{A,lo}(i, j) + P^\Sigma_{B,lo}(j, k) + P^\Sigma_{B,hi}(\frac{n}{2}, k) \right) = \min_{j \in [0, \frac{n}{2}]} \left( P^\Sigma_{A,lo}(i, j) + P^\Sigma_{B,lo}(j, k) \right) + P^\Sigma_{B,hi}(\frac{n}{2}, k) = P^\Sigma_{C,lo}(i, k) + P^\Sigma_{C,hi}(0, k)
\]

The second argument can be rewritten analogously, so we have

\[
P^\Sigma_C(i, k) = \min( P^\Sigma_{C,lo}(i, k) + P^\Sigma_{C,hi}(0, k), P^\Sigma_{C,hi}(i, k) + P^\Sigma_{C,lo}(i, n) )
\]

for all \( i, k \in [0:n] \). In order to compute the nonzeros of matrix \( P_C \) efficiently, consider the difference of arguments in the above expression:

\[
\delta(i, k) = (P^\Sigma_{C,lo}(i, k) + P^\Sigma_{C,hi}(0, k)) - (P^\Sigma_{C,hi}(i, k) + P^\Sigma_{C,lo}(i, n)) = (P^\Sigma_{C,hi}(0, k) - P^\Sigma_{C,hi}(i, k)) - (P^\Sigma_{C,lo}(i, n) - P^\Sigma_{C,lo}(i, k)) = \sum_{\bar{i} \in (0:n), \bar{k} \in (0:k)} P_{C,hi}(\bar{i}, \bar{k}) - \sum_{\bar{i} \in (i:n), \bar{k} \in (k:n)} P_{C,lo}(\bar{i}, \bar{k})
\]

Since \( P_{C,lo} \), \( P_{C,hi} \) are subpermutation matrices, and \( P_{C,lo} + P_{C,hi} \) a permutation matrix, it follows that function \( \delta \) is unit-monotone increasing in each of its arguments.

The sign of function \( \delta \) plays an important role in determining the positions of nonzeros in \( P_C \). Let us fix some \( i, k \in (0:n) \), and consider the four values \( \delta(i \pm \frac{1}{2}, k \pm \frac{1}{2}) \). Three cases are possible.

Case \( \delta(i + \frac{1}{2}, k + \frac{1}{2}) \leq 0 \). By monotonicity of \( \delta \), we have \( \delta(i \pm \frac{1}{2}, k \pm \frac{1}{2}) \leq 0 \) for all four sign combinations. Therefore,

\[
P^\Sigma_C(i \pm \frac{1}{2}, k \pm \frac{1}{2}) = P^\Sigma_{C,lo}(i \pm \frac{1}{2}, k \pm \frac{1}{2})
\]
for all four sign combinations chosen consistently on both sides of the equation. Hence, \( P_C(\hat{i}, \hat{k}) = P_{C,lo}(\hat{i}, \hat{k}) \).

Case \( \delta(\hat{i} - \frac{1}{2}, \hat{k} - \frac{1}{2}) \geq 0 \). Symmetrically, we have \( P_C(\hat{i}, \hat{k}) = P_{C,lo}(\hat{i}, \hat{k}) \).

Case \( \delta(\hat{i} - \frac{1}{2}, \hat{k} - \frac{1}{2}) < 0 \) and \( \delta(\hat{i} + \frac{1}{2}, \hat{k} + \frac{1}{2}) > 0 \). By unit-monotonicity of \( \delta \), we have

\[
\delta(\hat{i} - \frac{1}{2}, \hat{k} + \frac{1}{2}) = \delta(\hat{i} + \frac{1}{2}, \hat{k} - \frac{1}{2}) = 0
\]

Therefore,

\[
P_C^S(\hat{i} + \frac{1}{2}, \hat{k} + \frac{1}{2}) = P_{C,lo}^S(i + \frac{1}{2}, \hat{k} + \frac{1}{2}) < P_{C,lo}^S(i + \frac{1}{2}, \hat{k} + \frac{1}{2}) \tag{2.4}
\]

and, furthermore,

\[
P_C^S(i + \frac{1}{2}, \hat{k} + \frac{1}{2}) = P_{C,lo}^S(i + \frac{1}{2}, \hat{k} + \frac{1}{2}) \tag{2.5}
\]

for the remaining three sign combinations chosen consistently on both sides of the equation. By Definitions 1, 2, we have

\[
P_C(i, \hat{k}) = P_C^S(\ldots) - P_C^S(\ldots) - P_C^S(i + \frac{1}{2}, \hat{k} + \frac{1}{2}) + P_C^S(\ldots)
\]

\[
P_{C,lo}(\hat{i}, \hat{k}) = P_{C,lo}^S(\ldots) - P_{C,lo}^S(\ldots) - P_{C,lo}^S(i + \frac{1}{2}, \hat{k} + \frac{1}{2}) + P_{C,lo}^S(\ldots)
\]

By (2.4), we have a strict inequality between the two additive terms fully shown above, and by (2.5), the abbreviated additive terms are pairwise equal between the two expressions. Hence, \( P_C(i, \hat{k}) > P_{C,lo}(\hat{i}, \hat{k}) \). Since both \( P_C \), \( P_{C,lo} \) are zero-one matrices, this implies that \( P_C(i, \hat{k}) = 1 \) and \( P_{C,lo}(\hat{i}, \hat{k}) = 0 \) (symmetrically, also \( P_{C,lo}(i, \hat{k}) = 0 \)).

Summarising the above three cases, we have \( P_C(i, \hat{k}) = 1 \), if and only if one of the following mutually exclusive conditions holds:

\[
P_{C,lo}(\hat{i}, \hat{k}) = 1 \quad \text{and} \quad \delta(\hat{i} + \frac{1}{2}, \hat{k} + \frac{1}{2}) \leq 0 \tag{2.6}
\]

\[
P_{C,lo}(i, \hat{k}) = 1 \quad \text{and} \quad \delta(\hat{i} - \frac{1}{2}, \hat{k} - \frac{1}{2}) \leq 0 \tag{2.7}
\]

\[
\delta(\hat{i} - \frac{1}{2}, \hat{k} - \frac{1}{2}) < 0 \quad \text{and} \quad \delta(\hat{i} + \frac{1}{2}, \hat{k} + \frac{1}{2}) > 0 \tag{2.8}
\]

In order to perform the checks (2.6)–(2.8) efficiently, it is sufficient to find for each \( d \in [-n + 1 : n - 1] \) a value \( r(d) \in [1 : 2n - 1] \), such that \( r(d) + d \) is odd, and

\[
\delta(i, k) \leq 0 \quad \text{if} \quad i + k < r(k - i)
\]

\[
\delta(i, k) \geq 0 \quad \text{if} \quad i + k > r(k - i)
\]

for all \( i, k \in [0 : n] \) (note that the above list of two cases is exhaustive, since \( r(k - i) - (i + k) = r(k - i) + (k - i) - 2k \) must be odd, and therefore \( i + k \neq r(k - i) \)). Such a value \( r(d) \) is guaranteed to exist by the monotonicity of function \( \delta \). Furthermore, values \( r(d) \) can be chosen so that \( |r(d + \frac{1}{2}) -
If this choice, we obtain the new witnesses as
\[ w^-(d) = \delta\left(\frac{r(d)-d-1}{2}, \frac{r(d)+d-1}{2}\right) \in [-1, 0] \]
\[ w^+(d) = \delta\left(\frac{r(d)-d+1}{2}, \frac{r(d)+d+1}{2}\right) \in [0, 1] \]
We call the values \( w^-(d) \), \( w^+(d) \) witnesses for \( r(d) \).

Array \( r \) can be computed efficiently as follows. We loop from \( d = -n+1 \) to \( d = n-1 \). For each \( d \), we obtain the value \( r(d) \) along with its two witnesses.

Initially, we have \( d = -n+1, r(-n+1) = n \); the witnesses \( w^-(n+1) = \delta(n-1, 0) \) and \( w^+(n+1) = \delta(n, 1) \) can be easily computed in time \( O(1) \).

Now assume that for a current value of \( d \), we have the value \( r(d) \), and the witnesses \( w^-(d) \), \( w^+(d) \). Our next goal is to compute \( r(d+1) \), along with its two witnesses. Let
\[ w^* = \delta\left(\frac{r(d)-d-1}{2}, \frac{r(d)+d+1}{2}\right) \in [-1 : 1] \]
Value \( w^* \) can be obtained from either \( w^-(d) \) or \( w^+(d) \) by Theorem 2 in time \( O(1) \). We now let
\[ r(d+1) = r(d) + \begin{cases} 1 & \text{if } w^* \in [-1 : 0] \\ -1 & \text{if } w^* \in [0 : 1] \end{cases} \]
If \( w^* = 0 \), then the choice between 1 and \( -1 \) is made arbitrarily. Following this choice, we obtain the new witnesses as
\[ w^-(d+1) = \begin{cases} \delta\left(\frac{r(d)-d-2}{2}, \frac{r(d)+d-2}{2}\right) & \text{if } w^* \in [-1 : 0] \\ w^* & \text{if } w^* \in [0 : 1] \end{cases} \]
\[ w^+(d+1) = \begin{cases} w^* & \text{if } w^* \in [-1 : 0] \\ \delta\left(\frac{r(d)-d+2}{2}, \frac{r(d)+d+2}{2}\right) & \text{if } w^* \in [0 : 1] \end{cases} \]
In each case, the value for the new witness can be obtained from respectively \( w^-(d) \), \( w^+(d) \) by Theorem 2 in time \( O(1) \). If \( w^* = 0 \), then the choices are made consistently with the arbitrary choice made in the definition of \( r(d+1) \).

The described loop runs until \( d = n-1 \). At this point, we necessarily have \( r(n-1) = n \), \( w^-(n-1) = \delta(0, n-1) \) and \( w^+(n-1) = \delta(1, n) \). The whole loop runs in time \( O(n) \).

Given arrays \( r \), \( w^- \), \( w^+ \), conditions (2.6)–(2.8) can now be expressed as follows:
\[ P_{C,lo}(i, k) = 1 \text{ and } i + k < r(k - i) \quad (2.9) \]
\[ P_{C,hi}(i, \hat{k}) = 1 \text{ and } i + \hat{k} > r(\hat{k} - i) \tag{2.10} \]
\[ i + \hat{k} = r(\hat{k} - i) \text{ and } w^-(\hat{k} - i) = -1 \text{ and } w^+(\hat{k} - i) = 1 \tag{2.11} \]

The nonzeros of \( P_C \) satisfying either of the conditions (2.9), (2.10) can be found in time \( O(n) \) by checking directly each of the nonzeros in matrices \( P_{C,lo} \) and \( P_{C,hi} \). The nonzeros of \( P_C \) satisfying condition (2.11) can be found in time \( O(n) \) by a linear sweep of the values \( r(d) \) for all \( d \in [-n + 1 : n - 1] \). For each \( d \), we let \( \hat{i} = r(d) + \frac{d}{2}, \hat{k} = r(d) - \frac{d}{2} \), and substitute these values into (2.11). We have now obtained all the nonzeros of matrix \( P_C \).

(End of recursive step)

A generalisation to subpermutation matrices is as in Theorem 3.

Time analysis. The recursion tree is a balanced binary tree of height \( \log n \). In the root node, the computation runs in time \( O(n) \). In each subsequent level, the number of nodes doubles, and the running time per node decreases by a factor of 2. Therefore, the overall running time is \( O(n \log n) \).

\begin{example}
Figure 2.3 illustrates the proof of Theorem 8 on a problem instance with a solution generated by the Mathematica 7 software. Subfigure 2.3a shows a pair of input 20 \( \times \) 20 permutation matrices \( P_A, P_B \), with nonzeros indicated by green bullets. Subfigure 2.3b shows the partitioning of the implicit 20 \( \times \) 20 matrix distance multiplication problem into two 10 \( \times \) 10 subproblems. The nonzeros in the two subproblems are shown respectively by red crosses and blue diamonds. Subfigure 2.3c shows a recursive step. The boundaries separating set \( \delta^{-1}([-10 : -1]) \) from \( \delta^{-1}([0]) \), and set \( \delta^{-1}([0]) \) from \( \delta^{-1}([1 : 10]) \), are shown respectively by the red line and the blue line. Function \( r \) corresponds to an arbitrary monotone rectilinear path within \( \delta^{-1}([0]) \), i.e. between the red and the blue lines, inclusive of the boundaries. In particular, either of the boundaries itself can be taken to define \( r \). The nonzeros in the output matrix \( P_C \) satisfying (2.9), (2.10), (2.11) are shown respectively by red crosses, blue diamonds and green bullets; note that overall, there are 20 such nonzeros, and that they define a permutation matrix.
\end{example}

2.4 Bruhat order

Let \( P_A, P_B \) be permutation matrices over \( \langle 0 : n \mid 0 : n \rangle \). A classical algebraic measure of permutation comparison is given by the following partial order.

\begin{definition}
Matrix \( P_A \) is lower than matrix \( P_B \) in the Bruhat order, if \( P_A \) can be transformed to \( P_B \) by a sequence of steps, each step substituting a submatrix of the form \( \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \) by a submatrix of the form \( \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \).
\end{definition}
Figure 2.3: Proof of Theorem 8: $P_A \boxplus P_B = P_C$
Informally, $P_A$ is lower than $P_B$, if $P_A$ defines a “more sorted” permutation than $P_B$. More precisely, $P_A$ is lower than $P_B$, if the permutation defined by $P_B$ can be transformed into the one defined by $P_A$ by a comparison network; this elegant interpretation of the definition was given by Knuth [83].

Bruhat order is a very important group-theoretic concept, which can be generalised to an arbitrary Coxeter group. We refer the reader to the monograph by Björner and Brenti [22] for more details and further references.

Many equivalent definitions of the Bruhat order are known; see e.g. Björner and Brenti [22], Drake et al. [48], Johnson and Nasserat [73]. Probably the simplest one, referred to in [22] as the “dot criterion”, is as follows.

**Theorem 9** Matrix $P_A$ is below matrix $P_B$ in the Bruhat order, if and only if $P_A^\Sigma \leq P_B^\Sigma$ elementwise.

**Proof** Straightforward from the definitions; see [22]. ■

Theorem 9 immediately gives an algorithm for deciding whether two permutation matrices are Bruhat-comparable in time $O(n^2)$. To the author’s knowledge, no asymptotically faster algorithms for deciding Bruhat comparability have been known.

As a first application of our results from Section 2.1, we now give a fast algorithm for Bruhat comparability, improving on the algorithm based on Theorem 9.

**Theorem 10** Matrix $P_A$ is below matrix $P_B$ in the Bruhat order, if and only if $P_A^R \sqcap P_B = Id^R$.

**Proof** Let $P_A$ be below $P_B$. Let $i,j \in [0 : n]$. We have

\[
(P_A^R)^\Sigma(i,j) + P_B^\Sigma(j,n-i) =
\]

\[
(n - i - P_A(j,n-i)) + P_B^\Sigma(j,n-i) =
\]

\[
(P_B(j,n-i) - P_A(j,n-i)) + n - i \geq
\]

\[
n - i
\]

For $j = 0$ (and, symmetrically, $j = n$), this lower bound is attained: we have $(P_A^R)^\Sigma(i,0) + P_B^\Sigma(0,n-i) = 0 + (n-i) = n-i$. Therefore,

\[
(P_A^R \sqcap P_B)^\Sigma(i,n-i) =
\]

\[
\min_j((P_A^R)^\Sigma(i,j) + P_B^\Sigma(j,n-i)) =
\]

\[
n - i
\]

By (2), this implies that $P_A^R \sqcap P_B = Id^R$.

Conversely, $P_A^R \sqcap P_B = Id^R$ implies that for all $i$, $\min_j((P_A^R)^\Sigma(i,j) + P_B^\Sigma(j,n-i)) = n - i$. From the above, this is equivalent to $P_B^\Sigma(j,n-i) - P_A^\Sigma(j,n-i) \geq 0$, therefore $P_A^\Sigma(j,n-i) \leq P_B^\Sigma(j,n-i)$ for all $i,j$. By Theorem 9, this implies that $P_A$ is below $P_B$ in the Bruhat order. ■
In combination with Theorem 8, Theorem 10 gives an algorithm for deciding Bruhat comparability in time $O(n \log n)$. 
Chapter 3

Semi-local string comparison

3.1 Semi-local LCS

We will consider strings of characters taken from an alphabet. No a priori
assumptions are made on the size of the alphabet and on the model of com-
putation; we will make specific assumptions in different contexts (e.g. a fixed
finite alphabet with only equality comparisons, or an alphabet of integers
up to a given $n$ with standard arithmetic operations, etc.) Two alphabet
characters $\alpha, \beta$ match, if $\alpha = \beta$, and mismatch otherwise. In addition to
alphabet characters, we introduce the special non-alphabet character ‘$\$$’,
which only matches itself and does not match any alphabet characters, and
the wildcard character ‘$\diamond$’, which matches both itself and all the alphabet
characters.

It will be convenient to index strings by odd half-integer, rather than
integer indices, e.g. string $a = \alpha_{\frac{1}{2}}\alpha_{\frac{3}{2}}\ldots\alpha_{\frac{m-1}{2}}$. We will index strings as
vectors, writing e.g. $a(i) = \alpha_i, a(i:j) = \alpha_{i+\frac{1}{2}}\ldots\alpha_{j-\frac{1}{2}}$. String concatenation
will be denoted by juxtaposition.

Given a string, we distinguish between its contiguous substrings, and
not necessarily contiguous subsequences. Special cases of a substring are
a prefix and a suffix of a string. Given a string $a$ of length $m$, we use the
take/drop notation of [130] for prefixes and suffixes of $a$:

\[
\begin{align*}
a \uparrow k &= a(0 : k) & a \downarrow k &= a(k : m) \\
a \uparrow k &= a(m - k : m) & a \downarrow k &= a(0 : m - k)
\end{align*}
\]

Unless indicated otherwise, our algorithms will take as input a string $a$ of
length $m$, and a string $b$ of length $n$.

A classical approach to string comparison is based on the following nu-
merical measure of string similarity.

**Definition 13** Given strings $a, b$, the longest common subsequence (LCS)
problem asks for the length of the longest string that is a subsequence of
both $a$ and $b$. We will call this length the LCS score of strings $a, b$.  

\[
\text{LCS}\text{ score of } a, b
\]

31
Example 8  The LCS score of strings $a = \text{“baabcba”}$, $b = \text{“baabcabcabaca”}$ is 8; string $b$ contains the whole string $a$ as a subsequence. The LCS score of string $a$ against substring $b(4:11) = \text{“cabcba”}$ is 5; this score is realised by a common subsequence “abcba”. This example, which will serve as a running example for this chapter, is borrowed from Alves et al. [7].

The best known algorithms for the LCS problem run within (model-dependent) polylogarithmic factors of $O(mn)$. We will recall the necessary background on LCS algorithms in Sections 4.1, 4.2, 8.3.

A simple special case of the LCS problem is the (global) subsequence recognition problem (also known as the “subsequence matching problem”). Given a text string $t$ of length $n$ and a pattern string $p$ of length $m \leq n$, the problem asks whether $t$ contains the whole $p$ as a subsequence. This is equivalent to asking whether the LCS score of $t$ against $p$ is exactly $m$. The global subsequence recognition problem has been considered e.g. by Aho et al. [3, Section 9.3], who describe a straightforward algorithm running in time $O(n)$. Various extensions of this problem have been explored by Crochemore et al. [41].

In this work, we generalise the LCS problem to provide a more detailed measure of string similarity.

Definition 14 Given strings $a$, $b$, the semi-local LCS problem asks for the LCS scores as follows:

- $a$ against every substring of $b$ (the string-substring LCS scores);
- every prefix of $a$ against every suffix of $b$ (the prefix-suffix LCS scores);
- symmetrically, the substring-string LCS scores and the suffix-prefix LCS scores, defined as above but with the roles of $a$ and $b$ exchanged.

A traditional distinction, especially in computational biology, is between global (full string against full string) and local (all substrings against all substrings) comparison. Semi-local comparison lies between these two extremes, which explains the terminology. A common alternative term used in biological texts is “end-free comparison”, see e.g. [63, Subsection 11.6.4]. It turns out that this is a very natural and useful type of string comparison.

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, Definition 14 asks for all the locally optimal comparison scores. This approach is more flexible, and will be useful for various applications described later in this work.

It turns out that, although more general than the LCS problem, the semi-local LCS problem can also be solved within (model-dependent) polylogarithmic factors of $O(mn)$. We will consider semi-local LCS algorithms on plain strings in Sections 4.1, 4.2, and on compressed strings in Section 8.3.
A special case of the semi-local LCS problem is the local subsequence recognition problem, which, given a text \( t \) and a pattern \( p \), asks for the substrings in \( t \) containing \( p \) as a subsequence. This problem can also be regarded as a basic form of approximate pattern matching. We will consider algorithms for local subsequence recognition and other types of approximate pattern matching on plain strings in Section 5.2, and on compressed strings in Sections 8.4–8.5.

In certain contexts, such as when one of the input strings is very long, we may not wish to deal with all the substrings of the longer string, but still to consider the other three components of the semi-local LCS problem.

**Definition 15** Given strings \( a, b \), the three-way semi-local LCS problem asks for the string-substring, prefix-suffix and suffix-prefix LCS scores as in Definition 14, but excludes the substring-string LCS scores. The same term three-way semi-local LCS problem will also refer to the symmetric version of the problem, that asks for the prefix-suffix, suffix-prefix and substring-string LCS scores, but excludes the string-substring LCS scores.

We will occasionally use the term full semi-local LCS to distinguish the standard four-way semi-local LCS problem from its restricted three-way version.

### 3.2 Alignment dags and score matrices

It is well-known that an instance of the LCS problem can be represented by a dag (directed acyclic graph) on a rectangular grid of nodes, where character matches correspond to edges scoring 1, and mismatches to edges scoring 0.

**Definition 16** An alignment dag is a weighted dag, defined on the set of nodes \( v_{l,i} \), \( l \in \langle l_0 : l_1 \rangle \), \( i \in \langle i_0 : i_1 \rangle \). The edge and path weights are called scores. For all \( l \in \langle l_0 : l_1 \rangle \), \( \hat{l} \in \langle l_0 : l_1 \rangle \), \( i \in \langle i_0 : i_1 \rangle \), \( \hat{i} \in \langle i_0 : i_1 \rangle \), the alignment dag contains:

- the horizontal edge \( v_{l,i-\frac{1}{2}} \rightarrow v_{l,i+\frac{1}{2}} \) and the vertical edge \( v_{l-\frac{1}{2},i} \rightarrow v_{l+\frac{1}{2},i} \), both with score 0;

- the diagonal edge \( v_{l-\frac{1}{2},i-\frac{1}{2}} \rightarrow v_{l+\frac{1}{2},i+\frac{1}{2}} \) with score either 0 or 1.

An alignment dag can be viewed as an \( (l_1 - l_0) \times (i_1 - i_0) \) grid of cells. An instance of the semi-local LCS problem on strings \( a, b \) corresponds to an \( m \times n \) alignment dag \( G_{a,b} \); a cell indexed by \( \hat{l} \in \langle 0 : m \rangle \), \( \hat{i} \in \langle 0 : n \rangle \) is called a match cell, if \( a(\hat{l}) \) matches \( b(\hat{i}) \), and a mismatch cell otherwise (recall that the strings may contain wildcard characters). The diagonal edges in match cells have score 1, and in mismatch cells score 0. Clearly, the diagonal edges with score 0 do not affect maximum node-to-node scores, and can therefore be ignored.
Particular examples of an alignment dag are the full-mismatch dag and the full-match dag, which consist entirely of mismatch or match cells, respectively. The dag $G_{a,b}$ is the full-mismatch dag, when the strings $a$, $b$ have no characters in common. The dag $G_{a,b}$ is the full-match dag, when both strings consist of a single repeated character, or when one of the strings consists entirely of wildcard characters.

Given an instance of the LCS problem on strings $a$, $b$, common string-substring, suffix-prefix, prefix-suffix, and substring-string subsequences correspond, respectively, to paths of the following form in the alignment dag $G_{a,b}$:

$$v_{0,i} \leadsto v_{m,i'} \quad v_{l,0} \leadsto v_{m,i'} \quad v_{0,i} \leadsto v_{i',n} \quad v_{l,0} \leadsto v_{i',n}$$  \hspace{1cm} (3.1)

where $l,l' \in [0 : m]$, $i,i' \in [0 : n]$. The length of each subsequence is equal to the total score of its corresponding path. The semi-local LCS problem is equivalent to the problem of finding the highest-scoring paths for each of the four path types (3.1) and each possible pair of endpoints on the boundary of the alignment dag.

**Example 9** Figure 3.1 shows the alignment dag for strings $a = \text{"baabcbca"}$, $b = \text{"baabcabcabaca"}$. The highlighted path of score 5 corresponds to the string-substring LCS score for string $a$ against substring $b\langle 4 : 11 \rangle = \text{"cabcaba"}$.

Finding the highest-scoring paths is 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, assigning lengths 0 and 2 to diagonal edges in match and mismatch cells respectively, and ignoring edge directions. Thus, the problem is equivalent to the problem of finding distances between boundary nodes and all nodes on a special case of a weighted undirected planar graph. This problem has been previously studied by Schmidt [117] on real-weighted alignment dags, and by Klein [81].
and Cabello and Chambers [27] on general real-weighted undirected planar graphs. In contrast with these approaches, we exploit both the special structure of the alignment dag, and the discreteness of the weights.

The analysis of the four different path types (3.1) can be simplified by padding one of the input strings with wildcard characters. Accordingly, we need to consider an extended alignment dag for string $a$ over $\langle 0 : m \rangle$ against string $b \circ^m$ over $\langle -m : m + n \rangle$.

**Definition 17** Given strings $a, b$, the corresponding semi-local score matrix $H_{a,b}$ is a matrix over $[-m : n | 0 : m + n]$, defined by

$$H_{a,b}(i,j) = \max \text{score}(v_0,i \leadsto v_m,j)$$

where $i \in [-m : n], j \in [0 : m + n]$, and the maximum is taken across all paths between the given endpoints in the $m \times (2m + n)$ alignment dag $G_{a,\circ^m b \circ^m}$. If $i = j$, we have $H_{a,b}(i,j) = 0$. By convention, if $j < i$, then we let $H_{a,b}(i,j) = j - i < 0$.

**Example 10** Figure 3.2 shows the matrix $H_{a,b}$, giving all semi-local LCS scores for strings $a, b$ as in the previous examples. The entry $H_{a,b}(4,11) = 5$ is circled.

The solution for each of the four components (3.1) of the semi-local LCS problem can now be obtained from the semi-local score matrix $H_{a,b}$ as follows:

$$\max \text{score}(v_0,j \leadsto v_{m,j'}) = H_{a,b}(j,j')$$

$$\max \text{score}(v_i,0 \leadsto v_{m,j'}) = H_{a,b}(-i,j') - i$$

$$\max \text{score}(v_0,j \leadsto v_{i',n}) = H_{a,b}(j,m + n - i') - m + i'$$

$$\max \text{score}(v_i,0 \leadsto v_{i',n}) = H_{a,b}(-i,m + n - i') - m - i + i'$$

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

Special properties of semi-local score matrices have been extensively used in algorithm design. These properties are captured by the following theorem.

**Theorem 11** Given strings $a, b$, the corresponding semi-local score matrix $H_{a,b}$ is unit-anti-Monge. In particular, we have

$$H_{a,b}(i,j) = j - i - \rho_{a,b}^*(i,j)$$

---

1These matrices are called “DIST matrices” e.g. in [117, 39], and “score matrices” in [122]. Our current terminology is chosen to reflect the semi-local score-maximising nature of the matrix elements, while avoiding confusion with pairwise substitution score matrices used in comparative genomics (see e.g. [74]).
Figure 3.2: Matrices $H_{a,b}$ and $P_{a,b}$

where $P_{a,b}$ is a permutation matrix over $\langle -m : n \mid 0 : m + n \rangle$. We also have

$$H_{a,b}(i, j) = m - (P_{a,b}^T)^\Sigma(i, j)$$

(note the matrix transposition and the exchange of indices). In particular, string $a$ is a subsequence of substring $b(i : j)$ for some $i, j \in [0 : n]$, if and only if $(P_{a,b}^T)^\Sigma(i, j) = 0$. □

**Proof** Let $i \in \langle -m : n \rangle$, $j \in (0 : m + n)$. Any crossing pair of paths $v_{0,i+\frac{1}{2}} \leadsto v_{m,j-\frac{1}{2}}$ and $v_{0,i-\frac{1}{2}} \leadsto v_{m,j+\frac{1}{2}}$ can be rearranged into a non-crossing pair of paths $v_{0,i-\frac{1}{2}} \leadsto v_{m,j-\frac{1}{2}}$ and $v_{0,i+\frac{1}{2}} \leadsto v_{m,j+\frac{1}{2}}$ of the same total score.

Therefore, we have $H_{a,b}(i, j) \leq 0$, hence matrix $H_{a,b}$ is anti-Monge.

Let $Q_{a,b}(i, j) = j - i - H_{a,b}(i, j)$. From the above, matrix $Q_{a,b}$ is Monge. Furthermore, we have

$$Q_{a,b}(i-\frac{1}{2}, m+n) - Q_{a,b}(i+\frac{1}{2}, m+n) =$$

$$1 - H_{a,b}(i-\frac{1}{2}, m+n) + H_{a,b}(i+\frac{1}{2}, m+n) = 1 - m + m = 1$$

and

$$Q_{a,b}(-m, j+\frac{1}{2}) - Q_{a,b}(-m, j-\frac{1}{2}) =$$

$$1 - H_{a,b}(-m, j+\frac{1}{2}) + H_{a,b}(-m, j-\frac{1}{2}) = 1 - m + m = 1$$

36
Together, the above properties imply that $Q_{a,b}$ is a simple unit-Monge matrix. Therefore, $Q_{a,b} = \theta_{a,b}$, where $P_{a,b}$ is a permutation matrix.

The second part of the theorem statement is now straightforward from the definitions.

Intuitively, Theorem 11 can be interpreted as follows. Let $a'$ be a substring of $a$, and consider its LCS score against string $b$. If substring $a'$ is extended at either end by one character, the LCS score may stay constant, if the new character cannot be usefully matched to a character of $b$; the LCS score may also increase by 1, if a useful match can be found. Starting from an empty substring $a'$, any new match is clearly useful. The unit-anti-Monge property of matrix $H_{a,b}$ describes the fact that, as substring $a'$ grows in size, not all the new matches can be used simultaneously: some useful matches will become useless, but a useless match will always remain to be so. The “critical points” at which useful matches become useless correspond to the nonzeros of the permutation matrix $P_{a,b}$. However, it should be noted that this is only general intuition: in fact, the notion of a useful match is not absolute, but depends on the choice of a particular highest-scoring path through the alignment dag.

The proof of Theorem 11 holds more generally for any alignment dag with an arbitrary mix of match and mismatch cells, not necessarily defined by the matching between any particular pair of strings. However, the given form of the theorem will be sufficient for the rest of this work.

The key idea of our approach is to view Theorem 11 as a description of an implicit solution to the semi-local LCS problem. The semi-local score matrix $H_{a,b}$ is represented implicitly by the nonzeros of the permutation matrix $P_{a,b}$.

**Definition 18** Given strings $a, b$, the semi-local seaweed matrix is a permutation matrix $P_{a,b}$ over $\langle -m : n \mid 0 : m + n \rangle$, defined by Theorem 11.

Definition 18 leads to the following interpretation of Theorem 11: the LCS score for string $a$ against substring $b(i : j)$ is determined by the number of nonzeros in the semi-local seaweed matrix $P_{a,b}$, that are $\leq$-dominated by (respectively, that $\leq$-dominate) the point $(i, j)$.

**Example 11** Figure 3.2 shows the unit-anti-Monge property of matrix $H_{a,b}$ by a coloured grid pattern, where the red (respectively, blue) lines separate matrix elements that differ by 1 (respectively, by 0). The nonzeros of the semi-local seaweed matrix $P_{a,b}$ over $\langle -8 : 13 \mid 0 : 8 + 13 \rangle$ are shown by green bullets.

The nonzeros of $P_{a,b}$ that are $\leq$-dominated by the point $(4, 11)$ correspond to the green bullets lying below and to the left of the circled entry. Note that there are exactly two such nonzeros, and that $H_{a,b}(4, 11) = 11 - 4 - 2 = 5$. The nonzeros of $P_{a,b}$ that $\leq$-dominate the point $(4, 11)$
A semi-local seaweed matrix can be naturally identified with a seaweed braid on $m + n$ seaweeds. As we show in the following section, divide-and-conquer solutions to the semi-local LCS problem generally correspond to implicit distance multiplication of seaweed matrices, and therefore also to multiplication of the corresponding seaweed braids.

**Example 12** Figure 3.3 shows matrix $P_{a,b}$ as a seaweed braid, laid out directly on the alignment dag $G_{a,b}$. The nonzeros correspond to seaweeds, laid out as paths in the dual graph. In particular, every nonzero $P_{a,b}(\hat{i}, \hat{j}) = 1$, where $\hat{i}, \hat{j} \in \langle 0 : 13 \rangle$, is represented by a seaweed originating between the nodes $v_{0,\hat{i} - \frac{1}{2}}$ and $v_{0,\hat{i} + \frac{1}{2}}$, and terminating between the nodes $v_{8,\hat{j} - \frac{1}{2}}$ and $v_{8,\hat{j} + \frac{1}{2}}$. The remaining seaweeds, originating or terminating at the sides of the dag, correspond to nonzeros $P_{a,b}(\hat{i}, \hat{j}) = 1$, where either $\hat{i} \in \langle -8 : 0 \rangle$ or $\hat{j} \in \langle 13 : 8 + 13 \rangle$ (or both). In particular, every nonzero $P_{a,b}(\hat{i}, \hat{j}) = 1$, where $\hat{i} \in \langle -8 : 0 \rangle$ (respectively, $\hat{j} \in \langle 13 : 8 + 13 \rangle$) is represented by a seaweed originating between the nodes $v_{-\hat{i} - \frac{1}{2}, 0}$ and $v_{-\hat{i} + \frac{1}{2}, 0}$ (respectively, terminating between the nodes $v_{8+13-\hat{j} - \frac{1}{2}, 13}$ and $v_{8+13-\hat{j} + \frac{1}{2}, 13}$). For the purposes of this example, 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 chapter.

The full set of $8 + 13 = 21$ nonzeros in Figure 3.2 corresponds to the full set of 21 seaweeds in Figure 3.3. The two nonzeros that are $\preceq$-dominated by the point $(4, 11)$ correspond to the two seaweeds fitting completely between the two dashed vertical lines $i = 4$ and $j = 11$. The three nonzeros that $\preceq$-dominate the point $(4, 11)$ correspond to the three seaweeds piercing both these lines. \qed
With minimal modification, the definitions of score and seaweed matrices can also be applied separately to each component of the semi-local LCS problem. For that, we split up the definitions as follows.

**Definition 19** Given strings $a$, $b$, the corresponding string-substring, prefix-suffix, suffix-prefix and substring-string score matrices are respectively the following submatrices of the semi-local score matrix $H_{a,b}$ over $[-m:n \mid 0 : m+n]$:

\[
H^\text{B}_{a,b} = H_{a,b}[0 : n \mid 0 : n] \quad H^\text{G}_{a,b} = H_{a,b}[0 : n \mid n : m+n]
\]
\[
H^\text{B}_{a,b} = H_{a,b}[-m : 0 \mid 0 : n] \quad H^\text{G}_{a,b} = H_{a,b}[-m : 0 \mid n : m+n]
\]

Note that the ranges of the four score submatrices in Definition 19 are not disjoint: they overlap along the boundaries corresponding to $i = 0$ and $j = n$. In particular, the entry $H_{a,b}(0,n)$, which gives the LCS score of the whole strings $a$ and $b$, belongs to all four submatrices. The ranges of the four seaweed submatrices in Definition 20 are disjoint.

**Example 13** Figure 3.2 shows the partition of $H_{a,b}$ and $P_{a,b}$ in Definitions 19, 20 by thin dotted lines. The string-substring, prefix-suffix, suffix-prefix and substring-string submatrices are respectively on the bottom-left, bottom-right, top-left and top-right. The elements of $H_{a,b}$ lying directly on the dotted lines are shared by the bordering submatrices. Note that the substring-string submatrices $H^\text{B}_{a,b}$ and $H^\text{G}_{a,b}$ are both trivial; this is due to the fact that the whole string $a$ is a subsequence of $b$.

The nonzeros of each matrix introduced in Definition 20 can be regarded as an implicit solution to the corresponding component of the semi-local LCS problem. Similarly, the first three and the last three matrices can serve as an implicit solution to the respective versions of the three-way semi-local LCS problem. The combined number of nonzeros in the first three matrices is at least $n$ and at most $\min(m+n, 2n)$; note that for $m \geq n$, this number is independent of $m$. Analogously, for $m \leq n$, the combined number of nonzeros in the last three matrices is independent of $n$.

Both the full semi-local seaweed matrices, and the component seaweed matrices introduced in Definition 20, can be processed into an efficient data structure of Theorem 1 for answering individual semi-local score queries.
The definitions of score and seaweed matrices are not symmetric with respect to the order of the input strings. The precise relationship between score matrices $H_{a,b}$, $H_{b,a}$, as well as between their seaweed counterparts, is given by the following lemma.

**Lemma 3** Given input strings $a$, $b$, we have

$$H_{b,a}(i,j) = H_{a,b}(-i, m + n - j) - i + j - n$$

for all $i \in [0 : n]$, $j \in [0 : m]$, and

$$P_{b,a}(i,j) = P_{a,b}(-i, m + n - j)$$

for all $i \in \langle 0 : n \rangle$, $j \in \langle 0 : m \rangle$. □

**Proof** Straightforward by definitions.

### 3.3 Score matrix composition

We now describe how the previously introduced techniques can be applied within a divide-and-conquer framework.

Let $a'$, $a''$, $b'$, $b''$ be nonempty strings of length $m', m'', n'$, $n''$ respectively. We will consider the comparison of a concatenation string $a = a'a''$ of length $m = m' + m''$ against a fixed string $b$ of length $n$; symmetrically, the definitions and results will also apply to the comparison of a fixed string $a$ of length $m$ against a concatenation string $b = b'b''$ of length $n = n' + n''$.

Given a concatenation string $a = a'a''$, a substring $a(i' : i'')$ with $i' \in [0 : m' - 1]$, $i'' \in [m' + 1 : m]$ will be called a cross-substring. In other words, a cross-substring of $a$ is a concatenation of a nonempty suffix of $a'$ and a nonempty prefix of $a''$. A cross-substring that is a prefix or a suffix of $a$ will be called a cross-prefix and a cross-suffix, respectively.

**Definition 21** Given strings $a = a'a''$ and $b$, the corresponding seaweed cross-matrix is the subpermutation matrix

$$P_{a',a'',b} = P_{a,b}(-m' : n | 0 : m'' + n)$$

Symmetrically, given strings $a$ and $b = b'b''$, the corresponding seaweed cross-matrix is the subpermutation matrix

$$P_{a,b',b''} = P_{a,b}(-m : n' | n' : m + n)$$

In contrast with the seaweed matrices in Definition 20, the dimensions of the seaweed cross-matrices introduced by Definition 21 depend on the lengths of individual component strings in the concatenation ($a'$, $a''$ or $b'$, $b''$, respectively). Hence, the notation for these matrices involves three, rather than two, string subscripts. There is a slight abuse of terminology in that
it does not distinguish directly the two symmetric versions of the definition; however, the versions are distinguished by the notation, and will always be clear from the context. Note that \( P_{a,b} \) is a submatrix of \( P_{a',a'',b'} \) and \( P_{a,b} \) is a submatrix of \( P_{a',b''} \).

The alignment dag \( G_{a,b} \) consists of alignment subdags \( G_{a',b}, G_{a'',b} \), sharing a horizontal row of \( n \) nodes and \( n - 1 \) edges, which is simultaneously the bottom row of \( G_{a',b} \) and the top row of \( G_{a'',b} \). We will say that dag \( G_{a,b} \) is the composite of dags \( G_{a',b} \) and \( G_{a'',b} \).

Our goal is, given the semi-local score matrices \( H_{a',b}, H_{a'',b} \), to compute matrix \( H_{a,b} \) efficiently. We call this procedure score matrix composition.

By Theorem 11, score matrices \( H_{a',b}, H_{a'',b}, H_{a,b} \) can be represented implicitly by seaweed matrices \( P_{a',b}, P_{a'',b}, P_{a,b} \). We argue that score matrix composition corresponds generally to implicit distance multiplication of these seaweed matrices.

**Theorem 12** We have

\[
P_{a,b} = (I_{m'} ⊔ P_{a',b}) □ (P_{a'',b} ⊔ I_{m''})
\]

where

\[
I_{m'} = \text{Id}_{m'}(-m : -m' | -m'' : 0)
\]
\[
I_{m''} = \text{Id}_{m''}(m : m' + n | m'' : m + n)
\]

We also have

\[
P_{a,b} = P_{a',b} □ P_{a'',b}
\]
\[
P_{a',a'',b} = (P_{a',b} ⊔ P_{a'',b}) □ (P_{a',b} ⊔ P_{a'',b})
\]

**Proof** By Theorem 11, we have

\[
H_{a,b}(i,k) = k - i - P_{a,b}^\Sigma(i,k)
\]

for all \( i ∈ [-m : n], k ∈ [0 : m + n] \). Three cases are possible, based on the partitioning of the index ranges.

**Case** \( i ∈ [-m' : n], k ∈ [0 : m+n] \). By Definition 17 and Theorem 11, we have

\[
H_{a,b}(i,k) = \max_{j ∈ [0:n]} (H_{a',b}(i,j) + H_{a'',b}(j,k)) =
\]
\[
\max_{j ∈ [0:n]} (j - i - P_{a',b}^\Sigma(i,j) + k - j - P_{a'',b}^\Sigma(j,k)) =
\]
\[
k - i - \min_{j ∈ [0:n]} (P_{a',b}^\Sigma(i,j) + P_{a'',b}^\Sigma(j,k))
\]

41
Figure 3.4: Semi-local score matrix composition: \( P_{a',b} \otimes P_{a'',b} = P_{a,b} \)
Therefore,

\[ P_{a,b}(i,k) = \min_{j \in [0:n]} \left( P_{a',b}(i,j) + P_{a'',b}(j,k) \right) = (P_{a',b} \odot P_{a'',b})(i,k) \]

In particular, this holds for \( i, k \in [0 : n] \). Hence, we have

\[ P_{a,b} = P_{a',b} \odot P_{a'',b} \]

Case \( i \in [-m : -m'], k \in [0 : m + n] \). We have

\[ H_{a,b}(i,k) = m' + H_{a'',b}(i + m', k) = m' + k - (i + m') - P_{a'',b}(i + m', k) = k - i - P_{a'',b}(i + m', k) \]

Therefore,

\[ P_{a,b}(i,k) = P_{a'',b}(i + m', k) = (I_{m'} \boxdot P_{a'',b})(i,k) \]

Case \( i \in [-m : n], k \in [m'' + n : m + n] \). Symmetrically to the previous case, we have

\[ P_{a,b}(i,k) = (P_{a',b} \boxdot I_{m''})(i,k) \]

Summarising the above three cases, we have the proof of (3.2). From the first case, we also have the proof of (3.3). The proof of (3.4) is similar. ■

**Example 14** Figure 3.4 shows an instance of semi-local score matrix composition represented by seaweed matrices. The nonzeros in the matrices are given by seaweed braids (using an arbitrary layout of individual seaweeds). Subfigure 3.4a shows the input matrices \( P_{a',b}, P_{a'',b} \). Additionally, it shows the auxiliary matrices \( I_{m'}, I_{m''} \) by dotted seaweeds. Subfigure 3.4c shows the output matrix \( P_{a,b} \).

Let \( n^* = \min(m', m'', n) \) and \( m^* = \min(m, n', n'') \).

**Theorem 13** The implicit distance product (3.2) can be computed in time \( O(m + n \log n^*) \). Each of the implicit distance products (3.3), (3.4) can be computed in time \( O(n \log n^*) \).

**Proof** First, we consider the implicit distance product (3.2). If \( n^* = n \), then we compute the product directly by Theorem 8.

We may now assume without loss of generality that \( n^* = m'' \), and that \( n^*/m'' \geq 1 \) is an integer. It will be convenient to express the proof in terms of seaweed braids. Consider a layout of the seaweed braid corresponding to \( P_{a''}, b \) within the \( m'' \times n \) alignment dag \( G_{a'',b} \). Let us partition this dag into \( n/m'' \) square blocks of size \( m'' \). Clearly, every boundary between two successive
blocks is crossed by at most $m''$ seaweeds. This property is independent of any particular seaweed layout. Given the seaweed matrix $P_{a',b}$, it is straightforward to lay out the seaweed braids between the block boundaries within each block in time $O(m'')$ per block, and therefore in overall time $\frac{n}{m'} \cdot O(m'') = O(n)$. Note that this is done with only the matrix $P_{a',b}$ as input, and therefore the resulting layout may not correspond to the original alignment dag. The seaweed braid of size $m'' + n$, corresponding to $P_{a',b}$, is now decomposed into $\frac{n}{m'}$ “staggered” seaweed subbraids, each of size $2m''$.

Recall that by Theorem 12, we have

$$P_{a,b} = (I_{m'} \sqcup P_{a',b}) \boxtimes (P_{a'',b} \sqcup I_{m''})$$

We compute this product by first performing $\frac{n}{m'}$ successive seaweed braid multiplications of size $m''$, one in each block of $G_{a',b}$. Every one of these multiplications can be performed by Theorem 8 in time $O(m'' \log m'')$. After that, we perform a multiplication by the identity subbraid of size $m'$, corresponding to the offset identity matrix $I_{m''}$. This can be done in time $O(m') = O(m)$. The overall running time is $O(m + \frac{n}{m'} \cdot m'' \log m'') = O(m + n \log m'')$.

The proof for (3.3), (3.4) is analogous. ■

Example 15 Figure 3.4 illustrates the proof of Theorem 13 as follows. Decomposition of the seaweed braid corresponding to matrix $P_{a',b}$ into blocks is shown in Subfigure 3.4a by thin dotted lines. Subfigure 3.4b shows the blocks of $P_{a',b}$ in “staggered” form. Clearly, the whole seaweed braid for $P_{a,b}$ can be obtained from $P_{a',b}$ by successive composition with staggered block subbraids of $P_{a'',b}$, block-by-block. Subfigure 3.4c shows the resulting seaweed braid for the composition matrix $P_{a,b}$.

Theorems 12, 13 give a divide-and-conquer solution to the string-substring and substring-string LCS problems, which we formulate as follows.

**Corollary 1** Given the nonzeros of matrices $P_{a',b}$, $P_{a'',b}$, it is possible to compute the nonzeros of matrix $P_{a,b}$ in time $O(n \log n^*)$. Symmetrically, given the nonzeros of matrices $P_{a',b}$, $P_{a'',b}$, it is possible to compute the nonzeros of matrix $P_{a,b}$ in time $O(m \log m^*)$.

**Proof** The first claim is directly by Theorems 12, 13. The symmetric claim follows by Lemma 3. ■

Similarly, we obtain a divide-and-conquer solution to both symmetric versions of the three-way semi-local LCS problem.

**Corollary 2** Given the nonzeros of matrices

$$P_{a',b}, P_{a',b}, P_{a',b}, P_{a',b}, P_{a',b}, P_{a',b}$$

44
it is possible to compute the nonzeros of matrices
\[ P_{a,b}, P_{a,b}', P_{a,b}'', P_{a',a'',b} \]
in time \( O(n \log n^*) \).

Symmetrically, given the nonzeros of matrices
\[ P_{a,b'}, P_{a,b}', P_{a,b}'', P_{a,b''}, P_{a,b'''} \]
it is possible to compute the nonzeros of matrices
\[ P_{a,b}, P_{a,b'}, P_{a,b}'', P_{a',b'''} \]
in time \( O(m \log m^*) \).

**Proof** To prove the first claim, we compute \( P_{a,b}, P_{a,b}' \) and \( P_{a',a'',b} \) in time \( O(n \log n^*) \) by Theorems 12, 13. We now have
\[
P_{a,b} = (P_{a,b} \boxplus I_{m''}) \uplus P_{a',a'',b}(0 : n | n : m'' + n)
P_{a,b}' = P_{a',a'',b}(-m' : 0 | 0 : n) \uplus (I_{m'} \boxplus P_{a,b}^*)
\]
Both these expressions can be computed trivially in time \( O(n) \). The overall running time is \( O(n \log n^*) \).

The symmetric claim follows by Lemma 3.

While the cross-matrices in the statement of Corollary 2 do not contribute to the solution of the three-way semi-local LCS problem for \( a \) against \( b \), they are included for the sake of other applications, described in subsequent chapters.

Finally, we obtain a divide-and-conquer solution to the full semi-local LCS problem.

**Corollary 3** Given the nonzeros of matrices \( P_{a',b}, P_{a'',b} \), it is possible to compute the nonzeros of matrix \( P_{a,b} \) in time \( O(m+n \log n^*) \). Symmetrically, given the nonzeros of matrices \( P_{a,b'}, P_{a,b}'', P_{a,b'''} \), it is possible to compute the nonzeros of matrix \( P_{a,b} \) in time \( O(m \log m^* + n) \).

**Proof** The first claim is directly by Theorems 12, 13. The symmetric claim follows by Lemma 3.
Chapter 4

The seaweed method

4.1 The seaweed algorithm

A classical solution to the global LCS problem is given by the dynamic programming algorithm, discovered independently by Needleman and Wunsch (without an explicit analysis) [104], and by Wagner and Fischer [129]. This algorithm assumes a character comparison model that only allows comparison outcomes “equal” and “unequal”, and the unit-cost RAM computation model. The algorithm runs in time $O(mn)$. As a byproduct, it solves the LCS problem for all prefixes of input string $a$ against all prefixes of input string $b$.

A naive algorithm for the semi-local LCS problem runs in time $O((m + n)^4)$. Based on the ideas of Schmidt [117], Alves et al. [7] gave an algorithm for the string-substring LCS problem that runs in time $O(mn)$, which therefore extends the functionality of the standard dynamic programming LCS algorithm, while matching its asymptotic running time. As a byproduct, the algorithm solves the string-substring LCS problem for all prefixes of $a$ against the whole $b$.

We now give a simple algorithm for the semi-local LCS problem, which further improves on the functionality of the above algorithms, while still matching their model assumptions and asymptotic running time. We call it the seaweed algorithm, since it has a simple interpretation in terms of seaweed braids introduced in Chapter 1.

Algorithm 1 (Semi-local LCS: The seaweed algorithm)

**Input:** strings $a$, $b$ of length $m$, $n$, respectively.

**Output:** nonzeros of semi-local seaweed matrix $P_{a,b}$.

**Description.** It will be convenient to express the algorithm in terms of seaweed braids. Informally, the algorithm works as follows. We construct a seaweed braid on $m + n$ seaweeds, laid out on the alignment dag $G_{a,b}$. Every
seaweed is traced across the alignment dag in the top-to-bottom or left-to-right direction. A seaweed runs in a straight horizontal or vertical line by default; however, its direction may be affected either by passing through a match cell, or by meeting another seaweed. Every cell has two seaweeds passing through it, entering the cell at the top and the left, and leaving it at the bottom and the right, not necessarily in that order.

The layout of the two seaweeds within a cell is decided as follows. In a match cell, both seaweeds “bend away” from each other, so the seaweed entering at the top exits on the right, and the seaweed entering on the left exits at the bottom. In a mismatch cell, the two seaweeds keep straight and cross each other, if and only if this pair of seaweeds have not previously crossed; otherwise, they bend as in a match cell. Therefore, any given pair of seaweeds are only allowed to cross at most once in the course of the computation. Notice that the same property of avoiding double-crossings also holds for any pair of highest-scoring paths in the alignment dag.

We now formalise the algorithm as follows. We start with a full-match alignment dag \( G_{m, n} \), and transform it by incremental cell updates into the alignment dag \( G_{a, b} \). We will maintain a variable seaweed braid on \( m + n \) seaweeds, corresponding to the current alignment dag. We index the starting positions of the seaweeds by \( \langle -m : n \rangle \), and the terminating positions by \( \langle 0 : m + n \rangle \). The current seaweed braid will thus be represented by a variable permutation matrix \( P \) over \( \langle -m : n | 0 : m + n \rangle \). A nonzero \( P_{a,b}(i, j) = 1 \), \( i \in \langle -m : n \rangle \), \( j \in \langle 0 : m + n \rangle \), represents a seaweed starting at position \( i \) and terminating at position \( j \).

The initial full-match alignment dag \( G_{m, n} \) corresponds to an identity seaweed braid. Therefore, we initialise

\[
P \leftarrow \text{Id}_{m\langle -m : n | 0 : m + n \rangle}
\]

The algorithm then sweeps the alignment dag in an arbitrary order compatible with the \( \ll \)-dominance order of the cells. Recall that every cell in the dag is initially a match cell. If the current cell in \( G_{a, b} \) is also a match cell, no update is needed. Otherwise, the current match cell has to be transformed into a mismatch cell, resulting in an update on the current seaweed braid, which has to be reflected in its representing matrix \( P \).

Consider a cell defined by the characters \( a(\hat{l}), b(\hat{i}) \), \( \hat{l} \in \langle 0 : m \rangle \), \( \hat{i} \in \langle 0 : n \rangle \). Let \( i^* = \hat{i} + m - \hat{l} \). The two seaweeds passing through the current cell terminate at positions \( \{i^* - \frac{1}{2}, i^* + \frac{1}{2}\} = \{i^* - 1 : i^* + 1\} \). We update a 2 × 2 induced permutation submatrix of \( P \) as follows:

\[
\text{if } a(\hat{l}) \neq b(\hat{i}) \text{ and } P(\cdot | i^* - 1 : i^* + 1) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \\
\text{then } P(\cdot | i^* - 1 : i^* + 1) \leftarrow \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}
\]

By Theorem 11, the algorithm maintains the invariant “current state of matrix \( P \) is the semi-local seaweed matrix for the current state of the alignment
Figure 4.1: A snapshot of Algorithm 1 (the seaweed algorithm)

dag\textsuperscript{\dagger}. Therefore, at the end of the sweep, we have the output matrix

\[ P = P_{a,b} \]

**Cost analysis.** For every cell, the $2 \times 2$ column-induced submatrix $P_{\langle \cdot \mid i^* - 1 : i^* + 1 \rangle}$ can be obtained from matrix $P$ in time $O(1)$. The cell update also runs in time $O(1)$. Therefore, the overall running time is $O(mn)$.

The memory cost is dominated by storing the input and the linear representation of the current matrix $P$. Therefore, the overall memory cost is $O(m + n)$. \hfill \Box

**Example 16** Figure 4.1 shows a snapshot of Algorithm 1. The dag $G_{a,b}$ is swept in the top-to-bottom, left-to-right lexicographic cell order. The dag area that has already been swept is shown by the dark border; the current cell is shaded in yellow. Since the two seaweeds meeting in the current cell have previously crossed, the current step will leave the seaweed braid unchanged, so that double crossing does not occur. The final layout of the seaweeds is the one shown in Figure 3.3; it describes the full sequence of states of the seaweed braid represented by matrix $P$ in Algorithm 1. \hfill \Box

Recall that the dag cells in Algorithm 1 can be processed in any order consistent with the $\ll$-dominance partial order. In particular, the cell processing order can be fixed so that the algorithm will compute incrementally the semi-local seaweed matrix for all prefixes of string $a$ against whole string $b$. By keeping the algorithm’s intermediate data, we obtain a data structure that allows efficient LCS queries for every prefix of $a$ against every substring of $b$. As in the classical dynamic programming approach, this data structure can be used to trace back (i.e. to obtain character by character) the actual LCS corresponding to a prefix-substring LCS query, in time proportional to the size of the output (i.e. the length of the output subsequence). Alternatively, a technique similar to memory-efficient dynamic programming by
Hirschberg [66] can be applied to achieve prefix-substring LCS traceback in the same asymptotic time, but in a linear amount of memory.

Assuming the “equal/unequal” character comparison model, Aho et al. [2] gave a lower bound of $\Omega(mn)$ on the solution running time of the (global) LCS problem (see also a survey by Bergroth et al. [16]). Both the standard dynamic programming algorithm, and the seaweed algorithm Algorithm 1 match this lower bound, and therefore are asymptotically optimal in this model.

4.2 The micro-block speedup

In the previous section, we assumed the character comparison model that only allows comparison outcomes “equal” and “unequal”. We now switch to a more powerful character comparison model, assuming that the alphabet is a totally ordered set, and comparison outcomes are “less than”, “equal” and “greater than”. In this model, we no longer need to process every dag cell individually, so algorithms with running time $o(mn)$ become possible.

A classical LCS computation speedup originates from a matrix multiplication method by Arlazarov et al. [11], often nicknamed the “four Russians method”. In this work, we call it the micro-block method, adopting the terminology of Bille and Gørtz [20]. The main idea of the method is to sweep the alignment dag in regular micro-blocks of a small, suitably chosen size, such that running time can be saved by precomputing all possible micro-block updates in advance. Without loss of generality, we assume that $m \leq n$. Using the micro-block method, Masek and Paterson [98] gave an algorithm for the (global) LCS problem running in time $O\left(\frac{mn}{\log^2 n} + n\right)$ for a constant-size alphabet. An alternative approach to subquadratic LCS computation was developed by Crochemore et al. [40].

An extension of the micro-block subquadratic LCS algorithm to an unbounded-size alphabet, running in time $O\left(\frac{mn(\log \log n)^2}{\log^2 n} + n\right)$, was suggested by Paterson and Dančík [105], and fully developed by Bille and Farach-Colton [19]. In this extension, a second, coarser level of alignment dag partitioning is introduced. The blocks of this second level, called macro-blocks, are used for reducing the effective alphabet size, maximising the number of input string characters that fit into a machine word for each micro-block update.

We now give an algorithm for semi-local LCS running in subquadratic

---

1This holds true even if the computation model assumption is weakened, so that character comparisons and arithmetic operations are charged using the log-cost RAM model. However, for uniformity we will stick to our original assumption of the unit-cost RAM model.

2The original algorithm by Masek and Paterson [98] runs in time $O\left(\frac{mn}{\log n} + n\right)$ for a constant-size alphabet in the log-cost RAM model. The unit-cost RAM version of the algorithm was given in [132, 19].
time, which makes a slight improvement on Algorithm 1. The algorithm uses the two-level micro-block method, similar to the global LCS algorithm of [19], and matches it in running time. At the same time, our algorithm provides a substantially more detailed string comparison. However, in contrast to the global LCS algorithms, our algorithm does not appear to be able to take advantage of a bounded alphabet size.

Algorithm 2 (Semi-local LCS: The seaweed algorithm with micro-block speedup)

**Input, output:** as in Algorithm 1; we assume $m \leq n$.

**Description.** Without loss of generality, we may assume that the alphabet size is at most $2n$, and that the characters are encoded by odd half-integers in the range $\langle -n : n \rangle$. We call two strings of equal length isomorphic, if one can obtained from the other by a permutation of the alphabet.

We process the alignment dag in square micro-blocks of size

$$t = \min \left( \frac{\log n}{16 \log \log n}, m \right)$$

where the logarithms are base 2. Similarly to Algorithm 1, we start with a full-match alignment dag $G_{\cdot,\cdot}$, and transform it by incremental micro-block updates into the alignment dag $G_{a,b}$. We maintain a variable seaweed braid on $m+n$ seaweeds, represented by a permutation matrix $P$ over $\langle -m : n \mid 0 : m+n \rangle$. As in Algorithm 1, a nonzero $P_{a,b}(\hat{i}, \hat{j}) = 1$, $\hat{i} \in \langle -m : n \rangle$, $\hat{j} \in \langle 0 : m+n \rangle$, represents a seaweed starting at position $\hat{i}$ and terminating at position $\hat{j}$. We initialise

$$P \leftarrow \text{Id}_{m\langle -m : n \mid 0 : m+n \rangle}$$

The algorithm then sweeps the alignment dag in some order compatible with the $\ll$-dominance order of the micro-blocks. For each micro-block, we perform an update on the current seaweed braid, which has to be reflected in its representing matrix $P$.

Consider a micro-block defined by the substrings $a(l : l+t)$, $b(i : i+t)$, where $l \in [0 : m-h]$, $i \in [0 : n-h]$. Let $i^* = i + m - l$. The $2t$ seaweeds passing through the current micro-block terminate at positions $\langle i^* - t : i^* + t \rangle$. A micro-block can be regarded as a function, parameterised by the current input substrings, and performing an update on the $2t \times 2t$ column-induced permutation submatrix $P(\cdot \mid i^* - t : i^* + t)$. The states of the submatrix before and after the updates will be called the micro-block’s input submatrix and output submatrix, respectively. Note that both of these are permutation matrices, and therefore can be represented implicitly by their nonzeros.

The alignment dag can be swept in an arbitrary order compatible with the $\ll$-dominance partial order of the micro-blocks. Combined with pre-computation, this is already sufficient to obtain a subquadratic algorithm.
However, in order to achieve higher speedup, we introduce a second level of macro-blocks of size
\[ s = \min\left(\frac{\log^2 n}{2}, m\right) \]
We define a macro-block’s input and output submatrices similarly to a micro-block’s ones.

The characters of a macro-block’s defining input substrings are encoded by values in the range \((-n : n)\). The macro-block’s input and output submatrices are represented by the row and column indices of the nonzeros; the natural range of these indices is also \((-n : n)\). In order to perform the computation efficiently, we remap each of these ranges to a smaller range \((-s : s)\) before passing the values to the micro-block level. The range remapping preserves the linear order of the values (both characters and matrix indices).

We process each macro-block as follows. First, we obtain its defining substrings and the input submatrix; overall, we have \(O(s)\) characters and index values for the macro-block. We then remap both these characters and index values from \((-n : n)\) to \((-s : s)\) by removing \(2n - 2s\) unused values from the range, while preserving the relative order of the remaining \(2s\) values. This remapping requires sorting of the \(O(s)\) values, and can be performed in time \(O(s \log s)\). We then sweep the current macro-block in an arbitrary order compatible with the \(\ll\)-dominance partial order of the micro-blocks. For each micro-block, we obtain its defining input substrings and the input submatrix; overall, we have \(O(t)\) values for the micro-block of size \(t\). We then apply a precomputed update to a \(2t \times 2t\) column-induced permutation submatrix of \(P\) as follows:

\[
P(\cdot \mid i^* - t : i^* + t) \leftarrow \text{update}(a(l : l + t), b(i : i + t), P(\cdot \mid i^* - t : i^* + t))
\]

The micro-block’s defining substrings and the input state consist each of \(2t\) values, ranging over \((-s : s)\). For each of the at most \((2s)^{2t+2t} = (2s)^{4t}\) possible combinations of the input character and index values, the output index values given by the function \text{update} are precomputed in advance, using Algorithm 1.

The algorithm maintains the invariant “current state of matrix \(P\) is the implicit semi-local score matrix for the current state of the alignment dag”. Therefore, at the end of the sweep, we have the output matrix

\[ P = P_{a,b} \]

**Cost analysis.** In the precomputation stage, there are at most \((2s)^{4t}\) problem instances, each of which runs in time \(O(t^2)\). Therefore, the running time of the precomputation is

\[
O((2s)^{4t} \cdot t^2) = O\left((\log^2 n)^{\frac{\log n}{\log \log n}} \cdot t^2\right) = O\left(2^{\log (\log^2 n) \frac{\log n}{\log \log n}} \cdot t^2\right) =
\]
Figure 4.2: A snapshot of Algorithm 2 (the micro-block seaweed algorithm)

\[
O\left(2^{2 \log \log n \cdot \log \log n} \cdot t^2\right) = O\left(2^{\log n \cdot t^2}\right) = o(n)
\]

which is negligible, compared to the subsequent main computation stage.

In the main computation stage, there are \(\frac{m n}{t^2}\) micro-block update steps. The micro-block’s defining substrings and input-output submatrices are each represented by \(2t\) values in the range \((-s : s)\). The full micro-block data are of size

\[
O\left(2t \cdot \log(2s)\right) = O\left(\frac{\log n}{\log \log n} \cdot \log(\log^2 n)\right) = O(\log n)
\]

and hence fit into a constant number of machine words. Therefore, the total running time of the algorithm is

\[
\frac{m n}{t^2} \cdot O(1) = O\left(\frac{m n (\log \log n)^2}{\log n} + n\right)
\]

**Example 17** Figure 4.2 shows a snapshot of Algorithm 2, using the same conventions as Figure 4.1. For simplicity, the macro-blocks are not shown, and the micro-blocks are assumed to be of size 2. As in Algorithm 1, the final layout of the seaweeds is identical to the one given in Figure 3.3.

### 4.3 Incremental LCS and semi-local LCS

The *incremental LCS problem* was introduced by Landau et al. [89], and by Kim and Park [80]. Given a fixed string, the problem asks its LCS score against a variable string, which can be updated on-line by either appending or prepending a character. An extension, called *fully-incremental LCS problem*, was introduced by Ishida et al. [70]. Here, both strings can be updated 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.
Let $a$, $b$ denote the current state of each input string, and $m$, $n$ their respective current size. Landau et al. [89] and Kim and Park [80] gave incremental LCS algorithms with worst-case time $O(m)$ per update of string $b$. Ishida et al. [70] extended this result to a fully-incremental LCS algorithm with worst-case time $O(m)$ (respectively, $O(n)$) per update of string $b$ (respectively, $a$).

We now give an algorithm for the fully-incremental LCS problem, matching the above algorithms in running time. Our algorithm is a straightforward generalisation of the seaweed algorithm (Algorithm 1). The dynamic data structure consists of the nonzeros of semi-local seaweed matrix $P_{a,b}$. When a new character is prepended or appended to string $a$ (respectively, $b$), the seaweed matrix is updated by processing a new row of cells along the top or bottom boundary (respectively, a new column of cells along the left or right boundary) of the alignment dag $G_{a,b}$.

The same technique also extends to the case where the dynamic data structure is required to support, in addition to the global LCS score, also semi-local LCS queries. A data structure of Theorem 1, allowing efficient semi-local LCS score queries, can be maintained on top of the matrix $P_{a,b}$ at an extra time $O((m + n) \log(m + n))$ per string update.

We now give another generalisation of incremental string comparison. Consider a fixed string of length $n$, and a variable pattern string, which can be modified 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 preprocessing of this set is allowed. The block-incremental LCS problem asks, as before, to maintain a data structure that will store the LCS score for the text against the pattern, and will allow efficient on-line updates of this score.

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 $O(n l)$.

We now give an algorithm for the block-incremental LCS problem, that improves on the naive algorithm in running time. The set of admissible blocks is preprocessed off-line by computing the string-substring seaweed matrix for every admissible block against the fixed string. The preprocessing runs in time $O(n \cdot L)$, where $L$ is the total length of the admissible blocks; if the admissible blocks are sufficiently long, the micro-block speedup (Algorithm 2) can be applied. Given the precomputed string-substring seaweed matrices, an individual string update can be processed in time $O(n \log l)$ by Corollary 1.
4.4 Common-substring LCS and semi-local LCS

The common-substring LCS problem was introduced by Landau and Ziv-Ukelson [91] (see also [39]). Given a text string $t$ of length $n$ and an unspecified number of pattern strings, the problem asks for the LCS score of the text against each of the patterns. The pattern strings may share a known common substring $c$ of length $l$: we assume $l \leq n$. A pattern string may contain several copies of the common substring; the locations of all the copies 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. Time $O(n)$ per pattern character is allowed outside any occurrences of $c$.

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

An improved algorithm was given by Landau and Ziv-Ukelson [91]. This algorithm, following some preprocessing in time $O(nl)$, runs in time $O(n)$ per occurrence of the common substring.

We now give an algorithm for the common-substring LCS problem, that matches the above algorithm in both preprocessing and running time, but has the potential for a micro-block speedup in the preprocessing phase. We preprocess the common substring $c$ by computing the nonzeros of the string-substring seaweed matrix $P_{ct}$. The preprocessing runs in time $O(nl)$; if the common substring length $l$ is sufficiently large, the micro-block speedup (Algorithm 2) can be applied. For every pattern string $p$, we now compute the vector $h_{p,t}(\langle 0 : n \rangle)$ over $\langle 0 \mid \ast \rangle$. This vector can be computed incrementally by repeated application of Theorem 6. Each vector update takes time $O(n)$ per occurrence of the common substring.

The common-substring LCS problem can be generalised to the semi-local common-substring LCS problem. As in the ordinary semi-local LCS problem, string-substring, substring-string, prefix-suffix and suffix-prefix LCS score queries are now allowed between the text and each of the patterns.

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

We now give an algorithm for the semi-local common-substring LCS problem, improving on the naive algorithm in running time. We preprocess the common substring $c$ in time $O(nl)$ by computing the nonzeros of the semi-local seaweed matrix $P_{ct}$. As before, the preprocessing runs in time $O(nl)$; if $l$ is sufficiently large, the micro-block speedup can be applied. For every pattern string $p$, the semi-local seaweed matrix $P_{p,t}$ can now be computed incrementally, starting from an arbitrary occurrence of $c$. 

54
in \( p \). The resulting algorithm can be regarded as a special case of the block-incremental LCS algorithm from Section 4.3. Each incremental update takes time \( O(n \log l) \) per occurrence of the common substring. Overall, the algorithm takes time \( O(n) \) for the first occurrence of the common substring in a pattern, and time \( O(n \log l) \) for each of its subsequent occurrences in the same pattern. In particular, if the common substring occurs only once in every pattern string, our algorithm improves on the algorithm of [91, 39] in functionality and preprocessing time, without any increase in the asymptotic update time.

### 4.5 Cyclic LCS

Given strings \( a, b \) of length \( m, n \) respectively, the cyclic LCS problem asks for the highest LCS score between \( a \) and all cyclic shifts of \( b \) (or, equivalently, all cyclic shifts of \( a \) and all cyclic shifts of \( b \)).

Cyclic string comparison has been considered by Maes [95], Bunke and Bühler [25], Landau et al. [89], Schmidt [117], Marzal and Barrachina [97]. Works [89, 117] give algorithms that solve the cyclic LCS problem in worst-case time \( O(mn) \).

We now give a new algorithm for the cyclic LCS problem, improving on the existing algorithms in running time. Without loss of generality, we assume \( m \leq n \). First, we call Algorithm 2 on strings \( a \) and \( bb \) (a concatenation of string \( b \) with itself), obtaining the semi-local seaweed matrix \( P_{a,bb} \). Then, we perform \( n \) string-substring LCS score queries for \( a \) against every substring of \( bb \) of length \( n \); this can be done efficiently via Theorem 2. Finally, we take the maximum score among all the queries. The overall running time is dominated by the call to Algorithm 2, which runs in time \( O\left( \frac{mn(\log\log n)^2}{\log^2 n} + n \right) \).

### 4.6 Longest repeating subsequence

Given a string \( a \) of length \( n \), the longest repeating subsequence problem asks for the length of the longest subsequence of \( a \) that is a square, i.e. a concatenation of two identical strings.

This problem has been considered under the name “longest tandem scattered subsequence problem” by Kosowski [84], who gave an algorithm running in time \( O(n^2) \).

We now give a new algorithm for the longest repeating subsequence problem, improving on the existing algorithm in running time. First, we call the seaweed algorithm with the micro-block speedup (Algorithm 2) on string \( a \) against itself, obtaining the semi-local seaweed matrix \( P_{a,a} \) in time \( O\left( \frac{n^2(\log\log n)^2}{\log^2 n} \right) \). Then, we perform \( n - 1 \) prefix-suffix LCS score queries for every possible non-trivial prefix-suffix decomposition of \( a \); this can be done...
in time $O(1)$ per query by Theorem 2. Finally, we take the maximum score among all the queries. The overall running time is $O\left(\frac{n^2 (\log \log n)^2}{\log^2 n}\right)$. 
5.1 Weighted scores and edit distances

The concept of LCS score is generalised by that of (weighted) alignment score (see e.g. [71]). An alignment of strings $a$, $b$ is obtained by putting a subsequence of $a$ into one-to-one correspondence with a (not necessarily identical) subsequence of $b$, character by character and respecting the index order. The corresponding pair of characters, one from $a$ and the other from $b$, are said to be aligned. A character that is not aligned against a character of another string is said to be aligned against a gap in that string. Each of the resulting character alignments is given a real weight:

- a pair of aligned matching characters has weight $w_{\oplus} \geq 0$;
- a pair of aligned mismatching characters has weight $w_{\otimes} < w_{\oplus}$;
- a gap-character or character-gap pair has weight $w_{\ominus} \leq \frac{1}{2}w_{\otimes}$; it is normally assumed that $w_{\ominus} \leq 0$.

The intuition behind the weight inequalities is as follows: aligning a matching pair of characters is always better than aligning a mismatching pair of characters, which in its turn is never worse than leaving both characters unaligned.

**Definition 22** The (weighted) alignment score for strings $a$, $b$ is the maximum total weight of character pairs in an alignment of $a$ against $b$. 

**Example 18** The LCS alignment score is given by

$$w_{\oplus} = 1 \quad w_{\otimes} = w_{\ominus} = 0$$

A slightly more sophisticated alignment score, intended to penalise gaps in DNA sequence alignment, is given by

$$w_{\oplus} = 1 \quad w_{\otimes} = 0 \quad w_{\ominus} = -0.5$$

57
Another alignment score used for DNA sequence comparison [33, Section 1.3] is given by
\[ w_\oplus = 2 \quad w_\otimes = -1 \quad w_\ominus = -1.5 \]

We define the semi-local and the three-way semi-local (weighted) alignment score problems by straightforward extension of Definitions 14 and 15. A semi-local alignment score corresponds to a highest-scoring path in a weighted alignment dag, where diagonal match edges, diagonal mismatch edges, and horizontal and vertical edges have weight \( w_\oplus, w_\otimes, w_\ominus \), respectively. The output of the semi-local alignment score problem is a semi-local (weighted) score matrix; to distinguish such matrices from (unweighted) LCS score matrices, we will use a script font, e.g. \( H_{a,b} \). A semi-local (weighted) score matrix is anti-Monge; however, in contrast with the unweighted case, it is not necessarily unit-anti-Monge.

When comparing globally a fixed pair of strings, it is convenient to normalise the weights so that \( w_\oplus = 1 \), assuming that originally \( w_\oplus \neq 0 \). More generally, we only need to consider alignment scores with \( 0 = w_\otimes \leq w_\ominus < w_\oplus = 1 \). (A similar observation was made by Rice et al. [110]; see also [64, 74].) Indeed, given any unrestricted original weights \( w_\oplus, w_\otimes, w_\ominus \), we can transform them to normalised weights:
\[ w^*_\oplus = 1 \quad w^*_\otimes = \frac{w\otimes - 2w\ominus}{w\oplus - 2w\otimes} \quad w^*_\ominus = 0 \] (5.1)

We call the corresponding alignment score the normalised score.

**Example 19** In Example 18, the LCS score is already normalised. The other two scores correspond to the normalised scores with weights \( w^*_\oplus = 1 \), \( w^*_\otimes = 0.5 \) and \( w^*_\ominus = 0.4 \) respectively, and \( w^*_\ominus = 0 \).

The original alignment score \( h \) can be restored from the normalised score \( h^* \) as
\[ h = h^* \cdot (w_\oplus - 2w_\otimes) + (m + n) \cdot w_\ominus \] (5.2)

For fixed string lengths \( m \) and \( n \), maximising the normalised score \( h^* \) is equivalent to maximising the original score \( h \). However, more care is needed when maximising the score for variable string lengths, e.g. the semi-local alignment score. In such cases, an explicit conversion from normalised weights to original weights will be necessary prior to the maximisation.

In this work, we will mostly restrict ourselves to character alignment weights that satisfy the following rationality condition.

**Definition 23** A set of character alignment weights will be called rational, if all the weights are rational numbers.
Figure 5.1: Alignment dag $G_{\tilde{a}, \tilde{b}}$ for the blown-up strings

Given a rational set of normalised weights, the semi-local alignment score problem on strings $a$, $b$ can be reduced to the semi-local LCS problem by the following blow-up procedure. Let $w_\oplus = \frac{\mu}{\nu} < 1$, where $\mu$, $\nu$ are positive natural numbers. We transform input strings $a$, $b$ of lengths $m$, $n$ into new blown-up strings $\tilde{a}$, $\tilde{b}$ of lengths $\tilde{m} = \nu m$, $\tilde{n} = \nu n$. The transformation consists in replacing every character $\gamma$ in each of the strings by a substring $\mu \gamma^{\nu-\mu}$ of length $\nu$. Here, ‘$\$’ stands for a special character not present in the original strings. We have

$$H_{a,b}(i,j) = \frac{1}{\nu} \cdot H_{\tilde{a}, \tilde{b}}(\nu i, \nu j)$$

for all $i \in [-m : n]$, $j \in [0 : m + n]$, where the matrix $H_{a,b}$ is defined by the normalised weights on the original strings $a$, $b$, and the matrix $H_{\tilde{a}, \tilde{b}}$ by the LCS weights on the blown-up strings $\tilde{a}$, $\tilde{b}$. Therefore, all the techniques of the previous sections apply to the rational-weighted semi-local alignment score problem, assuming that $\nu$ is a constant.

Example 20 Figure 5.1 shows the alignment dag for the blown-up strings $\tilde{a}$, $\tilde{b}$. We assume the normalised alignment weights $w_\oplus = 1$, $w_\otimes = 0.5$, $w_\ominus = 0$, hence $\nu = 2$. The highlighted path of score 5.5 corresponds to the string-substring weighted alignment score for string $a$ against substring $b(4:11) = \text{“cabcaba”}$. \qed

An important special case of weighted string alignment is the edit distance problem. Here, the characters are assumed to match “by default”: $w_\oplus = 0$. The mismatches and gaps are penalised: $2w_\ominus \leq w_\otimes < 0$. The resulting score is always nonpositive. It is traditional we regard string $a$ as being transformed into string $b$ by a sequence of weighted character edits:

- character insertion or deletion (indel) has weight $-w_\ominus > 0$;
- character substitution has weight $-w_\otimes > 0$. 

59
Definition 24 The (weighted) edit distance between strings $a$, $b$ is the minimum total weight of a sequence of character edits transforming $a$ into $b$. Hence, it is the nonnegative opposite of the corresponding alignment score. □

The edit distance is a metric: it is nonnegative, positive except on equal strings, symmetric, and satisfies the triangle inequality.

Example 21 The indel distance (also called the LCS distance) [104, 10] has indel weight 1, and substitution weight 2, making a substitution equivalent to an insertion-deletion pair. The corresponding indel alignment score is given by

$$w_{\oplus} = 0 \quad w_{\otimes} = -2 \quad w_{\ominus} = -1$$

The indelsub distance (also called the Levenshtein distance) [92] has both indel weight and substitution weight equal to 1. The corresponding indelsub alignment score is given by

$$w_{\oplus} = 0 \quad w_{\otimes} = w_{\ominus} = -1$$  □

The definition of edit distance can be generalised by allowing insertions and deletions to have two separate, distinct weights. For example, the asymmetric episode distance [43] corresponds to insertion weight 0, and strictly positive deletion and substitution weights.

In the rest of this work, the edit distance problem will be treated as a special case of the weighted alignment problem. In particular, all the techniques of the previous sections apply to the semi-local edit distance problem, as long as the character edit weights are rational.

5.2 Approximate pattern matching

Approximate pattern matching is a natural generalisation of both the ordinary (exact) pattern matching, and of the alignment score and the edit distance problems. Given a text string $t$ of length $n$ and a pattern string $p$ of length $m \leq n$, the approximate pattern matching problem asks to find the substrings of the text that are locally closest to the pattern, i.e. that have the locally highest alignment score (or, equivalently, lowest edit distance) against the pattern. The precise definition of “locally” may vary in different versions of the problem, and will typically correspond to a certain set of maxima (global, local, row, column etc.) in the string-substring score matrix $H_{p,t}$.

The most general form of approximate pattern matching is as follows.

Definition 25 The complete approximate matching problem assumes an alignment score with arbitrary weights, and for every suffix of text $t$, asks for a prefix of this suffix that has the highest alignment score against pattern $p$. This corresponds to the set of all row maxima in the matrix $H_{p,t}$. □
The complete approximate pattern matching problem can be solved by a classical algorithm by Sellers [118] in time $O(mn)$. Assuming a rational set of weights, the micro-block method gives an algorithm running in time $O\left(\frac{mn}{\log^2 n} + n\right)$ for a constant-size alphabet, and in time $O\left(\frac{m(n \log n)^2}{\log^2 n} + n\right)$ for an unbounded-size alphabet. Various extensions of the problem have been considered by Cormode and Muthukrishnan [38] and many others (see e.g. a survey by Navarro [103] and references therein).

The complete approximate pattern matching problem has output $O(n)$. Most of this output may be redundant, if we are only interested in the areas of sufficiently high similarity. This motivates the introduction of a similarity threshold. Given a matrix $A$ and a threshold $h$, it will be convenient to denote the subset of entries above the threshold by

$$\tau_h(A) = \{(i,j), \text{ such that } A(i,j) \geq h\}$$

**Definition 26** The threshold approximate matching problem (often called simply “approximate matching”) assumes an alignment score with arbitrary weights, and, given a threshold score $h$, asks for all substrings of text $t$ that have alignment score at least $h$ against pattern $p$. This corresponds to all points in the set $\tau_h(H_{p,t})$.

An important special case of the threshold approximate matching problem is given by the unweighted LCS score, and the maximum possible threshold $h = m$.

**Definition 27** The local subsequence recognition problem (also known as the “episode matching problem”) asks for all substrings in text $t$ containing pattern $p$ as a subsequence. This corresponds to all points in the set $\tau_m(H_{p,t})$. If this set is nonempty (i.e. if $p$ is contained in the whole $t$ as a subsequence), it is the set of all global maxima in the matrix $H_{p,t}$.

The substrings asked for by Definitions 26–27 will be called matching substrings.

Although we have now concentrated our search on the areas of high similarity, the output of these approximate matching problems may still be highly redundant. In particular, the output of the threshold approximate matching is likely to contain highly overlapping matching substrings, where the starting and/or ending positions only differ by a few characters. In the more extreme case of local subsequence recognition, any superstring of a matching substring will also be matching. The usual convention for eliminating such redundancy is to filter the output, by retaining only a subset of the matching substrings. The full output can be efficiently reconstructed, if necessary, from the filtered one. For instance, the filtered output may retain:
- unique starting (or, symmetrically, ending) positions of matching substrings;
- inclusion-minimal matching substrings;
- matching substrings of a fixed length $w$.

In particular, the minimal-window subsequence recognition problem asks for all inclusion-minimal substrings in the text containing the pattern as a substring. This corresponds to all $\preceq$-minimal points in the set $\tau_m(H_{p,t})$. The fixed-window subsequence recognition problem, given a window length $w$, asks for all substrings of length $w$ of the text containing the pattern as a substring. This corresponds to all points lying on the intersection of the diagonal $j - i = w$ with the set $\tau_m(H_{p,t})$.

In the results we cite, the filtering is usually left implicit, and can be assumed to be of one of the types introduced above.

The local subsequence recognition problem has been considered by Das et al. [43]. For both the minimal-window and fixed-window versions, they give an algorithm running in time $O(mn \log^2 n + n)$ for a constant-size alphabet, which can be modified to an algorithm running in time $O\left(mn(\log \log n)^2 \log^2 n + n\right)$ for an unbounded-size alphabet. A multi-pattern version of the problems has been considered by Cégielski et al. [29].

The threshold approximate pattern matching problem has also been considered under the indelsub alignment score, given by alignment weights $w_\oplus = 0$, $w_\otimes = w_\ominus = -1$. This form of approximate string matching, with the threshold score $h < 0$, is usually defined in terms of the corresponding threshold edit distance $k = -h > 0$, and is traditionally known as approximate matching with $k$ differences. When the threshold $k$ is low, the best known algorithm is by Cole and Hariharan [37], running in time $O(m + n + \frac{nk^4}{m})$. For higher values of $k$, the best known algorithm is by Landau and Vishkin [90], running in time $O(nk)$. We now give a new unified algorithm for approximate pattern matching, applicable to any of the problem’s versions described by Definitions 25–27, in the case of rational weights. Our algorithm matches the micro-block version of Sellers’ algorithm in running time for an unbounded-size alphabet. The algorithm’s running time does not include the distance threshold $k$ as a parameter.

The new algorithm is as follows. First, we call Algorithm 2 on strings $p$, $t$ (if necessary, using the blow-up technique of Section 5.1), obtaining the semi-local seaweed matrix $P_{p,t}$. By Theorem 1, we then build a data structure that allows to query any element of the semi-local score matrix $H_{p,t}$ in polylogarithmic time. Since matrix $H_{p,t}$ is anti-Monge, all the row maxima can now be found efficiently by the algorithm of Lemma 1, which solves the complete approximate matching problem. The threshold approximate
matching problem can now be solved by considering the matrix entries in the neighbourhood of the row maxima. Both the minimal-window and the fixed-window versions of the local subsequence recognition problem can be solved by selecting the row maxima that satisfy the additional filtering conditions.

In all the described versions of the algorithm, the overall running time is dominated by the call to Algorithm 2, which runs in time $O\left(\frac{mn(\log \log n)^2}{\log^2 n} + n\right)$. 
Chapter 6

Periodic string comparison

6.1 The periodic seaweed algorithm

In many string comparison applications, one or both of the input strings may have periodic structure. In this chapter, we show how to exploit such structure efficiently, using a variant of the seaweed method.

Consider the problem of comparing a finite string \( a \) of length \( m \) against a string \( b \), which is infinite in both directions and periodic: \( b = u^{±∞} = \ldots uuuu\ldots \). The period string \( u \) is finite of length \( p \).

**Definition 28** Given strings \( a, u \), the periodic string-substring LCS problem asks for the LCS score of \( a \) against every finite substring of \( b = u^{±∞} \).

Without loss of generality, we assume that every character of \( a \) occurs in \( u \) at least once. Clearly, the length of the substring of \( b \) in Definition 28 can be restricted to be at most \( mp \) (for longer substrings of \( b \), every character of \( a \) can be matched to a different copy of \( u \) within the substring, and therefore the string-substring LCS score will be equal to \( m \)).

The definition of the alignment dag (Definition 16) extends naturally to the periodic string-substring LCS problem. The alignment dag for such a problem is itself periodic: the edges \( v_{l,i−\frac{p}{2}+kp} \rightarrow v_{l,i+\frac{p}{2}+kp} \) (respectively, \( v_{l−\frac{1}{2},i+k} \rightarrow v_{l+\frac{1}{2},i+k} \), \( v_{l−\frac{1}{2},i−\frac{1}{2}+kp} \rightarrow v_{l+\frac{1}{2},i+\frac{1}{2}+kp} \) have equal scores for all \( l \in [l_0 : l_1], i \in [i_0, i_1], \hat{i} \in [i_0 : i_1], k \in [−∞ : +∞] \). Such an alignment dag can also be regarded as a horizontal composition of an infinite sequence of period subdags, each of which is isomorphic to the \( m \times p \) alignment dag \( G_{a,u} \).

Since string \( b \) is infinite, the semi-local score and seaweed matrices can be understood as just the respective string-substring matrices: matrix \( H_{a,b} = H_{a,b}^{∞} \) is over \([−∞ : +∞]\), and matrix \( P_{a,b} = P_{a,b}^{∞} \) is over \((-∞ : +∞)\). Furthermore, matrices \( H_{a,b}, P_{a,b} \) are again periodic: we have

\[
H_{a,b}(i, j) = H_{a,b}(i + p, j + p)
\]
\[ P_{a,b}(i,j) = P_{a,b}(i+p,j+p) \]

for all \( i, j \in [\infty : \infty], i, j \in (-\infty : \infty) \). To represent such matrices, it is sufficient to store the \( p \) nonzeros of the row-period submatrix \( P_{a,b}(0 : p | \cdot) \), or, symmetrically, of the column-period submatrix \( P_{a,b}(\cdot | 0 : p) \). The nonzero sets of the two period submatrices can be obtained from one another in time \( O(p) \). When working with an infinite periodic seaweed matrix, we will assume such a representation implicitly. For example, accessing a column \( P_{a,b}(\ast, \hat{i}) \) will correspond to accessing all columns \( P_{a,b}(\ast, \hat{i}’) \) with \( \hat{i} = \hat{i}' \mod p \).

For the periodic semi-local LCS problem, the seaweeds only need to be traced within a single period subdag, with appropriate wraparound. Therefore, the problem can be solved by the following variant of the seaweed algorithm (Algorithm 1).

**Algorithm 3 (Periodic string-substring LCS: The periodic seaweed algorithm)**

**Input:** strings \( a, u \) of length \( m, p \), respectively; we have \( b = u^{\pm\infty} \).

**Output:** nonzeros of semi-local seaweed matrix \( P_{a,b} \), represented by nonzeros of (say) column-period submatrix \( P_{a,b}(<p|0:p) \).

**Description.** Similarly to Algorithm 1, we start with a full-match alignment dag \( G_{\varnothing^m, \varnothing^{\pm\infty}} \), and transform it by incremental cell updates into the alignment dag \( G_{a,b} \). We will maintain an infinite periodic seaweed braid, corresponding to the current alignment dag. The current seaweed braid will be represented by a variable permutation matrix \( P \) over \( \langle -\infty : +\infty | -\infty : +\infty \rangle \), with period \( p \); implicitly, we will assume a column-period submatrix representation. A nonzero \( P_{a,b}(i,j) = 1, i, j \in (-\infty : +\infty) \), represents a seaweed starting at position \( \hat{i} \) and terminating at position \( \hat{j} \).

The initial full-match alignment dag \( G_{\varnothing^m, \varnothing^{\pm\infty}} \) corresponds to an identity seaweed braid. Therefore, we initialise

\[ P \leftarrow \text{Id}_{m}(-\infty : +\infty | -\infty : +\infty) \]

which is clearly a periodic matrix for any period, including \( p \).

The algorithm then sweeps the period subdag in the following order. In the outer loop, we run through the rows of cells top-to-bottom. For the current row \( \hat{l} \in (0 : m) \), we start the inner loop at an arbitrary index \( \hat{i}_0 \in (0 : p) \), such that \( a(\hat{l}) = b(\hat{i}) \), hence the corresponding cell in \( G_{a,b} \) is a match cell. Such an index \( \hat{i}_0 \) is guaranteed to exist by the assumption that every character of \( a \) occurs in \( u \) at least once. Then, starting from \( \hat{i} = \hat{i}_0 \), we sweep the cells left-to-right, wrapping around from \( \hat{i} = p - \frac{1}{2} \) to \( \hat{i} = \frac{1}{2} \), and continuing the sweep left-to-right up to \( \hat{i} = (\hat{i}_0 - 1) \mod p \).

Consider a cell defined by the characters \( a(\hat{l}), b(\hat{i}) = u(\hat{i}), \hat{l} \in (0 : m), \hat{i} \in (0 : n) \). Let \( \hat{i}^* = \hat{i} + m - \hat{l} \). The two seaweeds passing through the current cell terminate at positions \( \{ \hat{i}^* - \frac{1}{2}, \hat{i}^* + \frac{1}{2} \} = \langle \hat{i}^* - 1 : \hat{i}^* + 1 \rangle \). As
Figure 6.1: A snapshot of Algorithm 3 (the periodic seaweed algorithm)

in Algorithm 1, we update a $2 \times 2$ induced permutation submatrix of $P$ as follows:

\[
\begin{align*}
&\text{if } a(\hat{l}) \neq b(\hat{i}) \text{ and } P(\cdot \mid i^* - 1 : i^* + 1) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \\
&\text{then } P(\cdot \mid i^* - 1 : i^* + 1) \leftarrow \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}
\end{align*}
\]

Recall that conceptually, this update is performed on an infinite periodic set of columns of $P$. However, in a column-period submatrix representation, it is sufficient to update just a single $2 \times 2$ submatrix. Note that the first update in an inner loop is always trivial: we have $a(\hat{l}) = b(\hat{i}_0)$, therefore $P$ remains unchanged.

At the end of the sweep, we have the output matrix

\[
P = P_{a,b}
\]

**Cost analysis.** Similarly to Algorithm 1, every cell processing step runs in time $O(1)$. Therefore, the overall running time is $O(mp)$.

The memory cost is dominated by storing the input and the period submatrix of the current matrix $P$. Therefore, the overall memory cost is $O(m + p)$.

**Example 22** Figure 6.1 shows a snapshot of Algorithm 3, using the same conventions as Figure 4.1. The seaweed braid is laid out on a period subdag $G_{a,u}$; a seaweed leaving the subdag on the right is assumed to wrap around and enter the next period subdag at the corresponding point on the left. Note that the two seaweeds meeting in the current cell have both wrapped around from the previous period subdag, where they did cross. Therefore, the current step will leave the seaweed braid unchanged, so that double crossing does not occur.

The dag sweeping order in Algorithm 3 is significantly more restricted than in Algorithm 1, due to the extra data dependencies caused by the wraparound. This seems to rule out the possibility of a micro-block version of the algorithm.
6.2 Tandem alignment

The periodic LCS problem has many variations that can be solved by an application of the periodic seaweed algorithm.

The first such variation is the tandem LCS problem. The problem asks for the LCS score of a string $a$ of length $m$ against a tandem $k$-repeat string $b = u^k$ of length $n = kp$. As before, we assume that every character of $a$ occurs in $u$ at least once; we may also assume that $k \leq m$.

The tandem LCS problem can be solved naively by considering the LCS problem directly on strings $a$ and $b$, ignoring the periodic structure of string $b$. The standard dynamic programming algorithm (see Section 4.1) solves the problem in time $O(mn) = O(mkp)$. This running time can be slightly improved by the micro-block method (see Section 4.2).

The tandem LCS problem can also be regarded as a special case of the common-substring LCS problem [91, 39] (see also Section 4.4). Using this technique, the problem can be solved in time $O(m(k+p))$. The techniques of Landau et al. [39, 88] give an algorithm for the tandem LCS problem, parameterised by the LCS score of the input strings; however, the worst-case running time of this algorithm is still $O(m(k+p))$. Landau [87] asked if the running time for the tandem LCS problem can be improved to $O(m(\log k + p))$.

We now give an algorithm that improves on the current algorithms in time and functionality, and even exceeds Landau’s expectation. First, we call Algorithm 3 on strings $a$ and $u$. Then, we count the number of nonzeros $\preceq$-dominated by point $(0,n)$, i.e. nonzeros in the submatrix $P_{a,b}(0:+\infty | -\infty: n)$. Given the (say) column-period submatrix $P_{a,b}(0: p | ·)$, this can be done by a sweep of its $p$ nonzeros, counting every nonzero with appropriate multiplicity. More precisely, every nonzero $P_{a,b}(i,j) = 1$, $i \in (0:p)$, $j \in (-\infty: \infty)$, is counted with multiplicity $k - |j/p|$, if $j \in (0:n)$, and is skipped (counted with multiplicity 0) otherwise. The solution to the tandem LCS problem is then obtained by Theorem 11. The overall running time is dominated by the call of Algorithm 3, which runs in time $O(mp)$.

Another set of variations on the periodic LCS problem was introduced by Benson [14] as the tandem alignment problem. Instead of asking for all string-substring LCS scores of $a$ against $b = u^{\pm\infty}$, the tandem alignment problem asks for a substring of $b$ that is closest to $a$ in terms of alignment score (or edit distance), under different restrictions on the substring. In particular:

- the pattern global, text global (PGTG) tandem alignment problem restricts the substring of $b$ to consist of a whole number of copies of $u$, i.e. to be of the form $u^k = uu\ldots u$ for an arbitrary integer $k$;

- the tandem cyclic alignment problem restricts the substring of $b$ to be of length $kp$ for an arbitrary integer $k$ (but it may not consist of a
whole number of copies of $u$);

- the pattern local, text global (PLTG) tandem alignment problem leaves the substring of $b$ unrestricted.

All three versions of the tandem alignment problem can be regarded as special cases of the approximate pattern matching problem (see Section 5.2) on strings $a$ of length $m$ and $b' = u^m$ of length $n = mp$ (but with the roles of the text and the pattern reversed with respect to Benson’s terminology). Therefore, the tandem LCS problem can be solved naively by considering the approximate pattern matching problem directly on strings $a$ and $b'$, ignoring the periodic structure of string $b'$. Given an arbitrary (real) set of alignment weights, the classical algorithm by Sellers (see Section 5.2) solves the problem in time $O(mn) = O(m^2p)$. For a rational set of weights, the running time can be slightly improved by the micro-block precomputation method (see Section 5.2).

The PGTG and PLTG tandem alignment problems can be solved more efficiently by the technique of wraparound dynamic programming [101, 53] (see also [14]) in time $O(mp)$. For the tandem cyclic alignment problem, Benson [14] modified this technique to give an algorithm running in time $O(mp \log p)$ and memory $O(mp)$.

We now give a new algorithm for the tandem cyclic alignment problem, which improves on the existing algorithm in running time, assuming a rational set of alignment weights. The running time of the new algorithm matches the current algorithms for the PGTG and PLTG tandem alignment problems.

Given input strings $a, u$, we first solve the periodic string-substring problem by calling Algorithm 3. This gives us a period submatrix of matrix $P_{a,b}$, where $b = u^{\pm \infty}$. Then, for each $k$, $0 < k < m$, we perform independently the following procedure. We solve the tandem LCS problem for strings $a$ and $u^k$ by the method described earlier in this section, counting every nonzero in the period submatrix $P_{a,b}$ with an appropriate multiplicity. This gives us the LCS score for $a$ against $u^k$ for every $k$. We then update this score incrementally, obtaining the LCS score for string $a$ against a window of length $p$ in $b$, sliding through $p$ successive positions. This is equivalent to querying $p$ successive elements in a diagonal of matrix $P_{a,b}$, which can be achieved by $2p$ incremental dominance counting queries. By Theorem 2, every one of these queries can be performed in time $O(1)$.

The call to Algorithm 3 runs in time $O(mp)$; its output is shared by the tandem LCS computation for all $k$. For each $k$, the running time of the remaining computation is $O(p)$. Therefore, the combined running time for all values of $k$ is $m \cdot O(p) = O(mp)$. Overall, the algorithm runs in time $O(mp)$.
Chapter 7

Permutation string comparison

7.1 Semi-local LCS between permutations

An important special case of string comparison is where each of the input strings \(a, b\) is a permutation string, i.e. a string that consists of all distinct characters. Without loss of generality, we may assume that \(m = n\), and that both strings are permutations of a given totally ordered alphabet of size \(n\). The semi-local LCS problem on permutation strings is equivalent to the following classical problem.

**Definition 29** Given a string \(a\), the longest increasing subsequence (LIS) problem asks for the length of the longest string that is an increasing subsequence of \(a\).

The LIS problem has a long history, going back to Erdős and Szekeres [50] and Robinson [112]. Later, Knuth [82], Fredman [55] and Dijkstra [47] gave algorithms running in time \(O(n \log n)\). The problem was studied further by Chang and Wang [32] and by Bespamyatnikh and Segal [17].

The semi-local LCS problem on permutation strings is equivalent to solving the LIS problem in every substring of a given string. In the rest of this section, we give an efficient algorithm for this problem.

For consistency with the notation in previous chapters, we will assume that a permutation string of length \(n\) is indexed by odd half-integers \(\langle 0 : n \rangle\), and is over the alphabet \(\langle 0 : n \rangle\), unless indicated otherwise. The identity permutation string of length \(n\) is the string \(id = (\frac{1}{2}, \frac{3}{2}, \ldots, n - \frac{1}{2})\).

Given a string \(a\), we denote its reverse string by \(\bar{a}\). In particular, the reverse identity permutation string is \(\bar{id} = (n - \frac{1}{2}, n - \frac{3}{2}, \ldots, \frac{1}{2})\). We denote by \(\Sigma(a)\) the set of characters appearing in \(a\) at least once. For a set of characters \(S\), we denote by \(a|_S\) the filtered subsequence of \(a\), which consists only of those characters that belong to \(S\).
Algorithm 4 (Semi-local LCS between permutation strings)

**Input:** permutation strings $a, b$ of length $n$ over an alphabet of size $n$.

**Output:** nonzeros of the semi-local seaweed matrix $P_{a,b}$.

**Description.** Recursion on $n$.

Recursion base: $n = 1$. The computation is trivial.

Recursive step: $n > 1$. Assume without loss of generality that $n$ is even. We partition the input string $a$ into a concatenation $a = a' a''$ of two strings of length $\frac{n}{2}$. Each of the strings $a', a''$ is a permutation string of length $\frac{n}{2}$.

The semi-local seaweed matrices $P_{a',b}, P_{a'',b}$, are each over $\langle -\frac{n}{2} : 0 : \frac{3n}{2} \rangle$, and each contain $\frac{3n}{2}$ nonzeros. The semi-local seaweed matrix $P_{a,b}$ is over $\langle -n : n | 0 : 2n \rangle$, and contains $2n$ nonzeros.

Note that for all $i \in \langle 0 : n \rangle$, we have $P_{a',b}(i, i) = 1$, whenever $b(i) \not\in \Sigma(a')$.

More formally, let $\langle 0 : n \rangle = I' \cup I''$, where

$I' = \{ i \in \langle 0 : n \rangle, \text{ such that } b(i) \in \Sigma(a') \}$

$I'' = \{ i \in \langle 0 : n \rangle, \text{ such that } b(i) \in \Sigma(a') \}$

Sets $I', I''$ can be computed easily from strings $a', a'', b$, at the cost of sorting their character sets. We have a decomposition

$$P_{a',b} = P_{a',b}(\langle -\frac{n}{2} : 0 \rangle \cup I' | I' \cup \langle n : \frac{3n}{2} \rangle) \sqcup Id(I'' | I'')$$

The two permutation matrices in the above decomposition are of size $n$ and $\frac{n}{2}$, respectively. Only the $n$ nonzeros in the former matrix are non-trivial; they can be obtained by solving recursively the semi-local LCS problem on strings $a'$ and $b|_{\Sigma(a')} = b(I')$, both of which are permutations strings with alphabet size $\frac{n}{2}$. Similarly, only $n$ out of $\frac{3n}{2}$ nonzeros of $P_{a'',b}$ are non-trivial; they can be obtained by solving recursively the semi-local LCS problem on strings $a''$ and $b|_{\Sigma(a'')} = b(I'')$.

Finally, given matrices $P_{a',b}, P_{a'',b}$, the output matrix $P_{a,b}$ is computed by a call to the algorithm of Corollary 3, which calls the algorithm of Theorem 8 as a subroutine. Note that we now have two nested recursions: the current recursion, and the recursion of Theorem 8.

(End of recursive step)

**Cost analysis.** The recursion tree is a balanced binary tree of height $\log n$.

In the root node, the running time is dominated by the call to the algorithm of Corollary 3, and is therefore $O(n \log n)$. In each subsequent level of the recursion tree, the number of nodes doubles, and the running time per node is reduced by at least a factor of 2. Therefore, the running time per level is $O(n \log n)$. The overall running time is $\log n \cdot O(n \log n) = O(n \log^2 n)$. ■
Example 23 Figure 7.1 shows a series of snapshots of an execution of Algorithm 4 on permutation strings $a = \text{"cfaedhgb"}$, $b = \text{"dehcbafg"}$. □

By keeping the algorithm’s intermediate data, we obtain a data structure that allows efficient traceback of any semi-local LCS query on a pair of permutations, in time proportional to the size of the output (i.e. the length of the output subsequence). A related problem of tracing back LIS in every substring of a fixed size in a permutation has been studied by Albert et al. [6] and by Chen et al. [35]. In particular, work [35] gives an algorithm that runs in time proportional to the size of the output (i.e. the combined lengths of all the output subsequences). In the same work, the algorithm is also generalised for tracing back the LIS in an arbitrary subset of $n$ substrings, possibly of different sizes. In both versions of the problem, the size of the output, and therefore the algorithm’s running time, can be as high as $\Theta(n^2)$. In contrast, Algorithm 4 allows efficient LIS traceback for any individual substring.

7.2 Cyclic LCS between permutations

The cyclic LCS problem has been defined in Section 4.5. Given permutation strings $a$, $b$ of length $n$, the cyclic LCS problem on $a$, $b$ is equivalent to the LIS problem on a circular string.

This problem has been considered by Albert et al. [5], who gave a Monte Carlo randomised algorithm, running in time $O(n^{1.5} \log n)$ with small error probability.

We now give a new algorithm for cyclic LCS between permutations, that improves on the above algorithm both in running time, and by being deterministic. First, we call Algorithm 4, obtaining the semi-local seaweed matrix $P_{a,b}$. Then, we run the algorithm of Corollary 3 on matrix $P_{a,b}$ against itself, obtaining the semi-local seaweed matrix $P_{aa,b}$. Finally, we perform $n$ substring-string LCS queries for every substring of $aa$ of length 71.
$n$ against string $b$. The overall running time is dominated by the call to Algorithm 4, which runs in time $O(n \log^2 n)$.

A version of the cyclic LCS problem between permutations, parameterised by the output LCS length $l$, has also been considered by Albert et al. [5], who gave an algorithm running in time $O(nl \log n)$. This was improved upon by Deorowicz [46], who gave an algorithm running in time $O\left(\min(nl, n \log n + l^3 \log n)\right)$. Our algorithm described above improves on the algorithm of [46], unless $l = o((n \log n)^{1/3})$.

### 7.3 Longest pattern-avoiding subsequences

Two given permutation strings $a$, $b$ of equal length (but generally over different alphabets) are called isomorphic, if they have the same relative order of characters, i.e. $a(i) < a(j)$ iff $b(i) < b(j)$ for all $i$, $j$. Given a target permutation string $t$ of length $n$ and a pattern permutation string $p$ of fixed length, the longest $p$-isomorphic subsequence problem, or simply the longest $p$-subsequence problem, asks for the longest subsequence of $t$ that is isomorphic to $p$. More generally, given a set of pattern permutation strings $X$, the longest $X$-subsequence problem asks for the longest subsequence of $t$ that is isomorphic to any pattern string in $p$.

**Example 24** The LIS problem can be interpreted as the longest $X$-subsequence problem, where $X$ is a set of identity permutation strings, one of each length $m \in [1 : n]$.

Given a set of antipattern permutation strings $Y$, the longest $Y$-avoiding subsequence problem asks for the longest subsequence of $t$ that does not contain a subsequence isomorphic to any string in $Y$.

**Example 25** The LIS problem on a permutation string can be interpreted as the longest \{"21"\}-avoiding subsequence problem.

For a detailed introduction into these problems and their connections, see the work by Albert et al. [4] and references therein.

The LIS problem is the only nontrivial example of the longest $Y$-avoiding subsequence problem with antipatterns of length 2. Albert et al. [4] gave the full classification of the longest $Y$-avoiding subsequence problem for all sets of antipatterns of length 3. There are 10 non-trivial sets of such antipatterns. For each of these sets, the algorithms given in [4] run in polynomial time, ranging from $O(n \log n)$ to $O(n^5)$. Two particular antipattern sets considered in [4] are (in that work’s original notation):

\[
C_3 = \{\text{“132”, “213”, “321”}\}
\]

\[
C_4 = \{\text{“132”, “213”, “312”}\}
\]
For both these antipattern sets, algorithms given in [4] run in time $O(n^2 \log n)$.

We now give new algorithms for the longest $C_3$- and $C_4$-avoiding subsequence problems, improving on the above algorithms in running time.

Permutation strings that are $C_3$-avoiding are all cyclic rotations of an increasing permutation string. The longest such subsequence in the target string can be found by the algorithm for the cyclic LCS problem between permutations (Section 7.2), running in time $O(n \log^2 n)$.

Permutation strings that are $C_4$-avoiding are all obtained from an increasing permutation string by reversing some suffix. The longest such subsequence in the target string can be found as follows. Let the target string $t$ be over the alphabet $\langle 0 : n \rangle$. First, we call the standard LIS algorithm on $t$, obtaining explicitly prefix-prefix LCS scores $lcs(t \uparrow (i + \frac{1}{2}), id \downarrow (t(i) + \frac{1}{2})) = lcs(t \uparrow (i - \frac{1}{2}), id \downarrow (t(i) - \frac{1}{2})) + 1$ for all $i \in \langle 0 : n \rangle$. Independently, we call Algorithm 4 on $t$ against the reverse identity permutation $id$, and use Theorem 1 to process its output into a data structure that allows efficient queries of all suffix-prefix LCS scores $lcs(t \downarrow k, id \uparrow l)$ for all $k, l \in [0 : n]$. Finally, we obtain the solution to the longest $C_4$-avoiding subsequence problem as

$$\max_{i \in [0:n]} \left( lcs(t \uparrow (i + \frac{1}{2}), id \downarrow (t(i) + \frac{1}{2})) + lcs(t \downarrow (i + \frac{1}{2}), id \uparrow (t(i) + \frac{1}{2})) \right)$$

The overall running time is dominated by the call to Algorithm 4, which runs in time $O(n \log^2 n)$.

### 7.4 Longest piecewise monotone subsequences

The classical LIS problem asks for the longest increasing (or, equivalently, decreasing) subsequence in a permutation string. A natural generalisation is to ask for the longest subsequence that consists of a constant number of monotone pieces. In particular, given a permutation string $a$ of length $n$, the longest $k$-increasing subsequence (respectively, longest $k$-modal subsequence) problem asks for the longest subsequence in $a$ that is a concatenation of at most $k$ sequences, all of which are increasing (respectively, alternate between increasing and decreasing). In the case of the longest $k$-modal subsequence problem, we assume without loss of generality that $k$ is even. Both problems can be solved as an instance of the LCS problem, comparing the input permutation string $a$ against string $id^k$, i.e. the concatenation of $k$ copies of the identity permutation $id$ (respectively, against string $(id \overline{id})^{k/2}$, i.e. the concatenation of $k$ alternating copies of $id$ and its reverse $\overline{id}$). The resulting alignment dag is of size $n \times kn$, and contains $kn$ match cells. Using standard sparse LCS algorithms [69, 10], such an instance of the LCS problem can be
solved in time $O(nk \log n)$. Demange et al. [45] gave a similar algorithm for the longest $k$-modal subsequence problem, also running in time $O(nk \log n)$.

We now give a new algorithm for the longest $k$-increasing subsequence and the longest $k$-modal subsequence problems, improving on the above algorithms in running time, unless $k$ is very small.

Our algorithm is as follows. In the case of the longest $k$-increasing subsequence problem, we run Algorithm 4, obtaining the semi-local seaweed matrix $P_{id,a}$. Then, we extract the string-substring seaweed matrix $P_{id,a}$ and run on it $\log k$ times the algorithm of Corollary 1, obtaining the string-substring seaweed matrix $P^\text{string}_{id,k,a}$. In the case of the longest $k$-modal subsequence problem, we assume without loss of generality that $k$ is even. We run Algorithm 4 twice, obtaining the semi-local seaweed matrices $P_{id,a}$ and $P_{id,a}^\text{string}$, from which we obtain the matrix $P_{id,a}$ by Corollary 3. Then, we extract the string-substring seaweed matrix $P_{id,a}$, and run on it $\log k - 1$ times the algorithm of Corollary 1, obtaining the string-substring seaweed matrix $P_{id,a}^\text{string}_{id,k/2,a}$. Finally, for both problems we use the resulting string-substring seaweed matrix to query the global LCS score. The described algorithm runs in time $O(n \log^2 n) + \log k \cdot O(n \log n) = O(n \log^2 n)$. This is an improvement on both the sparse LCS approach and the algorithm of [45], unless $k = O(\log n)$.

7.5 Maximum clique in a circle graph

A circle graph [51, 60] is defined as the intersection graph of a set of chords in a circle, i.e. the graph where each node represents a chord, and two nodes are adjacent, whenever the corresponding chords intersect. We consider the maximum clique problem on a circle graph.

The interval model of a circle graph is obtained by cutting the circle at an arbitrary point and laying it out on a line, so that the chords become (closed) intervals. The original circle graph is isomorphic to the overlap
graph of its interval model, i.e. the graph where each node represents an interval, and two nodes are adjacent, whenever the corresponding intervals intersect but do not contain one another.

Example 26 Figure 7.2 shows an instance of the maximum clique problem on a six-node circle graph. Subfigure 7.2a shows the set of chords defining a circle graph, with one of the maximum cliques highlighted in bold red. The cut point is shown by scissors. Subfigure 7.2b shows the corresponding interval model; the dotted diagonal line contains the intervals, each defined by the diagonal of a square. The squares corresponding to the maximum clique are highlighted in bold red.

It has long been known that the maximum clique problem in a circle graph on \( n \) nodes is solvable in polynomial time [56]. A number of algorithms have been proposed for this problem [114, 67, 99, 9]; the problem has also been studied in the context of line arrangements in the hyperbolic plane [76, 49]. Given an interval model of a circle graph, the running time of the above algorithms is \( O(n^2) \) in the worst case, i.e. when the input graph is dense. In [122, 124], we gave an algorithm running in time \( O(n^{1.5}) \).

We now give a new algorithm for the maximum clique problem in a circle graph, improving on existing algorithms in running time. The algorithm is based on the fast matrix distance multiplication procedure of Theorem 8.

Our algorithm takes as input the interval model of a circle graph \( G \) on \( n \) nodes. Without loss of generality, we may assume that the set of interval endpoints is \( \langle 0 : 2n \rangle \). The interval model is represented by a permutation string \( a \) of size \( 2n \), where for each left (respectively, right) interval endpoint \( \hat{i} \in \langle 0 : 2n \rangle \), \( a(\hat{i}) \) is the corresponding right (respectively, left) endpoint. Note that for all \( \hat{i} < \hat{j} \), an interval with left endpoint \( \hat{i} \) does not contain an interval with left endpoint \( \hat{j} \), if and only if \( a(\hat{i}) < a(\hat{j}) \). Various alternative representations of interval models (e.g. the ones used in [114, 9]) can be converted to this representation in linear time.

In the interval model, a clique corresponds to a set of pairwise intersecting intervals, none of which contains another interval from the set. Recall that intervals in the line satisfy the Helly property: if all intervals in a set intersect pairwise, then they all intersect at a common point. In our context, we only need to consider integer indices as intersection points.

Consider a clique in \( G \). Let \( k \in [1 : 2n - 1] \) be a common intersection point of the intervals representing the clique, which is guaranteed to exist by the Helly property. Since the intervals representing the clique cannot contain one another, the sequence of their right endpoints is an increasing subsequence of \( a \). Let \( id \) be the identity permutation string of length \( 2n \). From the observations above, it follows that the clique corresponds to a common subsequence of a prefix \( a \upharpoonright k \) and a suffix \( id \upharpoonright k \). Therefore, the maximum clique can be solved as an instance of the semi-local LCS problem.
Algorithm 5 (Maximum clique in a circle graph)

**Input:** interval model of circle graph $G$, represented by permutation string $a$ of size $2n$.

**Output:** maximum-size clique of $G$, represented by the set of (say) left endpoints of the corresponding intervals.

**Description.**

First phase. We run Algorithm 4 on the input permutation string $a$ against the identity permutation string $id$, obtaining a semi-local seaweed matrix of size $4n$. We then build the data structure of Theorem 1 for querying semi-local LCS scores of $a$ against $id$.

Second phase. For each $k \in [1 : 2n - 1]$, we query the LCS score of prefix $a \upharpoonright k$ against suffix $id \downarrow k$. The maximum of the $2n$ returned values gives the size of the maximum clique in $G$, and the corresponding value $k$ gives a common intersection point of the clique intervals.

Third phase. The intervals defining the maximum clique can now be obtained by running a standard LIS algorithm on string $(a \upharpoonright k)\Sigma(id \downarrow k)$.

**Cost analysis.**

First phase. The running time of Algorithm 4 is $O(n \log^2 n)$.

Second phase. By Theorem 1, the combined running time of all the prefix-suffix queries is $O(n \log^2 n)$, if the queries are performed independently. This time can be reduced to $O(n)$ by observing that the queries can be performed as a single diagonal batch query.

Third phase. The LIS algorithm runs in time $O(n \log n)$.

Total. The overall running time is $O(n \log^2 n)$.

Like many algorithmic problems, the problem of finding a maximum clique in a circle graph admits various parameterised versions. Some relevant parameters are:

- the size $l$ of the maximum clique;
- the thickness $d$ of the interval model, i.e. the maximum number of intervals containing a point, taken across all points in the line;
- the number $e$ of graph edges.

For any interval model of a non-trivial circle graph, we have $l \leq d \leq n \leq e \leq n^2$. Notice that, given a permutation representing an interval model, its thickness can be found in time $O(n \log^2 n)$ by building a range tree on the corresponding set of planar points, and then performing $O(n)$ dominance counting queries.
Apostolico et al. [9] give algorithms for the parameterised version of the maximum clique problem in a circle graph, running in time $O(n \log n + e)$ and $O(n \log n + n\log(n/l))$. They also describe an algorithm for the maximum independent set problem, parameterised by the interval model’s thickness.

We now give a new algorithm for the maximum clique problem in a circle graph, parameterised by the thickness of the input interval model. Our algorithm improves on the parameterised algorithms of [9] for most values of the parameters. The algorithm is an extended version of Algorithm 5.

**Algorithm 6 (Maximum clique in a circle graph, parameterised by thickness)**

**Input:** interval model of circle graph $G$, represented by string $a$ of size $2n$.

**Output:** maximum-size clique of $G$, represented by the set of (say) left endpoints of the corresponding intervals.

**Parameter:** thickness $d$, $d \leq n$, of the input interval model.

**Description.**

**First phase.** We run Algorithm 4 on string $a \uparrow (r + 1)$ against string $id \downarrow rd$, independently for all $r \in [0 : \frac{2n}{d} - 1]$. As will be shown in the algorithm’s analysis, in each run we obtain an semi-local seaweed matrix with at most $4d$ non-trivial nonzeros. For every $r$, we then build the data structure of Theorem 1 for querying semi-local LCS scores of $a \uparrow (r + 1)$ against $id \downarrow rd$.

**Second phase.** For each $k \in [1 : 2n - 1]$, we query the LCS score of prefix $a \uparrow k = (a \uparrow (r + 1)d) \uparrow k$ against suffix $id \downarrow k = (id \downarrow rd) \downarrow (k - rd)$, where $r = \lfloor k/d \rfloor$. The maximum of the $2n$ returned values gives the size of the maximum clique in $G$, and the corresponding value $k$ gives a common intersection point of the clique intervals.

**Third phase.** The intervals defining the maximum clique can now be obtained by running a standard LIS algorithm on string $(a \uparrow k)|\Sigma(id \downarrow k)$.

**Cost analysis.**

**First phase.** We have $a \uparrow (r + 1)d = (a \uparrow rd)\{(a \uparrow rd) \uparrow d\}$. The alignment dag of $a \uparrow rd$ against $id \downarrow rd$ contains at most $d$ match cells, since every match corresponds to an interval containing point $rd$, and there can be at most $d$ such intervals by the definition of thickness. The alignment dag of $(a \uparrow rd) \downarrow d$ against $id \downarrow rd$ also contains at most $d$ match cells, since the length of the string $(a \uparrow rd) \downarrow d$ is $d$. The alignment dag of $a \uparrow (r + 1)d$ against $id \downarrow rd$ is the composition of the above two alignment dags, and therefore contains at most $d + d = 2d$ matches. Therefore, the time for each run of Algorithm 4 is $O(d \log^2 d)$, and the overall running time of this phase is $O(n/d \cdot d \log^2 d) = O(n \log^2 d)$.

**Second phase.** By Theorem 1, the combined running time of all the prefix-suffix queries is $O(n \log^2 d)$, if the queries are performed independently. This
time can be reduced to $O(n/d \cdot d) = O(n)$ by observing that the queries can be performed as a single diagonal batch query. 

**Third phase.** The alignment dag of $a \uparrow k$ against $id \downarrow k$ contains at most $d$ matches, since every such match corresponds to an interval containing point $k$. Therefore, string $(a \uparrow k)_{|\Sigma(id \downarrow k)}$ has length at most $d$. The LIS algorithm runs in time $O(d \log d)$. 

**Total.** The resulting overall running time is $O(n \log^2 d)$. 

Algorithm 6 improves on the $O(n \log n + e)$ algorithm of [9], unless $e = o(n \log^2 d) = O(n \log^2 n)$. It also improves on the $O(n \log n + nl \log(n/l))$ algorithm of [9], unless $l = o(l \log(n)) = O(\log(n))$.

### 7.6 Maximum common pattern between linear graphs

The concept of a linear graph, introduced by Davydov and Batzoglou [44], is similar to an interval model of a circle graph defined in Section 7.5. The interval relations of disjointness, containment and overlapping are denoted respectively by symbols $<$, $\sqsubset$ and $\sqsupset$. A pattern in a linear graph is defined as an ordered subset of intervals, all of which satisfy pairwise a prescribed subset of relations.

Fertin et al. [52] considered the maximum common $S$-structured pattern (S-MCSP) problem. The problem asks for the maximum common pattern in a set of $n$ linear graphs, each defined by at most $m$ intervals, where the structure of the common pattern is restricted by a prescribed subset of relations $S \subseteq \{<, \sqsubset, \sqsupset\}$. In particular, the $\{\sqsupset\}$-MCSP problem asks for the maximum commonly-structured subset of pairwise overlapping intervals; for $n = 1$ this is equivalent to finding the maximum clique of a circle graph, and for general $n$ is equivalent to finding the minimum-sized clique among maximum cliques of the $n$ input circle graphs. The $\{<, \sqsubset\}$-MCSP problem asks for the maximum commonly-structured subset of intervals, no two of which are overlapping; for $n = 1$ this is equivalent to finding the maximum independent set of a circle graph; however, for general $n$ the maximum commonly-structured independent set of the $n$ input circle graphs may be significantly different from (and smaller than) each of the $n$ individual maximum independent sets. The $\{<, \sqsubset, \sqsupset\}$-MCSP problem asks for the maximum commonly-structured subset of intervals without any a priori restriction on its structure.

Extending and generalising a number of previous results, paper [52] considers the $S$-MCSP problem, where $S$ runs over all seven nonempty subsets of $\{<, \sqsubset, \sqsupset\}$. For some of these seven variants, the algorithms use as a subroutine the algorithm of [122, 124] for the maximum clique problem in a circle graph. By plugging in the more efficient Algorithm 5, we can ob-
tain improved algorithms for those variants of the S-MCSP problem, where finding the maximum clique in a circle graph is a bottleneck.

In particular, the \{\emptyset\}-MCSP problem is solved in [52] by finding the maximum clique independently for \(n\) circle graphs, each corresponding to one of the input linear graphs, in overall time \(O(nm^{1.5})\). By plugging in Algorithm 5, the running time is improved to \(O(nm \log^2 m)\).

The \(<, \emptyset\>\)-MCSP problem is shown in [52] to be NP-hard, and to admit a polynomial-time \(2h(k)\)-approximation, where \(h(k) = \sum_{i=1}^{k} 1/i = \ln n + O(1)\); for the rest of this section, \(k\) denotes the size of the optimal solution to the problem. The approximation is obtained by \(O(nm)\) calls to the following subroutine: given a circle graph of size \(m\), and two integers \(m_1, m_2\), decide whether the graph contains \(m_1\) disjoint cliques, each of size \(m_2\). This subroutine is performed in time \(O(m^{2.5} \log m)\), and therefore the overall running time is \(O(nm) \cdot O(m^{2.5} \log m) = O(nm^{3.5} \log m)\). By a straightforward extension of Algorithm 5, the running time of the subroutine is improved to \(O(m \log^2 m)\), and therefore the overall running time of the approximation algorithm is improved to \(O(nm) \cdot O(m \log^2 m) = O(nm^2 \log^2 m)\).

The \([\emptyset, <, \emptyset]\)-MCSP problem is also shown in [52] to be NP-hard, and to admit a polynomial-time \(k^{1/2}\)-approximation. The approximation is obtained by combining exact solutions for the \([\emptyset, <, \emptyset]\)-MCSP and \(<, \emptyset\>-MCSP problems on the same input sets. The exact solution for the \(<, \emptyset\>-MCSP is the bottleneck; by plugging in the improved algorithm for this problem described above, the running time of the approximation algorithm for the \([\emptyset, <, \emptyset]\)-MCSP problem is improved from \(O(nm^{1.5})\) to \(O(nm \log^2 m)\).

Finally, paper [52] argues that the \(<, [\emptyset, <, \emptyset]\>-MCSP problem is NP-hard, and gives several polynomial-time approximation algorithms. In particular, it gives an \(O(k^{2/3})\)-approximation algorithm running in time \(O(nm^{1.5})\), and an \(O((k \log k)^{1/2})\)-approximation algorithm running in time \(O(nm^{3.5} \log m)\). By using the techniques described above, the running times of these approximation algorithms are improved respectively to \(O(nm \log^2 m)\) and \(O(nm^2 \log^2 m)\).
Chapter 8

Compressed string comparison

8.1 Grammar-compressed strings

String compression is a classical area of computer science. It is natural to ask whether compressed strings can be processed efficiently without decompression. Early examples of such algorithms were given e.g. by Amir et al. [8] and by Rytter [115].

We consider the following general model of compression.

Definition 30 Let $t$ be a string of length $n$ (typically large). String $t$ will be called a grammar-compressed string (GC-string), if it is generated by a context-free grammar, also called a straight-line program (SLP). An SLP of length $\bar{n}$, $\bar{n} \leq n$, is a sequence of $\bar{n}$ statements. A statement numbered $k$, $1 \leq k \leq \bar{n}$, has one of the following forms:

\[
t_k = \alpha \quad \text{where } \alpha \text{ is an alphabet character}
\]

\[
t_k = t_i t_j \quad \text{where } 1 \leq i, j < k
\]

We identify every symbol $t_r$ with the string it represents; in particular, we have $t = t_{\bar{n}}$. In general, the plain string length $n$ can be exponential in the GC-string length $\bar{n}$.

Example 27 The Fibonacci string “abaabaabaaba” of length 13 can be represented by the following SLP of length 7:

\[
t_1 = 'b' \quad t_2 = 'a' \\
t_3 = t_2 t_1 \quad t_4 = t_3 t_2 \quad t_5 = t_4 t_3 \quad t_6 = t_5 t_4 \quad t_7 = t_6 t_5
\]

In general, a Fibonacci string of length $n$ can be represented by an SLP of length $\bar{n}$, where $n = F_{\bar{n}} = \left( \frac{1}{\sqrt{5}} - o(1) \right) \left( \frac{1 + \sqrt{5}}{2} \right)^{\bar{n}}$ is the $\bar{n}$-th Fibonacci number.

This example is borrowed from Hermelin et al. [65].
Kida et al. [79] introduced a more general compression model, called *collage systems*. Grammar compression is a equivalent to a subclass of collage systems called *regular*. As a special case, grammar compression includes the classical LZ78 and LZW compression schemes by Ziv, Lempel and Welch [134, 131]. Both these schemes can be expressed by an SLP that consists of three sections:

- in the first section, all statements are of the form $t_k = \alpha$;
- in the second section, all statements are of the form $t_k = t_i t_j$, where statement $j$ is from the first section;
- in the third section, all statements are of the form $t_k = t_{k-1} t_j$, where statement $j$ is from the second section.

It should also be noted that certain other compression methods, such as e.g. LZ77 [133] and run-length compression, do not fit directly into the grammar compression model.

The algorithms in this section will take as input a grammar-compressed text string $t$ of length $n$, generated by an SLP of length $\bar{n}$, and a plain pattern string $p$ of length $m$. We aim at algorithms with running time that is a low-degree polynomial in $m$, $\bar{n}$, but is independent of $n$ (which could be exponential in $\bar{n}$).

### 8.2 Global subsequence recognition

The global subsequence recognition problem has been defined in Section 3.1. We recall that the problem asks whether the text $t$ contains the pattern $p$ as a subsequence. It is a simple special case of the LCS problem.

In this section, we revisit the global subsequence recognition problem, now assuming a GC-text $t$ and a plain pattern $p$ as inputs. In this setting, the problem can be solved by the following simple folklore algorithm.

**Algorithm 7 (Global subsequence recognition)**

**Input:** SLP of length $\bar{n}$, generating text string $t$ of length $n$; plain pattern string $p$ of length $m$.

**Output:** an integer $k$, giving the length of the longest prefix of $p$ that is a subsequence of $t$. String $t$ contains $p$ as a subsequence, if and only if $k = m$.

**Description.** Recursion on the input SLP generating $t$.

Recursion base: $n = \bar{n} = 1$. The output value $k \in \{0, 1\}$ is determined by a single character comparison.

Recursive step: $n \geq \bar{n} > 1$. Let $t = t' t''$ be the SLP statement defining string $t$. Let $k'$ be the length of the longest prefix of $p$ that is a subsequence of $t'$. Let $k''$ be the length of the longest prefix of $p | k'$ that is a subsequence of
We call the algorithm recursively to obtain $k'$ and $k''$, and then return $k = k' + k''$.

(End of recursive step)

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

### 8.3 Three-way semi-local LCS

We recall from Chapter 4 that the LCS problem on a pair of plain strings can be solved in time $O(m n \log^2 n)$, assuming that $m \leq n$ and that $m$ and $n$ are reasonably close, by the micro-block method of Masek and Paterson [98]. The LCS problem on a pair of GC-strings has been considered by Lifshits and Lohrey [94], and proven to be NP-hard.

In this section, we revisit the LCS problem, now assuming a GC-text $t$ and a plain pattern $p$ as inputs. Recall that we aim at algorithms with running time independent of $n$ (which could be exponential in $\bar{n}$). This rules out any attempt at solving the full semi-local LCS problem, since the resulting semi-local seaweed matrix would require memory $O(m + n)$. However, we are still able to consider the three-way semi-local LCS problem, excluding the computation of LCS on substrings of $t$.

In the special case of LZ77 or LZW compression of the text, the algorithm of Crochemore et al. [40] solves the LCS problem in time $O(m \bar{n})$; in other words, LZ77 or LZW compression of one of the input strings only slows down the LCS computation by a polylogarithmic factor.

The general case of an arbitrary GC-text appears more difficult. A GC-text is a special case of a context-free language, which consists of a single string. Therefore, the LCS problem between a GC-text and a plain pattern can be regarded as a special case of the edit distance problem between a context-free language given by a grammar of size $\bar{n}$, and a pattern string of size $m$. For this more general problem, Myers [102] gave an algorithm running in time $O(m^3 \bar{n} + m^2 \cdot \bar{n} \log \bar{n})$. In [125], we gave an algorithm for the three-way semi-local LCS problem between a GC-text and a plain pattern, running in time $O(m^{1.5} \bar{n})$. Lifshits [93] asked whether the LCS problem in the same setting can be solved in time $O(m\bar{n})$.

A new algorithm for the three-way semi-local LCS problem can be obtained by an application of the techniques described in Chapter 3. The resulting algorithm improves on existing algorithms in running time, and approaches an answer to Lifshits’ question within a logarithmic factor.
Algorithm 8 (Three-way semi-local LCS)

Input: SLP of length $\bar{n}$, generating text $t$ of length $n$; plain pattern $p$ of length $m$.

Output: nonzeros of matrices $P_{p,t}^q, P_{p,t}^q, P_{p,t}^q$.

Description. First, we observe that, although the output matrices contain at most $m$ nonzeros, the index range of these nonzeros is of size $m + n$, which may be exponentially larger. To avoid an exponential growth of the indices, we will clean up the range by removing unused indices, and deleting the corresponding zero row-column pairs from the matrices. Formally, we describe this process as an order-preserving remapping of the index range.

First phase. Recursion on the input SLP generating $t$.

Recursion base: $n = \bar{n} = 1$. The output can be computed by a linear sweep of string $p$.

Recursive step: $n \geq \bar{n} > 1$. Let $t = t't''$ be the SLP statement defining string $t$. We call the algorithm recursively to obtain the nonzeros of matrices $P_{p,t'}^q, P_{p,t'}^q, P_{p,t'}^q, P_{p,t''}^q, P_{p,t''}^q, P_{p,t''}^q$. The total number of nonzeros in each matrix triple is between $m$ and $2m$.

Conceptually, these matrices are submatrices of $P_{p,t'}$ over $\langle -m : n' | 0 : m + n' \rangle$, and $P_{p,t''}$ over $\langle -m : n'' | 0 : m + n'' \rangle$. However, the actual remapped index range after the recursive calls is $\langle -m : 2m | 0 : 3m \rangle$ for both matrix triples. We now compute the composition seaweed matrices $P_{p,t}^q, P_{p,t}^q, P_{p,t}^q$ by Corollary 2. The total number of nonzeros in this matrix triple is again between $m$ and $2m$. Conceptually, these matrices are submatrices of $P_{p,t}$ over $\langle -m : n | 0 : m + n \rangle$. However, the actual remapped index range after the composition is $\langle -m : 4m | 0 : 5m \rangle$. Therefore, there are at least $2m$ indices $i \in \langle 0 : 4m \rangle$, such that the row $P_{p,t}^q(i, \ast)$ and the column $P_{p,t}^q(\ast, i)$ both contain only zeros. We now delete exactly $2m$ such rows and columns from the respective matrices, and remap the index range to $\langle -m : 2m | 0 : 3m \rangle$, while preserving the linear order of the indices.

(End of recursive step)

Second phase. We now have the nonzeros of the output matrices, remapped to the index range $\langle -m : 2m | 0 : 3m \rangle$. This is already sufficient to query the global LCS score, or substring-string LCS scores for $p$ against $t$. However, if explicit indices of the nonzeros in the output seaweed matrices are required, the index range can be remapped back to $\langle -m : n | 0 : m + n \rangle$ by reversing every remapping step in the recursion.
Cost analysis.

First phase. By Corollary 2, each recursive step runs in time $O(m \log m)$. There are $\bar{n}$ recursive steps in total, therefore the first phase runs in time $O(m \log m \cdot \bar{n})$.

Second phase. For each nonzero, the inverse remapping can be performed recursively in time $O(\bar{n})$. There are $m$ nonzeros in total, therefore the second phase runs in time $O(m \bar{n})$.

Total. The overall running time is $O(m \log m \cdot \bar{n})$.

Algorithm 8 provides, as a special case, an algorithm for the LCS problem between a GC-string and a plain string, running in time $O(m \log m \cdot \bar{n})$; the LCS score can easily be queried from any one of the algorithm’s three output matrices by Theorem 11. This running time should be contrasted with standard LCS algorithms on plain strings, running in time $O(\min_{\log(m+n)})$ [98, 40], and with the NP-hardness of the LCS problem on two GC-strings [94].

Hermelin et al. [65] gave a more detailed analysis of the LCS problem on GC-strings, by considering the weighted alignment problem on a pair of GC-strings $a, b$ of total compressed length $\bar{r} = \bar{m} + \bar{n}$, parameterised by the strings’ total plain length $r = m + n$. They gave an algorithm running in time $O(r^{1.34} \bar{r}^{1.34})$ for general weights, and in time $O(r^{1.2} \bar{r}^{1.4})$ for rational weights.

In the case of rational weights, the parameterised running time of weighted GC-string alignment can be improved by the following straightforward algorithm. First, we uncompress one of the input strings — say, string $b$. Then, we run Algorithm 8 on the GC-string $a$ as a text against the plain string $b$ as a pattern. The resulting running time is $O(m \log m \cdot \bar{n}) = O(r \log r \cdot \bar{r})$.

8.4 Local subsequence recognition

The local subsequence recognition problem was introduced in Section 3.1 as a special case of the semi-local LCS problem. Definition 27 in Section 5.2 described this problem as a variant of the approximate matching problem. In the context of local subsequence recognition, a substring of text $t$ is called a matching substring, if it contains the pattern $p$ as a subsequence. A matching substring will be called minimally matching, if it is inclusion-minimal, i.e. it has no proper matching substring.

We recall that, depending on the output filtering, local subsequence recognition can take the following forms: the minimum-window subsequence recognition problem, which asks for the locations of all substrings of $t$ that are minimally matching, and the fixed-window subsequence recognition problem, which asks for the locations of all the matching substrings of a fixed
length $w$. A combination of these two problems is the bounded minimal-window subsequence recognition problem, which asks for the locations of all the minimally matching substrings below a fixed length $w$.

Clearly, the output size for the described reporting versions of these problems may be exponential in $\bar{n}$; therefore, we have to parameterise the running time by the output size, which we denote by $\text{output}$. An algorithm for the reporting version of any of the above problems can typically be converted to solve the corresponding counting version of the problem. Such a counting algorithm, instead of reporting all the matching substrings, only returns their overall number. The running time of the counting versions for all algorithms described in this section will correspond to the running time of the reporting algorithm with $\text{output} = O(1)$.

The minimal-window, fixed-window and bounded minimal-window subsequence recognition problems for a GC-text against a plain pattern have been considered by Cégielski et al. [28]. For each problem, they gave an algorithm running in time $O(m^2 \log m \cdot \bar{n} + \text{output})$. In [125], we gave an algorithm improving the running time for these problems to $O(m^{1.5} \bar{n} + \text{output})$.

We now give a more efficient local subsequence recognition algorithm, based on Algorithm 8, which we extend as follows. In addition to the seaweed matrices $P_{p,t}$, $P_{p,t}$, $P_{p,t}$, we now also make use of the seaweed cross-matrix $P_{p,t',t''}$. We extend every recursive step by the reporting of minimally matching substrings that are cross-substrings in the current seaweed matrix composition.

**Algorithm 9 (Local subsequence recognition)**

**Input:** SLP of length $\bar{n}$, generating text $t$ of length $n$; plain pattern $p$ of length $m$.

**Output:** locations (or count) of minimally matching substrings in $t$.

**Description.** Similarly to Algorithm 8, index remapping has to be performed in the background in order to avoid an exponential growth of the indices. To simplify the exposition, we now assume constant-time index arithmetic, keeping the index remapping implicit.

**First phase.** Recursion on the input SLP generating $t$.

Recursion base: $n = \bar{n} = 1$. As in Algorithm 8, the seaweed matrices $P_{p,t}$, $P_{p,t}$, $P_{p,t}$ can be computed by a linear sweep of string $p$. String $t$ is matching, if and only if $m = 1$ and $t = p$; in this case, $t$ is also minimally matching.

Recursive step: $n \geq \bar{n} > 1$. Let $t = t't''$ be the SLP statement defining string $t$. We run a recursive step of Algorithm 8, obtaining the seaweed matrices $P_{p,t}$, $P_{p,t}$, $P_{p,t}$.
In addition, we obtain the seaweed cross-matrix $P_{p,t',t''}^s$ by Corollary 2. This matrix has exactly $m$ nonzeros. Let
\[ \mathcal{L} = \left\{ (i_1, j_1) \ll (i_2, j_2) \ll \cdots \ll (i_{s-\frac{1}{2}}, j_{s-\frac{1}{2}}) \right\} \]
be the $\ll$-chain of all $\ll$-maximal nonzeros in $P_{p,t',t''}^s$, where $s = |\mathcal{L}| \leq m$.

By Theorem 11, a substring $t(i : j)$ is matching, if and only if point $(i, j)$ is not $\ll$-dominated by any nonzeros in the seaweed matrix $P_{p,t}$. Recall that a substring $t(i : j)$ is a cross-substring, if $i \in [0 : n' - 1]$, $j \in [n' + 1 : n]$; in other words, a cross-substring consists of a non-empty suffix of $t'$ and a non-empty prefix of $t''$. A point $(i, j)$ corresponding to a cross-substring can only be $\ll$-dominated by nonzeros within the seaweed cross-matrix $P_{p,t',t''}^s$. Therefore, a cross-substring $t(i : j)$ is matching, if and only if point $(i, j)$ is not $\ll$-dominated by any of the nonzeros in $P_{p,t',t''}^s$, or, equivalently, by any point in $\mathcal{L}$.

Consider the set of all points in $[-m : n' \mid n' : m + n]$, not $\ll$-dominated by any point in $\mathcal{L}$. The $\ll$-minimal points in this set, excluding the irrelevant boundary points $([i_1], n')$ and $(n', \lceil j_{s-\frac{1}{2}} \rceil)$, are interleaved with the points of $\mathcal{L}$, and form themselves a $\ll$-chain of size $s - 1$:
\[ \mathcal{M} = \{ ([i_2], j_2) \ll ([i_3], j_3) \ll \cdots \ll ([i_{s-\frac{1}{2}}], j_{s-\frac{1}{2}}) \} \]
Let $i \in [0 : n' - 1]$, $j \in [n' + 1 : n]$. Then, a cross-substring $t(i : j)$ is minimally matching, if and only if $(i, j) \in \mathcal{M}$. The number of such points $(i, j)$ is at most $|\mathcal{M}| = m - 1$.

(End of recursive step)

Second phase. For every SLP symbol, we now have the locations of its minimally matching cross-substrings. Furthermore, every non-trivial substring of $t$ corresponds to a cross-substring for some SLP symbol, under an appropriate transformation of indices. By another recursion on the structure of the SLP, it is now straightforward to obtain either the locations or the count of all the minimally matching substrings in $t$.

Cost analysis.

First phase. As in Algorithm 8, each seaweed matrix composition runs in time $O(m \log m)$. The $\ll$-chains $\mathcal{L}$ and $\mathcal{M}$ can be obtained in time $O(m)$. Hence, the running time of a recursive step is $O(m \log m)$. There are $n$ recursive steps in total, therefore the whole recursion runs in time $O(m \log m \cdot n)$.

Second phase. For every SLP symbol, there are at most $m - 1$ minimally matching cross-substrings. Given the output of the first phase, the locations of all minimally matching substrings in $t$ can be reported in time $O(mn \cdot \text{output})$.

Total. The overall running time is $O(m \log m \cdot n + \text{output})$. ■
Example 28 Figure 8.1 shows a snapshot of a recursive step in the first phase of Algorithm 9. Subfigure 8.1a shows the seaweed cross-matrix $P_{p,t',t''}$; in this particular example, all its nonzeros belong to the string-substring seaweed matrix $P_{p,t',t''}$. Subfigure 8.1b shows the corresponding seaweed braid. Matrix $P_{p,t',t''}$ contains $m = 5$ nonzeros, shown by green bullets in Subfigure 8.1a, and by green seaweeds in Subfigure 8.1b. Out of these five nonzeros, three are $\leq$-maximal; they are shown by larger bullets (respectively, by thicker seaweeds). The three $\leq$-maximal nonzeros form the $\preceq$-chain $L$. Consequently, there are $3 - 1 = 2$ points in the interleaved $\preceq$-chain $M$. In Subfigure 8.1a, these two points are shown by asterisks; in Subfigure 8.1b, the corresponding two substrings of $t$ are shown by dotted brackets. The
interleaving between \(\ll\)-chains \(\mathcal{L}\) and \(\mathcal{M}\) is shown in Subfigure 8.1a by solid black lines. Both points of \(\mathcal{M}\) lie within the range \([0:n'-1 \mid n'+1:n]\), and therefore each of them corresponds to a minimally matching cross-substring in \(t\).

By Theorem 11, a substring in \(t\) is matching, if and only if the corresponding rectangle in the alignment dag is not pierced by a seaweed entering at its left-hand boundary and leaving at its right-hand boundary. Notice that the bracketed substrings of \(t\) in Figure 8.1b are exactly the two inclusion-minimal cross-substrings satisfying this property.

An algorithm for the fixed-window subsequence recognition problem can be obtained from Algorithm 9 as follows. Substrings \(t(i:j)\) of length \(w\) correspond to points \((i,j)\) lying on the diagonal \(j-i=w\) in the semi-local score matrix \(H_{p,t}\). Consider the set of all points on this diagonal, \(\ll\)-dominated by any point in the \(\ll\)-chain \(\mathcal{L}\), introduced in Algorithm 9. This set consists of a (not necessarily disjoint) union of \(s\) diagonal intervals

\[
\mathcal{U} = \bigcup_{\hat{u} \in (0:s)} \{(i, i + w) \mid i \in \left[\lceil \hat{i}_{\hat{u}} \rceil : \lfloor \hat{i}_{\hat{u}} \rfloor - w \right]\}
\]

where any interval of negative length is by convention considered empty. In every recursive step, the interval endpoints in the set \(\mathcal{U}\) can be computed in time \(O(m)\).

Let \(i \in [0:n'-1], i + w \in [n'+1:n]\). Then, a cross-substring \(t(i:i+w)\) is matching, if and only if \((i, i+w) \notin \mathcal{U}\). Therefore, each point corresponding to a cross-substring of \(t\) can be reported in constant time.

An algorithm for the bounded minimal-window subsequence recognition problem can be obtained from Algorithm 9 by discarding in every recursive step the minimally matching cross-substrings of length exceeding \(w\).

The overall running time of both the above modifications of Algorithm 9 is still \(O(m \log m \cdot \bar{n} + \text{output})\).

### 8.5 Threshold approximate matching

The threshold approximate matching problem was introduced by Definition 26 in Section 5.2. The problem assumes an alignment score with arbitrary weights. In the context of threshold approximate matching, a substring of text \(t\) is called a matching substring, if it has alignment score at least \(h\) against pattern \(p\) (alternatively, edit distance at most \(k\)), where \(h\) (respectively, \(k\)) is a fixed threshold.

Approximate pattern matching on compressed text has been studied by Kärkkäinen et al. [75]. For a GC-text of length \(\bar{m}\), an uncompressed pattern of length \(n\), and an edit distance threshold \(k\), the (suitably generalised) algorithm of [75] solves the threshold approximate matching problem in time
$O(m\bar{n}k^2 + \text{output})$. In the special case of LZ78 or LZW compression, the running time is reduced to $O(m\bar{n}k + \text{output})$. Bille et al. [21] gave an efficient general scheme for adapting an arbitrary threshold approximate matching algorithm to work on a GC-text. In particular, when the algorithms by Landau and Vishkin [90] and by Cole and Hariharan [37] are each plugged into their scheme, the resulting algorithm runs in time $O(\bar{n}(\min(mk,k^4 + m) + \bar{n}^2) + \text{output})$. In the special case of LZ78 or LZW compression, Bille et al. [18] show that the running time can be improved to $O(\bar{n} \cdot \min(mk,k^4 + m) + \text{output})$.

Using the techniques of the previous sections, we now show how the threshold approximate matching problem on a GC-text can be solved more efficiently, unless the edit distance threshold $k$ is very small. The algorithm extends Algorithms 8 and 9, and assumes an arbitrary rational-weighted alignment score. As in Algorithm 9, we assume for simplicity the constant-time index arithmetic, keeping the index remapping implicit.

**Algorithm 10 (Threshold approximate matching)**

**Parameters:** character alignment weights $w_{\oplus}$, $w_{\otimes}$, $w_{\ominus}$, assumed to be constant rationals.

**Input:** SLP of length $\bar{n}$, generating text string $t$ of length $n$; plain pattern string $p$ of length $m$; score threshold $h$.

**Output:** locations (or count) of matching substrings in $t$.

**Description.**

First phase. Recursion on the input SLP generating $t$.

To reduce the problem to an unweighted LCS score, we apply the blow-up technique described in Section 5.1. Consider the normalised weights (5.1), and define the corresponding blown-up strings $\tilde{p}$, $\tilde{t}$ of length $\tilde{m} = \nu m$, $\tilde{n} = \nu n$, respectively.

Recursion base: $n = \bar{n} = 1$. The seaweed matrices $P_{\tilde{p},\tilde{t}}$, $P_{\tilde{p},\tilde{t}',\tilde{t}}$, $P_{\tilde{p}',\tilde{t}}$ can be computed by Algorithm 1 (the seaweed algorithm) in $\nu$ linear sweeps of string $\tilde{p}$. Each of these matrices can be used to query the LCS score $H_{\tilde{p},\tilde{t}}(0,\nu)$ between $\tilde{p}$ and $\tilde{t}$. String $t$ is matching, if and only if the corresponding weighted alignment score $H_{p,t}(0,1)$ is at least $h$.

Recursive step: $n \geq \bar{n} > 1$. Let $t = t't''$ be the SLP statement defining string $t$. We have $\tilde{t} = \tilde{t}'\tilde{t}''$ for the corresponding blown-up strings.

As in Algorithm 9, we run a recursive step of Algorithm 8, obtaining the seaweed matrices

\[
P_{\tilde{p},\tilde{t}}', P_{\tilde{p},\tilde{t}}'', P_{\tilde{p}',\tilde{t}}', P_{\tilde{p}',\tilde{t}}''
\]

In addition, we obtain the seaweed cross-matrix $P_{\tilde{p},\tilde{t}',\tilde{t}'}$ by Corollary 2. This matrix has exactly $\tilde{m} = \nu m$ nonzeros.
In contrast to Algorithm 9, it is no longer sufficient to consider just the \( \leq \)-maximal nonzeros of \( P_{\tilde{p}, \tilde{r}, \tilde{p}'} \); we now have to consider all its \( \tilde{m} \) nonzeros. Let us denote the coordinates of these nonzeros, in increasing order independently for each coordinate, by

\[
i_1 < i_2 < \ldots < i_{\frac{m-1}{2}} \quad j_1 < j_2 < \ldots < j_{\frac{m-1}{2}}
\]

(8.1)

Every nonzero of the seaweed cross-matrix can now be represented as

\[
P_{\tilde{p}, \tilde{r}, \tilde{p}'}(i_s, j_t) = 1
\]

for some \( i_s, j_t \in (0 : \tilde{m}) \).

The first (respectively, second) of the index sequences (8.1) partitions the range \([-\tilde{m} : \tilde{n}'] \) (respectively, \([\tilde{n}' : \tilde{m} + \tilde{n}] \) into \( \tilde{m} + 1 \) disjoint intervals of varying size. Therefore, we have a partitioning of the score cross-matrix \( H_{\tilde{p}, \tilde{r}, \tilde{p}'} \) into \((\tilde{m} + 1)^2\) disjoint rectangular blocks of varying dimensions. Consider a particular block

\[
H_{\tilde{p}, \tilde{r}, \tilde{p}'}[\tilde{i}_u : \tilde{i}_v^+ | \tilde{j}_v : \tilde{j}_v^+]
\]

(8.2)

where

\[
i_0 = \lfloor \tilde{i}_u - \frac{1}{2} \rfloor \quad i_+ = \lceil \tilde{i}_u + \frac{1}{2} \rceil \quad j_0 = \lfloor \tilde{j}_v - \frac{1}{2} \rfloor \quad j_+ = \lceil \tilde{j}_v + \frac{1}{2} \rceil
\]

for \( u, v \in [0 : \tilde{m}] \). For the boundary blocks, where some of the above bounds are not defined, we let

\[
i_0 = -\tilde{m} \quad i_+ = \tilde{n}' \quad j_0 = \tilde{n}' \quad j_+ = \tilde{m} + \tilde{n}
\]

All the entries in the block (8.2) are \( \leq \)-dominated by a fixed set of nonzeros in \( P_{\tilde{p}, \tilde{r}, \tilde{p}'} \). Let \( d \) be the number of nonzeros in this set. By Theorem 11, all the entries within the block (8.2) have identical value: we have \( H_{\tilde{p}, \tilde{r}, \tilde{p}'}(i, j) = \tilde{m} - d \) for all \( i \in [i_u : i_u^+] , j \in [j_v : j_v^+] \).

We now switch our focus from the blown-up strings \( \tilde{p} , \tilde{r} , \tilde{p}' \) back to the original strings \( p , t' , t'' \). The partitioning of matrix \( H_{\tilde{p}, \tilde{r}, \tilde{p}'} \) induces a partitioning of matrix \( H_{p, t', t''} \) into \((m + 1)^2\) disjoint rectangular blocks of varying dimensions. The block corresponding to block (8.2) is

\[
H_{p, t', t''}[i_u^+ : i_v^+ | j_v : j_v^+]
\]

(8.3)

where

\[
i_u^- = \left\lfloor \frac{1}{p} \tilde{i}_u - \frac{1}{2} \right\rfloor \quad i_u^+ = \left\lceil \frac{1}{p} \tilde{i}_u + \frac{1}{2} \right\rceil \quad j_v^- = \left\lfloor \frac{1}{p} \tilde{j}_v - \frac{1}{2} \right\rfloor \quad j_v^+ = \left\lceil \frac{1}{p} \tilde{j}_v + \frac{1}{2} \right\rceil
\]

for \( u, v \in [0 : m] \). For the boundary blocks, where some of the above bounds are not defined, we let

\[
i_0^- = -m \quad i_+ = m' \quad j_0 = m' \quad j_+ = m + n
\]
Note that, although the block (8.2) is never empty, the corresponding block (8.3) may be empty. This will be the case if either $i_u^- > i_u^+$, or $j_v^- > j_v^+$.

We now show that the maximum entry across the block (8.3) always lies in its bottom-left corner. Recall that for the blown-up strings, we have $H_{\tilde{p},\tilde{t}',\tilde{t}''}(i,j) = \tilde{m} - d$ for all $i \in [\tilde{i}_u^- : \tilde{i}_u^+]$, $j \in [\tilde{j}_v^- : \tilde{j}_v^+]$. By (5.2), for the original strings we have

$$H_{p,t',t''}(i,j) = \tilde{m} - d \nu \cdot (w \oplus -2w \ominus) + (m + j - i) \cdot w \ominus$$

where $i \in [i_u^- : i_u^+]$, $j \in [j_v^- : j_v^+]$. Recall that $w_{\ominus} \leq 0$. Therefore, the score is maximised when $j - i$ is minimised, and the maximum score is attained by the block’s bottom-left entry $H_{p,t',t''}(i_u^+,j_v^-)$. If $w_{\ominus} < 0$, then this maximum is strictly greater than all other entries in the block; If $w_{\ominus} = 0$, then all the entries in the block, including the bottom-left entry, have an identical value $\tilde{m} - d \nu \cdot w_{\ominus}$.

Without loss of generality, we now assume that all the blocks (8.3) are non-empty. The bottom-left entries attaining block maxima, taken across all the blocks, form an $(\tilde{m} + 1) \times (\tilde{m} + 1)$ submatrix

$$M(u,v) = H_{p,t',t''}(i_u^+,j_v^-)$$

where $u,v \in [0 : n']$, $j \in [n' + 1 : n]$. Then, a cross-substring $t(i : j)$ is matching, if and only if $H_{p,t',t''}(i,j) \geq h$. Therefore, the set of all matching cross-substrings corresponds to the set $\tau_h(H_{p,t',t''}[0 : n' - 1 \mid n' + 1 : n])$ of all entries in the submatrix $H_{p,t',t''}[0 : n' - 1 \mid n' + 1 : n]$ scoring above the threshold $h$. This set can now be obtained by a local search in the neighbourhood of the bottom-left corner of each block corresponding to an element of $\tau_h(M)$. Any kind of filtering described in Section 5.2 can be applied to the output.

(End of recursive step)

**Second phase.** As in Algorithm 9, substituting “matching” for “minimally matching”.

**Cost analysis.**

**First phase.** As in Algorithm 8, each seaweed matrix composition runs in time $O(\tilde{m} \log \tilde{m}) = O(m \log m)$. The algorithm of Lemma 2 runs in time
$O(\tilde{m}) = O(m)$. Hence, the running time of a recursive step is $O(m \log m)$. There are $\tilde{n}$ recursive steps in total, therefore the whole recursion runs in time $O(m \log m \cdot \tilde{n})$.

**Second phase.** As in Algorithm 9, the locations of all matching substrings in $t$ can be found in time $O(m \tilde{n} + \text{output})$.

**Total.** The overall running time is $O(m \log m \cdot \tilde{n} + \text{output})$. ■

Algorithm 10 improves on the algorithm of [75], as long as $\log m = o(k^2)$ in the case of general GC-compression, and $\log m = o(k)$ in the case of LZ78 or LZW compression. Algorithm 10 also improves on the algorithms of [18, 21], as long as $m \log m = o(\min(mk, k^4 + m) + \tilde{n})$ in the case of general GC-compression, and $m \log m = o(\min(mk, k^4 + m))$ in the case of LZ78 or LZW compression.

**Example 29** Figure 8.2 shows a simplified snapshot of a recursive step in the first phase of Algorithm 10, which is assumed to run on the same input as in Figure 8.1. For simplicity, we assume the unweighted LCS score, hence $\nu = 1$, and the blown-up strings are identical to the original input strings. Subfigure 8.2a shows the seaweed cross-matrix $P_{p,t,t'}$. Subfigure 8.2b shows the corresponding seaweed braid. Both the seaweed cross-matrix and the seaweed braid are identical to the ones in Figure 8.1. In Subfigure 8.2a, the partitioning into a grid of $6 \times 6$ blocks is shown by solid lines. The bottom-left entry in each block, attaining its maximum score, is shown by an asterisk; taken together, all these entries form the $6 \times 6$ matrix $M$. In Subfigure 8.2b, the corresponding substring boundaries are shown by dotted lines (except the ones that coincide with the alignment dag boundaries).
Figure 8.2: A snapshot of Algorithm 10 (threshold approximate matching)
Chapter 9

Beyond semi-locality

9.1 Window-local LCS and alignment plots

So far, we have considered mostly global and semi-local string comparison. Our aim now is to approach local string comparison — the type of string comparison that is the most important for biological applications, but also the most difficult. In this chapter, we consider a version of local string comparison that is restricted to a fixed subset of prescribed substrings in one of the input strings, comparing them to all substrings in the other string.

An important special case is where all the prescribed substrings have equal length. Given a fixed parameter $w$, we call a substring of length $w$ a $w$-window in the corresponding string.

String comparison in windows has a long history. One of its early instances is dot plots (also known as diagonal plots or dot matrices), introduced by Gibbs and McIntyre [59] and by Maizel and Lenk [96]. In addition to numerical scores, dot plots provide a convenient visualisation of string comparison. In the context of dot plots, processing a pair of windows is usually referred to as filtering. The standard filtering method compares every $w$-window of string $a$ against every $w$-window of string $b$ in terms of their Hamming score, i.e. the count of matching character pairs under a rigid alignment model, where all characters must be aligned and no gaps are allowed. A Hamming-filtered dot plot can be computed in time $O(mn)$ by the algorithm of [96, 100]. This algorithm has been implemented in several software packages (see e.g. [120, 109, 36]). A faster suffix-tree based algorithm has been proposed and implemented by Krumsiek et al. [85]. Enhancement of the dot plot approach have been proposed by Huang and Zhang [68] and by Putonti et al. [108].

Numerous other methods of local string comparison have been proposed. The Smith–Waterman–Gotoh algorithm [119, 62] allows one to obtain the highest-scoring pair across all substring pairs in the input strings. It can also be generalised to report all substring pairs scoring above a certain threshold.
A significant drawback of the Smith–Waterman–Gotoh algorithm is that it generally favours long, less precise substring alignments over short, more precise ones (as noted e.g. by Arslan et al. [13]). The quality of the alignment is also 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 weighted alignment scores with sufficiently high penalties (negative score weights) for gaps.

In contrast with the Smith–Waterman–Gotoh algorithm, the dot plot method gives the user more flexibility to select the biologically significant substring alignments, by providing all the local scores between fixed-size windows of the input strings. However, the Hamming scoring scheme used by this method within each window pair is less sensitive than even the LCS score, and especially than the weighted alignment score used by Smith–Waterman–Gotoh. This tradeoff motivates us to combine the best features of the two approaches in the following definition.

**Definition 31** Given strings $a$, $b$, and a window length $w$, the window-window (respectively, window-substring) LCS problem asks for the LCS score of for every $w$-window in $a$ against every $w$-window (respectively, every substring) in $b$.

The window-window LCS problem can be seen as a refinement of the dot plot method and a complement to the Smith–Waterman–Gotoh 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, and is therefore potentially more sensitive. The method can be further extended to use weighted alignment scores. By analogy with Hamming-filtered dot plots, we call the resulting matrix of window-window alignment scores an alignment-filtered dot plot, or simply an alignment plot. Recently, we applied the alignment plot method to the detection of alignment-conserved regions in DNA [106].

The solution of the window-substring LCS problem can be represented in space $O(mn \log n)$ by the data structure of Theorem 1, built on the string-substring seaweed matrix for each window of $a$ against $b$. An individual window-substring LCS score query can be performed on this data structure by Theorem 1 in time $O(\log^2 n)$. The full explicit solution to the window-window LCS problem can be obtained in time $O(mn)$, by performing a diagonal batch query directly on each of the string-substring seaweed matrices. Thus, string-substring seaweed matrices provide a unified solution for both the window-substring and the window-window LCS problems.

A naive algorithm for the window-window LCS problem runs in time $O(mnw^2)$, and for the window-substring LCS problem runs in time $O(mn^3 w)$. The running time for both problems can be improved by applying Algorithm 1 (the seaweed algorithm) independently to each window of string
a against whole string b. The resulting algorithm runs in time $O(mnw)$. If window length $w$ is sufficiently large, the running time can be improved slightly by using Algorithm 2 (the micro-block seaweed algorithm).

We now give a new algorithm for the window-substring LCS problem (and therefore also for the window-window LCS problem as a special case). Our algorithm provides a further substantial improvement on the above approach, and matches the asymptotic running time of both the Hamming-scored dot plot and the Smith–Waterman–Gotoh methods.

**Algorithm 11 (Window-substring LCS)**

**Input:** strings $a$, $b$ of length $m$, $n$, respectively; window length $w$.

**Output:** nonzeros of the string-substring seaweed matrix for every $w$-window of string $a$ against full string $b$.

**Description.** Without loss of generality, we assume that $m$ is a power of 2. Let $s$ be an arbitrary power of 2, $1 \leq s \leq m$. We call a substring $a(i : j)$ **canonical**, if $j - i = s$, and $i$, $j$ are both multiples of $s$. In particular, both the whole string $a$, and every one of its individual characters, are canonical.
substrings. Every substring of \( a \) can be decomposed into a concatenation of \( O(\log m) \) canonical substrings.

In the following, by processing a substring \( a' \) of \( a \), we mean computing the string-substring seaweed matrix \( P_{a',b} \).

**First phase.** We process all canonical substrings of \( a \) by the following recursive procedure.

Recursion base: \( m = 1 \). Matrix \( P_{a,b} \) is computed by a linear sweep of string \( b \).

Recursive step: \( m > 1 \). We have \( a = a'a'' \), where \( a', a'' \) are canonical substrings of length \( m/2 \). We call the first phase procedure recursively on each of \( a', a'' \) against \( b \), obtaining matrices \( P_{a',b}, P_{a'',b} \). Then, we obtain the matrix \( P_{a,b} = P_{a',b} \odot P_{a'',b} \) by Corollary 1.

(End of recursive step)

**Second phase.** Let \( s \) be an arbitrary power of 2, \( 1 \leq s \leq m \). We define \((s,t)\)-window to be a substring \( a(i:j) \), such that \( j - i = st \), and \( i \) and \( j \) are both multiples of \( s \). Intuitively, \( t \) is the maximum number of windows that overlap at any single character in the string \( a \). Note that a \((1,t)\)-window is the same as a \(t\)-window, and an \((s,1)\)-window is the same as a canonical substring. Given parameters \( s, t \), we solve the problem of processing all \((s,t)\)-windows by the following recursive procedure.

Recursion base: \( t = 1 \). For any \( s \), all \((s,1)\)-windows are canonical substrings, therefore they have already been processed in the first phase.

Recursive step: \( t > 1 \). If \( t \) is even, then we call the second phase recursively to process all \((2s,t-1)\)-windows. We then process all \((s,t)\)-windows \( a(i:j) \) as follows:

\[
P_{a(i:j),b} = \begin{cases} 
P_{a(i:j-s),b} \odot P_{a(j-s:j),b} \\
P_{a(i+i+s),b} \odot P_{a(i+s:j),b} 
\end{cases}
\]

where the choice between the two cases is made arbitrarily.

If \( t \) is odd, then we call the second phase recursively to process all \((2s,\frac{t-1}{2})\)-windows. We then process all \((s,t)\)-windows \( a(i:j) \) by (9.1), where the first case is followed for \( \frac{i}{s} \) even and \( \frac{j}{s} \) odd, and the second case is followed for \( \frac{i}{s} \) odd and \( \frac{j}{s} \) even.

The implicit distance products in (9.1) are computed by Corollary 1. In each case, the product is between two string-substring seaweed matrices: one for substring \( a(i:j-s) \) or \( a(i+s:j) \), already processed by the recursive call, and the other for a canonical substring \( a(j-s:j) \) or \( a(i:i+s) \).

(End of recursive step)
Cost analysis.

First phase. In every subsequent level down the recursion, the number of matrix multiplications doubles. The running time for each matrix multiplication decreases by a factor \( o(1) \). Therefore, the running time is dominated by the bottom level of the recursion, where we have \( m/2 \) matrix multiplications, each running in time \( O(n) \). The running time of the whole phase is \( m/2 \cdot O(n) = O(mn) \).

Second phase. In every two consecutive levels down the recursion, the number of matrix multiplications is reduced by at least a factor of 2. The running time for each matrix multiplication increases by a factor \( o(1) \). Therefore, the running time is dominated by the top level of the recursion, where we have \( O(m) \) matrix multiplications, each running in time \( O(n) \). Therefore, the running time is \( O(m) \cdot O(n) = O(mn) \).

Total. The running time for both the first and the second phase, and therefore the overall running time, is \( O(mn) \).

Note that the asymptotic running time of Algorithm 11 is independent of the window length \( w \).

Example 30 Figure 9.1 shows an execution of Algorithm 11 on string \( a \) of length 16 with window size 7, against string \( b \) of arbitrary length. Subfigure 9.1a shows the canonical substrings of \( a \) of lengths 1, 2, 4, 8, 16 in black, and windows of length 7 in blue. For each window, the figure shows its decomposition into canonical substrings. For one of the windows, highlighted in thick red, the corresponding area is outlined in the alignment dag. Subfigure 9.1b represents substrings of \( a \) by points in the plane, and the decompositions into canonical substrings by a forest of trees. Each window \( a(i : j) \) corresponds to a leaf of a decomposition tree. The canonical substrings in a decomposition of the window correspond to the edges on the path from the corresponding leaf to the root of the tree. The leaves, internal nodes and roots of decomposition trees are shown respectively by diamond, circle and square bullets.

9.2 Quasi-local LCS

We now consider an arbitrary set of prescribed substrings of various lengths in string \( a \). We assume that all the prescribed substrings are non-empty, and denote their number by \( k \), \( m \leq k \leq \binom{m}{2} \).

Definition 32 Given strings \( a, b \), the quasi-local LCS problem asks for the LCS score of every prescribed substring in \( a \) against every substring in \( b \).
The quasi-local LCS problem includes as special cases the semi-local, window-window, window-substring and fully-local LCS problems, as well as length-constrained local alignment considered by Arslan and Egecioglu [12]. The solution of the quasi-local LCS problem can be represented in space $O(kn \log n)$ by the data structure of Theorem 1, built on the string-substring seaweed matrix for each prescribed substring of $a$ against $b$. An individual quasi-local LCS score query can be performed on this data structure in time $O(\log^2 n)$.

A naive algorithm for the quasi-local LCS problem runs in time $O(kmn^3)$. This running time can be improved by applying Algorithm 1 (the seaweed algorithm) independently to each prescribed substring $a$ against whole string $b$. The resulting algorithm runs in time $O(kmn)$. If all the prescribed substrings are sufficiently long, then the running time can be improved slightly by using Algorithm 2 (the micro-block seaweed algorithm).

We now give an algorithm for the quasi-local LCS problem that provides a further improvement on the above approach.

**Algorithm 12 (Quasi-local LCS)**

**Input:** strings $a$, $b$ of length $m$, $n$, respectively; a set of $k$ endpoint index pairs for the prescribed substrings in $a$. 
**Output:** nonzeros of the string-substring seaweed matrix for every prescribed substring of string $a$ against full string $b$.

**Description.** The algorithm structure is similar to the one of Algorithm 11.

*First phase.* As in Algorithm 11.

*Second phase.* We process all prescribed substrings of $a$ by the following recursive procedure. Let $s$ be a parameter, assumed to be a power of 2. Initially, we set $s = 1$. At every level of recursion, the endpoint indices of the prescribed substrings are multiples of $s$.

Recursion base: the set of prescribed substrings is empty. In this case, the problem is trivial.

Recursive step: there are some prescribed substrings. First, we remove from consideration all canonical prescribed substrings, since they have already been processed in the first phase. We then call the second phase procedure recursively with the parameter $2s$, and the following set of prescribed substrings. For each currently prescribed non-canonical substring $a⟨i,j⟩$, the corresponding new prescribed substring in the recursive call is $a⟨2s⌈i/2s⌉ : 2s⌊j/2s⌋⟩$, unless this substring is empty. Informally, we round $i$ and $j$ to the nearest multiple of $2s$; index $i$ is rounded up and index $j$ down. Note that different currently prescribed substrings may correspond to the same new prescribed substring in the recursive call.

The recursive call results in the processing of all the prescribed substrings $a⟨i,j⟩$ where $i, j$ are multiples of $2s$; in other words, where $\frac{i}{s}$ and $\frac{j}{s}$ are both even. We then process all remaining prescribed substrings $a⟨i:j⟩$ as follows:

$$
P_{a⟨i:j⟩,b} = \begin{cases}
P_{a⟨i:j−s⟩,b} \square P_{a⟨j−s:j⟩,b} & \text{if } \frac{i}{s} \text{ even, } \frac{j}{s} \text{ odd} \\
P_{a⟨i:i+s⟩,b} \square P_{a⟨i+s:j⟩,b} & \text{if } \frac{i}{s} \text{ odd, } \frac{j}{s} \text{ even} \\
P_{a⟨i:i+s⟩,b} \square P_{a⟨i+s:j−s⟩,b} \square P_{a⟨j−s:j⟩,b} & \text{if } \frac{i}{s} \text{ odd, } \frac{j}{s} \text{ odd}
\end{cases}
$$

The implicit distance products are computed by Corollary 1. In each case, the product is between two or three string-substring seaweed matrices: one for substring $a⟨i:j−s⟩$, $a⟨i+s:j⟩$ or $a⟨i+s:j−s⟩$, already processed by the recursive call; and the other one or two for canonical substrings $a⟨j−s:j⟩$ and/or $a⟨i:i+s⟩$.

(End of recursive step)

**Cost analysis.**

*First phase.* As in Algorithm 11, the total running time of this phase is $O(mn)$.

*Second phase.* In every level of the recursion, the number of matrix multiplications is at most $O(k)$. The running time for each matrix multiplication is at most $O(n \log m)$. The recursion has $\log m$ levels. Therefore, the running time is dominated by the top level of the recursion, where we have $O(m)$.
matrix multiplications, each running in time $O(n)$. Therefore, the running time of the whole phase is $O(k \log m \cdot n \log m) = O(kn \log^2 m)$.

For values of $k$ close to the fully-local case $k = \binom{m}{2} = O(m^2)$, a sharper analysis is possible. In this case, the running time of the whole phase is $O(m^2 n)$.

**Total.** The overall running time is dominated by the second phase, and is therefore $O(kn \log^2 m)$. For values of $k$ close to $\binom{m}{2}$, the running time is $O(m^2 n)$.

Note that in the fully-local case, the same asymptotic time can be obtained by running $m$ independent instances of the seaweed algorithm (Algorithm 1), each instance computing the implicit highest-score matrices incrementally for $O(n)$ different substrings of $a$.

**Example 31** Figure 9.2 shows an execution of Algorithm 11 on string $a$ of length 16, with 16 prescribed substrings of various sizes, against string $b$ of arbitrary length. Conventions are the same as in Figure 9.1.

### 9.3 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 such approaches is *spliced alignment* by Gelfand et al. [57] (see also [63]), 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 $m$, $n$ respectively. 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 [57] runs in time $O(m^2 n)$.

In general, the number of candidate exons $k$ may be as high as $\binom{m}{2} = O(m^2)$. 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. [78] give an algorithm for sparse spliced alignment that, in the special case $k = O(m)$, runs in time $O(m^{1.5} n)$. By a direct application of the quasi-local LCS problem (Section 9.2), the running time can be reduced to $O(mn \log^2 m)$. Sakai [116] gave an improved algorithm, running in time $O(mn \log n)$.

For higher values of $k$, all the described algorithms provide a smooth transition in running time to the dense case $k = \binom{m}{2}$. In this case, the algorithms’ running time $O(m^2 n)$ is asymptotically equal to the algorithm of [57].
We now describe an algorithm for sparse spliced alignment, based on the approach of [116]. We keep the notation and terminology of the previous sections; in particular, candidate exons are represented by prescribed substrings of string $a$. We say that substring $a(i': j')$ precedes substring $a(i'': j'')$, if $j' \leq i''$. A precedence chain of substrings is a chain in the partial order of substring precedence. We identify every precedence chain with the string obtained by concatenating all its component substrings in the order of precedence.

The algorithm uses a generalisation of the standard network alignment method, equivalent to the one used by [78]. For simplicity, we describe the algorithm under LCS score; using the blow-up technique of Section 5.1, the algorithm can be generalised to an arbitrary alignment score with rational weights.

**Algorithm 13 (Sparse spliced alignment)**

*Input:* strings $a$, $b$ of length $m$, $n$, respectively; a set of $k$ endpoint index pairs for the prescribed substrings in $a$.

*Output:* the precedence chain of prescribed substrings in $a$, giving the highest LCS score against string $b$.

*Description.* The algorithm runs in two phases.

**First phase.** As in Algorithm 11.

**Second phase.** The problem is now solved by dynamic programming as follows. Let $u_j(s)$ denote the highest LCS score across all precedence chains of prescribed substrings in prefix string $a \upharpoonright j$, taken against prefix string $b \upharpoonright s$. With each $j \in [0 : n]$, we associate the integer vector $u_j = u_j(*)$. We initialise $u_0$ as the zero vector. We then compute the vectors $u_j$, $j \in [1 : n]$, in the order of increasing $j$. Let $a_0 = a(i_0 : j)$, $a_1 = a(i_1 : j)$, $\ldots$, $a_t = a(i_t : j)$ be all the prescribed substrings of $a$ terminating at index $j$. We have

$$u_j = u_{j-1} \oplus (u_{i_0} \odot H_{a_0,b}) \oplus (u_{i_1} \odot H_{a_1,b}) \oplus \ldots \oplus (u_{i_t} \odot H_{a_t,b})$$

(9.2)

for all $j \in [1 : n]$.

The matrices $H_{a_0,b}$, $H_{a_1,b}$, $\ldots$, $H_{a_t,b}$ do not need to be computed explicitly. Instead, it is straightforward to compute each of the vector-matrix distance products in (9.2) by up to $\log m$ instances of implicit vector-matrix distance product, using the decomposition of each of $a_0$, $a_1$, $\ldots$, $a_t$ into up to $\log m$ canonical substrings, along with the corresponding seaweed matrices obtained in the first phase.

The solution score is now given by the value $u_m(n)$. The solution precedence chain of prescribed substrings can be obtained by tracing the dynamic programming sequence backwards from vector $u_m$ to vector $u_0$.

**Cost analysis.**
First phase. As in Algorithm 11, the total running time of this phase is $O(nm)$.

Second phase. For each of $k$ prescribed substrings of $a$, we execute up to $\log m$ instances of implicit matrix-vector distance multiplication. Every such instance runs in time $O(n)$ by Theorem 6. Therefore, the total running time of this phase is $O(kn \log m)$.

Total. The overall running time of the algorithm is dominated by the second phase, and is therefore $O(kn \log m)$. 

Similarly to Algorithm 12, a sharper analysis for $k \approx \left(\frac{m}{2}\right)$ leads to a smooth transition to the running time $O(m^2n)$ in the dense case $k = \left(\frac{m}{2}\right)$, which is asymptotically equal to the running time of the dense spliced algorithm of [57].
Chapter 10

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 nonlinear 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.
Chapter 11

Acknowledgement

This work was conceived in a discussion with Gad Landau in Haifa. The imaginative term “seaweeds” was coined by Yu. V. Matiyasevich during an author’s presentation in St. Petersburg. I thank Philip Bille, Peter Krusche, Gad Landau, Victor Levandovsky, Sergei Nechaev, Luis Russo, Andrei Sobolevski, Nikolai Vavilov, Oren Weimann, and Michal Ziv-Ukelson for fruitful discussions, and the anonymous referees of my previous papers for their comments, which helped to improve this work.
Bibliography

[1] A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(1):195–208, 1987.

[2] 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.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1976.

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

[5] 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.

[6] 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.

[7] C. E. R. Alves, E. N. Cáceres, and S. W. Song. An all-substrings common subsequence algorithm. *Discrete Applied Mathematics*, 156(7):1025–1035, 2008.

[8] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Computer and System Sciences*, 52(2):299–307, 1996.

[9] 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.

[10] A. Apostolico and C. Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2(1):315–336, 1987.
[11] V. L. Arlazarov, E. A. Dinic, M. A. Krourod, and I. A. Faradzev. On economical construction of the transitive closure of an oriented graph. Soviet Mathematical Doklady, 11:1209–1210, 1970.

[12] A. N. Arslan and ¨O. E¨egcio˘glu. Approximation algorithms for local alignment with length constraints. International Journal of Foundations of Computer Science, 13(5):751–767, 2002.

[13] A. N. Arslan, ¨O. E¨egcio˘glu, and P. A. Pevzner. A new approach to sequence comparison: Normalized sequence alignment. Bioinformatics, 17(4):327–337, 2001.

[14] G. Benson. Tandem cyclic alignment. Discrete Applied Mathematics, 146(2):124–133, 2005.

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

[16] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In Proceedings of the 7th SPIRE, pages 39–48, 2000.

[17] S. Bespamyatnikh and M. Segal. Enumerating longest increasing subsequences and patience sorting. Information Processing Letters, 76:7–11, 2000.

[18] P. Bille, R. Fagerberg, and I. L. Gørtz. Improved approximate string matching and regular expression matching on Ziv-Lempel compressed texts. ACM Transactions on Algorithms, 6(1):3:1–3:14, 2009.

[19] P. Bille and M. Farach-Colton. Fast and compact regular expression matching. Theoretical Computer Science, 409(3):486–496, 2008.

[20] P. Bille and I. L. Gørtz. Matching subsequences in trees. Journal of Discrete Algorithms, 7(3):306–314, 2009.

[21] P. Bille, G. M. Landau, and O. Weimann. Random access to grammar compressed strings. Technical Report 1001.1565, ArXiv.

[22] A. Björner and F. Brenti. Combinatorics of Coxeter Groups. Number 231 in Graduate Texts in Mathematics. Springer, 2005.

[23] N. Bourbaki. Groupes et algèbres de Lie. Chapitres 4,5 et 6. Hermann, 1968.

[24] A. S. Buch, A. Kresch, M. Shimozono, H. Tamvakis, and A. Yong. Stable Grothendieck polynomials and K-theoretic factor sequences. Mathematische Annalen, 340(2):359–382, 2008.
[25] H. Bunke and U. Bühler. Applications of approximate string matching to 2D shape recognition. *Pattern Recognition*, 26(12):1797–1812, 1993.

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

[27] S. Cabello and E. W. Chambers. Multiple-source shortest paths in a genus $g$ graph. In *Proceedings of the 18th ACM-SIAM SODA*, pages 89–97, 2007.

[28] 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.

[29] P. Cégielski, I. Guessarian, and Y. Matiyasevich. Multiple serial episodes matching. *Information Processing Letters*, 98(6):211–218, 2006.

[30] T. Chan and M. Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. In *Proceeding of ACM–SIAM SODA*, pages 161–173, 2010.

[31] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. In *Proceedings of the 39th ACM STOC*, pages 590–598, 2007.

[32] Maw-Shang Chang and Fu-Hsing Wang. Efficient algorithms for the maximum weight clique and maximum weight independent set problems on permutation graphs. *Information Processing Letters*, 43(6):293–295, 1992.

[33] Kun-Mao Chao and Louxin Zhang. *Sequence Comparison: Theory and Methods*, volume 7 of *Computational Biology Series*. Springer, 2009.

[34] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17:427–462, 1988.

[35] 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.

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

[37] R. Cole and R. Hariharan. Approximate string matching: A simpler faster algorithm. *SIAM Journal on Computing*, 31(6):1761–1782, 2002.
[38] G. Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Transactions on Algorithms*, 3(1), 2007.

[39] 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.

[40] M. Crochemore, G. M. Landau, and M. Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted score matrices. *SIAM Journal on Computing*, 32(6):1654–1673, 2003.

[41] M. Crochemore, B. Melichar, and Z. Troníček. Directed acyclic subsequence graph — overview. *Journal of Discrete Algorithms*, 1(3–4):255–280, 2003.

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

[43] G. Das, R. Fleischer, L. Gasieniec, D. Gunopulos, and J. Kärkkäinen. Episode matching. In *Proceedings of CPM*, volume 1264 of *Lecture Notes in Computer Science*, pages 12–27, 1997.

[44] E. Davydov and S. Batzoglou. A computational model for RNA multiple structural alignment. *Theoretical Computer Science*, 368(3):205–216, 2006.

[45] M. Demange, T. Ekim, and D. de Werra. A tutorial on the use of graph coloring for some problems in robotics. *European Journal of Operational Research*, 192(1):41–55, 2009.

[46] S. Deorowicz. An algorithm for solving the longest increasing circular subsequence problem. *Information Processing Letters*, 109(12):630–634, 2009.

[47] E. W. Dijkstra. Some beautiful arguments using mathematical induction. *Acta Informatica*, 13(1):1–8, 1980.

[48] B. Drake, S. Gerrish, and M. Skandera. Two new criteria for comparison in the Bruhat order. *The Electronic Journal of Combinatorics*, 11(1), 2004.

[49] A. Dress, J. H. Koolena, and V. Moulton. On line arrangements in the hyperbolic plane. *European Journal of Combinatorics*, 23(5):549–557, 2002.

[50] P. Erdős and G. Szekeres. A combinatorial problem in geometry. *Compositio Mathematica*, 2:463–470, 1935.
[51] S. Even and A. Itai. Queues, stacks and graphs. In Theory of Machines and Computations, pages 71–86. Academic Press, 1971.

[52] G. Fertin, D. Hermelin, R. Rizzi, and S. Vialette. Finding common structured patterns in linear graphs. Theoretical Computer Science, 411(26–28):2475–2486, 2010.

[53] V. A. Fischetti, G. M. Landau, P. H. Sellers, and J. P. Schmidt. Identifying periodic occurrences of a template with applications to protein structure. Information Processing Letters, 45(1):11–18, 1993.

[54] S. Fomin and C. Greene. Noncommutative Schur functions and their applications. Discrete Mathematics, 193(1–3):179–200, 1998. Reprinted in Discrete Mathematics, v. 306 (10–11), pages 1080–1096, 2006.

[55] M. L. Fredman. On computing the length of longest increasing subsequences. Discrete Mathematics, 11:29–35, 1975.

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

[57] 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.

[58] R. Giancarlo. Dynamic programming: Special cases. In A. Apostolico and Z. Galil, editors, Pattern Matching Algorithms, chapter 7, pages 201–236. Oxford University Press, 1997.

[59] A. J. Gibbs and G. A. McIntyre. The diagram, a method for comparing sequences. European Journal of Biochemistry, 16(1):1–11, 1970.

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

[61] M. Gondran and M. Minoux. Graphs, Dioids and Semirings. Springer, 2008.

[62] O. Gotoh. An improved algorithm for matching biological sequences. Journal of Molecular Biology, 162(3):705–708, 1982.

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

[64] D. Gusfield, K. Balasubramanian, and D. Naor. Parametric optimization of sequence alignment. Algorithmica, 12:312–326, 1994.
[65] D. Hermelin, G. M. Landau, S. Landau, and O. Weimann. A unified algorithm for accelerating edit-distance computation via text-compression. In Proceedings of the 26th STACS, pages 529–540, 2009.

[66] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. Communications of the ACM, 18(6):341–343, 1975.

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

[68] Yue Huang and Ling Zhang. Rapid and sensitive dot-matrix methods for genome analysis. Bioinformatics, 20(4):460–466, 2004.

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

[70] 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.

[71] B. N. Jackson and S. Aluru. Pairwise sequence alignment. In Handbook of Computational Molecular Biology, Chapman and Hall/CRC Computer and Information Science Series, chapter 1, pages 1–1 – 1–31. Chapman and Hall/CRC, 2006.

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

[73] C. R. Johnson and S. Nasserasr. TP2 = Bruhat. Discrete Mathematics. To appear.

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

[75] J. Kärkkäinen, G. Navarro, and E. Ukkonen. Approximate string matching on Ziv–Lempel compressed text. Journal of Discrete Algorithms, 1:313–338, 2003.

[76] A. Karzanov. Combinatorial methods to solve cut-determined multflow problems. In Combinatorial Methods for Flow Problems, volume 3, pages 6–69. VNIISI, 1979. In Russian.

[77] C. Kassel and V. Turaev. Braid Groups, volume 247 of Graduate Texts in Mathematics. Springer, 2008.
[78] 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.

[79] T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Collage system: A unifying framework for compressed pattern matching. *Theoretical Computer Science*, 298(1):253–272, 2003.

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

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

[82] D. E. Knuth. Permutations, matrices, and generalized Young tableaux. *Pacific Journal of Mathematics*, 34(3):709–727, 1970.

[83] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison Wesley, 1998.

[84] 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.

[85] J. Krumsiek, R. Arnold, and T. Rattei. Gepard: A rapid and sensitive tool for creating dotplots on genome scale. *Bioinformatics*, 23(8):1026–1028, 2007.

[86] P. Krusche and A. Tiskin. String comparison by transposition networks. In *London Algorithmics 2008: Theory and Practice*, volume 11 of *Texts in Algorithmics*. College Publications, 2009.

[87] G. Landau. Can DIST tables be merged in linear time? An open problem. In *Proceedings of the Prague Stringology Conference*, page 1. Czech Technical University in Prague, 2006.

[88] G. M. Landau, E. Myers, and M. Ziv-Ukelson. Two algorithms for LCS consecutive suffix alignment. *Journal of Computer and System Sciences*, 73(7):1095–1117, 2007.

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

[90] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989.

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

112
[92] V. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. Problems of Information Transmission, 1:8–17, 1965.

[93] Y. Lifshits. Processing compressed texts: A tractability border. In Proceedings of CPM, volume 4580 of Lecture Notes in Computer Science, pages 228–240, 2007.

[94] 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.

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

[96] 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.

[97] A. Marzal and S. Barrachina. Speeding up the computation of the edit distance for cyclic strings. In Proceedings of the 15th ICPR, part 2, pages 891–894, 2000.

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

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

[100] 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.

[101] E. W. Myers and W. Miller. Approximate matching of regular expressions. Bulletin of Mathematical Biology, 51(1):5–37, 1989.

[102] G. Myers. Approximately matching context-free languages. Information Processing Letters, 54:85–92, 1995.

[103] G. Navarro. A guided tour to approximate string matching. ACM Computing Surveys, 33(1):31–88, 2001.

[104] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology, 48(3):443–453, 1970.
[105] M. S. Paterson and V. Dančík. Longest common subsequences. In Proceedings of MFCS, volume 841 of Lecture Notes In Computer Science, pages 127–142, 1994.

[106] E. Picot, P. Krusche, A. Tiskin, I. Carré, and S. Ott. Evolutionary analysis of regulatory sequences (EARS) in plants. The Plant Journal. Submitted.

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

[108] C. Putonti, Yi Luo, C. Katili, S. Chumakov, G. E. Fox, D. Graur, and Y. Fofanov. A computational tool for the genomic identification of regions of unusual compositional properties and its utilization in the detection of horizontally transferred sequences. Molecular Biology and Evolution, 23(10):1863–1868, 2006.

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

[110] S. V. Rice, H. Bunke, and T. A. Hart. Classes of cost functions for string edit distance. Algorithmica, 18:271–280, 1997.

[111] R. W. Richardson and T. A. Springer. The Bruhat order on symmetric varieties. Geometriae Dedicata, 35(1–3):389–436, 1990.

[112] G. de B. Robinson. On the representations of the symmetric group. American Journal of Mathematics, 60(3):745–760, 1938.

[113] G. Rote. Path problems in graphs. Computing Supplementum, 7:155–189, 1990.

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

[115] W. Rytter. Algorithms on compressed strings and arrays. In Proceedings of SOFSEM, volume 1725 of Lecture Notes in Computer Science, pages 48–65, 1999.

[116] Y. Sakai. An almost quadratic time algorithm for sparse spliced alignment. Theory of Computing Systems. To appear.

[117] 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.
[118] P. H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms*, 1(4):359–373, 1980.

[119] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[120] 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):GC1–GC10, 1995.

[121] A. Tiskin. All semi-local longest common subsequences in subquadratic time. In *Proceedings of CSR*, volume 3967 of *Lecture Notes in Computer Science*, pages 352–363, 2006.

[122] 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.

[123] A. Tiskin. Semi-local longest common subsequences in subquadratic time. *Journal of Discrete Algorithms*, 6(4):570–581, 2008.

[124] A. Tiskin. Semi-local string comparison: Algorithmic techniques and applications. *Mathematics in Computer Science*, 1(4):571–603, 2008.

[125] A. Tiskin. Faster subsequence recognition in compressed strings. *Journal of Mathematical Sciences*, 158(5):759–769, 2009.

[126] A. Tiskin. Periodic string comparison. In *Proceedings of CPM*, volume 5577 of *Lecture Notes in Computer Science*, pages 193–206, 2009.

[127] A. Tiskin. Fast distance multiplication of unit-Monge matrices. In *Proceedings of ACM–SIAM SODA*, pages 1287–1296, 2010.

[128] S. Tsaranov. Representation and classification of Coxeter monoids. *European Journal of Combinatorics*, 11(2):189–204, 1990.

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

[130] B. W. Watson and G. Zwaan. A taxonomy of sublinear multiple keyword pattern matching algorithms. *Science of Computer Programming*, 27(2):85–118, 1996.

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

[132] S. Wu, U. Manber, and G. Myers. A subquadratic algorithm for approximate limited expression matching. *Algorithmica*, 15:50–67, 1996.
[133] G. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.

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

(I | J): Cartesian product, 7
(min, +)-semiring, 13
*: implicit range, 7
AR: matrix rotation, 7
AT: matrix transpose, 7
A: density matrix, 8
DΣ: distribution matrix, 8
Ga,b: alignment dag, 33
Ha,b: prefix-suffix score matrix, 39
Ha,b: string-substring score matrix, 39
Ha,b: substring-string score matrix, 39
Ha,b: suffix-prefix score matrix, 39
O⊙: ⊙-annihilator, 14
P′a′,a′′,b: seaweed cross-matrix, 40
P′a,b: substring-string seaweed matrix, 39
P′a,b: suffix-prefix seaweed matrix, 39
P′a,b: semi-local seaweed matrix, 37
Id: identity matrix, 9
Idh: offset identity matrix, 9
Id⊙: ⊙-identity, 14
Σ(a): character set, 69
(i : j): odd half-integer interval, 6
(i0 : i1 | j0 : j1): interval Cartesian product, 7
a: string reverse, 69
□: implicit distance multiplication, 16
[i : j]: integer interval, 6
[i0 : i1 | j0 : j1]: interval Cartesian product, 7
::: induced subrange, 9
id: identity permutation string, 69
|, |: left/right drop, 31
≪, ≻: dominance orders, 6
|, |: left/right take, 31
⊙: distance multiplication (+), 13
⊙: matrix distance multiplication, 13
⊕: distance addition (min), 13
□: matrix disjoint union, 7
τh(A): threshold subset, 61
wϕ: gap weight, 57
ϕ: wildcard character, 31
w: match weight, 57
w: mismatch weight, 57
a|s: filtered subsequence, 69
$: non-alphabet character, 31
algorithm
dynamic programming, 46, 61
micro-block, 49
wraparound, 68
seaweed, 46
micro-block, 50
periodic, 65
Smith–Waterman–Gotoh, 94
alignment dag, 33
alignment score
LCS, 31, 57
weighted, 57
for DNA, 57
indel, 60
indelsub, 60
normalised, 58
rational, 58
antichain, 6
Bruhat order, 27
chain, 6
circle graph, 74
  interval model, 74
distance (tropical) algebra, 13
distance multiplication
  matrix-matrix, 13
  implicit, 16
  matrix-vector, 13
  implicit, 16
dominance, 6
  counting, 10
dot plot
  alignment-filtered, 95
  Hamming-filtered, 94
edit distance, 59
  episode, 60
  indel (LCS), 60
  indelsub (Levenshtein), 60
function
  unit-monotone, 6
interval notation, 6
linear graph, 78
matrix
  anti-Monge, 9
  density, 8
  distribution, 8
general
  ⊙-annihilator, 14
  ⊙-identity, 14
  ⊙-monoid, 13
highest-score
  composition, 41
  implicit, 7
  batch query, 12
  element query, 10
incremental query, 11
Monge, 9
  ⊙-identity, 16
  ⊙-monoid, 14
periodic, 64
  period submatrix, 65
permutation, 9
  identity, 9
  offset identity, 9
score
  prefix-suffix, 39
  semi-local, 35
  string-substring, 39
  substring-string, 39
  suffix-prefix, 39
seaweed
  prefix-suffix, 39
  semi-local, 37
  string-substring, 39
  substring-string, 39
  suffix-prefix, 39
seaweed cross-, 40
simple, 8
subpermutation, 9
subunit-anti-Monge, 10
subunit-Monge, 10
totally monotone, 8
unit-anti-Monge, 10
unit-Monge, 10
  ⊙-annihilator, 16
  ⊙-identity, 16
  ⊙-monoid, 14
problem
  approximate matching, 60
  complete, 60
  output filtering, 61
  threshold, 61, 88
  block-incremental LCS, 53
circle graph maximum clique, 75
common-substring LCS, 54
  semi-local, 54
cyclic LCS, 55
  between permutations, 71
episode matching, 61
fully-incremental LCS, 52
incremental LCS, 52
linear graph maximum common
pattern, 78
longest \( k \)-increasing subsequence,
73
longest \( k \)-modal subsequence, 73
longest \( Y \)-avoiding subsequence,
72
longest common subsequence (LCS),
31
longest increasing subsequence (LIS),
69
longest repeating subsequence, 55
periodic string-substring LCS, 64
quasi-local LCS, 98
semi-local LCS, 32
three-way, 33, 82
subsequence matching, 32
subsequence recognition
global, 32
local, 32, 61, 84
tandem alignment, 67
tandem LCS, 67
window-substring LCS, 95
window-window LCS, 95

score
Hamming, 94
seaweed, 17
braid, 17
monoid, 18
spliced alignment, 101
sparse, 101
straight-line program (SLP), 80
statement, 80
string
grammar-compressed (GC), 80
periodic, 64
permutation, 69
identity, 69
prefix, 31
reverse, 69